Feature Summary

- 1.0 GFLOPS - 1.5 GOPS at 100 MHz
- AHB Master Port, integrated DMA Engine and AHB Slave Port
- VLIW Architecture with five Independent Execution Units
- Up to 10 Arithmetic Operations per Cycle (4 Multiply, 2 Add/Subtract, 1 Add, 1 Subtract 40-bit Floating Point and 32-bit Integer) performing a Single Cycle FFT Butterfly
- Native Support for Complex Arithmetic and Vectorial SIMD Operations: One Complex Multiply with Dual Add/Sub per Clock Cycle or Two Real Multiply and Two Add/Sub or Simple Scalar Operations
- 32-bit Integer and IEEE® 40-bit Extended Precision Floating Point Numeric Format
- 16-port Data Register File: 256 Registers organized in two 128-register Banks
- 5-issue predicated VLIW Architecture with Orthogonal ISA, Code Compression and Hardware Support for Code Efficient Software Pipeline Loops
- 4 Data Accesses per Cycle supported
- 2 Independent Address Generation Units Operating on a 16-register Address Register File Supporting DSP features: Programmable Stride and Circular Buffers
- 1.7 Mbits of On-chip SRAM: 2 x 8K x 40-bit Data Memory Locations, 8 K x 128-bit Program Memory Locations, Equivalent to ~50K DSP Assembler Instructions thanks to Code Compression and SW Pipelining
- Program Management Unit with HW Page-replacing Algorithm
- DMA access to AHB SOC Peripherals and Memories
- Hardware support for Debug
- Support for Multi-core Integration: Mutex, Cross-triggering

mAgicV DSP

Architecture Document

7011A–DSP–12/08

1. About this manual

This manual describes mAgicV DSP high-level architecture and the programming user interface. mAgicV VLIW assembly language is described in the **Assembly Reference Manual**.

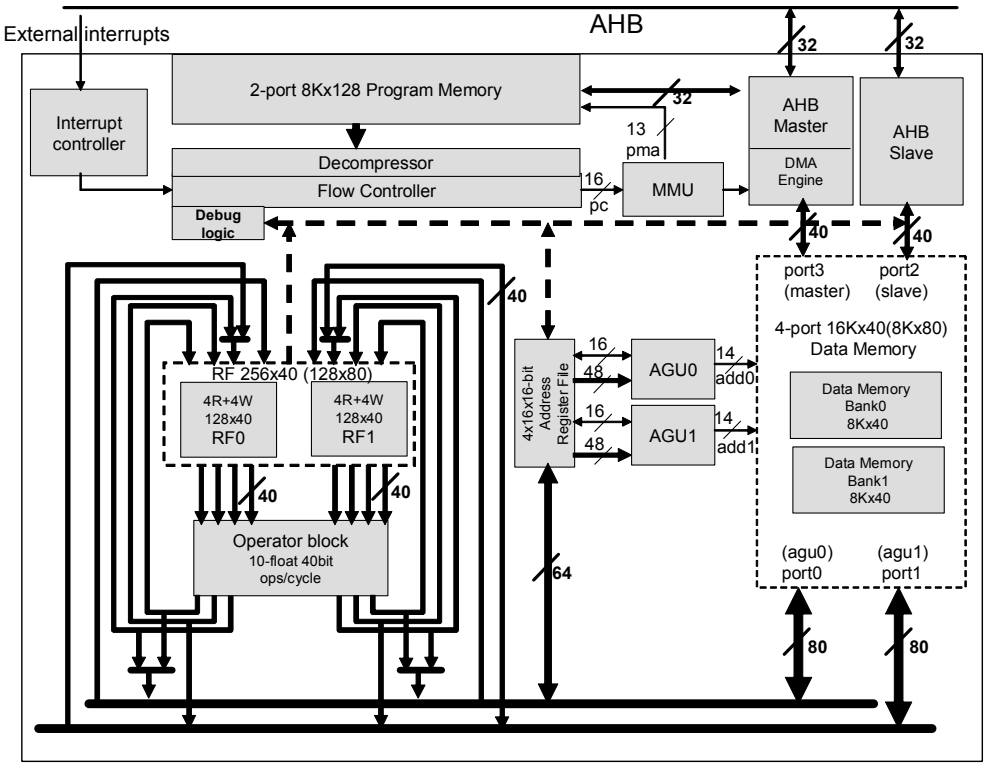
2. References

- Parallel Assembly Syntax User Guide - doc7013.pdf
- mAgicV C Compiler User Guide - doc7016.pdf

3. mAgicV VLIW DSP Architecture

mAgicV is a high performance Very Long Instruction Word (VLIW) DSP delivering 1.0 Giga floating-point operations per second (GFLOPS) and 1.5 GOPS at a clock rate of 100 MHz. It is equipped with an AHB master port and an AHB slave port for system-on-chip integration. It has 256x40-bit data registers, 16x64-bit multi-field address registers to support DSP oriented addressing modes like circular and stride accesses, 10 arithmetic operating units, two independent AGUs (Address Generation Unit) and a DMA engine. To sustain the internal parallelism, the data bandwidth through the Register File is 80 bytes/cycle. The architecture is optimized to work in the complex domain. When activating all the computing units, mAgicV can produce a complete FFT butterfly per cycle. It also supports native 2D vectorial arithmetic operations. mAgicV operates on IEEE 754 40-bit extended precision floating-point and 32-bit integer numeric format for numerical computations.

Figure 3-1. mAgicV Block Diagram



The Harvard memory architecture is composed of an on-chip 2x8Kx40-bit data memory and an on-chip 8Kx128-bit program memory. Efficient usage of the program memory is achieved through a mechanism of program compression performed by the software tool chain and supported by a hardware decompression engine. A program memory management unit supports a virtual program space of 64Kx128-bit locations. Interrupts are vectorized to minimize the interrupt service latency.

3.1 VLIW overview

VLIW processors execute parallel operations based on a fixed schedule determined at compile-time. The processor does not need any hardware support for scheduling because the operations execution schedule is handled by the compiler (including the operations to be executed simultaneously). As a result, VLIW CPUs offer significant computational power with less hardware complexity (but greater compiler complexity) than most superscalar CPUs.

The rows in mAgicV program memory are 128 bit wide. When the “default” decoding scheme is applied, the program word (composed of 120 bits) drives five execution units through five operation fields named issues. Eight additional bits drive the program decompression engine. mAgicV issues are named: FLOW, AGU0, MUL, AGU1, ADD.

Table 3-1. Conceptual Representation of Issues in the Default VLIW Decoding Scheme

FLOW	AGU0	MUL	AGU1	ADD

Two issues are associated to the pair of independent AGUs. The ADD and the MUL issues drive respectively the add/subtract unit and the multiplier operators unit. The FLOW issue manages the program flow unit. Each issue is predicated by a predication register for conditional execution without pipeline breaking penalties.

3.1.1 Description of the Issues

Every issue describes the operation to be performed by the associated execution unit. Each issue is composed of one or more VLIW fields. Operations on the FLOW issue have the maximum priority and can inhibit the execution of other issues or change their default format.

Table 3-2 shows the main fields that constitute a VLIW program word. Issues can either be composed of these fields (default issues) or of completely different fields. Default issues are orthogonal and grant maximum parallelism (five operations per cycle). The following paragraphs show the default format of the five issues, further details about all possible issues and their formats can be found in the **Assembly Reference Manual**.

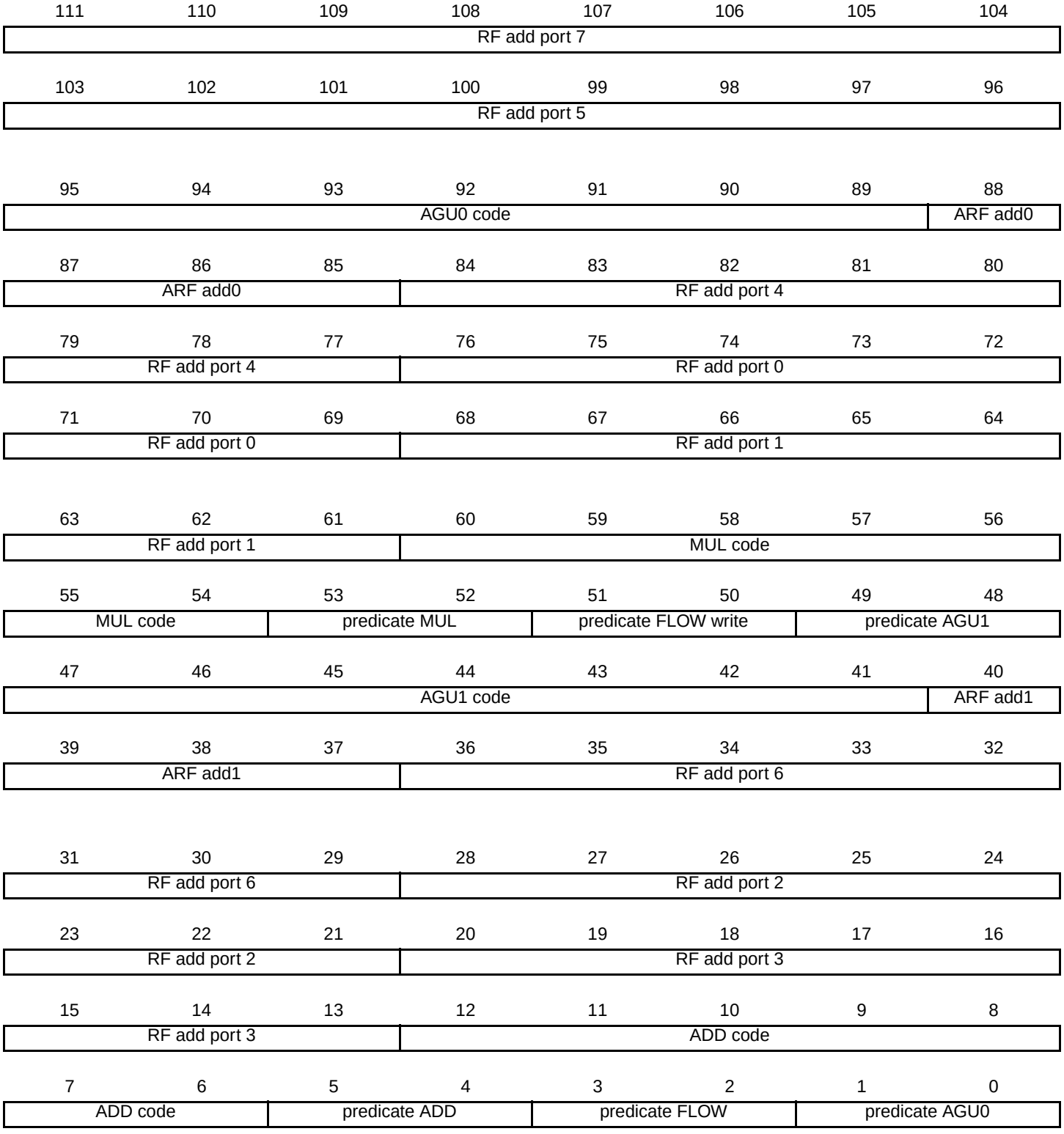
3.1.2 MGCVLIW Register

The decompressed VLIW word ready for the decoding stage is visible to the external AHB masters in the MGCVLIW register. The default structure of the VLIW word after decoding is shown in Table 3-2.

Table 3-2. MGCVLIW Register. Default Decoding Scheme.

127	126	125	124	123	122	121	120
used in compression							

119	118	117	116	115	114	113	112
spare	FLOW code						RF add port 7


111 110 109 108 107 106 105 104
RF add port 7
103 102 101 100 99 98 97 96
RF add port 5
95 94 93 92 91 90 89 88
AGU0 code
ARF add0
87 86 85 84 83 82 81 80
ARF add0
RF add port 4
79 78 77 76 75 74 73 72
RF add port 4
RF add port 0
71 70 69 68 67 66 65 64
RF add port 0
RF add port 1
63 62 61 60 59 58 57 56
RF add port 1
MUL code
55 54 53 52 51 50 49 48
MUL code
predicate MUL
predicate FLOW write
predicate AGU1
47 46 45 44 43 42 41 40
AGU1 code
ARF add1
39 38 37 36 35 34 33 32
ARF add1
RF add port 6
31 30 29 28 27 26 25 24
RF add port 6
RF add port 2
23 22 21 20 19 18 17 16
RF add port 2
RF add port 3
15 14 13 12 11 10 9 8
RF add port 3
ADD code
7 6 5 4 3 2 1 0
ADD code
predicate ADD
predicate FLOW
predicate AGU0

3.2 Program Memory

The Program Memory system contains 8K*128-bit on-chip memory locations supporting up to 2 accesses/cycle. One-access/cycle is reserved to the core to fetch the program, while the other access is used either by the internal AHB master access (i.e: DMA) or by the internal AHB slave access (e.g.: debug or accesses executed by an external AHB master).

The read latency during program fetch is 1-cycle. While write and read latencies through AHB are shown in Table 3-26 on page 39.

An efficient usage of the Program Memory is achieved through a program memory decompressor engine that is able to decompress, in a single clock cycle, words that are stored using a compression format. Thus the total latency per program fetch, including the compression, is 2 cycles.

3.2.1 Decompressor

The mAgicV VLIW architecture is natively designed for optimal program density. Moreover, the program compression scheme allows an average compression of 165%. Therefore, about eight issues are stored for each 128-bit program memory locations. A higher Program Memory density is achieved thanks to the combined effect of Program Compression and HW support for software pipelining (see Section 3.9.7 ”Software Pipeline HW Support” on page 55). Many applications can be implemented on mAgicV using only the internal program memory. In fact, the 8Kx128bit program memory provides, through code compression, about 50K DSP assembler instructions stored on-chip (typically).

This device is able to decompress at each cycle a 128-bit Compressed Program Word (CPW) coming from the Program Memory into one or more 120-bit mAgicV VLIW words, by using a patented compression/decompression scheme.

3.2.1.1 Compressed Program Word

The CPW is composed of an 8-bit Super-Header (SH), an optional 16-bit Field-Header (FH), 7 optional 16-bit fields and an optional 8-bit field.

The Super-Header field is used to drive the decompression of the CPW. Table 3-3 shows an example of 4 VLIWs compressed into a single 128-bit CPW. The SH field must always exist (there is a SH for each compressed VLIW), all others fields are optional. The FH drives eight decompressing units, named dyprodes, with 16 control bits (2-bit for each dyprode).

Table 3-3. Compressed Program Word

127						24	23 8	0
FIELDS3	FH3	SH3	FH2	SH2	SH1	FIELDS0	FH0	SH0

The SH format is described in Table 3-4.

Table 3-4. SH Format

7 6 5 4	3 2	1	0
word length	SH code	parity	fetch

- **fetch**

FETCH=’1’, fetches another CPW, PMA is incremented.

FETCH=’0’, doesn’t fetch, PMA is not modified.

- **parity**

It's the parity bit of the decompressed VLIW word.

NOTE: When a program is compressed many parity bits can exist in the same Compressed Program Word.

- **code**

2-bit of global decompressing code.

Table 3-5. SH Codes

SH Code	Mnemonic	Description
0x0	NOP	a NOP is generated, no FH needed
0x1	SHIFTALL	a realign request (i.e. branch). The next CPW word will start with a SH
0x2	CTRL	the decompressing units are driven by the Field Header
0x3	STORE	All the decompressing units are requested to store the contents of the current CPW, no FH needed, see Table 3-6 on page 6 and Table 3-7 on page 7

- **word length**

The length of the current compressed VLIW is a multiple of 8 bits.

NOTE: A compressed VLIW can be stored in two successive CPWs.

3.2.1.2 *Dyprodes*

The dyprodes are the decompressing units. There are two types of dyprodes:

- derivative dyprodes: which decompress RF addresses
- value dyprodes: which decompress operation microcodes and other bits

Both dyprode types are driven by a 2-bit control word and operate either on a 16-bit input data producing a 16-bit output or on an 8-bit input data producing an 8-bit output. Actually each 120-bit decompressed program word is produced by 7 dyprodes operating on 16-bit plus one dyprode operating on 8-bit. Accordingly, the Field Header is the control word (8x2 bits) of eight dyprodes. Table 3-8 on page 7 shows the mapping between the decompressed VLIW and the dyprodes outputs.

Each dyprode has two 16-bit registers: SAMEREG and SWAPREG that hold respectively the current value (or current derivative) and the last value (last derivative). The derivative dyprode has a third register used as an accumulator that holds the sum between one of the previous registers and the old accumulated sum. Table 3-6 on page 6 shows the operations performed by the dyprodes.

Table 3-6. Value Dyprode Control Codes

Code	Mnemonic	Operation	Description
0x0	NOP	output=constant	a constant output is generated

Code	Mnemonic	Operation	Description
0x1	SAME	output=samereg	output is not changed
0x2	SWAP	output=swapreg swapreg=samereg	output is a previously stored input, the current output is stored
0x3	STORE	output=input samereg=input swapreg=samereg	input is stored and put in output

Table 3-7. Derivative Dyprode Control Codes

Code	Mnemonic	Operation	Description
0x0	NOP	output=constant acc=constant	a constant output is generated, accumulator reset
0x1	SAME	output= acc + samereg	output is the accumulator value plus the stored derivative
0x2	SWAP	output=acc + swapreg swapreg=samereg	output is the accumulator value plus the last stored derivative
0x3	STORE	output=input samereg=input swapreg=samereg	input is stored and put in output

Table 3-8. VLIW to Dyprode Mapping

VLIW	Dyprode Output
vliw[3:0]	Value dyprode #4 [3:0]
vliw[12:4]	Value dyprode #5 [8:0]
vliw[28:13]	Derivative dyprode #0 [15:0]
vliw[36:29]	Derivative dyprode #1 [7:0]
vliw[52:37]	Value dyprode #6 [15:0]
vliw[60:53]	Value dyprode #7 [7:0]
vliw[76:61]	Derivative dyprode #2 [15:0]
vliw[84:77]	Derivative dyprode #1 [15:8]
vliw[95:85]	Value dyprode #4 [14:4]
vliw[111:96]	Derivative dyprode #3 [15:0]
vliw[112]	Value dyprode #4 [15]
vliw[119:113]	Value dyprode #5 [15:9]

3.3 Register File

In order to provide optimal data bandwidth and to give the best support to the RISC-like programming model, mAgicV arithmetic computations are supported by a 16-port 256x40-bit entries Register File (RF). The registers are numbered from RF0 to RF255 and they can be accessed either individually for scalar operations or in pairs, aligned to even addresses, for operations in the complex or vectorial domain.

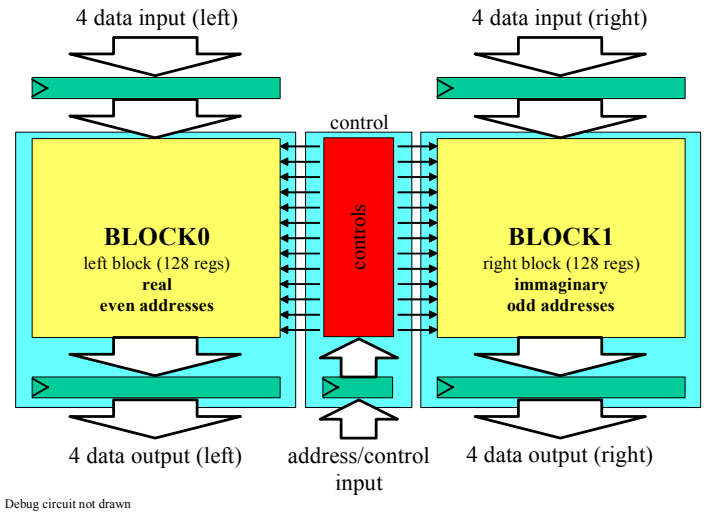
3.3.1 Description

The RF is a multi-port memory containing 256 registers (RF0-RF255). They are arranged in two blocks of 128x40-bit registers each. BLOCK0 holds even registers (real part of a complex number) while the BLOCK1 holds odd registers (imaginary part of a complex number). At each cycle eight addresses are delivered to both register arrays in order to move up to eight in/out data pairs. Both the odd and the even side of the register file are 9-ported (4-read ports and 4-write ports for computing/move operations + 1 port for independent debug access), making a total of 16 I/O ports available for the data move to and from the operators block and the memory, plus ports for debug accesses.

The software tool chain must avoid concurrent double write with different data on the same RF address.

NOTE: In case of a simultaneous (i.e same cycle) write and read at the same location there is a bypass logic that allows to read the currently written data. This can be used for reducing the loop length because the read after the write data latency takes one cycle instead of two cycles.

Figure 3-2. Register File



3.4 Operators Block

The Operators Block performs arithmetical operations. It works on IEEE 754 extended precision 40-bit floating-point and 32-bit signed integers data. The 16-bit unsigned and signed integers are managed by the AGUs (see Section 3.6.3 ”Arithmetic” on page 36). The operators are arranged in order to support:

- arithmetic on complex domain (throughput of one complex multiply, add or multiply and add per cycle)
- fast FFT (throughput of one complete butterfly computation per cycle)
- vectorial arithmetic acting on operands constituted of data pairs. The operators block is able to launch a vectorial multiply plus a vectorial add at every cycle
- scalar arithmetic acting on scalar data. The operator block is able to launch a scalar multiplication and a scalar addition at every cycle

The peak performance of mAgicV is achieved during single cycle FFT butterfly execution, when mAgicV delivers 10 floating-point or 32-bit signed integer operations per clock cycle.

The operands manipulated by the operators block are specified by the RF addresses. The RF addresses for scalar domain operations can be either odd or even. Vectorial and Complex operand pairs need even RF addresses.

3.4.1 Description

The Operators Block is composed of 4 integer/floating point multipliers, 4 integer/floating point adders, 2 shift/logic units and 2 seed generators (see Figure 3-3 on page 10).

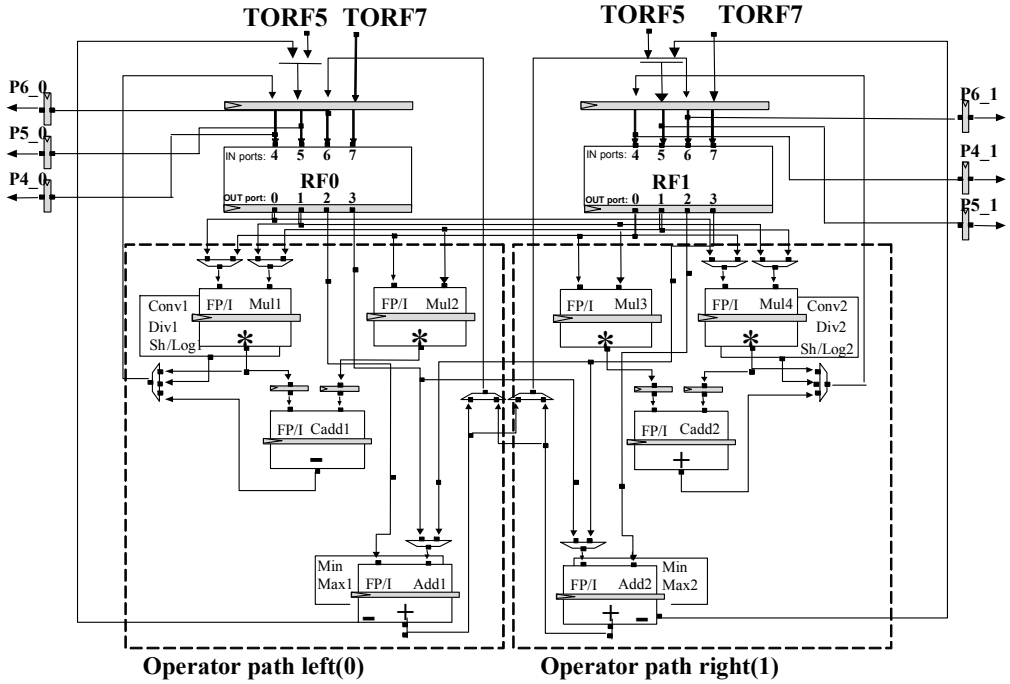
The complex product is hardwired in the operators block. This is obtained by using 4 multipliers and 2 adders with integer and floating-point capabilities.

Vector operations are performed using 2 multipliers, 2 adders, 2 logic units and 2 seed generators with floating point and integer capabilities, on the left and right path simultaneously.

Scalar operations (floating or integer) operate on one path only (left or right depending on the parity of the register address involved).

The inputs of the 4 multipliers are RF Port 0 and RF Port 1, while the inputs of the 2 independent adders are RF Port 2 and RF Port 3. In output, the adder returns its result in RF Port 6 and the multiplier returns it in RF Port 4. RF Port 5 collects the SUB output of the adder unit in simultaneous ADD/SUB instructions only.

Figure 3-3. Operators Block



Arithmetic operations performed by the operators block can be classified either as “ADD operations” or “MUL operations”. ADD and MUL operations can be performed in parallel, because they are driven by two orthogonal issues in the VLIW program word.

The ADD operations involve solely adders/subtractors. All other operations are classified as MUL operations (i.e a complex multiplication is classified in the group of MUL operations even if adders are involved in the calculation).

