

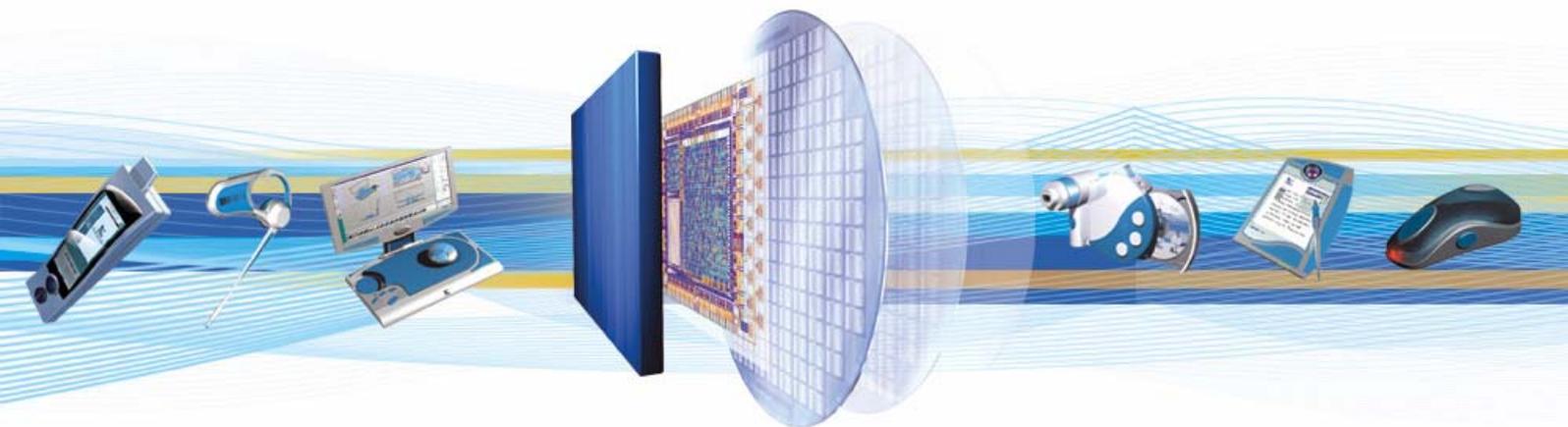


BlueCore™3-Multimedia

Kalimba DSP

User Guide

June 2005



CSR

Churchill House
Cambridge Business Park
Cowley Road
Cambridge CB4 0WZ
United Kingdom

Registered in England 3665875

Tel: +44 (0)1223 692000

Fax: +44 (0)1223 692001

www.csr.com

Contents

- 1 Introduction 5**
- 2 Key Features 6**
- 3 System Overview 7**
 - 3.1 Kalimba DSP Core 8
 - 3.2 Kalimba DSP Memory 8
 - 3.3 Kalimba DSP Peripherals 8
 - 3.3.1 Memory Management Unit Interface 8
 - 3.3.2 Programmable I/O Control 8
 - 3.3.3 Interrupt Control 8
 - 3.3.4 Clock Source Select and Timer 8
 - 3.3.5 Debug Interface 8
- 4 Kalimba DSP Core Architecture 9**
 - 4.1 Arithmetic Logic Unit 9
 - 4.2 Address Generators 9
 - 4.3 Registers 10
 - 4.4 Bank 1 Registers 10
 - 4.5 rFlags Register 11
 - 4.5.1 Negative Flag (N) 11
 - 4.5.2 Zero Flag (Z) 11
 - 4.5.3 Carry Flag (C) 11
 - 4.5.4 Overflow Flag (V) 11
 - 4.5.5 Sticky Overflow Flag (SV) 11
 - 4.5.6 User Definable Flag (UD) 12
 - 4.5.7 Bit Reverse Flag (BR) 12
 - 4.5.8 User Mode Flag (UM) 12
 - 4.5.9 Condition Codes 12
 - 4.6 rMAC Register 13
 - 4.7 Bank 2 Registers 14
 - 4.7.1 Index Registers 14
 - 4.7.2 Modify Registers 14
 - 4.7.3 Length Registers 14
 - 4.8 Instruction Decode 15
 - 4.8.1 Type A 15
 - 4.8.2 Type B 15
 - 4.8.3 Type C 15
 - 4.8.4 Special Cases 15
 - 4.9 Program Flow 16
 - 4.10 Debug 16
- 5 Memory Organisation 17**
 - 5.1 Memory Map 17
 - 5.1.1 PM Memory Map 17
 - 5.1.2 DM1 Memory Map 18
 - 5.1.3 DM2 Memory Map 18
- 6 Instruction Set Description 19**
 - 6.1 SUBTRACT and SUBTRACT With Borrow 21
 - 6.2 Bank1/2 Register Operations: ADD and SUBTRACT 22
 - 6.3 Logical Operations: AND, OR and XOR 23
 - 6.4 Shifter: LSHIFT and ASHIFT 24
 - 6.5 rMAC Move Operations 25
 - 6.6 Multiply: Signed 24-Bit Fractional and Integer 26
 - 6.7 MULTIPLY and ACCUMULATE (56-bit) 27

6.8	LOAD / STORE with Memory Offset.....	28
6.9	Sign Bits Detect and Block Sign Bits Detect.....	29
6.10	Divide Instruction.....	30
6.11	Program Flow: CALL, JUMP, RTS, RTI, SLEEP, DO...LOOP and BREAK.....	31
6.12	Indexed MEM_ACCESS_1 and MEM_ACCESS_2.....	32
7	Instruction Coding.....	33
7.1	Type A Instruction.....	34
7.2	Type B Instruction.....	34
7.3	Type C Instruction.....	34
7.4	Special Cases.....	34
7.5	OP_CODE Coding.....	35
7.6	AM Field.....	36
7.7	Carry Field (C Field).....	36
7.8	Bank 1/2 Register Select Field (B2RS Field).....	36
7.9	Saturation Select Field (V Field).....	36
7.10	Sign Select Field (S Field).....	37
7.11	k ₁₆ Coding for LSHIFT and ASHIFT.....	37
7.12	rMAC Sub Registers.....	37
7.13	ASHIFT.....	37
7.14	LSHIFT.....	38
7.15	k ₁₆ Coding Divide Instructions.....	38
8	Kalimba DSP Peripherals.....	39
8.1	MMU Interface.....	40
8.1.1	Read Ports.....	40
8.1.2	Write Ports.....	40
8.2	DSP Timers.....	40
8.3	Kalimba Interrupt Controller.....	41
8.3.1	DSP Core Functionality During Interrupt.....	41
8.3.2	Interrupt Controller Functionality.....	41
8.4	Generation of MCU Interrupt.....	42
8.5	PIO Control.....	42
8.6	MCU Memory Window in DM2.....	42
8.7	Flash Memory Window in DM2.....	42
8.8	PM Window in DM1.....	43
8.9	General Registers.....	43
8.10	Clock Rate Divider Control.....	43
8.11	Debugging.....	43
	Appendix A: Number Representation.....	44
	Appendix B: DSP Registers.....	46
	Appendix C: Software Examples.....	66
	Document References.....	72
	Acronyms and Definitions.....	73
	Record of Changes.....	74

List of Figures

Figure 3.1: Kalimba DSP Co-Processor Sub System.....	7
Figure 4.1: Kalimba DSP Core Base Architecture.....	9
Figure 4.2: rMAC Register	13
Figure 5.1: Memory Organisation.....	17
Figure 8.1: Kalimba DSP Peripheral Interfaces.....	39
Figure 8.2: Example of MMU Interface Usage for a Wireless MP3 Player	40

List of Tables

Table 4.1: Bank 1 Registers.....	10
Table 4.2: rFlags Register.....	11
Table 4.3: Condition Codes.....	12
Table 4.4: Bank 2 Registers.....	14
Table 5.1: PM Memory Map.....	17
Table 5.2: DM1 Memory Map.....	18
Table 5.3: DM2 Memory Map.....	18
Table 6.1: Notational Convention.....	19
Table 7.1: Instruction Coding Format.....	33
Table 7.2: OPCODE Coding Format.....	35
Table 7.3: AM Field.....	36
Table 7.4: C Field Options.....	36
Table 7.5: B2RS Field.....	36
Table 7.6: V Field.....	36
Table 7.7: S Field.....	37
Table 7.8: k_{16} Coding Shift Format.....	37
Table 7.9: rMAC Sub-Registers.....	37
Table 7.10: ASHIFT.....	37
Table 7.11: LSHIFT.....	38
Table 7.12: Divide Field.....	38
Table 7.13: Divide Field States.....	38

1 Introduction

The **BlueCore™3-Multimedia** Kalimba DSP User Guide is a user guide for developers of software applications and algorithms for the DSP co-processor on the BlueCore3-Multimedia device. It documents the architecture of the Kalimba DSP; instruction set mnemonics, peripheral features, and includes some example code. Read this document in conjunction with the other Kalimba tools documents that are available:

- BlueLab Kalimba DSP Assembler User Guide (CSR reference bc3-ug-002Pd) which covers the assembler software
- BlueLab xIDE User Guide (CSR reference blab-ug-002Pa) which covers xIDE, the software debugging tool

One of the features of the BlueCore3-Multimedia device is an on-chip DSP co-processor Kalimba. The Kalimba DSP particularly targets audio processing applications for BlueCore. The likely audio processing applications include:

- Sub-Band Coding (SBC) decoding and encoding, as defined in the Bluetooth Advanced Audio Distribution Profile
- MP3 decoding, as defined in ISO/IEC 11172-3, and the sample rate extensions defined in ISO/IEC 13818-3
- Advanced Audio Coding (AAC) decoding, as defined in ISO/IEC 13818-7
- Alternative voice/Hi-Fi CODECs
- Echo and noise cancellation

2 Key Features

The key features of the Kalimba DSP core include:

- 24-bit fixed point Kalimba DSP core
- 32MIPS performance which can be divided down for power saving
- 32-bit instruction word, dual 24-bit data memory:
 - 16-bit program address space with 4Kword (4K x 32-bits) of physical RAM
 - 16-bit data address space with 2 x 8Kword (8K x 24-bits) of physical RAM
- Up to two data memory accesses can be performed in the same cycle as a program memory read
- Single cycle 24 x 24-bit multiply with 56-bit accumulator
- Single cycle barrel shifter with 56-bit input and 24-bit output
- Multi-cycle divide (performed in the background)
- Majority of instructions can be conditional
- Zero overhead ring buffer indexing
- Zero overhead looping and branching
- Bit reversed addressing capability
- Largely orthogonal instruction set, which is quick to learn and easy to write in algebraic assembler language

The key features of the Kalimba DSP peripherals include:

- Close integration into the rest of the BlueCore3-Multimedia
- Four low overhead read/write ports to transfer streaming data to and from the BlueCore3-Multimedia subsystem
- Four shared memory mapped registers
- Two memory mapped windows into the BlueCore3-Multimedia MCU RAM for data exchange
- A window for access to the flash memory
- Multiple interrupt sources including two 24-bit timers
- Read and write access to external programmable I/O (PIO) lines

3 System Overview

The BlueCore3-Multimedia contains the Kalimba DSP shown in Figure 3.1 and consists of the following functional elements:

- Kalimba DSP core
- DSP memory, this RAM is used for:
 - Data memory
 - Program memory
 - Memory mapped I/O
- Memory management unit (MMU) interface
- Programmable I/O control
- Interrupt control
- Clock source
- Timer
- Debug interface

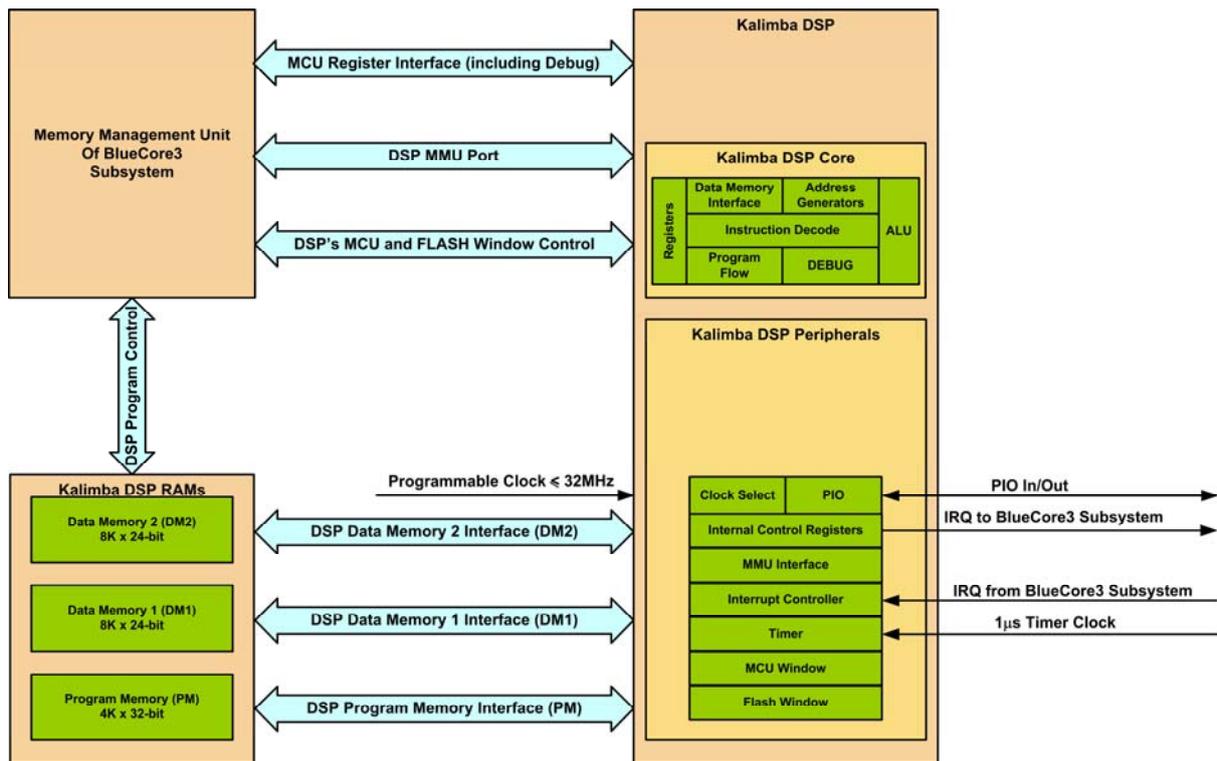


Figure 3.1: Kalimba DSP Co-Processor Sub System

3.1 Kalimba DSP Core

The Kalimba DSP core is an open platform DSP that can perform signal processing functions on over air data or CODEC data in order to enhance audio applications. Figure 3.1 and Section 8 shows how the DSP interfaces to other functional blocks within BlueCore3-Multimedia.

3.2 Kalimba DSP Memory

The Kalimba DSP contains on-chip RAM as follows:

- 8K x 24-bit for data memory 1 (DM1)
- 8K x 24-bit for data memory 2 (DM2)
- 4K x 32-bit for program memory (PM)

3.3 Kalimba DSP Peripherals

3.3.1 Memory Management Unit Interface

The MMU Interface consists of a series of virtual read and write ports that are used to stream transfers to/from the rest of the BlueCore3-Multimedia.

3.3.2 Programmable I/O Control

BlueCore3-Multimedia has 16 programmable I/O terminals (12 digital and 4 analogue/digital) controlled by firmware running on the device. The Kalimba DSP core can read any digital I/O directly but can only write to digital I/O that the MCU has enabled (this is done through your VM application). A full description of the I/O control is in Section 8.5.

3.3.3 Interrupt Control

The interrupt controller function within the Kalimba DSP covers interrupt control of the Kalimba DSP core. It allows interrupt sources selection and control of their priority setting within three levels. Alongside the interrupts caused by hardware, there are four software event interrupts available.

3.3.4 Clock Source Select and Timer

The Kalimba DSP consists of a clock source select interface, which is a clock-rate divider circuit that is controllable from both the Kalimba DSP core and the on board MCU. The Kalimba DSP also has two timers with a 1 μ s time base available.

3.3.5 Debug Interface

The BlueCore3-Multimedia contains hardware interface that assists in the debugging of applications running on the Kalimba DSP core.

4 Kalimba DSP Core Architecture

The Kalimba DSP core architecture is in Figure 4.1 below:

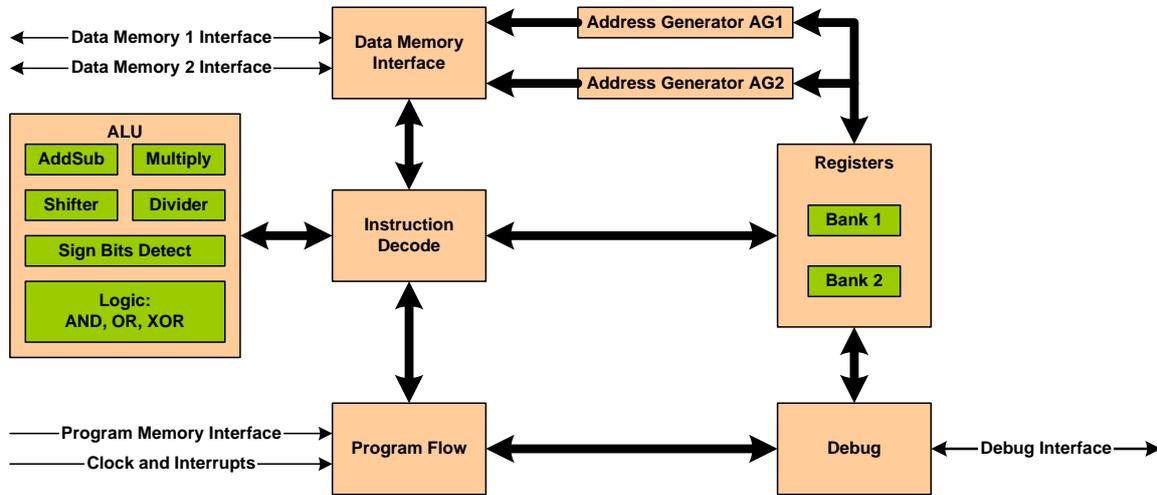


Figure 4.1: Kalimba DSP Core Base Architecture

4.1 Arithmetic Logic Unit

The arithmetic logic unit (ALU) performs the following functions:

- Add and subtract arithmetic
- Logic operations:
 - AND, OR, and XOR
- Single-cycle multiply, multiply/add and multiply/subtract
- Logical and arithmetic shift operations with a 56-bit input and 24-bit output
- Derive exponent and block derive exponent operations, which detect the number of redundant sign bits
- Signed divide, taking a 48-bit dividend (numerator) and a 24-bit divisor (denominator).⁽¹⁾

Note:

- ⁽¹⁾ This operation performed in the background taking 24 clock cycles

4.2 Address Generators

The address generators (AGs) form data memory addresses for when indexed memory reads or writes occur. Each AG has four associated address pointers (index registers). When an index register is used for a memory access, it is post-modified by a value in a specified modify register, or by a 2-bit constant. With two independent AGs, the DSP can generate two addresses simultaneously for dual indexed memory accesses.

Length values may be associated with four of the index registers to implement automatic modulo addressing for circular buffers.

When the appropriate mode bit is set in the rFlags register, the output of AG1 is bit-reversed then driven on to the address bus. This feature enables addressing in radix-2 Fast Fourier Transform (FFT) algorithms; see the Radix-2 FFT code in appendix C for an example of the usage of this feature.

4.3 Registers

There are two banks of 16 registers. Bank 1 are general registers described in Section 4.4 that can be used by virtually all instructions and Bank 2 registers are used for the control of index memory accesses described in Section 4.5.

4.4 Bank 1 Registers

Bank 1 registers are general registers used by virtually all instructions.

