Push every boundary.™

BlueCore® Kalimba Architecture 5

Kalimba DSP

User Guide

Issue 1

CSR

Document History

Revision	Date	Change Reason
1	19 FEB 15	Original publication of this document. If you have any comments about this document, email comments@csr.com giving the number, title and section with your feedback.

Contents

List of Figures

List of Tables

1 Introduction

This user guide is for developers of software applications and algorithms for the fifth-generation Kalimba DSP coprocessor, as implemented on selected CSR devices.

It documents the architecture of the Kalimba DSP, instruction set description and peripheral features, and includes some example code. Read this document in conjunction with the other Kalimba DSP tools documents that are available.

Kalimba DSP is present in many CSR devices. The Kalimba Architecture 5 DSP particularly targets audio processing applications. Example audio processing applications include:

- SBC encoding and decoding, as defined in the Bluetooth® A2DP
- MP3 encoding and decoding, as defined in ISO/IEC 11172-3, and the sample rate extensions defined in ISO/IEC 13818-3
- AAC encoding and decoding, as defined in ISO/IEC 13818-7
- CSR aptX™ encoding and decoding
- Alternative voice/Hi-Fi codecs
- Echo and noise cancellation
- Audio signal enhancement:
  - Stereo enhancement
  - Equaliser
  - Lost packet concealment
- Text-to-speech
- Voice recognition

The Kalimba DSP is also suitable for non-audio applications such as picture compression, image processing, communications protocols and general-purpose application code.

Table 1.1 shows Kalimba architecture development.

Kalimba Architecture	Products
1	BlueCore3-Multimedia
2	BlueCore5-Multimedia
3	CSR8311, CSR860x, CSR861x, CSR862x, CSR863x, CSR864x, CSR8670, CSR8810, CSR8811, CSR8820
5	CSR8675

Table 1.1: Kalimba Architecture Development

2 Key Features

The key features of the Kalimba DSP core include:

- 24-bit fixed point DSP core
- 120 MHz performance, which can be divided down for power saving
- 1 program memory and 2 data memory banks, all 3 of which can be accessed simultaneously in a single cycle
- Flash/ROM support for both data and code, with caches to improve code performance
- Single-cycle 24 x 24-bit multiply with 2 56-bit accumulators
- Single-cycle barrel shifter with 56-bit input and 56-bit or 24-bit output
- 13-cycle divide (performed in the background)
- Majority of instructions can be conditional
- Zero overhead ring buffer indexing
- Zero overhead looping and unconditional branching
- Bit reversed addressing capability, and bit reverse data function
- Largely orthogonal instruction set, which is quick to learn and easy to write in the algebraic assembler language
- Stack instructions: `PUSH`, `POP`, `PUSHM`, `POPM`, `FP/SP Adjust`, `FP/SP Relative LOAD/STORE`, and other instructions featuring overflow detection
- Low-power internal architecture
- 8 hardware program breakpoints and 2 data breakpoints (each covering an address range start to end)

The key features of the Kalimba DSP peripherals include:

- Close integration with the on-chip MMU, giving access to features such as DACs and ADCs
- 12 low-overhead read/write ports to transfer streaming data to and from the BlueCore subsystem
- 2 memory-mapped windows into the MCU RAM for data exchange
- 3 windows for access to the flash/ROM data memory
- Memory-mapped interface to the I/O address map
- Multiple interrupt sources including 2 32-bit timers
- Read, write and direction control access to external PIO lines

3 System Overview

All Kalimba Architecture 5 devices contain 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
- MMU interface
- Programmable I/O control
- Interrupt control
- Clock source
- Timers
- Debug interface

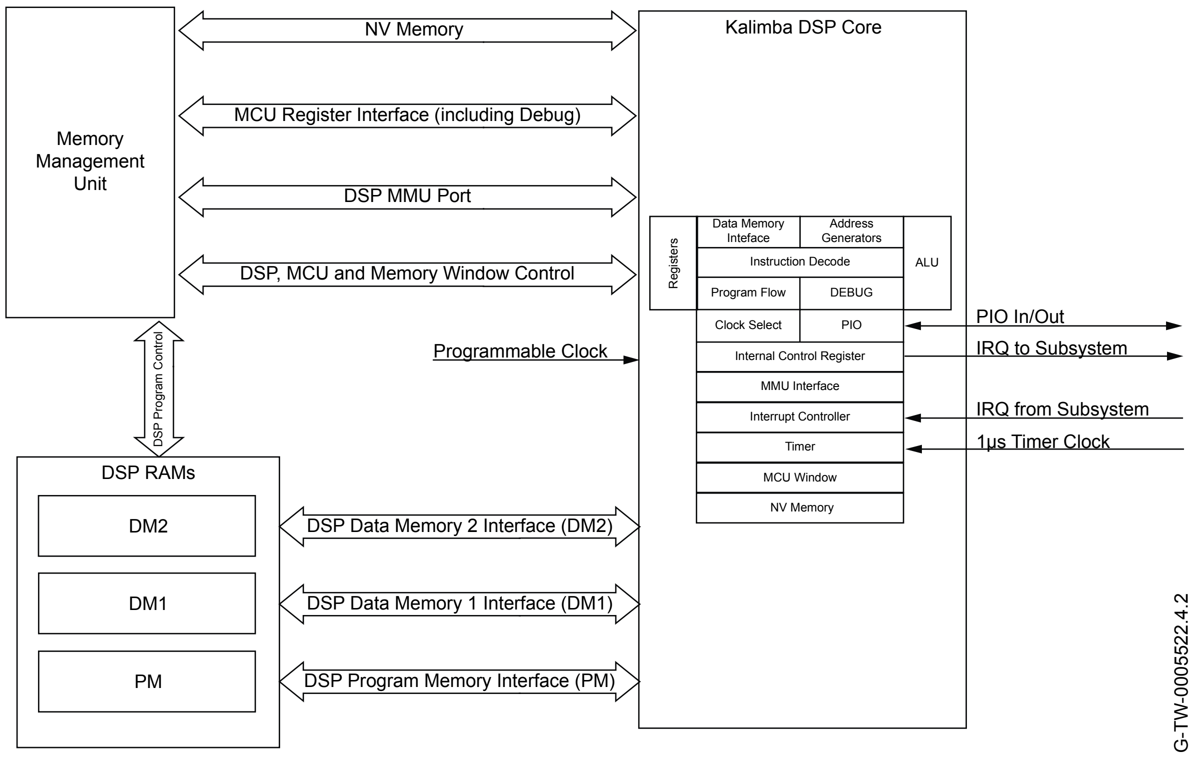


Figure 3.1: Kalimba DSP Coprocessor Subsystem

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 audio codec data to enhance audio applications. The Kalimba DSP is also available as a general-purpose processor. Figure 3.1 and Section 8 show how the DSP interfaces to other functional blocks within the BlueCore device.

3.2 Kalimba DSP Memory

The Kalimba DSP contains the following on-chip RAM:

- DM1: 24-bit data memory 1
- DM2: 24-bit data memory 2
- PM: 32-bit program memory, with 1024-word or 512-word direct-mapped cache for flash/ROM program space and 32-word loop cache

Both data and code have a 24-bit address width. Section 5 shows DSP RAM sizes for an example BlueCore IC. Separate documents describe specific product information.

3.3 Kalimba DSP Peripherals

3.3.1 Memory Management Unit Interface

The MMU interface consists of a number of read and write ports that can efficiently transfer streams of data to and from the rest of the IC.

3.3.2 Programmable I/O Control

BlueCore ICs have up to 32 programmable I/O lines controlled by firmware running on the device. The Kalimba DSP core can read any digital I/O directly but can only write to or change the pin direction of digital outputs that the MCU has enabled (carried out through the VM application, if present). Section 8.5 describes the I/O control.

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 3 levels. Alongside the interrupts caused by hardware, there are 4 software event interrupts available. Interrupts are serviced between 3 and 5 instructions after the interrupt request line goes high (up to and including the automatic branch to the beginning of the interrupt handler).

3.3.4 Clock Source Select and Timers

The Kalimba DSP has a clock source select interface. This clock-rate divider circuit is controllable from both the Kalimba DSP core and the MCU. The Kalimba DSP also has 2 timers with a 1µs time base available.

3.3.5 Debug Interface

The BlueCore device contains a hardware interface that assists in the debugging of applications running on the Kalimba DSP core.

3.4 Kalimba Architecture 5 Changes

Section 3.4.1 to Section 3.4.2 show the changes in Kalimba Architecture 5 with respect to Kalimba Architecture 3.

3.4.1 Interrupt controller

The CSR8675 devices have modifications to the interrupt controller. In previous devices, each interrupt source could be mapped to three different priority levels, which enabled higher priority interrupt sources to interrupt lower priority sources.

The implementation in CSR8675 is slightly different. Each interrupt source is routed into the interrupt controller several times, once for each priority level. For example, the PIO interrupt appears three times: `$INT_LOW_PRI_SOURCES_EN_PIO_EVENT_POSN`, `$INT_MED_PRI_SOURCES_EN_PIO_EVENT_POSN` and `$INT_HIGH_PRI_SOURCES_EN_PIO_EVENT_POSN`. The functionality is the same as CSR8670, higher priority interrupts interrupt lower priority ones.

The reference libraries provided by CSR support the new hardware and maintain the current API. Applications that use the CSR libraries will not be impacted by these hardware changes.

3.4.2 Program Memory Addressing

CSR8675 addresses program memory in 8-bit words. In previous devices, program memory is addressed in 32-bit words. This results in two changes:

- Program counter increments
- Region sizes in link scripts

The program counter increments differently in CSR8675. In linear code execution the program counter increments by 4. In previous devices the program counter increments by 1.

Regions defined in link scripts are now 4 times larger. The amount of memory is exactly the same. On the CSR8670, 1 KB of memory is listed as 256 words. On the CSR8675, 1 KB is listed as 1024 words. In both cases, exactly the same amount of memory is used.

This means regions defined in the link scripts may appear larger, but are the same as in the CSR8670.

As the linker resolves program memory symbols, no code needs to be updated. Once code is reassembled the linker automatically updates any references. However, if an application uses custom link scripts, these will need updating. The standard link scripts provided by CSR have been updated.

4 Kalimba DSP Core Architecture

Figure 4.1 shows the Kalimba DSP core architecture.

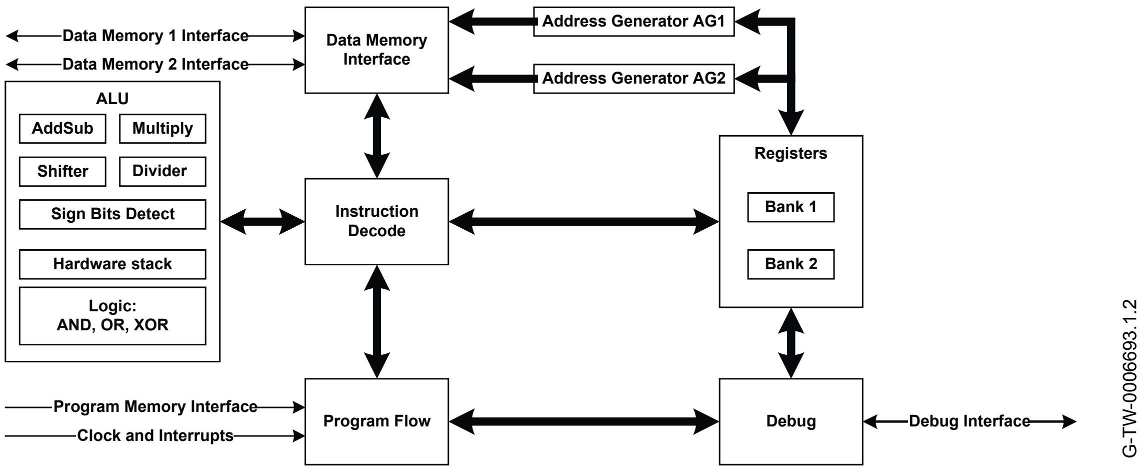


Figure 4.1: Kalimba DSP Core Base Architecture

4.1 Arithmetic Logic Unit

The ALU performs the following functions:

- Add and subtract arithmetic
- Logic operations: `AND`, `OR`, and `XOR`
- Single-cycle multiply, multiply/add and multiply/subtract
- Single-cycle multiply, multiply/add and multiply/subtract with parallel add/subtract
- Logical and arithmetic shift operations with up to 56-bit output and up to 56-bit input
- 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)

  Note:

  This is a background operation, taking 13 clock cycles.
- Stack instruction control: `PUSH`, `POP`, `PLOOK`, `PUSHM`, `POPM` with overflow detection

4.2 Address Generators (AGs)

The AGs form data memory addresses for indexed memory reads or writes. Each AG has 4 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 2 independent AGs, the DSP can generate 2 addresses simultaneously for dual indexed memory accesses.

Length values can be associated with 4 of the index registers to implement automatic modulo addressing for circular buffers. Start addresses may also be associated, removing the need for circular buffer alignment.

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 efficient addressing in Radix-2 FFT algorithms; see the Radix-2 FFT code in Appendix C for an example of its use.

4.3 Registers

There are 3 banks of 16 registers:

- Bank 1 is general registers available to virtually all instructions described in Section 4.4.
- Bank 2 registers are for the control of index memory accesses described in Section 4.7.
- Bank 3 registers, described in Section 4.8, can be altered using the stack instructions described in Section 6.15.

4.4 Bank 1 Registers

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

No.	Name	# bits	Description
0	`Null`	N/A	Always read as 0, writing only affects flags (so can be used for condition testing)
1	`rMAC`	24/56	The 56 bits are used for multiply accumulate instructions and the input to shift operations. For 24-bit operations: ■ Read as bits [47:24] with saturation and unbiased rounding[(a)] ■ Written as bits [47:24] with sign extension and trailing 0 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`	24	Call instructions put the return PC address in this register for use by `rts` instructions.[(b)]
14	`rFlags`	24	Status and mode flags. See below for a description.
15	`rMACB`	24/56	The 56 bits are used for multiply accumulate instructions and the input to shift operations. For 24-bit operations: ■ Read as bits [47:24] with saturation and unbiased rounding[(a)] ■ Written as bits [47:24] with sign extension and trailing 0 padding to make 56-bits

Table 4.1: Bank 1 Registers

[(a)] Unbiased rounding is:

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

When `rMAC`[23:0] is exactly at the midpoint (`0x800000`), 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.

[(b)] There is no hardware subroutine stack, to enable multi-depth subroutine calls the return `PC` addresses are usually stored on the general-purpose hardware stack.

Note:

Only registers `rMAC` and `r0` to `r5` can be used by indexed memory access instructions.

All registers are set to 0 on Kalimba reset.

4.5 rFlags Register

The `rFlags` register is a 24-bit register that is located in register bank 1 of Kalimba, see Table 4.1. Table 4.2 shows the individual bits that make up the `rFlags` register and their value after reset. This register has a natural split into 3 bytes:

- The LS byte contains the presently active flags used by Kalimba.
- The middle byte contains a stored value of the flags.
- The MS byte has no meaning.

The flags are stored when an interrupt has occurred and then restored after Kalimba has finished servicing the interrupt. The `INT_` versions of the various flags are the copies that are stored at the point of interrupt service. On an interrupt, the LS byte is copied to the middle byte, which stores a copy of the flags. The `rti` instruction then automatically restores the flags to their previous value.

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

Table 4.2: rFlags Register

4.5.1 Negative Flag (N)

This is set if the result of the instruction is negative, i.e. if the most significant bit is set, and cleared otherwise.

4.5.2 Zero Flag (Z)

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

4.5.3 Carry Flag (C)

The state of the carry flag is:

- For an addition, `C` is set if the addition produced a carry, i.e. an unsigned overflow, and is cleared otherwise.
- For a subtraction, `C` is cleared if the subtraction produces a borrow, i.e. an unsigned underflow, and is set otherwise.
- For other operations (including multiply accumulate), `C` is cleared.

4.5.4 Overflow Flag (V)

The state of the overflow flag is:

- For addition, subtraction, arithmetic shifts, integer multiplies and multiply accumulates, `V` is set if signed overflow occurred, regarding the operands and result as 2's complement signed integers, and is cleared otherwise.
- The setting/clearing of the `V` flag for the `rMAC` and/or `rMACB` registers occurs if there is overflow past the 56th bit, whereas for the 24-bit registers it is if overflow occurs past the 24th bit.
- `V` cleared for other operations.

