Jan 23, 2004

Using MAXQ's Multiplier Module

A MAXQ-based microcontroller can be equipped with a 16x16 hardware multiplier peripheral module. This application note describes how the multiplier operates and how to write code for this module in applications to maximize math performance.

Introduction

The hardware multiplier (hereafter a Multiply-Accumulate, or MAC) module is a very powerful tool, especially for applications that require heavy calculations, e.g., firmware for an electricity-metering microcontroller. This multiplier is capable of executing the multiply or multiply-negate or multiply-accumulate or multiply-subtract operation for signed or unsigned operands in a single machine cycle, and even faster for special cases. This article examines the hardware organization and functionality, explains how to write code for MAC and provides simple examples of typical calculations using MAXQ's hardware multiplier.

Getting Started

To begin, we need basic knowledge about the MAXQ architecture, register map and instruction set, which information can be obtained from the MAXQ Family User's Guide or from any MAXQ-based microcontroller data sheet, e.g., MAXQ2000. We also need reference to the MAC hardware description, which is in the MAXQ Family User's Guide document. Certain familiarity with assembly language in general, and with MAXQ assembler in particular, is assumed.

Multiplier Overview

From the programming point of view, the MAC module appears as 8 special function registers, mapped in modules' space from M0 through M5. One register (MCNT) contains control bits, two registers (MA, MB) designated for input operands, three registers (MC0, MC1, MC2 considered as one long MC register) keep the output result, i.e. either product or accumulation, and two read-only registers (MC0R,MC1R considered as one MCR register, "R" stands for "read-only") are used in special cases as output registers. The accumulator MC is configurable, typically 40- or 48-bits wide. In any configuration, there are two 16-bit registers MC0 and MC1, whereas the MC2 register is implemented as 8-bit, 16-bit or other. For the sake of simplicity let us assume that the accumulator MC is 48-bits wide and MAC registers are mapped in module M3:

```
#define MCNT    M3[0]
#define MA      M3[1]
#define MB      M3[2]
#define MC2     M3[3]
#define MC1     M3[4]
#define MC0     M3[5]
#define MC1R    M3[6]
#define MC0R    M3[7]
```

The multiplier is always ready to do math, usage is very simple and consists of four basic steps: 1) set configuration, 2) load operands, 3) wait, and 4) unload result. Figure 1 represents MAC registers and typical operation flow. Actual calculations start immediately after step 2. When an operation is started, the multiplier calculates the product of operands MA and MB whose product is then either placed or added to the accumulator MC by the end of step 3. The MCR register participates as internal working memory and we normally do not care for it. The way that the register works is rather tricky and confusing at first sight. It is advisable for beginners to ignore the MCR register, with exception for special case MCW=1. Advanced user can enjoy the benefits of shorter and faster code by sneaking intermediate data from the MCR register, but such user must fully understand the way it works.

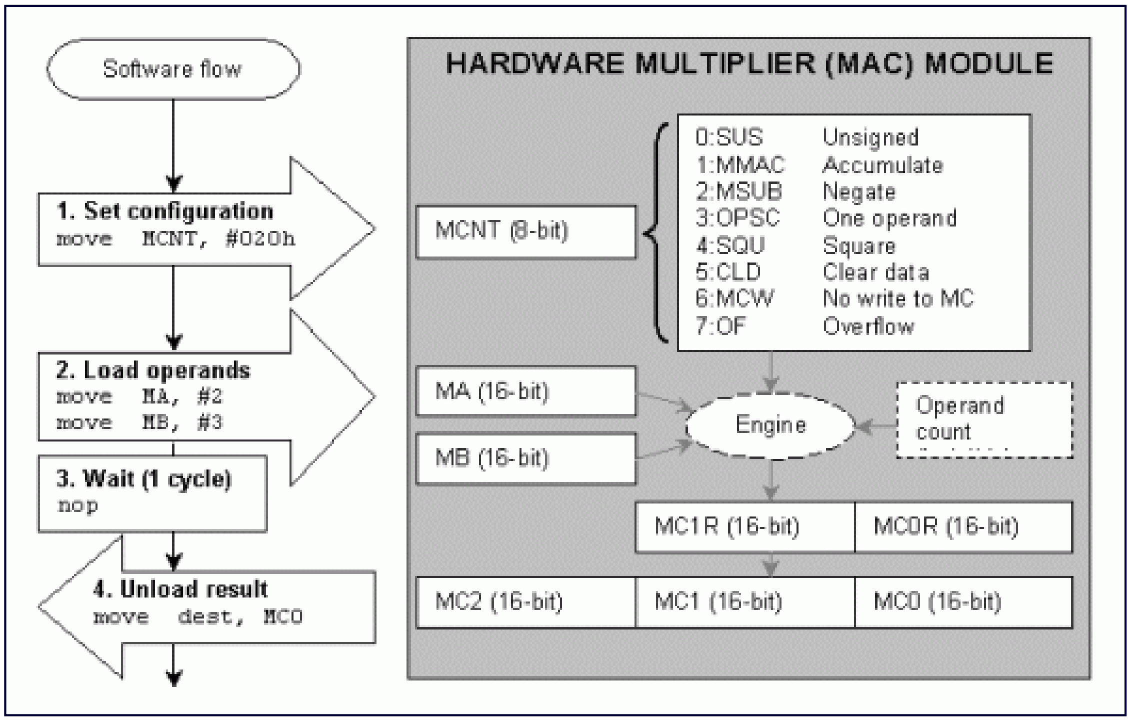

Figure 1. Hardware multiplier registers structure and typical operation flow.