No.	Name	# bits	Description
0	Null	NA	Always read as zero, writing only affects flags (hence can be used for condition testing)
1	rMAC	56	The 56-bits are used for multiply accumulate instructions and the input to shift operations. For 24-bit operations: <ul style="list-style-type: none"> ▪ Read as bits [47:24] (with saturation and unbiased rounding)⁽¹⁾, ▪ Written as bits [47:24] with sign extension and trailing zero padding to make 56-bits
2	r0	24	General register
3	r1	24	General register
4	r2	24	General register
5	r3	24	General register
6	r4	24	General register
7	r5	24	General register
8	r6	24	General register
9	r7	24	General register
10	r8	24	General register
11	r9	24	General register
12	r10	24	General register and is used as the loop counter for zero overhead loops
13	rLink	16	Call instructions put the return PC address in this register for use by <code>rts</code> instructions ⁽²⁾
14	rFlags	16	Status and mode flags see below for a description
15	rIntLink	16	The return PC address is stored in this register for use by <code>rti</code> instructions

Table 4.1: Bank 1 Registers

Notes:

Only registers rMAC, r0-r5 can be used by indexed memory access instructions

16-bit registers (rLink, rFlags and rIntLink) are zero padded to 24-bit. Writing to them does not affect the flags register

All registers are set to 0 (zero) on DSP reset

⁽¹⁾ Unbiased rounding is as follows:

```
rMACrounded = rMAC[47:24] + rMAC[23];
if (rMAC[23:0] == 0x800000) then rMACrounded[0] = 0;
```

This has the effect of rounding odd rMAC[47:24] values away from zero and even rMAC[47:24] values towards zero, yielding a zero large sample bias assuming uniformly distributed values

⁽²⁾ There is no hardware stack, so to allow multi depth subroutine calls a software stack implementation or equivalent is required

4.5 rFlags Register

The rFlags register is a 16-bit register that is located in register bank 1 of Kalimba indicated in Table 4.1. The individual bits that make up the rFlags register and their value after reset are as shown in Table 4.2. Table 4.2 indicates that the rFlags register has a natural split into two bytes, the least significant byte contains the presently active flags used by Kalimba. The most significant byte contains a stored value of the flags. The flags are stored when an interrupt has occurred and then restored back to these values after Kalimba has finished servicing the interrupt. Interrupts are serviced three instructions after the interrupt request line goes high (not including prefix instructions). The INT_ versions of the various flags are the copies that are stored at the point of interrupt service. The `rti` instruction then automatically restores the flags to their previous value.

Name	INT_UM_FLAG	INT_BR_FLAG	INT_SV_FLAG	INT_UD_FLAG	INT_V_FLAG	INT_C_FLAG	INT_Z_FLAG	INT_N_FLAG	UM_FLAG	BR_FLAG	SV_FLAG	UD_FLAG	V_FLAG	C_FLAG	Z_FLAG	N_FLAG
Reset State	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 4.2: rFlags Register

4.5.1 Negative Flag (N)

Set if the result of the instruction is negative (most significant bit set), and cleared otherwise.

4.5.2 Zero Flag (Z)

Set if the result of the instruction is zero, and cleared otherwise.

4.5.3 Carry Flag (C)

The state of the carry flag and interrupt carry flag is as follows:

- For an addition, C is set if the addition produced a carry (that is, an unsigned overflow), and is cleared otherwise
- For a subtraction, C is cleared if the subtraction produced a borrow (that is, an unsigned underflow), and is set otherwise
- For other operations (including multiply accumulate), C is left unchanged

4.5.4 Overflow Flag (V)

The state of the overflow flag and interrupt overflow flag is as follows:

- For addition, subtraction, arithmetic shifts, integer multiplies, and multiply accumulates, V is set if signed overflow occurred, regarding the operands and result as two's complement signed integers, and is cleared otherwise
- The setting/clearing of the V flag for the rMAC register occurs if there is overflow past the 56th bit, whereas for the 24-bit registers it is if overflow occurs past the 24th bit. When writing to the 16-bit registers overflow has limited meaning and so the V flag remains unchanged
- For other operations, V is left unchanged

4.5.5 Sticky Overflow Flag (SV)

Set whenever the V flag is set but can only be cleared in software by explicitly writing to the rFlags register.

4.5.6 User Definable Flag (UD)

A special 'USERDEF' condition code is TRUE if this flag is set and FALSE if this flag is clear. Use it in the code sections to improve speed and code clarity where optionally a particular instruction needs to execute.

4.5.7 Bit Reverse Flag (BR)

If set, the output of Address Generator 1 (index registers I0-3) is bit-reversed before being driven to the address bus.

4.5.8 User Mode Flag (UM)

If set, interrupts will be serviced. On entry to the interrupt service routine (PC address 0x0002), this flag is cleared and so no further interrupt will be serviced unless the flag is manually set for example to support interrupt priority. Execution of a return from interrupt (`rti`) instruction will set this flag to the value of INT_UM_FLAG (normally set unless altered in software). During an interrupt service routine, the INT_UM_FLAG contains the stored value of the user mode flag prior to the interrupt.

4.5.9 Condition Codes

The state of the flags present in the rFlags register form the basis of the condition codes in Table 4.3 for the Kalimba DSP.

Condition	Condition Flag State	Condition Code
Z (Zero) / EQ (Equal)	Z = 1	0 0 0 0
NZ (Not Zero) / NE (Not equal)	Z = 0	0 0 0 1
C (ALU carry)	C = 1	0 0 1 0
NC (Not ALU carry)	C = 0	0 0 1 1
NEG (Negative)	N = 1	0 1 0 0
POS (Positive)	N = 0	0 1 0 1
V (ALU overflow)	V = 1	0 1 1 0
NV (Not ALU overflow)	V = 0	0 1 1 1
HI (unsigned higher)	C = 1 AND Z = 0	1 0 0 0
LS (unsigned lower or same)	C = 0 OR Z = 1	1 0 0 1
GE (Signed Greater than or equal)	N = V	1 0 1 0
LT (Signed Less than)	N != V	1 0 1 1
GT (Signed Greater than)	Z = 0 AND N = V	1 1 0 0
LE (Signed Less than or equal)	Z = 1 OR N != V	1 1 0 1
USERDEF (user defined)	USERDEF = 1	1 1 1 0
Always true	don't care	1 1 1 1

Table 4.3: Condition Codes

4.6 rMAC Register

The rMAC register is a 56-bit register that is located in register bank 1 of Kalimba indicated in Table 4.1. The rMAC register splits into a set of separately accessible sub registers, listed below is the size of these registers and shown in Figure 4.2:

- rMAC is the overall 56-bit register
- rMAC0 is a 24-bit register that forms the lower part of the rMAC register
- rMAC1 is a 24-bit register that forms the middle part of the rMAC register
- rMAC2 is a 8-bit register that forms the higher part of the rMAC register
- rMAC12 is a 32-bit register that is a combination of rMAC0 and rMAC1 that forms part of the rMAC register

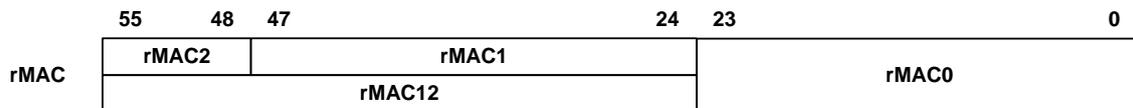


Figure 4.2: rMAC Register

4.7 Bank 2 Registers

Bank 2 registers are for the control of index memory accesses, and set up modulo addressing through a circular memory buffer.

No.	Name	# bits	Description
0	I0	16	Index register for Address Generator 1 (AG1)
1	I1	16	Index register for AG1
2	I2	16	Index register for AG1
3	I3	16	Index register for AG1
4	I4	16	Index register for Address Generator 2 (AG2)
5	I5	16	Index register for AG2
6	I6	16	Index register for AG2
7	I7	16	Index register for AG2
8	M0	16	Modify register for any index register
9	M1	16	Modify register for any index register
10	M2	16	Modify register for any index register
11	M3	16	Modify register for any index register
12	L0	16	Length register for Index register I0
13	L1	16	Length register for Index register I1
14	L4	16	Length register for Index register I4
15	L5	16	Length register for Index register I5

Table 4.4: Bank 2 Registers

Note:

All Bank 2 registers are sign extended to 24-bit for arithmetic operations

All registers are set to 0 (zero) on DSP reset

4.7.1 Index Registers

The index register contains the address pointers to data memory used with indexed addressing. These registers allow transfer of data to and from selected Bank1 registers using the address contained with this register see Section 6.13 for further details. The index registers I0 to I3 are associated with AG1 and I4 to I7 are associated with AG2.

4.7.2 Modify Registers

When an index register is used for a memory access, it can be post-modified by a value contained in the modify register, or by a 2-bit constant.

4.7.3 Length Registers

Length register values are associated with four of the index registers listed in Table 4.4 to implement automatic modulo addressing for circular buffers. To disable automatic modulo addressing set the corresponding length register to zero.

4.8 Instruction Decode

There are three basic instruction types these are Type A, Type B and Type C, alongside special cases.

4.8.1 Type A

A Type A instruction has the following syntax:

```
<if cond> RegC = RegA OP(1) RegB <MEM_ACCESS_1>;
```

Type A is a conditional instruction, with an additional single memory read or write operation. The instruction can accept two input operands and one output operand (which may be different). Operands can be any of the 16 Bank1 registers (Load/Store instructions permit the usage of Bank2 registers). For the memory access, index registers I0-I3 select the address, the destination or source register can be any of the first eight Bank1 registers: rMAC, r0 - r5. At the end of the instruction the index register is post modified by one of the modify registers M0-M3.

4.8.2 Type B

A Type B instruction has the following syntax:

```
RegC = RegA OP(1) constant;
```

Type B is a non-conditional instruction similar to Type A, but with one of the operands being a 16-bit constant stored in the instruction word. To use a 24-bit constant prefix the Type B instruction with the prefix (PFX) instruction. No additional memory access operation permitted. The PFX instruction is automatic if needed by the assembler **kalasm2**.

4.8.3 Type C

A Type C instruction has the following syntax:

```
RegC = RegC OP(1) RegA <MEM_ACCESS_1> <MEM_ACCESS_1>;
```

Type C is a non-conditional instruction similar to a Type A, except that one of the input operands is also the output operand. In addition, two memory reads or writes may occur in the same clock cycle; one memory access uses Address Generator 1 (index registers I0-I3) and the other uses Address Generator 2 (index registers I4-I7). The index registers can be either post modified by the M0-M3 registers or by a 2-bit signed modify constant (valid values: -1, 0, 1, or 2).

4.8.4 Special Cases

Program flow instructions such as `jump`, `call`, `rts`, etc; use a slight variation on the above types, as they can always be conditional. Also the multiply accumulate instructions all write their result to the rMAC register.

Note:

⁽¹⁾ OP refers to operation which can include Addition, Subtraction, Multiplication, OR, AND, Exclusive OR

4.9 Program Flow

The program counter (PC) supplies addresses to the program memory. An instruction register holds the currently executing instruction. This instruction register introduces a single level of pipelining into the program flow. During one clock cycle, the instruction register has instructions fetched and loaded into it and during the following cycle are executed. To allow zero overhead branching and no pipeline hazards with memory reads the processor also acts on the direct signal from the memory, i.e. the next instruction executed, to set up the address bus and control lines for a memory read.

Hence, there are no stall cycles in the following code examples:

- Zero overhead branching:


```
r0 = r0 - 1;
if POS jump dont_add_ten;
    r0 = r0 + 10;
dont_add_ten:
```
- No pipeline hazards:


```
r0 = 100;           // memory location 100 will be read. i.e. the new
r1 = M[r0];        // value of r0 is fed though to the DM address bus
```

Another feature of the program flow of the Kalimba DSP is the hardware zero overhead looping instruction. The register r10 is loaded with the number of times that the code between the DO instruction and the 'loop' label executes. r10 is then automatically decremented and a jump taken (if needed) at the same time as executing the instruction before the 'loop' label. For example:

- Zero overhead looping:


```
r10 = 10;
DO loop;           // copies 10 words of data from
    r0 = M[I0,1];  // address I0 to address I2.
    M[I2,1] = r0;  // Takes 22 cycles in total.
loop:
```

4.10 Debug

There is debugging hardware in the Kalimba DSP used by the debugger, **xIDE**. It provides the following features:

- Reset, Run, Stop, Step
- Setting and reading of the program counter (PC)
- Program breakpoint
- Data memory breakpoint (read, write, or read/write)
- Instruction Break
- Read/write of register values
- External to the DSP core itself there is debug circuitry to read/write memory locations as seen by the DSP on either of its 3 memory buses PM, DM1 and DM2

5 Memory Organisation

The Kalimba DSP core has two 24-bit data memory banks, DM1 and DM2, each having a 15-bit address space, and a single 32-bit program memory (PM) having a 16-bit address space. Accessing of all three memories simultaneously is possible in the same clock cycle, assuming no conflicts; this is a three bank Harvard architecture. Conflicts introduce an appropriate number of wait cycles. Figure 5.1 is a view of the Kalimba DSP memory organisation containing size information and peripheral memory mapping.

BlueCore3-Multimedia has the following physical RAM for the Kalimba DSP:

- DM1 = 8K x 24-bit
- DM2 = 8K x 24-bit
- PM = 4K x 32-bit

The BlueCore3-Multimedia subsystem MCU initialises the Kalimba DSP. During initialisation the Kalimba DSP memory maps into the MCU memory map, to enable program and data coefficient download. The MCU then sets the initial clock frequency for the Kalimba DSP to use before starting it running. API calls from the virtual machine (VM) running on the MCU invokes the program download, and Kalimba DSP initialisation.

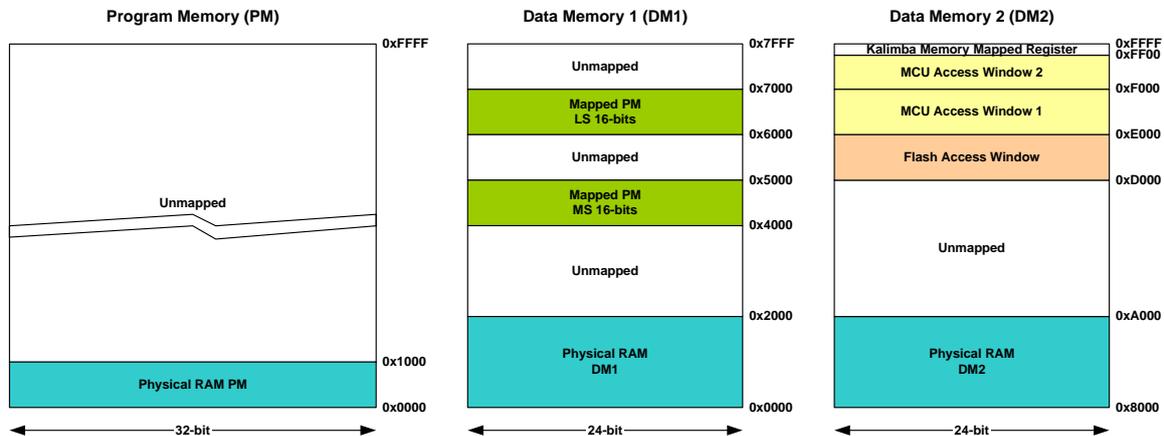


Figure 5.1: Memory Organisation

5.1 Memory Map

The memory organisation shown in Figure 5.1 can be broken down into their individual memory maps as depicted in the tables listed in Section 5.1.1 to Section 5.1.3 (and Appendix B Table 1 for the DSP memory map).

5.1.1 PM Memory Map

The program memory map for Kalimba shown in Table 5.1 contains 4Kwords (4K x 32-bits) of physical program memory RAM, the remaining 60Kwords are available for future variants.

Data		Length (Words)	Description
Start	End		
0x0000	0x0FFF	4K	General purpose RAM (program memory)
0x1000	0xFFFF	60K	Available for RAM expansion in future variants

Table 5.1: PM Memory Map

5.1.2 DM1 Memory Map

The first data memory bank DM1 has a memory map shown in Table 5.2 containing:

- 8Kwords (8K x 24-bits) of data memory for the DSP
- A window of 4Kwords of program memory split into the most and least significant halves
- The remaining areas of the memory map allow for future variants

Data		Length (Words)	Description
Start	End		
0x0000	0x1FFF	8K	General purpose RAM (data memory 1)
0x2000	0x3FFF	8K	Available for RAM expansion in future variants
0x4000	0x4FFF	4K	Mapped to program memory MS-16bits
0x5000	0x5FFF	4K	Available for RAM expansion in future variants
0x6000	0x6FFF	4K	Mapped to program memory LS-16bits
0x7000	0x7FFF	4K	Available for RAM expansion in future variants

Table 5.2: DM1 Memory Map

5.1.3 DM2 Memory Map

The second data memory bank DM2 memory map shown in Table 5.3 has distinct areas, these are:

- 8Kwords (8K x 24-bits) of general purpose data RAM for the DSP
- A 4Kwords window into the flash memory, which could be used by the DSP for items such as slower access coefficient tables
- There are two MCU windows 4K and 3.75Kwords into the MCU memory to allow for control information and message passing
- 16Kwords of memory map for future variants
- The final 256words of the memory map are reserved for the memory mapped I/O for the DSP that is explained further in Appendix B: DSP Registers.

Data		Length (Words)	Description
Start	End		
0x8000	0x9FFF	8K	General purpose RAM (data memory 2)
0xA000	0xCFFF	16K	Available for RAM expansion
0xD000	0xDFFF	4K	Flash window
0xE000	0xEFFF	4K	Window into MCU memory 1
0xF000	0xFEFF	4K-256	Window into MCU memory 2
0xFF00	0xFFFF	256	DSP Memory mapped I/O registers (See Section Appendix B: DSP Registers for details)

Table 5.3: DM2 Memory Map

6 Instruction Set Description

The instruction set for the BlueCore3-Multimedia is an algebraic assembler instruction set compared to the mnemonic assemblers found in traditional microcontrollers. Algebraic assemblers suit the architecture of a DSP, as it is able to express complex and parallel instructions in an understandable way. This section describes each instruction in more detail. Table 6.1 outlines the notation conventions that used in describing the syntax:

Parallel lines	Vertical parallel bars enclose lists of syntax options. One of the choices listed must be chosen.
Angled brackets < <i>non bold italics</i> >	Anything in <i>non bold italics</i> enclosed by angled brackets is an optional part of the instruction statement.
A, B, C	Denotes a register operand. By default, the register must be chosen from the list of Bank1 registers. If subscribed with 'Bank1/2' then the register can be chosen from either Bank1 or Bank2
k_{16}, k_8	Denotes a constant, the subscripted number being the size of the number in bits.
M[x]	Means the data in the memory location with address 'x'.
M[i,m]	Means the data in the memory location with address 'i', and that after the read/write the value of the register 'i' is modified according to the equation: $i=i+m$;
cond	A condition code from Table 4.3, e.g. NZ
MEM_ACCESS_1 MEM_ACCESS_2	Represents a memory access instruction that can be appended to an instruction where indicated. The valid memory access instructions that can be appended are covered by the Indexed MEM_ACCESS_1/2 in Section 6.13

Table 6.1: Notational Convention

Important Note:

The Kalimba DSP architecture has been designed to carry out single cycle instructions, any exceptions from this are noted in the description for the instruction, e.g. the divide instruction. If the cycle time is not stated, it is a single cycle instruction. Any other exceptions are also noted in the description for each instruction.