4.5.5 Sticky Overflow Flag (SV)

Set whenever `V` is set but can only be cleared by software by explicitly writing to the `rFlags` register. If `V` is set by explicitly writing to the `rFlags` register, `SV` will be set if the next instruction is a `nop` or a `prefix` instruction is executed, e.g. not on a type A instruction where the condition is not satisfied.

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 code sections to improve speed and code clarity where a particular instruction needs to be executed conditionally.

4.5.7 Bit Reverse Flag (BR)

If set, the output of `AG1` (index registers `I0` to `I3`) is bit-reversed before being driven to the address bus. Here bit-reversing applies to the least significant 23 bits. The most significant bit selects the data memory used so it stays in place.

4.5.8 User Mode Flag (UM)

If set, interrupts are serviced. On entry to the interrupt service routine (default `PC` address `0x0002`), this flag is cleared so no further interrupts are serviced unless the flag is manually set, for example to support interrupt priority. Execution of a `rti` instruction sets this flag to the value of `INT_UM_FLAG` (normally set unless altered in software).

4.5.9 rFlags Behaviour During Interrupts

Before entering the interrupts, each element of `rFlags` is copied into its interrupt duplicate, e.g. `UM_FLAG` is copied into `INT_UM_FLAG`. With the exception of the `UM_FLAG`, all bits remain unchanged. `UM_FLAG` is cleared and remains cleared unless software enables it. When returning from interrupt, using the `rti` instruction, the interrupt bits are copied back, over-writing the non-interrupt values.

4.5.10 Condition Codes

The state of the flags present in the `rFlags` register forms the basis of the condition codes supported by conditional instructions in the Kalimba, see Table 4.3.

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) / **NB** (not ALU borrow)	C = 1	0 0 1 0
NC (not ALU carry) / **B** (ALU borrow)	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 and rMACB Registers

The `rMAC` and `rMACB` registers are 56-bit registers that are located in register bank 1 of Kalimba, see Table 4.1. The `rMAC(B)` registers split into a set of separately accessible sub-registers. Figure 4.2 shows the size of these registers:

- `rMAC(B)` is the overall 56-bit register
- `rMAC(B)0` is a 24-bit register that forms the lower part of the `rMAC(B)` register
- `rMAC(B)1` is a 24-bit register that forms the middle part of the `rMAC(B)` register
- `rMAC(B)2` is an 8-bit register that forms the higher part of the `rMAC(B)` register
- `rMAC(B)12` is a 32-bit register that is a combination of `rMAC(B)2` and `rMAC(B)1` that forms part of the `rMAC(B)` register
- The 24-bit rounded and saturated version of `rMAC(B)` is often referred to as `rMAC(B)24`. See Section 4.4 for details of the rounding operation.

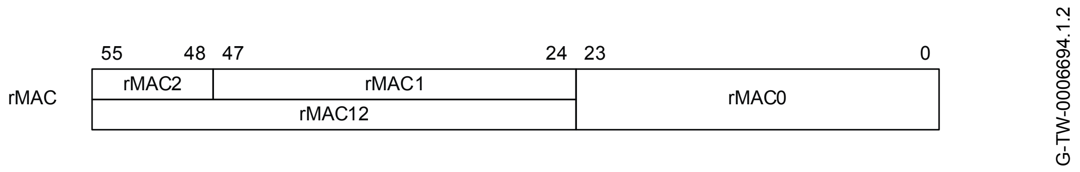


Figure 4.2: rMAC and rMACB Sub-register Definitions

4.7 Bank 2 Registers

Bank 2 registers control index memory accesses, and set up modulo addressing through a circular memory buffer. Table 4.4 lists the registers.

No.	Name	# bits	Description
0	`I0`	24	Index register for `AG1`
1	`I1`	24	Index register for `AG1`
2	`I2`	24	Index register for `AG1`
3	`I3`	24	Index register for `AG1`
4	`I4`	24	Index register for `AG2`
5	`I5`	24	Index register for `AG2`
6	`I6`	24	Index register for `AG2`
7	`I7`	24	Index register for `AG2`
8	`M0`	24	Modify register for any index register
9	`M1`	24	Modify register for any index register
10	`M2`	24	Modify register for any index register
11	`M3`	24	Modify register for any index register
12	`L0`	24	Length register for Index register `I0`
13	`L1`	24	Length register for Index register `I1`
14	`L4`	24	Length register for Index register `I4`
15	`L5`	24	Length register for Index register `I5`

Table 4.4: Bank 2 Registers

Note:

Even though the modify and length registers are 24-bit, only the LS 16-bits are used for indexed memory accesses, i.e. circular buffers greater than 64Kword in size are not supported.

All registers are set to 0 on DSP reset.

4.7.1 Index Registers

An index register contains an address pointer to data memory and is for indexed addressing. These registers allow transfer of data to and from selected Bank 1 registers using the address contained within this register. For more information see Section 6.18. The index registers `I0` to `I3` are associated with `AG1` (address generator 1) and `I4` to `I7` are associated with `AG2` (address generator 2).

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 a modify register, or by a 2-bit constant.

4.7.3 Length Registers

Length register values are associated with 4 of the index registers, see Table 4.4, and are for implementing automatic modulo addressing for circular buffers. To disable automatic modulo addressing, set the corresponding length register to 0.

4.8 Bank 3 Registers

Table 4.5 lists the bank 3 registers.

The `rMAC(B)` sub-registers are available in `LSHIFT`, `ASHIFT` and `rMAC(B)` move instructions, see Section 6.5 and Section 6.7. The `DivResult` and `DivRemainder` registers can be accessed using `Divide` instructions, see Section 6.14. The stack instructions can use the `SP` and `FP` registers, which are particularly useful for subroutine entry and exit.

No.	Name	# bits	Description
0	`rMAC2`	8	The MS 8 bits of `rMAC`
1	`rMAC12`	24	The middle 24 bits of `rMAC` (with sign extension into `rMAC2` when popping)[a]
2	`rMAC0`	24	The LS 24 bits of `rMAC`
3	`DoLoopStart`	24	Start address for `do…loops`. Identical to memory-mapped `MM_DOLOOP_START`
4	`DoLoopEnd`	24	End address for `do…loops`. Identical to memory-mapped `MM_DOLOOP_END`
5	`DivResult`	24	Divider result. Identical to memory-mapped `MM_QUOTIENT`
6	`DivRemainder`	24	Divider remainder. Identical to memory-mapped `MM_REM`
7	`rMACB2`	8	The MS 8 bits of `rMACB`
8	`rMACB12`	24	The middle 24 bits of `rMACB` (with sign extension into `rMACB2` when popping)[b]
9	`rMACB0`	24	The LS 24 bits of `rMACB`
10	`B0`	24	Base register 0 (when non-zero, sets the base address of circular buffers and bit-reverse arrays)
11	`B1`	24	Base register 1
12	`B4`	24	Base register 4
13	`B5`	24	Base register 5
14	`FP`	24	Frame pointer. Identical to memory-mapped `FRAME_POINTER`
15	`SP`	24	Stack pointer. Identical to memory-mapped `STACK_POINTER`

Table 4.5: Bank 3 Registers

[a] The source register is in `rMAC1`; the destination register is in `rMAC12`. See individual instructions for an explanation of how the top 8 bits of `rMAC12` are filled.

[b] The source register is in `rMACB1`; the destination register is in `rMACB12`. See individual instructions for an explanation of how the top 8 bits of `rMACB12` are filled.

Note:

Pushing `DivResult` or `DivRemainder` stalls the DSP until any background divide has completed. Popping cancels any current background divide.

All registers are set to 0 on DSP reset.

4.8.1 Base Registers

Base registers, `B0`, `B1`, `B4` and `B5` enable circular buffers to be located at an arbitrary address in memory, for more information see Section 4.11.

Base registers enable the option of easier memory management by removing the constraint on circular buffer placement.

4.9 Stalls

The Kalimba DSP has pre-fetch logic that looks ahead to fetch the next instructions. The pre-fetch stores the next two instructions and on request returns the stored instruction if it is a hit.

For straight line code execution, i.e. no branches, instructions are returned without any stalls. However, if there's a discontinuity in the program flow, such as `jump`, `call` or wrapping of `do...loop`, the core is stalled for 2 clock cycles. The pre-fetch remembers the last discontinuity by storing the branch from and branch to address and the instruction after the branch. If the program flow comes back, it returns the correct instruction without stalling.

As an example, if the Kalimba DSP enters into a `do...loop` that does not have any branches, the code will execute without any stalls until it reaches the end of `do...loop` then a discontinuity in the program flow occurs. The pre-fetch logic prediction of straight line code execution is then invalid as the code is non-sequential, which causes 2 stalls for the program counter to return to the start of the loop. However, the next time round the pre-fetch remembers the jump and no stalls occurs.

4.10 Prefixes

To use a 24-bit constant in a Type B instruction a prefix instruction is required. The assembler kalasm3, adds this automatically, as required. The top 8 bits of the constant are in the prefix instruction with the remaining 16-bits in the main instruction. For example:

```
r0 = r1 + 0x12345;
```

becomes:

```
Prefix (0x1);
r0 = r1 + 0x2345;
```

Without a prefix instruction, the hardware must convert a 16-bit constant to 24 bits to be consistent with the contents of a register. Depending on the associated instruction, there are 3 different ways this conversion can occur. Therefore, whether or not a prefix is needed is different for different instructions. The 3 types of conversion are:

- Logical operations, see Section 6.4, i.e. `AND`, `OR` and `XOR`. The top 8 bits are 0, the constant is placed in the bottom 16 bits.
- Fractional multiplies, see Section 6.8 and Section 6.9, for example:

  ```
  r0 = r1 * 0.5 (frac);  // 0.5 is 0x400000 as a 24-bit fractional
  rMAC = r0 * 0x123400;  // Fractional specified in hex format
  rMAC = rMAC + r0 * 0x123400;
  ```

  In the above examples, the bottom 8 bits are 0, so the constant is in the top 16 bits and no prefix is required.
- Everything else, the constant is sign extended, i.e. the top 8 bits are the same as the top bit of the constant. The constant is in the bottom 16 bits.

  Note:

  Even where the constant is entered as a fractional number this is the conversion method that is used, for example:

  ```
  r0 = r1 + 0.5;
  ```

  translates to:

  ```
  r0 = r1 + 0x400000;
  ```

  This requires a prefix, see Table 4.6.

	Requires Prefix	No Prefix Required
Logical	`0x123456`	`0x003456`
Fractional multiply	`0x123456`	`0x123400`

Everything else	`0x123456`
	`0x009456`
	`0xff7456`
		`0x003456`
		`0x007456`
		`0xff9456`

Table 4.6: Prefix Requirement Examples

4.11 Circular Buffers

A circular or ring buffer is a vector that uses a single, fixed-size buffer as if it were connected end to end. Some hardware assistance is required to do this. This assistance is only available in indexed memory accesses, see Section 6.18, when the appropriate length register is set, see Section 4.7.3. Circular buffers can be of any length less than 64Kwords.

Kalimba Architecture 5 contains new features extending the functionality of circular buffers. It adds a new circular buffer mode called *base register mode*, in which any address can be a valid start address for a circular buffer.

It is possible to use circular buffers with or without base registers. The advantage of using base registers is that circular buffers can be at any location in RAM, whereas there are strict alignment requirements if base registers are not used. The advantage of not using base registers is a reduction in set-up code required before each use.

4.11.1 Circular Buffers Without Use of Base Registers

Use one of the `DMCIRC`, `DM1CIRC` or `DM2CIRC` directives to declare a circular buffer. This ensures that the buffer has a valid start address. Valid start addresses have zeros in all of the locations set in the offset mask (see below). An example of how to use circular buffers is:

```
.VAR/DMCIRC buf[13];   // say buf gets placed at 0x70
I0 = &buf;             // I0 = 0x70
L0 = LENGTH(buf);      // L0 = 0xd
r0 = M[I0,-1];         // Read from address 0x70
                       // (I0 = 0x7c after post-modify and wrap)
```

The following pseudocode explains the logic used to calculate increments of index registers. First, create an offset mask:

```
offset_mask = 2^(ceil(log2(length(buf)))) - 1
```

Then calculate the new value of `I0` according to the following pseudocode where increment in the above example equals `-1`:

```
base = I0 & ~offset_mask
offset = (I0 & offset_mask) + increment
if offset >= 0 && offset > length(buf)
   offset = offset – length(buf)
else if offset < 0
   offset = offset + length(buf)
I0 = base | offset
```

4.11.2 Circular Buffers With Use of Base Registers

Use any one of the standard `DM`, `DM1` or `DM2` directives to declare a circular buffer using base registers. The base register is set to the start address of the buffer. An example of this mode is:

```
.VAR/DM buf[13];        // say buf gets placed at 0x1234
I0 = &buf;              // I0 = 0x1234
push I0;
pop B0;             // B0 = 0x1234
L0 = LENGTH(buf);       // L0 = 0xd
r0 = M[I0,-1];          // Read from address 0x1234
                        // (I0 = 0x1240 after post-modify and wrap)
```

4.12 Zero Overhead Looping: do loop

For zero overhead looping, instructions between `do` and `loop` execute until register `r10` is zero, `r10` is decremented by 1 each loop. This decrement executes just before the last instruction in the loop. Take care using the value of `r10` inside a `do...loop`.

If `r10` is zero at the start of the loop, then no loop instructions execute and a jump to the loop-end label occurs.

If a `do...loop` is executed in an ISR, `r10` and the memory mapped registers `MM_DOLOOP_START` and `MM_DOLOOP_END` must be saved.

Example:

```
r10 = 100;
r0 = 1;
r1 = 1;
do loop;
   r2 = r0 + r1;        // this code will
   r0 = r1 + r2;        // be executed
   r1 = r2 + r0;        // 100 times
loop:
```

Kalimba Architecture 5 contains a 32-word `do…loop` cache to reduce power consumption while executing a `do…loop`. Typically, for many programs, a significant proportion of processing time is inside these loops, and therefore this improvement leads to a considerable reduction in average power consumption.

The cache prevents the core from needing to fetch instructions from program RAM, thus requiring less power. Program RAM is already accessible in a single cycle, so there is no speed advantage gain. If code is being run from flash/ROM, the do loop cache still provides a power-saving benefit over the 1024-word/512-word flash/ROM cache in RAM.

4.13 Debug

The debugging hardware in the Kalimba DSP provides the following features to an external debugger:

- Reset, Run, Stop, Step
- Setting and reading of the `PC`
- 8 program breakpoints
- 2 data memory breakpoints (read, write, or read/write), each covering an address range start to end
- Read/write of register values
- Read/write memory locations as seen by the DSP on any of its 3 memory buses PM, DM1 and DM2
- Read/clear profiling counters:
  - Number of clock cycles
  - Number of instructions executed
  - Number of stall cycles

5 Memory Organisation

The Kalimba DSP core has:

- A single 32-bit program memory, PM, with a 24-bit address space, see Figure 5.1
- 2 24-bit data memory banks, DM1 and DM2, each with a 23-bit address space, see Figure 5.2

Accessing all 3 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.