The set configuration step (step 1) implies a simple write to the control register MCNT. Besides updating the data it also implicitly initializes the hardware, i.e., prepares MAC to accept new operands and perform new operation. This step can be skipped if desired configuration is already set in the MCNT register. When an operation is started, the multiplier always uses the current content of MCNT.

Actual calculations start as soon as the necessary operands are loaded into the MAC (step 2). There are no limitations on how quickly data is entered into operand registers or on the order of data entry. The invisible operand count register keeps track of all writes to MA and MB and triggers a calculation accordingly. For example, in two-operand modes, the first loaded operand can then be re-loaded many times without starting a calculation. The calculations are started only when the second operand is loaded. In one-operand modes, the calculations are triggered when the first operand is loaded to the MA or MB register. The operand count gets initialized by hardware when a calculation is started or when the MCNT register is written.

The wait state (step 3) is needed when the result of operation is expected from the MC register. Think of it as a two-phase process: 1) execute an operation, e.g. calculate the product MA*MB, and 2) update accumulator MC. The first phase is done right away, but second phase is equivalent to "move" instruction. It takes one machine cycle of time for MAC hardware to execute this move. However, the wait state can be skipped in special case MCW=1 when the result is expected from MCR register, because the output of the first phase is stored there. One can put any meaningful instruction instead of "no operation" during the wait state, except a write into MCNT, which will disrupt current multiplier operation.

Multiplier Configuration Options

The meaning of the control bits in the MCNT register is straightforward.

- bit 0: SUS treat operands MA, MB as unsigned 16-bit numbers when SUS=1; treat operands MA, MB as signed 16-bit numbers when SUS=0;
- bit 1: MMAC add product to accumulator MC when MMAC=1; place product into accumulator MC when MMAC=0;
- bit 2: MSUB use (-MA) value instead of MA when MSUB=1; use MA operand as is when MSUB=0;
- bit 3: OPSC loading one operand will trigger calculations when OPSC=1; loading two operands will trigger calculations when OPSC=0;
- bit 4: SQU loading one operand will copy the loaded value into the other operand register and trigger calculations when SQU=1 (square mode); square mode is disabled when SQU=0;
- bit 5: CLD all data registers (MA,MB,MC, operand count) are cleared to 0 when CLD=1. This bit is automatically cleared by hardware.

- bit 6: MCW prevent multiplier from writing result to accumulator MC when MCW=1. multiplier will write the result into accumulator MC when MCW=0;
- bit 7: OF read-only bit, signifies that errors (like overflow) occurred when OF=1; no errors if OF=0;

The operand select bit OPSC turns on the one-operand mode. The multiplier will start calculation after loading the first operand when OPSC=1. The SQU bit switches the square mode, in which the first loaded operand is automatically copied to the other operand register and the operation is triggered. The OPSC bit has no meaning when SQU=1.

The MCW bit is a write-protect flag for the accumulator MC. Setting MCW=1 allows performing a multiplication without disturbing the MC register. In this special case the result of operation (to be precise, the least 32 bits of the result) can be obtained without wait state delay from the MCR register. Note that the write-protection feature only affects the multiplier operations. User still can write directly to the MC register when MCW=1, but MAC cannot update it. That provides a possibility of unconventional use of the MAC as an additional storage. Five 16-bit registers MA, MB, MC0, MC1, MC2 can temporarily store data, pointers, counters, etc, which is especially useful for the 8-bit MAXQ10 core. By setting MCW=1 we ensure that the accumulator MC will not be accidentally updated when we write to MA or MB.

The multiplier sets read-only status bit OF when critical error happens during operation:

- overflow or underflow of MC register for unsigned operation;
- overflow or underflow of MC register for signed operation;
- attempt to execute unsigned multiply-negate operation.

In all other cases the OF bit is cleared by hardware after operation. There is no OF bit set for multiply-only and signed multiply-negate operation, the bit is cleared. Note that in case of overflow/underflow event the MC register contains the correct low bits of the result.

As mentioned above, any write to MCNT also implicitly initializes the operand count register and prepares MAC to start new operation. More details about the control bits can be found below in the code examples section.

Multiplier Data Flow

The function of accumulator MC and the read-only register MCR can be presented by the following equations:

$$\begin{aligned} MC_n &= MC_{n-1} * MMAC + (-1)^{MSUB} (MA_n * MB_n), && \text{for MCW=0,} && (1)\\ &= MC_{n-1}, && \text{for MCW=1,} && \\ MCR_n &= MC_n * MMAC + (-1)^{MSUB} (MA_n * MB_n), && \text{independent of MCW,} && (2) \end{aligned}$$

where n is the execution cycle, MMAC is accumulation flag (0 or 1), MSUB is negation flag (0 or 1) in the MCNT register. The MCR register keeps only 32 low bits of the right-hand side of the equation (2), but it gets updated immediately at execution cycle n. The MC register keeps a full length of the result, extended with OF bit if necessary, but both MC and OF get updated 1 cycle later. The data flow in the multiplier during operation is shown on the Figure 2.