6.1 ADD and ADD with CARRY

Syntax:

Type A:	<i><if cond></i>	C = A + B C = A + M[B] C = M[A] + B M[C] = A + B ^(note 1)	<+Carry> <MEM_ACCESS_1>;
----------------	------------------------	---	-----------------------------

^(note 1) Conditional memory writes are not supported on BlueCore3-Multimedia

Example: if Z r3 = r1 + M[r2] + Carry r4 = M[i0, m0];

Type B:	C = A + k ₁₆ C = A + M[k ₁₆] C = M[A] + k ₁₆ M[k ₁₆] = A + C	<+Carry>;
----------------	---	-----------

Example: r3 = M[r1] + 10+ Carry;

Type C:	C = C + A C = C + M[A]	<+Carry> <MEM_ACCESS_1> <MEM_ACCESS_2>;
----------------	---------------------------	---

Example: r3 = r3 + M[r1] + Carry
 r4 = M[I0, M0]
 r5 = M[I4, M1];

Description:

Test the optional condition and, if TRUE, perform the addition. If the condition is FALSE, perform a no-operation (NOP) but *MEM_ACCESS_1* still carried out. Omitting the condition performs the addition unconditionally. The addition operation adds the first source operand to the second source operand and, if designated by the “+Carry” notation, adds the ALU carry bit, C. The result is stored in the destination operand. The operands may be either one of the 16 Bank1 registers, a 16-bit sign extended constant (24-bit with prefix instruction), or memory pointed to by the register or constant.

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Set if an arithmetic overflow occurs and cleared otherwise	C	Set if a carry is generated and cleared otherwise

Note:

If one of the source operands is Null then a load/store assumed. Therefore, the C and V flags are unchanged.

If all operands are Null then a NOP assumed. Therefore, all flags are unchanged.

When writing to rLink, rIntLink or rFlags all flags are affected

6.2 SUBTRACT and SUBTRACT With Borrow

Syntax:

Type A: $\langle \text{if cond} \rangle$ $\left\{ \begin{array}{l} C = A - B \\ C = A - M[B] \\ C = M[A] - B \\ M[C] = A - B \text{ (note 1)} \end{array} \right. \langle \text{-Borrow} \rangle \langle \text{MEM_ACCESS_1} \rangle;$

(note 1) Conditional memory writes are not supported on BlueCore3-Multimedia

Example: `if Z r3 = r1 - r2 - Borrow r4 = M[i0, m0];`

Type B: $\left\{ \begin{array}{l} C = A - k_{16} \\ C = A - M[k_{16}] \\ C = M[A] - k_{16} \\ M[k_{16}] = A - C \end{array} \right. \langle \text{-Borrow} \rangle;$

Example: `r3 = M[r1] - 10- Borrow;`

Type C1/2: $\left\{ \begin{array}{l} C = C - A \\ C = C - M[A] \end{array} \right. \langle \text{-Borrow} \rangle \langle \text{MEM_ACCESS_1} \rangle \langle \text{MEM_ACCESS_2} \rangle;$

Example: `r3 = r3 - r1 - Borrow
r4 = M[I0, M0]
r5 = M[I4, M1];`

Description:

Test the optional condition and, if TRUE, perform the subtraction. If the condition is FALSE, perform a NOP, but *MEM_ACCESS_1* still carried out. Omitting the condition performs the subtraction unconditionally. The subtraction operation subtracts the second source operand from the first source operand and optionally, if designated by the “- Borrow” notation, subtracts the inverse of the ALU carry bit, C. The result is stored in the destination operand. The operands may be either one of the 16 Bank1 registers, a 16-bit sign extended constant (24-bit with prefix instruction), or memory pointed to by the register or constant.

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Set if an arithmetic overflow occurs and cleared otherwise	C	Set if a borrow is generated and set otherwise

Note:

If one of the source operands is Null then a load/store or negate assumed. The C and V flags are left unchanged

When writing to rLink, rIntLink or rFlags all flags are affected

6.3 Bank1/2 Register Operations: ADD and SUBTRACT

Syntax:

Type A: $\langle \text{if cond} \rangle \left\{ \begin{array}{l} C_{\text{BANK1/2}} = A_{\text{BANK1/2}} + B_{\text{BANK1/2}} \\ C_{\text{BANK1/2}} = A_{\text{BANK1/2}} - B_{\text{BANK1/2}} \end{array} \right\} \langle \text{MEM_ACCESS_1} \rangle;$

Example: `if Z i0 = i4 + r2 r1 = M[i1, M1];`

Type B: $\left\{ \begin{array}{l} C_{\text{BANK1/2}} = A_{\text{BANK1/2}} + K_{16} \\ C_{\text{BANK1/2}} = A_{\text{BANK1/2}} - K_{16} \end{array} \right\};$

Example: `i0 = r2 + 5;`

Type C1/2: $\left\{ \begin{array}{l} C_{\text{BANK1/2}} = C_{\text{BANK1/2}} + A_{\text{BANK1/2}} \\ C_{\text{BANK1/2}} = C_{\text{BANK1/2}} - A_{\text{BANK1/2}} \end{array} \right\} \langle \text{MEM_ACCESS_1} \rangle \langle \text{MEM_ACCESS_2} \rangle;$

Example: `r2 = r2 + i2
r0 = M[I0, M0]
r1 = M[I4, M1];`

Description:

Test the optional condition and, if TRUE, perform the specified cross-bank addition or subtraction. If the condition is FALSE, perform a NOP, but *MEM_ACCESS_1* still carried out. Omitting the condition performs the addition or subtraction unconditionally. The operands may be either one of the 16 Bank1 or 16 Bank2 registers or a 16-bit sign extended constant (24-bit with prefix instruction).

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Set if an arithmetic overflow occurs and cleared otherwise	C	For addition: set if a carry generated. For subtraction: cleared if a borrow is generated.

Notes:

If one of the source operands is Null then a load/store assumed. The C and V flags are left unchanged
If the destination register is from bank2 (i.e. 16-bit) then the C and V flags are left unchanged

6.4 Logical Operations: AND, OR and XOR

Syntax:

Type A: $\langle \text{if cond} \rangle \left\{ \begin{array}{l} C = A \text{ AND } B \\ C = A \text{ OR } B \\ C = A \text{ XOR } B \end{array} \right. \langle \text{MEM_ACCESS_1} \rangle;$

Example: `if Z r3 = r1 AND r2 r0 = M[I0,M0];`

Type B: $\left\{ \begin{array}{l} C = A \text{ AND } k_{16} \\ C = A \text{ OR } k_{16} \\ C = A \text{ XOR } k_{16} \end{array} \right. ;$

Example: `r3 = r1 XOR 10;`

Type C1/2: $\left\{ \begin{array}{l} C = C \text{ AND } A \\ C = C \text{ OR } A \\ C = C \text{ XOR } A \end{array} \right. \langle \text{MEM_ACCESS_1} \rangle \langle \text{MEM_ACCESS_2} \rangle;$

Example: `r3 = r3 OR r1
r0 = M[I0,M0]
r2 = M[I4,M1];`

Description:

Test the optional condition and, if TRUE, perform the specified bit wise logical operation (logical AND, OR, or XOR). If the condition is FALSE, perform a NOP, but *MEM_ACCESS_1* still carried out. Omitting the condition performs the operation unconditionally. The operands may be either one of the 16 Bank1 registers or a 16-bit sign extended constant (24-bit with prefix instruction).

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Left unchanged	C	Left unchanged

6.5 Shifter: LSHIFT and ASHIFT

Syntax:

Type A: $\langle \text{if cond} \rangle \mid C = A \text{ OP } B \mid \langle \text{MEM_ACCESS_1} \rangle;$
Example: `if Z r3 = r2 LSHIFT r1 r0 = M[I0,M0];`

Type B:

$C = A \text{ OP } k_7$ $rMAC0 = A \text{ OP } k_7$ $rMAC12 = A \text{ OP } k_7$ $rMAC2 = A \text{ OP } k_7$ $rMAC = A \text{ OP } k_7 \text{ (LO)}$ $rMAC = A \text{ OP } k_7 \text{ (MI)}$ $rMAC = A \text{ OP } k_7 \text{ (HI)}$	$;$
--	-----

Example: `rMAC0 = r2 LSHIFT 4;`

:

Type C1/2: $C = C \text{ OP } A \mid \langle \text{MEM_ACCESS_1} \rangle \mid \langle \text{MEM_ACCESS_2} \rangle;$
Example: `r2 = r2 ASHIFT r7`
`r5 = M[I0,M2]`
`r1 = M[I4,M1];`

Description:

Test the optional condition and, if TRUE, perform the specified shift operation (arithmetic or logical). If the condition is FALSE, perform a NOP, but *MEM_ACCESS_1* still carried out. Omitting the condition performs the shift unconditionally. A positive number causes a shifting to the left and a negative number causes a shifting to the right. For an arithmetic shift to the right sign extension bits added as needed. If overflow occurs in an arithmetic shift, i.e. non-sign bits being shifted out, then the overflow flag is set and the result is saturated to $2^{23}-1$ or -2^{23} depending on the sign of the input. No rounding occurs for ASHIFT or LSHIFT. The operands may be either one of the 16 Bank1 registers or a constant specified in the instruction.

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	ASHIFT: Set if an arithmetic overflow occurs and cleared otherwise	C	Left unchanged
	LSHIFT: Left unchanged		

Note:

OP is either ASHIFT (arithmetic) or LSHIFT (logical)

If *rMAC* is the source operand, the full 56-bits are used as input to the shifter

For Type B instructions the destination operand can be:

- Either, *rMAC0*, *rMAC12*, or *rMAC2*, causing the other bits of *rMAC* to be unaffected (writing to *rMAC12* writes the data into *rMAC1* and causes sign extension (ASHIFT) or zero fill (LSHIFT) into *rMAC2*)
- Or, *rMAC*, with a data format tag (LO, MI, HI) to select which word of *rMAC* the 24-bit result from the shifter should be written to, the other bits of *rMAC* are sign-extended / zero-padded as appropriate

6.6 rMAC Move Operations

Syntax:

$$\text{Type B: } \begin{array}{|c|} \hline \text{rMAC0} \\ \hline \text{rMAC2} \\ \hline \end{array} = \begin{array}{|c|} \hline \text{rMAC0} \\ \hline \text{rMAC1} \\ \hline \text{rMAC2} \\ \hline \text{A} \\ \hline \end{array};$$

Example: $\text{rMAC0} = \text{rMAC1};$

$$\text{rMAC12} = \begin{array}{|c|} \hline \text{rMAC0} \\ \hline \text{rMAC1} \\ \hline \text{rMAC2} \\ \hline \text{A} \\ \hline \end{array} \begin{array}{|c|} \hline (\text{SE}) \\ \hline (\text{ZP}) \\ \hline \end{array};$$

Example: $\text{rMAC12} = \text{rMAC0} (\text{SE});$

$$\text{C} = \begin{array}{|c|} \hline \text{rMAC0} \\ \hline \text{rMAC1} \\ \hline \end{array};$$

Example: $\text{r3} = \text{rMAC0};$

$$\text{C} = \begin{array}{|c|} \hline \text{rMAC2} \\ \hline \end{array} \begin{array}{|c|} \hline (\text{SE}) \\ \hline (\text{ZP}) \\ \hline \end{array};$$

Example: $\text{r3} = \text{rMAC2} (\text{ZP});$

Description:

These are move instructions, implemented as a special case of LSHIFT and ASHIFT, to support loading and reading of the individual sections of the rMAC register (rMAC2, rMAC1, and rMAC0) shown in Figure 4.2. When writing to rMAC1, the data is either sign extended or zero padded into rMAC2. The format specifiers, SE and ZP, are required to specify how rMAC2 filled.

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Left unchanged	C	Left unchanged

6.7 Multiply: Signed 24-Bit Fractional and Integer

Syntax:

Type A: $\langle if\ cond \rangle \left| C = A * B \right| \begin{array}{l} (frac) \\ (int) \end{array} \left| \langle (sat) \rangle \quad \langle MEM_ACCESS_1 \rangle;$

Example: `if Z r3 = r2 * r1 (int) (sat) r5 = M[I0,M2];`

Type B: $\left| C = A * k_{16} \right| \begin{array}{l} (frac) \\ (int) \end{array} \left| \langle (sat) \rangle;$

Example: `r6 = r2 * 0.34375 (frac);`

Type C1/2: $\left| C = C * B \right| \begin{array}{l} (frac) \\ (int) \end{array} \left| \langle (sat) \rangle \quad \langle MEM_ACCESS_1 \rangle \quad \langle MEM_ACCESS_2 \rangle;$

Example: `r2 = r2 * r7 (int) (sat)
r0 = M[I0,1]
r1 = M[I4,-1];`

Description:

Test the optional condition and, if TRUE, perform the specified multiply operation (fractional or integer). If the condition is FALSE, perform a NOP, but *MEM_ACCESS_1* still carried out. Omitting the condition performs the operation unconditionally. A fractional multiply, (*frac*), treats the source and destination operands as fractional numbers with 2^{23} representing +1 and -2^{23} representing -1. An integer multiply, (*int*), treats the source and destination operands as integer numbers. Optionally, if designated by the (*sat*) notation, the result of an integer multiply is saturated if overflow occurs. Unbiased rounding is always done for a fractional multiply operation. The operands may be either one of the 16 Bank1 registers or a 16-bit left justified constant (24-bit with prefix instruction). See Appendix A on number representation for information on multiply operations

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	frac: Left unchanged	C	Left unchanged
	int: Set if an arithmetic signed overflow occurs and cleared otherwise		

Note:

A saturated fractional multiplication has no significance therefore (*sat*) is not an option with (*frac*)

The 16-bit constant is left justified to 24-bits by adding 8 zeros as the LSBs. This allows the 16-bit constant to represent fixed point fractional numbers between +1 and -1.

Unbiased rounding is as follows:

```
rMACrounded = rMAC[47:24] + rMAC[23];
if (rMAC[23:0] == 0x800000) then rMACrounded[0] = 0;
```

This has the effect of rounding odd rMAC[47:24] values away from zero and even rMAC[47:24] values towards zero, yielding a zero large sample bias assuming uniformly distributed values

6.8 MULTIPLY and ACCUMULATE (56-bit)

Syntax:

Type A: $\langle \text{if cond} \rangle \left[\begin{array}{l} \text{rMAC} = A * B \\ \text{rMAC} = \text{rMAC} + A * B \\ \text{rMAC} = \text{rMAC} - A * B \end{array} \right] \left[\begin{array}{l} \langle (SS) \rangle \\ \langle (SU) \rangle \\ \langle (US) \rangle \\ \langle (UU) \rangle \end{array} \right] \langle \text{MEM_ACCESS_1} \rangle ;$

Example: `if Z rMAC = rMAC + r1*r2 (SS)
r5 = M[I0,M0];`

Type B: $\left[\begin{array}{l} \text{rMAC} = A * k_{16} \\ \text{rMAC} = \text{rMAC} + A * k_{16} \\ \text{rMAC} = \text{rMAC} - A * k_{16} \end{array} \right] \left[\begin{array}{l} \langle (SS) \rangle \\ \langle (SU) \rangle \\ \langle (US) \rangle \\ \langle (UU) \rangle \end{array} \right] ;$

Example: `rMAC = rMAC + r1 * 0.24254 (SS);`

Type C1/2: $\left[\begin{array}{l} \text{rMAC} = C * A \\ \text{rMAC} = \text{rMAC} + C * A \\ \text{rMAC} = \text{rMAC} - C * A \end{array} \right] \left[\begin{array}{l} \langle (SS) \rangle \\ \langle (SU) \rangle \\ \langle (US) \rangle \\ \langle (UU) \rangle \end{array} \right] \langle \text{MEM_ACCESS_1} \rangle \langle \text{MEM_ACCESS_2} \rangle ;$

Example: `rMAC = rMAC - r3 * r1 (SS)
r2 = M[I0,1]
r1 = M[I4,-1];`

Description:

Test the optional condition and, if TRUE, perform the specified multiply/accumulate. If the condition is FALSE, perform a NOP, but *MEM_ACCESS_1* still carried out. Omitting the condition performs the operation unconditionally. The data format field to the right of the operands specifies whether each respective operand is in signed (S) or unsigned (U) format. The effective binary point is between bits 47 and 46. The operands may be either one of the 16 Bank1 registers or a 16-bit left justified constant (24-bit with prefix instruction). See Appendix A on number representation for information on multiply operations

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Set if overflow occurs past the 56 th bit and cleared otherwise	C	Left unchanged

Note:

Where *(SS)* is the default if no data format is specified

The 16-bit constant is left justified to 24-bits by adding 8 zeros as the LSBs. This allows the 16-bit constant to represent fixed point fractional numbers between +1 and -1.

To get the result of the equivalent integer multiplication, the result should be shifted to the right by 1-bit.

6.9 LOAD / STORE with Memory Offset

Syntax:

Type A: $\langle \text{if cond} \rangle \left\{ \begin{array}{l} C = M[A + B] \\ M[C + A] = B \end{array} \right. \langle \text{MEM_ACCESS_1} \rangle;$

(note 1) Conditional memory writes are not supported on BlueCore3-Multimedia

Example: `if Z r3 = M[r1 + r2]
r4 = M[I0, M0];`

Type B: $\left\{ \begin{array}{l} C = M[A + k_{16}] \\ M[C + k_{16}] = A \end{array} \right. ;$

Example: `M[r3 + 6] = r1;`

Type C: $\left\{ \begin{array}{l} C = M[C + A] \end{array} \right. \langle \text{MEM_ACCESS_1} \rangle \langle \text{MEM_ACCESS_2} \rangle;$

Example: `r3 = M[r3 + r2]
r4 = M[I0, 1]
r5 = M[I4, -1];`

Description:

Test the optional condition and, if TRUE, perform the specified load/store, including memory offset. If the condition is FALSE, perform a NOP, but *MEM_ACCESS_1* still carried out. Omitting the condition performs the load/store unconditionally. The operands may be either one of the 16 Bank1 registers or a 16-bit constant specified in the instruction.

Flags Generated:

Z	Set if the result operand equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Left unchanged	C	Left unchanged

6.10 Sign Bits Detect and Block Sign Bits Detect

Syntax:

Type A: `<if cond> | C = SIGNDET A | <MEM_ACCESS_1>;`
Example: `if Z r3 = SIGNDET rMAC`
`r4 = M[I0, M0];`

Type C: `C = BLKSIGNDET A | <MEM_ACCESS_1> <MEM_ACCESS_2>;`
Example: `r3 = BLKSIGNDET r1`
`r4 = M[I0, 1]`
`r5 = M[I4, -1];`

Description:

SIGNDET returns the number of redundant sign bits of the source operand. For example:

0000 1101 0101 0101 1100 1111	-	has 3 redundant sign bits
1001 0101 0101 0100 0111 1111	-	has 0 redundant sign bits
0000 0000 0000 0000 0000 0001	-	has 22 redundant sign bits
1111 1111 1111 1111 1111 1111	-	has 23 redundant sign bits
0000 0000 0000 0000 0000 0000	-	has 23 redundant sign bits (special case)

Valid results are 0 to 23 for the 24-bit registers, and -8 to 47 for rMAC.

BLKSIGNDET returns the smaller of the result of SIGNDET and the present value of the destination operand. When performed on a series of numbers, it can derive the effective exponent of the number largest in magnitude.

Flags Generated:

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Left unchanged	C	Left unchanged

6.11 Divide Instruction

Syntax:

```

Type B:  | Div = rMAC / A      | ;
            | C = DivResult |
            | C = DivRemainder |
  
```

```

Example: Div = rMAC / r1;
             r2 = DivResult;
             r3 = DivRemainder;
  
```

Description:

The `Div = rMAC/A` instruction initiates the divide block to start its multi-cycle 48-bit / 24-bit integer divide. The overflow flag is set if the `DivResult` is wider than 24-bits. In this case, the result is saturated to -2^{23} or $2^{23} - 1$ and the remainder is invalid. The result or remainder of the divide is available after 24 cycles. If the result or remainder is requested before the 24 cycles has elapsed then program flow is suspended until the result is ready. To carry out a fractional divide the value in `rMAC` needs to be first right shifted by 1 bit before carrying out the divide operation.