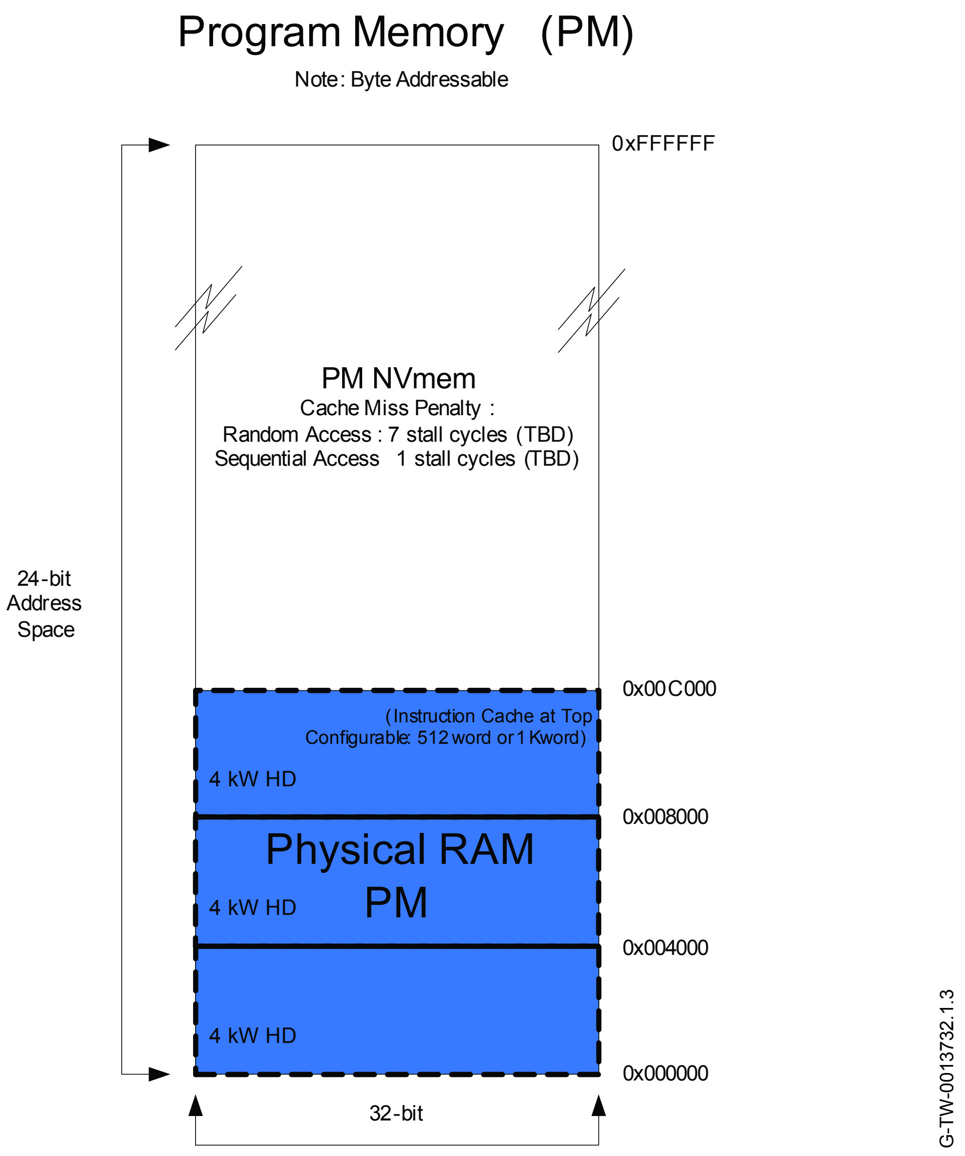


Figure 5.1: Program Memory Organisation for CSR8675

As an example, the CSR8675 BlueCore device has the following physical RAM for the Kalimba DSP:

- DM1 = 32K x 24-bit
- DM2 = 32K x 24-bit
- PM = 12K x 32-bit

The BlueCore MCU initialises the Kalimba DSP. During initialisation, program and data coefficient download to the DSP occurs through auto-incrementing memory-mapped registers in the MCU. The MCU then sets the initial clock frequency for the Kalimba DSP to use before starting it running. An application running on the MCU invokes the program download, and Kalimba DSP initialisation.

Figure 5.2 shows the Kalimba DSP memory organisation with size information and peripheral memory mapping.

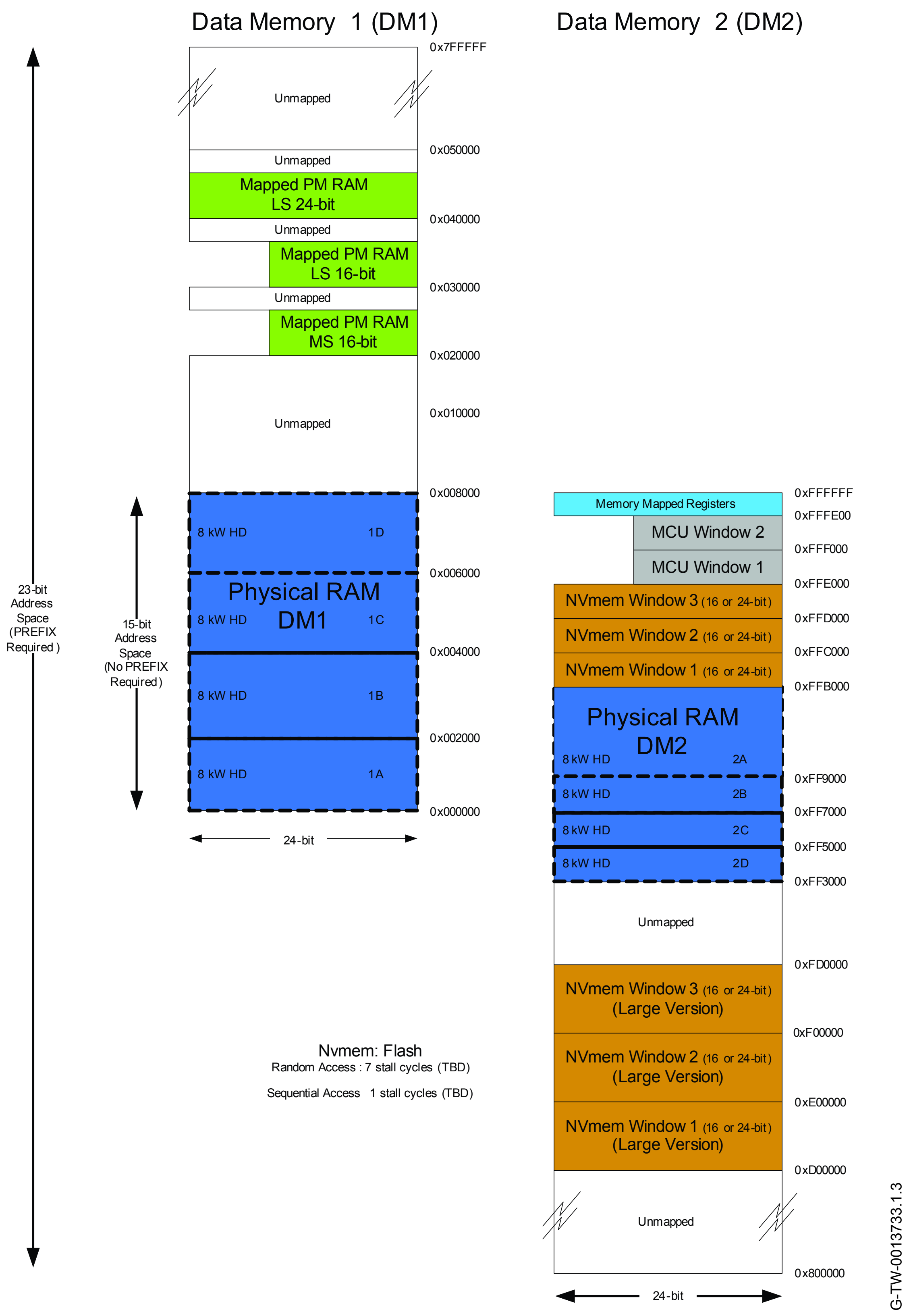


Figure 5.2: Data Memory Organisation for CSR8675

5.1 Memory Map

The memory organisation in Figure 5.1 and Figure 5.2 breaks down into individual memory maps described in Table 5.1, Table 5.2 and Table 5.3. The tables show the DSP memory layout for CSR devices containing Kalimba Architecture 5, using CSR8675 as an example.

5.1.1 PM Memory Map

Table 5.1 shows the program memory map for Kalimba, which contains the physical program memory RAM. Some CSR devices contain a low-power RAM region, which is available as cache for the PM flash/ROM space. The remaining 16Mwords can map into flash/ROM. Memory-mapped registers in the DSP select the flash/ROM start address and size of the region of flash/ROM mapped. The MCU can control the priority given to flash/ROM accesses between the Kalimba DSP and MCU.

Label	Description
Physical RAM PM	General-purpose RAM for program code
Physical RAM PM (instruction cache)	Cache for flash/ROM PM space, or general-purpose RAM if flash/ROM not used. This consists of low-power memory in some CSR devices.
PM NVMem	Mapped to flash/ROM

Table 5.1: PM Memory Map

5.1.2 DM1 Memory Map

Table 5.2 shows the memory map of the first data memory bank DM1, it contains:

- Data memory for the DSP, some of which is low-power data RAM in some CSR devices
- A window of program memory split into the most and least significant halves, and including a 24-bit window
- The remaining areas of the memory map allow for future variants

Label	Description
Physical RAM DM1	General purpose RAM (data memory 1)
Unmapped	Available for RAM expansion in future variants
Mapped PM RAM (MS 16 bits)	Mapped to program memory MS 16 bits (equal in size to Program Memory)
Mapped PM RAM (LS 16 bits)	Mapped to program memory LS 16 bits (equal in size to Program Memory)
Mapped PM RAM (LS 24 bits)	Mapped to program memory LS 24 bits (equal in size to Program Memory)

Table 5.2: DM1 Memory Map

5.1.3 DM2 Memory Map

Table 5.3 shows the memory map of the second data memory bank DM2, it contains:

- General purpose data RAM for the DSP: a portion of this is low-power data RAM in some CSR devices
- 3 1Mwords/4Kwords windows into the flash/ROM: available to the DSP for items such as constant coefficient tables
- 2 MCU windows (4K and 3.5Kwords) into the MCU memory: these enable control information and message passing
- 512 words: these are reserved for the memory mapped I/O for the DSP

Label	Description
Unmapped	Available for RAM expansion in future variants
NVmem Window 1 (16 or 24-bit) (Large version)	Large Non-volatile Memory Window 1 (1Mword)
NVmem Window 2 (16 or 24-bit) (Large version)	Large Non-volatile Memory Window 2 (1Mword)
NVmem Window 3 (16 or 24-bit) (Large version)	Large Non-volatile Memory Window 3 (832Kword)
Physical RAM DM2	General-purpose RAM (data memory 2)
NVmem Window 1 (16 or 24-bit)	Non-volatile Memory Window 1 (4Kwords)
NVmem Window 2 (16 or 24-bit)	Non-volatile Memory Window 2 (4Kwords)
NVmem Window 3 (16 or 24-bit)	Non-volatile Memory Window 3 (4Kwords)
MCU window 1	Window into MCU memory 1 (4Kwords)
MCU window 2	Window into MCU memory 2 (3.5Kwords)
Memory-mapped registers	DSP Memory-mapped I/O registers (512words)

Table 5.3: DM2 Memory Map

6 Instruction Set Description

The instruction set for the Kalimba DSP is suited to an algebraic assembler, rather than the mnemonic assemblers found in traditional MCUs. Algebraic assemblers suit the architecture of a DSP, as they can express complex and parallel instructions in an understandable way. This section describes each instruction in more detail. Table 6.1 lists the notation conventions used in describing the syntax.

Parallel lines | |	Vertical parallel bars enclose lists of syntax options. It is compulsory to select one of the options.
Angled brackets *<non bold italics>*	Anything in *non-bold italics* enclosed by angled brackets is an optional part of the instruction statement.
A, B, C	Denotes a register operand. By default, the register is selected from the list of Bank 1 registers. If subscripted with 'Bank 1/2' then the register can be chosen from either Bank 1 or Bank 2. If subscripted with 'Bank 1/2/3' then the register can be chosen from either Bank 1, Bank 2 or Bank 3.
k_{16}, k_7, k_n	Denotes a constant, the subscripted number being the size of the number in bits.
M[x]	Data in memory location with address 'x'.
M[i,m]	i is the index register, m is the modify register.
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. Section 6.18 describes the valid memory access instructions that can be appended.
rMAC(B)	Represents either rMAC or rMACB registers.

Table 6.1: Instruction Set Descriptions

Important Note:

The Kalimba DSP architecture is designed to execute single-cycle instructions. Any exceptions are noted in Section 6 and in the description for the instruction, e.g. the divide instruction.

6.1 ADD and ADD with CARRY

Syntax:

Type A:	*<if cond>*	C = A + B C = A + M[B] C = M[A] + B M[C] = A + B	*<+Carry>*	*<MEM_ACCESS_1>* ;	

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 `NOP` but *`MEM_ACCESS_1`* is 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 Bank 1 registers, a 16-bit sign extended constant (24-bit with prefix instruction), or memory pointed to by a register or a 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` is assumed. Therefore, the `C` and `V` flags are cleared.

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

When writing to `rFlags` the result is stored in `rFlags` and so the flags above are not applicable.

Additions and subtractions can saturate on overflow by setting the `ARITHMETIC_MODE` memory-mapped register. This causes saturation of signed numbers (positive saturation to `0x7fffff`, negative saturation to `0x800000`).

kalasm3 can understand the following instructions as the `0` can be implemented using the `Null` register.

```
if Z r1 = 0,
 r4 = M[I1, M1];
```

6.2 SUBTRACT and SUBTRACT With Borrow

Syntax:

Type A:	*<if cond>*	C = A - B C = A - M[B] C = M[A] - B M[C] = A - B	*<-Borrow>*	*<MEM_ACCESS_1>* ;	

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

Type B:		$C = A - k_{16}$ $C = A - M[k_{16}]$ $C = M[A] - k_{16}$ $M[k_{16}] = A - C$	*<-Borrow>*		

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

Type C:		C = C - A C = C - M[A]	*<-Borrow>*	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;

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`* is 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 ALU carry bit, `B`. The result is stored in the destination operand. The operands may be either one of the 16 Bank 1 registers, a 16-bit sign extended constant (24-bit with prefix instruction), or memory pointed to by the register or a 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	Cleared if a borrow is generated and set otherwise

Note:

If one of the source operands is `Null` then a negate is assumed. Therefore, the `C` and `V` flags are cleared as appropriate.

When writing to `rFlags` the result is stored in `rFlags` and so the flags above are not applicable.

Additions and subtractions can saturate on overflow by setting the `ARITHMETIC_MODE` memory-mapped register. This causes saturation of signed numbers (positive saturation to `0x7fffff`, negative saturation to `0x800000`).

6.3 Bank 1/2 Register Operations: ADD and SUBTRACT

Syntax:

Type A:	*<if cond>*	$C_{BANK\ 1/2} = A_{BANK\ 1/2} + B_{BANK\ 1/2}$ $C_{BANK\ 1/2} = A_{BANK\ 1/2} - B_{BANK\ 1/2}$	*<MEM_ACCESS_1>* ;	

Example:
```
if Z I0 = I4 + r2,
 r1 = M[I1,M1];
```

Type B:		$C_{BANK\ 1/2} = A_{BANK\ 1/2} + k_{16}$ $C_{BANK\ 1/2} = A_{BANK\ 1/2} - k_{16}$ $C_{BANK\ 1/2} = k_{16} - A_{BANK\ 1/2}$	;

Example:
```
I0 = r2 + 5;
```

Type C:		$C_{BANK\ 1/2} = C_{BANK\ 1/2} + A_{BANK\ 1/2}$ $C_{BANK\ 1/2} = C_{BANK\ 1/2} - A_{BANK\ 1/2}$	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;

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`* is still carried out. Omitting the condition performs the addition or subtraction unconditionally. The operands may be either one of the 16 Bank 1 or 16 Bank 2 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 is generated For subtraction: cleared if a borrow is generated

Note:

If one of the source operands is `Null` then a `LOAD/STORE` is assumed. Therefore, the `C` and `V` flags are cleared.

When writing to `rFlags` the result is stored in `rFlags` and so the flags are not applicable.

Additions and subtractions can saturate on overflow by setting the `ARITHMETIC_MODE` memory-mapped register. This causes saturation of signed numbers (positive saturation to `0x7fffff`, negative saturation to `0x800000`). This only occurs where the destination register is a Bank 1 24-bit register.

6.4 Logical Operations: AND, OR and XOR

Syntax:

Type A:	*<if cond>*	C = A AND B
		C = A OR B
		C = A XOR B
			<MEM_ACCESS_1> ;	

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

Type B:		C = A AND k_{16}
		C = A OR k_{16}
		C = A XOR k_{16}
			;	

Example:
```
r3 = r1 XOR 10;
```

Type C:		C = C AND A
		C = C OR A
		C = C XOR A
			<MEM_ACCESS_1>	*<MEM_ACCESS_2>* ;

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`* is still carried out. Omitting the condition performs the operation unconditionally. The operands may be either one of the 16 Bank 1 registers or a 16-bit, zero padded at the most significant end, 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	Cleared	C	Cleared

CSR

6.5 Shifter: LSHIFT and ASHIFT

Syntax:

Type A:	*<if cond>*	C = A OP B	*<MEM_ACCESS_1>* ;	

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

Type B:		C = A OP k_7
		rMAC(B)0 = A OP k_7
		rMAC(B)12 = A OP k_7
		rMAC(B)2 = A OP k_7
		rMAC(B) = A OP k_7 (LO)
		rMAC(B) = A OP k_7 <(MI)>
		rMAC(B) = A OP k_7 (HI)
		C = k_{16} OP A
			;	

	Example:	`rMAC0 = r2 LSHIFT 4;`		
Type C:		C = C OP A	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;

	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`* is 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 are 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 Bank 1 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 `LSHIFT`: Cleared	C	Cleared

Note:

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

If `rMAC` or `rMACB` is the source operand, the full 56 bits are used as input to the shifter. The output of the shifter is 24-bit in this instruction.