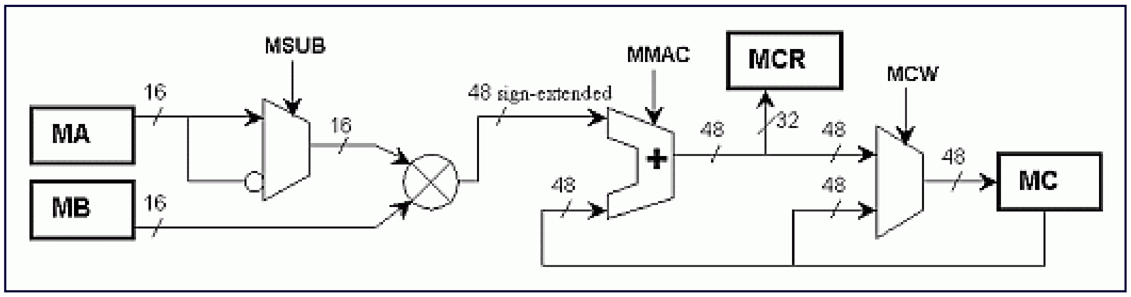

Figure 2. Multiplier operation data flow.

Code Examples

It is convenient to denote the control bits with corresponding names,

```
#define SUS   0x01
#define MMAC  0x02
#define MSUB  0x04
#define OPSC  0x08
#define SQU   0x10
#define CLD   0x20
#define MCW   0x40
#define OF    0x80
```

and use those names instead of numbers. For example, the instruction 'move MCNT, #(SUS+MMAC+MSUB+SQU)' is a mnemonic for 'move M3[0], #23'. Programming the multiplier requires the data movement to and from the MAC, so the most used instruction is 'move dest,src'. Every example in this section is accompanied with a table showing resulting changes in the multiplier registers after execution of each instruction. A bold entry denotes that a register was updated, even if the value is not changed. All numbers are hexadecimal in tables, the 'xxxx' entry denotes the value of no importance. Note the behaviour of the MCR register: it gets updated every time the content of MA, or MB, or MC is changed. Examples 1-7 illustrate simple use of the MAC for a single operation, examples 8-12 show more complex configurations, e.g. how to avoid the wait state for continuous calculations. Example 13 demonstrates how the multiplier can speed up a square root calculation.

1. Unsigned Multiplication.
   Calculate the product 3*5 and place it to the register A[0].

```
move    MCNT, #(SUS+CLD)        ; unsigned, multiply-only, clear data
move    MA, #3  ; load first operand into MA
move    MB, #5  ; load second operand into MB
                                ; the product is in MCR register
nop                     ; wait for MAC to update MC
move    A[0],MC0        ; unload the product
```

   The A[0] register contains the product 3*5 = 15 (0x000F).

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT,#(SUS+CLD)	01	0000	0000	0000	0000	0000	0000	0000
move MA, #3	01	0003	0000	0000	0000	0000	0000	0000
move MB, #5	01	0003	0005	0000	0000	0000	0000	000F
nop	01	0003	0005	0000	0000	000F	0000	000F
move A[0],MC0	01	0003	0005	0000	0000	000F	0000	000F

   Advanced user may save a cycle by reading from the MCR ('move A[0],MC0R' instead of 'nop').

2. Signed Multiplication.
   Calculate the product 3*(-5) and place it to the register A[0].

```
move    MCNT, #(CLD); signed, multiply-only, clear data registers
move    MA, #3  ; load first operand into MA
move    MB, #(-5)       ; load second operand into MB
                                ; the product is in MCR register
nop                     ; wait for MAC to update MC
move    A[0],MC0        ; unload the product
```

   The A[0] register contains the product 3*(-5) = -15 (0xFFF1).

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT,#(CLD)	00	0000	0000	0000	0000	0000	0000	0000
move MA, #3	00	0003	0000	0000	0000	0000	0000	0000
move MB, #(-5)	00	0003	FFFB	0000	0000	0000	FFFF	FFF1
nop	00	0003	FFFB	FFFF	FFFF	FFF1	FFFF	FFF1
move A[0],MC0	00	0003	FFFB	FFFF	FFFF	FFF1	FFFF	FFF1

Advanced user may save a cycle by reading from the MCR ('move A[0],MC0R' instead of 'nop').

3. Signed Multiply-Negate operation.
   Calculate the negative product of 3 and (-5), place it to the register A[0].

```
move    MCNT, #(MSUB+CLD)       ; signed, multiply-negate, clear data registers
move    MA, #3  ; load first operand into MA
move    MB, #(-5)       ; load second operand into MB
                                ; the result is in MCR register
nop                     ; wait for MAC to update MC
move    A[0],MC0        ; unload the result
```

The A[0] register contains the result -(3*(-5)) = 15 (0x000F).

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT,#(MSUB+CLD)	04	0000	0000	0000	0000	0000	0000	0000
move MA, #3	04	0003	0000	0000	0000	0000	0000	0000
move MB, #(-5)	04	0003	FFFB	0000	0000	0000	0000	000F
nop	04	0003	FFFB	0000	0000	000F	0000	000F
move A[0],MC0	04	0003	FFFB	0000	0000	000F	0000	000F

Advanced user may save a cycle by reading from the MCR ('move A[0],MC0R' instead of 'nop').

4. Unsigned Multiply-Accumulate operation.
   Calculate the expression 2+3*5 and place it to the register A[0].

```
move    MCNT, #(SUS+MMAC)       ; unsigned, multiply-accumulate
                        ; no CLD set, data registers hold their content
move    MC0, #2 ; pre-load accumulator
move    MC1, #0 ; pre-load accumulator
move    MC2, #0 ; pre-load accumulator
move    MA, #3  ; load first operand into MA
move    MB, #5  ; load second operand into MB
                                ; the result is in MCR register!
nop                     ; wait for MAC to update MC
                                ; the MCR was changed to MC+MA*MB!
move    A[0],MC0        ; unload the result
```

The A[0] register contains the result 2+3*5 = 17 (0x0011).