Integer Divide example (86420 / 7 = 12345 remainder 5):

```

rMAC = 0;           // clear rMAC
r0 = 86420;
rMAC0 = r0;        // LS word of rMAC now equals 86420;
r0 = 7;
Div = rMAC / r0;
r1 = DivResult;   // r1 = 12345
r2 = DivRemainder; // r2 = 5
  
```

Fractional Divide example (0.25/0.75 = 0.3333):

```

rMAC = 0.25;
r0 = 0.75;
rMAC = rMAC ASHIFT -1;
Div = rMAC / r0;
r1 = DivResult;   // r1 = 0.33333;
  
```

Flags Generated:

After: `Div = rMAC/A;`

Z	Left unchanged	N	Left unchanged
V	Set if a divide exception occurs and cleared otherwise	C	Left unchanged

After: `C = DivResult;` or `C = DivRemainder;`

Z	Set if the result equals zero and cleared otherwise	N	Set if the result is negative and cleared otherwise
V	Left unchanged	C	Left unchanged

6.12 Program Flow: CALL, JUMP, RTS, RTI, SLEEP, DO...LOOP and BREAK

Syntax:

Type A: `<if cond> | jump A | ;`
 `| call A |`
 `| rts |`
 `| rti |`
 `| sleep |`

Example: `if Z jump r1;`

Type B: `<if cond> | jump k16 | ;`
 `| call k16 |`
 `Do k16 |`

Example: `r10 = 100;`
`DO loop;`
`rMAC = rMAC + r0 * r1`
`r3 = M[I4,1];`
`loop:`

Type C: `<if cond> | rts | ;`
 `| rti |`

Example: `if NZ rts;`

`| break | ;`
Example: `break;`

Description:

Omitting the condition performs the program flow. If condition TRUE perform program flow below, else a NOP

jump Program execution jumps to the address in operand (either a constant, e.g. an address label or register)

do For zero overhead looping, instructions between `DO` and `loop` are executed until register `r10` is zero, `r10` is decremented by one each loop. If `r10` is zero at start then no loop instructions executed and a jump to `loop` occurs. If a `do...loop` is executed in an interrupt service routine (ISR), `r10` and the memory mapped registers `MM_DOLOOP_START` and `MM_DOLOOP_END` should be saved

sleep Program execution paused and the DSP put in lower power mode. Interrupts still handled in sleep. The ISR is responsible for issuing a software event that causes wake up from a sleep instruction.

call Loads `LINK` register with return address (`PC+1`) and jumps to address in operand (either a constant e.g. an address label or register)

rti Sets `PC` equal to value of `rIntLink` register, and restores flags to their pre-interrupt status, i.e. the MS byte of `rFlags` register is copied to the LS byte

rts Sets `PC` equal to value of `rLink` register. Stack depths greater than one must be implemented in software

break For program debug, used by `xIDE`, a break instruction either acts as a `nop`, or `jump` to self

Flags Generated:

Flags Z, N, V and C left unchanged

Note:

The instruction pipeline is not stalled by a program flow instruction, i.e. “`if cond jump <address>`” instruction takes 1 clock cycle with or without jump. Condition for program flow is evaluated the cycle before

6.13 Indexed MEM_ACCESS_1 and MEM_ACCESS_2

Syntax:

	MEM_ACCESS_1	MEM_ACCESS_2
Type A:	$\begin{aligned} \text{Reg}_{AG1} &= M[\text{I}_{AG1}, \text{M}_{AG1}] \\ M[\text{I}_{AG1}, \text{M}_{AG1}] &= \text{Reg}_{AG1} \end{aligned}$	
Example:	$M[\text{I}0, \text{M}0] = \text{r}1;$	
Type C:	$\begin{aligned} \text{Reg}_{AG1} &= M[\text{I}_{AG1}, \text{M}_{AG1}] \\ M[\text{I}_{AG1}, \text{M}_{AG1}] &= \text{Reg}_{AG1} \end{aligned}$	$\begin{aligned} \text{Reg}_{AG2} &= M[\text{I}_{AG2}, \text{M}_{AG2}] \\ M[\text{I}_{AG2}, \text{M}_{AG2}] &= \text{Reg}_{AG2} \end{aligned}$
Example:	$\text{r}0 = M[\text{I}0, \text{M}0] \quad M[\text{I}4, \text{M}1] = \text{r}1;$	

Permitted Registers

Reg _{AG1} / Reg _{AG2}	r0, r1, r2, r3, r4, r5 and rMAC	I _{AG1}	I0, I1, I2 and I3
M _{AG1} / M _{AG2}	M0, M1, M2 and M3	I _{AG2}	I4, I5, I6 and I7
MK _{AG1} / MK _{AG2}	-1, 0, 1 and 2		

Description:

Any Type C instructions can also perform up to two memory reads/writes in the same instruction cycle as the main ALU part of the instruction. Reg_{AG1/AG2} selects the source or destination register for the memory read or write. I_{AG1/AG2} selects the index register to use for the memory read, and either M_{AG1/AG2} selects the modify register or MK_{AG1/AG2} selects the modify constant (-1, 0, +1, or +2) to use for the post modify of the index register. Type A instruction can perform single memory read/write with limitation that the modify operand is not a constant, i.e. must be M_{AG1/AG2}

Memory Access Timing:

Only one access (read or write) permitted per memory bank per clock cycle. If AG1 and AG2 both access DM1 in the same instruction then the two memory accesses are queued with the AG1 access occurring first. Memory reads set up the memory bus the instruction before, whereas memory writes take place at the end of the current instruction. This means that if the previous instruction does a memory write, then the current instruction delays by one clock cycle if it tries to read the same memory bank as the previous instruction wrote to. External wait signal from peripherals may slow down the instruction cycle.

Flags Generated:

No flags affected by the memory access part of instructions.

Note:

Reg_{AG1/AG2} selects one of the first eight Bank1 register, i.e. Null, rMAC, r0–r5. If Null register selected then no memory read/write is performed

Type A instructions use Address Generator 1 (AG1) so can only use index registers I0–I3. They must use a modify register rather than a modify constant.

Type C instructions use AG1 and AG2 so one memory access must use I0–I3 and the other must use I4–I7. They must either both use a modify register or both use a modify constant.

7 Instruction Coding

Instruction coding is relatively simple and orthogonal so that the instruction decode is efficient. There are three basic coding formats: Type A, B, and C. Table 7.1 outlines The format of the instruction with corresponding definitions covered in Section 7.5 to Section 7.15.

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	Type
OP_CODE						0	0	RegC				RegA				AG1 Write	Reg _{AG1}	I _{AG1}	M _{AG1}	RegB				cond				A				
						0	1									k ₁₆																B
						1	0									AG1 Write	Reg _{AG1}	I _{AG1}	M _{AG1}	AG2 Write	REG _{AG2}	I _{AG2}	M _{AG2}	C _{REG}								
						1	1									MK _{AG1}	MK _{AG2}	C _{CONST}														
1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	K _{PREFIX} [7:0]				PFIX				

Table 7.1: Instruction Coding Format

Notes:

- OP_CODE Selects the instruction operation, see Section 7.5
- RegA Selects a register to use as the first source operand for instructions. Bank1 registers are used by default, Bank 2 is selected within OP_CODE for certain instructions
- RegB Selects a register to use as the first source operand for instructions. Bank1 registers are used by default, Bank 2 is selected within OP_CODE for certain instructions
- RegC Selects 1 of 16 registers to use as the destination register for instructions. For Type C instructions RegC also defines the first source operand, see Table 7.2. Bank1 registers are used by default, Bank2 is selected within OP_CODE for certain instructions
- cond Selects an optional condition to be met for the instruction to be executed, otherwise a no-operation will be executed
- k₁₆ / K_{PREFIX} A 16-bit / 8-bit constant used by Type B instructions
- AG1 / AG2 Selects whether the indexed memory access is a read (0) or a write (1)
- Reg_{AG1} / Reg_{AG2} Selects one of the first eight Bank1 registers: rMAC, r0-r5; for the source/destination register of multifunction memory reads and writes. If Null is selected then no read or write is performed
- I_{AG1} / I_{AG2} Selects one of the index registers, I0-I3 for AG1, and I4-I7 for AG2, for multifunction memory reads and writes
- M_{AG1} / M_{AG2} Selects one of the modify registers, M0-M3, for multifunction memory reads and writes
- MK_{AG1} / MK_{AG2} Selects a fixed constant to use for the modify: -1, 0, +1, or +2

7.1 Type A Instruction

Type A is a conditional instruction, with an additional single memory read or write operation. The instruction can accept two input operands and one output operand (which may be different). Operands can be any of the 16 Bank1 registers (Load/Store instructions permit the usage of Bank2 registers). For the memory access, index registers I0-I3 select the address, the destination or source register can be any of the first eight Bank1 registers: rMAC, r0 - r5. At the end of the instruction the index register is post modified by one of the modify registers M0-M3.

A Type A instruction has the following syntax:

```
<if cond> RegC = RegA OP(1) RegB <MEM_ACCESS_1>;
```

7.2 Type B Instruction

Type B is a non-conditional instruction similar to Type A, but with one of the operands being a 16-bit constant stored in the instruction word. To use a 24-bit constant prefix the Type B instruction with the prefix (PFIX) instruction. No additional memory access operation permitted. The PFIX instruction is automatic if needed by the assembler **kalasm2**.

A Type B instruction has the following syntax:

```
RegC = RegA OP(1) constant;
```

7.3 Type C Instruction

Type C is a non-conditional instruction similar to a Type A, except that one of the input operands is also the output operand. In addition, two memory reads or writes may occur in the same clock cycle; one memory access uses Address Generator 1 (index registers I0-I3) and the other uses Address Generator 2 (index registers I4-I7). The index registers can be either post modified by the M0-M3 registers or by a 2-bit signed modify constant (valid values: -1, 0, 1, or 2).

A Type C instruction has the following syntax:

```
RegC = RegC OP(1) RegA <MEM_ACCESS_1> <MEM_ACCESS_2>;
```

7.4 Special Cases

Program flow instructions such as `jump`, `call`, `rts`, etc; use a slight variation on the above types, as they can always be conditional. Also the multiply accumulate instructions all write their result to the rMAC register.

Note:

⁽¹⁾ OP refers to operation which can include Addition, Subtraction, Multiplication, OR, AND, Exclusive OR

7.5 OP_CODE Coding

Table 7.2 shows the instruction decoding of the OP_CODE field shown in Table 7.1

OP_CODE			Action (Type A) ⁽¹⁾	Action (Type B)	Action (Type C _{REG/CONST}) ⁽²⁾	Description	
000	AM	C	RegC = RegA + RegB	RegC = RegA + k ₁₆	RegC = RegC + RegA	Add ⁽³⁾	
001	AM	C	RegC = RegA – RegB	RegC = RegA - k ₁₆	RegC = RegC – RegA	Subtract	
010	B2RS		RegC _{B1/2} = RegA _{B1/2} + RegB _{B1/2}	RegC _{B1/2} = RegA _{B1/2} + k ₁₆	RegC _{B1/2} = RegC _{B1/2} + RegA _{B1/2}	Bank1/2 Add	
011	B2RS		RegC _{B1/2} = RegA _{B1/2} - RegB _{B1/2}	RegC _{B1/2} = RegA _{B1/2} - k ₁₆	RegC _{B1/2} = RegC _{B1/2} - RegA _{B1/2}	Bank1/2 Subtract	
100	0	0	0	RegC = RegA AND RegB	RegC = RegA AND k ₁₆	RegC = RegC AND RegA	Logical AND
100	0	0	1	RegC = RegA OR RegB	RegC = RegA OR k ₁₆	RegC = RegC OR RegA	Logical OR
100	0	1	0	RegC = RegA XOR RegB	RegC = RegA XOR k ₁₆	RegC = RegC XOR RegA	Logical XOR
100	0	1	1	RegC = RegA LSHIFT RegB	RegC = RegA LSHIFT k ₁₆ ⁽⁴⁾	RegC = RegC LSHIFT RegA	Logical Shift
100	1	0	0	RegC = RegA ASHIFT RegB	RegC = RegA ASHIFT k ₁₆ ⁽⁴⁾	RegC = RegC ASHIFT RegA	Arithmetic Shift
100	1	1	V	RegC = RegA * RegB (int)	RegC = RegA * k ₁₆ (int)	RegC = RegC * RegA (int)	Integer signed multiply
100	1	0	1	RegC = RegA * RegB (frac)	RegC = RegA * k ₁₆ (frac)	RegC = RegC * RegA (frac)	Fractional signed multiply
101	0	S	S	rMAC = rMAC + RegA * RegB	rMAC = rMAC + RegA * k ₁₆	rMAC = rMAC + RegC * RegA	Multiply accumulate (56-bit)
101	1	S	S	rMAC = rMAC – RegA * RegB	rMAC = rMAC – RegA * k ₁₆	rMAC = rMAC – RegC * RegA	Multiply subtract (56-bit)
110	0	S	S	rMAC = RegA * RegB	rMAC = RegA * k ₁₆	rMAC = RegC * RegA	Multiply (48-bit)
110	1	0	0	RegC = M[RegA + RegB]	RegC = M[RegA + k ₁₆]	RegC = M[RegC + RegA]	Load with offset
110	1	0	1	M[RegA + RegB] = RegC	M[RegA + k ₁₆] = RegC	-	Store with offset
110	1	1	0	RegC = SignDet RegA	Div = rMAC / RegA RegC = DivResult ⁽²⁾ RegC = DivRemainder ⁽²⁾	RegC = BlkSignDet RegA	Sign detect / Divide / Block sign detect
110	1	1	1	JUMP RegA	if [RegC=cond] JUMP k ₁₆	if [RegC = cond] RTS	Jump to program address / return from subroutine
111	0	0	0	CALL RegA	if [RegC=cond] CALL k ₁₆	if [RegC = cond] RTI	Call subroutine / return from interrupt
111	0	0	1	SLEEP	DO LOOP	BREAK	Go into sleep mode / Do Loop / Debug Break point
111	0	1	0	FUTURE USE ⁽⁵⁾			
111	0	1	1				
111	1	x	x				
111	1	1	1	PREFIX instruction			Allows 24-bit constants for Type B operations – by prefixing the following instruction's k ₁₆ by the 8-bit k _{PREFIX} value.

Table 7.2: OPCODE Coding Format

Note:

- (1) Type A is conditional with `if [cond]` and can contain single memory access `MEM_ACCESS_1`
- (2) Type C permits two simultaneous memory access `MEM_ACCESS_1` and `MEM_ACCESS_2`
- (3) The add instruction is also used to implement load/stores to registers/memory by setting one of the source registers as Null. Setting all 3 operands as Null implements a no-operation (NOP) instruction
- (4) See Section 7.11 and Section 7.15 for encoding of k₁₆ for rMAC shift instructions and Divide instructions
- (5) There is six spare OP_CODES plus various other spare coding space for future instructions

7.6 AM Field

Selects different memory addressing modes

AM	Type A	Type B	Type C _{REG/CONST}
00	RegC = RegA [OP] RegB	RegC = RegA [OP] K16	RegC = RegC [OP] RegA
01	RegC = RegA [OP] M[RegB]	RegC = RegA [OP] M[K16]	RegC = RegC [OP] M[RegA]
10	RegC = M[RegA] [OP] RegB	RegC = M[RegA] [OP] K16	-
11	M[RegC] = RegA [OP] RegB	M[K16] = RegC [OP] RegA	-

Table 7.3: AM Field

7.7 Carry Field (C Field)

Selects whether addition/subtraction performed with carry and the appropriate state shown in Table 7.4.

C	Description
0	Do not use carry / borrow
1	Use carry / borrow

Table 7.4: C Field Options

7.8 Bank 1/2 Register Select Field (B2RS Field)

Bank 1/2 register select for add/subtract instructions are shown in Table 7.5

B2RS	Type A	Type B	Type C _{REG/CONST}
000	$\text{RegC}_{\text{Bank1}} = \text{RegA}_{\text{Bank1}} \pm \text{RegB}_{\text{Bank1}}$	$\text{RegC}_{\text{Bank1}} = \text{RegA}_{\text{Bank1}} \pm K_{16}$	$\text{RegC}_{\text{Bank1}} = \text{RegC}_{\text{Bank1}} \pm \text{RegA}_{\text{Bank1}}$
001	$\text{RegC}_{\text{Bank1}} = \text{RegA}_{\text{Bank1}} \pm \text{RegB}_{\text{Bank2}}$		$\text{RegC}_{\text{Bank1}} = \text{RegC}_{\text{Bank1}} \pm \text{RegA}_{\text{Bank2}}$
010	$\text{RegC}_{\text{Bank1}} = \text{RegA}_{\text{Bank2}} \pm \text{RegB}_{\text{Bank1}}$	$\text{RegC}_{\text{Bank1}} = \text{RegA}_{\text{Bank2}} \pm K_{16}$	
011	$\text{RegC}_{\text{Bank1}} = \text{RegA}_{\text{Bank2}} \pm \text{RegB}_{\text{Bank2}}$		
100	$\text{RegC}_{\text{Bank2}} = \text{RegA}_{\text{Bank1}} \pm \text{RegB}_{\text{Bank1}}$	$\text{RegC}_{\text{Bank2}} = \text{RegA}_{\text{Bank1}} \pm K_{16}$	
101	$\text{RegC}_{\text{Bank2}} = \text{RegA}_{\text{Bank1}} \pm \text{RegB}_{\text{Bank2}}$		
110	$\text{RegC}_{\text{Bank2}} = \text{RegA}_{\text{Bank2}} \pm \text{RegB}_{\text{Bank1}}$	$\text{RegC}_{\text{Bank2}} = \text{RegA}_{\text{Bank2}} \pm K_{16}$	$\text{RegC}_{\text{Bank2}} = \text{RegC}_{\text{Bank2}} \pm \text{RegA}_{\text{Bank1}}$
111	$\text{RegC}_{\text{Bank2}} = \text{RegA}_{\text{Bank2}} \pm \text{RegB}_{\text{Bank2}}$		$\text{RegC}_{\text{Bank2}} = \text{RegC}_{\text{Bank2}} \pm \text{RegA}_{\text{Bank2}}$

Table 7.5: B2RS Field

7.9 Saturation Select Field (V Field)

Selects whether to enable saturation on the result and the options shown in Table 7.6.

V	Description
0	No saturation
1	Saturation

Table 7.6: V Field

7.10 Sign Select Field (S Field)

Selects signed/unsigned multiplies and the various options shown in Table 7.7.

S	Description
00	unsigned x unsigned
01	unsigned x signed
10	signed x unsigned
11	signed x signed

Table 7.7: S Field

7.11 k_{16} Coding for LSHIFT and ASHIFT

k_{16} coding section from the instruction coding format shown in Table 7.1 splits into its individual bits and their functionality listed in Table 7.8

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

Table 7.8: k_{16} Coding Shift Format

Note:

ShiftAmount: Is a signed 7-bit number that is the amount to shift the input.
Positive is a left shift, negative is a right shift.

Dest_Sel: When the destination register is rMAC, selects how the 24-bit output from the shifter is used.

7.12 rMAC Sub Registers

The full 56-bits of the rMAC register are accessible as their individual sub-registers outlined in Table 7.9. See Section 7.13 and Section 7.14 for further details on how to load the individual sub registers.