`rFlags` cannot be a destination operand for `ASHIFT` and `LSHIFT`.

For Type B instructions the destination operand can be a Bank 1 register or:

- `rMAC(B)0`, `rMAC(B)12`, or `rMAC(B)2`, causing the other bits of `rMAC(B)` to be unaffected (writing to `rMAC(B)12` writes the data into `rMAC(B)1` and causes sign extension (`ASHIFT`) or zero fill (`LSHIFT`) into `rMAC(B)2`)
- `rMAC(B)`, with a data format tag (`LO`, `MI`, `HI`) to select which word of `rMAC(B)` the 24-bit result from the shifter should be written to, the other bits of `rMAC(B)` are sign-extended / zero-padded as appropriate, see Section 7.13 and Section 7.14. `MI` is the default tag if none is supplied.

The source register can also be a 16-bit sign-extended constant (or 24-bit constant with prefix).

6.6 LSHIFT and ASHIFT (56bit)

Syntax:

Type A:	*<if cond>*	rMAC(B) = A OP B (56bit)	*<MEM_ACCESS_1>* ;	
Type B:		rMAC(B) = A OP k_7 (56bit)	;	
Type C:		rMAC(B) = rMAC(B) OP A (56bit)	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;

Type A Example:

```
if Z rMACB = r2 LSHIFT r1 (56bit),
 r0 = M[I0,M0];
```

Type B Example:

```
rMAC = r2 LSHIFT 4 (56bit);
rMACB = 0x1234 ASHIFT r3 (56bit);
```

Type C Example:

```
rMACB = rMACB ASHIFT r7 (56bit),
 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`* is 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 are 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 $2^{47}-1$ or -2^{47} depending on the sign of the input. No rounding occurs for `ASHIFT` or `LSHIFT`. The operands may be either one of the 16 Bank 1 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 `LSHIFT`: Cleared	C	Cleared

Note:

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

If `rMAC` or `rMACB` is the source operand, the full 56 bits are used as input to the shifter. The output of the shifter is always 56-bit in this instruction.

For Type B instructions, the source register may be a 16-bit sign-extended constant (or 24-bit constant with prefix).

6.7 rMAC(B) Move Operations

Syntax:

Type B:	rMAC(B)0 rMAC(B)2	=	rMAC(B)0 rMAC(B)1 rMAC(B)2 A	;	

Example: `rMAC0 = rMAC1;`

	rMAC(B)12	=	rMAC(B)0 rMAC(B)1 rMAC(B)2 A	(SE) (ZP)	;

Example: `rMAC12 = rMAC0 (SE);`

	C	=	rMAC(B)0 rMAC(B)1	;	

Example: `r3 = rMAC0;`

	C	=	rMAC(B)2	(SE) (ZP)	;

Example: `r3 = rMAC2 (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(B)` registers (`rMAC(B)2`, `rMAC(B)1`, and `rMAC(B)0`), see Figure 4.2. When writing to `rMAC(B)1`, the data is either sign extended or zero padded into `rMAC(B)2`. The format specifiers, `SE` (sign extend) and `ZP` (zero pad), are required to specify how `rMAC(B)2` is filled.

Flags Generated:

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

Note:

These operations are always a Type B instruction, i.e. no parallel memory reads performed.

6.8 Multiply: Signed 24-bit Fractional and Integer

Syntax:

Type A:	*<if cond>* C = A * B	(frac) (int) *<(sat)>*	*<MEM_ACCESS_1>* ;	
	Example: `if Z r3 = r2 * r1 (int) (sat), r5 = M[I0,M2];`			
Type B:	C = A * k_{16}	(frac) (int) *<(sat)>*	;	
	Example: `r6 = r2 * 0.34375 (frac);`			
Type C:	C = C * A	(frac) (int) *<(sat)>*	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;
	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`* is 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 +1.0 and -2^{23} representing -1.0. 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 saturates if overflow occurs. Unbiased rounding is always done for a fractional multiply operation. The operands may be either one of the 16 Bank 1 registers or a 16-bit 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: Cleared int: Set if an arithmetic signed overflow occurs and cleared otherwise	C	Cleared

Note:

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

The 16-bit constant has a different meaning for `(int)` and `(frac)` operations:

- For `(frac)`, the constant is left justified to 24 bits by adding 8 zeros as the LSBs. This enables the 16-bit constant to represent fixed-point fractional numbers between +1.0 and -1.0.
- For `(int)`, the constant is sign extended to 24 bits by adding 8 ones or zeros as the MSBs. This enables the 16-bit constant to represent numbers between +32767 and -32768.

Unbiased rounding of a fractional multiply is carried out according to the following pseudocode:

```
temp = (operand1 * operand2) << 1; // fractional multiply
result = (temp + 0x800000) >> 24;  // round result
if ((temp & 0xFFFFFF) == 0x800000) // unbias above rounding
   result = result & 0xFFFFFE;
```

When the number to be rounded is precisely half way between the 2 nearest values of the result, odd values are rounded away from zero and even values towards zero, yielding a zero large sample bias assuming uniformly distributed values.

6.9 MULTIPLY and ACCUMULATE (56-bit)

Syntax:

Type A:	*<if cond>*	C" = A * B C" = C" + A * B C" = C" - A * B	*<(SS)>* *<(SU)>* *<(US)>* *<(UU)>*	*<MEM_ACCESS_1>* ;	

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

Type B:	C" = A * k_{16} C" = C" + A * k_{16} C" = C" - A * k_{16}	*<(SS)>* *<(SU)>* *<(US)>* *<(UU)>*	;	

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

Type C:	rMAC = C * A
	rMAC = rMAC + C * A
	rMAC = rMAC - C * A
		<(SS)>
		<(SU)>
		<(US)>
		<(UU)>
			<MEM_ACCESS_1>	*<MEM_ACCESS_2>* ;

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`* is 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 Bank 1 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 56th bit and cleared otherwise	C	Cleared

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 enables the 16-bit constant to represent fixed-point fractional numbers between +1.0 and -1.0.

To get the result of the equivalent integer multiplication, shift the result to the right by 1 bit.

For Type A and B instructions not using the supplementary `ADD`/`SUB` operation, the accumulator register C'' may be selected from `rMAC`, `rMACB`, `r0`, `r1`, `r2`, `r3`, `r4` or `r5`, and the data format of the operators may be chosen to be signed (`S`) or unsigned (`U`).

6.10 MULTIPLY and ACCUMULATE (56-bit) with ADD/SUB

Syntax:

Type A:

<if cond>	C'' = A * B C''= C'' + A * B C''= C'' - A * B	*<r0 = D + E>/* *<SS>/<SU>/* *<US>/<UU>*	*<MEM_ACCESS_1>* ;

Example:

```
if NZ r1 = r1 + r7 * r8, r0 = r1 – rMACB,
 r2 = M[I1,M1];

if C rMACB = rMACB + r4 * r5, r0 = r2 + rMACB,
 M[I2,M2] = r1;

if Z r4 = r4 + r1*r2 (SU),
 r5 = M[I0,M0];
```

Type B:

C''= A * k_{16} C''= C'' + A * k_{16} C''= C'' - A * k_{16}	*<r0 = D + E>/* *<SS>/<SU>/* *<US>/<UU>*	;

Example:

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

Type C:

rMAC(B) = C' * A rMAC(B) = rMAC(B) + C' * A rMAC(B) = rMAC(B) – C' * A	r0 = D +/- rMAC(/B)	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;

Example:

```
rMAC = rMAC - r3 * r1, r0 = r1 + rMACB,
 r2 = M[I0,1], r1 = M[I4,-1];
```

Description:

Test the optional condition and, if `TRUE`, perform the specified multiply/accumulate along with the separate add/subtract operation. If the condition is `FALSE`, perform a `NOP`, but *`MEM_ACCESS_1`* is still carried out. Omitting the condition performs the operation unconditionally. The effective binary point is between bits 47 and 46. See Appendix A on number representation for information on multiply operations. See Section 7.19 for tables describing all possible combinations of this 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 overflow occurs past the 56th bit and cleared otherwise	C	Cleared

Note:

To get the result of the equivalent integer multiplication, shift the result to the right by 1 bit.

For the supplementary `ADD`/`SUB` operation, Reg D can be selected from `r1` and `r2`.

For Type A and B instructions using the supplementary `ADD`/`SUB` operation, Reg E can be selected from `rMAC` or `rMACB`.

The accumulator register C'' may be selected from `rMAC`, `rMACB`, `r1` or `r2`, and the data format of both operators is assumed to be signed (`SS`).

For Type C instructions, the second operand of the add/sub is always an `rMAC(B)` register, and is the opposite `rMAC(B)` register to that used for the multiply part. Reg `C'` specifies the first multiplicand, and may select from: `rMACB`, `rMAC`, `r0`, `r1`, `r2`, `r3`, `r4` or `r5`. The data format of both operators is assumed to be signed (`SS`).

Flags are set due to the result of the multiply operation; no flags are altered due to the result of the additional `ADD`/`SUB` operation.

6.11 LOAD/STORE with Memory Offset

Syntax:

Type A:	*<if cond>*	C = M[A + B] M[A + B] = C	*<MEM_ACCESS_1>* ;	

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

Type B:		C = M[A + k_{16}] M[A + k_{16}] = C	;	
	Example:	`M[r3 + 6] = r1;`		
Type C:		C = M[C + A]	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;

	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`* is still carried out. Omitting the condition performs the `LOAD/STORE` unconditionally. The operands may be either one of the 16 Bank 1 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	Cleared	C	Cleared

Note:

As Kalimba has 2 data memory banks, 3 memory accesses will take at least 2 cycles, depending on which banks the addresses are in.

CSR

6.12 Extended LOAD/STORE with Memory Offset

Syntax:

Type A:	*<if cond>*	$C_{1/2}$ = Mxx[$A_{1/2} \pm B_{1/2}$] Myy[$A_{1/2} \pm B_{1/2}$] = $C_{1/2}$	*<MEM_ACCESS_1>* ;

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

Type B:		$C_{1/2}$ = Mxx[$A_{1/2} \pm k_{11}$] Myy[$A_{1/2} \pm k_{11}$] = $C_{1/2}$	;
	Example:	`M[r3 + 6] = r1;`	

Description:

Test the optional condition and, if `TRUE`, perform the specified extended `LOAD/STORE`, including memory offset. If the condition is `FALSE`, perform a `NOP`, but *`MEM_ACCESS_1`* is still carried out. Omitting the condition performs the extended `LOAD/STORE` unconditionally. The operands may be either one of the 16 Bank 1, 16 Bank 2 registers or a 11-bit constant specified in the instruction.

Note:

The Kalimba Architecture 5 DSP does not support sub-word access so:

- Mxx is read a word (32-bit) value
- Myy is write a word (32-bit) value

Flags Generated:

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

Note:

As Kalimba has 2 data memory banks, 3 memory accesses will take at least 2 cycles, depending on which banks the addresses are in.

6.13 Sign Bits Detect and Block Sign Bits Detect

Syntax:

Type A:	*<if cond>*	C = SIGNDET A	*<MEM_ACCESS_1>* ;	
Type C:		C = BLKSIGNDET A	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;

Example:

```
if Z r3 = SIGNDET rMAC,
 r4 = M[I0,M0];
```

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)

If the condition is `FALSE`, perform a `NOP`, but *MEM_ACCESS_1* is still carried out. Omitting the condition performs the operation unconditionally.

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

`BLKSIGNDET` returns the smaller of the result of `SIGNDET` and the present value of the destination register. 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	Cleared	C	Cleared

6.14 Divide Instruction

Syntax:

Type B:	Div = C / A
	C = DivResult
	C = DivRemainder
		;

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

Description:

The `Div = C/A` instruction initiates the divide block's 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. There must be at least 12 cycles between the divide instruction and using the result and/or remainder. If the result or remainder request is before the 12 cycles have elapsed then program flow stalls until the result is ready. To carry out a fractional divide the value in `rMAC/rMAC(B)` first needs to be right-shifted by 1 bit before carrying out the divide operation. The result of the divide can also be pushed directly onto the stack, which stalls the DSP if the divide is not yet completed.

Note:

The value of `DivResult` and `DivRemainder` are available as memory-mapped registers called `MM_QUOTIENT` and `MM_REM` respectively.

`DivResult` returns the result of the division truncated to an integer.

`DivRemainder` returns the remainder such that the equations below hold.

`DivResult` = trunc (`Dividend` / `Divisor`)

`DivRemainder` = `Dividend` – (`Divisor` x `DivResult`)

Example:

Dividend / Divisor	DivResult	DivRemainder
5 / 3	1	2
-5 / 3	-1	-2
5 /-3	-1	2
-5 /-3	1	-2

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

```
r0 = 86420;
rMAC = r0 ASHIFT 0 (LO); // LS word of rMAC now equals 86420, with
                         // sign extension in higher bits;
r0 = 7;
Div = rMAC / r0;
r1 = DivResult;         // r1 = 12345
r2 = DivRemainder;      // r2 = 5
```

Fractional Divide example (0.25/0.75 = 0.3333):

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

Flags Generated:

After: `Div = rMAC(B)/A;`

Z	Unchanged	N	Unchanged
V	Set if a divide exception occurs (divide by zero or overflow in `DivResult`) and cleared otherwise	C	Cleared

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	Cleared	C	Cleared

6.15 Stack Instructions

Syntax:

Type A:	*<if cond>*	push $C_{BANK\ 1/2/3}$	*<MEM_ACCESS_1>* ;
		pop $C_{BANK\ 1/2/3}$	
		$C_{BANK\ 1/2/3}$ = M[FP + A]	
		M[FP + A] = $C_{BANK\ 1/2/3}$	
		C = SP + A	
		SP = SP + A	
		C = FP + A	
		FP = FP + A	

Example:

```
if Z push r0,
 r5 = M[I0,M2];
if NZ pop DivResult;
if Z SP = SP + r5;
M[FP + rMAC] = B0;
```

Type B:	pushm <regList>, SP = SP + k_4	;
	SP = SP – k_4, popm <regList>	
	$C_{BANK\ 1/2/3}$ = M[FP + k_{15}]	
	M[FP + k_{15}] = $C_{BANK\ 1/2/3}$	
	$C_{BANK\ 1/2/3}$ = M[SP +k_{15}]	
	M[SP+k_{15}] = $C_{BANK\ 1/2/3}$	
	C = SP + k_{16}	
	SP = SP + k_{16}	
	C = FP + k_{16}	
	FP = FP + k_{16}	
	push k_{16}	
	push C + k_{16}	

Example:

```
push r0 + 4;
pushm <r0, r7, rMACB>, SP = SP + 13;
SP = SP - 11, popm <I0, I1, I2, I3, I4, I5, M0, L0>;
I0 = M[SP + 0x123];
FP = FP + 0x500;
```

Type C:	push $C_{BANK\ 1/2/3}$	*<MEM_ACCESS_1>*	*<MEM_ACCESS_2>* ;
	pop $C_{BANK\ 1/2/3}$		

Example:

```
push DivRemainder,
 r5 = M[I0,1],
 r1 = M[I4,-1];
```

Description:

Test the optional condition and, if `TRUE`, perform the specified stack operation. If the condition is `FALSE`, perform a `NOP`, but *`MEM_ACCESS_1`* is still carried out. Omitting the condition performs the operation unconditionally. When using `PUSH` or `POP`, any single register from Bank 1, 2 or 3 may be used. When using `PUSH Multiple` or `POP Multiple`, all registers must be in the same bank, i.e. all from Bank 1, 2 or 3.

Note:

`PUSH Multiple` and `POP Multiple` take one cycle per register pushed/popped.

Memory-mapped registers set the stack start address, frame pointer and end address. The memory-mapped stack pointer is incremented with a `push` and decremented with a `pop`, and holds the current stack write address. A stack overflow or underflow causes a `STACK_FAULT` interrupt to occur, with the `STACK_OVERFLOW_PC` memory-mapped register being set to the `PC` where the most recent stack overflow/underflow event occurred.

Flags Generated:

All `PUSH/POP  Multiple` Instructions:

All flags left unchanged.

`SP/FP` Adjust Instructions (e.g. `SP = SP + 5`):

All flags set as per an `ADD` instruction, see Section 6.1.

`PUSH` Instructions:

Flags set depending on the value being pushed.

Note:

V and C can only be set with instructions such as `push r0 + 0x123`

Z	Set if the value equals zero and cleared otherwise	N	Set if the value 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

`POP` Instructions:

Flags set depending on the value being popped.

Z	Set if the value equals zero and cleared otherwise	N	Set if the value is negative and cleared otherwise
V	Cleared	C	Cleared

Note:

A Conditional `pop` with condition `FALSE` carries out the read regardless, but destination register, flags and the memory mapped register `STACK_POINTER` remain unchanged.

`push` of `DivResult` or `DivRemainder` while a divide is busy stalls program flow until the divide completes and then `push` the divide result. A conditional `push` of `DivResult` or `DivRemainder` with condition `FALSE` causes the program flow to stall anyway until the divide completes, but not push the result, set flags or increment stack pointer.

`pop` of `DivResult` or `DivRemainder` while the divider is busy fails to alter the value of `DivResult` or `DivRemainder`, but the stack pointer is decremented and flags are set accordingly.

When popping `DoLoopStart`, `DoLoopEnd` or `r10`, all values must be popped at least 2 instructions before the end of a `do...loop`.

Writing to memory-mapped `M[$STACK_START_ADDR]` sets `M[$STACK_POINTER]` equal to `M[$STACK_START_ADDR]`. Popping to `rMAC(B)12` sign-extends into `rMAC(B)2`.

CSR

6.16 Program Flow: CALL, JUMP, RTS, RTI, and DO...LOOP

Syntax:

Type A:	*<if cond>*	jump A call A rts rti	;

Example: `if Z jump r1;`

Type B:	*<if cond>*	jump k_{16} call k_{16}	;
		do k_{16}	

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

Description:

Omitting the condition performs the program flow. If the condition is `TRUE` perform the program flow described below. Otherwise, execute a `NOP`:

`jump`	Program execution jumps to the address in operand, either a register or a constant, e.g. an address label.
`do`	See Section 4.12
`call`	Loads `rLink` register with return address (`PC+1`) and jumps to address in operand, either a register or a constant, e.g. an address label.
`rti`	Sets `PC` equal to value of the memory-mapped `MM_RINTLINK` register, and restores flags to their pre-interrupt status, i.e. bits 8 to 17 of `rFlags` register copied to bits 0 to 7.
`rts`	Sets `PC` equal to value of `rLink` register. Implement stack depths greater than one in software.

Flags Generated:

Flags `Z`, `N`, `V` and `C` left unchanged.

Note:

Branches using an immediate k_{16} value (no prefix) will be relative, i.e. the k_{16} value is a signed offset from the current `PC` of where to branch to. Branches using a prefix (24-bits) and branches to a register location, e.g. `jump r0`, are interpreted as absolute branches.

6.17 Type A Miscellaneous One and Two Operand Instructions

Syntax:

Type A:	*<if cond>*	C = A + 1
		C = A – 1
		C = ABS A
		C = MIN A
		C = MAX A
		C = ONEBITCOUNT A
		C = TWOBITCOUNT A
		C = MOD24 A
		C = A + 2
		C = A + 4
		C = A - 2
		C = A - 4
		C = SE8 A
		C = SE16 A
			<MEM_ACCESS_1> ;

Example:

```
if Z r0 = MOD3 r1,
 r5 = M[I0,M2];
if NZ rMAC = rMACB - 1;
```

Description:

Test the optional condition and, if `TRUE`, perform the specified operation. If the condition is `FALSE`, perform a `NOP`, but *`MEM_ACCESS_1`* is still carried out. Omitting the condition performs the operation unconditionally. The operands may be any one of the 16 Bank 1 registers.

`INC (C = A + 1)` and `DEC (C = A – 1)` provide conditional increment and decrement instructions.

`ABS` provides a conditional absolute instruction, which multiplies negative numbers by -1.

`MIN`/`MAX` selects the signed smallest/largest operand from Reg C and Reg A, and places the result in Reg C.

`ONEBITCOUNT` accumulates together Reg A assuming it represents 24 1-bit values. The result goes to Reg C. For example, if Reg A = `0x123456`, then after the instruction Reg C = `9`.

`TWOBITCOUNT` accumulates together Reg A assuming it represents 12 2-bit values. The result goes to Reg C. For example, if Reg A = `0x123456`, then after the instruction `RegC = 0+1+0+2+0+3+1+0+1+1+1+2 = 12`.

`MOD24` provides a conditional modulo 24 instruction. Reg A is interpreted as an unsigned number.

`INC (C = A + 2)`, `INC (C = A + 4)`, `INC (C = A - 2)` and `INC (C = A - 4)` provide conditional increment and decrement instructions

`SE8 A`/`SE16 A` provide a conditional instruction that takes the least significant 8-bits/16-bits of Reg A and sign extends to Reg C.

Flags Generated:

For `INC`, `DEC` and `ABS` Instructions:

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

For All Other Instructions:

Z	Set if the value equals zero and cleared otherwise	N	Set if the value is negative and cleared otherwise
V	Cleared	C	Cleared

6.18 Indexed MEM_ACCESS_1 and MEM_ACCESS_2

Syntax:

	MEM_ACCESS_1	MEM_ACCESS_2	
Type A:	Reg_{AG1} = M[I_{AG1}, M_{AG1}] M[I_{AG1}, M_{AG1}] = Reg_{AG1}	;	
Example:	`M[I0,M0] = r1;`		
Type C_{REG}:	Reg_{AG1} = M[I_{AG1}, M_{AG1}] M[I_{AG1}, M_{AG1}] = Reg_{AG1}	Reg_{AG2} = M[I_{AG2}, M_{AG2}] M[I_{AG2}, M_{AG2}] = Reg_{AG2}	;
Type C_{CONST}:	Reg_{AG1} = M[I_{AG1}, MK_{AG1}] M[I_{AG1}, MK_{AG1}] = Reg_{AG1}	Reg_{AG2} = M[I_{AG2}, MK_{AG2}] M[I_{AG2}, MK_{AG2}] = Reg_{AG2}	;

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

Permitted Registers

Reg_{AG1} / Reg_{AG2}	`r0`, `r1`, `r2`, `r3`, `r4`, `r5` and `rMAC`	I_{AG1}	`I0`, `I1`, `I2` and `I3`
	`M0`, `M1`, `M2` and `M3`	I_{AG2}	`I4`, `I5`, `I6` and `I7`
M_{AG1} / M_{AG2}	`-1`, `0`, `1` and `2`		
MK_{AG1} / MK_{AG2}			

Description:

Any Type C instruction 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.

Flags Generated:

No flags are affected by the memory access part of instructions.

Note:

`RegAG1/AG2` selects one of the first 8 Bank 1 registers, i.e. `Null`, `rMAC`, `r0` to `r5`. If the `Null` register is selected, then no memory read/write is performed.

Type A instructions use `AG1` so only index registers `I0` to `I3` may be used. They must use a modify register rather than a modify constant.

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

The index register post-modify takes place after the instruction has executed. The modify register or constant does not affect the address that is used in the memory access, or the result of any calculation in the main instruction involving the index register.

Modulo indexing, controlled by the length and base registers, is applied during the post-modify part of an indexed memory access.

7 Instruction Coding

Instruction coding is relatively simple and orthogonal so that the instruction decode is efficient. There are 3 basic coding formats: Type A, B and C. Figure 7.1 outlines the format of the instruction with corresponding definitions described in Section 7.1 to Section 7.20.

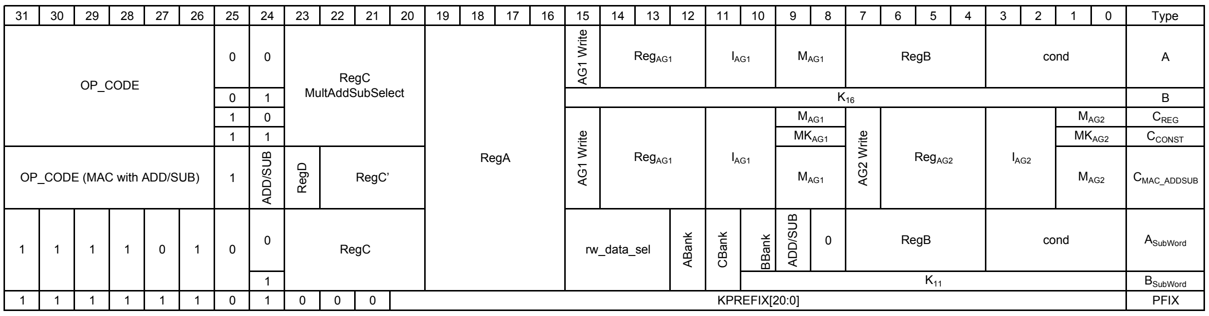


Figure 7.1: Instruction Coding Format

Note:

Type	Instruction type, see Section 7.1, Section 7.2 and Section 7.3
`OP_CODE`	Selects the instruction operation, see Section 7.5
$\texttt{RegA}_{\texttt{B1/2}}$	Selects a register for the 1st source operand for instructions. Bank 1 registers are default, Bank 2 is selected in `OP_CODE` for certain instructions indicated by '$_{B1/2}$'
$\texttt{RegB}_{\texttt{B1/2}}$	Selects a register for the 2nd source operand for instructions. Bank 1 registers are default, Bank 2 is selected in `OP_CODE` for certain instructions indicated by '$_{B1/2}$'
$\texttt{RegC}_{\texttt{B1/2/3}}$	Selects 1 of 16 registers for the destination register for instructions. For Type C instructions `RegC` also defines the first source operand, see Table 7.1. Bank 1 registers are used by default, Bank 2 or 3 is selected in `OP_CODE` or `StackBankSelect` for certain instructions indicated by '$\texttt{B}_{\texttt{1/2/3}}$'
`cond`	Selects an optional condition to be met for the instruction to be executed, otherwise a no-operation will be executed, see Section 4.5.10
$\texttt{k}_{\texttt{16}}$ / $\texttt{k}_{\texttt{PREFIX}}$	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`)
$\texttt{Reg}_{\texttt{AG1}}$ / $\texttt{Reg}_{\texttt{AG2}}$	Selects one of the first 8 Bank 1 registers: `rMAC`, `r0` to `r5`; for the source/destination register of multifunction memory reads and writes. If `Null` is selected then no read or write is performed.
$\texttt{I}_{\texttt{AG1}}$ / $\texttt{I}_{\texttt{AG2}}$	Selects one of the index registers, `I0` to `I3` for `AG1`, and `I4` to `I7` for `AG2`, for multifunction memory reads and writes
$\texttt{M}_{\texttt{AG1}}$ / $\texttt{M}_{\texttt{AG2}}$	Selects modify register, `M0` to `M3`, for multifunction memory reads and writes
$\texttt{MK}_{\texttt{AG1}}$ / $\texttt{MK}_{\texttt{AG2}}$	Selects a fixed constant to use for the modify: `-1`, `0`, `+1`, or `+2`
`StackBankSelect`	Selects the bank of registers to use for stack operations

Note:

Throughout, `OP` refers to operation which can include addition (`+`), subtraction (`-`), multiplication (`*`), `AND`, `OR` and exclusive OR (`XOR`).

7.1 Type A Instruction

Type A is a conditional instruction, with an additional single memory read or write operation. The instruction can accept 2 input operands and 1 output operand (which can be different). Operands can be any of the 16 Bank 1 registers (`LOAD/STORE` instructions permit the usage of Bank 2 registers and `PUSH/POP` Instructions permit the use of Bank 1, 2 and special Bank 3 registers). For the memory access, index registers `I0` to `I3` select the address, the destination or source register can be any of the first 8 Bank 1 registers except `Null`: `rMAC`, `r0` to `r5`. At the end of the instruction the index register is post modified by one of the modify registers `M0` to `M3`.

A Type A instruction has the following syntax:

```
<if cond> RegC = RegA OP 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, see Section 4.10. No additional memory access operation is permitted. The `PFIX` instruction is automatically added if needed by the assembler **kalasm3**.

A Type B instruction has the following syntax:

```
RegC = RegA OP 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, 2 memory accesses (reads, writes or a combination) can occur in the same clock cycle; one memory access uses `AG1` (index registers `I0` to `I3`) and the other uses `AG2` (index registers `I4` to `I7`). The index registers can be either post modified by the `M0` to `M3` registers (this is a Type C_{REG} instruction) or by a 2-bit signed modify constant (valid values: `-1`, `0`, `1`, or `2`) (this is a Type C_{CONST} instruction).

A Type C instruction has the following syntax:

```
RegC = RegC OP 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, because they can always be conditional.

7.5 OP_CODE Coding

Table 7.1 describes the instruction decoding of the `OP_CODE` field in Figure 7.1.

OP_CODE				Action (Type A)[a]	Action (Type B)	Action (Type C$_{REG/CONST}$)[b]	Description
000	AM		C	RegC = RegA + RegB	RegC = RegA + k$_{16}$	RegC = RegC + RegA	Add[c]
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$_{B1/2}$	Bank 1/2 Add
011	B2RS			RegC$_{B1/2}$ = RegA$_{B1/2}$ - RegB$_{B1/2}$	RegC$_{B1/2}$ = RegA$_{B1/2}$ - k$_{16}$ RegC$_{B1/2}$ = k$_{16}$ - RegA$_{B1/2}$	RegC$_{B1/2}$ = RegC$_{B1/2}$ - RegA$_{B1/2}$	Bank 1/2 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$_{16}$	RegC = RegC LSHIFT RegA	Logical Shift
100	1	0	0	RegC = RegA ASHIFT RegB	RegC = RegA ASHIFT k$_{16}$	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(B) = rMAC(B) + RegA * RegB [r0 = RegD ± rMAC(/B)] [a]	rMAC(B) = rMAC(B) + RegA * k$_{16}$ [r0 = RegD ± rMAC(/B)]	rMAC = rMAC + RegC * RegA	Multiply accumulate (56-bit) (with optional ADD/SUB)

OP_CODE				Action (Type A)[a]	Action (Type B)	Action (Type $C_{REG/CONST}$)[b]	Description
101	1	S	S	`rMAC(B) = rMAC(B) - RegA * RegB` `[r0 = RegD ± rMAC(/B)]` [a]	`rMAC(B) = rMAC(B) – RegA *` k_{16} `[r0 = RegD ± rMAC(/B)]`	`rMAC = rMAC – RegC * RegA`	Multiply subtract (56-bit) (with optional ADD/SUB)
110	0	S	S	`rMAC(B) = RegA * RegB` `[r0 = RegD ± rMAC(/B)]` [a]	`rMAC(B) = RegA *` k_{16} `[r0 = RegD ± rMAC(/B)]`	`rMAC = RegC * RegA`	Multiply (48-bit) (with optional ADD/SUB)
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`	`rMAC = rMAC + RegC' * RegA`
						`r0 = RegD ± rMACB`
							Store with offset
							/ Mult-acc (56-bit) with ADD/SUB

110	1	1	0	`RegC = SignDet RegA`	`Div = rMAC / RegA` [d] `RegC = DivResult` [d] `RegC = DivRemainder` [d]	`RegC = BlkSignDet RegA`	Sign detect / Divide / Block sign detect

110	1	1	1	`JUMP RegA`
				(RTS *if RegA=rLink*)
				(RTI *if RegA=rFlags*)
					`if [RegC=cond] JUMP` k_{16}	`rMAC = rMAC – RegC' * RegA`
						`r0 = RegD ± rMACB`
							Jump to program address / return from subroutine / return from interrupt /
							Mult-sub (56-bit) with ADD/SUB

111	0	0	0	`CALL RegA`	`if [RegC=cond] CALL` k_{16}	`rMAC = RegC' * RegA`
						`r0 = RegD ± rMACB`
							Call subroutine / return from interrupt /
							Multiply (56-bit) with ADD/SUB

111	0	0	1	`RegC = MIN/MAX/ABS...RegA`	`DO …. LOOP`	`rMACB = rMACB + RegC' * RegA`
						`r0 = RegD ± rMAC`
							Misc 1 and 2 Operand instructions /
							Do Loop /
							Mult-acc (56-bit) with ADD/SUB

OP_CODE				Action (Type A)[a]	Action (Type B)	Action (Type $C_{REG/CONST}$)[b]	Description
111	0	1	0	Future use	RegC = k_{16} LSHIFT RegA	rMACB = rMACB – RegC' * RegA r0 = RegD ± rMAC	Logical Shift of a constant / Mult-sub (56-bit) with ADD/SUB
111	0	1	1	Future use	RegC = k_{16} ASHIFT RegA	rMACB = RegC' * RegA r0 = RegD ± rMAC	Arithmetic Shift of a constant / Multiply (56-bit) with ADD/SUB
111	1	0	0	PUSH $RegC_{B1/2/3}$ POP $RegC_{B1/2/3}$ LOAD/STORE to stack with offset Stack/frame pointer adjust	PUSH Multiple POP Multiple LOAD/STORE to stack with offset Stack/frame pointer adjust	PUSH $RegC_{B1/2/3}$ POP $RegC_{B1/2/3}$	New stack operations
111	1	0	1	$RegC_{B1/2}$ = Mxx[$RegA_{B1/2}$ ± RegB] Myy[$RegA_{B1/2}$ ± RegB] = $RegC_{B1/2}$	$RegC_{B1/2}$ = Mxx[$RegA_{B1/2}$ ± k_{11}] Myy[$RegA_{B1/2}$ ± k_{11}] = $RegC_{B1/2}$	Not available (used by type-A/B)	Subword memory read/write instructions
111	1	1	0	Future use			
111	1	1	1		PREFIX instruction		Enables 24-bit constants for Type B operations – by prefixing the following instruction's k_{16} by the 8-bit k_{PREFIX} value.

