AN205433

FR, MB91470/480, 32-BIT MAC Unit

Author:Cypress

The MB91470 and MB91480 Series are Flash microcontrollers especially optimized for motor control and other control applications. It features a Multi-Function Timer incl. Waveform generator to generate the output voltage waveforms needed to control three-phase motors, and other dedicated peripherals like multiple ADCs as well as the 32-bit MAC (Multiply -Accumulate) unit for fast processing, which is also referred to as µDSP.

Contents

1 Introduction....1
2 Features....2
2.1 Features....2
2.2 Block diagram....2
2.3 MAC unit registers description....3
2.4 Instructions of the MAC unit....4
3 Operation of the MAC unit....8
4 Usage of the MAC unit....9
4.1 Using the MAC unit for FIR filters....9
4.2 Using the MAC unit for PID control loops....11
4.3 Using the MAC unit for function approximation....12
5 Appendix....14
5.1 Register list and memory map of the MAC unit....14
5.2 Related documents....15
5.3 Related software examples....15
5.4 Glossary....15
Document History....16

1 Introduction

The MB91470 and MB91480 Series are Flash microcontrollers especially optimized for motor control and other control applications. It features a Multi-Function Timer incl. Waveform generator to generate the output voltage waveforms needed to control three-phase motors, and other dedicated peripherals like multiple ADCs as well as the 32-bit MAC (*M*ultiply -*AC*cumulate) unit for fast processing, which is also referred to as µDSP.

The MAC unit has an own instruction and data RAM and works completely independent of the CPU as soon as the calculation is started. It can execute 32 bits * 32 bits + 72 bits Multiply-Accumulate calculations in one machine cycle (12.5ns @80MHz), and shift values arbitrarily in Y-RAM for efficient filter processing. Additional jump (also as conditional branch) and store commands make the MB91470/480 Series MAC unit ideal for digital filtering of any kind, like FIR (*Finite Impulse Response*) or IIR (*Infinite Impulse Response* or *recursive*) filters, and many other algorithms. Once the algorithm is set up in the MAC unit, the CPU only has to update new input values, start the MAC unit and can work on other tasks while waiting for the result. Also, an interrupt request can be generated upon calculation end. Therefore, it is easily possible e.g. to process a signal sampled by the A/D converter with a 64-tap filter at high sampling rates and low CPU usage. Some examples will be shown later in this document.

This application note will explain the configuration and usage of the 32-bit MAC unit, and give some short application examples as well as some programming helps.

2 Features

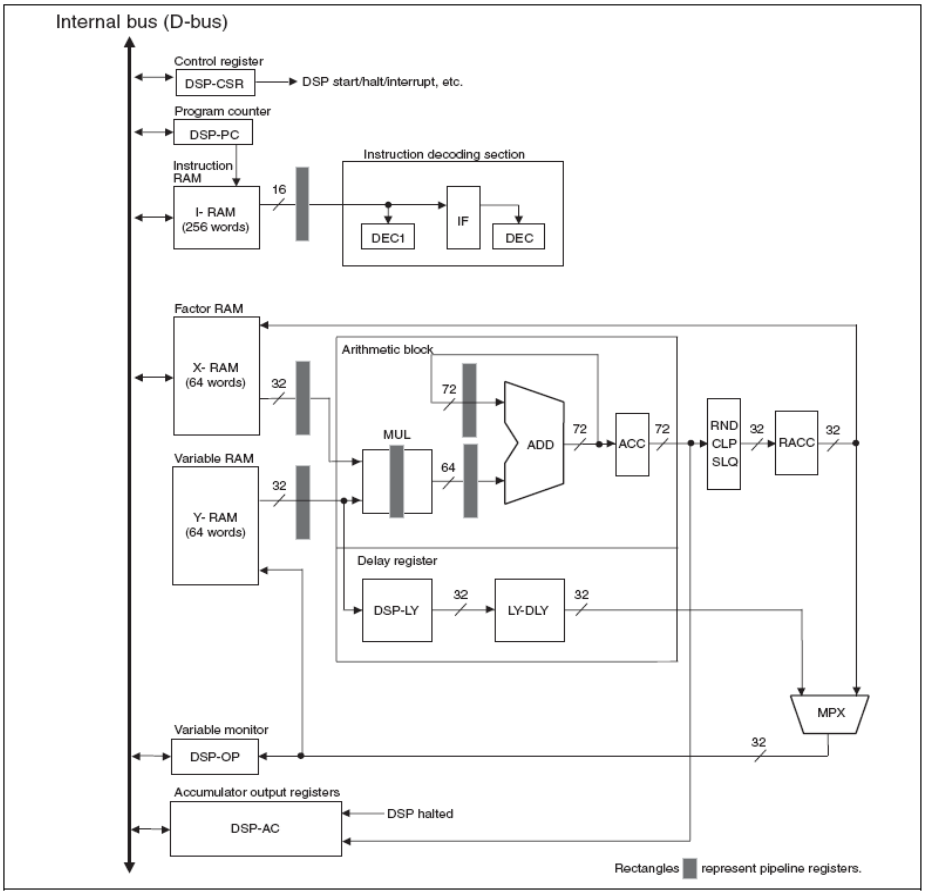
This section describes the features, the register set and the block diagram of the 32-bit MAC unit.