3.4.2 ADD/SUB Issue

The ADD/SUB issue drives a pair of adder/subtractor and minmax operators (ADD1, ADD2, MIN/MAX1, MIN/MAX2 in Figure 3-3 above). It takes two scalar, either vectorial or complex, input operands from the Register File. Usually, a scalar result, either vectorial or complex, is returned to the Register File. Only the ADDSUB operations produce two results. Compare operations produce no result at all (only condition flags are updated).

There are two possible formats for an ADD/SUB issue:

Table 3-9. ADD/SUB Issue Format 1

vliw[5:4]	vliw[12:6]	vliw[36:29]	vliw[20:13]	vliw[28:21]
ADD predication	ADD code	RF add 6 (D)	RF add3 (S1)	RF add2 (S2)

- **RF add 2: Register File address port 2**

It specifies one of the 256 RF registers as input operand0

- **RF add 3: Register File address port 3**

It specifies one of the 256 RF registers as input operand1

- **RF add 6: Register File address port 6**

It specifies one of the 256 RF registers as output operand

- **ADD opcode: ADD microcode**

It specifies the ADD operation to be performed

- **ADD predication**

It specifies one of the four predication registers, if the condition of the predication register is false the issue will not be executed

The second format is reserved to operations that generate two results like: minmax or addsub. In this case one more output address must be specified:

Table 3-10. ADD/SUB Issue Format 2

vliw[5:4]	vliw[12:6]	vliw[102:96]	vliw[36:29]	vliw[20:13]	vliw[28:21]
ADD predication	ADD opcode	RF add5 (D2)	RF add6 (D1)	RF add3 (S1)	RF add2 (S2)

- **RF add 5: Register File address port 5**

It specifies one of the 256 RF registers as second output operand

Table 3-11. ADD/SUB Operations

ADD/SUB Opcode	Mnemonic	Description
0x0	CFADD D S1 S2	Complex Floating Point Addition $(D_0,\ D_1) = S1_c + S2_c = ((S1_r + S2_r),\ (S1_i + S2_i))$
0x1	CFADD D S1 -S2	Complex Floating Point Subtraction $(D_0,\ D_1) = S1_c - S2_c = ((S1_r - S2_r),\ (S1_i - S2_i))$
0x2	CFADD D -S1 -S2	Complex Floating Point Addition $(D_0,\ D_1) = -S1_c - S2_c = ((-S1_r - S2_r),\ (-S1_i - S2_i))$
0x3	CFADD D -S1 S2	Complex Floating Point Addition $(D_0,\ D_1) = -S1_c + S2_c = ((-S1_r + S2_r),\ (-S1_i + S2_i))$
0x4	CFJADD D S1 S2	Complex Conjugate Floating Point Addition $(D_0,\ D_1) = S1_c + \overline{S2_c} = ((S1_r + S2_r),\ (S1_i - S2_i))$
0x5	CFJADD D S1 -S2	Complex Conjugate Floating Point Subtraction $(D_0,\ D_1) = S1_c - \overline{S2_c} = ((S1_r - S2_r),\ (S1_i + S2_i))$

ADD/SUB Opcode	Mnemonic	Description
0x6	CFJADD D -S1 -S2	Complex Conjugate Floating Point Addition $(D_0,\ D_1) = -S1_c - \overline{S2_c} = ((-S1_r - S2_r),\ (-S1_i + S2_i))$
0x7	CFJADD D -S1 S2	Complex Conjugate Floating Point Addition $(D_0,\ D_1) = -S1_c + \overline{S2_c} = ((-S1_r + S2_r),\ (-S1_i - S2_i))$
0x8	CJJADD D S1 S2	Complex Double-Conjugate Floating Point Addition $(D_0,\ D_1) = \overline{S1_1} + \overline{S2_2} = ((S1_r + S2_r),\ (-S1_i - S2_i))$
0x9	CJJADD D S1 -S2	Complex Double-Conjugate Floating Point Subtraction $(D_0,\ D_1) = \overline{S1_1} - \overline{S2_2} = ((S1_r - S2_r),\ (-S1_i + S2_i))$
0xA	CJJADD D -S1 -S2	Complex Double-Conjugate Floating Point Addition $(D_0,\ D_1) = \overline{-S1_1} - \overline{S2_2} = ((-S1_r - S2_r),\ (S1_i + S2_i))$
0xB	CJJADD D -S1 S2	Complex Double-Conjugate Floating Point Addition $(D_0,\ D_1) = \overline{-S1_1} + \overline{S2_2} = ((-S1_r + S2_r),\ (S1_i - S2_i))$
0xC	CFRADD D S1 S2	Complex with Real Floating Point Addition $(D_0,\ D_1) = S1_c + S2_r = ((S1_r + S2_r),\ S1_i)$
0xD	CFRADD D S1 -S2	Complex with Real Floating Point Subtraction $(D_0,\ D_1) = S1_c - S2_r = ((S1_r - S2_r),\ S1_i)$
0xE	CFRADD D -S1 -S2	Complex with Real Floating Point Addition $(D_0,\ D_1) = -S1_c - S2_r = ((-S1_r - S2_r),\ -S1_i)$
0xF	CFRADD D -S1 S2	Complex with Real Floating Point Addition $(D_0,\ D_1) = -S1_c + S2_r = ((-S1_r + S2_r),\ -S1_i)$
0x10	CIADD D S1 S2	Integer Complex Integer Addition $(D_0,\ D_1) = S1_c + S2_c = ((S1_r + S2_r),\ (S1_i + S2_i))$
0x11	CIADD D S1 -S2	Complex Integer Subtraction $(D_0,\ D_1) = S1_c - S2_c = ((S1_r - S2_r),\ (S1_i - S2_i))$
0x12	CIADD D -S1 -S2	Complex Integer Addition $(D_0,\ D_1) = -S1_c - S2_c = ((-S1_r - S2_r),\ (-S1_i - S2_i))$
0x13	CIADD D -S1 S2	Complex Integer Addition $(D_0,\ D_1) = -S1_c + S2_c = ((-S1_r + S2_r),\ (-S1_i + S2_i))$
0x14	CIJADD D S1 S2	Complex Conjugate Integer Addition $(D_0,\ D_1) = S1_c + \overline{S2_c} = ((S1_r + S2_r),\ (S1_i - S2_i))$
0x15	CIJADD D S1 -S2	Complex Conjugate Integer Subtraction $(D_0,\ D_1) = S1_c - \overline{S2_c} = ((S1_r - S2_r),\ (S1_i + S2_i))$
0x16	CIJADD D -S1 -S2	Complex Conjugate Integer Addition $(D_0,\ D_1) = -S1_c - \overline{S2_c} = ((-S1_r - S2_r),\ (-S1_i + S2_i))$

ADD/SUB Opcode	Mnemonic	Description
0x17	CIJADD D -S1 S2	Complex Conjugate Integer Addition $(D_0,\ D_1) = -S1_c + \overline{S2_c} = ((-S1_r + S2_r),\ (-S1_i - S2_i))$
0x18	CIJJADD D S1 S2	Complex Double-Conjugate Integer Addition $(D_0,\ D_1) = \overline{S1_1} + \overline{S2_2} = ((S1_r + S2_r),\ (-S1_i - S2_i))$
0x19	CJIJADD D S1 -S2	Complex Double-Conjugate Integer Subtraction $(D_0,\ D_1) = \overline{S1_1} - \overline{S2_2} = ((S1_r - S2_r),\ (-S1_i + S2_i))$
0x1A	CIJJADD D -S1 -S2	Complex Double-Conjugate Integer Addition $(D_0,\ D_1) = \overline{-S1_1} - \overline{S2_2} = ((-S1_r - S2_r),\ (S1_i + S2_i))$
0x1B	CIJJADD D -S1 S2	Complex Double-Conjugate Integer Addition $(D_0,\ D_1) = \overline{-S1_1} + \overline{S2_2} = ((-S1_r + S2_r),\ (S1_i - S2_i))$
0x1C	CIRADD D S1 S2	Complex with Real Integer Addition $(D_0,\ D_1) = S1_c + S2_r = ((S1_r + S2_r),\ S1_i)$
0x1D	CIRADD D S1 -S2	Complex with Real Integer Subtraction $(D_0,\ D_1) = S1_c - S2_r = ((S1_r - S2_r),\ S1_i)$
0x1E	CIRADD D -S1 -S2	Complex with Real Integer Addition $(D_0,\ D_1) = -S1_c - S2_r = ((-S1_r - S2_r),\ -S1_i)$
0x1F	CIRADD D -S1 S2	Complex with Real Integer Addition $(D_0,\ D_1) = -S1_c + S2_r = ((-S1_r + S2_r),\ -S1_i)$
0x20	VFABSADD D S1 S2	Vectorial Absolute Floating Point Addition $(D_0,\ D_1) = \lvert S1 + S2\rvert = (\lvert(S1_0 + S2_0)\rvert,\ \lvert(S1_1 + S2_1)\rvert)$
0x21	VFABSADD D S1 -S2	Vectorial Absolute Floating Point Subtraction $(D_0,\ D_1) = \lvert S1 - S2\rvert = (\lvert(S1_0 - S2_0)\rvert,\ \lvert(S1_1 - S2_1)\rvert)$
0x22	VFABSADD D -S1 -S2	Vectorial Absolute Floating Point Addition $(D_0,\ D_1) = \lvert -S1 - S2\rvert = (\lvert(-S1_0 - S2_0)\rvert,\ \lvert(-S1_1 - S2_1)\rvert)$
0x23	VFABSADD D -S1 S2	Vectorial Absolute Floating Point Addition $(D_0,\ D_1) = \lvert -S1 + S2\rvert = (\lvert(-S1_0 + S2_0)\rvert,\ \lvert(-S1_1 + S2_1)\rvert)$
0x24	VIABSADD D S1 S2	Integer Vectorial Absolute Integer Addition $(D_0,\ D_1) = \lvert S1 + S2\rvert = (\lvert(S1_0 + S2_0)\rvert,\ \lvert(S1_1 + S2_1)\rvert)$
0x25	VIABSADD D S1 -S2	Vectorial Absolute Integer Subtraction $(D_0,\ D_1) = \lvert S1 - S2\rvert = (\lvert(S1_0 - S2_0)\rvert,\ \lvert(S1_1 - S2_1)\rvert)$
0x26	VIABSADD D -S1 -S2	Vectorial Absolute Integer Addition $(D_0,\ D_1) = \lvert -S1 - S2\rvert = (\lvert(-S1_0 - S2_0)\rvert,\ \lvert(-S1_1 - S2_1)\rvert)$
0x27	VIABSADD D -S1 S2	Vectorial Absolute Integer Addition $(D_0,\ D_1) = \lvert -S1 + S2\rvert = (\lvert(-S1_0 + S2_0)\rvert,\ \lvert(-S1_1 + S2_1)\rvert)$

ADD/SUB Opcode	Mnemonic	Description
0x28	VFADDSUB D1D2 S1 S2	Vectorial Floating Point Addition and Subtraction $(D1_0,\ D1_1) = (S1_0 + S2_0,\ S1_1 + S2_1)$ $(D2_0,\ D2_1) = (S1_0 - S2_0,\ S1_1 - S2_1)$
0x29	FADD D S1 S2	Floating Point Addition $D = S1 + S2$
0x2A	FADD D S1 -S2	Floating Point Addition $D = S1 - S2$
0x2B	FADD D -S1 -S2	Floating Point Addition $D = -S1 - S2$
0x2C	FADD D -S1 S2	Floating Point Addition $D = -S1 + S2$
0x2D	FADDSUB D1D2 S1 S2	Floating Point Addition and Subtraction $D1 = (S1 + S2)$ $D2 = (S1 - S2)$
0x2E	IADD D S1 S2	Integer Addition $D = S1 + S2$
0x2F	IADD D S1 -S2	Integer Addition $D = S1 - S2$
0x30	IADD D -S1 -S2	Integer Addition $D = -S1 - S2$
0x31	IADD D -S1 S2	Integer Addition $D = -S1 + S2$
0x32	IADDSUB D1D2 S1 S2	Integer Addition and Subtraction $D1 = (S1 + S2)$ $D2 = (S1 - S2)$
0x33	FABSADD D S1 S2	Floating Point Absolute Addition $D = \lvert S1 + S2 \rvert$
0x34	FABSADD D S1 -S2	Floating Point Absolute Addition $D = \lvert S1 - S2 \rvert$
0x35	FABSADD D -S1 -S2	Floating Point Absolute Addition $D = \lvert -S1 - S2 \rvert$
0x36	FABSADD D -S1 S2	Floating Point Addition $D = \lvert -S1 + S2 \rvert$
0x37	IABSADD D S1 S2	Integer Addition $D = \lvert S1 + S2 \rvert$
0x38	IABSADD D S1 -S2	Integer Addition $D = \lvert S1 - S2 \rvert$

ADD/SUB Opcode	Mnemonic	Description
0x39	IABSADD D -S1 -S2	Integer Addition $D = \lvert -S1 - S2 \rvert$
0x3A	IABSADD D -S1 S2	Integer Addition $D = \lvert -S1 + S2 \rvert$
0x3B	VFMIN D S1 S2	Vectorial Floating Point Minimum $(D_0,\ D_1) = (MIN(S1_0,\ S2_0),\ MIN(S1_1,\ S2_1))$
0x3C	VIMIN D S1 S2	Vectorial Integer Minimum $(D_0,\ D_1) = (MIN(S1_0,\ S2_0),\ MIN(S1_1,\ S2_1))$
0x3D	VFMAX D S1 S2	Vectorial Floating Point Maximum $(D_0,\ D_1) = (MAX(S1_0,\ S2_0),\ MAX(S1_1,\ S2_1))$
0x3E	VIMAX D S1 S2	Vectorial Integer Maximum $(D_0,\ D_1) = (MAX(S1_0,\ S2_0),\ MAX(S1_1,\ S2_1))$
0x3F	VFMINMAX D1D2 S1 S2	Vectorial Floating Point Min And Max $(D1_0,\ D1_1) = (MIN(S1_0,\ S2_0),\ MIN(S1_1,\ S2_1))$ $(D2_0,\ D2_1) = (MAX(S1_0,\ S2_0),\ MAX(S1_1,\ S2_1))$
0x40	VIMINMAX D1D2 S1 S2	Vectorial Integer Min And Max $(D1_0,\ D1_1) = (MIN(S1_0,\ S2_0),\ MIN(S1_1,\ S2_1))$ $(D2_0,\ D2_1) = (MAX(S1_0,\ S2_0),\ MAX(S1_1,\ S2_1))$
0x41	VSWAP S1 S2	Vectorial swap (vectorially conditioned) $fpucond0 = 1 \Rightarrow S1_0 \leftarrow S2_0; S1_0 \rightarrow S2_0$ $fpucond1 = 1 \Rightarrow S1_1 \leftarrow S2_1; S1_1 \rightarrow S2_1$ $fpucond0 = 0 \Rightarrow S1_0 \leftarrow S1_0; S2_0 \rightarrow S2_0$ $fpucond1 = 0 \Rightarrow S1_1 \leftarrow S1_1; S2_1 \rightarrow S2_1$
0x42	FMIN D S1 S2	Floating Point Minimum $D = MIN(S_1,\ S_2)$
0x43	IMIN D S1 S2	Integer Minimum $D = MIN(S_1,\ S_2)$
0x44	FMAX D S1 S2	Floating Point Maximum $D = MAX(S_1,\ S_2)$
0x45	IMAX D S1 S2	Integer Maximum $D = MAX(S_1,\ S_2)$
0x46	FMINMAX D1D2 S1 S2	Floating Point Min And Max $D1 = MIN(S1,\ S2)$ $D1 = MAX(S1,\ S2)$

ADD/SUB Opcode	Mnemonic	Description
0x47	IMINMAX D1D2 S1 S2	Integer Min And Max $D1 = MIN(S1, S2)$ $D1 = MAX(S1, S2)$
0x48	FSEL D S1 S2	Floating Point Selection (conditioned) $fpucond0 = 1 \Rightarrow D = S1$ $fpucond0 = 0 \Rightarrow D = S2$
0x49	ISEL D S1 S2	Integer Selection (conditioned) $fpucond0 = 1 \Rightarrow D = S1$ $fpucond0 = 0 \Rightarrow D = S2$
0x5A	SWAP S1 S2	Swap (conditioned) $fpucond0 = 1 \Rightarrow S1 \leftarrow S2; S1 \rightarrow S2$ $fpucond0 = 0 \Rightarrow S1 \leftarrow S1; S2 \rightarrow S2$
0x61	CFEQ S1 S2	Complex Floating Point Equal
0x62	CFNE S1 S2	Complex Floating Point not Equal
0x7B	CIEQ S1 S2	Complex Integer Equal
0x7C	CINE S1 S2	Complex Integer not Equal
0x63	VFEQ S1 S2	Vectorial Floating Point Equal
0x64	VFLT S1 S2	Vectorial Floating Point Less Than
0x65	VFGT S1 S2	Vectorial Floating Point Greater Than
0x66	VFLE S1 S2	Vectorial Floating Point Less or Equal
0x67	VFGE S1 S2	Vectorial Floating Point Greater or Equal
0x68	VFNE S1 S2	Vectorial Floating Point Not Equal
0x69	VIEQ S1 S2	Vectorial Integer Equal
0x6A	VILT S1 S2	Vectorial Signed Integer Less Than
0x6B	VIGT S1 S2	Vectorial Signed Integer Greater Than
0x6C	VILE S1 S2	Vectorial Signed Integer Less or Equal
0x6D	VIGE S1 S2	Vectorial Signed Integer Greater or Equal
0x6E	VINE S1 S2	Vectorial Integer Not Equal
0x6F	FEQ S1 S2	Floating Point Equal
0x70	FLT S1 S2	Floating Point Less Than
0x71	FGT S1 S2	Floating Point Greater Than
0x72	FLE S1 S2	Floating Point Less or Equal
0x73	FGE S1 S2	Floating Point Greater or Equal
0x74	FNE S1 S2	Floating Point Not Equal
0x75	IEQ S1 S2	Integer Equal

ADD/SUB Opcode	Mnemonic	Description
0x76	ILT S1 S2	Integer Signed Less Than
0x77	IGT S1 S2	Integer Signed Greater Than
0x78	ILE S1 S2	Integer Signed Less or Equal
0x79	IGE S1 S2	Integer Signed Greater or Equal
0x7A	INE S1 S2	Integer Not Equal
0x7D	VIADDSUB D1D2 S1 S2	Vectorial Integer Addition and Subtraction $(D1_0,\ D1_1) = (S1_0 + S2_0,\ S1_1 + S2_1)$ $(D2_0,\ D2_1) = (S1_0 - S2_0,\ S1_1 - S2_1)$
0x7F	NOP	Not OPeration

3.4.3 ADD/SUB Operators

The ADD/SUB operators allow computing different types of additions and subtractions on integer and floating point data types: complex, vectorial and single additions. They also allow to explicitly execute an addition and a subtraction concurrently on the same data and also execute a certain number of miscellaneous instructions such as absolute value computation, min/max, conditioned selection of input data, swap, etc. In complex additions, it is feasible to conjugate the second or both the operands as well as add a complex number with a real one.

ADD/SUB instructions on floating point or 32-bit signed integers include:

- Complex addition/subtraction
  - standard,
  - standard with conjugation of input operands
  - between real and complex operands
- Vectorial and scalar addition, subtraction, add/sub
  - addition, subtraction
  - add/sub
  - absolute value
  - min, max, min/max
  - swap input operands
  - selection and compare

The ADD/SUB floating-point execution unit operates on 40-bit floating-point operands and outputs 40-bit floating-point results. The ADD/SUB integer device operates on 32-bit integer operands and outputs 32-bit integer results. In the case of integer instructions, only 32 LSBs are treated, while the 8 MSBs remain unchanged see Section 4.2 ”Data Organization” on page 63.

3.4.3.1 Latency

Each ADD/SUB computation takes two pipeline cycles to be completed. During the first cycle the input data (read from the Register File at the previous cycle) enters the adder and a first half of the instruction is executed. During the second cycle the last part of the instruction is executed and the result is ready to be written in the Register File. At the next cycle, it is actually written in the Register File. From this cycle on the result can be read for the subsequent operations. So,

the total adder pipeline is four cycles long, two cycles for arithmetic computation plus two cycles necessary for the input data read from and the output data write to the Register File.

The Table 3-12 illustrates this pipeline scheme for a simple addition operation:

Z = X + Y

where X and Y are the adder input data already present in the Register File and Z is the result (adder output data) that will be written to the Register File.

Table 3-12. ADD/SUB Pipeline

Cycle	Operation	Note
0	Data X and Y are read from the Register File	decoding and RF read
1	Data X and Y enter the adder - First half instruction execution	computation
2	Second half instruction execution - Data Z (result) output the adder	computation
3	Data Z written to the Register File	write back to RF

As shown in the above table, cycle 1 and cycle 2 are the two adder pipeline cycles (computation cycles), while cycle 0 and cycle 3 are used to access the Register File. From cycle 3 on the result is already available in the Register File for a read access.

This bypass logic allows reading the written data saving a pipeline cycle (latency is still 4 cycles, but loops on accumulated data can be reduced to 3 cycles only).

3.4.4 MUL Issue

The MUL issue drives:

- the block containing 4 multiply arithmetic operators plus 2 adders for complex domain multiplies (MUL1, MUL2, MUL3, MUL4, CADD1, CADD2 in Figure 3-3 on page 10);
- a pair of shifters (SH1, SH2);
- a pair of float integer converters (CONV1, CONV2);
- seed generators for division and trascendent functions (DIV1, DIV2);

The MUL issue takes two scalar, vectorial or complex input operands from the RF. A scalar, vectorial or complex result is returned to the Register File.

Table 3-13. MUL Issue

vliw[53:52]	vliw[60:54]	vliw[84:77]	vliw[68:61]	vliw[76:69]
MUL predication	MUL opcode	RF add4 (D)	RF add1 (S1)	RF add0 (S2)

- **RF add 0: Register File address port 0**

It specifies one of the 256 RF registers as input operand0.

- **RF add 1: Register File address port 1**

It specifies one of the 256 RF registers as input operand1.

- **RF add 4: Register File address port 6**

It specifies one of the 256 RF registers as output operand.

- **MUL opcode: MUL microcode**

It specifies the MUL operation to perform.

- **MUL predication**

It specifies one of the four predication registers, if the condition of the predication register is false the issue will not be executed.