Bit	55 – 48	47 – 24	23 – 0
rMAC Register	rMAC2	rMAC1	rMAC0

Table 7.9: rMAC Sub-Registers

7.13 ASHIFT

Table 7.10 represents how the arithmetic shift instruction is encoded within the instruction coding format shown in Table 7.1 for further information on the LSHIFT instruction see Section 6.5.

Dest_Sel	New rMAC2	New rMAC1	New rMAC0	Example
001	Sign extend	Sign extend	SHIFTER_OUTPUT	rMAC = r? ASHIFT k_{16} (0)
000	Sign extend	SHIFTER_OUTPUT	Trailing zeros	rMAC = r? ASHIFT k_{16} (1)
010	SHIFTER_OUTPUT	Trailing zeros	Trailing zeros	rMAC = r? ASHIFT k_{16} (2)
101	Old rMAC2	Old rMAC1	SHIFTER_OUTPUT	rMAC0 = r? ASHIFT k_{16}
100	Sign extend	SHIFTER_OUTPUT	Old rMAC0	rMAC1 = r? ASHIFT k_{16}
110	SHIFTER_OUTPUT	Old rMAC1	Old rMAC0	rMAC2 = r? ASHIFT k_{16}

Table 7.10: ASHIFT

7.14 LSHIFT

Table 7.11 represents how the logical shift instruction is encoded within the instruction coding format shown in Table 7.1, for further information on the LSHIFT instruction see Section 6.5.

Dest_Sel	New rMAC2	New rMAC1	New rMAC0	Example
001	Sign extend	Sign extend	SHIFTER_OUTPUT	rMAC = r? LSHIFT k_{16} (0)
000	Sign extend	SHIFTER_OUTPUT	Trailing zeros	rMAC = r? LSHIFT k_{16} (1)
010	SHIFTER_OUTPUT	Trailing zeros	Trailing zeros	rMAC = r? LSHIFT k_{16} (2)
101	Old rMAC2	Old rMAC1	SHIFTER_OUTPUT	rMAC0 = r? LSHIFT k_{16}
100	Sign extend	SHIFTER_OUTPUT	Old rMAC0	rMAC1 = r? LSHIFT k_{16}
110	SHIFTER_OUTPUT	Old rMAC1	Old rMAC0	rMAC2 = r? LSHIFT k_{16}

Table 7.11: LSHIFT

Both the LHIFT and ASHIFT instruction enable the user to write to the three individual sub-registers rMAC2/1/0, hence providing a way of loading rMAC with a double precision number. It also allows shift operations of rMAC with the destination being the rMAC register, which speeds up double precision calculations. See rMAC move operations in Section 6 for more details.

7.15 k_{16} Coding Divide Instructions

Table 7.12 k_{16} coding divide instruction represents how the background divide instruction is encoded within the instruction coding format shown in Table 7.1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Not used														Div	

Table 7.12: Divide Field

The functionality of the individual divide bits is explained more fully in Table 7.13.

Div	Assembly Syntax	Operation
00	Div = rMAC / RegA	Initiates a 32-bit/16-bit divide. The overflow flag is set if a divide exception occurs. Calculation of the divide takes 16 clock cycles but program execution continues while the divide is calculated.
01	RegC = DivResult	The result and/or remainder of a divide are available 16 cycles after the divide is initiated. In this period, normal program execution continues. If the result is requested early then program execution is automatically delayed until the result is available.
10	RegC = DivRemainder	
11	NA	Not used

Table 7.13: Divide Field States

8 Kalimba DSP Peripherals

The Kalimba DSP for the BlueCore3-Multimedia consists of the DSP core, the DSP peripherals and their associated interfaces. This section covers the peripherals and interfaces.

See Appendix B for information relating to registers

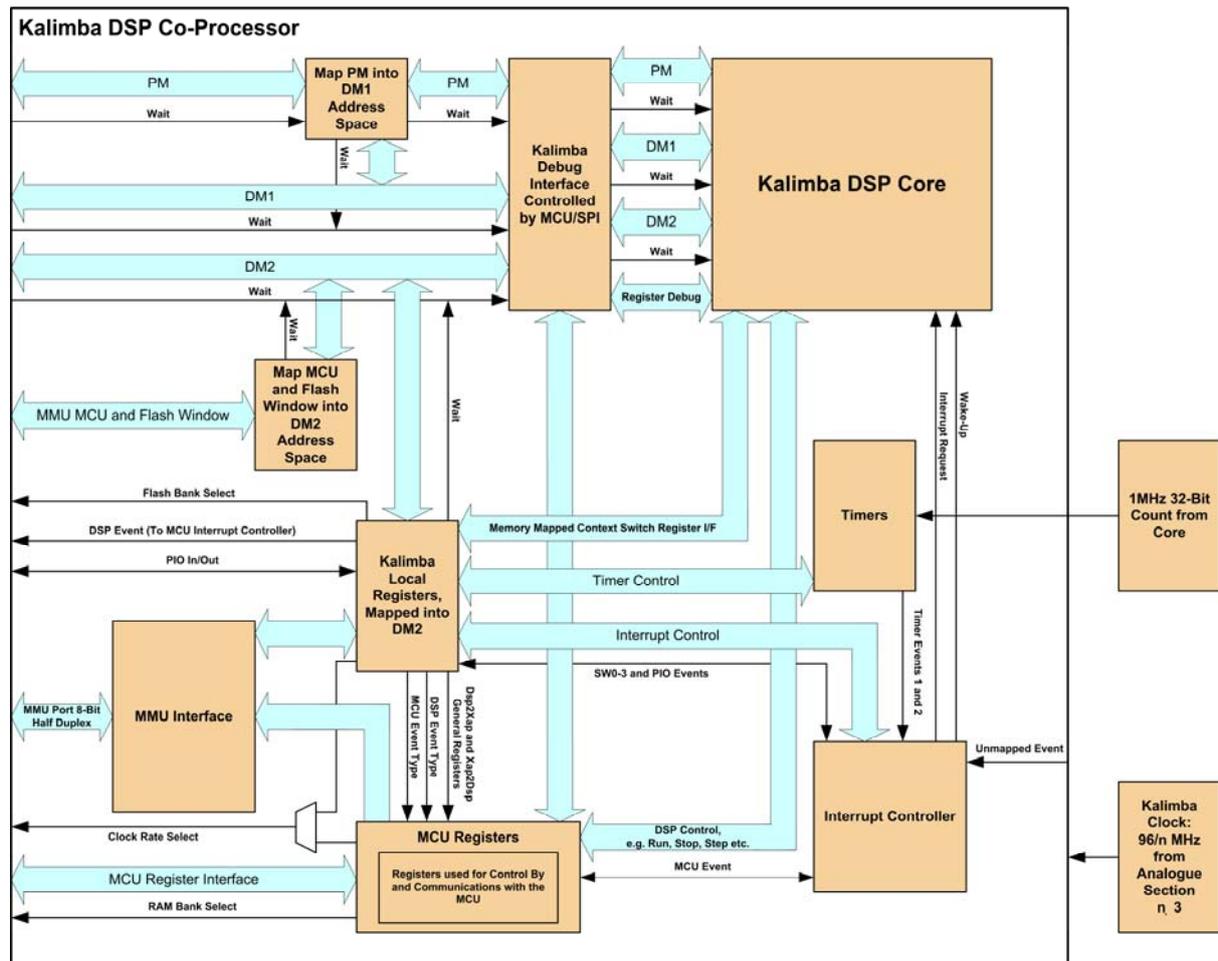


Figure 8.1: Kalimba DSP Peripheral Interfaces

The Kalimba DSP peripherals include:

- Memory management unit (MMU) interface, for stream transfers to/from the BlueCore3-Multimedia subsystem
- Memory mapped window into the flash
- Two memory mapped windows into the MCU RAM
- Two 1 μ s timers
- Interrupt controller with three priority levels and wake up from sleep
- Memory mapped access to the DSP program memory through DM1
- Clock rate divider controllable by both the DSP and the MCU
- Debug interface

8.1 MMU Interface

The MMU Interface on BlueCore3-Multimedia for the Kalimba DSP contains four virtual read ports and four virtual write ports; an example of the usage of these ports is in Figure 8.2.

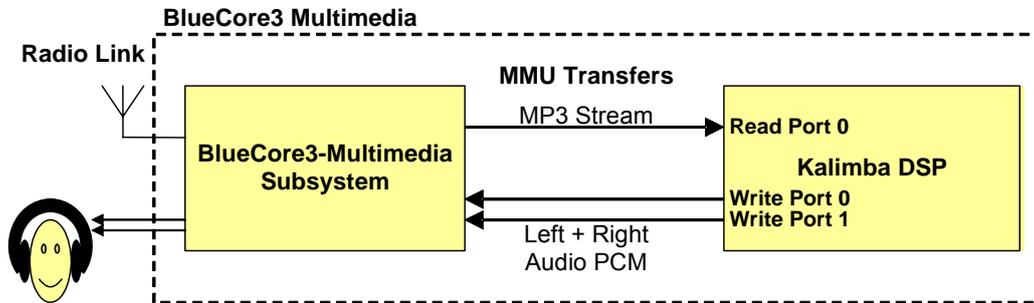


Figure 8.2: Example of MMU Interface Usage for a Wireless MP3 Player

8.1.1 Read Ports

Four virtual read ports appear as memory mapped registers in DM2. The ports have the ability to use:

- Prefetch, if required, to improve speed
- 8-bit or 16-bit word size
- Byte swap capability (Little Endian/Big Endian)

8.1.2 Write Ports

Four virtual write ports appear as memory mapped registers in DM2. The ports have the ability to use:

- 8-bit or 16-bit word size
- Byte swap capability (Little Endian/Big Endian)

8.2 DSP Timers

The features of the Kalimba DSP timer are as follows:

- A 24-bit TIMER_TIME register (read only) clocked @ 1MHz
- Two trigger value registers, each, if enabled, cause an appropriate interrupt

See Appendix B for control register details.

8.3 Kalimba Interrupt Controller

This section covers the functionality of Kalimba DSP during interrupts looking at the events of the Kalimba DSP core and the interrupt controller, as well as investigating the interrupt controller registers.

8.3.1 DSP Core Functionality During Interrupt

Upon reception of an interrupt, this is when the IRQ is high; the DSP core performs the following:

- rIntLink is loaded with the contents of current Program counter
- Program counter is loaded with the address of the interrupt service routine e.g. 0x0002
- The flags register is saved
- Perform interrupt service routine

When the interrupt service routine completes the DSP needs to return to the routine it had been running prior to the interrupt; this executed with a `rti` instruction. The `rti` instruction carries out the following:

- Restores the rflags register to the non-interrupt state
- Loads the program counter(PC) with the contents of the rIntLink register

Note:

Saving/restoring of further registers is up to the programmer

8.3.2 Interrupt Controller Functionality

The functionality of the interrupt controller is

- Selectable interrupt sources:
 - Timer1 and Timer2
 - MCUEvent
 - PIOEvent
 - SoftwareEvent0
 - SwEvent1, SwEvent2 and SwEvent3
- Three interrupt priority levels as well as wake-up from sleep
- Registers to save and restore the interrupt controller which allows for nested interrupts
- Optional event signal to cause clock rate change

See Appendix B for control register details.

8.4 Generation of MCU Interrupt

Writing to the DSP2MCU_EVENT_DATA register will cause an interrupt to be sent to the MCU, the value of this register can be seen by the MCU. In a similar way the MCU can generate an interrupt to the DSP, with the event type being stored in the MCU2DSP_EVENT_DATA register. These registers can be used to pass messages between MCU and DSP and vice versa.

See Appendix B for control register details.

8.5 PIO Control

This section describes the interface between the Kalimba DSP and the programmable I/O (PIO) of the BlueCore3-Multimedia. Control of the PIO from the Kalimba DSP is as follows:

- Kalimba DSP can read BlueCore3-Multimedia PIO lines
- Under the control of the MCU the Kalimba DSP can write to PIO lines
- PIO line change can generate a Kalimba DSP interrupt

The MCU controls which PIO bits are inputs and which are outputs. It also controls which write access permission for the Kalimba DSP. This information is set through VM functions (PioSetKalimba and PioGetKalimba – see the BlueLab on-line documentation C Reference guide \ file list \ pio.h for details of these functions)

See Appendix B for control register details.

8.6 MCU Memory Window in DM2

Two windows in DM2 shown in Figure 5.1 allow the Kalimba DSP to access MCU memory. The primary use of this memory window is for message passing and control information.

The MCU controls access to this window. A start address and a size for each window along with whether the DSP has read access or read/write access can be set. This information is set by the MCU firmware used.

See Appendix B for control register details.

8.7 Flash Memory Window in DM2

The purpose of the Flash Memory Window in DM2 is to permit the DSP access to the 8Mbit flash, this could be for example to access further coefficients, download a new program. The window size is 4Kwords and the FLASH_BANK_SELECT register selects which 4K block is visible to the DSP.

Note:

Firmware does not currently support the use of the DSP's flash memory window.

See Appendix B for control register details.

8.8 PM Window in DM1

The PM window within the DM1 memory bank permits the DSP to change its own program or to be extra data memory (16-bit). For safety, it is possible to disable the window.

The MCU must enable the DSP to have access to the PM mapped into DM1. The DSP must then enable access as well.

Note:

MCU firmware does not currently support the use of the DSP's PM window in DM1.

See Appendix B for control register details.

8.9 General Registers

The General Registers are for communication between the MCU and the Kalimba DSP.

See Appendix B for control register details.

8.10 Clock Rate Divider Control

The Kalimba DSP may control its own clock frequency.

See Appendix B for control register details.

8.11 Debugging

The MCU registers and therefore the serial peripheral interface (SPI) can:

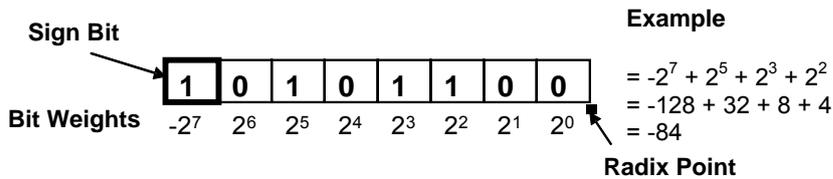
- Read and write any of the DSP core's internal registers
- Control: Single Step, Run, Stop, Breakpoints, etc
- Read and write any **individual** location in PM, DM1 or DM2, as seen by the DSP

Appendix A: Number Representation

The number representation used by the Kalimba DSP is outlined in this Appendix.

Binary Integer Representation

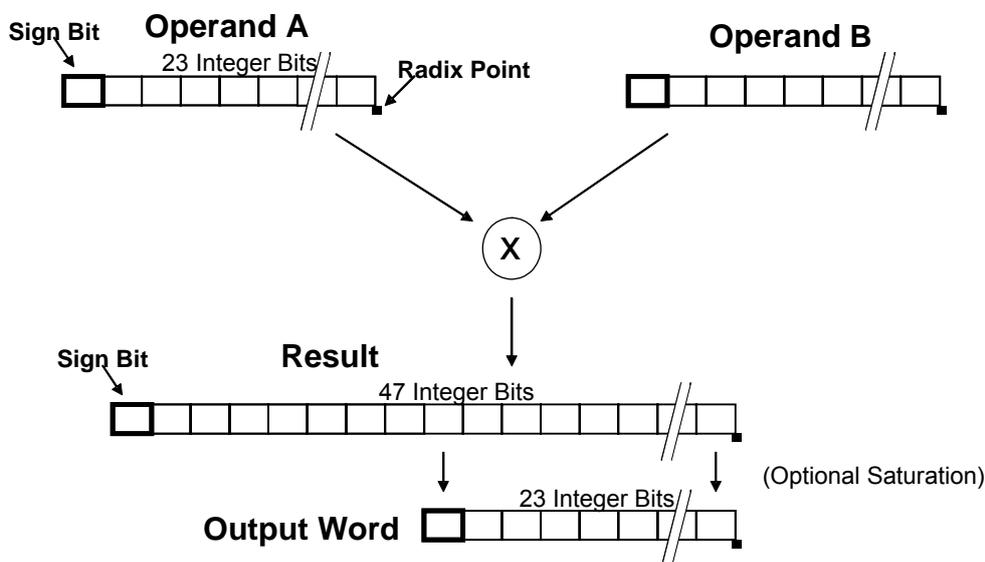
Two's complement, only 8-bits shown for clarity.



Binary Fractional Representation

Only 8-bits shown for clarity.

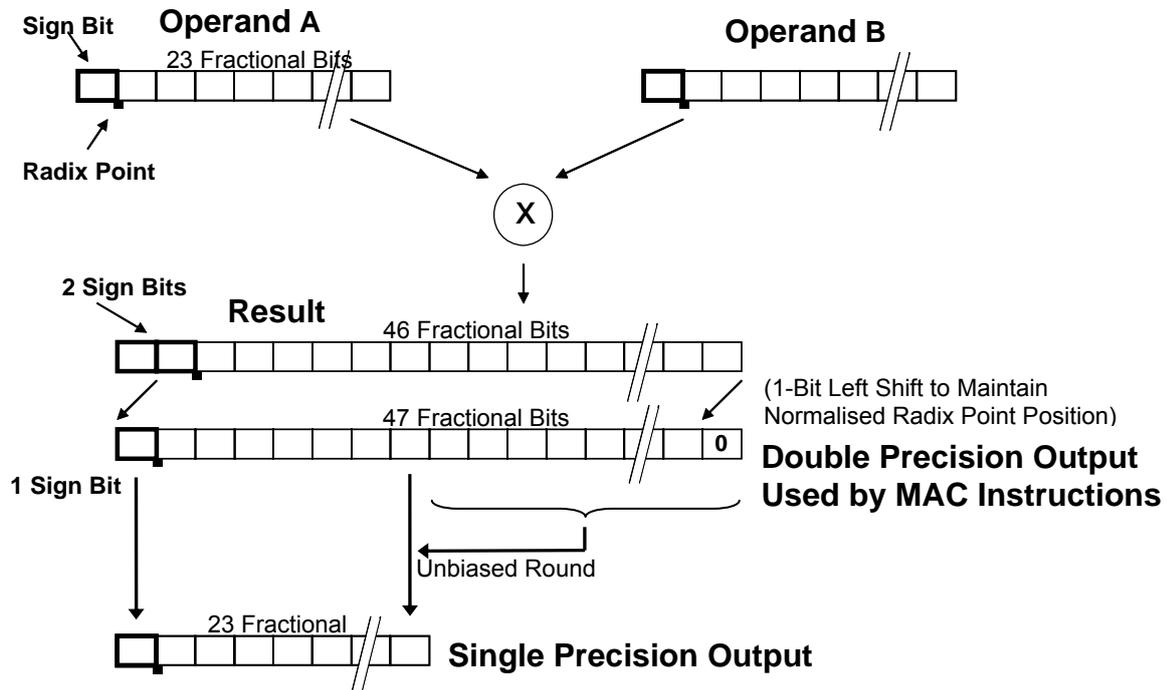
Integer Multiplication



Example 1:
 Operand A = 123
 Operand B = 456
 Output Word = 56088

Example 2:
 Operand A = 12345
 Operand B = 67890
 Output Word = 8388607 (With Saturation)
 Output Word = -758750 (Without Saturation)

Fractional Multiplication



Example:

Operand A = 0x654321 = 0.79111111
 Operand B = 0x123456 = 0.14222217
 Output word = 0x0E66D7F2822C
 = 0.11251353589 (Double Precision)
 = 0x0E66D8 (Unbiased Round)
 = 0.11251354 (Single Precision)

Appendix B: DSP Registers

DSP Memory Mapped I/O

As described in Section 5.1.3 the DSP memory mapped I/O forms a reserved part of the DM2 memory map. Listed in Appendix B Table 1 are the memory mapped registers.