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT,#(SUS+MMAC)	03	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
move MC0, #2	03	xxxx	xxxx	xxxx	xxxx	0002	xxxx	xxxx
move MC1, #0	03	xxxx	xxxx	xxxx	0000	0002	xxxx	xxxx
move MC2, #0	03	xxxx	xxxx	0000	0000	0002	xxxx	xxxx

move MA, #3	03	0003	xxxx	0000	0000	0002	xxxx	xxxx
move MB, #5	03	0003	0005	0000	0000	0002	0000	0011
nop	03	0003	0005	0000	0000	0011	0000	0020
move A[0],MC0	03	0003	0005	0000	0000	0011	0000	0020

Note the behavior of the MCR register. It follows the equation (2) above, i.e. always reflects the current state of MA, MB, and MC. In this example, the MCR register holds 32 bits of the result only 1 machine cycle during wait state, then changes its value because the MC register gets updated after the wait state, the new value is MCR = 0x0020 = 32 = 17+3*5 = MC+MA*MB.

Advanced user may save a cycle by reading from the MCR ('move A[0],MC0R' instead of 'nop'). But two things must be remembered: i) there are only 32 low bits of the result available from the MCR, and ii) even those 32 bits are available limited time, only one machine cycle in this example configuration.

5. Signed Multiply-Accumulate operation.
   Calculate the expression (-2)+3*(-5) and place it to the register A[0].

```
move    MCNT, #(MMAC)   ; signed, multiply-accumulate
move    MC0, #0FFFEh    ; pre-load accumulator
move    MC1, #0FFFFh    ; pre-load accumulator
move    MC2, MC1                ; pre-load accumulator
move    MA, #3          ; load first operand into MA
move    MB, #(-5)               ; load second operand into MB
                                ; the result is in MCR register!
nop                     ; wait for MAC to update MC
                                ; the MCR was changed to MC+MA*MB!
move    A[0],MC0        ; unload the result
```

The A[0] register contains the result (-2)+3*(-5) = -17 (0xFFEF).

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT,#(MMAC)	02	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
move MC0, #0FFFEh	02	xxxx	xxxx	xxxx	xxxx	FFFE	xxxx	xxxx
move MC1, #0FFFFh	02	xxxx	xxxx	xxxx	FFFF	FFFE	xxxx	xxxx
move MC2, MC1	02	xxxx	xxxx	FFFF	FFFF	FFFE	xxxx	xxxx
move MA, #3	02	0003	xxxx	FFFF	FFFF	FFFE	xxxx	xxxx
move MB, #(-5)	02	0003	FFFB	FFFF	FFFF	FFFE	FFFF	FFEF
nop	02	0003	FFFB	FFFF	FFFF	FFEF	FFFF	FFE0
move A[0],MC0 02 0003 FFFB FFFF FFFF FFEF FFFF FFE0								

As in example 4 above, advanced user may save a cycle by reading from the MCR ('move A[0],MC0R' instead of 'nop') before this register gets updated with new value MCR = 0xFFFF FFE0 = -32 = -17+3*(-5) = MC+MA*MB.

6. Unsigned Multiply-Subtract operation.
   Calculate the expression 17-3*5 and place it to the register A[0]. Let us assume that the operands are stored in the registers A[1]=3 and A[2]=5.

```
move    MCNT, #(SUS+MMAC+MSUB)  ; unsigned, multiply-subtract
move    MC0, #17        ; pre-load accumulator
move    MC1, #0 ; pre-load accumulator
move    MC2, #0 ; pre-load accumulator
move    MA, A[1]        ; load first operand into MA
move    MB, A[2]        ; load second operand into MB
                                ; the result is in MCR register!
nop                     ; wait for MAC to update MC
                                ; the MCR was changed to MC-MA*MB!
move    A[0],MC0        ; unload the result
```

The A[0] register contains the result 17-3*5 = 2 (0x0002).

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT,#(SUS+MMAC+MSUB)	07	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
move MC0, #17	07	xxxx	xxxx	xxxx	xxxx	0011	xxxx	xxxx
move MC1, #0	07	xxxx	xxxx	xxxx	0000	0011	xxxx	xxxx
move MC2, #0	07	xxxx	xxxx	0000	0000	0011	xxxx	xxxx
move MA, A[1]	07	0003	xxxx	0000	0000	0011	xxxx	xxxx
move MB, A[2]	07	0003	0005	0000	0000	0011	0000	0002
nop	07	0003	0005	0000	0000	0002	FFFF	FFF3
move A[0],MC0	07	0003	0005	0000	0000	0002	FFFF	FFF3

As in example 4 above, advanced user may save a cycle by reading from the MCR ('move A[0],MC0R' instead of 'nop') before this register gets updated with new value MCR = 0xFFFF FFF3 = -13 = 2-3*5 = MC-MA*MB.

7. Signed Multiply-Subtract operation.
   Calculate the expression (_2)-3*(-5) and place it to the register A[0]. Let us assume that the operands are stored in the registers A[1]=3 and A[2]= -5.

```
move    MCNT, #(MMAC+MSUB); signed, multiply-subtract
move    MC0, #0FFFEh    ; pre-load accumulator
move    MC1, #0FFFFh    ; pre-load accumulator
move    MC2, #0FFFFh    ; pre-load accumulator
move    MA, A[1]        ; load first operand into MA
move    MB, A[2]        ; load second operand into MB
                                ; the result is in MCR register!
nop                     ; wait for MAC to update MC
                                ; the MCR was changed to MC-MA*MB!
move    A[0],MC0        ; unload the result
```

The A[0] register contains the result (-2)-3*(-5) = 13 (0x000D).