Table 3-14. MUL Opcodes

MUL Code	Mnemonic	Description
0x0	CFMUL D S1 S2	Complex Floating Point Multiplication $(D_0,\ D_1) = S1_c \times S2_c = ((S1_r \cdot S2_r - S1_i \cdot S2_i),\ (S1_i \cdot S2_r + S2_i \cdot S1_r))$
0x1	CFMUL D S1 -S2	Complex Floating Point Multiplication $(D_0,\ D_1) = S1_c \times -S2_c = ((-S1_r \cdot S2_r + S1_i \cdot S2_i),\ (-S1_i \cdot S2_r - S2_i \cdot S1_r))$
0x2	CFJMUL D S1 S2	Complex Floating Point Conjugated Multiplication $(D_0,\ D_1) = S1_c \times \overline{S2_c} = ((S1_r \cdot S2_r + S1_i \cdot S2_i),\ (S1_i \cdot S2_r - S2_i \cdot S1_r))$
0x3	CFJMUL D S1 -S2	Complex Floating Point Conjugated Multiplication $(D_0,\ D_1) = S1_c \times \overline{-S2_c} = ((-S1_r \cdot S2_r - S1_i \cdot S2_i),\ (-S1_i \cdot S2_r + S2_i \cdot S1_r))$
0x4	CFJJMUL D S1S2	Complex Double-Conjugate Floating Point Multiplication $(D_0,\ D_1) = \overline{S1_c} \times \overline{S2_c} = ((S1_r \cdot S2_r - S1_i \cdot S2_i),\ (-S1_i \cdot S2_r - S2_i \cdot S1_r))$
0x5	CFJJMUL D S1 -S2	Complex Double-Conjugate Floating Point Multiplication $(D_0,\ D_1) = \overline{S1_c} \times \overline{-S2_c} = ((-S1_r \cdot S2_r + S1_i \cdot S2_i),\ (S1_i \cdot S2_r + S2_i \cdot S1_r))$
0x6	CFRMUL D S1 S2	Complex with Real Floating Point Multiplication $(D_0,\ D_1) = S1_c \times S2_r = (S1_r \cdot S2_r,\ S1_i \cdot S2_r)$
0x7	CFRMUL D S1 -S2	Complex Integer with Real Integer Multiplication $(D_0,\ D_1) = S1_c \times -S2_r = (-S1_r \cdot S2_r,\ -S1_i \cdot S2_r)$
0x8	CIMUL D S1 S2	Complex Integer Multiplication $(D_0,\ D_1) = S1_c \times S2_c = ((S1_r \cdot S2_r - S1_i \cdot S2_i),\ (S1_i \cdot S2_r + S2_i \cdot S_{r1}))$
0x9	CIMUL D S1 -S2	Complex Integer Multiplication $(D_0,\ D_1) = S1_c \times -S2_c = ((-S1_r \cdot S2_r + S1_i \cdot S2_i),\ (-S1_i \cdot S2_r - S2_i \cdot S1_r))$
0xA	CIJMUL D S1 S2	Complex Integer Conjugated Multiplication $(D_0,\ D_1) = S1_c \times \overline{S2_c} = ((S1_r \cdot S2_r + S1_i \cdot S2_i),\ (S1_i \cdot S2_r - S2_i \cdot S1_r))$

MUL Code	Mnemonic	Description
0xB	CIJMUL D S1 -S2	Complex Integer Conjugated Multiplication $(D_0,\ D_1) = S1_c \times \overline{-S2_c} = ((-S1_r \cdot S2_r - S1_i \cdot S2_i),\ (-S1_i \cdot S2_r + S2_i \cdot S1_r))$
0xC	CIJJMUL D S1 S2	Complex Double-Conjugate Integer Multiplication $(D_0,\ D_1) = \overline{S1_c} \times \overline{S2_c} = ((S1_r \cdot S2_r - S1_i \cdot S2_i),\ (-S1_i \cdot S2_r - S2_i \cdot S1_r))$
0xD	CIJJMUL D S1 -S2	Complex Double-Conjugate Integer Multiplication $(D_0,\ D_1) = \overline{S1_c} \times \overline{-S2_c} = ((-S1_r \cdot S2_r + S1_i \cdot S2_i),\ (S1_i \cdot S2_r + S2_i \cdot S1_r))$
0xE	CIRMUL D S1 S2	Complex with Real Integer Multiplication $(D_0,\ D_1) = S1_c \times S2_r = (S1_r \cdot S2_r,\ S1_i \cdot S2_r)$
0xF	CIRMUL D S1 -S2	Complex with Real Integer Multiplication $(D_0,\ D_1) = S1_c \times -S2_r = (-S1_r \cdot S2_r,\ -S1_i \cdot S2_r)$
0x20	VFMUL D S1 S2	Vectorial Floating Point Multiplication $(D_0,\ D_1) = (S1_0 \times S2_0,\ S1_1 \times S2_1)$
0x21	VFMUL D S1 -S2	Vectorial Floating Point Multiplication $(D_0,\ D_1) = (S1_0 \times -S2_0,\ S1_1 \times -S2_1)$
0x22	VIMUL D S1 S2	Vectorial Integer Multiplication $(D_0,\ D_1) = (S1_0 \times S2_0,\ S1_1 \times S2_1)$
0x23	VIMUL D S1 -S2	Vectorial Integer Multiplication $(D_0,\ D_1) = (S1_0 \times -S2_0,\ S1_1 \times -S2_1)$
0x28	FMUL D S1 S2	Floating Point Multiplication $D = S1 \times S2$
0x29	FMUL D S1 -S2	Floating Point Multiplication $D = S1 \times -S2$
0x2A	IMUL D S1 S2	Integer Multiplication $D = S1 \times S2$
0x2B	IMUL D S1 -S2	Integer Multiplication $D = S1 \times -S2$
0x30	VSH D S1 S2	Vectorial Shift Store in (D_0,D_1) the operand ($S1_0$,$S1_1$) shifted by MSBs of ($S2_0$,$S2_1$). See Section 3.4.6.1 "Shift and Bitwise Operations" on page 24
0x31	VSHAND D S1 S2	Vectorial Shift and bitwise AND Store in (D_0,D_1) the operand ($S1_0$,$S1_1$) shifted and bitwise ANDed of ($S2_0$,$S2_1$). See Section 3.4.6.1 "Shift and Bitwise Operations" on page 24

MUL Code	Mnemonic	Description
0x32	VSHOR D S1 S2	Vectorial Shift and bitwise OR Store in (D_0,D_1) the operand ($S1_0$,$S1_1$) shifted and bitwise ORed of ($S2_0$,$S2_1$).
0x33	VSHXOR D S1 S2	Vectorial Shift and bitwise XOR Store in (D_0,D_1) the operand ($S1_0$,$S1_1$) shifted and bitwise XORed of ($S2_0$,$S2_1$).
0x34	VBITSET D S1 S2	Vectorial Bit Set Store in (D_0,D_1) the operand ($S1_0$,$S1_1$) with the N-th bit (value 0 to 31, specified into the ($S2_0$,$S2_1$) MSB) set. See Section 3.4.6.2 ”Bit Manipulation” on page 25
0x35	VBITCLR D S1 S2	Vectorial Bit Clear Store in (D_0,D_1) the operand ($S1_0$,$S1_1$) with the N-th bit (value 0 to 31, specified into the ($S2_0$,$S2_1$) MSB) cleared.
0x36	VBITTGL D S1 S2	Vectorial Bit Toggle Store in (D_0,D_1) the operand ($S1_0$,$S1_1$) with the N-th bit (value 0 to 31, specified into the ($S2_0$,$S2_1$) MSB) inverted.
0x37	VBITTST D S1 S2	Vectorial Bit Test Store in (D_0,D_1) the value of the N. bit (value 0 to 31, specified into the ($S2_0$,$S2_1$) MSB) of operand ($S1_0$,$S1_1$).
0x38	SH D S1 S2	Shift See Section 3.4.6.1 ”Shift and Bitwise Operations” on page 24
0x39	SHAND D S1 S2	Shift and bitwise AND See Section 3.4.6.1 ”Shift and Bitwise Operations” on page 24
0x3A	SHOR D S1 S2	Shift and bitwise OR
0x3B	SHXOR D S1 S2	Shift and bitwise XOR
0x3C	BITSET D S1 S2	Bit Set Store in D the operand S1 with the N. bit (value 0 to 31, specified into the S2 MSB) set.
0x3D	BITCLR D S1 S2	Bit Clear Store in D the operand S1 with the N. bit (value 0 to 31, specified into the S2 MSB) cleared.
0x3E	BITTGL D S1 S2	Bit Toggle Store in D the operand S1 with the N. bit (value 0 to 31, specified into the S2 MSB) inverted.
0x3F	BITTST D S1 S2	Bit Test Store in D the value of the N. bit (value 0 to 31, specified into the S2 MSB) of operand S1.
0x40	VFTOI D S1	Vectorial Floating Point To Integer Conversion
0x41	VITOF D S1	Vectorial Integer To Floating Point Conversion
0x42	FTOI D S1	Floating Point To Integer Conversion
0x43	ITOF D S1	Integer To Floating Point Conversion
0x44	VRF2RF D S1	Vectorial Register File To Register File Movement $(D_0,\ D_1) = (S1_0,\ S1_1)$
0x45	RF2RF D S1	Register File To Register File Movement $D = S1$

MUL Code	Mnemonic	Description
0x46	INVSEED D S1	Inversion Seed Generation $D = \frac{1}{S1}$
0x47	INVSQRTSEED D S1	Inversion Square Root Seed Generation $D = \frac{1}{\sqrt{S1}}$
0x48	VINVSEED D S1	Vectorial Inversion Seed Generation $(D_0, D_1) = \left(\frac{1}{S1_0}, \frac{1}{S1_1}\right)$
0x49	VINVSQRTSEED D S1	Vectorial Inversion Square Root Seed Generation $(D_0, D_1) = \left(\frac{1}{\sqrt{S1_0}}, \frac{1}{\sqrt{S1_1}}\right)$
0x7F	NOP	No Operation

3.4.4.1 *Latency of MUL Issue*

Two cycles are necessary to fetch the input operands and store the result. The arithmetic operation is performed in two additional cycles (scalar or vectorial case) or 4 cycles (complex domain operations). So, the total multiplier pipeline is 4 cycles long (vectorial and scalar instructions) or 6 cycles long (complex instructions).

As an example of these two different pipeline cycles, two cases are described:

1. Vectorial or single product with two pipeline cycles

Table 3-15 shows a product between two real numbers.

The operation is

Z = X × Y

where X and Y are the multiplier input data already present in the Register File and Z is the result (multiplier output data) that will be written to the Register File. X, Y and Z are real numbers (not complex numbers). The pipeline scheme is shown in Table 3-15.

Table 3-15. MUL Pipeline for Scalar or Vectorial Operations

Cycle	Operation	Note
0	Data X and Y are read from the Register File	decoding plus RF read
1	Data X and Y enter the multiplier - First half instruction execution	computation
2	Second half instruction execution - Data Z (result) output the multiplier	
3	Data Z written to the Register File	write back to RF

As shown in the table above, cycle 1 and cycle 2 are the two multiplier pipeline cycles (computation cycles), while cycle 0 and cycle 3 are used to access the Register File.

The bypass logic implemented in the Register File allows to read the written data starting from cycle 3 thus saving a pipeline cycle (so latency is still 4 cycles, but loops on accumulated data can be reduced to 3 cycles).

2. Complex product with four pipeline cycles

Table 3-16 on page 23 shows a product between two complex numbers.

The operation is always the same:

$Z = X \times Y$

but in this case X, Y and Z are complex values. Note that when dealing with complex numbers, the product is equivalent to

$ZRe = (XRe \times YRe) - (XIm \times YIm)$

$ZIm = (XIm \times YRe) + (XRe \times YIm)$

where the subscript Re identifies the real part of the complex number and the suffix Im identifies the imaginary part of the complex number.

The pipeline scheme is shown in Table 3-16 on page 23.

Table 3-16. MUL Complex Pipeline

Cycle	Operation	Note
0	Data XRe, XIm, YRe and YIm are read from the Register File	decoding plus RF read
1	Data XRe, XIm, YRe and YIm enter the multiplier - First half of the four multiplication instructions execution	computation
2	Second half of the four multiplication instructions execution - Intermediate results output the four multipliers	
3	Intermediate results enter the subtractor and the adder - First half of the addition and of the subtraction instructions execution	
4	Second half of the addition and of the subtraction instructions execution - Data Z (ZRe and ZIm) outputs the adder and the subtractor	
5	Data Z (ZRe and ZIm) written to the Register File	write back to RF

Cycles 1, 2, 3 and 4 are the four-computation pipeline cycles (cycles 1 and 2 are used for the four products and cycles 3 and 4 are used for the addition and the subtraction), while cycle 0 and cycle 5 are used to access the Register File.

3.4.5 Multiplier Operator

The multiplier operator executes products between 40-bit floating-point numbers or 32-bit signed integers. In the case of integer instructions, only 32 LSBs are treated, while the 8 MSBs remain unchanged see Section 4.2 ”Data Organization” on page 63.

Products are allowed between scalar, vectorial and complex numbers. In particular, the complex product is hardwired with the use of four multipliers, one adder and one subtractor, so that the complex multiply can deliver a new result at every cycle.

The pipeline depth of the multiply operation is four cycles. Besides, in complex products it is feasible to conjugate the second or both the operands as well as to multiply a complex number with a real one.

Multiplier instructions on floating point or 32-bit signed integers include:

- Complex products
  - standard,
  - standard with conjugation of input operands
  - between real and complex operands
- Vectorial and scalar products

3.4.6 Shifter Operator

The shifter operator performs scalar and vectorial instructions.

Shifter operations include:

- Arithmetic left and right shifts on 32-bit signed integers
- Left rotate of the 32 Least Significant Bits of a 40-bit data (see “Data Organization” on page 63)
- Left rotate module 40-bit
- Logical Shift Right
- Shift Left inserting "1"
- Bit manipulation operations, including set, clear, toggle, and test bits
- Encode of "0" or "1" starting from either left or right

The shifter operator belongs to the MUL issue (see Section 3.4.4 ”MUL Issue” on page 18) with a total latency of 4 cycles like in Section 3.4.4.1 ”Latency of MUL Issue” on page 22.

3.4.6.1 Shift and Bitwise Operations

The Shifter operator can combine a shift/rotate operation with a bitwise operation (see (V)SH, (V)SHAND, (V)SHOR, (V)SHXOR Table 3-11 on page 11). The first operand S1 is the data to be shifted. The 8 MSBs of the second operand S2 encode the shift operation, see Table 3-18 on page 25. The remaining 32 LSBs of the operand S2 (mask data) are used as second operands for bitwise operations: (V)SHAND, (V)SHOR and (V)XOR.

Table 3-17. Second Operand Format (S2)

39	38	37	36	35	34	33	32	31	0
shift type								mask data	

Table 3-18. Shift Encoding

39	38	37	36	35	34	33	32	
0	0	0	N	N	N	N	N	Left arithmetic shift and N. of shifts (0 to +31)
0	0	1	N	N	N	N	N	Right arithmetic shift and N. of shifts (0 to +31)
0	1	N	N	N	N	N	N	Left rotate mod. 40 and N. of shifts (0 to +39)
1	0	0	N	N	N	N	N	"1" Insert Left Shift and N. of shifts (0 to 31). Shift left and insert '1'.
1	0	1	N	N	N	N	N	Logic shift Right (0 to 31)
1	1	0	N	N	N	N	N	Left rotate mod. 32 and N. of shifts (0 to 31)
1	1	1	-	-	-	L | R	0 | 1	Encode 0|1 starting from Left(0)|Right(1). It returns in D an integer between 0 and 31 corresponding to the bit position of the first occurrence of a '1' or '0' (bit 32), starting from left or right (bit 33)

NOTE Rev A specific: The sign bit of Arithmetic Shift Left is not shifted out (arithmetic shift left and logic shift left are not equal).

3.4.6.2 Bit Manipulation

(V)BITSET, (V)BITCLR, (V)BITTGL and (V)BITTST use the second operands bits from 32 to 37 (S2) to specify the bit position, the 32 LSBs are unused.

Table 3-19. Bit Manipulation Format

39 38	37 36 35 34 33 32	31 ... 0
not used	N. bit position (values from 0 to 39)	not used

3.4.7 Converter Operator

The converter operator performs scalar and vectorial conversion from 40-bit floating point to 32-bit signed integer and from 32-bit signed integer to 40-bit floating-point numbers.

The conversion is ANSI C standard compliant, i.e. it truncates the floating point decimal part.

The converter operator belongs to the MUL issue (see Section 3.4.4 "MUL Issue" on page 18) with a total latency of 4 cycles like in Section 3.4.4.1 "Latency of MUL Issue" on page 22.

3.4.8 Mathematic Seed Generator

This device generates seeds that can be used in algorithms for the implementation of 1/x and 1/SQRT(x) 40-bit floating-point functions.

It takes one input floating-point operand from the Register File and returns one floating-point result (the appropriate seed read from a look-up table) to the Register File.

The seed generator belongs to the MUL issue (see Section 3.4.4 "MUL Issue" on page 18) with a total latency of 4 cycles like in Section 3.4.4.1 "Latency of MUL Issue" on page 22.

3.4.9 Operator Status Flags

The mAgicV DSP is IEEE 754 compliant for most aspects. As a consequence, there are five types of exceptions:

- inexact result
- underflow
- overflow
- divide by zero
- invalid operation

Rounding mode is round to nearest (the number is rounded to the nearest representable value; this mode has the smallest errors associated with it because statistically rounding up and rounding down occur with the same frequency).

There are however a few differences from the IEEE 754 standard:

- No traps are implemented. When an exception is detected, a status flag is set.
- Denormalized numbers are not implemented.

Notes:

- computed result - means the result which would have been computed if both exponents range and precision were unbounded (infinitely precise result).
- rounded result - means the result after rounding to nearest.
- delivered result - means the results that is returned by the operation and which fits with the format.

3.4.9.1 *Special Floating Point Values*

Table 3-20 shows a set of special values as coded in the 40-bit extension to the IEEE 754 floating point format used by mAgicV.

Table 3-20. Special Floating Point Values

Number	Sign	Exponent	Mantissa	Representation
+NaN	0	255	any configuration	
-NaN	1	255	any configuration	
+infinity	0	255	0	0x7F80000000
-infinity	1	255	0	0xFF80000000
+zero	0	0	0	0x0000000000
-zero	1	0	0	0x8000000000
0.5	0	126	0	0x3F00000000
1.0	0	127	0	0x3F80000000
1.5	0	127	2^{-1}	0x3FC0000000
2.0	0	128	0	0x4000000000
3.0	0	128	2^{-1}	0x4040000000

3.4.9.2 *Invalid Operation (Inop)*

Only invalid operands for the operation to be performed can cause invalid operation flag to be set.

NaN is signaled by the following operations:

- $(+\infty) + (-\infty)$
- $0 \times \infty$

- $1/\sqrt{(\leq 0)}$
- ∞or NaN input operand for floating-point to integer conversion

a quiet NaN is delivered when an invalid operation exception occurs.

3.4.9.3 *Division by Zero (Div by Zero)*

If input operand is zero, a division by zero flag is set.

The delivered result is ∞ with correct sign when a division by zero exception occurs.

3.4.9.4 *Overflow (Ovf)*

If the magnitude of the rounded result exceeds 2^{128}, an overflow exception is signaled.

The delivered result is ∞, with the sign of the computed result when an overflow exception occurs.

3.4.9.5 *Underflow (Udf)*

Two events can cause underflow: tininess and loss of accuracy.

As no traps are implemented, an underflow is signaled only when both tininess and loss of accuracy occur.

Tininess: computed result lies between $\pm 2^{-126}$ and is not zero

Ex: 2^{-130} (1.*x*..*x*)

Loss of accuracy: delivered result differs from computed result (which means the result is inexact)

Ex: computed result=2^{-130} (1.*x*..*x*1100)

delivered result = 2^{-126} (0.0001*x*..*x*1)

As denormalized numbers are not implemented the delivered result is zero when an underflow exception occurs.

3.4.9.6 *Inexact (Inex)*

If rounded result differs from computed result or if an overflow exception occurs, an inexact exception is signaled.

The delivered result is the rounded result or the overflowed result (∞with the right sign), or the underflowed result (0 with the right sign).

Note that if the underflow or overflow flag is high, the inexact flag will be high.

3.4.9.7 *32-bit Signed Integer Flags*

When the input operands are in 32-bit signed integer format the following rules apply:

- Flags Inop, Udf, and Inex are cleared
- Flag overflow has the following meaning: if the result of the operation is not between 2^{31} and $-(2^{31}+1)$ the overflow flag is set. The delivered result is the overflowed result;

There is one more flag for the ALU: the Carry flag can be used to extend the result with one more bit.

3.4.9.8 *mAgicV DSP Flags Summary*

Each operator in Figure 3-3 on page 10 sets its own flag:

MUL1,MUL2,MUL3,MUL4:

- Inop, Ovf, Udf, Inex

CADD1,CADD2:

- Inop, Ovf, Udf, Inex, Carry

ADD1, ADD2:

- Inop, Ovf, Udf, Inex, Carry for Add output
- Inop, Ovf, Udf, Inex, Carry for Sub output

CONV1, CONV2:

- Integer to floating-point: (Inex, Inop, Ovf are all cleared)
- Floating-point to integer: Inex, Inop, Ovf

SH/LOG1, SH/LOG2:

- No flag

MIN/MAX1, MIN/MAX2:

- Inop

DIV1, DIV2:

- Inop, Ovf, Udf, Inex, Div by Zero

the total amount of flags is 64 (32 for operator path0 and 32 for operator path1), stored in the *MGCSTIKY0* and *MGCSTIKY1* registers.

The MGCEXCEPTION register (see Section 5-16 ”MGCEXCEPTION Register” on page 87) has dedicated bits which collect:

- A logical OR of all inop flags (BADIN bit);
- A logical OR of all overflow flags (BADOUT bit);
- A logical OR of all divzero flags (DIVZERO bit)

3.4.9.9 *MGCSTIKY0 and MGCSTIKY1*

Operators flags are accumulated in two 32 bit Sticky Status Registers (MGCSTKY0 for the Operator path0 and MGCSTKY1 for the path1) that are reset after reading.

Flags refer to the units of Figure 3-3 on page 10.

State flags are meaningful only if they assume the logic value ‘1’. These registers are cleared upon read.

Table 3-21. Format of MGCSTIKY0 and MGCSTIKY1 Registers

31	30	29	28	27	26	25	24
inopminmax	inopconv	ovfconv	inexconv	inopdiv	divbyzero	ovfdiv	udfdiv
23	22	21	20	19	18	17	16
inexdiv	inopadd1	ovfadd1	carryadd1	udfadd1	inexadd1	inopsub	ovfsub
15	14	13	12	11	10	9	8
carrysub	udfsub	inexsub	inopadd2	ovfadd2	carryadd2	udfadd2	inexadd2
7	6	5	4	3	2	1	0
inopmul2	ovfmul2	udfmul2	inexmul2	inopmul1	ovfmul1	udfmul1	inexmul1

- **inexmul1: inexact primary multiplier**
inexact from primary multiplier. Section 3.4.9.6 ”Inexact (Inex)” on page 27

- **udfmul1: underflow primary multiplier**
underflow from primary multiplier. Section 3.4.9.5 ”Underflow (Udf)” on page 27

- **ovfmul1: overflow primary multiplier**
overflow from primary multiplier. 3.4.9.4 ”Overflow (Ovf)” on page 27

- **inopmul1: inexact primary multiplier**
invalid operation from primary multiplier. Section 3.4.9.2 ”Invalid Operation (Inop)” on page 26

- **inexmul2: inexact secondary multiplier**
inexact from secondary multiplier (used in complex products).

- **udfmul2: underflow secondary multiplier**
underflow from secondary multiplier.

- **ovfmul2: overflow secondary multiplier**
overflow from secondary multiplier.

- **inopmul2: inexact secondary multiplier**
invalid operation from secondary multiplier.

- **inexadd2: inexact secondary adder**
inexact from secondary adder (used in complex products).

- **udfadd2: underflow secondary adder**
underflow from secondary adder.

- **carryadd2: carry secondary adder**
carry from secondary adder. Section 3.4.9.7 ”32-bit Signed Integer Flags” on page 27

- **ovfdd2: overflow secondary adder**
overflow from secondary adder.

- **inopadd2: inexact secondary adder**
invalid operation from secondary adder.

- **inexsub: inexact subtractor**
inexact from primary subtractor (used in complex products).

- **udfsub: underflow subtractor**
underflow from subtractor.

- **carrysub: carry subtractor**
carry from subtractor.

- **ovfsub: overflow subtractor**
overflow from subtractor.

- **inopsub: inexact subtractor**

invalid operation from subtractor.

- **inexadd1: inexact primary adder**

inexact from primary adder (used in complex products).

- **udfadd1: underflow primary adder**

underflow from primary adder.

- **carryadd1: carry primary adder**

carry from primary adder.

- **ovfdd1: overflow primary adder**

overflow from primary adder.

- **inopadd1: inexact primary adder**

invalid operation from primary adder.

- **inexdiv: inexact divider**

inexact from divider.

- **udfdiv: underflow divider**

underflow from divider.

- **ovfdiv: overflow divider**

overflow from divider.

- **divbyzero: division by zero divider**

division by zero operation from divider. Section 3.4.9.3 ”Division by Zero (Div by Zero)” on page 27

- **inopdiv: inexact divider**

invalid operation from divider.

- **inexconv: inexact converter**

inexact from converter.

- **pvfconv: overflow converter**

overflow from converter.

- **inopconv: invalid operand converter**

invalid operation from converter.

- **inopminmax: invalid operand minmax**

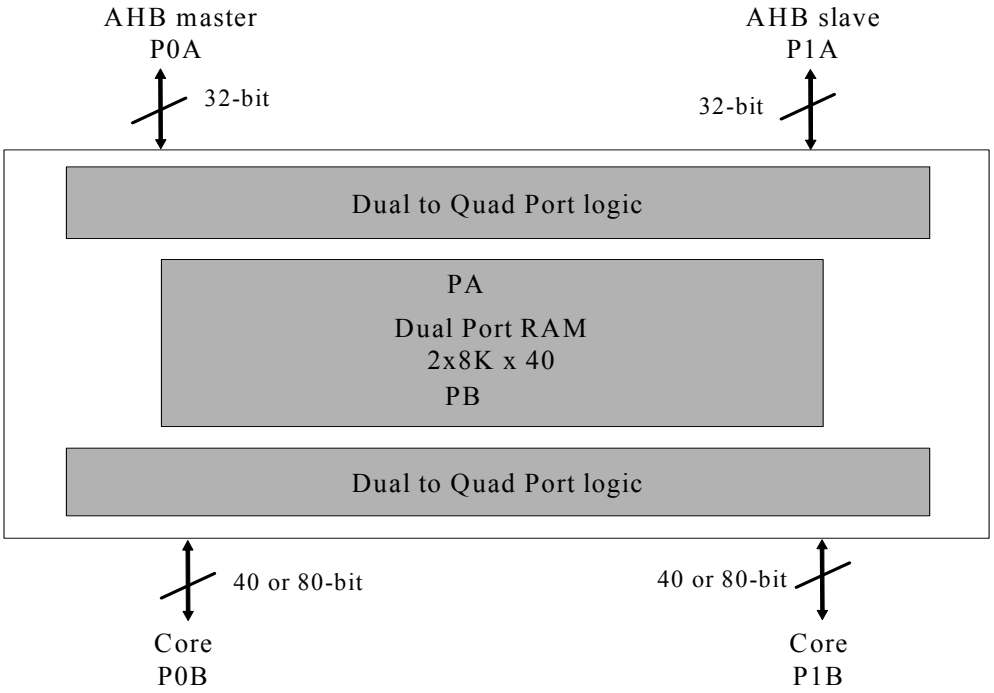
invalid operation from minmax.

3.5 On-Chip Data Memory

The on-chip Data Memory System contains 2 banks of 8K 40-bit words. It provides a maximum throughput of 6 words/cycle and it can be simultaneously accessed by three devices: the computational data path, the AHB master and the AHB slave. Simultaneously, the computational data-path can fetch and store a maximum of four 40-bit data per cycle, the AHB master can drive a single access of 32-bit word per cycle and the AHB slave can support single accesses of 32-bit word per cycle (for the management of the 32-bit accesses inside the 40-bit word see the alias in Section 3.7.2 ”Data Memory Accesses” on page 40). The simultaneous activity of the AHB master and slave requires an external multi-layer bus matrix implementation.