Address	Name	Size (Bit)	RW	Description
0xFF00	INT_GBL_ENABLE	1	RW	Resets interrupt controller state
0xFF01	INT_ENABLE	1	RW	Enable searching for an interrupt source
0xFF02	INT_CLK_SWITCH_EN	1	RW	Enable switching of DSP clock to a special interrupt clock rate
0xFF03	INT_SOURCES_EN	9	RW	Enables interrupt sources for Kalimba
0xFF04	INT_PRIORITIES	18	RW	Set priority levels of individual interrupts
0xFF05	INT_LOAD_INFO	12	RW	Restore information of lowest priority interrupt during a nested interrupt
0xFF06	INT_ACK	1	RW	Clears current interrupt request
0xFF07	INT_SOURCE	4	R	Contains current interrupt source
0xFF08	INT_SAVE_INFO	11	R	Save information of lowest priority interrupt during a nested interrupt
0xFF09	DSP2MCU_EVENT_DATA	16	RW	Interrupt event data to MCU
0xFF0A	MCU2DSP_EVENT_DATA	16	R	Interrupt event data from MCU
0xFF0B	TIMER_TIME	24	R	24-bit timer value increment every 1 μ s
0xFF0C	TIMER1_EN	1	RW	Enable Timer1 interrupt
0xFF0D	TIMER2_EN	1	RW	Enable Timer2 interrupt
0xFF0E	TIMER1_TRIGGER	24	RW	Timer1 trigger value
0xFF0F	TIMER2_TRIGGER	24	RW	Timer2 trigger value
0xFF10	WRITE_PORT0_DATA	8/16	RW	Write port 0
0xFF11	WRITE_PORT1_DATA	8/16	RW	Write port 1
0xFF12	WRITE_PORT2_DATA	8/16	RW	Write port 2
0xFF13	WRITE_PORT3_DATA	8/16	RW	Write port 3
0xFF14	WRITE_PORT0_CONFIG	2	RW	Data size and Endian mode for write port 0
0xFF15	WRITE_PORT1_CONFIG	2	RW	Data size and Endian mode for write port 1
0xFF16	WRITE_PORT2_CONFIG	2	RW	Data size and Endian mode for write port 2
0xFF17	WRITE_PORT3_CONFIG	2	RW	Data size and Endian mode for write port 3
0xFF18	READ_PORT0_DATA	8/16	R	Read port 0
0xFF19	READ_PORT1_DATA	8/16	R	Read port 1
0xFF1A	READ_PORT2_DATA	8/16	R	Read port 2
0xFF1B	READ_PORT3_DATA	8/16	R	Read port 3
0xFF1C	READ_PORT0_CONFIG	3	RW	Data size, Endian mode and prefetch for read port 0
0xFF1D	READ_PORT1_CONFIG	3	RW	Data size, Endian mode and prefetch for read port 1
0xFF1E	READ_PORT2_CONFIG	3	RW	Data size, Endian mode and prefetch for read port 2

Address	Name	Size (Bit)	RW	Description
0xFF1F	READ_PORT3_CONFIG	3	RW	Data size, Endian mode and prefetch for read port 3
0xFF20	MM_DOLOOP_START	16	RW	Start address of zero overhead loop
0xFF21	MM_DOLOOP_END	16	RW	End address of zero overhead loop
0xFF22	MM_QUOTIENT	24	RW	Quotient result of division instruction
0xFF23	MM_REM	24	RW	Remainder result of division instruction
0xFF24	GENERAL_FROM_MCU0	16	R	For message passing from on board MCU to the Kalimba
0xFF25	GENERAL_FROM_MCU1	16	R	Register for message passing from on board MCU to the Kalimba
0xFF26	GENERAL_FROM_MCU2	16	R	Register for message passing from on board MCU to the Kalimba
0xFF27	GENERAL_FROM_MCU3	16	R	Register for message passing from on board MCU to the Kalimba
0xFF28	GENERAL_TO_MCU0	16	W	Register for message passing from Kalimba to the on board MCU
0xFF29	GENERAL_TO_MCU1	16	W	Register for message passing from Kalimba to the on board MCU
0xFF2A	GENERAL_TO_MCU2	16	W	Register for message passing from Kalimba to the on board MCU
0xFF2B	GENERAL_TO_MCU3	16	W	Register for message passing from Kalimba to the on board MCU
0xFF2C	CLOCK_DIVIDE_RATE	4	RW	Configuration for clock frequency used by Kalimba during normal operation
0xFF2D	INT_CLOCK_DIVIDE_RATE	4	RW	Configuration for clock frequency used by Kalimba during interrupt service, when selected
0xFF2E	PIO_IN	16	R	Programmable input register used by DSP to read PIO0 –11 and AIO0-3
0xFF2F	PIO_OUT	16	RW	Programmable output register used by DSP to write PIO0 –11 and AIO0-3
0xFF30	PIO_EVENT_EN_MASK	16	RW	Used to select which PIOs and AIOs are used for interrupt sources
0xFF31	INT_SW0_EVENT	1	RW	Select software event 0 to cause interrupt request on Kalimba
0xFF32	INT_SW1_EVENT	1	RW	Select software event 1 to cause interrupt request on Kalimba
0xFF33	INT_SW2_EVENT	1	RW	Select software event 2 to cause interrupt request on Kalimba
0xFF34	INT_SW3_EVENT	1	RW	Select software event 3 to cause interrupt request on Kalimba
0xFF35	FLASH_BANK_SELECT	12	RW	Selects 4K block of flash overlaid into the Flash Access Window of DM2 RAM, see Figure 5.1
0xFF36	NOSIGNX_GENREGS	1	RW	General sign extension enable
0xFF37	NOSIGNX_MCUWIN1	1	RW	Enable sign extension in MCU window 1
0xFF38	NOSIGNX_MCUWIN2	1	RW	Enable sign extension in MCU window 2

Address	Name	Size (Bit)	RW	Description
0xFF39	NOSIGNX_FLASHWIN	1	RW	Enable sign extension in flash window
0xFF3A	NOSIGNX_PMWIN	1	RW	Enable sign extension in PM window
0xFF3B	PM_WIN_ENABLE	1	RW	Allows program memory to be mapped into DM1 memory map see Figure 5.1

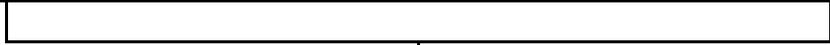
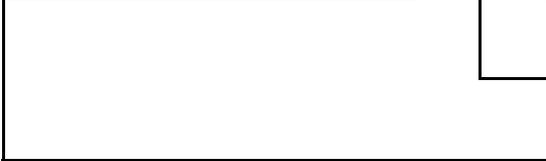
Appendix B Table 1: DSP Memory Mapped I/O

MMU Interface DSP Registers

Tables in this section list a group of registers used for communication and control between the Kalimba DSP and the MCU on the BlueCore3-Multimedia.

WRITE_PORT[0/1/2/3]_DATA		Description
Bit		
23 ~ 0		Data bits 0-23
		
Note	Memory mapped location for the write port 0 to 3 data	

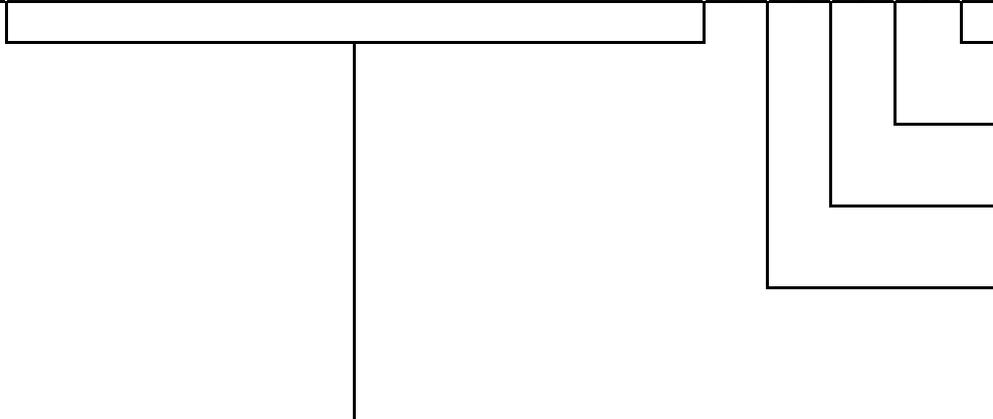
Appendix B Table 2: Data Bits for Write Port 0 to 3

WRITE_PORT[0/1/2/3]_CONFIG				Description	
Bit					
23 ~ 2			1	0	1 = 16-bit write 0 = 8-bit write
					
					Unused
Note	This register is cleared on system reset				
	The stereo audio interface of BlueCore3-Multimedia uses Little Endian format				

Appendix B Table 3: Configuration Bits for Write Port 0 to 3

READ_PORT[0/1/2/3]_DATA		Description
Bit		
23 ~ 0		Data bits 0-23
		
Note	Memory mapped location for the read port 0 to 3 data	

Appendix B Table 4: Data Bits for Read Port 0 to 3

READ_PORT[0/1/2/3]_CONFIG					Description	
Bit						
23 ~ 4			3	2	1	0
			1 = 16-bit write 0 = 8-bit write			
			1 = Big Endian 0 = Little Endian			
			1 = Pre-fetch 0 = No pre-fetch			
			1 = No sign extension 0 = Sign extension			
			Unused			
Note	Configuration bits for the corresponding read port This register is cleared on system reset The stereo audio interface of BlueCore3-Multimedia uses Little Endian format Sign extension is to the 23 rd bit					

Appendix B Table 5: Configuration Bits for Read Port 0 to 3

Timers DSP Registers

Three groups of registers make up the Kalimba DSP timer, these are:

TIMER_TIME		Description
Bit		
23 ~ 0		
		Data bits 0-23 for 1µs count value
Note	A 24-bit read only register, the value is incremented every 1µs	

Appendix B Table 6: The Timer Time Register

TIMER[1/2]_TRIGGER		Description
Bit		
23 ~ 0		
		Data bits 0-23 for 1µs trigger count value
Note	A 24-bit trigger value. If the appropriate enable register, TIMER[1/2]_EN, is set then an interrupt is generated when $TIMER_TIME \geq TIMER[1/2]_TRIGGER$	

Appendix B Table 7: The Threshold Value Registers for Timer Trigger 1 and 2

TIMER[1/2]_EN		Description
Bit		
23 ~ 1	0	1 = Enable timer interrupt 0 = Disable timer interrupt
Unused		
Note	This register is cleared on system reset	

Appendix B Table 8: Enable Bits for Timer 1 and Timer 2 Interrupts

Interrupt Controller DSP Registers

This section covers the registers concerned with interrupt control on the DSP.

INT_GBL_ENABLE		Description
Bit		
23 ~ 1	0	0 = Reset interrupt controller state Unused
Unused		
Note	This register is cleared on system reset	

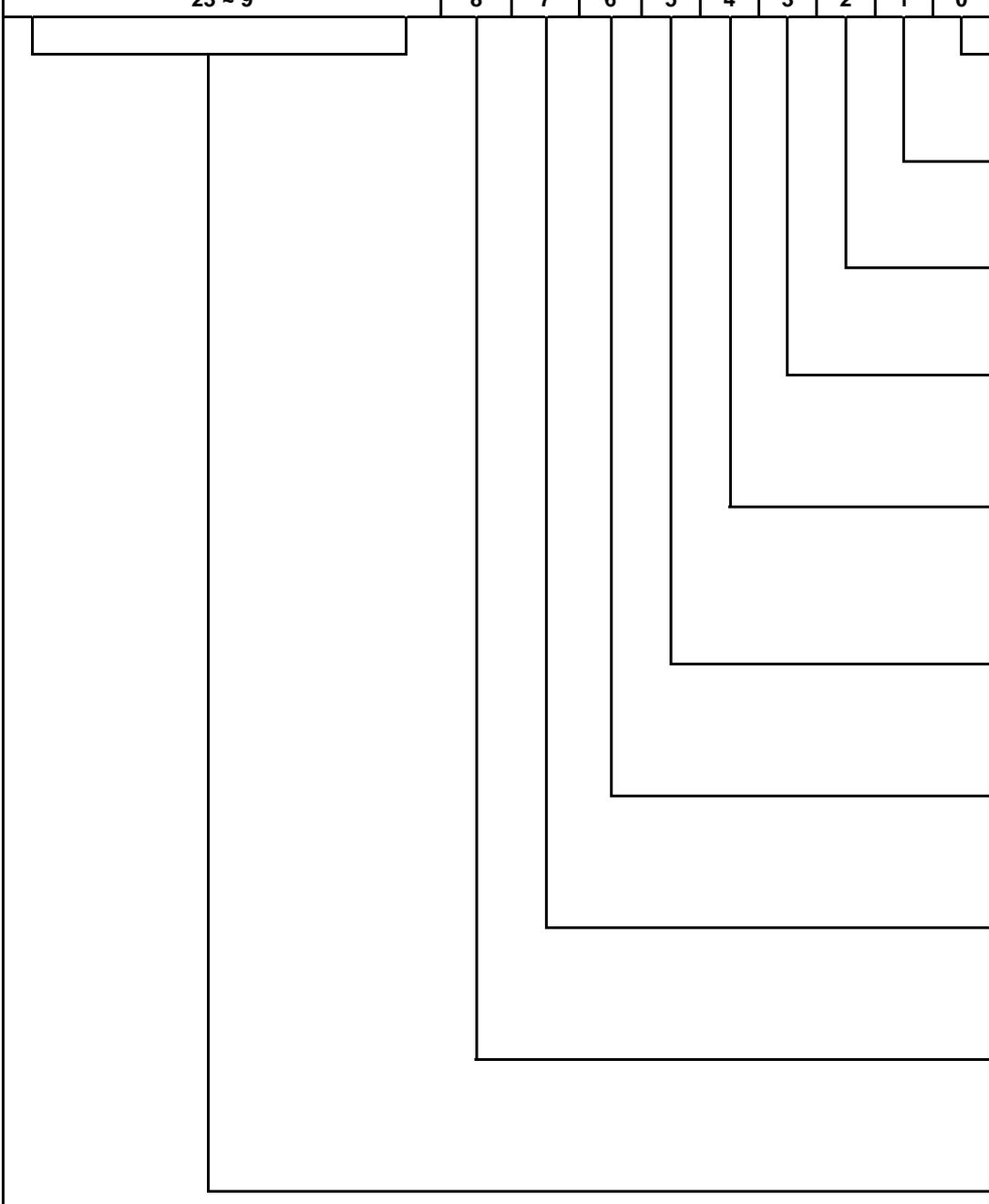
Appendix B Table 9: Interrupt Controller State Reset Bit

INT_ENABLE		Description
Bit		
23 ~ 1	0	1 = Enable searching for an interrupt source 0 = Disable searching for an interrupt source
Unused		
Note	This register is cleared on system reset	

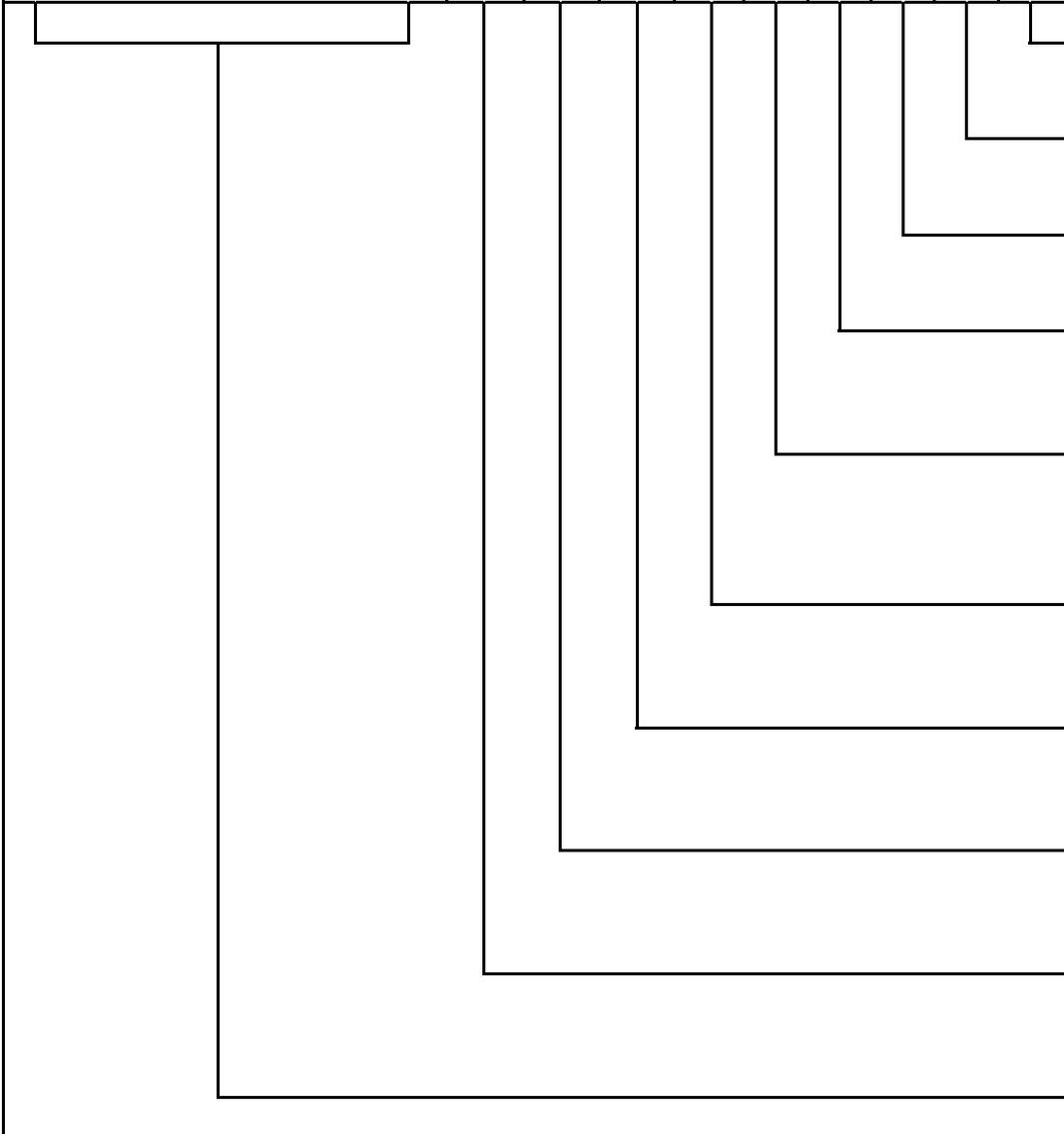
Appendix B Table 10: Enable Interrupt Searching Bit

INT_CLK_SWITCH_EN		Description
Bit		
23 ~ 1	0	1 = Enable DSP clock rate switch during interrupt 0 = Disable DSP clock rate switch during interrupt
Unused		
Note	The DSP special clock rate during interrupt is DSP_INT_CLOCK_RATE described in Section 8.10 This register is cleared on system reset	

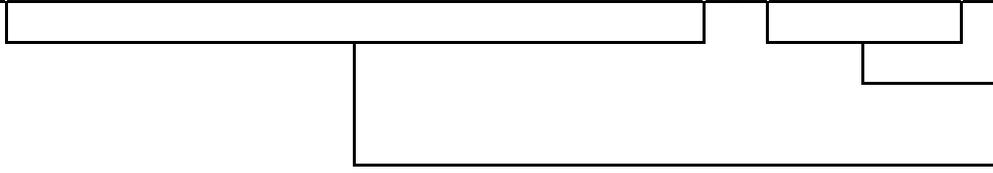
Appendix B Table 11: Enable Interrupt Clock Rate Bit