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT,#(SUS+MMAC+MSUB)	06	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
move MC0, #0FFFEh	06	xxxx	xxxx	xxxx	xxxx	FFFE	xxxx	xxxx
move MC1, #0FFFFh	06	xxxx	xxxx	xxxx	FFFF	FFFE	xxxx	xxxx
move MC2, #0FFFFh	06	xxxx	xxxx	FFFF	FFFF	FFFE	xxxx	xxxx
move MA, A[1]	06	0003	xxxx	FFFF	FFFF	FFFE	xxxx	xxxx
move MB, A[2]	06	0003	FFFB	FFFF	FFFF	FFFE	0000	000D
nop	06	0003	FFFB	0000	0000	000D	0000	001C
move A[0],MC0	06	0003	FFFB	0000	0000	000D	0000	001C

As in example 4 above, advanced user may save a cycle by reading from the MCR ('move A[0],MC0R' instead of 'nop') before this register gets updated with new value MCR = 0x001C = 28 = 13-3*(-5) = MC-MA*MB.

8. One-operand mode.
   This mode is useful when an application needs to perform an operation with the same constant many times. Since

one of the operands is not changing, no need to load it every time. Just load the constant operand once and switch the multiplier to the one-operand mode. Then every write to another operand register will trigger the calculations.

For example, assume that analog-to-digital conversion hardware repeatedly returns voltage as 16-bit values in a register called ADC with the least significant bit scaled as 1/8 V, and a microcontroller application must re-calculate every voltage value to mV units and store it in data RAM. That re-calculation can be done by multiplying the row data by coefficient 1000/8=125 (0x7D). Following code will do the job.

```
; configure multiplier initially (this code is executed once)
move    MA, #125                ; load the constant factor into MA.
move    MCNT, #(OPSC)   ; signed, multiply-only, one-operand mode
......
; this code is executed repeatedly with every new ADC value
move    MB, ADC ; load operand into MB
                                ; the result is in MCR register!
move    DP[0],#8        ; set address=0x0008 while waiting for MAC to update MC
move    @DP[0], MC0     ; store the result in data memory at address 0x0008
......
```

The example shows, by the way, how the wait state can be used for setting an address instead of wasting time with nop. Let assume that three consecutive raw voltages were 5.0V, 1.5V, -2.5V (0x0028, 0x000C, 0xFFEC) then software flow can be represented by following table.

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MA, #125	xxxx	007D	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
move MCNT,#(OPSC)	08	007D	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
.....								
move MB, ADC	08	007D	0028	xxxx	xxxx	xxxx	0000	1388
move DP[0], #8	08	007D	0028	0000	0000	1388	0000	1388
move @DP[0], MC0	08	007D	0028	0000	0000	1388	0000	1388
.....	Data memory[8] = 0x1388 (= 5000 mV)							
move MB, ADC	08	007D	000C	xxxx	xxxx	xxxx	0000	05DC
move DP[0], #8	08	007D	000A	0000	0000	05DC	0000	05DC
move @DP[0], MC0	08	007D	000A	0000	0000	05DC	0000	05DC
.....	Data memory[8] = 0x05DC (= 1500 mV)							
move MB, ADC	08	007D	FFEC	xxxx	xxxx	xxxx	FFFF	F63C
move DP[0], #8	08	007D	FFEC	FFFF	FFFF	F63C	FFFF	F63C
move @DP[0], MC0	08	007D	FFEC	FFFF	FFFF	F63C	FFFF	F63C
.....	Data memory[8] = 0xF63C (= -2500 mV)							

Again, advanced user may save a cycle of the repeated code by reading from the MCR:

```
; configure multiplier initially (this code is executed once)
move    DP[0],#8        ; set address=0x0008
move    MA, #125                ; load the constant factor into MA.
move    MCNT, #(OPSC)   ; signed, multiply-only, one-operand mode
......
; this code is executed repeatedly with every new ADC value
move    MB, ADC ; load operand into MB
                                ; the result is in MCR register!
move    @DP[0], MC0R; store the result in data memory at address 0x0008
......
```

But remember the limitations: only 32 low bits are available from the MCR, and its content is subject to change without notice!

9. Square mode.
In this mode, a square can be calculated by loading only 1 operand. For example, a microcontroller application must calculate RMS value of the voltage. Part of that calculation is a sum of squares of the samples, which sum can be easily done by using MAC's square mode. Assume the voltage samples are 16-bit signed values scaled in mV units and stored (repeatedly) in data RAM at address 0x0008. The code might look as follows.

```
; configure multiplier initially (this code is executed once)
move    DP[0],#8        ; set address=0x0008
move    MCNT,#(MMAC+SQU+CLD)    ; signed, square-accumulate, clear registers
......
; this code is executed repeatedly with every new sample
move    MA, @DP[0]      ; load operand into MA (assume DP[0] is active data ptr)
                                ; the result is in MCR register!
nop                     ; wait for MAC to update MC
                                ; the MCR was changed to MC+MA*MB!
......
```

The sum of squares is being accumulated in the MC register. Assume that the first three consecutive raw voltages were 5.0V, 1.5V, -2.5V (0x1388, 0x05DC, 0xF63C), then software flow can be represented by following table.