Each access through P0B (and/or through P1B) can either transfer a single 40-bit data (scalar access) or access a pair of consecutive memory locations aligned to even address (for operation on complex or vectorial data types). Accesses through P0B and P1B are reserved to the computational data-path and their addresses are generated by AGU0 and AGU1. See Figure 3-4 for the Data Memory system and Section 3.6 ”Address Generation Units” on page 32.

Figure 3-4. Quad Port Data Memory



3.5.1 Latency

This memory is a static RAM with 2-cycles of latency for read accesses and 1-cycle to write. Write and read latencies through the AHB are shown on Table 3-26 on page 39.

3.5.2 Access rules

To grant correct application functionality, simultaneous multiple Read and Write, or multiple Write accesses (see Section 3.5.2.2 on page 32 and Section 3.5.2.3 on page 32).

Simultaneous multiple read accesses are allowed (see Section 3.5.2.1 on page 32).

3.5.2.1 *Simultaneous Multiple Read Accesses to Identical Locations*

Simultaneous read accesses are allowed. This means that any operation like

- AHB master (P0A), AHB slave (P1A), Core (P0B and P1B) contemporary Read

can be executed without causing unpredictable results.

3.5.2.2 *Simultaneous Read/Write and Write/Write Accesses to Identical Locations producing Undefined Results*

1. AHB master (P0A) and Core (P0B) contemporary Write
2. AHB slave (P1A) and Core (P1B) contemporary Write
3. AHB master (P0A) and Core (P0B) contemporary Write and Read
4. AHB master (P0A) and Core (P0B) contemporary Read and Write
5. AHB slave (P1A) and Core (P1B) contemporary Write and Read
6. AHB slave (P1A) and Core (P1B) contemporary Read and Write

3.5.2.3 *Simultaneous Read/Write and Write/Write Accesses to Identical Locations producing Defined Results*

1. AHB master (P0A) and AHB slave (P1A) contemporary Write (P0A writes, P1A Write fails)
2. Core (P0B and P1B) contemporary Write (P0B writes, P1B Write fails)
3. AHB master (P0A) and Core (P1B) contemporary Write and Read (P0A writes, P1B reads old value)
4. AHB master (P0A) and Core (P1B) contemporary Read and Write (P0A reads new value, P1B writes)
5. AHB slave (P1A) and Core (P0B) contemporary Write and Read (P1A writes, P0B reads new value)
6. AHB slave (P1A) and Core (P0B) contemporary Read and Write (P1A reads old value, P0B writes)
7. AHB slave (P1A) and Core (P0B) contemporary Write (P1A Write fails, P0B writes)
8. AHB master (P0A) and Core (P1B) contemporary Write (P0A writes, P1B Write fails)

Only these simultaneous accesses at the same address are allowed

3.6 Address Generation Units

There are two identical Address Generation Units in mAgicV (see Figure 3-1 on page 2) named AGU0 and AGU1. Each AGU is driven by a dedicated VLIW issue (see Table 3-1 on page 3) and Section 3.6.1 below.

The AGU can generate complex/vectorial and scalar accesses. In complex/vectorial mode two words are accessed instead of one (scalar mode). The AGU supports linear addressing and DSP oriented features like circular buffers. The address generation unit is supported by a multi field Address Register File (ARF) composed of 4x16x16-bit registers, for a total of 64 16-bit integer registers. Registers named A0-A15 are used to manage 16-bit integers/pointers, while M0-M15 registers are for the 16-bit integer/pointer modifiers. When circular buffers are used, S0-S15 store the start addresses of the buffers, while L0-L15 store their lengths (zero length means no circular buffer). Each AGU contains also a private 16-bit TMP register (TMP0 and TMP1) which can be used by the AGU arithmetic and addressing operations. The AGU is able to per-

form 16-bit signed/unsigned integer arithmetic operations in parallel to the activities of the 40-bit floating point and the 32-bit signed integer operators block.

Table 3-22. 64-bit ARF Register

63 48	47 32	31 16	15 0
S	L	A	M

At every clock cycle each AGU can perform addressing (addressing mode) or arithmetic operations (arithmetic mode).

The output of both arithmetic and addressing operations are written either in the A field of an ARF register or in an internal AGU register named TMP.

The compiler, generating addressing or arithmetic AGU operations, can exploit different solutions in terms of AGU issue generation.

The most compact and orthogonal solution is to generate issues that select a single 64-bit ARFx (see Section 3.6.1.1 ”Default AGU Issue” on page 34); but sometimes it is convenient to use the My 16-bit field from a different 64-bit ARF. When two different ARFs are used, some other issues are inhibited because of the need for additional coding bits, thus creates overlapping on other issues.

3.6.1 AGU Issues

AGU issues are composed of at least an AGU code, an AGU predication field and 4 bits generally used to address a 64-bit ARF register. See Table 3-22 above.

The format of the AGU issue depends on the AGU code. Table 3-23 shows the AGU code format:

Table 3-23. AGU Code Format

6	5	4	3 2 1 0
0	write	vector	Addressing opcode
1	arithmetic opcode		

- **Bit 6: mode**

MODE=’1’, selects the addressing mode

MODE=’0’, selects the arithmetic mode

- **write**

In addressing mode:

WRITE=’1’, selects a write access

WRITE=’0’, selects a read access

- **vector**

In addressing mode:

VECTOR=’1’, selects a vector access

VECTOR=’0’, selects a scalar access

- **Addressing Opcode**

In addressing mode specifies the addressing operation

- **Arithmetic Opcode**

In arithmetic mode specifies the arithmetic operation

3.6.1.1 *Default AGU Issue*

This AGU issue is used in the arithmetic/addressing operations that use only one 64-bit ARF bundle composed of four 16-bit registers (Sx, Ax, Mx, Lx). The result is in Ax or TMP registers and it uses Ax, Mx, Sx, Lx or TMP as sources.

The format of the AGU0 issue is:

vliw[1:0]	vliw[95:89]	vliw[88:85]
AGU0 predication	AGU0 code	ARF add0

of the AGU1:

vliw[49:48]	vliw[47:41]	vliw[40:37]
AGU1 predication	AGU1 code	ARF add1

- **ARF addx: the ARF source/destination address**

It specifies one of the 16 boundled ARF registers. Note that each ARF register is composed of four sub-fields S,L,A,M for a total of 64-bit.

- **AGUx code: AGU microcode**

It specifies the AGU operation to perform.

- **AGUx predication**

It specifies one of the four predication registers. If the condition of the predication register is false the issue will not be executed.

3.6.2 Addressing

The addressing can be:

- circular, if the L field of the involved ARF is not zero
- normal, if L=0

In the case of non circular addressing, only the A and the M fields are used to generate the address. S and L are ignored and no modular addressing is performed.

The S field is considered only in circular addressing and it contains the base address of the addressed circular vector.

By executing the circular address operation S + (A-S+/-M)% L the A field will be modified as follows:

- If M>=0

  A=A+M if A+M-S<L

  A=A+M-L if A+M-S>=L

- If M<0

  A=A+M if A+M>=S

  A=A+M+L if A+M<S

In circular addressing, a **Boundary Flag** with the following rule is generated:

Bnd flag: 1 if Ax+Mx>=Sx+Lx

1 if Ax+Mx<Sx

0 otherwise

The boundary flag is registered into the MGCCONDITION register (see Section 3.9.2 ”Conditions and Status Flags” on page 47).

All the operations are in twos complement.

Addressing operations can generate the following two exceptions:

Addr_ovfl: signal address memory overflow when address > 0x3FFF (size of the “On-Chip Data Memory), counted in 40-bit words (see Section 3.5 ”On-Chip Data Memory” on page 31).

Parity_err: signal memory address odd in vectorial addressing

In case of exception the addressing operation is disabled. An addressing operation can modify the A field of an ARF (A-ARF) register.

Table 3-24 on page 35 shows the possible addressing modes. For further details see Section 8. ”Revision History” on page 101.

Table 3-24. Addressing Modes

Addressing Opcode	Updated Value of Ax (when L=0, S is ignored and % is not performed)	Generated Address (when L=0, S is ignored and % is not performed)	Description
0x0	Ax=Ax	Addr=Ax	direct register
0x1	Ax=Sx+(Ax-Sx+Mx)%Lx	Addr=Ax	post-auto-increment
0x2	Ax=Sx+(Ax-Sx-Mx)%Lx	Addr=Ax	post-auto-decrement
0x3	Ax=Sx+(Ax-Sx+Mx)%Lx	Addr=Sx+(Ax-Sx+Mx)%Lx	pre-auto-increment
0x4	Ax=Sx+(Ax-Sx-Mx)%Lx	Addr=Sx+(Ax-Sx-Mx)%Lx	pre-auto-decrement
0x5	Ax=Sx+(Ax-Sx+My)%Lx	Addr=Ax	post-auto-increment, uses two ARFs
0x6	Ax=Sx+(Ax-Sx-My)%Lx	Addr=Ax	post-auto-decrement, uses two ARFs
0x7	Ax=Sx+(Ax-Sx+My)%Lx	Addr=Sx+(Ax-Sx+My)%Lx	pre-auto-increment, using two ARFs
0x8	Ax=Ax	Addr=Sx+(Ax-Sx+Mx)%Lx	base plus index regs
0x9	Ax=Ay+simm16	Addr=Ay+simm16	base reg plus signed immediate, with reg update, using two ARFs
0xA	Ax=Ax+simm16	Addr=Ax+simm16	base reg plus signed immediate, with reg update,
0xB	Ax=Ax	Addr=Ax+simm16	base reg plus signed immediate
0xC	Ax=Ax	Addr=imm16	16-bit immediate

Addressing Opcode	Updated Value of Ax (when L=0, S is ignored and % is not performed)	Generated Address (when L=0, S is ignored and % is not performed)	Description
0xD	Ax=simm16	Addr=simm16	16-bit immediate, plus reg loading
0xE	Ax=Ax	Addr=tmp	direct TMP register
0xF	Ax=Ax	Addr=Ax+tmp	base reg plus TMP register

3.6.3 Arithmetic

The outputs of arithmetic operations are written either in the A field of an ARF register or in the TMP register.

Almost all arithmetic operations generate the following registered flags: Neg(negative), Zero(zero), V(overflow).

- **Z**ero: 1 if res=0; 0 otherwise;
- **N**eg: 1 in signed operation if res[15]=1;
- **V**: overflow result of a signed ADD or signed SUB or signed MUL

Compare instructions generate flag **C,** while instructions involving Lx and Sx generate the boundary flag **B.**

All these flags are registered into the MGCCONDITION register and can be used for branches, predication or pushed into the MGCSTKIQ stack condition register (see Section 3.9.4 ”Condition Stack Manipulation” on page 50).

Table 3-25. Arithmetic Operations

Arithmetic Opcode	Output	Notes	Flag
0x0		NOP	
0x1	Ax=Ay	copy two different A-ARF registers	
0x2	Ax=Ax-TMP	signed	N,Z,V
0x3	Ax=Ax+TMP	signed	N,Z,V
0x4	Ax=Ay-TMP	needs two A-ARF registers, signed	N,Z,V
0x5	Ax=Ay+TMP	needs two A-ARF registers, signed	N,Z,V
0x6	TMP=Ax-TMP	signed	N,Z,V
0x7	TMP=Ax+TMP	signed	N,Z,V
0x8	Ax=Ax+simm16	signed	N,Z,V
0x9	Ax=Ay+simm16	needs two A-ARF registers, signed	N,Z,V
0xA	TMP=AX+simm16	signed	N,Z,V
0xB	TMP=TMP+simm16	signed	N,Z,V
0xC	TMP=Ax*simm16	signed	N,Z,V

Arithmetic Opcode	Output	Notes	Flag
0xD	TMP=TMP*simm16	signed	N,Z,V
0xE	TMP=Ax*My	needs two ARF registers,signed	N,Z,V
0xF	EQ(Ax,TMP)	compare equal Ax==TMP signed/usigned	C
0x10	NEQ(Ax,TMP)	compare not equal Ax!=TMP signed/usigned	C
0x11	GE(Ax,TMP)	compare greater equal Ax>=TMP signed	C
0x12	GT(Ax,TMP)	compare greater Ax > TMP signed	C
0x13	LE(Ax,TMP)	compare less equal Ax<=TMP signed	C
0x14	LT(Ax,TMP)	compare less Ax <TMP signed	C
0x15	Ax=TMP	copy	
0x16	TMP=Ax	copy	
0x17	TMP=Mx	copy	
0x18	TMP=Sx	copy	
0x19	TMP=Lx	copy	
0x1A	Ax=(Ax+Mx)<Lx	generates condition true if A+M<L, signed	N,Z,V,B
0x1B	Ax=Ax-Mx	generates condition true if A-M>0, signed	N,Z,V,B
0x1C	Ax=Ax-TMP	generates condition true if A-TMP>0, signed	N,Z,V,B
0x1D	Ax=(Ax+TMP)<Lx	generates condition true if A+TMP<L, signed	N,Z,V,B
0x1E	Ax=Ax-simm16	generates condition true if A-simm16>0 (signed operation)	N,Z,V,B
0x1F	Ax=simm16		
0x20	TMP=0		
0x21	Ax=ABS(Ax)	Absolute value of Ax	
0x22	TMP=ABS(Ax)	Absolute value of Ax	
0x23	TMP=-Mx	sign inversion	
0x24	TMP=uimm4	use four bits of ARF addressing (default issue) as 4-bit immediate	
0x25	BITTEST(Ax,TMP)	BIT TEST Ax&(1<<TMP)	C

Arithmetic Opcode	Output	Notes	Flag
0x26	Ax=Ax<<TMP	logical shift left of TMP positions	N,Z
0x27	Ax=Ax>>TMP	logical shift right of TMP positions	N,Z
0x28	Ax=Ax>>TMP	arithmetic shift right of TMP positions	N,Z
0x29	Ax=Ax & TMP	bitwise AND	N,Z
0x2A	Ax=Ax | TMP	bitwise OR	N,Z
0x2B	Ax=Ax ^ TMP	bitwise XOR	N,Z
0x2C	Ax=Ax - TMP	unsigned	Z
0x2D	Ax=Ax + TMP	unsigned	Z
0x2E	Ax=Ay-TMP	needs two A-ARF registers, unsigned	Z
0x2F	Ax=Ay+TMP	needs two A-ARF registers, unsigned	Z
0x30	TMP=Ax-TMP	unsigned	Z
0x31	TMP=Ax+TMP	unsigned	Z
0x32	Ax=Ax+imm16	unsigned	Z
0x33	Ax=Ay+imm16	needs two A-ARF registers, unsigned	Z
0x34	TMP=Ax+imm16	unsigned	Z
0x35	TMP=TMP+imm16	unsigned	Z
0x36	TMP=Ax*imm16	unsigned	Z
0x37	TMP=TMP*imm16	unsigned	Z
0x38	TMP=Ax*My	needs two ARF registers, unsigned	Z
0x39	GE(Ax,TMP)	compare greater equal Ax>=TMP unsigned	C
0x3A	GT(Ax,TMP)	compare greater Ax > TMP unsigned	C
0x3B	LE(Ax,TMP)	compare less equal Ax<=TMP unsigned	C
0x3C	LT(Ax,TMP)	compare less Ax <TMP unsigned	C
0x3D	Ax=(Ax+Mx)<Lx	generates condition true if A+M<L, unsigned	Z,B
0x3E	Ax=Ax-Mx	generates condition true if A-M>0, unsigned	Z,B

3.7 AHB Slave Port

The AHB slave is AMBA® rev 2.0 compliant, and it is directly pluggable into an AHB-lite system.

It can give only “OK” or “ERROR” responses to the AMBA AHB transactions, but it never issues a “RETRY” or “SPLIT”.

Errors are revealed in the following cases:

1. wrong address space (address out of space or non existent)
2. data size not 32-bit (i.e. byte and half-word accesses are non allowed)
3. address not 32-bit aligned (i.e. 2 LSBs need to be "00")

In case of error a pulse signal is raised and registered into the MGCEXCEPTION register (Section 5.3.1.1 ”MGCEXCEPTION” on page 87).

The slave decoder receives 2 clocks, one for the AHB side and the other related to the core side.

There must be an integer ratio between the 2 clock frequencies (i.e. 1:2, 3:1, etc. etc.); skew between rising edges of clocks need to be carefully controlled and the relative phase must be stable.

See mAgicV DSP Implementation Manual for details on how to insert clock-tree for the IP inside a SoC.

Slave accesses are not pipelined; each access is decoded and issued to a slave decoder block running at core frequency and when it is completed a new access can be processed. During all the processing time the AHB slave emits a “WAIT” answer.

The slave decodes three DSP addressing regions: program memory, data memory and registers, with different access times.

Table 3-26. Addressing Regions

Resource	Start Address	End Address	Size	Access	Write Latency	Read Latency
PM	0x00600000	0x0061FFFF	128Kbyte	4 x word32	5	6
DM_I	0x00620000	0x0062FFFF	64Kbyte	word32	5	7
DM_F	0x00640000	0x0064FFFF	64Kbyte	word32	5	7
DM_D	0x00660000	0x0067FFFF	128Kbyte	2 x word32	5	7
REGS	0x00680000	0x00681FFF	8Kbyte	word32	5	6
RESERVED	0x00682000	0x006FFFFF	632Kbyte	word32	5	6

3.7.1 Program Memory Accesses

Internal Program Memory has a size of 8K 128-bit words. A word is made of 4 consecutive increasing word addresses, so a complete 128-bit word is allocated in the slave addresses x, x+4, x+8, x+12 where x is a 128-bit aligned address. x stores the lowest part of the program memory line [31:0]; x+12 stores its highest part [127:96].

Table 3-27. PM Alias

AHB Address	AHB Data	Operation	PM[127:0]			
WA	data0[31:0]	write to PM				
WA+4	data1[31:0]					
WA+8	data2[31:0]					
WA+12	data3[31:0]		data3	data2	data1	data0
RA	VLIW[31:0]	read from PM	VLIW[127:0]			
RA+4	VLIW[63:32]					
RA+8	VLIW[95:64]					
RA+12	VLIW[127:96]					

3.7.2 Data Memory Accesses

Internal Data memory has a size of 16Kx40 bit words. It can contain integers or single extended precision floating data. There are three different aliased memory regions to access this memory according to the content of the data itself.

3.7.2.1 DM-I memory alias

This memory alias is used for the 32-bit integer accesses (32 LSBs of the 40-bit word):

- A 32-bit WRITE in the internal data memory writes 32 LSBs of the 40-bit word and it clears the remaining 8 MSBs.
- A 32-bit READ in the internal data memory reads only 32 LSBs of the 40-bit word.

Table 3-28. DM-I Alias

AHB Data	Operation	DM[39:0]	
integer[31:0]	write to DM-I	00000000	integer[31:0]
data40[31:0]	read from DM-I	data40[39:0]	

3.7.2.2 DM-F Memory Alias

This memory alias is used for the 32-bit floating point accesses (32 MSBs of the 40-bit word):

- A 32-bit WRITE in the internal data memory writes 32 MSBs of the 40 bit word and it clears the remaining 8 LSBs.
- A 32-bit READ in the internal data memory reads only the 32 MSBs of the 40-bit word.

Using this alias it is possible to convert a 40-bit floating point into a 32-bit floating point simply by cutting the 8 LSBs of the 32-bit mantissa.

Table 3-29. DM-F Alias

AHB Data	Operation	DM[39:0]	
float32[31:0]	write to DM-F	float32[31:0]	00000000
float40[39:8]	read from DM-F	float40[39:0]	

3.7.2.3 DM-D Memory Alias

This memory alias is used to access the full 40-bit data word, without considering the data type.

In this case a data word is made of two consecutive increasing word addresses, so a complete 40-bit word is stored in addresses: x, x+4, where x is a 64-bit word aligned address. x stores 32 LSBs of memory data [31:0], x+4 stores its 8 MSBs part.

Table 3-30. DM-D Alias

AHB Address	AHB Data	Operation	DM[39:0]	
WA	data0[31:0]	write to DM-D		
WA+4	data1[7:0]		data1[7:0]	data0[31:0]
RA	data40[31:0]	read from DM-D	data40[39:0]	
RA+4	data40[39:32]			

3.8 AHB Master Port

The AHB master is AMBA rev 2.0 compliant, and it is directly pluggable into an AHB system.

It does not implement protection (a default value is issued).

It supports only 32 bit accesses.

It issues only incremental bursts of unspecified length, even in case of single transfers.

It does not emit wait states.

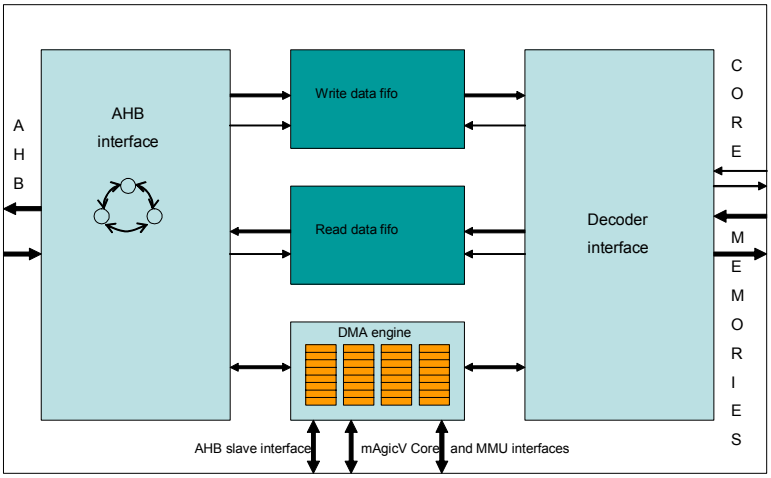
Master grant is always asserted (no arbitration is present, the on-going of AHB transfer modulating HREADY signal is under the addressed slave's responsibility)

Figure 3-5 below indicates the main parts of the AHB master and the DMA engine.

When a DMA channel is ready to start a transfer it turns on the AHB master FSM for data move to/from the DSP core memories.

FIFOs are controlled by the AHB signals on one side and a decoder interface that either transmits or receives data to or from the memories through a master decoder block that is responsible for the correctness check and the data dispatching.

Figure 3-5. AHB Master and DMA Engine



The AHB master is activated by a DMA engine companion (see Section 7.1 ”DMA interface” on page 92).

The AHB master first chooses the next winning DMA channel according to a fixed priority algorithm, then it copies the transfer parameters and starts the AHB cycles as soon as possible.

A programmable length up to 64K words burst is then managed directly by the AHB master: a core engine asks for or delivers data to the internal memories and the AHB bus side manages the AHB protocol. Between the AHB part and the core part there are 2 FIFOs, one for transmitting (10 locations) and the other for receiving data (16 locations).

The two sides can be clocked by different clock frequencies, but with a fixed ratio and with a fixed relative phase (ratios like 2:1, 4:1, 1:3 etc. are allowed). The two different clocked words are separated by the FIFO.

Whenever a transfer write from the internal DSP memories to the AHB bus is running out of data the transfer is interrupted after current data transfer completion and then continued after re-gaining bus grant, without the need for busy issuing that would occupy the bus and waste useful bandwidth.

Whenever a transfer read from the AHB bus to the DSP memories is filling up the FIFO, the transfer is interrupted after current data transfer completion; the master then will transfer the FIFO content to the internal memories and only when the FIFO is empty the transfer will continue after re-gaining bus grant, without the need for busy issuing that would waste bus cycles.

3.9 FLOW Control Block

The FLOW control block performs the following tasks:

- Registers movement
- Programs flow control
- Condition management

The FLOW issue has many formats and the FLOW code can change the default format of other issues.

The basic default format of the FLOW is shown in Table 3-31.

Table 3-31. Default FLOW Issue

vliw[3:2]	vliw[118:113]	vliw[51:50]
FLOW predication	FLOW code	FLOW predication write

- **FLOW predication**

It specifies one of the four predication registers, if the condition of the pointed predication register is “false” (logic ‘0’) the issue will not be executed.

NOTE: Not all FLOW codes are predicated.

- **FLOW code**

It specifies the FLOW operation to be performed.

- **FLOW predication write**

It specifies the predication register when the FLOW code has a value comprised between 0x2E and 0x36

For further details see Section 8. ”Revision History” on page 101.

3.9.1 Register Movements

There are four types of register movements:

1. immediate to register movement
2. register to register movement
3. memory to register
4. register to memory

Movements of type 1 and 2 are performed using explicit FLOW issues, while 3 and 4 are performed using AGU issues.

Register movements use two additional bits of RFA7 field (VLIW[112:104]) to decode the transfer type as specified in Table 3-32 on page 44.

Table 3-32. Transfer Type

RFA7[8]	RFA7[7]	Description
0	0	Movement toward an ARF, flower, dma registers. Destination address RFA7[6:0] using FLOW
0	1	Movement of ARF, flower, dma registers toward memory. Source address RFA7[6:0] using AGU0
1	X	Movement toward a RF register destination address RFA7[7:0], using AGU0

Movements can be vectorial or scalar. ARF, RF registers and data memory allow vector movements. Vectorial ARF movements have multiplicity 4: they move four 16-bit registers. RF vectorial movements have multiplicity 2: they move two 40 bit registers.