INT_SOURCES_EN										Description
Bit										
23 ~ 9	8	7	6	5	4	3	2	1	0	
										Timer1 interrupt 1 = Enable 0 = Disable
										Timer2 interrupt 1 = Enable 0 = Disable
										MCU interrupt 1 = Enable 0 = Disable
										PIO line change interrupt 1 = Enable 0 = Disable
										MMU unmapped event interrupt 1 = Enable 0 = Disable
										Software0 event interrupt 1 = Enable 0 = Disable
										Software1 event interrupt 1 = Enable 0 = Disable
										Software2 event interrupt 1 = Enable 0 = Disable
										Software3 event interrupt 1 = Enable 0 = Disable
										Unused
Note										
Bit 0: INT_SOURCE_TIMER1_POSN		5: INT_SOURCE_SW0_POSN								
1: INT_SOURCE_TIMER2_POSN		6: INT_SOURCE_SW1_POSN								
2: INT_SOURCE_MCU_POSN		7: INT_SOURCE_SW2_POSN								
3: INT_SOURCE_PIO_POSN		8: INT_SOURCE_SW3_POSN								
4: INT_SOURCE_MMU_UNMAPPED_POSN										

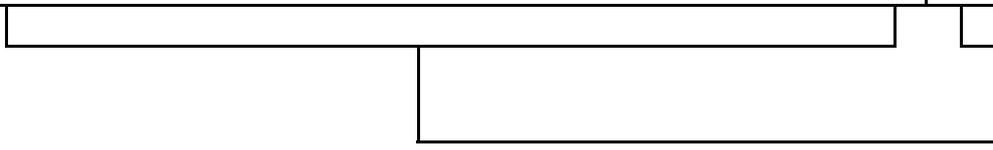
Appendix B Table 12: Interrupt Source Enable Bits

INT_PRIORITIES										Description
Bit										
23 ~ 18	17 ~ 16	15 ~ 14	13 ~ 12	11 ~ 10	9 ~ 8	7 ~ 6	5 ~ 4	3 ~ 2	1 ~ 0	
										Timer1 interrupt priority
										Timer2 interrupt priority
										MCU interrupt priority
										PIO line change interrupt priority
										MMU unmapped event interrupt priority
										Software0 event interrupt priority
										Software1 event interrupt priority
										Software2 event interrupt priority
										Software3 event interrupt priority
Unused										
Note	Priority level for each interrupt is a 2-bit value where the range is 3 (highest priority level) to 0 (lowest)									

Appendix B Table 13: Interrupt Priority Level Bits

INT_SOURCE		Description
Bit		
23 ~ 4	3 ~ 0	
		Current Interrupt Source
		Unused
Note	Current Interrupt Source is a 4-bit value that represents the current interrupt as follows: <ul style="list-style-type: none"> 0000 Timer1 interrupt 0001 Timer2 interrupt 0010 MCU interrupt 0011 PIO line change interrupt 0100 MMU unmapped event interrupt 0101 Software0 event interrupt 0110 Software1 event interrupt priority 0111 Software2 event interrupt 1000 Software3 event interrupt 	

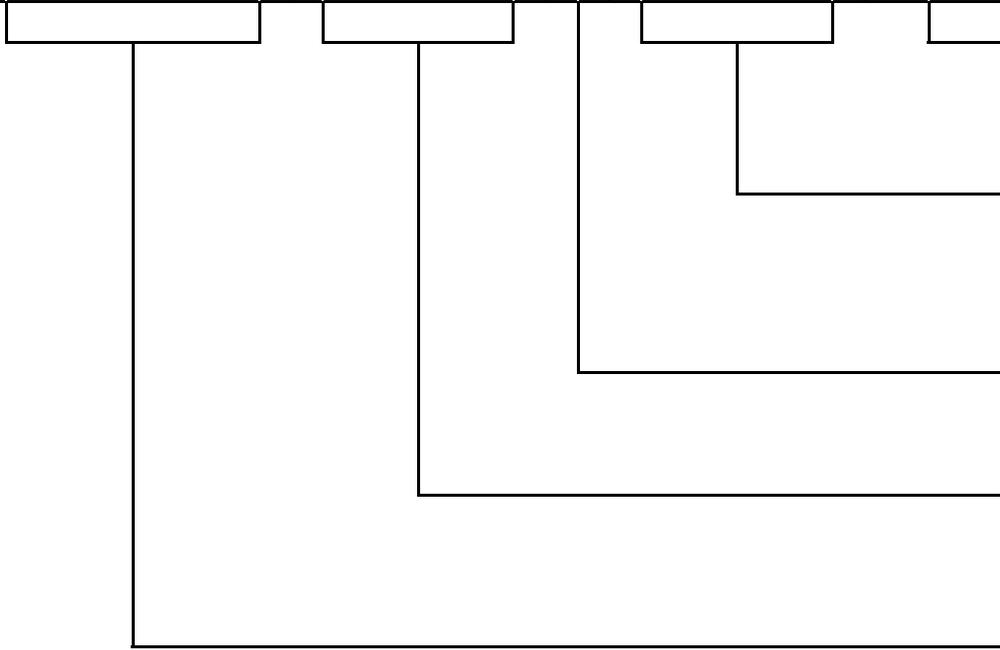
Appendix B Table 14: Current Interrupt Source Bits

INT_ACK		Description
Bit		
23 ~ 1	0	
		0 = Clears current interrupt request (IRQ)
		Unused
Note	Writing to this register acknowledges an interrupt request. It de-asserts the IRQ line to Kalimba DSP, reverts the Kalimba clock back to the non interrupt version, and initiates the search for a new interrupt	

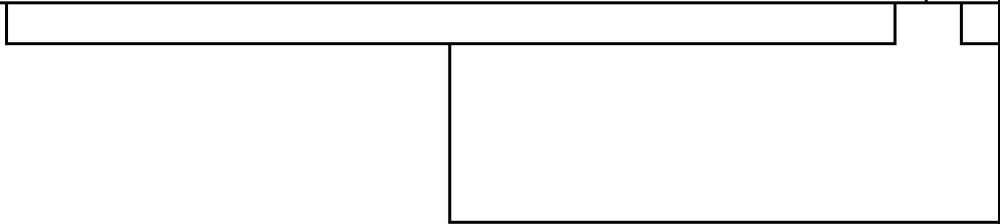
Appendix B Table15: Interrupt Request Acknowledge Bit

INT_LOAD_INFO						Description
Bit						
23 ~ 12	11	10 ~ 7	6	5 ~ 2	1 ~ 0	
						Interrupt priority value to restore see Appendix B Table 13
						Interrupt source number (0-8) to restore see Appendix B Table 12
						Interrupt active signal state to restore
						Interrupt request number (0-8) to optionally clear see Appendix B Table 12 and 14
						0 = Clear the Interrupt request number in bits 7 to 10
						Unused
Note	Used to restore information about a previous lower priority interrupt See the nested interrupts example code for an example of its use.					

Appendix B Table 16: Restore Information about a Previous Lower Priority Interrupt

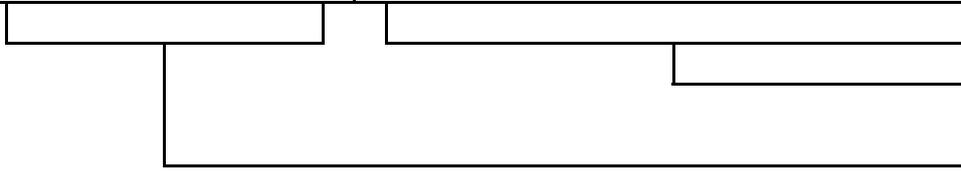
INT_SAVE_INFO					Description
Bit					
23 ~ 11	10 ~ 7	6	5 ~ 2	1 ~ 0	
					Previous lower priority interrupt priority value see Appendix B Table 13
					Previous lower priority interrupt source number (0-8) see Appendix B Table 12
					Previous lower priority interrupt active signal state
					New interrupt source number (0-8) see Appendix B Table 12.
					Unused
Note	Used to save information about the current interrupt See the nested interrupts example code for an example of its use				

Appendix B Table 17: Save Information about Current Interrupt

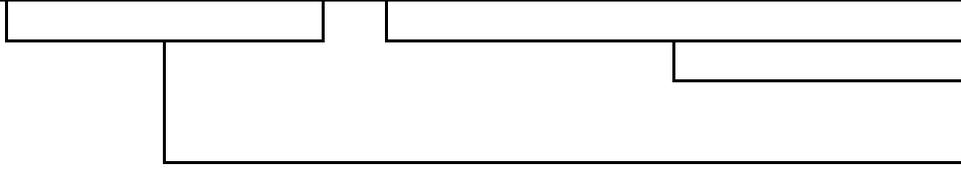
INT_SW[0/1/2/3]_EVENT		Description
Bit		
23 ~ 1	0	
		1 = Software Event (causes interrupt if enabled see Appendix B Table 12)
		Unused
Note	This register is cleared on system reset	

Appendix B Table 18: Software Event 0 to 3 Interrupt Request

MCU Interrupt DSP registers

DSP2MCU_EVENT_DATA		Description
Bit		
23 ~ 16	15 ~ 0	
		Event data from DSP to MCU
		Unused
Note	This register is cleared on system reset	

Appendix B Table 19: Interrupt Event Data to MCU

MCU2DSP_EVENT_DATA		Description
Bit		
23 ~ 16	15 ~ 0	
		Event data from MCU to DSP
		Unused
Note	This register is cleared on system reset	

Appendix B Table 20: Interrupt Event Data from MCU

PIO Control DSP Registers

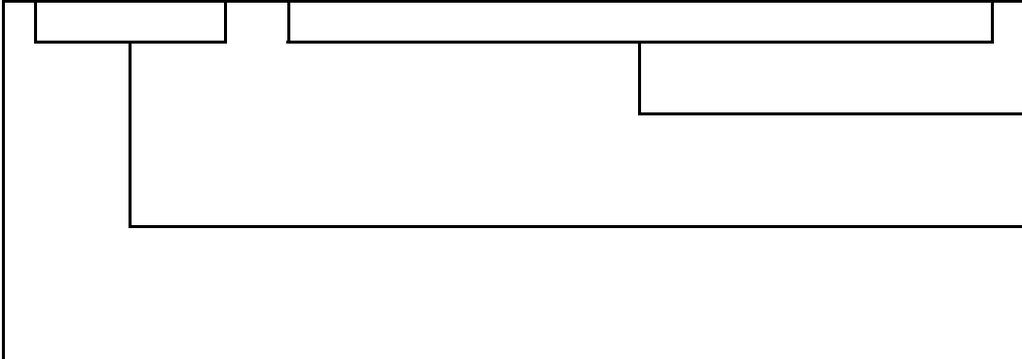
Shown below are the PIO registers accessed by the DSP:

PIO_IN		Description
Bit		
23 ~ 12	11 ~ 0	
		Reads value on PIO 0 to 11 Bit 0 = PIO 0 ↓ ↓ Bit 11 = PIO 11
		Reads value on AIO 0 to 3 Bit 0 = AIO 0 ↓ ↓ Bit 3 = AIO 3
Note	A read only register of the PIO input. This register is cleared on system reset	

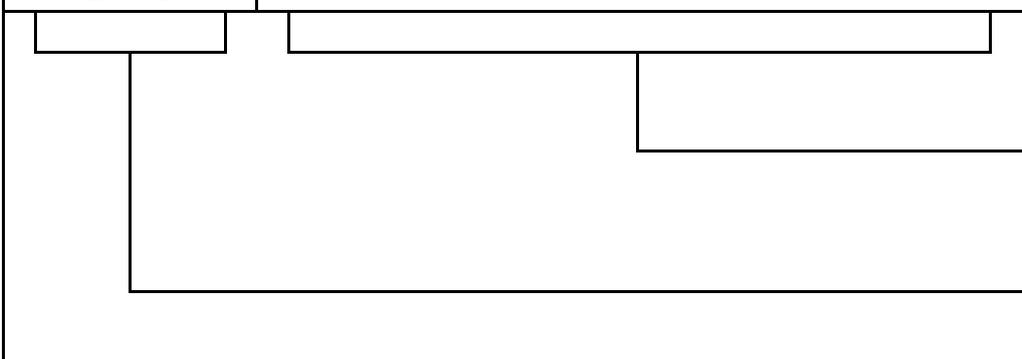
Appendix B Table 21: PIO Input Register

Note:

(1) Details of this VM function are under development

PIO_OUT		Description
Bit		
23 ~ 12	11 ~ 0	
		Write value to PIO 0 to 11 Bit 0 = PIO 0 ↓ ↓ Bit 11 = PIO 11
		Writes value to AIO 0 to 3 Bit 0 = AIO 0 ↓ ↓ Bit 3 = AIO 3
Note	Writes to the PIO lines. There is a MCU register which controls which bits of the PIO port are controlled by the MCU and which by Kalimba.	

Appendix B Table 22: PIO Output Register

PIO_EVENT_EN_MASK		Description
Bit		
23 ~ 12	11 ~ 0	
		1 = Select event change to cause an interrupt on PIO 0 to 11 Bit 0 = PIO 0 ↓ ↓ Bit 11 = PIO 11
		1 = Select event change to cause an interrupt on AIO 0 to 3 Bit 0 = AIO 0 ↓ ↓ Bit 3 = AIO 3
Note	A bit mask that selects which bits of the PIO port should cause an interrupt if PIO changes state. This register is cleared on system reset	

Appendix B Table 23: PIO Event Change Interrupt Mask Register

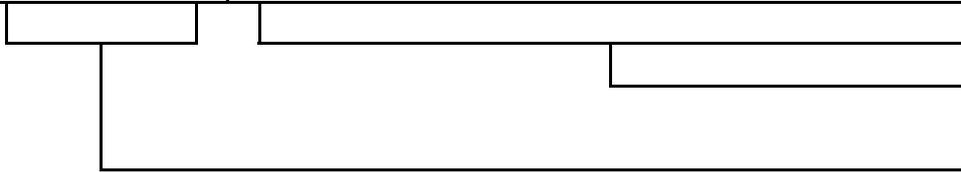
MCU Window DSP Registers

The data word size for the MCU and the DSP differ. The MCU has a 16-bit data width and the DSP has 24-bit data width. A register controls how the MCU presents data to the DSP, i.e. whether it is sign extended or not.

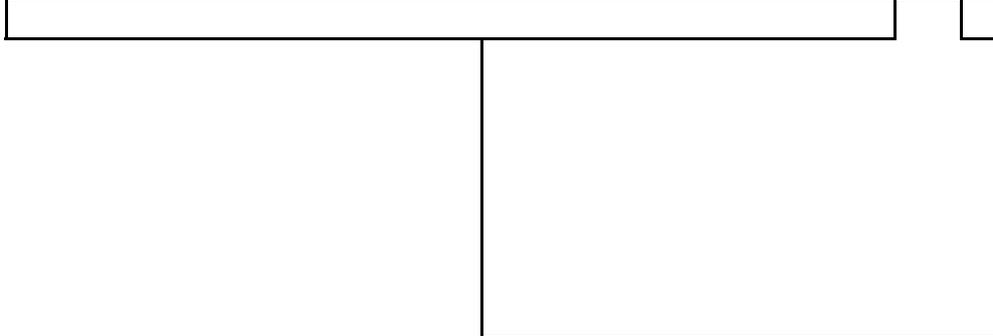
NOSIGNX_MCUWIN[1/2]		Description
Bit		
23 ~ 1	0	1 = Disable sign extension in MCU Access Window see Figure 5.1 0 = Enable sign extension in MCU Access Window see Figure 5.1
Note	Sign extension is from 16-bit MCU value to the 24-bit Kalimba DSP value This register is cleared on system reset	

Appendix B Table 24: MCU Access Window 0 to 1 Sign Extension Enable Bit

Flash Window DSP Registers

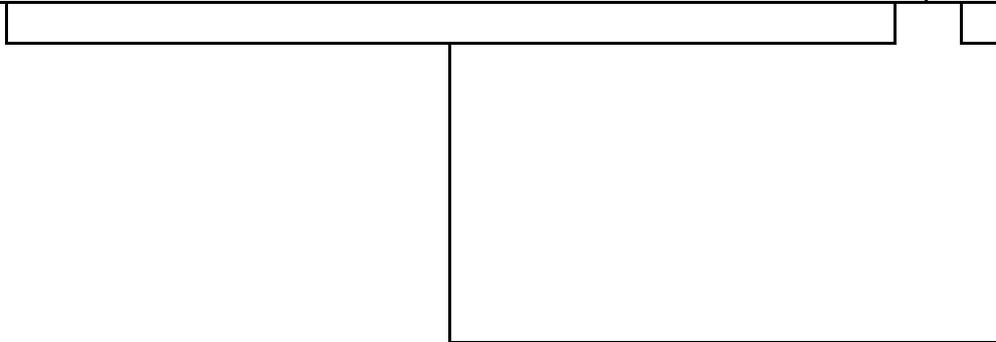
FLASH_BANK_SELECT		Description
Bit		
23 ~ 12	11 ~ 0	
		Forms most significant 12-bits of flash address
		Unused
Note	Selects which 4Kbyte block of flash is memory mapped into the DSP flash window, i.e. value forms the most significant bits of the flash address that is used	

Appendix B Table 25: 4Kbyte Block Select Register for Flash Memory Mapped into the DSP Flash Window

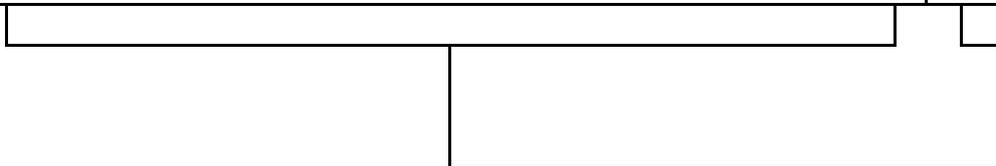
NOSIGNX_FLASHWIN		Description
Bit		
23 ~ 1	0	
		1 = Disable sign extension from Flash Window value see Figure 5.1 0 = Enable sign extension from Flash Window value see Figure 5.1
		Unused
Note	Sign extension of the 16-bit flash value to the 24-bit Kalimba DSP value. This register is cleared on system reset	

Appendix B Table 26: Flash Access Window Sign Extension Enable Bit

PM Window DSP Registers

NOSIGNX_PMWIN		Description
Bit		
23 ~ 1		0
		1 = Disable sign extension from PM Window value in DM1 see Figure 5.1 0 = Enable sign extension from PM Window value in DM1 see Figure 5.1
		Unused
Note	Sign extension of the PM values when it is windowed as two 16-bit values in the two 24-bit banks of DM in the Kalimba DSP. This register is cleared on system reset	

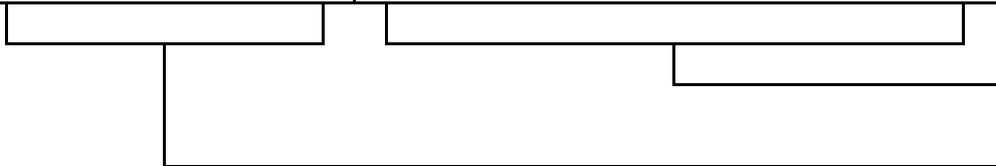
Appendix B Table 27: PM Access Window Sign Extension Enable Bit

PM_WIN_ENABLE		Description
Bit		
23 ~ 1		0
		1 = Enable PM mapping into DM1 see Figure 5.1
		Unused
Note	This register is cleared on system reset	

Appendix B Table 28: Flash Access Window Sign Extension Enable Bit

General DSP Registers

The General Registers are:

GENERAL_FROM_MCU[0/1/2/3]		Description
Bit		
23 ~ 16	15 ~ 0	
		16-bit read only registers from MCU
		Unused
Note	16-bit read only registers with their value coming from a MCU register. See below for optional sign extension to 24-bits	

Appendix B Table 29: General Register 0 to 4 from MCU to Kalimba

GENERAL_TO_MCU[0/1/2/3]		Description
Bit		
23 ~ 16	15 ~ 0	
		16-bit registers written by Kalimba and read by MCU
		Unused
Note	16-bit registers written by Kalimba, read by MCU, and seen by a VM function ⁽¹⁾ .	