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move DP[0], #8	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
move MCNT,#(MMAC+SQU+CLD)	12	0000	0000	0000	0000	0000	0000	0000
.....								
move MA, @DP[0]	12	1388	1388	0000	0000	0000	017D	7840
nop	12	1388	1388	0000	017D	7840	02FA	F080
.....	MC = 0x017D 7840 (= 25 000 000 mV2)							
move MA, @DP[0]	12	05DC	05DC	0000	017D	7840	019F	CD50
nop	12	05DC	05DC	0000	019F	CD50	01C2	2260
.....	MC = 0x019F CD50 (= 25 000 000 + 2 250 000 mV2)							
move MA, @DP[0]	12	F63C	F63C	0000	019F	CD50	01FF	2B60
nop	12	F63C	F63C	0000	01FF	2B60	025E	8970
.....	MC = 0x01FF 2B60 (= 25 000 000 + 2 250 000 + 6 250 000 mV2)							

10. Accumulator write-protect feature.
This feature (MCW control bit) allows a multiply without accumulate, i.e. the contents of the accumulator are not updated by multiplier's operation. It may be useful if application requires to perform two different tasks in parallel, one involves accumulation and another one is just to multiply numbers. Then the multiplier can be switched back and forth between those tasks without saving/restoring its accumulator. Say, we can combine the above examples 8 and 9 to accumulate the sum of squares in parallel with conversion to millivolts by using MCW control bit.

```
; initialize MAC (this code is executed once)
move    DP[0],#8        ; set address=0x0008
move    MCNT,#(CLD)     ; clear registers
......
; this code is executed repeatedly with every new ADC value
; multiply by constant factor but keep the MC intact
move    MCNT, #(MCW); signed, multiply-only, MCW-protect
move    MA, #125        ; load constant factor into MA
move    MB, ADC ; load raw sample into MB
                                ; the result is in MCR register!
                                ; special case MCW=1, no wait state needed here!
move    @DP[0],MC0R     ; store the result in data memory at address 0x0008
; accumulate sum of squares in MC
move    MCNT, #(MMAC+SQU)       ; signed, square-accumulate, clear MCW-protect
move    MA, MC0R        ; load operand into MA
                                ; the result is in MCR register!
nop                     ; wait for MAC to update MC
                                ; the MCR was changed to MC+MA*MB!
......
```

The sum of squares is being accumulated in MC register, while result of multiplication (conversion to mV) is accessible via MCR register. Note that no wait state is necessary for the multiplication. Let assume that the first three consecutive raw voltages were 5.0V, 1.5V, -2.5V (0x0028, 0x000C, 0xFFEC) then software flow can be represented by following table.

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move DP[0], #8	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
move MCNT,#(CLD)	00	0000	0000	0000	0000	0000	0000	0000
.....								
move MCNT, #(MCW)	40	xxxx	xxxx	0000	0000	0000	xxxx	xxxx
move MA, #125	40	007D	xxxx	0000	0000	0000	xxxx	xxxx
move MB, ADC	40	007D	0028	0000	0000	0000	0000	1388
move @DP[0], MC0R	40	007D	0028	0000	0000	0000	0000	1388
move MCNT, #(MMAC+SQU)	12	007D	0028	0000	0000	0000	0000	1388
move MA, MC0R	12	1388	1388	0000	0000	0000	017D	7840
nop	12	1388	1388	0000	017D	7840	02FA	F080
.....	Data memory[8] = 0x1388 (= 5000 mV)MC = 0x017D 7840 (= 25 000 000 mV2)							
move MCNT, #(MCW)	40	xxxx	xxxx	0000	017D	7840	xxxx	xxxx
move MA, #125	40	007D	xxxx	0000	017D	7840	xxxx	xxxx
move MB, ADC	40	007D	000C	0000	017D	7840	0000	05DC
move @DP[0], MC0R	40	007D	000C	0000	017D	7840	0000	05DC
move MCNT, #(MMAC+SQU)	12	007D	000C	0000	017D	7840	0000	05DC
move MA, MC0R	12	05DC	05DC	0000	017D	7840	019F	CD50
nop	12	05DC	05DC	0000	019F	CD50	01C2	2260
.....	Data memory[8] = 0x05DC (= 1500 mV)MC = 0x019F CD50 (= 25 000 000 + 2 250 000 mV2)							
move MCNT, #(MCW)	40	xxxx	xxxx	0000	019F	CD50	xxxx	xxxx
move MA, #125	40	007D	xxxx	0000	019F	CD50	xxxx	xxxx
move MB, ADC	40	007D	FFEC	0000	019F	CD50	FFFF	F63C
move @DP[0], MC0R	40	007D	FFEC	0000	019F	CD50	FFFF	F63C
move MCNT, #(MMAC+SQU)	12	007D	FFEC	0000	019F	CD50	FFFF	F63C
move MA, MC0R	12	F63C	F63C	0000	019F	CD50	01FF	2B60
nop	12	05DC	05DC	0000	01FF	2B60	025E	8970

.....	Data memory[8] = 0xF63C (= -2500 mV)MC = 0x01FF 2B60 (= 25 000 000 + 2 250 000 + 6 250 000 mV2)

11. Continuous (back-to-back) calculations.
The wait state can be used for loading an operand for the next operation without disrupting current operation. For example, if the sum S=1*2+3*4+5*6+7*8+9*10 is to be calculated, that could be done by the following code.

```
move    MCNT,#(MMAC+CLD)        ; signed, multiply-accumulate, clear registers
move    MA, #1  ; load operand MA
move    MB, #2  ; load operand MB, trigger 1*2
move    MA, #3  ; load operand MA while waiting for MAC to complete 1*2
move    MB, #4  ; load operand MB, trigger 3*4
move    MA, #5  ; load operand MA while waiting for MAC to complete 3*4
move    MB, #6  ; load operand MB, trigger 5*6
move    MA, #7  ; load operand MA while waiting for MAC to complete 5*6
move    MB, #8  ; load operand MB, trigger 7*8
move    MA, #9  ; load operand MA while waiting for MAC to complete 7*8
move    MB, #10 ; load operand MB, trigger 9*10
nop                     ; wait for MAC to update MC
```

The sum S=190 (0xBE) is in MC register. Note that all wait states except last one were not wasted but used for loading operands in this example.