NOTE: Only movements with multiplicity 2 can be vectorially predicated (see Section 3.9.5 "Predication" on page 52).

3.9.1.1 *Immediate Movements*

The immediate movements occupy the flow code to specify the immediate operation, the RFA7 field to specify the destination address and zero (when loading the following integer or floating point constants: 0,1,-1, 2, -2), one or more VLIW fields to specify the 16/32/64/80-bit immediate values. Table 3-33 shows the immediate opcodes.

NOTE: Loading large constants reduces parallelism because VLIW fields are used to accommodate constants.

Table 3-33. Immediate FLOW Opcodes

FLOW Opcode	Mnemonic	Description	Latency
0x1	immv	Vector Immediate loading, to specify 64 or 80 bits values depending on the destination register (ARF register 4x16-bit, RF 2x40-bit)	1
0x2	immvc	Vector Conjugate Float loading uses 40-bit immediate for the real part and its negate for the imaginary part	1
0x3	immsi	Scalar Integer immediate loading uses a 32-bit immediate	1
0x4	immsf	Scalar Floating immediate loading uses a 40-bit immediate	1
0x5	immsic	Scalar integer constant loading uses the FLOW predication write field of default issue to specify the integer constant to be loaded see Table 3-36 on page 45	1
0x6	immsfc	Scalar floating constant loading uses the FLOW predication write field of default issue to specify the floating constant to be loaded	1
0x7	imm0v	Vector zero constant loading	1
0x8	imm0s	Scalar zero constant loading	1

Table 3-34. Immediate Issue (imm0,imm0v)

vliw[3:2]	vliw[118:113]	vliw[112:104]
FLOW predication	imm0/imm0v	destination addr

Table 3-35. Immediate Issue (immsic, immsfc)

vliw[3:2]	vliw[118:113]	vliw[112:104]	vliw[51:50]
FLOW predication	immsic/immsfc	destination addr	constant sel

Table 3-36. Constant Selection

Constant Sel	Scalar Float	Scalar Integer
0x0	1.0	1
0x1	-1.0	-1
0x2	2.0	2
0x3	-2.0	-2

Table 3-37. Immediate Issue (immsi, immsf,immvc)

vliw[3:2]	vliw[118:113]	vliw[112:104]	vliw[92:53]
FLOW predication	immsi, immsf,immvc	destination addr	imm

Table 3-38. Immediate Issue (immv)

vliw[3:2]	vliw[118:113]	vliw[112:104]	vliw[92:53]	vliw[43:4]
FLOW predication	immv	destination addr	immh	imml

3.9.1.2 Register to Register Movements

Register movements can be:

- movements that have a RF register as source (require a dedicated FLOW opcode)
- other register to register movements (use RFA5 and RFA7 fields to identify respectively source and destination addresses)

The Table 3-39 resumes the possible movements.

Table 3-39. Register to Register FLOW Opcodes

FLOW Opcode	Mnemonic	Description	Latency
0x9	movv	Vectorial movement	1
0xA	movs	Scalar movement	1

FLOW Opcode	Mnemonic	Description	Latency
0xB	movvf	Vectorial movement (RF as source)	3
0xC	movsf	Scalar movement (RF as source)	3
0xD	movss	Signed extended scalar register movement: 16-bit ARF register into RF register	1

NOTE: Signed extension movements only from the ARF to Register File registers

NOTE: Only RF and ARF registers classes support movement toward registers of the same class (i.e. DMA registers can't be moved into other DMA registers)

NOTE: dedicated MUL opcodes can be used to move the RF registers into registers of the same class.

Table 3-40. Register to Register Issue (movv,movs,movss)

vliw[3:2]	vliw[118:113]	vliw[112:104]	vliw[103:96]
FLOW predication	movv,movs, movss	destination addr	source addr

Table 3-41. Register to Register Issue (movvf,movsf)

vliw[3:2]	vliw[118:113]	vliw[112:104]	vliw[76:69]
FLOW predication	movvf,movsf	destination addr	source addr

3.9.1.3 Register to Memory

Register to memory movements can be:

- movements that have a RF register as source
- movements that have no RF register as source

- **RF register as source:**

RF registers can't be transferred directly to the memory but only through MUL/ADD issue operations. MUL issue results are written in the memory by setting the appropriate write operation on the AGU0 issue. The ADD issue results are written in the memory by setting the appropriate write operations on the AGU1 issue.

For instance, to transfer a RF register into the memory the compiler can generate an integer multiplication with “1” plus an AGU0 addressing issue or an integer addition with “0” plus an AGU1 addressing issue.

NOTE: Normally the compiler is able to write RF registers to the memory during intensive calculation without adding dummy operations.

The write latency is the ADD/MUL operation latency plus one cycle.

No RF register as source:

In this case only the AGU0 issue can be used and the registers are transferred to the memory by setting bit 8 and bit 7 of the RFA7 field according the transfer types see Table 3-32 on page 44.

The write latency is one cycle.

3.9.1.4 *Memory to Register*

Memory to register operations can be activated both on AGU0 or AGU1 issues specifying a read operation.

The AGU0 issues uses also the RFA7 field (see Table 3-32 on page 44) to specify the destination register.

The AGU1 issue performs only operations toward the RF registers and uses the RFA5 field to specify the RF destination address.

The read latency is three cycles long.

3.9.2 Conditions and Status Flags

Condition and status flags are registered in the MGCONDITION register. They are used to perform Condition Stack Manipulation (see Section 3.9.4 ”Condition Stack Manipulation” on page 50), conditional branches and predication.

These flags can be generated by: compare instructions, AGU addressing and arithmetic instructions or by writing Write And Test (WAT) registers.

Conditional branches and calls use condition flags.

3.9.2.1 *MGCCONDITION Register*

This register is reset to zero at every program start (i.e.setting the START bit on the MGCCTRL register).

Table 3-42. MGCCONDITION

15	14	13	12	11	10	9	8
-	-	sirq3	sirq2	rptloop	WATcond	fpucond1	fpucond0
7	6	5	4	3	2	1	0
agu1neg	agu1zero	agu1bnd	agu1cond	agu0neg	agu0zero	agu0bnd	agu0cond

- **agu0cond: AGU0 condition**

condition flag generated by the AGU0 compare instructions or by the AGUFLAG FLOW code that converts one of the following status flags: AGU0BND, AGU0ZERO, AGU0NEG into the condition flag AGU0COND**.**

- **agu0bnd: AGU0 boundary**

status flag generated by the AGU0 circular addressing mode (see Section 3.6.2 ”Addressing” on page 34).

- **agu0zero: AGU0 zero**

status flag generated by the AGU0 arithmetic operations, when the result is zero.

- **agu0neg: AGU0 negative**

status flag generated by the AGU0 arithmetic signed operations, when the result is negative.

- **agu1cond: AGU1 condition**

condition flag generated by the AGU1 compare instructions or by the *aguflag* FLOW code that converts one of the following status flags: AGU1BND, AGU1ZERO, AGU1NEG into the condition flag AGU1COND**.**

- **agu1bnd: AGU1 boundary**

status flag generated by the AGU1 circular addressing.

- **agu1zero: AGU1 zero**

status flag generated by the AGU1 arithmetic operations, when the result is zero.

- **agu1neg: AGU1 negative**

status flag generated by the AGU1 arithmetic signed operations, when the result is negative.

- **fpucond0: FPU left condition**

condition left (or real) flag generated by the ADD issue compare instructions. In scalar and complex compare left and right conditions are equal. In a vectorial compare left and right conditions may be different.

- **fpucond1: FPU right condition**

condition right (or imaginary) flag generated by the ADD issue compare instructions. In scalar and complex compare left and right conditions are equal. In a vectorial compare left and right conditions may be different.

- **WATcond: Write And Test condition**

condition flag generated by writing WAT registers. The written value is the mask of bits to test in the register. WATcond='1' if the bits tested are '1'.

- **rptloop: repeat loop**

condition flag generated if there is an active hardware loop.

RPTLOOP='1', when MGCLOOPCNT register is > 0.

RPTLOOP='0', when MGCLOOPCNT == '0'.

- **sirq2**

status flag generated by the FLOW SIRQ2 code.

- **sirq3**

status flag generated by the FLOW SIRQ3 code.

3.9.3 Conditioned and Unconditioned Jumps

mAgicV supports PC-relative and Indirect Register Jumps:

- PC-relative Jumps use a 16-bit signed immediate offset, allowing forward and backward branches of up to 32K locations. The size of the physical on-chip program memory is 8K locations. When the PMU is enabled the total addressable space for program memory is extended to 64K locations. Starting from any 16-bit PC value, the addition or subtraction of a 32K value allows to span the full addressable 16-bit space. For this purpose, no exception overflow exception is generated by relative address computation.
- Indirect Register Jumps use the 16-bit MGCBRANCH register. This register must contain an absolute Program Memory address.

Jumps can be either unconditioned or conditioned. The branch instruction specifies the condition source to be used, selecting among five possible sources (AGU0COND, AGU1COND, FPUCOND0, WATCOND, TOP0) using an 8-bit JUMP TYPE field (see Section 3.9.3.1 "Selec-

tion of Jump Condition" on page 49). The value of the condition must have been previously stored in the MCGCONDITION register.

The pair of CALL and RETurn FLOW instructions provide a fast jump and link mechanism to enter into and exit from leaf functions. The FLOW code (CALL) preserves the address of the instruction after the branch in the MGCSTK register (link register). The RET FLOW code replaces the program counter MGCPC with the MGCSTK register content.

Branches and calls can be predicated (see Section 3.9.5 "Predication" on page 52).

NOTE: To return from an Interrupt Service Routine the RETI code must be used.

Table 3-43. Branch Operations

FLOW Opcode	Mnemonic	Description
0x26	br	PC-Relative Jump (Immediate Offset)
0x27	br_reg	Indirect Register Jump to MGCBRANCH
0x28	call	PC-Relative CALL (Immediate Offset), saves return address to MGCSTK
0x29	call_reg	Indirect Reg CALL to MGCBRANCH, return address in MGCSTK
0x2C	ret	RETURN from a CALL, absolute jump to MGCSTK register
0x2D	reti	RETUN from an INTERRUPT service routine

Table 3-44. Opcodes for PC-relative Jumps and Calls

vliw[3:2]	vliw[118:113]	vliw[84:77]	vliw[76:61]
FLOW predication	br,call	jump type	offset

Table 3-45. Opcodes for Indirect Registers Jumps and Calls

vliw[3:2]	vliw[118:113]	vliw[84:77]
FLOW predication	br_reg,call_reg	jump type

Table 3-46. Opcodes for RET and RETI

vliw[3:2]	vliw[118:113]
FLOW predication	**ret,reti**

3.9.3.1 *Selection of Jump Condition*

Jumps have an 8-bit type field to specify the type of condition used (see Section 3.9.2 "Conditions and Status Flags" on page 47). For unconditioned Jumps this field must be set to 0.

NOTE: For conditioned Jumps only one bit must be set.

Table 3-47. Jump Condition Selector

7	6	5	4	3	2	1	0
reserved	reserved	reserved	agu1cond	agu0cond	top0	fpucond0	WATcond

- **WATcond**

WATcond=’1’, jump conditioned by WATcond.

WATcond=’0’, jump not conditioned by this flag.

- **fpucond0**

fpucond0=’1’, jump conditioned by FPU compare fpucondflag0 (vector conditions are not used).

fpucond0=’0’, jump not conditioned by this flag.

- **top0**

top0=’1’, jump conditioned by the top of the condition stack0.

top0=’0’, jump not conditioned by this flag.

- **agucond0**

agucond0=’1’, jump conditioned by aguflag0.

agucond0=’0’, jump not conditioned by this flag.

- **agucond1**

agucond1=’1’, jump conditioned by aguflag1.

agucond1=’0’, jump not conditioned by this flag.

3.9.4 Condition Stack Manipulation

In addition to the mechanism of conditions direct generation performed by AGU0, AGU1, and Operators Block (directly stored in MGCCONDITION), two 16-bit condition stack registers stored in the 32-bit MGCSTKIQ register can be used to push and manipulate condition flags previously generated.

Table 3-48 Lists the FLOW codes that operate on the stack registers.

Table 3-48. Condition Stack Operations

FLOW Opcode	Mnemonic	Description
0xE	push_op	Pushes vector conditions generated by ADD
0xF	push_agu0	Pushes scalar condition generated by AGU0 in both STACK0 and STACK1
0x10	push_agu1	Pushes scalar condition generated by AGU1 in both STACK0 and STACK1
0x11	pop	Pops out conditions from both STACK0 and STACK1
0x12	true	Pushes true ‘1’ on both STACK0 and STACK1
0x13	false	Pushes false ‘0’ on both STACK0 and STACK1
0x14	fsnot	inverts the TOPs of the STACKs
0x15	fsand	Makes the logic AND between the first and second elements of the STACKs and replaces TOPs with the results
0x16	fsor	Makes the logic OR between the first and second elements of the STACKs and replaces TOPs with the results
0x17	fsxor	Makes the logic XOR between the first and second elements of the STACKs and replaces TOPs with the results

FLOW Opcode	Mnemonic	Description
0x18	repush	Pushes TOPs into the STACKs
0x19	fsswap	Swaps the first and the second value of the STACKs
0x1A	fsand2	Makes the logic AND between the first and second value of the STACKs and pushes the results in TOPs
0x1B	fsi2q	pushes TOP0 condition on the STACKs
0x1C	fsq2i	pushes TOP1 condition on the STACKs

The default FLOW issue shown in Table 3-31 on page 43 is used for condition stack manipulation.

3.9.4.1 MGCSTKIQ

The stack registers are accessed as the LSB and MSB parts of the MGCSTKIQ 32-bit register and they accommodate respectively the right and the left part of a vectorial flag.

Table 3-49. MGCSTKIQ 32-bit Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
STACK1															top1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STACK0															top0

STACK0 field stores the right condition vector flag, while STACK1 field stores left condition vector flag. Stack0[0] is named TOP0, while STACK1[16] is named TOP1.

While a push operation on an AGU condition flag (PUSH_AGU0, PUSH_AGU1) copies the same flag value on the top of both the I and Q stacks, the PUSH_OP code, which receives the conditions from the Operators Block, can push different values on the top of the I and Q stacks when executing a Vectorial compare.

The TOP0 can be used in conditional branches, while both TOPs can be used in some ADD issue operations (SWAP,VSWAP,SEL).

3.9.5 Predication

mAgicV has 4 predication registers, packed into a single 16-bit MGCPRED register.

Each VLIW issue (ADD, MUL, AGU0, AGU1, FLOW) is associated with individual fields of 2 bits in the program word. Those 2-bit fields specify the number of the predication register to be used to predicate the execution of the associated issue.

If the value of the predication register is false (logic ‘0’) the operation of the associated issue is not executed.

NOTE: predication is always enabled. This means that every operation which must be actually executed must be associated to a predication register set to a TRUE value. At reset, the predication register #0 is set to TRUE and the operations referring to it will be executed.

The value of the 4 predication registers can be set by the predication operations specified by Table 3-50. The VLIW coding for these operations is described by Table 3-51 and Table 3-52 below.

NOTE: If the HW support for the software pipeline is enabled, all issues are forced to use the predication register #0.

Table 3-50. Opcodes for setting the Values of the Predication Registers.

FLOW Opcode	Mnemonic	Description
0x2E	clrp	clears predication register
0x2F	setp	sets true predication register, OPERATION NOT PREDICATED
0x30	setpv	sets predication register to immediate, OPERATION NOT PREDICATED
0x31	settop	sets predication register with the TOPs of condition stacks
0x32	setop	sets predication register with the fpucond0 and fpucond1 condition flags
0x33	setagu0	sets predication register with agu0cond
0x34	setagu1	sets predication register with agu1cond
0x35	setnotp	sets predication register <reg+1> equal to the not of <reg>
0x36	setnot	inverts condition of predication register

Table 3-51. VLIW Coding for the SETPV Operation

vliw[118:113]	vliw[51:50]	vliw[3:2]
setpv	destination addr	2-bit imm

Table 3-52. VLIW Coding for other Predication Setting Instructions

vliw[3:2]	vliw[118:113]	vliw[51:50]
FLOW predication	clrp,settop,setop,setagu0,setagu1,setnot	destination addr

3.9.5.1 MGCPRED

The MGCPRED register packs the 4 predication registers plus a PREMODE field which specifies a modality of behavior for the operations setting the values of the predication registers listed in Table 3-50 above.

Each predication register is 3-bit wide. A predication register with all the 3 bits set to 1 enables the execution of any issue predicated by it (TRUE value). *C0* and *C1* fields accommodate vector condition flags and they are used to enable/disable vector operations producing vectorial results to be stored in the RF and in the memory. All scalar operations must have C0=C1, otherwise undefined results may be generated. The third bit G0 is updated as shown in Table 3-50 on page 52 depending on the PREDMODE field as shown in Table 3-55 below.

Table 3-53. MGCPRED Register

15	14	13	12	11	10	9	8
-	predmode			predreg3			predreg2
7	6	5	4	3	2	1	0
predreg2		predreg1			predreg0		

Table 3-54. Predication Register

2	1	0
G	C1	C0

- **C0: condition flag0**

Right vector condition.

- **C1: condition flag1**

Left vector condition.

- **G: global predication flag**

Used to predicate:

- AGU addressing and arithmetic operations
- data movement towards the ARF, DMA and FLOW registers
- flow control operations.

Table 3-55. Global Predication Flag Update

Predmode	Description
0x0	G=C0, default
0x1	G=(C0 OR C1)
0x2	G=(C0 AND C1)
0x3	G=G, not update

3.9.6 Hardware Loop Support

mAgicV has hardware loop support to optimize loop overhead. Repeats can be combined with hardware software-pipeline to enhance code density and register reuse.

Table 3-56. Repeat Operations MGCLOOPCNT

FLOW Opcode	Mnemonic	Description
0x37	repb	repeat block
0x39	reserved	reserved

REPB opcode requires the initialization of the 32-bit loop counter MGCLOOPCNT register with the length of the loop. Repeats always execute at least one loop-cycle (their logic is similar to a do-loop C-statement) the counter is decremented by one at the end of the loop.

3.9.6.1 MGCREPEAT

REPB opcode loads the start and end addresses of the block to be repeated into the 32-bit MGCREPEAT register. The start of the block PMA is set to the current PC + 3, while the end of the block PMA is set to current PC + block len -1 + 3.

Table 3-57. MGCREPEAT Register

31	30	29	28	27	26	25	24
START OF BLOCK PMA							
23	22	21	20	19	18	17	16
START OF BLOCK PMA							
15	14	13	12	11	10	9	8
END OF BLOCK PMA							
7	6	5	4	3	2	1	0
END OF BLOCK PMA							

3.9.6.2 Repeat Block

The repeat block is used mainly to support hardware loops, with no jump overhead. The repeating block starts at PMA+3, where PMA is the location of the repeat instruction and it needs a BLOCK LENGTH field to specify the length of the block to be repeated.

Table 3-58. Repeat Block Issue (repb)

vliw[3:2]	vliw[118:113]	vliw[76:61]
FLOW predication	repb	block length

NOTE: If the block is compressed, the repeat block parameter must change to reflect the final length (after compression) of the block. To maintain logic coherency the compressor can't compress the code between the repeat block and the start of the repeating block.

3.9.7 Software Pipeline HW Support

The hardware support to the software pipeline allows eliminating prologues and epilogues of software-pipelined loops, saving program memory space. The drawback can be an increased execution time, due to more loop-cycles needed to complete the original loop.

In software pipeline mode, each computational issue (AGU0, AGU1, MUL, ADD) has an associated number (named STAGE) that represents the loop iteration K from which the issue will be enabled. The issue will remain enabled up to K=MGCLOOPLEN + STAGE. The number of iterations of the loop will be MGCLOOPLEN + MAXSTAGE, where MAXSTAGE is the maximum stage declared inside a software pipelined loop, and MGCLOOPLEN is the register that holds the loop length.

The SWPIPE FLOW code is used to enable or to disable a software pipelined loop. This microinstruction needs two parameters: MAXSTAGE a 3-bit field and ON/OFF (ON='1') flag (see Table 3-59 below). MAXSTAGE is the maximum stage declared inside a software pipelined loop. ON/OFF bit flags specify if we are entering or exiting a software pipelined loop.

Table 3-59. SW Pipeline Issue

vliw[118:113]	vliw[51]	vliw[50]	vliw[3:2]
swpipe=0x3B	ON/OFF	MAXSTAGE	

Table 3-60. STAGE Fields in SW Pipeline Mode

vliw[53:51]	vliw[50:49]	vliw[5:3]	vliw[2:0]
STAGE MUL	STAGE AGU1	STAGE ADD	STAGE AGU0

NOTE: If HW support for software pipeline is enabled, all predication addresses point to zero. This means that only one level of predication is supported (predreg0 register).

NOTE: Software pipeline is disabled when entering in interrupt routine. It is not possible to use it inside an Interrupt Service Routine.

3.9.7.1 MGCLOOPLEN

This 32-bit register is used only within HW software pipelined loops. It takes into account the number of iterations that the loop must do, while the MGCLOOPCNT is the loop counter. By writing this register the MGCLOOPCNT will be written as well (to save a write operation: at the beginning of the loop MGCLOOPCNT=MGCLOOPLEN).

NOTE: The command swpipe (ON) will add automatically MAXSTAGE to MGCLOOPLEN and will copy the result into MGCLOOPCNT and MGCLOOPLEN registers.

NOTE: The command swpipe (OFF) will subtract MAXSTAGE to MGCLOOPLEN and will copy the result into MGCLOOPCNT and MGCLOOPLEN registers.

3.10 Program Management Unit

The mAgicV architecture specifies a 16-bit virtual program memory space (64K 128-bit words). This virtual space is mapped into a physical 13-bit physical program memory space by a PMU.

The pm word (program word) is composed of 128 bits, the PMU maps 64K pm words of external program memory in 8K pm words of internal memory. The external program memory space is divided into 64 pages of 1K pm words. Each 1K pm word page is divided into sixteen chunks, each one composed of 64 pm words, as described in Table 3-61 below.

Table 3-61. Virtual Address

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
virtual page						chunk				offset					

An efficient page replacement algorithm is realized in hardware to avoid software overhead.

It is possible to instruct the PMU to fix a set of physical pages, excluding them from the replacement algorithm. Each of the 8 physical pages has an associated PMUMAPPEDVIRT register used to specify the virtual page (each one described by one of the 64 PMUVIRT registers) and the chunks already loaded on the internal memory.

Two types of faults can be generated at every cycle:

- Page fault
- Chunk fault

A page fault is generated when the virtual page isn't physically mapped into one of the eight internal physical pages. In this case the PMU finds a physical page to host the new virtual page. If all physical pages are allocated, the PMU will replace the last recently used page with the new one, using an hardware replacement algorithm (see Section 3.10.2 ”Replacement Algorithm” on page 60) which operates on the PMU register described by Table 3-62 below.

3.10.1 PMU Registers

PMU registers can be accessed only through the AHB slave accesses. Each PMU register is described later in a specific section. Table 3-62 contains the list of PMU registers.

Table 3-62. PMU Registers

AHB Offset	Name	Type	Reset Value
0x200- 0x2FC	PMUVIRT	RW	NA
0x300- 0x31C	PMUMAPPEDVIRT	RW	0x00000000
0x320	PMUFAULTADD	RW	NA
0x324	PMUEXTADD	RW	0x200
0x328	PMUCTL	WO	NA
0x328	PMUSTAT	RO	0x00000000
0x32C	PMUMISSCNT	RW	0x00000000
0x330	PMUPHYSPNT	RW	0x7

3.10.1.1 *PMUVIRT*

The PMU has 64 6-bit registers (PMUVIRT), one for each virtual page. PMUVIRT descriptors are used to translate virtual pages into physical pages to detect page miss and to perform page-replacement algorithm. Table 3-63 describes the format of PMUVIRT registers.

Table 3-63. Virtual Page Descriptor Register

5	4	3	2	1	0
phys page			ref	fixed	map

- **map**

MAP='1', the virtual address corresponding to the virtual page descriptor register is currently mapped to PHYS PAGE**.**

MAP='0', the virtual address isn't mapped.

- **fixed**

FIXED='1', the virtual page is fixed and can't be replaced.

FIXED='0', the virtual page can be replaced.

- **ref: reference (Read Only)**

REF='1', the virtual page has been recently accessed. See replacement algorithm.

REF='0', the virtual page has not been recently accessed.

- **phys page: physical page**

It's the 3-bit value identifying one of the eight PMUMAPPEDVIRT describing the physical page that hosts the virtual page.

3.10.1.2 *PMUMAPPEDVIRT*

The PMU has eight 32-bit descriptor registers, (PMUMAPPEVIRT0 - PMUMAPPEDVIRT7). Each PMUMAPPEVIRTx is associated to one of the 8 available on-chip physical pages.

Each PMUMAPPEVIRTx register holds a pointer to the virtual page which is mapped on the physical page #x and a bitmap that specifies the chunks loaded. Table 3-64 describes the format of PMUMAPPEDVIRT registers.