Table 7.1: OPCODE Coding Format

[a] Type A is conditional with if [cond] and can contain single memory access MEM_ACCESS_1

[b] Type C permits 2 simultaneous memory accesses MEM_ACCESS_1 and MEM_ACCESS_2

[c] The add instruction is also used to implement LOAD/STORE to registers/memory by setting one of the source registers as Null. Setting all 3 operands as Null implements a no-operation (NOP) instruction

[d] See Section 7.11 and Section 7.16 for encoding of k_{16} for rMAC shift instructions and Divide instructions

7.6 AM Field

The AM Field selects different memory addressing modes, see Table 7.2.

AM	Type A	Type B	Type C$_{REG/CONST}$
00	RegC = RegA [OP] RegB	RegC = RegA [OP] k_{16}	RegC = RegC [OP] RegA
01	RegC = RegA [OP] M[RegB]	RegC = RegA [OP] M[k_{16}]	RegC = RegC [OP] M[RegA]
10	RegC = M[RegA] [OP] RegB	RegC = M[RegA] [OP] k_{16}	-
11	M[RegC] = RegA [OP] RegB	M[k_{16}] = RegC [OP] RegA	-

Table 7.2: AM Field

7.7 Carry Field (C Field)

Selects whether addition/subtraction is performed with carry and the appropriate state, see Table 7.3.

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

Table 7.3: C Field Options

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

Bank 1/2 register select for add/subtract instructions, see Table 7.4. Subtract from constant Type B instructions are only valid for `SUBTRACT` opcodes.

B2RS	Type A	Type B	Type $C_{REG/CONST}$
000	$RegC_{Bank1} = RegA_{Bank1} \pm RegB_{Bank1}$	$RegC_{Bank1} = RegA_{Bank1} \pm K_{16}$	$RegC_{Bank1} = RegC_{Bank1} \pm RegA_{Bank1}$
001	$RegC_{Bank1} = RegA_{Bank1} \pm RegB_{Bank2}$	$RegC_{Bank1} = K_{16} - RegA_{Bank1}$	$RegC_{Bank1} = RegC_{Bank1} \pm RegA_{Bank2}$
010	$RegC_{Bank1} = RegA_{Bank2} \pm RegB_{Bank1}$	$RegC_{Bank1} = RegA_{Bank2} \pm K_{16}$	-
011	$RegC_{Bank1} = RegA_{Bank2} \pm RegB_{Bank2}$	$RegC_{Bank1} = K_{16} - RegA_{Bank2}$	-
100	$RegC_{Bank2} = RegA_{Bank1} \pm RegB_{Bank1}$	$RegC_{Bank2} = RegA_{Bank1} \pm K_{16}$	-
101	$RegC_{Bank2} = RegA_{Bank1} \pm RegB_{Bank2}$	$RegC_{Bank2} = K_{16} - RegA_{Bank1}$	-
110	$RegC_{Bank2} = RegA_{Bank2} \pm RegB_{Bank1}$	$RegC_{Bank2} = RegA_{Bank2} \pm K_{16}$	$RegC_{Bank2} = RegC_{Bank2} \pm RegA_{Bank1}$
111	$RegC_{Bank2} = RegA_{Bank2} \pm RegB_{Bank2}$	$RegC_{Bank2} = K_{16} - RegA_{Bank2}$	$RegC_{Bank2} = RegC_{Bank2} \pm RegA_{Bank2}$

Table 7.4: B2RS Field

7.9 Saturation Select Field (V Field)

Selects whether to enable saturation on the result, see Table 7.5.

Note:

The addition and subtraction instructions can also saturate on overflow by setting the `ARITHMETIC_MODE` memory mapped register bit.

V	Description
0	No saturation
1	Saturation

Table 7.5: V Field

7.10 Sign Select Field (S Field)

Selects signed/unsigned multiplies, see Table 7.6.

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

Table 7.6: S Field

7.11 k_{16} Coding for LSHIFT and ASHIFT

The k_{16} coding section from the instruction coding format in Figure 7.1 splits into its individual bits and Table 7.7 descibes their functionality.

Bit															
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 7.7: k_{16} Coding Shift Format

Note:

`ShiftAmount`: is the amount to shift the input, it is a signed 7-bit number. 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 as described in Table 7.8. See Section 7.13 and Section 7.14 for more information on how to load the individual sub registers.

Bits	[55:48]	[47:24]	[23:0]
rMAC Register	rMAC2	rMAC1	rMAC0

Table 7.8: rMAC Sub-registers

7.13 ASHIFT

Table 7.9 shows how the arithmetic shift instruction is encoded within the instruction-coding format described in Figure 7.1, for more information on the `ASHIFT` 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} `(LO)`
`000 (rFlags)`	Sign extend	`SHIFTER_OUTPUT`	Trailing zeros	`rMAC = r? ASHIFT` k_{16} `(MI)`
`000 (rMAC(B))`	`SHIFTER_OUTPUT`			`rMAC(B) = r? ASHIFT` k_{16} `(56bit)`
`010`	`SHIFTER_OUTPUT`	Trailing zeros	Trailing zeros	`rMAC = r? ASHIFT` k_{16} `(HI)`
`101`	Old `rMAC2`	Old `rMAC1`	`SHIFTER_OUTPUT`	`rMAC0 = r? ASHIFT` k_{16}
`100`	Sign extend	`SHIFTER_OUTPUT`	Old `rMAC0`	`rMAC12 = r? ASHIFT` k_{16}
`110`	`SHIFTER_OUTPUT`	Old `rMAC1`	Old `rMAC0`	`rMAC2 = r? ASHIFT` k_{16}

Table 7.9: ASHIFT

7.14 LSHIFT

Table 7.10 shows how the logical shift instruction is encoded within the instruction-coding format described in Table 7.10, for more information on the LSHIFT instruction see Section 6.5.

Dest_Sel (RegC)	New rMAC2	New rMAC1	New rMAC0	Example
001	Zero fill	Zero fill	SHIFTER_OUTPUT	rMAC(B) = r? LSHIFT k_{16} (LO)
000 (rFlags)	Zero fill	SHIFTER_OUTPUT	Trailing zeros	rMAC(B) = r? LSHIFT k_{16} (MI)
000 (rMAC(B))	SHIFTER_OUTPUT			rMAC(B) = r? LSHIFT k_{16} (56bit)
010	SHIFTER_OUTPUT	Trailing zeros	Trailing zeros	rMAC(B) = r? LSHIFT k_{16} (HI)
101	Old rMAC2	Old rMAC1	SHIFTER_OUTPUT	rMAC(B)0 = r? LSHIFT k_{16}
100	Zero fill	SHIFTER_OUTPUT	Old rMAC0	rMAC(B)12 = r? LSHIFT k_{16}
110	SHIFTER_OUTPUT	Old rMAC1	Old rMAC0	rMAC(B)2 = r? LSHIFT k_{16}

Table 7.10: LSHIFT

Note:

Both the LSHIFT and ASHIFT instructions enable the user to write to the 3 individual sub-registers rMAC2/1/0, providing a way of loading rMAC with a double precision number. It also enables shift operations of rMAC with the destination being the rMAC register, which speeds up double precision calculations. For more information about rMAC move operations see Section 6.7.

7.15 Type A and Type C LSHIFT56 and ASHIFT56

RegC	Shift Destination Register	Data Width of Shift Output (bit)
rFlags	rMAC	24
rMAC	rMAC	56
rMACB	rMACB	56

7.16 k_{16} Coding Divide Instructions

Table 7.11 describes the k_{16} coding divide instruction and shows how the background divide instruction is encoded within the instruction-coding format described in Figure 7.1.

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

Table 7.11: Divide Field

Table 7.12 describes the functionality of the individual divide bits.

DIV	Assembly Syntax	Operation
00	Div = rMAC / RegA	Initiates a 48-bit/24-bit divide using rMAC as the dividend and RegA as the divisor.
01	RegC = DivResult	There must be at least 12 cycles between the divide instruction and using the result and/or remainder. In this period, normal program execution continues. If the result request is early, then program execution automatically delays until the result is available.
10	RegC = DivRemainder	
11	Div = rMACB / RegA	Initiates a 48-bit/24-bit divide using rMACB as the dividend and RegA as the divisor.

Table 7.12: Divide Field States

7.17 Type A Miscellaneous One and Two Operand Instruction Encodings

RegA	RegC	RegB	Operation
RegA	RegC	0 0 1 0	RegC = RegA + 1
RegA	RegC	0 0 1 1	RegC = RegA – 1
RegA	RegC	0 1 0 0	RegC = ABS RegA
RegA	RegC	0 1 0 1	RegC = MIN RegA
RegA	RegC	0 1 1 0	RegC = MAX RegA
RegA	RegC	0 1 1 1	RegC = TWOBITCOUNT RegA
RegA	RegC	1 0 0 0	RegC = MOD24 RegA
RegA	RegC	1 0 0 1	RegC = ONEBITCOUNT RegA
RegA	RegC	1 0 1 0	RegC = RegA + 2
RegA	RegC	1 0 1 1	RegC = RegA + 4
RegA	RegC	1 1 0 0	RegC = RegA - 2
RegA	RegC	1 1 0 1	RegC = RegA - 4
RegA	RegC	1 1 1 0	RegC = SE8 RegA
RegA	RegC	1 1 1 1	RegC = SE16 RegA

Table 7.13: Encodings for One and Two Operand Instructions

7.18 Stack Encoding

7.18.1 Type A Stack Encoding

Instruction	RegC	RegA				RegB
`push rStack;`	**`RegSelect`**	0	0	**`BankSelect`**		*Unused (set to zero)*
`pop rStack;`		0	1			
`rStack = M[FP + RegOffset];`		1	0			**`RegOffset`**
`M[FP + RegOffset] = rStack;`		1	1			

`RegC = SP + RegOffset;`
`OR, if (RegC==Null)`
`SP = SP + RegOffset;`
	`RegC`	0	0	1	1	

`RegC = FP + RegOffset;`
`OR, if (RegC==Null)`
`FP = FP + RegOffset;`
		0	1	1	1	

Table 7.14: Type A Stack Operation Encoding

Note:

`rStack` is the register selected by `RegSelect` in the bank selected by `BankSelect`:

- 0 = Bank 1
- 1 = Bank 2
- 2 = Bank 3

`FP` is the frame pointer

`SP` is the stack pointer

7.18.2 Type B Stack Encoding

Instruction	RegC	RegA				K_{16}	

`pushm <regList>,`
` SP = SP + StackAdjust;`
	`StackAdjust`
	(unsigned)
		0	0	**`BankSelect`**		**`RegSelectBitfield`**	

`SP = SP - StackAdjust,`
` popm <reglist>;`
	`StackAdjust`
	(unsigned)
		0	1	**`BankSelect`**		**`RegSelectBitfield`**	

`rStack = M[FP + Offset];`	**`RegSelect`**	1	0	**`BankSelect`**		0	Offset (15-bit signed)
`rStack = M[SP + Offset];`	**`RegSelect`**	1	1	**`BankSelect`**		1	Offset (15-bit signed)
`M[FP + Offset] = rStack;`	**`RegSelect`**	1	1	**`BankSelect`**		0	Offset (15-bit signed)
`M[SP + Offset] = rStack;`	**`RegSelect`**	1	1	**`BankSelect`**		1	Offset (15-bit signed)

`RegC = SP + Offset;`
`OR, if (RegC==Null)`
`SP = SP + Offset;`
	`RegC`	0	0	1	1	Offset
						(signed)	

`RegC = FP + Offset;`
`OR, if (RegC==Null)`
`FP = FP + Offset;`
	`RegC`	0	1	1	1	Offset
						(signed)	

`push RegC + k16`	**`RegC`** (Bank1 only)	1	1	1	1	Value (signed)	

Table 7.15: Type B Stack Operation Encoding

Note:

`rStack` is the register selected by `RegSelect` in the bank selected by `BankSelect`:

- 0 = Bank 1
- 1 = Bank 2
- 2 = Bank 3

`FP` is the frame pointer

`SP` is the stack pointer

`StackAdjust` is a 4-bit unsigned value of how much to adjust `SP` by after a `pushm` (increment) or before a `popm` (decrement).

`RegSelectBitfield` has a bit per register, if a bit is set then that register (in the bank selected by `BankSelect`) is pushed/popped. This provides a push/pop multiple encoding, see Table 7.16.

Bit	BankSelect = 0 (Bank 1)	BankSelect = 1 (Bank 2)	BankSelect = 2 (Bank 3)
	Register	Register	Register
0	FP(=SP) (special case: after pushing, FP is set equal to the value of SP before the pushm instruction)	I0	rMAC2
1	rMAC (24-bit)	I1	rMAC12
2	r0	I2	rMAC0
3	r1	I3	DoLoopStart
4	r2	I4	DoLoopEnd
5	r3	I5	DivResult
6	r4	I6	DivRemainder
7	r5	I7	rMACB2
8	r6	M0	rMACB12
9	r7	M1	rMACB0

Bit	BankSelect = 0 (Bank 1)	BankSelect = 1 (Bank 2)	BankSelect = 2 (Bank 3)
	Register	Register	Register
10	r8	M2	B0
11	r9	M3	B1
12	r10	L0	B4
13	rLink	L1	B5
14	rFlags	L4	FP
15	rMACB	L5	SP

Table 7.16: BankSelect = 0 (Bank 1), or BankSelect = 1 (Bank 2)

7.18.3 Type C Stack Encoding

Instruction	RegC	RegA		
push rStack;	RegSelect	0	0	BankSelect
pop rStack;		0	1	

Table 7.17: Type C Stack Operation Encoding

Note:

`rStack` is the register selected by `RegSelect` in the bank selected by `BankSelect`:

- `0` = Bank 1
- `1` = Bank 2
- `2` = Bank 3

7.19 Type C MAC with ADD/SUB Instruction

Instruction	OP_CODE	25	24	23	22	21	20	19	18	17	16	15:0	Type
rMAC = rMAC + RegC' * RegA, r0 = RegD + rMACB	1 1 0 1 0 1	1	0	RegD	RegC'			RegA				2 indexed memory accesses (These are of type C_{REG} only, i.e. with modify registers rather than constants)	C_{MAC_ADDSUB}
rMAC = rMAC + RegC' * RegA, r0 = RegD - rMACB			1										
rMAC = rMAC - RegC' * RegA, r0 = RegD + rMACB	1 1 0 1 1 1	1	0										
rMAC = rMAC - RegC' * RegA, r0 = RegD - rMACB			1										
rMAC = RegC' * RegA, r0 = RegD + rMACB	1 1 1 0 0 0	1	0										
rMAC = RegC' * RegA, r0 = RegD - rMACB			1										
rMACB = rMACB + RegC' * RegA, r0 = RegD + rMAC	1 1 1 0 0 1	1	0										
rMACB = rMACB + RegC' * RegA, r0 = RegD - rMAC			1										
rMACB = rMACB - RegC' * RegA, r0 = RegD + rMAC	1 1 1 0 1 0	1	0										
rMACB = rMACB - RegC' * RegA, r0 = RegD - rMAC			1										

Instruction	OP_CODE	25	24	23	22	21	20	19	18	17	16	15..0	Type
`rMACB = RegC' * RegA, r0 = RegD + rMAC`	1 1 1 0 1 1	1	0	RegD	RegC'			RegA				2 indexed memory accesses (These are of type C_{REG} only, i.e. with modify registers rather than constants)	C_{MAC_ADDSUB}
`rMACB = RegC' * RegA, r0 = RegD - rMAC`	1 1 1 0 1 1	1	1	RegD	RegC'			RegA				2 indexed memory accesses (These are of type C_{REG} only, i.e. with modify registers rather than constants)	C_{MAC_ADDSUB}

Table 7.18: BlueCore7/BlueCore8 Type C MAC with ADD/SUB Encoding

Note:

`RegC'` specifies the first multiplicand. This is a 3-bit value and that selects: `rMACB` (special case when `RegC' = 0`), `rMAC`, `r0`, `r1`, `r2`, `r3`, `r4`, or `r5`.