2.1 Features

- High-speed multiply-accumulate operation (in one system clock cycle)
- Own program flow, independent of CPU
- Data format: 32-bit fixed-point (32 * 32 + 72 bits), Q0 (Integer) as well as Q26-Q32 fixed-point format selectable
- Instruction RAM (IRAM): 256 words * 16 bits
- Data RAM: 2 * 64 words * 32 bits (X-RAM: Coefficients, Y-RAM: Variables)
- Rounding and saturation of output value available
- Instruction set: MAC, STR (store), JMP (Jump/Branch, also conditional), NOP
- Delay handling: Variables can be shifted arbitrarily in Y-RAM together with the MAC operation
- Variable monitor: Up to eight Y-RAM values as well as the accumulator content can be monitored during operation without stopping the µDSP program execution

2.2 Block diagram

Figure 1. Block diagram of the 32-bit MAC unit



2.3 MAC unit registers description

2.3.1 DSP Control/Status Register (DSP-CSR)

This 8-bit register controls the operation state of the MAC unit and also holds the status flags (IRQ, SatDSP and RunDSP).

Bit 7: SatDSP: This flag is set when saturation occurs while clipping is enabled in the STR command (CLP=1). This bit is automatically cleared at the beginning of operation.

Bits 6…4: USR2-0: These bits can be used to control the program flow of the MAC unit. When the JMP command is used with COND=1, the condition coded in the JMP command is compared with these bits, and the jump is only performed on a match. Since the DSP-CSR register can be written and read also during MAC operation, this can be used together with the variable monitor function to react on intermediate results, e.g. for loop programming.

Bit 3: IrqDSP: This is the interrupt request flag for the MAC unit. If the MAC IRQ is enabled (IeDSP=1), this bit signals an IRQ to the CPU if the SIRQ bit in a STR or JMP instruction is set.

Bit 2: IeDSP: Interrupt request enable bit for the MAC unit

Bit 1: reserved (always write ‘0’ to this bit)

Bit 0: GoDSP (write) / RunDSP (Read): Setting this bit starts the MAC unit program execution, if the MAC unit is not already running. Reading the RunDSP bit returns ‘1’ as long as the calculations are ongoing, and ‘0’ when the operation is halted (e.g. after a NOP instruction with the HLT-bit set to ‘1’ was executed).

2.3.2 DSP Program Counter (DSP-PC)

The program counter is an 8-bit register. It points to the memory address (in I-RAM) holding the next instruction to be executed by the MAC unit. The program counter is automatically incremented after a command was executed, and can be overwritten by a JMP command. Note that the PC counts 16-bit words starting from the I-RAM base adress, so that it is incremented by 1 after every execution, so also odd numbers occur (0x00-0xff).

The program counter has to be initialized to point to the first instruction to be executed before the MAC unit is started by setting the GoDSP bit. By pre-setting the DSP-PC, different algorithms in the I-RAM can be selected before starting the MAC unit.