Table 3-64. PMUMAPPEDVIRT Register

31	30	29	28	27	26	25	24
chunk15	chunk14	chunk13	chunk12	chunk11	chunk10	chunk9	chunk8
23	22	21	20	19	18	17	16
chunk7	chunk6	chunk5	chunk4	chunk3	chunk2	chunk1	chunk0
15	14	13	12	11	10	9	8
00000000							
7	6	5	4	3	2	1	0
0	valid	mapped virtual page					

- **mapped virtual page**

Describes the virtual page number mapped to the physical page associated with the PMU-MAPPEDVIRTx register (0<=x<7)

- **valid**

VALID='0', the associated physical page doesn't host any virtual page.

VALID='1', the associated physical page maps the specified virtual page.

- **chunk15-chunk0**

it's a bitmap that specifies what chunks are loaded into the associated physical page

3.10.1.3 *PMUFAULT: PMU Fault Address*

This 16-bit register contains the virtual address that generated the last miss.

3.10.1.4 *PMUEXTADD*

This 12-bit register contains the MSB of the address generated by the PMU to access the program residing in the external memory.

This register allows handling of several program images on the 4-GB of external memory space.

The external address generated by the PMU is shown in Table 3-65.

Table 3-65. External Address

31	30	29	28	27	26	25	24
pmuextaddreg							
23	22	21	20	19	18	17	16
pmuextaddreg				pmufaultadd[15:6]			
15	14	13	12	11	10	9	8
pmufaultadd[15:6]						0	
7	6	5	4	3	2	1	0
0x00							

3.10.1.5 *PMUCTL*

It is the PMU control register (Write Only access).

Table 3-66. PMUCTL

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	-	-	-	prefetch unmask	prefetch mask	softreset

- **softreset: software reset**

SOFTRESET='1', resets the PMU state machine.

SOFTRESET='0', No effect.

NOTE: Only the PMUSTAT register is reset to the value shown in Table 3-62 on page 56. Other registers are left unchanged.

- **prefetch mask**

PREFETCH MASK='1', prefetch instructions are not performed.

PREFETCH MASK='0', No effect.

- **prefetch unmask**

PREFETCH UNMASK = '1', allows execution of prefetch instructions.

PREFETCH UNMASK = '0', No effect.

3.10.1.6 *PMUSTAT: PMU Status Register*

It is the PMU status register (Read Only Access).

Table 3-67. PMUSTAT

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	clkdis	fix page	-	prefetch chunk
7	6	5	4	3	2	1	0
prefetch page	chunk fault	Page fault	double fault	mask prefetch	waitmst	waitendtx	busy

- **busy**

BUSY = '1' , PMU state machine is working.

BUSY = '0', PMU state machine is idle.

- **waitendtx: wait end transfer**

WAITENDTX = '1', PMU is waiting for the completion of a DMA request.

WAITENDTX = '0', PMU is not waiting for a DMA transfer.

- **waitmst: wait master ready**

WAITMST = '1', PMU is waiting for the AHB master to be ready.

WAITMST = '0', PMU is not waiting for the AHB master.

- **mask prefetch**

MASKPREFETCH = '1', prefetched are masked.

MASKPREFETCH = '0', prefetches are not masked.

- **double fault**

DOUBLEFAULT = '1', a double miss has been detected.

DOUBLEFAULT = '0', no double fault.

- **page fault**

PAGEFAULT = '1', a page miss has been detected.

PAGEFAULT = ’0’, no page fault.

- **chunk fault**

CHUNKFAULT = ‘1’, a chunk miss has been detected (no need of page replacing).

CHUNKFAULT = ‘0’, no chunk fault.

- **prefetch page**

PREFETCHPAGE = ‘1’, a page must be prefetched.

PREFETCHPAGE = ‘0’, no prefetch page.

- **prefetch chunck**

PREFETCHCHUNK = ‘1’, a chunk must be prefetched.

PREFETCHCHUNK = ‘0’, no prefetch chunk.

- **fix page**

FIXPAGE = ‘1’, a page is temporary fixed by the PMU during a double fault.

FIXPAGE = ‘0’, no fixing needed.

- **clkdis: clock disable**

CLKDIS = ‘1’, clock is disabled to allow transparent code fetching.

CLKDIS = ‘0’, no clock disable.

3.10.1.7 PMUMISSCNT

This is a 32-bit register that counts the number of misses done by the program. It can be used to optimize the code by minimizing the misses (through page fixing and/or prefetches).

3.10.1.8 PMUPHYSPNT

This is a 3-bit register used to realize the replacement algorithm. It points to the physical page descriptor (**PMUMAPPEDVIRT**) that represents the virtual page likely to be replaced.

3.10.2 Replacement Algorithm

The PMU register PMUPHYSPNT points to the physical page next to the last loaded by a page miss.

The PMU will search starting from PMUPHYSPNT the first virtual page with “FIXED”=0 and “REF”=0. During the search, the PMU resets the “REF**”** bit of every virtual page checked. If no page is found (because each virtual page has the “REF”=1) the algorithm is repeated. If no page is found the exception of MISSPAGEFREE will be raised. This exception could be raised if there are at least seven pages fixed.

A chunk fault is generated when the virtual page is already mapped into a physical page, but the chunk isn't present in the physical page yet. In this case the PMU loads 64 pm words corresponding to the missing chunk, without replacing any page.

The PMU supports prefetching instructions in the FLOW issue to improve program page hits.

3.10.3 Translation from Virtual Address to Physical Address

The virtual page field (see Table 3-61 on page 56) is used to select one of the 64 PMUVIRT registers. If the virtual page descriptor is “MAP”=1 the 13-bit PMA (Program Memory Address) is

built by joining the “PHYS PAGE” field with the 10 LSBs of virtual address else a miss will be generated (see Table 3-63 on page 57).

Table 3-68. PMA

12	11	10	9	8	7	6	5	4	3	2	1	0
PHYS_PAGE[2:0]			virtual address[9:0]									

3.10.4 PMU Operations

The program flow can drive PMU to prefetch, fix, unfix virtual pages. Through these commands it is possible to dynamically optimize a program by reducing its misses.

Table 3-69. PMU Operations

FLOW Opcode	Mnemonic	Description
0x2B	prefetch_reg	prefetch the page pointed by MGCBRANCH register
0x3A	prefetch_off	prefetch the page corresponding to the PC relative offset
0x3D	pmucmd	fix, unfix virtual page specified by MGCBRANCH

Table 3-70. prefetch_reg

vliw[3:2]	vliw[118:113]
FLOW predication	prefetch_reg

Table 3-71. prefetch_off

vliw[3:2]	vliw[118:113]	vliw[76:61]
FLOW predication	prefetch_off	offset

Table 3-72. pmucmd

vliw[3:2]	vliw[118:113]	vliw[51:50]
FLOW predication	pmucmd	fix=1, unfix=2

3.10.4.1 Prefetching

By prefetching a page the PMU will check if the page is already mapped, otherwise it will start a DMA to load the requested page in parallel to the program execution.

3.10.4.2 Fixing and Unfixing

By issuing this commands it is possible to fix and unfix a virtual page.

4. Programming Model

This chapter describes the programming model and the set of registers accessible to the user.

4.1 Data Formats

mAgicV supports the data types shown in Table 4-1.

Table 4-1. Data Types

Type	Data Width	Description
half-word	16-bits	used for signed/unsigned 16-bit integers
word	32-bits	used for signed 32-bit integers
	32-bits	used either for external memory storage of 32-bit standard precision floating-point data or for 32-bit data communication through AHB AMBA interface
extended-word	40-bits	used for internal floating point computation (extended IEEE754 format)
	64-bits	used either for external memory storage of extended precision floating-point data or for extended precision data communication through AHB AMBA interface

4.1.1 16-bit Integer Format

The 16-bit unsigned integers represent numbers in the range between 0 and 65535.

The twos complement format is used for 16-bit signed integers, in the range between -2^{15} and $2^{15}-1$.

4.1.2 32-bit Signed Integer Format

mAgicV uses the twos complement for 32-bit signed integers, in the range between -2^{31} and $2^{31}-1$.

No HW support for 32-bit unsigned integers which must be emulated by software.

mAgicV supports vector operations on 32 bit signed integers.

4.1.3 Standard Precision 32-bit Floating-Point Format

Standard 32-bit floating point format is used for data storage in external memory and can be used for data exchange through AHB AMBA interface. All mAgicV internal floating point computations are performed on 40-bit floating point extended precision data.

4.1.4 Extended Precision 40-bit Floating-Point Format

mAgicV supports 40-bit floating point format, as an extension of the IEEE Standard 754 floating point, having 1-bit sign, 8-bit exponent and 31-bit mantissa instead of 23-bit mantissa.

Table 4-2. 40-bit Floating Point Format

39	38 … 32	30 … 0
sign	exponent	mantissa

As in the standard IEEE 754 representation of the floating point numbers, the most significant bit of the mantissa, also known as the “hidden bit”, is not represented.

There are however differences from the IEEE 754 standard:

- No traps are implemented. When an exception is detected, a status flag is set (see Section 3.4.9 ”Operator Status Flags” on page 25).
- De-normalized numbers are not implemented.
- Rounding mode is round to nearest (the number is rounded to the nearest representable value; this mode has the smallest errors associated with it because statistically rounding up and rounding down occur with the same frequency).

mAgicV supports vector operations on 40 bit floating point numbers.

4.2 Data Organization

In the memory and in the RF the data is stored as 40-bit quantities (extended-word). Integers quantities have the 8 MSB padded with zero. Vector accesses occupy two consecutive addresses (a vector memory access with odd addresses generates exception). The “right” part of a 2-D vector quantity is contained at a lower address.

The tables below show the representation of the data types listed in Table 4-1 on page 62.

Table 4-3. half-word unsigned

39 ... 16	15 ... 0
00000000000000000000000	halfword

Table 4-4. half-word signed extended

39 ... 32	31 ... 16	15 ... 0
00000000	1111111111111111	halfword

Table 4-5. word

39 ... 32	31 ... 0
00000000	word

A full 64-bit ARF register (see Table 3-22) is packed in the memory and in the RF by using two consecutive words.

Table 4-6. even word

39 ... 32	31	0
00000000	A field	M field

Table 4-7. odd word

39 ... 32	31	0
00000000	S field	L field

4.2.1 Register Classes

mAgicV registers are divided into three main classes having well defined data types and functionalities:

1. FLOW registers: it groups registers used to control and monitor program flow, to manage interrupts and DMA. No arithmetic operations are defined for these registers and they can't be copied directly in other FLOW registers.

2. ARF registers: it groups registers involved in address calculation and 16-bit signed/unsigned arithmetic.

3. Register File registers: it groups registers involved in 40-bit floating point arithmetic and in signed integer 32-bit arithmetic. This class of registers can't be accessed in write mode by an external AHB master.

4.3 DSP States

mAgicV supports two main modes: run and debug. When the processor is in run mode three other states are possible: step, sleep, interrupt.

Mode changes can be either caused by software control (i.e. FLOW opcodes or accesses from external masters through the AHB slave interfaces, both writing on the MGCCTRL register), or activated by external interrupts or exception processing. A mode can be interrupted by a higher priority mode, but never by a lower priority. An AHB external master can change any mAgicV state. Nested interrupts are not supported.

Table 4-8. DSP States

Priority	State	Description
4	debug	All core pipelines are frozen, it's safe to access internal memories and registers through the AHB slave interface. Pending DMA are completed.
3	sleep	All pipelines are frozen, but the DSP is running waiting for some events. This mode is used mainly in combination with write/read DMA operations to wait the end of the transfer (EOT).
2	step	Causes one cycle of run state followed by the debug state
1	interrupt	mAgicV executing an ISR. All pipelines are running, interrupts arriving on other lines are stored and will be served after execution of the RETI instruction. Hardware support for SW pipeline is disabled.
0	run	All pipelines are running. Interrupt will be served on execution of branches.

4.3.1 Debug

In debug mode, all DSP pipelines are frozen, only the DMA and the PMU are active, completing all pending I/O operations. An AHB master controller can access mAgicV internal memories either in write or in read, through the AHB slave port, without interfering with core pipelines. All mAgicV internal registers are accessible from external masters in read/write (see rev A Note below).

NOTE rev.A specific: with the exception of the RF registers which can be read only.

4.3.2 Sleep

See Section 5.2 ”Sleep and Wakeup” on page 84.

4.3.3 Step

See Section 6.2.4 ”Step Mode Support” on page 91.

4.3.4 Interrupt

See Section 5.1 ”Interrupt Handling” on page 78.

4.3.5 Run

In run mode, mAgicV is executing a program. When the HW support for software pipeline is enabled an interrupt service is deferred. An AHB master controller can access both mAgicV internal memories and mAgicV registers. In this mode any access to an internal resource should be protected by using mutexes (see Section 4.5.1 ”Mutex Support” on page 75). As in debug mode, an external AHB master can access the RF registers through the AHB slave only in read mode (see rev A Note above on page 71).

4.3.6 Transitions between States

The Table 4-9 illustrates how different states and modes are both entered and exited.

Table 4-9. Transition between States

State	Entry Cause	Exit Cause
debug	• Induced by FLOW opcodes: – non-masked exceptions, – HALT code. • Caused by external AHB masters: – by writing STOP or GBREAKON MGCCTRL registers bits, – breakpoints, – watchpoints, – step mode. • induced by Cross/triggering line: – external debug request (Cross Trigger),	• induced by external AHB master: – Clearing of exceptions or break conditions. – By writing START or CONTINUE or GBREAKOFF MGCCTRL registers bits.
sleep	• Induced by FLOW opcodes: – SLEEP FLOW code • Caused by external AHB masters: – by writing SLEEPON bit in MGCCTRL register	• induced by the events reported in the MGCWAKECTRL reg: – Interrupts – DMA end of transfers. • caused by external AHB masters: – by writing SLEEPOFF bit in MGCCTRL register.

State	Entry Cause	Exit Cause
step	• Caused by external AHB masters: – by writing STEPON bit in MGCCTRL register.	• Caused by external AHB masters: – by writing STEPOFF bit in MGCCTRL register.
interrupt	• Induced by non-masked interrupt on internal/external lines, which are served on the first occurrence of a BR_REG,BR or WATCHINT FLOW opcodes	• RETI FLOW opcode
run	• Any of the Exit Causes of higher priority states, which are listed in the right column above.	• Any of the Entry Causes of higher priority states, which are listed in the left column above.

4.3.7 Control and Status Registers

mAgicV control registers are write only. They allow to access the single bit of the register by setting ‘1’ in the respective position of the write mask, ‘0’ has no effect. In this way it is possible to save several cycles of bit manipulation. Two distinct control bits are needed to either set or clear a status/mode bit, one to set it and the other to clear it. Normally each control register has an associated status register that reflects the changes made by writing the control register. They often have the same address because they are selected by the operation type (write or read).

Status registers may support the Write And Test (WAT) optimization (see Section 3.9.3 ”Conditioned and Unconditioned Jumps” on page 48).

The most important mAgicV control register is the MGCCTRL which allows to start, stop and debug mAgicV programs.

4.3.7.1 MGCCTRL

Table 4-10. MGCCTRL Register

31	30	29	28	27	26	25	24
swreset	nofataloff	nofatalon	allfataloff	allfatalon	lockoff	lockon	tickoff
23	22	21	20	19	18	17	16
tickon	pmuoff	pmuon	pmchkoff	pmchkon	intovfoff	intovfon	breakoff
15	14	13	12	11	10	9	8
breakon	watchoff	watchon	decopff	decopon	triggoff	triggon	stepoff
7	6	5	4	3	2	1	0
stepon	gbreakoff	gbreakon	start	stop	continue	sleepoff	sleepon

- **sleepon**

SLEEPON=’1’, it forces mAgicV into sleep mode.

SLEEPON=’0’, no effect.

- **sleepoff**

SLEEPOFF=’1’, it forces mAgicV to exit from sleep mode.

SLEEPOFF=’0’, no effect.

- **continue**

CONTINUE=’1’, it puts mAgicV in run mode from the debug state (by continuing a previous interrupted program, i.e. step mode).

CONTINUE=’0’, no effect.

- **stop**

STOP=’1’, it puts mAgicV in debug mode by terminating a running program.

STOP=’0’, no effect.

- **start**

START=’1’, it starts a new program and puts mAgicV in run mode from the debug state.

START=’0’, no effect.

- **gbreakon**

GBREAKON=’1’, it puts mAgicV in debug mode by setting a “software break point”.

GBREAKON=’0’, no effect.

- **gbreakoff**

GBREAKOFF=’1’, it clears the “software break point” and it returns to run mode if no other break sources are active.

GBREAKOFF=’0’, no effect.

- **stepon**

STEPON=’1’, it enables the step mode. A break request (STEPREQ bit of MGCSTAT) will be generated at each cycle forcing mAgicV into debug mode.

STEPON=’0’, no effect.

- **stepoff**

STEPOFF=’1’, it disables the step mode.

STEPOFF=’0’, no effect.

- **triggon**

TRIGGON=’1’, it enables Cross Triggering (see Section 4.5.3 ”Programmable Output Lines” on page 77).

TRIGGON=’0’, no effect.

- **triggoff**

TRIGGOFF=’1’, disables Cross Triggering.

TRIGGOFF=’0’, no effect

- **decopon**

DECOMPON=’1’, it enables code decompressor. The internal Program Memory must contain a valid Compressed Program VLIW.

DECOMPON=’0’, no effect.

- **decopoff**

DECOMPOFF=’1’, it disables code decompressor.

DECOMPOFF=’0’, no effect.

- **watchon**

WATCHON=’1’, it enables data watchpoints. Write accesses to the internal data memory at the address MGCWATCH will generate a break (WATCH bit of MGCSTAT) forcing mAgicV into debug mode.

WATCHON=’0’, no effect.

- **watchoff**

WATCHOFF=’1’, it disables data watchpoints.

WATCHOFF=’0’, no effect.

- **breakon**

BREAKON=’1’, it enables program breakpoints, parity errors will be interpreted as breakpoints.

BREAKON=’0’, no effect

- **breakoff**

BREAKOFF=’1’, it disables program breakpoints.

BREAKOFF=’0’, no effect.

- **intovfon**

INTOVFON=’1’, it enables signed integer overflow detection.

INTOVFON=’0’, no effect.

- **intovfoff**

INTOVFOFF=’1’, it disables signed integer overflow detection.

INTOVFOFF=’0’, no effect.

- **pmuon**

PMUON=’1’, it enables PMU.

PMUON=’0’, no effect.

- **pmuoff**

PMUOFF=’1’, it disables PMU, virtual = physical address (8K physical address).

PMUOFF=’0’, no effect.

- **tickon**

TICKON=’1’, it enables clock tick counter (MGCSTEP register).

TICKON=’0’, no effect.

- **tickoff**

TICKOFF='1', it disables clock tick counter.

TICKOFF='0', no effect.

- **lockon**

LOCKON='1', it enables write lock on the MGCCTRL register. When the lock is active only an external AHB master can write the MGCCTRL register. If mAgicV core tries to write the MGCCTRL with this lock active the write fails and an exception is generated.

LOCKON='0', no effect.

- **lockoff**

LOCKOFF='1', it removes the write lock on the MGCCTRL register.

LOCKOFF='0', no effect.

- **allfatalon**

ALLFATALON='1', all exceptions are fatal and put mAgicV in debug mode.

ALLFATALON='0', no effect.

- **allfataloff**

ALLFATALOFF='1', only some exceptions are fatal and put mAgicV in debug mode (see Section 5.3.1.1 "MGCEXCEPTION" on page 87).

ALLFATALOFF='0', no effect.

- **nofatalon**

NOFATALON='1', all the exceptions are non-fatal and they can be recovered by an appropriate exception service routine.

NOFATALON='0', no effect.

- **nofataloff**

NOFATALOFF='1', only some exceptions are fatal and put mAgicV in debug mode.

NOFATALOFF='0', no effect.

- **swreset: software reset**

SWRESET='1', mAgicV state is reset and put in debug mode. To make an hard reset use the MGCRESET control register.

SWRESET='0', no effect.

4.3.7.2 MGCSTAT Register

State flags are meaningful only if they assume the logic value '1'.

Table 4-11. MGCSTAT Register

31	30	29	28	27	26	25	24
-	-	stepreq	clken	ticken	watchen	intovfen	tickovf
23	22	21	20	19	18	17	16
nonefatal	allfatal	mgctrlock	rpt single	pmcheck	pty2break	softpipe	pmuen

15	14	13	12	11	10	9	8
decopen	intdbgreq	gbreak	watch	break	extdbgreq	stop	CT
7	6	5	4	3	2	1	0
halt	interrupt	exception	step	start	debug	run	sleep

- **sleep**

mAgicV is in sleep mode.

- **run**

mAgicV is in run mode.

- **debug**

mAgicV is in debug mode.

- **start**

mAgicV is starting a program (enabled by setting the START bit of the MGCCTRL).

- **step**

mAgicV is in step mode (enabled by setting the STEPON bit of the MGCCTRL).

- **exception**

An exception occurred.

- **interrupt**

mAgicV is serving an interrupt service routine.

- **halt**

An HALT FLOW has been executed.

- **CT: Cross Triggering**

Cross Triggering enabled (enabled by setting the TRIGGON bit of the MGCCTRL).

- **stop**

mAgicV was stopped (enabled by setting the STOP bit of the MGCCTRL).

- **extdbgreq: external debug request**

This is an external debug request. The request is meaningful only if Cross Triggering is enabled.

- **break**

a breakpoint occurred (reset by the START and the CONTINUE bit of the MGCCTRL).

- **watch**

a data watchpoint occurred (reset by the START and the CONTINUE bit of the MGCCTRL).

- **gbreak: gui break**

a “software breakpoint” occurred (enabled by setting the GBREAKON bit of the MGCCTRL).

- **intdbgreq: internal debug request**

debug request sent to an external AHB controller. The request is meaningful only if Cross Triggering is enabled.

- **decopen: decompressor enabled**

program memory decompressor is enabled (enabled by setting the DECOPON bit of the MGCCTRL).

- **pmuen: PMU enabled**

PMU enabled (enabled by setting the PMUON bit of the MGCCTRL).

- **softpipe: HW software pipeline support enabled**

HW software pipeline support enabled (enabled by the program).

- **pty2break: Parity to break**

Parity errors are seen as breakpoints (enabled by setting the BREAKON bit of the MGCCTRL).

- **pmchecken: Program Memory check enabled**

Parity check on Program Memory enabled (enabled by setting the PMCHECKEN bit of the MGCCTRL).

- **rpt single: repeat single**

mAgicV is in repeat single mode (enabled by the program).

- **mgctrlock: mgcctrl lock**

MGCCTRL is locked (enabled by setting the LOCKON bit of the MGCCTRL). Only an external AHB master controller can write on MGCCTRL register.

- **allfatal: all exceptions fatal**

all exceptions put mAgicV in debug mode (enabled by setting the ALLFATALON bit of the MGCCTRL)

- **nonefatal: all exceptions non-fatal**

all exception are considered non-fatal (enabled by setting the NOFATALON bit of the MGCCTRL), they can be handled by an interrupt handler.

- **tickovf: tick counter overflow**

32 bits MGCSTEP counter overflow (reset by writing the MGCSTEP register).

- **intovfen: integer overflow enabled**

integer overflow generate exception (enabled by setting the INTOVFON bit of the MGCCTRL).

- **watchen: watch point enabled**

watch point enabled (enabled by setting the WATCHON bit of the MGCCTRL). Write accesses to the internal data memory at the address MGCWATCH set the WATCH status flag and put mAgicV in debug mode.

- **ticken: tick counter enabled**

cycle counting enabled (enabled by setting the TICKON bit of the MGCCTRL).

- **stepreq: step mode request**

in step mode this flag goes to ‘1’ at each run cycles forcing mAgicV into debug mode.

- **clken: clock enabled**

this flag shows if core pipelines are running.

4.3.7.3 *MGCRESET Register*

A hard reset is done by writing the value 0xAC1CDEAD in this register.

4.4 Register Map

mAgicV registers address mapping can be divided into two banks. The first bank is composed of 128 registers that group: FLOW registers, ARF registers and DMA registers.

The second bank is composed of 256 Register File registers.

In register movements a 9-bit destination address RFA7 is used to access these registers. Bank selection is done through the most significant RFA7 address bit (see Section 3.9.1 ”Register Movements” on page 43).

Table 4-13 on page 73 illustrates the internal and AHB external mapping of mAgicV internal registers.

4.4.1 Register Access Modes

mAgicV registers are classified in Table 4-12 below as:

- Read Only registers
- Write Only registers
- Read/Write registers
- Read and Clear
- Write And Test (see Section 4.4.1.2 ”Write And Test” on page 72)

4.4.1.1 *Read and Clear*

Some of the status and exception registers are cleared upon read.

4.4.1.2 *Write And Test*

Some of mAgicV registers support the Write And Test (WAT) optimization that allows to test one or more bits of a register by writing a mask ‘1’ in the positions to be tested, and ‘0’ in other positions. The WAT condition flag is set to ‘1’ if all the positions tested are ‘1’.

Table 4-12. Register Access Modes

Register Type	Description
WAT	Write And Test support
RC	Read and Clear after read (exception register)
RW	Read and Write register
RO	Read Only register (status register)
WO	Write Only register (control register).

4.4.2 Map of mAgicV Registers

The following table summarizes the name, access mode, reset value, internal address (accesses performed by mAgicV core) and offset for accesses performed by external AHB masters.