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT, #(MMAC+CLD)	02	0000	0000	0000	0000	0000	0000	0000
move MA, #1	02	0001	0000	0000	0000	0000	0000	0000
move MB, #2	02	0001	0002	0000	0000	0000	0000	0002
move MA, #3	02	0003	0002	0000	0000	0002	0000	0008
move MB, #4	02	0003	0004	0000	0000	0002	0000	000E
move MA, #5	02	0005	0004	0000	0000	000E	0000	0022
move MB, #6	02	0005	0006	0000	0000	000E	0000	002C
move MA, #7	02	0007	0006	0000	0000	002C	0000	0056
move MB, #8	02	0007	0008	0000	0000	002C	0000	0064
move MA, #9	02	0009	0008	0000	0000	0064	0000	00AC
move MB, #10	02	0009	000A	0000	0000	0064	0000	00BE
nop	02	0009	000A	0000	0000	00BE	0000	0118

Similar technique can be utilized in one-operand or square mode. For example, if the sum $S=1^2+2^2+3^2+4^2+5^2+6^2+7^2$ is to be calculated, it can be done with following code.

```
move    MCNT,#(MMAC+SQU+CLD)    ; signed, square-accumulate, clear registers
move    MA, #1  ; load operand into MA, trigger 1*1
move    MA, #2  ; load operand into MA, trigger 2*2 while waiting
move    MA, #3  ; load operand into MA, trigger 3*3 while waiting
move    MA, #4  ; load operand into MA, trigger 4*4 while waiting
move    MA, #5  ; load operand into MA, trigger 5*5 while waiting
move    MA, #6  ; load operand into MA, trigger 6*6 while waiting
move    MA, #7  ; load operand into MA, trigger 7*7 while waiting
nop                     ; wait for MAC to update MC
```

The sum S=140 (0x8C) is in MC register. Note that all wait states except last one were not wasted but used for loading operands.

INSTRUCTION	MCNT	MA	MB	MC2	MC1	MC0	MC1R	MC0R
move MCNT, #(MMAC+SQU+CLD)	12	0000	0000	0000	0000	0000	0000	0000
move MA, #1	12	0001	0001	0000	0000	0000	0000	0001
move MA, #2	12	0002	0002	0000	0000	0001	0000	0005
move MA, #3	12	0003	0003	0000	0000	0005	0000	000E
move MA, #4	12	0004	0004	0000	0000	000E	0000	001E
move MA, #5	12	0005	0005	0000	0000	001E	0000	0037
move MA, #6	12	0006	0006	0000	0000	0037	0000	005B
move MA, #7	12	0007	0007	0000	0000	005B	0000	008C
nop	12	0007	0007	0000	0000	008C	0000	00BD

12. Overflow.
    In case of overflow/underflow event the MC register contains the correct low bits of the result, so the OF flag can be used as multiplier's carry/borrow bit to implement an accumulator which is longer than MC. For example, a 64-bit accumulator could consist of 16-bit register A[0] and 48-bit MC register. Then 64-bit multiply-accumulate operation could be done by following code.

```
move    MCNT, #(SUS+MMAC)       ; unsigned, multiply-accumulate
move    MA, #32768      ; load operand into MA
move    MB, #16384      ; load operand into MB
nop                     ; wait for MAC to update MC
                                ; the OF flag is set (if necessary)
move    C, MCNT.7       ; copy OF bit into Carry
addc    #0              ; A[0]+=0+Carry (assume A[0] is active MAXQ's accumulator)
```

The improvised 64-bit accumulator is incremented by 32768*16384. Let assume the initial value was 0x1234 FFFF F000 4321, then software flow is represented by the following table, new value is 0x1235 0000 1000 4321 = old value + 0x2000 0000.

INSTRUCTION	MCNT	MA	MB	CARRY	A[0]	MC2	MC1	MC0
move MCNT,#(SUS+MMAC)	03	xxxx	xxxx	xxxx	1234	FFFF	4321	
move MA, #32768	03	8000	xxxx	xxxx	1234	FFFF	F000	4321
move MB, #16384	03	8000	4000	xxxx	1234	FFFF	F000	4321
nop	83	8000	4000	xxxx	1234	0000	1000	4321
move C, MCNT.7	83	8000	4000	1	1234	0000	1000	4321
addc #0	83	8000	4000	0	1235	0000	1000	4321