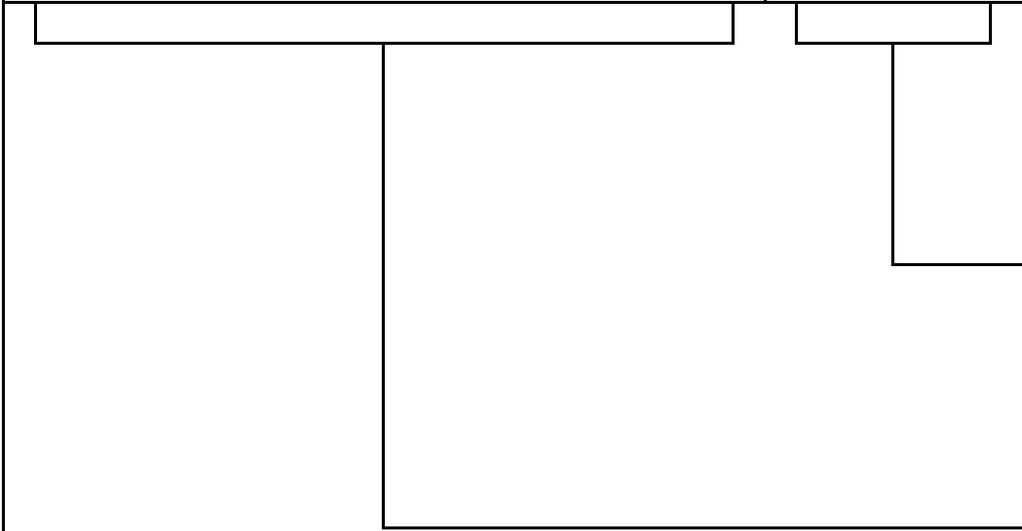
Appendix B Table 30: General Register 0 to 4 to MCU from Kalimba

Note:
⁽¹⁾ Details of this VM function are under development

NOSIGNX_GENREGS		Description
Bit		
23 ~ 1	0	
		1 = Disable sign extension from general MCU registers 0 = Enable sign extension from general MCU registers
		Unused
Note	Sign extension of the 16-bit general MCU registers to the 24-bit Kalimba DSP value. This register is cleared on system reset	

Appendix B Table 31: General MCU Registers Sign Extension Enable Bit

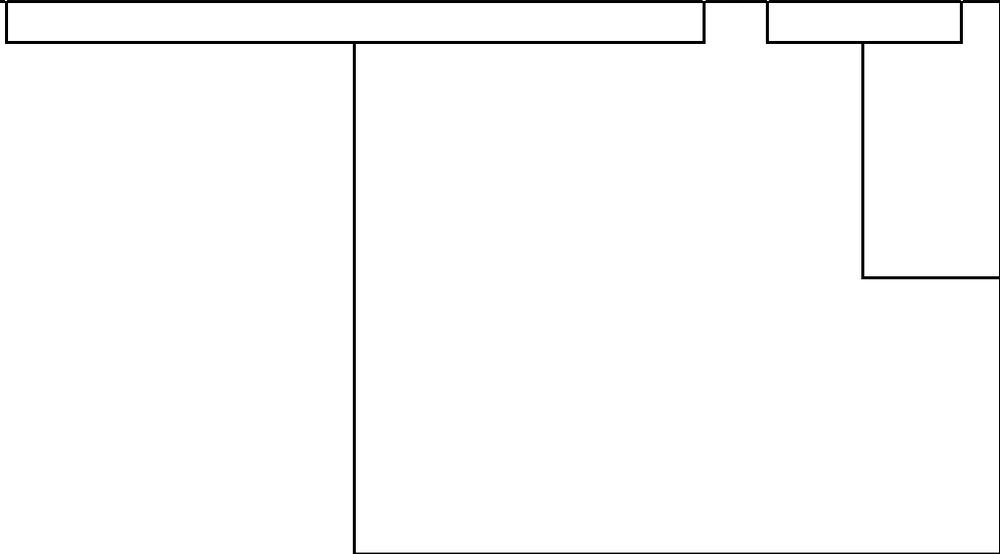
Clock Divide Rate DSP Registers

CLOCK_DIVIDE_RATE		Description
Bit		
23 ~ 4	3 ~ 0	
		Is 4-bit value, n, that sets divide ratio from the base clock frequency for the DSP. Divide ratio = 2^n e.g. $0 = \div 1^{(2)}$ $1 = \div 2$ $2 = \div 4$ $3 = \div 8$ ↓ $9 = \div 512$
		Unused
Note	This register is cleared on system reset	

Appendix B Table 32: DSP Clock Rate Register for Normal Operation

Note:

- (1) Details of this VM function are under development
- (2) Initial release of BlueCore3-Multimedia and Multimedia External is 32MHz or 32MIPs

INT_CLOCK_DIVIDE_RATE		Description
Bit		
23 ~ 4	3 ~ 0	
		Is 4-bit value, n, that sets divide ratio from the base clock frequency for the DSP during interrupt. Divide ratio = 2^n e.g. 0 = ÷ 1 1 = ÷ 2 2 = ÷ 4 3 = ÷ 8 ↓ 9 = ÷ 512
		Unused
Note	For Kalimba to use this clock frequency during interrupt INT_CLOCK_SWITCH_EN register must be enabled see Appendix B Table 10. This register is cleared on system reset	

Appendix B Table 33: DSP Clock Rate Register for Interrupt Operation

Appendix C: Software Examples

This section contains samples of example code for the Kalimba DSP.

Double-Precision Addition

```
// Double-precision addition: Z = X + Y
//
// Where:
//   X = {r0,r1};           - ie. r0 is MSW and r1 is LSW
//   Y = {r2,r3};
//   Z = {r5,r4};
// Computation time: 2 cycles

r4 = r1 + r3;           // add LSWs
r5 = r0 + r2 + Carry;  // add MSWs
```

Fractional Double-Precision Multiply

```
// Fractional double-precision multiply: Z = X * Y
//
// Where:
//   X = {r0,r1};           - ie. r0 is MSW and r1 is LSW
//   Y = {r2,r3};
//   Z = {r4,r5,r6,r7};    - ie. Z is 96-bit
// Computation time: 12 cycles

rMAC = r1 * r3 (UU);    // Compute LSW
r7 = rMAC0;            // save Z0
rMAC0 = rMAC1;         // shift right rMAC by 24-bits
rMAC12 = rMAC2;
rMAC = rMAC + r0 * r3 (SU); // compute inner products
rMAC = rMAC + r2 * r1 (SU);
r6 = rMAC0;            // save Z1
rMAC0 = rMAC1;         // shift right rMAC by 24-bits
rMAC12 = rMAC2;
rMAC = rMAC + r0 * r2 (SS); // compute MSWs
r5 = rMAC0;            // save Z2
r4 = rMAC1;            // save Z3
```

Integer Double-Precision Multiply

```
// Integer double-precision multiply: Z = X * Y
//
// Where:
//   X = {r0,r1};           - ie. r0 is MSW and r1 is LSW
//   Y = {r2,r3};
//   Z = {r4,r5,r6,r7};    - ie. Z is 96-bit
// Computation time: 12 cycles

rMAC = r1 * r3 (UU);    // Compute LSW
r7 = rMAC LSHIFT 15;   // save Z0
rMAC0 = rMAC1;         // shift right 24-bits
rMAC12 = rMAC2;
rMAC = rMAC + r0 * r3 (SU); // compute inner products
rMAC = rMAC + r2 * r1 (SU);
r6 = rMAC LSHIFT 15;   // save Z1
rMAC0 = rMAC1;         // shift right 24-bits
rMAC12 = rMAC2;
rMAC = rMAC + r0 * r2 (SS); // compute MSWs
r5 = rMAC LSHIFT 15;   // save Z2
r4 = rMAC LSHIFT -1;   // save Z3
```

FIR Filter

```
// FIR filter
//
// Input parameters:
// I0 = points to oldest input value in delay line
// L0 = filter length (N)
// I4 = points to beginning of filter coefficient table
// L4 = filter length (N)
// r10 = filter length - 1 (N-1)
// Return values:
// rMAC = sum of products output
// Computation time: N + 2 cycles

fir_filter:
rMAC = 0
r1 = M[I0,1]
r2 = M[I4,1];
Do fir_loop;
    rMAC = rMAC + r1 * r2
    r1 = M[I0,1]
    r2 = M[I4,1];
fir_loop:
rMAC = rMAC + r1 * r2;
```

Cascaded Bi-Quad IIR Filter

```

// Cascaded biquad IIR filter
//
// Equation of each section:
//  $y(n) = (b_0*x(n) + b_1*x(n-1) + b_2*x(n-2) - a_1*y(n-1) - a_2*y(n-2)) \ll \text{scalefactor}$ 
//
// Input Values:
// r0 = input sample
// I0 = points to oldest input value in delay line
//      (no biquads*2 + 2)
// I1 = points to a list of scale factors for each biquad section
// I4 = points to scaled coefficients b2,b1,b0,a2,a1,... etc
// L0 = 2 * num_biquads + 2
// L1 = num_biquads
// L4 = 5 * num_biquads
// M0 = -3
// M1 = 1
// r10 = num_biquads
// Return Values:
// r0 = output sample
// r10 - cleared
// I0,I1,I4,L0,L1,L4,M0,M1 - unaffected
// r1,r2,r3,r4 - affected
//
// Computation time: 8 * num_biquads + 3

biquad_filter:
do biquad_loop;
  r1 = M[I0,1]           // get x(n-2)
  r2 = M[I4,1];         // get coef b2
  rMAC = r1 * r2
  r3 = M[I0,1]           // get x(n-1)
  r2 = M[I4,1];         // get coef b1
  rMAC = rMAC + r3 * r2
  r4 = M[I1,1]           // get scalefactor
  r2 = M[I4,1];         // get coef b0
  rMAC = rMAC + r0 * r2
  r1 = M[I0,1]           // get y(n-2)
  r2 = M[I4,1];         // get coef a2
  rMAC = rMAC - r1 * r2
  r1 = M[I0,M0]          // get y(n-1)
  r2 = M[I4,M1];        // get coef a1
  rMAC = rMAC - r1 * r2
  M[I0,1] = r3;         // store new x(n-2)
  r0 = rMAC ASHIFT r4
  M[I0,M1] = r0;        // store new x(n-1)
biquad_loop:
M[I0,1] = r1;           // store new y(n-2)
M[I0,1] = r0;           // store new y(n-1)

```

Radix-2 FFT

```

// An optimised FFT subroutine with a simple interface
//
// Input Values:
//   $fft_npts   - Number of points (a power of 2)
//   $Inputreal  - Input array real parts
//   $Inputimag  - Input array imag parts
//   L0,L1,L4,L5 - should be initialised to 0.
//
// Return Values:
//   $Refft      - Output array real parts
//   $Inputreal  - Output array imag parts
// All registers altered
//
// Computation time:
//
// No clock cycles = (2^n)*(4n+8.5) + 16n + 10
// where n = log2(fft_npts)
//
// fftnpts:      64      128      256      512
//
// No Clks:      2186    4794    10506    22938

// Declare local variables:
.VAR groups;
.VAR node_space;
#include "twiddle_factors.h"

// -- ENTRY POINT --
fast_fft:

M0 = 0;
M1 = 1;

// -- Process the n-1 stages of butterflies --
r1 = 1;
M[groups] = r1;           // groups = 1
r0 = M[$fft_npts];
r0 = r0 ASHIFT -1;
M[node_space] = r0;      // node_space = Npts / 2

r0 = SIGNDDET r0;
r1 = 22;
r9 = r1 - r0;           // log2(Npts) - 1
stage_loop:

    r10 = M[node_space];
    M2 = r10;           // M2 = node_space

    r8 = M[groups];
    r2 = r8 LSHIFT 1;
    M[groups] = r2;     // groups = groups * 2;

    I0 = &twid_imag;   // I0 -> (-S) of W0
    I2 = &$Inputreal;  // I2 -> x0 in 1st group of stage
    I1 = I2 + M2;       // I1 -> x1 in 1st group of stage
    I6 = &$Inputimag;  // I6 -> y0 in 1st group of stage
    I3 = I6 + M2;       // I3 -> y1 in 1st group of stage
    I4 = &twid_real;   // I4 -> C of W0
  
```

```

group_loop:
    r2 = M[I4,1];
    r6 = r2;                // r6=C

    r3 = M[I1,0];          // r3=x1

    rMAC = r3 * r6         // rMAC=x1*C
    r5 = M[I3,0];         // r5=y1

    r2 = M[I0,1];
    r7 = r2;              // r7=(-S)

    r10 = M[node_space];

    DO bfly_loop;
        rMAC = rMAC - r5 * r7 // rMAC=x1*C-y1*-S
        r0 = M[I2,0];        // r0=x0

        r1 = r0 + rMAC       // r1=x0'=x0+(x1*C-y1*-S)
        r2 = M[I3,M1];      // r2=y1 (dummy read)

        r1 = r0 - rMAC       // r1=x1'=x0-(x1*C-y1*-S)
        M[I2,M1] = r1;      // DM=x0'

        rMAC = r3 * r7       // rMAC=x1*(-S)
        M[I1,M1] = r1        // DM=x1'
        r4 = M[I6,M0];       // r4=y0

        rMAC = rMAC + r5 * r6 // rMAC=x1*(-S)+y1*C
        r5 = M[I3,-1];      // r5=next y1

        r1 = r4 + rMAC       // r1=y0'=y0+(y1*C+x1*(-S))
        r3 = M[I1,M0];      // r3=next x1

        r4 = r4 - rMAC       // r1=y1'=y0-(y1*C+x1*(-S))
        M[I6,M1] = r1;      // DM=y0'

        rMAC = r3 * r6       // rMAC=x1*C
        M[I3,M1] = r4;       // DM=y1'
    bfly_loop:

    r2 = M[I1,M2];          // move: x1, x0, y1, and y0
    r3 = M[I3,M2];          // onto next group with dummy reads
    r8 = r8 - M1
    r2 = M[I2,M2]
    r3 = M[I6,M2];
    if NZ jump group_loop;

    r10 = M[node_space];
    r10 = r10 ASHIFT -1;    // node_space = node_space / 2;
    M[node_space] = r10;

    r9 = r9 - 1;
    if NZ jump stage_loop;

    // -- Process the last stage of butterflies separately --
    I0 = &twid_imag;        // I0 -> (-S) of W0
    I5 = &$Inputreal;       // I2 -> x0
    I1 = I5 + 1;            // I1 -> x1
    M2 = 2;
    I3 = BITREVERSE(&$Refft); // Refft bitreversed

    r0 = M[$fft_npts];
    r0 = SIGNDDET r0;
    r0 = r0 - 7;
    r1 = 1;
    r1 = r1 LSHIFT r0;
    M3 = r1;                // Bitreversed modifier

    I4 = &twid_real;        // I4 -> C of W0
    I6 = &$Inputimag;       // I6 -> y0
    I2 = I6 + 1;            // I5 -> y1
  
```

```

r2 = M[I4,M1]
r5 = M[I2,M2];           // r5=y1
r6 = r2                  // r6=C
r3 = M[I1,M2];           // r3=x1

rMAC = r3 * r6           // rMAC=x1*C
r2 = M[I0,1];           // r2=(-S)

r10 = M[$fft_npts];
r10 = r10 ASHIFT -1;     // Npts / 2

DO last_loop;
  rMAC = rMAC - r5 * r2   // rMAC=x1*C-y1*-S
  r0 = M[I5,M2];         // r0=x0

  r1 = r0 + rMAC;        // r1=x0'=x0+(x1*C-y1*-S)

  // enable Bit Reverse addressing on AG1
  rFlags = rFlags OR BR_FLAG;

  r1 = r0 - rMAC         // r1=x1'=x0-(x1*C-y1*-S)
  M[I3,M3] = r1;        // DM=x0'

  rMAC = r3 * r2         // rMAC=x1*(-S)
  M[I3,M3] = r1         // DM=x1'
  r4 = M[I6,M0];        // r4=y0

  // disable Bit Reverse addressing on AG1
  rFlags = rFlags AND NOT_BR_FLAG;

  rMAC = rMAC + r5 * r6   // rMAC=x1*(-S)+y1*C
  r2 = M[I4,M1]         // r2=C;
  r3 = M[I1,M2];        // r3=next x1

  r1 = r4 + rMAC         // r1=y0'=y0+(y1*C+x1*(-S))
  r5 = M[I2,M2];        // r5=next y1

  r4 = r4 - rMAC         // r1=y1'=y0-(y1*C+x1*(-S))
  M[I6,M1] = r1;        // DM=y0'

  r6 = r2                // r6=C
  r2 = M[I0,M1];         // r2=(-S)

  rMAC = r3 * r6         // rMAC=x1*C
  M[I6,M1] = r4;        // DM=y1'

last_loop:

I3 = BITREVERSE(&$Inputreal);
I5 = &$Inputimag;

// enable Bit Reverse addressing on AG1
rFlags = rFlags OR BR_FLAG;

r2 = M[I5,1];
r10 = N;
DO bit_rev_imag;
  r2 = M[I5,M1]
  M[I3,M3] = r2;
bit_rev_imag:

// disable Bit Reverse addressing on AG1
rFlags = rFlags AND NOT_BR_FLAG;
rts;

```

Document References

Document:	Reference, Date:
BlueCore3-Multimedia Kalimba DSP Assembler User Guide	bc3-ug-002Pd, May 2005
BluLab xIDE userguide	blab-ug-002Pa, June 2005
BlueCore3-Multimedia Data Sheet	BC358239A-ds-001Pf, July 2005

Acronyms and Definitions

BlueCore™	Group term for CSR's range of Bluetooth chips
Bluetooth®	Wireless technology providing audio and data transfer over short-range radio connections
CODEC	COder DECoder
CSR	Cambridge Silicon Radio Limited
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIR	Finite Infinite Response filter
IIR	Infinite Impulse Response filter
ISR	Interrupt Service Routine
Kalimba	A CSR DSP core architecture
Kalasm2	Product name for BlueCore3-Multimedia Kalimba DSP Core Assembler
LS	Least Significant
LSW	Least Significant Word
MAC	Multiply ACcumulate
MS	Most Significant
MSW	Most Significant
NOP	No Operation
PC	Program Counter
RTI	ReTurn from Interrupt
RTS	ReTurn from Subroutine

Record of Changes

Date:	Revision	Reason for Change:
01 SEP 03	a	Original publication of this document. (CSR reference: bc3-ug-001Pa)
14 OCT 03	b	Minor modifications and clarification of divide instruction
23 JUN 05	c	Minor modifications and some re-ordering of information.

BlueCore3-Multimedia Kalimba DSP

User Guide

bc3-ug-001Pc

June 2005

Unless otherwise stated, words and logos marked with [™] or [®] are trademarks registered or owned by Cambridge Silicon Radio Limited or its affiliates. Bluetooth[®] and the Bluetooth logos are trademarks owned by Bluetooth SIG, Inc. and licensed to CSR. Other products, services and names used in this document may have been trademarked by their respective owners.

The publication of this information does not imply that any license is granted under any patent or other rights owned by Cambridge Silicon Radio Limited.

CSR reserves the right to make technical changes to its products as part of its development programme.

While every care has been taken to ensure the accuracy of the contents of this document, CSR cannot accept responsibility for any errors.

CSR's products are not authorised for use in life-support or safety-critical applications.