Table 4-13. mAgicV Register Map

mAgicV Address	AHB Offset	Name	Access Mode	Reset Value
0x0	0x0	MGCCTRL	WO	NA
0x0	0x0	MGCEXCEPTION	RO	0x00000000
0x1	0x4	MGCSTAT	RO,WAT	0x00000204
0x2	0x8	MGCMASK	RW	0x00000000
0x3	0xC	MGCEXCEPTION	RC	0x00000000
0x4	0x10	MGCSTIKY0	RC	0x00000000
0x5	0x14	MGCSTIKY1	RC	0x00000000
0x6	0x18	MGCCONDITION	RW	0x00000000
0x7	0x1C	MGCREPEAT	RW	NA
0x8	0x20	MGCPC	RW	0x0000
0x9	0x24	MGCSTK	RW	0x0000
0xA	0x28	MGCBRANCH	RW	NA
0xB	0x2C	MGCPREDICATION	RW	0x007
0xC	0x30	MGCSTEP	RW	0x00000000
0xD	0x34	MGCWATCH	RW	NA
0xE	0x38	MGCMUTXCTRLSTAT	WAT	0x00000000
0xF	0x3C	MGCLOOPCNT	RW	0x00000000
0x10	0x40	MGCLOOPLEN	RW	0x00000000
0x11	0x44	General Purpose	RW	NA
0x12	0x48	General Purpose	RW	NA
0x13	0x4C	General Purpose	RW	NA
0x14-0x23	0x50-0x8C	ARF-AM	RW	NA
0x24-0x33	0x90-0xCC	ARF-M	RW	NA
0x34-0x43	0xD0-0x10C	ARF-A	RW	NA
0x44-0x53	0x110-0x14C	ARF-L	RW	NA
0x54-0x63	0x150-0x18C	ARF-S	RW	NA
0x64	0x190	MGCDMAEXTADD	RW	NA
0x65	0x194	MGCDMAEXTCIRCLEN	RW	NA
0x66	0x198	MGCDMAEXTMOD	RW	NA
0x67	0x19C	MGCDMAINTADD	RW	NA
0x68	0x1A0	MGCDMAINTCIRCLEN	RW	NA
0x69	0x1A4	MGCDMAINTMOD	RW	NA

mAgicV Address	AHB Offset	Name	Access Mode	Reset Value
0x6A	0x1A8	MGCDMALEN	RW	NA
0x6B	0x1AC	MGCDMAINTSEG	RW	NA
0x6C	0x1B0	MGCDMACURLEN	RO	0x0000
0x6D	0x1B4	MGCDMACUREXTADD	RO	NA
0x6E	0x1B8	MGCDMAreserved	RO	NA
0x6F	0x1BC	MGCDMASTAT	WAT	0x00000000
0x6F	0x1BC	MGCDMACTRL	WO	NA
0x70	0x1C0	MGCSTKIQ	RW	0x00000000
0x71-0x78	0x1C4-0x1E0	MGCINTSVR	RW	NA
0x79	0x1E4	MGCINTMASK	RW	0xFFFF
0x7A	0x1E8	MGCINTSTAT	RO	0x00000000
0x7B	0x1EC	MGCINTGSTAT	RO	0x0000
0x7C	0x1F0	MGCINTCTRL	WO	NA
0x7D	0x1F4	MGCWAKECTRL	WO	NA
0x7D	0x1F4	MGCWAKESTAT	RO	0x00000000
0x7E	0x1F8	MGCINTSETRESET	WO	NA
0x7E	0x1F8	MGCINTRET	RO	NA
0x7F	0x1FC	MGCINTPRIORITY	RW	0x0000000
NA	0x200-0x2FC	PMUVIRT	RW	NA
NA	0x300-0x31C	PMUMAPPEDVIRT	RW	NA
NA	0x320	PMUFAULTADD	RW	NA
NA	0x324	PMUEXTADD	RW	NA
NA	0x328	PMUCTL	WO	NA
NA	0x328	PMUSTAT	RO	0x00
NA	0x32C	PMUMISSCNT	RW	0x00000000
NA	0x330	PMUPHYSPNT	RW	0x7
0x100-0x1FF	0x400-0xBFC	RF	RO(AHB) RW(Core)	NA
NA	0xC00-0xC0C	MGCVLIW	RO	NA
NA	0xFC0	MGCRESET	WO	NA

4.5 Multicore Synchronization Support

4.5.1 Mutex Support

mAgicV provides 16 mutexes to safely manage resources sharing between an external AHB master controller and the mAgicV core. There is no predefined meaning for the mutex registers. The association between mutex and shared resources is driven by software that must add control code to manage the access to the shared resources. The hardware guarantees an atomic write and test operation to lock mutexes and a fixed priority (external AHB master first) for contemporaneous write accesses.

4.5.1.1 MGCMUTEXCTRLSTAT

The interface is based upon a 32-bit control and status register MGCMUTEXCTRLSTAT. When this register is read it reports the lock status and the owner of each mutex. In write it is used to lock and unlock mutexes. The status of the lock/unlock operation is reported into the WAT condition flag.

Table 4-14. MGCMUTEXCTRLSTAT (Read)

31	30	29	28	27	26	25	24
owner15	owner14	owner13	owner12	owner11	owner10	owner9	owner8
23	22	21	20	19	18	17	16
owner7	owner6	owner5	owner4	owner3	owner2	owner1	owner0
15	14	13	12	11	10	9	8
locked15	locked14	locked13	locked12	locked11	locked10	locked9	locked8
7	6	5	4	3	2	1	0
locked7	locked6	locked5	locked4	locked3	locked2	locked1	locked0

- **locked0-locked15**

lockedx='1' mutex 'x' is locked, owner 'x' is the owner of the mutex.

lockedx='0' mutex 'x' is free.

- **owner0-owner15**

These fields are meaningful only if the relative locked 'x' ='1'.

onwerx='1', an external AHB master controller is the owner of the mutex 'x'.

ownerx='0', mAgicV is the owner of the mutex 'x'.

Table 4-15. MGCMUTEXCTRLSTAT (Write)

31	30	29	28	27	26	25	24
unlock15	unlock14	unlock13	unlock12	unlock11	unlock10	unlock9	unlock8
23	22	21	20	19	18	17	16
unlock7	unlock6	unlock5	unlock4	unlock3	unlock2	unlock1	unlock0
15	14	13	12	11	10	9	8
lock15	lock14	lock13	lock12	lock11	lock10	lock9	lock8
7	6	5	4	3	2	1	0
lock7	lock6	lock5	lock4	lock3	lock2	lock1	lock0

- **lock0-lock15**

Either an external AHB master controller or mAgicV try to set a mutex lock by writing ‘1’ in these fields. If the lock succeeds (mutex free) the owner and the lock fields of the mutex status register of Table 4-14 on page 75 are set accordingly.

By writing ‘0’ no effect.

- **unlock0-unlock15**

Either an external AHB master controller or mAgicV try to remove a mutex lock by writing ‘1’ in these fields. If the unlock succeeds (the writer is the owner of the mutex) the lock field of the mutex status register of Table 4-14 on page 75 is cleared.

By writing ‘0’ no effect.

NOTE: by writing more than one bit in either the lock or the unlock field more than one mutex at a time can be respectively locked or unlocked.

NOTE: in case of a contemporary write by mAgicV and by an external AHB master controller on the mutex control register, the AHB master will perform the operation.

NOTE: the WAT condition flag is automatically updated when mAgicV writes on the MGCMUTEXCTRLSTAT, if the mutex control operation succeeds WAT=’1’ otherwise WAT=’0’. This flag can be checked directly by jumps to implement fast lock and unlock routines.

4.5.2 Lock and Unlock Software Procedures

The following pseudo C-code implements the pair of SW mutex lock/unlock (blocking) procedures to be executed by the mAgicV core and by an external AHB master (e.g. an ARM® processor).

```
do {/// LOCK procedure
 u32 res;
 // write the lock field
 WRITE32(AT91_MAGIC_REG_BASE,AT91_REG_MGCMUTEXCTL,(1<<x));
 // read the status register
 READ32(AT91_MAGIC_REG_BASE,AT91_REG_MGCMUTEXCTL,res);
#ifdef ARM
  // ARM loops until the mutex is not acquired (lock=’1’ and owner=’1’)
} while(((res & (1<<x)) && (res & (1<<(x+16))))==0);
#else
  // mAgicV loops until the mutex is not acquired (lock=’1’ and owner=’0’)
} while(((res & (1<<x)) && (!(res & (1<<(x+16)))))==0);
#endif
```

The unlock procedure does not have to test the owner; need only test if the mutex is unlocked.

```
do {/// UNLOCK procedure
 u32 res;
 // write the unlock field
 WRITE32(AT91_MAGIC_REG_BASE,AT91_REG_MGCMUTEXCTL,(1<<(x+16)));
 // read the status register
 READ32(AT91_MAGIC_REG_BASE,AT91_REG_MGCMUTEXCTL,res);
  // loop until the mutex is not released lock=’0’)
} while(((res & (1<<x))==0);
```

4.5.3 Programmable Output Lines

mAgicV has 5 output lines named: SIRQ0,SIRQ1,SIRQ2,SIRQ3, HALT, which can be used to generate waveforms (SIRQ2,SIRQ3) or external interrupts (SIRQ0,SIRQ1,HALT). The status of these lines is directly driven by an explicit FLOW code.

Each SIRQ2 or SIRQ3 FLOW opcodes causes a toggle of the corresponding line.

Table 4-16. Sirq Generation Issue

vliw[3:2]	vliw[118:113]	vliw[51:50]
FLOW predication	sirq=0x1f	sirqnum

Table 4-17. Sirq Description

Sirqnum	Description
0x0	cycle pulse on line SIRQ0
0x1	cycle pulse on line SIRQ1
0x2	toggle value on line SIRQ2, reset
0x3	toggle value on line SIRQ3, reset value 0

NOTE: SIRQ2 and SIRQ3 values are contained into the MGCCOND register so that they can also be modified by writing the MGCCOND register.

The HALT code, puts mAgicV in debug mode by setting the HALTED bit in the MGCSTAT register, this value is reported in the HALT output line.

Table 4-18. Halt Generation Issue

vliw[3:2]	vliw[118:113]
FLOW predication	halt=0x23

5. Event Handling

When an event occurs the execution of the instruction stream can be:

1. passed to an event handler at an address specified by one of the 8 MGCINTSVR registers. See Section 5.1 below.

2. resumed by a previous state of sleep (see Section 5.2 ”Sleep and Wakeup” on page 84).

3. halted and then pass in debug mode (see Section 5.3 ”Exceptions” on page 86).

5.1 Interrupt Handling

mAgicV allows very fast interrupt handling by treating interrupts as a routine processor instruction (branch, call, ret). Interrupts don’t break pipelines and save only return program counter into the read only MGCINTRET register. mAgicV doesn’t cross protection domains to take an interrupt. Since the protection domain remains unchanged on a interrupt, the Interrupt Service Routine is called as a normal function call.

There are 8 prioritized interrupt lines. Line0 and line1 multiplex four lines each (named shared lines), so that the number of interrupt lines is 14. Each interrupt line is associated to a 16-bit interrupt vector register (MGCINTSVR) that must be set to a valid program address, corresponding to the handler interrupt routine. An interrupt, on a previously enabled and not masked interrupt line (via the MGCINTCTRL register), is registered into the PEND field of the MGCINTSTAT interrupt status register and served in a synchronous way in correspondence of the FLOW codes shown in Table 5-1.

Table 5-1. interrupt Service FLOW Codes

FLOW Opcode	Mnemonic	Description
0x1D	watchint	check for pending interrupts
0x26	br	branch
0x27	br_reg	branch register
0x2C	ret	return from call
0x37	repb	repeat block

Table 5-2. Watchint Issue

vliw[3:2]	vliw[118:113]
FLOW predication	watchint=0x1D

Table 5-3. Reti Issue

vliw[3:2]	vliw[118:113]
FLOW predication	reti=0x2D

Interrupts can be masked using the MGCINTMASK. A masked interrupt is always registered as a pending interrupt, but it won't be served until it's unmasked.

When the program jumps to an Interrupt Service Routine the ISVR MGCSTAT bit will be set, indicating that no more interrupts will be served until a return from interrupt instruction (RETI) is executed. The user code return address is saved into the MGCINTRET register and it's automatically restored in the MGCPC register when a RETI issue is executed.

In case more than one interrupt is pending, the line with a greater priority will be served. In case of equal priority the interrupt line with a lower number will be served.

The priority register MGCINTPRIO is a 24-bit register that allows to associate three bits of priority to each line.

Pending interrupts can be set and cleared by using the MGCINTSETRESET; this feature can be used to generate or clear interrupts by software over each line.

5.1.1 Interrupt Sources

Interrupts line0 and line1 are special lines that group 4 interrupt lines each (LINESHx). It's a way to extend the number of interrupts without adding additional resources in terms of interrupt vector registers and prioritization logic. These shared lines can be connected to relatively slow and homogenous devices as for instance SPIs and SSCs.

An interrupt on line 0 or line 1 is the OR of interrupts on the shared lines.

INT0 = LINESH0 OR LINESH1 OR LINESH2 OR LINESH3.

INT1 = LINESH4 OR LINESH5 OR LINESH6 OR LINESH7.

Table 5-4 shows the interrupt sources and the associated interrupt Service Vector Routine registers.

Table 5-4. Interrupt Sources

INT #	Interrupt Line	Source	Description	SVR
0	LINESH0	external	D940 implementation: SSC0	MGCINTISVR0
	LINESH1	external	D940 implementation: SSC1	
	LINESH2	external	D940 implementation: SSC2	
	LINESH3	external	D940 implementation: SSC3	
1	LINESH4	external	D940 implementation: EXT0	MGCINTISVR1
	LINESH5	external	D940 implementation: EXT1	
	LINESH6	external	D940 implementation: EXT2	
	LINESH7	external	D940 implementation: EXT3	
2	line2	external	D940 implementation: Timer channel A1	MGCINTSVR2
3	line3	external	D940 implementation: SPI0	MGCINTSVR3
4	line4	internal	DMA End Of Transfer (one of the four channels raises an EOT)	MGCINTSVR4

INT #	Interrupt Line	Source	Description	SVR
5	line5	internal	Watch point. There is a write done in the data memory location specified by mgcwatch. Often used to wake up processor sleeping. (see Section 5.1.1.2)	MGCINTSVR5
6	line6	internal	non-fatal exception, MGCEXCEPTION register	MGCINTSVR6
7	line7	internal	MGCSTEP counter overflow (see Section 6. ”Profiling and Debug support” on page 90)	MGCINTSVR7

5.1.1.1 *EOT Interrupt*

The EOT interrupt (INT#4) is generated when one of the four DMA channels generates an EOT.

NOTE: EOTs generated by the DMA and launched by the PMU are not registered.

5.1.1.2 *Watch Interrupt*

This interrupt (INT#5) is generated when a write operation is performed in the internal data memory at the location pointed by the MGCWATCH register.

NOTE: if the WATCHEN bit of the MGCSTAT is set, mAgicV will go in debug mode because the register is used as watch point register (see Section 6.2 ”Debug” on page 90).

5.1.1.3 *Exception Interrupt*

The interrupt exception signal (INT #6) is the logic OR of all non fatal exceptions.

NOTE: if the NONEFATAL flag of the MGCSTAT is set, it is the logic OR of all exception sources.

5.1.1.4 *Tick overflow interrupt*

This interrupt (INT #7) is generated when the TICKOVF bit of the MGCSTAT is set (MGCSTEP=0xFFFFFFFF). The overflow bit is cleared by writing the MGCSTEP register.

5.1.2 Interrupt Registers

5.1.2.1 *MGCINTCTRL*

The MGCINTCTRL is a control register that allows to enable/disable an interrupt line x by writing '1' in the appropriate ENx or DISx field. The SHxNEG and SHxPOS fields allow to choose the edge level sensitivity of the 8 shared lines (used for external interrupts).

Table 5-5. MGCINTCTRL Register

31	30	29	28	27	26	25	24
sh7neg	sh7pos	sh6neg	sh6pos	sh5neg	sh5pos	sh4neg	sh4pos
23	22	21	20	19	18	17	16
sh3neg	sh3pos	sh2neg	sh2pos	sh1neg	sh1pos	sh0neg	sh0pos
15	14	13	12	11	10	9	8
dis7	en7	dis6	en6	dis5	en5	dis4	en4
7	6	5	4	3	2	1	0
dis3	en3	dis2	en2	dis1	en1	dis0	en0

- **en0-en7**

ENx='1', it enables the interrupt of the corresponding x line.

ENx='0', no effect.

- **DIS0-DIS7**

DISx='1', it disables the interrupt of the corresponding x line.

DISx='0', no effect.

- **sh0pos-sh7pos**

SHxPOS='1', it sets the sensitivity of the corresponding shared x line to be positive edge sensitive.

SHxPOS='0', no effect.

- **sh0neg-sh7neg**

SHxNEG='1' , it sets the sensitivity of the corresponding shared x line to be negative edge sensitive.SHxNEG='0', no effect.

5.1.2.2 *MGCINTSETRESET*

The MGCINTSETRESET is a control register that can be used to set and clear interrupt pending bits into the MGCINTSTAT by writing '1' in the corresponding bit field.

NOTE: this register can be used to realize SWI.

Table 5-6. MGCINTCTRL Register

15	14	13	12	11	10	9	8
clrint7	setint7	clrint6	setint6	clrint5	setint5	clrint4	setint4
7	6	5	4	3	2	1	0
clrint3	setint3	clrint2	setint2	clrint1	setint1	clrint0	setint0

- **setint0-seting7**

SETINTx='1', it arises an interrupt in line x (PENDx='1' in the MGCINTSTAT).

SETINTx='0', no effect.

- **clrint0-clrint7**

CLRINTx='1', it clears an interrupt in line x (PENDx='0' in the MGCINTSTAT).

CLRINTx='0', no effect.

5.1.2.3 *MGCINTSTAT*

The MGCINTSTAT contains the information about the enabled interrupt lines ENx, the pending lines PENDx and the edge sensitivity of the 8 shared lines.

Table 5-7. MGCINTSTAT Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
sh7edge	sh6edge	sh5edge	sh4edge	sh3edge	sh2edge	sh1edge	sh0edge

15	14	13	12	11	10	9	8
en7	en6	en5	en4	en3	en2	en1	en0
7	6	5	4	3	2	1	0
pend7	pend6	pend5	pend4	pend3	pend2	pend1	pend0

- **pend0-pend7**

PENDx=‘1’, pending interrupt on line x.

PENDx=’0’, no interrupt on line x.

- **en0-en7**

ENx=‘1’, enable interrupt line x.

ENx=’0’, no interrupt enabled on line x. MGCINSTAT does not register interrupts not enabled.

- **sh0edge-sh7edge**

SHxEDGE=‘1’, the corresponding shared x line is positive edge sensitive.

SHxEDGE=‘0’, the corresponding shared x line is negative edge sensitive.

5.1.2.4 *MGCINTGSTAT*

The MGCINTGSTAT register is used to save the status of pending shared interrupts when the associated interrupt service routine is called. When an INT0 or INT1 interrupt is served, the status of the relative pending shared interrupts is registered in the saved shared field of the MGCINTGSTAT, while the pending shared interrupt field is cleared. In this way the SVR0 or SVR1 interrupt service routine can check the shared line that generated the interrupt, and the HW will continue to register new shared interrupts.

Table 5-8. MGCINTGSTAT Register

15	14	13	12	11	10	9	8
shsave7	shsave6	shsave5	shsave4	shpend7	shpend6	shpend5	shpend4
7	6	5	4	3	2	1	0
shsave3	shsave2	shsave1	shsave0	shpend3	shpend2	shpend1	shpend0

- **shpend0-shpend3**

SHPENDx=’1’, pending interrupt on shared line x (belonging to INT 0).

SHPENDx=’0’, no interrupt on shared line x.

- **shsave0-shsave3**

SHSAVEx, status of the interrupt line x when the service routine is served.

- **shpend4-shpend7**

SHPENDx=’1’, pending interrupt on shared line x (belonging to INT 1).

SHPENDx=’0’, no interrupt on shared line x.

- **shsave4-shsave7**

SHSAVEx, status of interrupt line x when the service routine is served.

5.1.2.5 MGCINTMASK

The MGCINTMASK is a 16-bit register that is used to mask enabled interrupts MSKx and shared interrupts lines MSKSHx. Setting '1' means interrupt masked. An enabled interrupt is registered into the pending register even if it is masked.

Table 5-9. MGCINTMASK Register

15	14	13	12	11	10	9	8
msksh7	msksh6	msksh5	msksh4	msksh3	msksh2	msksh1	msksh0
7	6	5	4	3	2	1	0
msk7	msk6	msk5	msk4	msk3	msk2	msk1	msk0

- **msk0-msk7**

MSKx='1', interrupt line x is masked.

MSKx='0', interrupt line x is not masked.

- **msksh0-msksh7**

MSKSHx='1', interrupt line SHx is masked.

MSKSHx='0', interrupt line SHx is not masked.

5.1.2.6 MGCINTPRIO

In case of more than one pending interrupt, the line with a greater priority will be served, in case of equal priority the line with a lower number will be served.

The MGCINTPRIO is a 24-bit register, PRIOx[2:0] allows to define an eight level priority for line x. Lower values have higher priority (0: max priority-7=min priority).

Table 5-10. MGCINTPRIO Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
PRIO7			PRIO6			PRIO5	
15	14	13	12	11	10	9	8
PRIO5	PRIO4			PRIO3			PRIO2
7	6	5	4	3	2	1	0
PRIO2		PRIO1			PRIO0		

- **prio0-prio7**

priority level of associated interrupt.

5.1.2.7 MGCINTRET

This read only register contains the return address of the interrupted user program. A RETI FLOW code moves this register into the MGCPC and clears the INTERRUPT bit of the MGCSTAT.

NOTE: a double RETI causes exception.

5.2 Sleep and Wakeup

mAgicV can go to sleep mode by writing the MGCCTRL register or by using the explicit FLOW codes shown in Table 5-11 below. The processor will be woken up by one of the interrupts shown in Table 5-4 on page 79 or by four EOT (End of Transfer) events coming from the DMA. The events that can wake mAgicV up from a sleep state are selected using the MGCWAKECTRL control register.

NOTE Rev A specific: Sleep instruction must be scheduled at least three PM 128-bit locations before the 1K boundary of each PM virtual page.

Table 5-11. Sleep FLOW codes

FLOW Opcode	Mnemonic	Description
0x1E	sleepon	Put mAgicV in sleep mode waiting events enabled in MGCWAKESTAT
0x24	writedma	Start a pre-initialized dma write toward external resources, put the system in sleep mode and set wakeup EOT event to the currently used channel
0x38	readdma	Readma start a pre-initialized dma toward mAgicV internal resources, put the system in sleep mode and set wakeup eot event to the currently used channel

Table 5-12. Sleep Issue

vliw[3:2]	vliw[118:113]
FLOW predication	sleepon, writedma, readdma

5.2.1 Wakeup Registers

5.2.1.1 MGCWAKECTRL

Wake Up control register. By writing this register wake up events registered in the MGCWAKESTAT are cleared.

Table 5-13. MGCWAKECTRL Register

23	22	21	20	19	18	17	16
eotdis3	eoten3	eotdis2	eoten2	eotdis1	eoten1	eotdis0	eoten0
15	14	13	12	11	10	9	8
intdis7	inten7	intdis6	inten6	intdis5	inten5	intdis4	inten4
7	6	5	4	3	2	1	0
intdis3	inten3	intdis2	inten2	intdis1	inten1	intdis0	inten0

- **inten0-inten7**

INTENx=‘1’, it enables wakeup on interrupt line x.

INTENx=’0’, no effect.

- **intdis0-intdis7**

INTDISx=‘1’, it disables wake up on interrupt line x.

INTDISx=’0’, no effect.

- **eoten0-eoten7**

EOTENx='1', it enables wakeup on the EOT of channel x.

EOTENx='0', no effect.

- **eotdis0-eotdis7**

EOTDISx='1', it disables wakeup on the EOT of channel x.

EOTDISx='0', no effect.

5.2.1.2 *MGCWAKESTAT*

This register reports the wake up events currently enabled and the wakeup events occurred from the last MGCWAKECTRL write access.

Table 5-14. MGCWAKESTAT Register

23	22	21	20	19	18	17	16
eot3ev	eot2ev	eot1ev	eot0ev	int7ev	int6ev	int5ev	int4ev
15	14	13	12	11	10	9	8
int3ev	int2ev	int1ev	int0ev	eot3wk	eot2wk	eot1wk	eot0wk
7	6	5	4	3	2	1	0
int7wk	int6wk	int5wk	int4wk	int3wk	int2wk	int1wk	int0wk

- **int0wk-int7wk**

INTxWK='1', it wakes up on interrupt line x.

INTxWK='0', no wake up on interrupt line x.

- **eot0wk-eot3wk**

EOTxWK='1', it wakes up on the EOT of channel x.

EOTxWK='0', no wake up on the EOT of channel x.

- **int0ev-int7ev**

INTxEV='1', an event raised on interrupt line x.

INTxEV='0', no event on interrupt line x.