`RegA` specifies the second multiplicand. This is a standard 4-bit Bank 1 register select value.

`RegD` specifies the first operand of the add/sub. It is a 1-bit value and selects either `r1` (0) or r2 (1).

The second operand of the add/sub is always an `rMAC(B)` register, and is the opposite `rMAC(B)` register to that used for the multiply part.

7.20 Type A and B MAC and Optional ADD/SUB Instruction

Instruction	OP_CODE			RegC			RegA	RegB/k_{16}
`RegC' = RegC' + RegA * RegB;`	1 0 1 0	S	S	0	RegC'		RegA 1st multiplicand	RegB/k_{16} 2nd multiplicand
`RegC'' = RegC'' + RegA * RegB` `r0 = RegD + RegE`		RegD	0	1	RegE	RegC''		
`RegC'' = RegC'' + RegA * RegB` `r0 = RegD - RegE`			1	1				

Table 7.19: BlueCore7/BlueCore8 Type A and B MAC and Optional ADD/SUB Encoding

Note:

`RegC'` specifies the accumulator register (for when there is no supplementary `ADD`/`SUB` operation). This is a 3-bit value that selects: `rMACB` (special case when `RegC' = 0`), `rMAC`, `r0`, `r1`, `r2`, `r3`, `r4`, or `r5`.

`RegC''` specifies the accumulator register (for when there is a supplementary `ADD`/`SUB` operation). This is a 2-bit value that selects: `rMACB` (special case when `RegC' = 0`), `rMAC`, `r1` or `r2`.

`RegD` specifies the first operand of the `ADD`/`SUB` operation. This is a 1-bit value and selects either `r1` (0) or `r2` (1).

`RegE` specifies the second operand of the `ADD`/`SUB` operation. It is a 1-bit value and selects either `rMAC` (0) or `rMACB` (1).

8 Kalimba DSP Peripherals

The Kalimba DSP for BlueCore consists of the DSP core, the DSP peripherals and their associated interfaces. This section describes the peripherals and interfaces.

See Appendix B for information relating to registers.

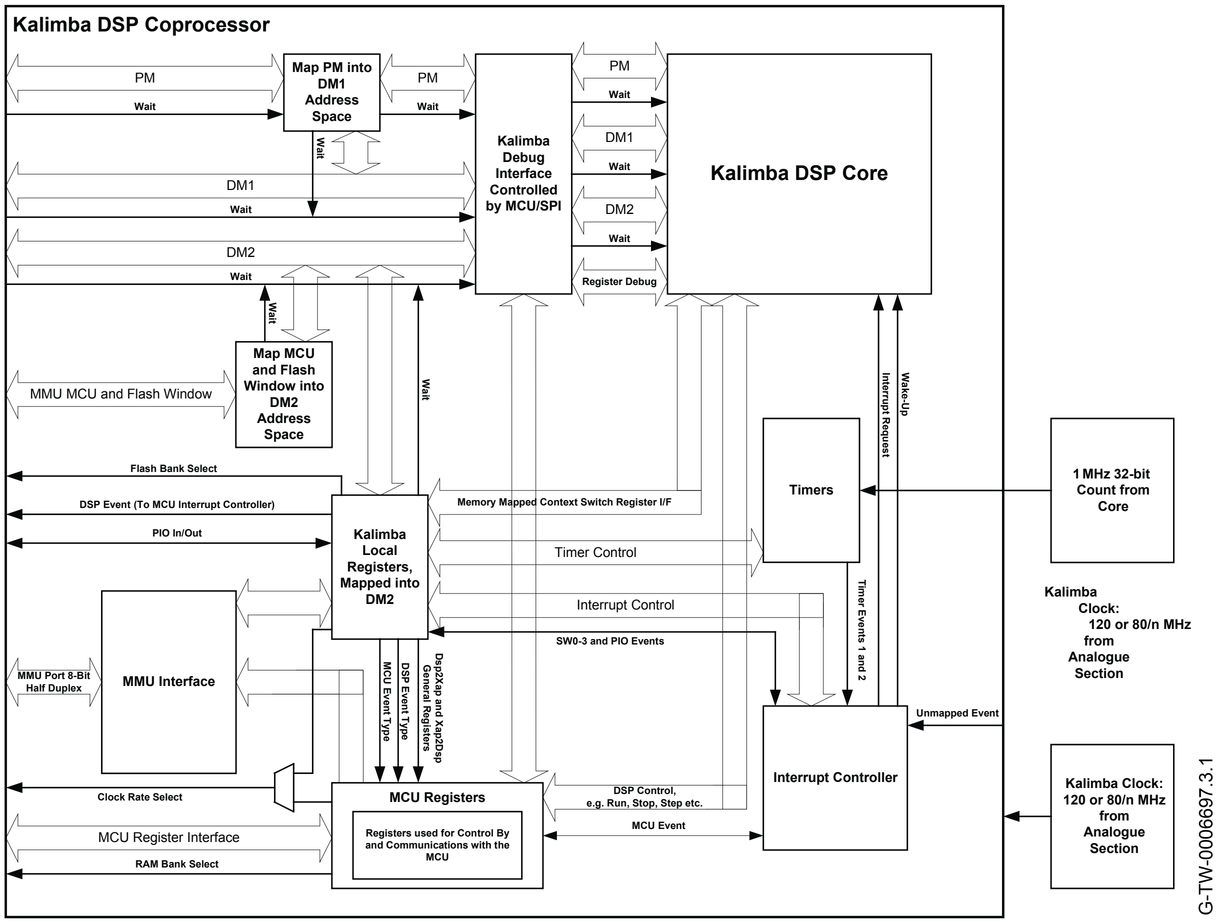

Kalimba DSP Coprocessor
PM
Wait
Map PM into DM1 Address Space
PM
Wait
Wait
DM1
Wait
DM2
Wait
Wait
Kalimba Debug Interface Controlled by MCU/SPI
PM
Wait
DM1
Wait
DM2
Wait
Register Debug
Kalimba DSP Core
Map MCU and Flash Window into DM2 Address Space
MMU MCU and Flash Window
Wait
Interrupt Request
Wake-Up
Timers
1 MHz 32-bit Count from Core
Flash Bank Select
DSP Event (To MCU Interrupt Controller)
PIO In/Out
Kalimba Local Registers, Mapped into DM2
Memory Mapped Context Switch Register I/F
Timer Control
Interrupt Control
SW0-3 and PIO Events
Timer Events 1 and 2
Kalimba Clock: 120 or 80/n MHz from Analogue Section
MMU Port 8-Bit Half Duplex
MMU Interface
MCU Event Type
DSP Event Type
Dsp2Xap and Xap2Dsp General Registers
Unmapped Event
Interrupt Controller
Clock Rate Select
MCU Registers
Registers used for Control By and Communications with the MCU
DSP Control, e.g. Run, Stop, Step etc.
MCU Event
MCU Register Interface
RAM Bank Select
Kalimba Clock: 120 or 80/n MHz from Analogue Section
G-TW-0006697.3.1

Figure 8.1: Kalimba DSP Peripheral Interfaces

The Kalimba DSP peripherals include:

- MMU interface, for stream transfers to/from the rest of the IC
- 3 memory mapped windows into the flash/ROM
- 2 memory mapped windows into the MCU RAM
- Memory-mapped register interface into the MCU RAM and I/O map
- Program memory window into flash/ROM with 1024-word or 512-word direct-mapped cache
- 2 1µs-resolution timers
- Interrupt controller with 3 priority levels
- Memory mapped access to the DSP program memory through DM1
- Clock rate divider controllable by either the DSP or the MCU
- Debug interface

8.1 MMU Interface

The Kalimba DSP MMU interface contains 12 virtual read ports and 12 virtual write ports. Figure 8.2 shows an example of the usage of these ports.

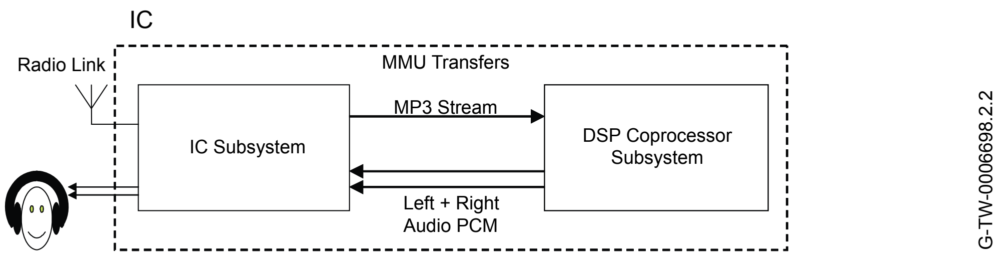


Figure 8.2: Example of MMU Interface Usage for a Wireless MP3 Player

8.1.1 Read Ports

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

- 8-bit, 16-bit or 24-bit word size
- Byte swap capability (little endian/big endian)
- Sign-extension, if required

8.1.2 Write Ports

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

- 8-bit, 16-bit or 24-bit word size
- Byte swap capability (little endian/big endian)
- 16-bit saturation, if required

8.2 DSP Timers

The Kalimba DSP timer features are:

- A 32-bit `TIMER_TIME` register (read-only) clocked at 1 MHz
- 2 trigger value registers, each, if enabled, cause an appropriate interrupt (LS 24-bits of `TIMER_TIME` only used for trigger values)

For control register details, see Appendix B.

8.3 Kalimba Interrupt Controller

This section covers the functionality and behaviour of the Kalimba DSP during interrupts.

8.3.1 Interrupt Controller Functionality

The functionality of the interrupt controller includes:

- Selectable interrupt sources:
  - Timer1 and Timer2
  - MCU Event
  - PIO Event
  - Buffer error / Software error
  - `SwEvent0`, `SwEvent1`, `SwEvent2` and `SwEvent3`
- 3 interrupt priority levels
- Registers to save and restore the interrupt controller which enables nested interrupts
- Optional event signal to cause clock rate change

See Appendix B for control register details.

The interrupt controller has many interrupt lines as inputs. The interrupt controller monitors these lines looking for rising edges. The lines which are monitored is controlled by `INT_SOURCES_EN`. The only time that it is not monitoring is if the interrupt controller is disabled (`INT_GBL_ENABLE == 0`). When interrupts are blocked (`INT_UNBLOCK == 0`) this monitoring of the lines is still done. This makes sure that an interrupt is never missed while they are blocked.

When a rising edge has been detected from an interrupt source, it is called a registered interrupt.

Note:

At any one time multiple different interrupt lines might be registered.

If the interrupt controller finds that it has a registered interrupt then it has two options:

1. The DSP core is in the process of handling a previous interrupt and running the early part of the ISR code up until the point of writing to the `INT_ACK` register.
2. All other cases.

In case 1 the interrupt controller waits until the `INT_ACK` register has been written to, then it looks at all the registered interrupts and the one with the highest priority that has asserted is recorded. This will become the next interrupt serviced by the ISR code. This is regardless of whether a higher priority one comes in between now and when the ISR code gets to run.

In case 2 then straight away the next interrupt is going to be whatever interrupt was registered. In the case of two registered interrupts appearing in the same clock cycle then the one with the highest priority is taken.

When the `INT_ACK` register is written to by the ISR code then (assuming the ISR code has support for nested interrupts) it is at this point that a higher priority interrupt can fire and effectively re-run the ISR code to handle that higher priority interrupt. When the higher-priority ISR has completed it drops back to competing the previous interrupt.

The key thing to appreciate is that the ISR code should write to the `INT_ACK` register early on so that higher-priority interrupts do not get held off being serviced for too long.

The interrupt controller itself understands the significance of higher and lower priority interrupts but it is the software ISR that provides the nesting of interrupts (with state stored to the software stack). The `INT_LOAD_INFO` and `INT_SAVE_INFO` registers enable the state of the interrupt controller to be saved and restored which is required for nested interrupts.

The `INT_LOAD_INFO` register also has to be made use of even without nested interrupt because it is the `INT_LOAD_INFO_CLR_REQ` bit within that register which clears the current interrupt from the store of “registered interrupts”, and also allows further interrupts to occur.

So with two interrupts at the same priority, it is once the `INT_LOAD_INFO` register has been written to that the next interrupt (of same priority) can fire. And likewise for a following lower priority interrupt.

And as explained above if a following higher priority interrupt occurs then it can get in as soon as `INT_ACK` is written to.

8.3.2 DSP Core Functionality During Interrupt

Upon reception of an interrupt, ie when the interrupt controller decides to interrupt the Kalimba core because of a registered interrupt, the DSP core performs the following:

- Memory-mapped register `MM_RINTLINK` is loaded with the contents of current `PC`
- `PC` is loaded with the address of the interrupt service routine, e.g. `0x0008`
- The flags register saved (the flags in bits [0:7] copied into bits [8:15]), and the `UM_FLAG` is cleared.
- If enabled, switches to faster interrupt clock rate

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

- Restores the `rFlags` register to the non-interrupt state
- Loads the `PC` with the contents of the `MM_RINTLINK` register

Note:

Saving/restoring of further registers is up to the programmer to do in software.

Further interrupts cannot occur while the interrupt service routine is executing, unless nested interrupts are enabled.

8.4 Generation of MCU Interrupt

Writing to the `DSP2MCU_EVENT_DATA` register sends an interrupt to the MCU. The MCU can see the value of this register. 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 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 PIO of the BlueCore device. Control of the PIO from the Kalimba DSP is as follows:

- Kalimba DSP can read IC PIO lines and enable interrupts on them
- Under the control of the MCU the Kalimba DSP can write to PIO lines
- Under the control of the MCU the Kalimba DSP can change the direction of PIO lines
- PIO line change can generate a Kalimba DSP interrupt

The MCU controls which PIO bits have write access permission for the Kalimba DSP. This information is set through VM functions if present, or BCCMDs otherwise.

See Appendix B for control register details.

8.6 MCU Memory Windows in DM2

The 2 windows in DM2 in Figure 8.1 enable the Kalimba DSP to access MCU memory. The primary use of this memory window is for passing message and control information.

The MCU controls access to this window. A start address and a size for each window can be set, and whether the DSP has read access or read/write access. The MCU firmware sets this information.

See Appendix B for control register details.

8.7 Non-volatile Memory Windows in DM2

The purpose of the non-volatile memory windows in DM2 is to permit the DSP access to the flash/ROM (up to 32Mb), this could be, e.g. to access further coefficients, or download a new program. The window size is 4Kwords for the small windows and 1Mwords for the large windows. The `FLASH_WINDOW[123]_START_ADDR` registers select which 4K/1M block is visible to the DSP for each of the 3 windows.

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 use program memory as data memory, either in 16-bit words or 24-bit words. 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.

See Appendix B for control register details.

8.9 PM Non-volatile Memory Window with Direct-mapped Cache

The PM non-volatile memory window permits the DSP to access program memory directly from flash/ROM. DSP memory-mapped registers control the start address and size of this flash/ROM space.

The 32-bit DSP PM bus is mapped into the 16-bit flash/ROM space with alternate MS and LS words stored in flash/ROM.

The MCU must enable the DSP to have access to the PM non-volatile memory window. The DSP must also enable access. When this interface is enabled, a 1024/512-word direct cache is mapped into the upper 512/256 words of physical program RAM, enabling zero-overhead accesses for cached instructions.

See Appendix B for control register details.

8.10 Clock Rate Divider Control

The Kalimba DSP can control its own clock frequency to run at 120 MHz, 80 MHz and further subdivisions of 80 MHz.

See Appendix B for control register details.

8.11 Debugging

The MCU registers and the 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

9 Document References

Document	Reference
Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s -- Part 3: Audio	ISO/IEC 11172-3
Information technology -- Generic coding of moving pictures and associated audio information -- Part 3: Audio	ISO/IEC 13818-3
Information technology -- Generic coding of moving pictures and associated audio information -- Part 7: Advanced Audio Coding (AAC)	ISO/IEC 13818-7

Appendix A Number Representation

This Appendix outlines the number representation used by the Kalimba DSP.