13. Square root subroutine.
    The MAC module can help to speed up with various math calculations, not only multiplication and accumulation. For example, examine a subroutine which calculates the square root of an unsigned 32-bit number A. This subroutine finds the result, an unsigned 16-bit number X such that $X^2 \le A<(X+1)^2$, by the hit-or-miss method. Starting with X_0=0x0000 it consecutively switches each bit of the partial result Xn from 0 to 1, starting with the most significant bit. If $X_n^2 \le A$ it keeps 1 in the bit position (hit), otherwise adopts 0 for the bit (miss). The efficiency of this method comes from the fact that multiplier can calculate $(A-X_n^2)$ in a few cycles. The whole process is complete in n=16 iterations and requires about 200 machine cycles. It is easy to see that without MAC module the calculation of a square root would take much longer for the MAXQ20 processor.

```
;=========================================
;   SQRT32MAC subroutine for MAXQ20
```

```
;========================================
;   input:  A[3:2]  = operand, unsigned 32-bit
;   output: A[0]    = sqrt(operand), unsigned 16-bit
;   used (modified) resources:  AP,APC,A[1:0],MAC registers
;========================================
sqrt32mac_MAXQ20:
; local data map:
;       A[0] = X
;       A[1] = iteration mask
move  A[0],#0           ; init X=0x0000
move  A[1],#08000h      ; init mask=0x8000
move  MCNT,#(SUS+MMAC+MSUB+SQU) ; unsigned, square-subtract
sqrt32mac_MAXQ20_loop:
; start an iteration
move  APC,#080h; set Acc to A[0], no auto-inc
or    A[1]      ; X |= mask (set bit to try)
; get MC = AA-X^2
move  MC2,#0    ; pre-load MC with A[3:2]
move  MC1,A[3]
move  MC0,A[2]
move  MA,A[0]   ; load operand MA <- X
nop             ; wait for MAC to update MC
; now MC=AA-X^2
; check if (AA-X^2) >< 0?
move  C,MC2.0   ; Carry ><- sign(AA-X^2)
jump NC,sqrt32mac_MAXQ20_update_mask    ; jump if No Carry (hit)
; undo mask bit (miss)
xor   A[1]      ; X ^= mask (clear tried bit)
sqrt32mac_MAXQ20_update_mask:
move  AP,#01    ; set Acc to A[1]
sr                      ; A[1] >>= 1 (shift mask right, LSb -> Carry)
sjump NC,sqrt32mac_MAXQ20_loop  ; next iteration if No Carry
; all 16 bits done, exit
ret     ; return
;   End SQRT32MAC subroutine for MAXQ20
;========================================
```

The table below demonstrates few last iterations for A=0x00AA 63CB. The number comes from the above example 9 and represents the voltage mean square value (=0x01FF 2B60/3). The square root from this number is the RMS voltage in mV units. The star symbol (*) denotes the active MAXQ's accumulator, A[0] or A[1]. The result X=0x0D0D (=3341 mV) is in A[0] register in the end.

INSTRUCTION	A[0] (X)	A[1] (MASK)	MCNT	MA	MB	MC2	MC1	MC0
....								
; iteration 14:								
move APC, #080h	*0D08	0004	17	0D08	0D08	0000	0000	938B
Or A[1]	*0D0C	0004	17	0D08	0D08	0000	0000	938B
move MC2, #0	*0D0C	0004	17	0D08	0D08	0000	0000	938B
move MC1, A[3]	*0D0C	0004	17	0D08	0D08	0000	00AA	938B
move MC0, A[2]	*0D0C	0004	17	0D08	0D08	0000	0000	63CB
move MA, A[0]	*0D0C	0004	17	0D0C	0D0C	0000	0000	63CB
nop	*0D0C	0004	17	0D0C	0D0C	0000	0000	2B3B
move C,MC2.0	Carry = 0							
sjump NC,<...>	Jump (hit)							

move AP,#01	0D0C	*0004	17	0D0C	0D0C	0000	0000	2B3B
Sr	0D0C	*0002	Carry = 0					
jump NC,<...>	Jump to next iteration							
; iteration 15:								
move APC, #080h	*0D0C	0002	17	0D0C	0D0C	0000	0000	2B3B
Or A[1]	*0D0E	0002	17	0D0C	0D0C	0000	0000	2B3B
move MC2, #0	*0D0E	0002	17	0D0C	0D0C	0000	0000	2B3B
move MC1, A[3]	*0D0E	0002	17	0D0C	0D0C	0000	00AA	2B3B
move MC0, A[2]	*0D0E	0002	17	0D0C	0D0C	0000	0000	63CB
move MA, A[0]	*0D0E	0002	17	0D0E	0D0E	0000	0000	63CB
nop	*0D0E	0002	17	0D0E	0D0E	FFFF	FFFF	F707
move C,MC2.0	Carry = 1							
jump NC,<...>	No Jump (miss)							
xOr A[1]	*0D0C	0002	17	0D0E	0D0E	FFFF	FFFF	F707
move AP,#01	0D0C	*0002	17	0D0E	0D0E	FFFF	FFFF	F707
sr	0D0C	*0001	Carry = 0					
sjump NC,<...>	Jump to next iteration							
; iteration 16:								
move APC, #080h	*0D0C	0001	17	0D0E	0D0E	FFFF	FFFF	F707
Or A[1]	*0D0D	0001	17	0D0E	0D0E	FFFF	FFFF	F707
move MC2, #0	*0D0D	0001	17	0D0E	0D0E	0000	FFFF	F707
move MC1, A[3]	*0D0D	0001	17	0D0E	0D0E	0000	00AA	F707
move MC0, A[2]	*0D0D	0001	17	0D0E	0D0E	0000	0000	63CB
move MA, A[0]	*0D0D	0001	17	0D0D	0D0D	0000	0000	63CB
nop	*0D0D	0001	17	0D0D	0D0D	0000	0000	1122
move C,MC2.0	Carry = 0							
jump NC,<...>	Jump (hit)							
move AP,#01	0D0D	*0001	17	0D0D	0D0D	0000	0000	1122
sr	0D0D	*0000	Carry = 1					
sjump NC,<...>	No Jump to next iteration							
ret	Return with A[0]= sqrt(A[3:2])							

More Information

MAXQ2000: QuickView -- Full (PDF) Data Sheet -- Free Samples