- **eot0ev-eot3ev**

EOTxEV='1', an event arose on EOT of channel x.

EOTxEV='0', no event on EOT of channel x.

5.3 Exceptions

mAgicV exceptions are divided into fatal and non fatal exceptions (see Table 5-15 below). Non masked fatal exceptions cause the processor to stop immediately and to enter in debug mode. Other exceptions can be handled in run mode by the exception interrupt routine number 6 (see Table 5-4 on page 79). Exception register MGCEXCEPTION collects exceptions.

5.3.1 Fatal and non Fatal Exceptions

The non fatal exceptions can be considered fatal if the ALLFATAL bit of the MGCSTAT (see Table 4-11 on page 69) is enabled. Conversely, all fatal exceptions can be considered as non fatal if the NOFATAL bit of the MGSTAT is enabled.

Table 5-15. Fatal Exceptions

Exception	Type
badin	not fatal
badout	not fatal
divbyzero	not fatal
addunkn	fatal
agu0ovf	not fatal
agu0pty	not fatal
agu0aovf	not fatal
agu1ovf	not fatal
agu1pty	not fatal
agu1aovf	not fatal
slverr	fatal
msterr	fatal
softex	not fatal
ctrlockex	fatal
rwagu0	fatal
rwagu1	fatal
write RF7	fatal
write RF5	fatal
write ARF	fatal
write FLOW	fatal
write DMA	fatal
agu0unkn	fatal
agu1unkn	fatal
ptyerr	fatal
rtierr	fatal
dblwrite	fatal

Exception	Type
pagefree	fatal
mulunkn	fatal
dmabusy	fatal

NOTE: The exception signal that halts the program execution is the logic OR of all non fatal exceptions.

5.3.1.1 *MGCEXCEPTION*

Exception flags assume the logic value ‘1’ in case of non masked exception.

NOTE: an exception is considered masked if the corresponding bit into the MGCMASK register is '1'.

This register has two different addresses:

- address 0x0, reads the register without clearing it.
- address 0x3 (0xC AHB offset), reads register and clears it.

Table 5-16. MGCEXCEPTION Register

31	30	29	28	27	26	25	24
-	-	-	dmabusy	mulunkn	pagefree	dblwrite	rtierr
23	22	21	20	19	18	17	16
ptyerr	agu1ukn	agu0ukn	writedma	writeflow	writearf	writerf5	writerf7
15	14	13	12	11	10	9	8
rwagu1	rwagu0	ctrlockex	softex	msterr	slverr	agu1aovf	agu1pty
7	6	5	4	3	2	1	0
agu1ovf	agu0aovf	agu0pty	agu0ovf	addunkn	divzero	badout	badin

- **badin: FPU bad input operand**

A logical OR of all inop flags (see Section 3.4.9.2 ”Invalid Operation (Inop)” on page 26). Non Fatal.

- **badout: FPU bad output operand**

A logical OR of all overflow flags. Non Fatal.

- **divzero: FPU division by zero**

Section 3.4.9.3 ”Division by Zero (Div by Zero)” on page 27. Non Fatal.

- **addunkn: ADD unknown**

ADD issue unknown code. Fatal.

- **agu0ovf: AGU0 overflow**

AGU0 address out of bounds (max address 16383). Non Fatal.

- **agu0pty: AGU0 parity**

AGU0 address parity error (a vector access with odd address). Non Fatal.

- **agu0aovf: AGU0 arithmetic overflow**

AGU0 arithmetic overflow. Non Fatal.

- **agu1ovf: AGU1 overflow**

AGU1 address out of bounds (max address 16383). Non Fatal.

- **agu1pty: AGU1 parity**

AGU1 address parity error (a vector access with odd address). Non Fatal.

- **agu1aovf: AGU1 arithmetic overflow**

AGU1 arithmetic overflow. Non Fatal.

- **slverr: AHB Slave error**

an error is detected by AHB Slave. Fatal.

- **msterr: AHB Master error**

an error is detected by AHB Master. Fatal.

- **softex: software exception**

generated by a SEX FLOW code. Non Fatal.

- **ctrlockex: mgcctrl locked exception**

a write operation has been issued by mAgicV core, but the mgcctrl register was locked. Fatal.

- **rwagu0: AGU0 read write memory conflict**

AGU0 read conflicts with a previous write (SW scheduling problem). Fatal.

- **rwagu1: AGU1 read write memory conflict**

AGU1 read conflicts with a previous write (SW scheduling problem). Fatal.

- **write RF7: write conflict on RF port7**

More than one write was scheduled on the RF port7 (SW scheduling problem). Fatal.

- **write RF5: write conflict on RF port5**

More than one write was scheduled on the RF port5 (SW scheduling problem). Fatal.

- **write ARF: write conflict on ARF IO port**

More than one write was scheduled on the ARF IO port (SW scheduling problem). Fatal

- **write FLOW: write conflict on FLOW IO port**

More than one write was scheduled on the FLOW (SW scheduling problem). Fatal

- **write DMA: write conflict on FLOW DMA port**

More than one write was scheduled on the DMA (SW scheduling problem). Fatal.

- **agu0unkn: AGU0 unknown**

AGU0 unknown code. Fatal.

- **agu1unkn: AGU1 unknown**

AGU1 unknown code. Fatal.

- **ptyerr: parity error**

Program memory parity error. Fatal.

- **rtierr: RTI error**

Unexpected return from interrupt. Fatal.

- **dblwrite: double write**

double write at the same data memory address. Fatal.

- **pagefree**

PMU can’t find a physical page to accommodate a new page. Fatal.

- **mulunkn: MUL unknown**

MUL unknown code. Fatal.

- **dmabusy**

started a new DMA on a busy channel. Fatal.

6. Profiling and Debug support

6.1 Profiling registers

The user can evaluate the performance of the system by means of two mAgicV 32-bit counter registers.

6.1.1 MGCSTEP

The MGCSTEP register is used to collect information about the cycles spent in run mode. It includes the cycles of pipeline stall due to program cache miss or sleep mode. This counter can be accessed by mAgicV and by an external AHB master controller (see Table 4-13 on page 73).

Accessing TICKON and TICKOFF MGCCTRL control bits is possible to respectively start and stop the MGCSTEP counter register. An interrupt handler can be installed on INT #7 line, signalling the overflow of this counter. The overflow is registered in the MGCSTAT register and it's cleared by write operations on the MGCSTEP register.

NOTE: The cycles reported by the MGCSTEP depends on the program flow (deterministic) and by the AHB bus occupation (non deterministic).

6.1.2 PMUMISSCNT

The PMUMISSCNT register is used to collect information about the number of program misses executed. This register can be accessed only by an external AHB master controller.

These miss events can be monitored by reading the PMUSTAT (see Table 3-67 on page 59).

6.2 Debug

All the debug features can be accessed by an external AHB master that can read and write all mAgicV internal resources (memories and registers). There is a limitation on writing RF registers.

6.2.1 Breakpoint Support

mAgicV supports breakpoints by toggling a bit of the program VLIW corresponding to the breakpoint PMA. By setting PMCHKON and BREAKON on the MGCCTRL control register, a parity error is detected and interpreted as a breakpoint (PTY2BREAK flag of the MGCSTAT). The external debug engine should check if the triggered breakpoint is a break point or a real parity exception.

In case of Compressed Program Words the parity bit is located into the Super Header. See Section 3.2.1.1 "Compressed Program Word" on page 5.

6.2.2 Watch Point Support

mAgicV supports watchpoints through a 16-bit watch point register MGCWATCH that must contain the 16-bit internal data address of the watched variable. The watch-point logic detects write operations upon the specified watch address. To enable watchpoints the WATCHON bit of the MGCCTRL must be set.

6.2.3 Cross Triggering Support

The main function of the Cross Triggering is to pass debug events from one processor to another. The CT can communicate debug state information from one core (mAgicV) to another, so that the program execution on both processors can be stopped at the same time, if required.

The CT mode is enabled in mAgicV by setting the TRIGGON bit of the MGCCTRL. In this mode a dedicated mAgicV input line (dbg_req_from_arm) is used to put mAgicV immediately (1 cycle latency) in debug mode. Vice versa mAgicV has a dedicated output line to communicate to another core its debug state (dbg_req_to_arm).

6.2.4 Step Mode Support

In this mode a program is executed step by step, in this way it is possible to examine internal registers at each cycle. An external AHB master controller can activate this mode by setting the STEPON bit of the MGCCTRL. By setting the CONTINUE bit of the MGCCTRL, the controller can advance the program execution by one cycle.

NOTE: in this mode the DMA is not interrupted (it continues even if the core is frozen), so that temporizations are altered with respect to the normal run mode. For example, in presence of the DMA, the MGCSTEP counts less cycles than in normal run mode.

7. DMA

The DMA engine is a single channel with 4 independent programmable set of registers.

The DMA is able to perform the following 32-bit word memory accesses:

- fixed external and/or internal address.
- incremental external and/or internal address.
- incremental address with a fixed external and/or internal modifier ("jump" or "stride").
- incremental address, wrapping around a specified length on external and/or internal address.
- all of the above mixed.
- all of the above, using last accessed external and/or internal addresses or reloading them.

All temporary conditions on the AHB bus, no granted bus or page fault or retry/split condition, do not change the DMA channel that is currently operating (i.e. no new arbitration).

The DMA channels are serially processed and have fixed priority, the highest being channel number 3, the lowest number 0.

Highest priority channel 3 is used by the PMU (if enabled). As a good programming rule, the user application should avoid the usage of this channel when the PMU is enabled, otherwise unpredictable results could occur. Several PMU DMA parameters (like chunck length, modifiers, external address) are set at bootstrap and they must be kept fixed during the program execution.

Many parameters could be fixed throughout the entire application; moreover, thanks to the possibility to redo the transfer or continue the transfer with the same parameters and the current addresses, it could be also convenient to assign a DMA channel to a specific repetitive task, saving most of programming costs (i.e to access peripheral registers).

NOTE: Only 32-bit word accesses are supported.

7.1 DMA interface

The DMA interface is listed below:

- four sets each composed of 8 channel registers to store source, destination and configuration parameters (addresses, modifiers, length) of the 4 DMA channels.
- a global control and status register to control the DMA engine.
- three global read only shadow registers that show the parameters (length, internal address, external address) of the current memory access on the running channel.

The DMA interface can be programmed either from an AHB master controller or from a mAgicV core access.

7.1.1 DMA Channel Registers

All read/write operations on the DMA channel registers (see Table 7-1 on page 93) are performed on the currently selected channel, which is specified by the ACTIVECHAN field of the MGCDMASTAT register. To change the currently selected channel a write operation must be performed to the CHGCHAN field of the MGCDMACTRL register.

The DMA outputs a BUSYx signal to indicate that a DMA operation has been requested on channel x (by writing the READ or WRITE fields of the MGCDMACTRL register). The DMA generates a RUNx signal when the request on channel x is effectively processed. The DMA raises

an EOTx signal upon the completion of the whole burst on channel x. Both BUSYx and RUNx signals are cleared when the transfer on channel x ends (EOTx).

All these signals are registered in the MGCDMASTAT register and can be quickly tested by using the Write And Test fields of the MGCDMACTRL register.

By reading the BUSYx bit it is possible to know if the channel can be safely programmed.

Each channel can be programmed with new external and/or internal parameters (RLDEXT or RLDINT fields of MGCDMACTRL register) or with its latest accessed addresses (PRSVEXT or PRSVINT fields of MGCDMACTRL register).

NOTE: Busy channels can't be programmed, otherwise a DMABUSY exception is raised and registered in the MGCEXCEPTION register.

Each DMA channel transfer can be aborted by writing the ABORT or ABORTALL fields of the MGCDMACTRL register.

In case of abort a following read and write operation in "preserve mode" (that use latest DMA channel parameters) can lead to unpredictable results.

Table 7-1. DMA Channel Registers

mAgicV Address	AHB Offset	Name	Type	Reset Value
0x64	0x190	MGCDMAEXTADD	RW	NA
0x65	0x194	MGCDMAEXTCIRCLEN	RW	NA
0x66	0x198	MGCDMAEXTMOD	RW	NA
0x67	0x19C	MGCDMAINTADD	RW	NA
0x68	0x1A0	MGCDMAINTCIRCLEN	RW	NA
0x69	0x1A4	MGCDMAINTMOD	RW	NA
0x6A	0x1A8	MGCDMALEN	RW	NA
0x6B	0x1AC	MGCDMAINTSEG	RW	NA

7.1.1.1 MGCDMAEXTADD

It is a 32-bit register that specifies an AHB address space. In case of DMA write operations it contains an AHB destination address, otherwise an AHB source address. The external memory is byte addressed. However, the external address must be 32-bit word aligned.

NOTE: A misaligned address raises an AHB master error registered in the MASTERR bit of the MGCEXCEPTION register.

7.1.1.2 MGCDMAEXTCIRCLEN

It is a 24-bit register that specifies the external address bit mask to be applied to the external modified address. In order to obtain address wrapping around 1Kbyte, for instance, the mask should be set up to the value: 0x3FF. The biggest mask is 0xFFFFFF. Not using mask features means that the 0xFFFFFF mask is used (see Section 7.1.3 "DMA Address Generation" on page 100).

NOTE: Only type 2^X-1 masks bring to circular contiguous buffers of size 2^X.

7.1.1.3 MGCDMAEXTMOD

It is a 16-bit register that specifies the increment to add every AHB cycle to the preceding external address. If it is not 4 the AHB master has to split the burst in a sequence of 1-data transfers at the calculated addresses, making the DMA transfer less efficient.

NOTE: The modifier must be a multiple of 4 (32-bit accesses), or zero to access the same location. Other values raise a MASTERR exception.

7.1.1.4 MGCDMAINTADD

It is a 16-bit register that contains the internal mAgicV address. In case of DMA write operations it specifies a source address, otherwise it's a destination address.

The complete internal address is constituted by decoding the MGCDMAINTSEG register that specify the internal memory segments (see Table 7-2 on page 95).

NOTE: The mAgicV internal memory is word addressed. The word size depends on the internal memory segment. Only in case of DM_D memory space the internal mAgicV address must be multiplied by 2.

7.1.1.5 MGCDMAINTCIRCLEN

It is a 16-bit register that specifies the bit mask to be applied to the internal modified address. In order to obtain an address wrapping around 512Words, for instance, the mask shall be set up to the value: 0x1FF. The biggest mask is 0xFFFF. Not using mask features means that the 0xFFFF mask is used.

NOTE: Only type 2^X-1 masks bring to circular contiguous buffers of size 2^X.

7.1.1.6 MGCDMAINTMOD

It is a 16 bit register that specifies the increment to add every AHB cycle to preceding internal address. The default is 1.

NOTE: In case of DM_D memory space the increment must be multiple of 2. A zero value accesses the same location.

7.1.1.7 MGCDMALEN

It is a 16-bit register that holds the number of 32-bit word transfers. It does not express the number of bytes to transmit. For example a MGCDMALEN=128 refers to a transfer of 32 program memory 128-bit wide VLIWs, or 128 integer data word, or 128 single precision floating point data words, or finally 64 double precision (see Section 4.1 "Data Formats" on page 62) floating point data words.

7.1.1.8 MGCDMASEG

It is a 4-bit register that selects one of the 4 internal memory regions (PM, DM_I, DM_F and DM_D) according to the mapping shown in Table 7-2 on page 95 (see Section 3.7.1 "Program Memory Accesses" on page 39 and Section 3.7.2 "Data Memory Accesses" on page 40).

Table 7-2. Memory Segment Regions

Resource	AHB Start Address	AHB End Address	Access	Word Size	MGCDMASEG
PM	0x00600000	0x0061FFFF	4 x word32	128-bit	0x1
DM_I	0x00620000	0x0062FFFF	word32	32-bit	0x2
DM_F	0x00640000	0x0064FFFF	word32	32-bit	0x4
DM_D	0x00660000	0x0067FFFF	2 x word32	40-bit	0x8

7.1.2 DMA Control and Status Register

The DMA control and status registers have the same address. Write operations access the MGCDMACTRL control register, while read operations access the MGCDMASTAT register.

7.1.2.1 MGCDMACTRL

The MGCDMACTRL is a 32-bit control register and it is used to start and configure DMA channel transfers. This register can also be used to generate WAT condition by writing in the WAT fields (see Section 3.9.3 ”Conditioned and Unconditioned Jumps” on page 48).

All DMA operations (write, read, abort) are performed on the currently selected channel (ACTIVECHAN field of MGCDMASTAT register). By writing CHGCHAN and ACTIVECHAN fields in the MGCDMACTRL it is possible to change the currently selected channel.

Table 7-3. MGCDMACTRL Register

31	30	29	28	27	26	25	24
-	-	wateot3	wateot2	wateot1	wateot0	-	waterror
23	22	21	20	19	18	17	16
unlock	lock	watrun	watbusy	watrun3	watrun2	watrun1	watrun0
15	14	13	12	11	10	9	8
watbusy3	watbusy2	watbusy1	watbusy0	abortall	activechan		chgchan
7	6	5	4	3	2	1	0
prsvext	rldext	prsvint	rldint	clreot	abort	read	write

- **write**

WRITE=’1’, starts a DMA write operation on the selected channel.

WRITE=’0’, no effect.

- **read**

READ=’1’, starts a DMA read operation on the selected channel.

READ=’0’, no effect

- **abort**

ABORT=’1’, aborts the DMA operation on the selected channel.

ABORT=’0’, no effect.

- **clreot: clear EOTs**

CLREOT='1', clears EOTs registered into MGCDMASTAT.

CLREOT='0', no effect.

- **rldint: reload internal**

RLDINT='1', the internal DMA parameters of the selected channel are re-loaded from channel registers.

RLDINT='0', no effect.

- **prsvint: preserve internal**

PRSVINT='1', the internal DMA parameters of the selected channel are not reloaded from channel registers. The DMA continues from the latest accessed (not included) internal memory location.

PRSVINT='0', no effect.

- **rldext: reload external**

RLDEXT='1', the external DMA parameters of the selected channel are re-loaded from the channel registers.

RLDEXT='0', no effect.

- **prsvext: preserve external**

PRSVEXT='1', the external DMA parameters of the selected channel are not reloaded from the channel registers. The DMA continues from the latest accessed (not included) external memory location.

PRSVEXT='0', no effect.

- **chgchan: change channel**

CHGCHAN='1', it changes the active channel in the MGCDMASTAT to the value specified in the field ACTIVECHAN of the MGCDMACTRL register. For instance, to set channel 2 as an active channel the MGCDMACTRL register must be written with 0x500 (CHGCHAN='1' and ACTIVECHAN='2').

CHGCHAN='0, no effect.

- **activechan: active channel**

two bit parameters that selects the current channel.

- **abortall**

ABORTALL='1', aborts all the current and pending DMAs (can corrupt program, if PMU is enabled).

ABORTALL='0', no effect.

- **watbusy0: Write And Test on BUSY0**

WATBUSY0='1', tests if channel 0 is busy.

WATBUSY0='0', no effect.

- **watbusy1: Write And Test on BUSY1**

WATBUSY1='1', tests if channel 1 is busy.

WATBUSY1='0', no effect.

- **watbusy2: Write And Test on BUSY2**

WATBUSY2='1', tests if channel 2 is busy.

WATBUSY2='0', no effect.

- **watbusy3: Write And Test on BUSY3**

WATBUSY3='1', tests if channel 3 is busy.

WATBUSY3='0', no effect.

- **watrun0: Write And Test on RUN0**

WATRUN0='1', tests if channel 0 is running.

WATRUN0='0', no effect.

- **watrun1: Write And Test on RUN1**

WATRUN1='1', tests if channel 1 is running.

WATRUN1='0', no effect.

- **watrun2: Write And Test on RUN2**

WATRUN2='1', tests if channel 2 is running.

WATRUN2='0', no effect.

- **watrun3: Write And Test on RUN3**

WATRUN3='1', tests if channel 3 is running.

WATRUN3='0', no effect.

- **watbusy: Write And Test on BUSY**

WATBUSY='1', tests if any channel is busy.

WATBUSY='0', no effect.

- **watrun: Write And Test on RUN**

WATRUN='1', tests if any channel is running.

WATRUN='0', no effect.

- **lock**

LOCK='1', forces the exclusive use of the AHB bus (use with care), until unlock.

LOCK='0', no effect.

- **unlock**

UNLOCK='1', unlocks the AHB bus (default).

UNLOCK='0', no effect.

- **waterror: Write And Test on error**

WATERROR='1', tests for master error.

WATERROR='0', no effect.

- **wateot0: Write And Test on End Of Transfer 0**

WATEOT0=’1’, tests for channel 0 EOT.

WATEOT0=’0’, no effect.

- **wateot1: Write And Test on End Of Transfer 1**

WATEOT1=’1’, tests for channel 1 EOT.

WATEOT1=’0’, no effect.

- **wateot2: Write And Test on End Of Transfer 2**

WATEOT2=’1’, tests for channel 2 EOT.

WATEOT2=’0’, no effect.

- **wateot3: Write And Test on End Of Transfer 3**

WATEOT3=’1’, tests for channel 3 EOT.

WATEOT3=’0’, no effect.

7.1.2.2 MGCDMASTAT

It is a 32-bit register that reports the status of the 4 DMA channels.

Table 7-4. MGCDMASTAT Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	lock	eot3	eot2	eot1	eot0
15	14	13	12	11	10	9	8
-	dmaerr	prsvext	prsvint	activechan		run	busy
7	6	5	4	3	2	1	0
run3	run2	run1	run0	busy3	busy2	busy1	busy0

- **busy0**

BUSY0=’1’, channel 0 busy.

BUSY0=’0’, channel 0 free.

- **busy1**

BUSY1=’1’, channel 1 busy.

BUSY1=’1’, channel 1 free.

- **busy2**

BUSY2=’1’, channel 2 busy.

BUSY2=’0’, channel 2 free.

- **busy3**

BUSY3=’1’, channel 3 busy.

BUSY3='0', channel 3 free.

- **run0**

RUN0='1', channel 0 running.

RUN0='0', channel 0 not running.

- **run1**

RUN1='1', channel 1 running.

RUN1='0', channel 1 not running.

- **run2**

RUN2='1', channel 2 running.

RUN2='0', channel 2 not running.

- **run3**

RUN3='1', channel 3 running.

RUN3='0', channel 3 not running.

- **busy**

BUSY='1', some channels are busy (useful for WAT generation).

BUSY='0', all channels are free.

- **run**

RUN='1', some channels are running (useful for WAT generation).

RUN='0', all channels are not running

- **activechan: active channel**

currently selected channel.

- **prsvint:preserve internal**

PRSVINT='1', channel internal parameters are preserved. The DMA continues from the last accessed (not included) internal memory location.

PRSVINT='0', channel internal parameters are re-loaded.

- **prsvext:preserve external**

PRSVEXT='1', channel external parameters are preserved. The DMA continues from the last accessed (not included) external memory location.

PRSVEXT='0', channel external parameters are re-loaded.

- **dmaerr**

DMAERR='1', an error occured during a DMA.

- **eot0**

EOT0='1', End Of Transfer signal on channel 0.

- **eot1**

EOT1='1', End Of Transfer signal on channel 1.

- **eot2**

EOT2='1', End Of Transfer signal on channel 2.

- **eot3**

EOT3='1', End Of Transfer signal on channel 3. (PMU EOT are not signalled).

- **lock**

LOCK='1', mAgicV DMA transfers lock the AHB bus.

LOCK='0', AHB bus not locked by mAgicV.

7.1.2.3 DMA Shadow Registers

The registers listed in Table 7-5 below show the addresses that the DMA engine is generating for the current access. They are updated during the transfer and they are used for re-starting a new transfer from the last point ("preserve mode").

Table 7-5. DMA Shadow Registers

mAgicV Address	AHB Offset	Name	Type	Reset Value
0x6C	0x1B0	MGCDMACURRLEN	RO	NA
0x6D	0x1B4	MGCDMACURREXTADD	RO	NA
0x6E	0x1B8	MGCDMACURRINTADD	RO	NA

7.1.3 DMA Address Generation

When a DMA operation starts by writing the WRITE/READ field of MGCDMACTRL register or by FLOW code WRITEDMA/READDMA (see Section 5.2 "Sleep and Wakeup" on page 84), the channel parameters are copied in the DMA shadow registers.

reload mode:

```
mgcdmaextadd=> mgcdmaextcuradd
mgcdmaintseg[3:0] & mgcdmaintadd[15:0] => mgcdmacurintadd[19:0]
mgcdmalen => mgccurlen
```

preserve mode:

```
mgcdmalen => mgccurlen
```

At each AHB cycle the DMA engine performs the following operation:

```
External address = mgcdmacurextadd = (mgcdmacurext BITWISEAND
BITWISENOT(mgcdmaextcirc)) + ((mgcdmacurext  + mgcdmaextmod) BITWISEAND
mgcdmaextcirc)
Internal address = mgcdmacurrintadd  = (mgcdmacurint BITWISEAND
BITWISENOT(mgcdmaintcirc)) + ((mgcdmacurint + mgcdmaintmod) BITWISEAND
mgcdmaintcirc)
mgccurlen = mgccurlen - 1
EOT=1 when mgccurlen==0
```

8. Revision History

Doc. Rev.	Date	Comments
7011A	12/08	• Initial document release

9. Table of Contents

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
diopsis@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Requests
www.atmel.com/literature