A.1 Binary Integer Representation

Kalimba uses 2's complement to represent signed integers. Figure A.1 shows an example of this format, with only 8-bits shown for clarity.

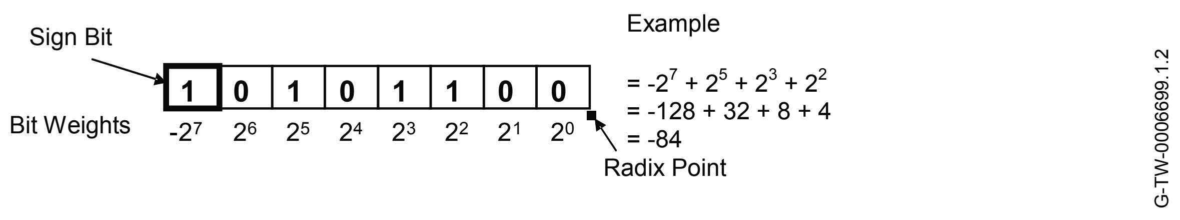


Figure A.1: Binary Integer Representation

A.2 Binary Fractional Representation

Kalimba implements a 24-bit fractional number type. The signed format can represent numbers from `-1.0` to `+0.99999988`, i.e. `0x800000` to `0x7fffff`. The unsigned format can represent numbers from `0` to `1.99999988`, i.e. `0x000000` to `0xffffff`. A signed fractional number type is illustrated below. While Kalimba's fractional type is 24 bits wide, the example in Figure A.2 shows only 8 bits for clarity.

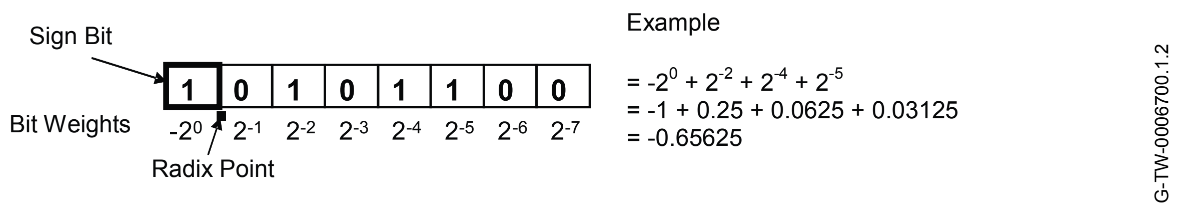
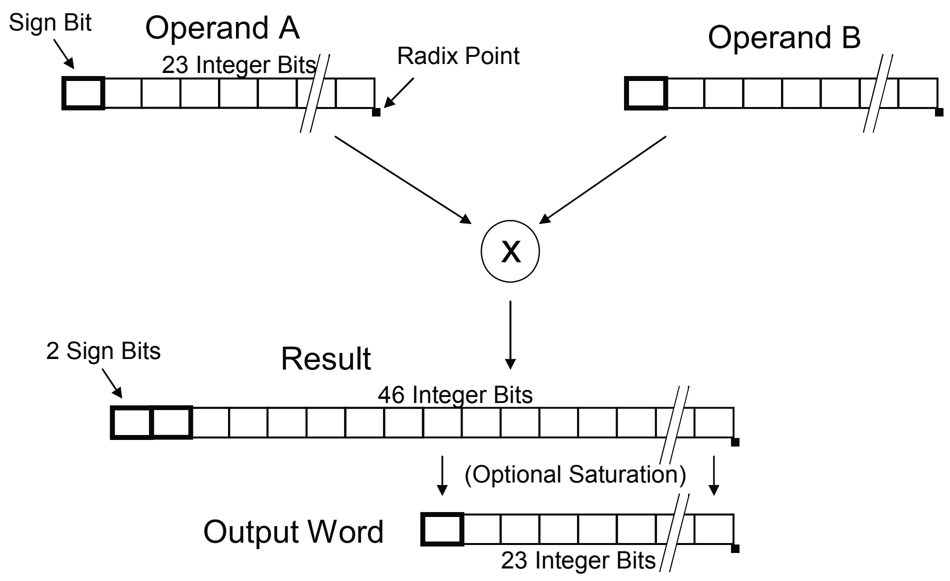


Figure A.2: Binary Fractional Representation

A.3 Integer Multiplication (Signed)



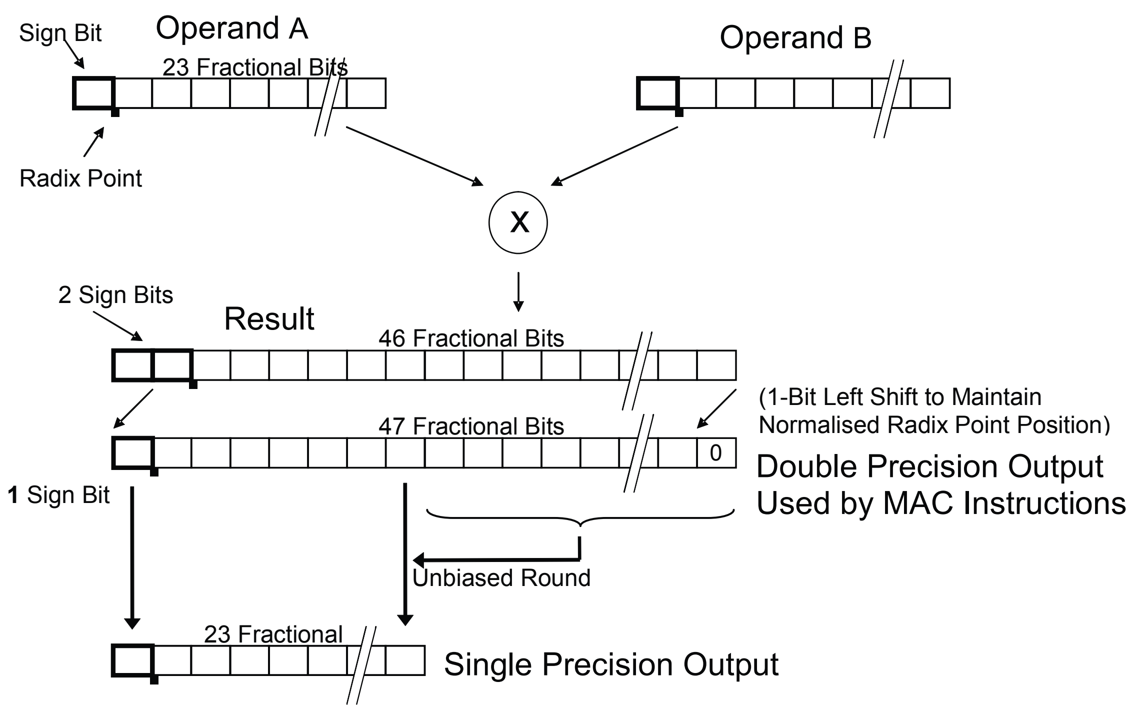
Example 1:	
Operand A	= 123
Operand B	= 456
Output Word	= 56088

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

G-TW-0006701.2.2

Figure A.3: Integer Multiplication

A.4 Fractional Multiplication (signed)



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)

G-TW-0006702.1.2

Appendix B DSP Memory Mapped Registers

B.1 DSP Memory Mapped I/O

As described in Section 6.1.3, the DSP memory mapped I/O forms a reserved part of the DM2 memory map. See the appropriate `_io_map.h` header file in the ADK for the specific CSR device for a list of the memory-mapped registers.

Appendix C Software Examples

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

C.1 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
```

C.2 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: 10 cycles

rMAC = r1 * r3 (UU);             // Compute LSW
r7 = rMAC0;                      // save Z0
rMAC = rMAC LSHIFT -24 (56bit); // shift right rMAC by 24 bits
rMAC = rMAC + r0 * r3 (SU);      // compute inner products
rMAC = rMAC + r2 * r1 (SU);
r6 = rMAC0;                      // save Z1
rMAC = rMAC ASHIFT -24 (56bit); // shift right rMAC by 24 bits
rMAC = rMAC + r0 * r2 (SS);      // compute MSWs
r5 = rMAC0;                      // save Z2
r4 = rMAC1;                      // save Z3
```

C.3 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: 10 cycles

rMAC = r1 * r3 (UU);             // Compute LSW
r7 = rMAC LSHIFT 23;             // save Z0
rMAC = rMAC LSHIFT -24 (56bit); // shift right 24 bits
rMAC = rMAC + r0 * r3 (SU);      // compute inner products
rMAC = rMAC + r2 * r1 (SU);
r6 = rMAC LSHIFT 23;             // save Z1
rMAC = rMAC ASHIFT -24 (56bit); // shift right 24 bits
rMAC = rMAC + r0 * r2 (SS);      // compute MSWs
r5 = rMAC LSHIFT 23;             // save Z2
r4 = rMAC LSHIFT -1;             // save Z3
```

CSR

C.4 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 = 1: 5 cycles
// N = 2: 4 cycles
// N > 2: N + 6 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;
```

C.5 Cascaded Bi-Quad IIR Filter

```
// Cascaded biquad IIR filter
//
// Equation of each section:
// y(n) = (b0*x(n) + b1*x(n-1) + b2*x(n-2)
//         - a1*y(n-1) - a2*y(n-2)) << 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:
// N = 1: 10 cycles
// N > 2: 8 * N + 5 cycles

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)
```

C.6 Radix-2 FFT

```
// An optimised FFT subroutine with a simple interface
//
// *************************************************************************
//  FILE
//          fast_fft.asm -  a fast FFT routine
//
//  CONTAINS
//          A fast optimised FFT routine with a simple interface
//
//  fftnpts:    64    128    256    512    1024    2048
//
//  No Clks:   TBA
//
//
//  REGISTERS
//          On entry: L0,L1,L4,L5 should be initialised to 0.
//
//          On exit: All registers altered
//
//  VARIABLES
//          Inputs:
//             fft_npts     - Number of points (a power of 2)
//             $Inputreal   - Input array real parts
//             $Inputimag   - Input array imag parts
//
//          Outputs:
//             $Refft       - Output array real parts
//             $Inputreal   - Output array imag parts
//
// ***************************************************************************

// Declare constants:

// Declare local variables:
.VAR/DM2 groups;
.VAR/DM2 node_space;
.include "twiddle_factors.h"

// Declare global variables:
```

```
.VAR/DM2 $fft_npts;

$fast_fft:

M1 = 1;
M2 = 2;
M3 = -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 = SIGNDET r0;
r1 = 22;
r9 = r1 - r0;           // log2(Npts) - 1

stage_loop:

   r10 = M[node_space];
   M0 = r10;                    // M0 = node_space

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

   I2 = &$Inputreal;            // I2 -> x0 in 1st group of stage
   I4 = I2;
   I0 = &$Inputimag;            // I0 -> y0 in 1st group of stage
   I6 = I0;
   I1 = I2 + M0;                // I1 -> x1 in 1st group of stage
   I5 = I6 + M0;                // I3 -> y1 in 1st group of stage

   I3 = &twid_real;             // I3 -> C of W0
   I7 = &twid_imag;             // I7 -> (-S) of W0
   group_loop:
      r10 = M0, r4 = M[I3,M1];
      r3 = M[I1,M1], r5 = M[I7,M1];
      r6 = r5;
      rMAC = r3*r4, r5 = M[I5,M1];
      rMAC = rMAC - r5 * r6, r1 = M[I2,M1], r2 = M[I6,M1];
      rMACB = r3*r6, r0 = r1+rMAC, r3=M[I1,M3];

      do bfly_loop;
         rMACB= rMACB+r5*r4, r0 = r1 - rMAC,  r1 = M[I2,M1], M[I4,M1] = r0;
         rMAC = r3*r4,       r0 = r2 + rMACB, M[I1,M2] = r0, r5 = M[I5,M3];
         rMAC = rMAC-r5*r6,  r0 = r2 - rMACB, M[I0,M1] = r0, r2 = M[I6,M1];
         rMACB= r3*r6,       r0 = r1 + rMAC,  r3 = M[I1,M3], M[I5,M2] = r0;
      bfly_loop:

      I0 = I0 + M0, r0 = M[I1,M0];
      I5 = I5 + M0, r0 = M[I2,M0], r0 = M[I6,M0];
      I5 = I5 - M1, r0 = M[I2,M3], r0 = M[I6,M3];
      r8 = r8 - M1, r0 = M[I4,M0];
   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 = SIGNDET r0;
r0 = r0 + 1;
r1 = 1;
r1 = r1 LSHIFT r0;
M3 = r1;                    // Bitreversed modifier
M0 = 0;

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;
```

Appendix D Bit-reversed Addressing Explained

Bit-reverse addressing is a processor mode designed to improve the performance of radix-2 FFT implementations. When enabled, the address that is fed out of the DSP core to the physical memory is 'bit-reversed' just before it connects to the memory. The mode can only be enabled on the addresses being formed by what is called address generator 1. This is the address generator concerned with index memory accesses using `I0-I3`. The mode is enabled by the use of the `BR_FLAG`, a bit in the `rFlags` register, see Section 4.5 for details.

When bit-reversed mode is enabled, the contents of index registers appear incorrect. However, these address values are reversed again before going to physical memory.

Take for example a buffer of 8 words located in physical memory at address `0xff8400`.

To access this with bit-reversed mode enabled you would access the first element of the buffer using the following code:

```
I0 = BITREVERSE(0xff8400);
```

```
r0 = M[I0,0];
```

Here `I0` equals `0x8010ff`. Which when re-bitreversed becomes `0xff8400`.

Bit-reversing on Kalimba Architecture 5 consists of reversing bits[0:22], but leaving bit[23] in place. This is because bit[23] selects the memory bank, DM1 or DM2.

The point of using bit-reversed addressing is to allow processing through a buffer in what is called bit-reversed order as opposed to the simpler linear order.

So if the buffer is of size 8. When processed in linear order that gives offsets of: 0, 1, 2, 3, 4, 5, 6, 7 However, when processed in bit reversed order the offsets would be: 0, 4, 2, 6, 1, 5, 3, 7. This order is worked out by looking at the binary representation of the offsets, see Table D.1:

Decimal Equivalent	Binary	Bit Reversed Binary	Decimal (Bit-reversed)
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table D.1: Binary Representation of 8 Word Buffer

To process the buffer in bit-reversed order means accessing the physical addresses:

- `0xff8400`
- `0xff8404`
- `0xff8402`
- `0xff8406`
- `0xff8401`
- `0xff8405`
- `0xff8403`
- `0xff8407`

However, with bit-reversed mode enabled the DSP address register, e.g. `I0`, needs to point to:

- `0x8010ff`
- `0x9010ff`
- `0xa010ff`
- `0xb010ff`
- `0xc010ff`
- `0xd010ff`
- `0xe010ff`
- `0xf010ff`

This is achieved by initialising `I0` to `0x8010ff` and then setting a modify register, e.g. `M0`, to `0x100000`.

For the more generic case of a buffer of size `N` (where `N` is always a power of 2), then the modify register needs setting to a value of 2^(23-log2(`N`))).

Terms and Definitions

Term	Definition
A2DP	Advanced Audio Distribution Profile
AAC	Advanced Audio Coding
ADC	Analogue to Digital Converter
AG	Address Generator
ALU	Arithmetic Logic Unit
BlueCore®	Group term for CSR's range of Bluetooth wireless technology ICs
Bluetooth®	Set of technologies providing audio and data transfer over short-range radio connections
codec	Coder decoder
CSR	Cambridge Silicon Radio
DAC	Digital to Analogue Converter
DSP	Digital Signal Processor (or Processing)
e.g.	*exempli gratia*, for example
etc	*et cetera*, and the rest, and so forth
FFT	Fast Fourier Transform
FP	Frame Pointer
i.e.	*Id est*, that is
I/O	Input/Output
IC	Integrated Circuit
ISR	Interrupt Service Routine
Kalimba	An open platform DSP co-processor, enabling support of enhanced audio applications, such as echo and noise suppression and file compression / decompression
LS	Least Significant
LSB	Least Significant Bit (or Byte)
Mb	Megabit
MCU	MicroController Unit
MMU	Memory Management Unit
MP3	MPEG-1 audio layer 3
MS	Most Significant
MSB	Most Significant Bit (or Byte)
NOP	No OPeration
NVM	Non-Volatile Memory
PC	Program Counter

Term	Definition
PIO	Programmable Input/Output, also known as general purpose I/O
RAM	Random Access Memory
ROM	Read Only Memory
SBC	Sub-Band Coding
SP	Stack Pointer
SPI	Serial Peripheral Interface
VM	Virtual Machine