2.3.3 DSP Delay Register (DSP-LY)

If the LDLY bit in a MAC instruction is set, first the content of the DSP-LY register is transferred to the LY-DLY register. Then, the Y-RAM data of the actual MAC instruction is copied to the DSP-LY register. If the STLY bit of the same MAC instruction is set, the content of the LY-DLY register is written to the Y-RAM address of the actual MAC instruction after MAC execution. Note that this write operation takes one clock cycle besides the MAC cycle. All together, this mechanism allows very effective programming of digital filters, since the input data can be automatically shifted through the filter stages by the MAC unit without CPU interaction. In this case, every filter stage needs two clock cycles for processing, already including the data transfer in the filter.

Please also refer to the MAC command description later in this document for details.

The DSP-LY register can only be accessed by the CPU if the MAC unit is halted. The LY-DLY register is not directly accessible by the CPU.

2.3.4 DSP Variable Monitor Register (DSP-OT0 to DSP-OT7)

The content of the first eight Y-RAM words are permanently mirrored to the DSP-OT0 to DSP-OT7 registers. Since the MAC unit’s memory is not accessible for the CPU during MAC operation, these registers can be used to monitor Y-RAM data by the CPU, e.g. for intermediate results or conditional actions. These registers are read-only and their read value is indeterminate after a reset.

2.3.5 DSP Accumulator Output Registers (DSP-AC0 to DSP-AC2)

These three 32-bit registers can be used to monitor the content of the 72-bit accumulator. They can be accessed also when the MAC unit is operating (DSP_CSR:RunDSP = 1), so that the CPU can e.g. check for boundaries etc. while calculations are ongoing. DSP-AC0 holds bits 71..64, DSP-AC2 holds bits 21..0.

2.3.6 X-RAM

The X-RAM (64 words * 32 bits) stores the first factor (i.e. the coefficients) for the MAC operations.

This RAM area can only be accessed by the CPU while the MAC unit is idle (RunDSP = 0). Initialize used X-RAM areas before starting the MAC operations.

2.3.7 Y-RAM

The Y-RAM (64 words * 32 bits) stores the second factor (i.e. the variables) for the MAC operations. Variables in this memory can be shifted using the delay function.

This RAM area can only be accessed by the CPU while the MAC unit is idle (RunDSP = 0). Initialize used Y-RAM areas before starting the MAC operations.

2.3.8 I-RAM

The I-RAM (256 words * 16 bits) is the instruction (program) RAM of the MAC unit. The instructions stored in this area are sequentially executed while the DSP-PC is incremented, and also a JMP command exists. This RAM area can only be accessed by the CPU while the MAC unit is idle (RunDSP = 0). Transfer the MAC program to I-RAM and set the MAC unit program counter (DSP-PC) before starting the MAC operations.

2.4 Instructions of the MAC unit

The MAC unit implements the following commands:

MAC: Multiply-and-Accumulate (works as multiply when CLAC bit is used to clear accu)

STR: Store

JMP: Jump

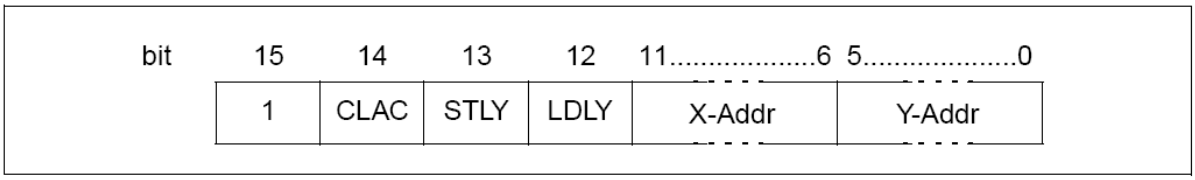
NOP: No Operation

Each of these commands has several option bits, which are described in the following. The instructions are stored in the I-RAM of the MAC unit, from where they are executed independently of the CPU. Usage examples are given later in this document.

2.4.1 MAC Instruction

The MAC instruction is the core functionality of the MAC unit. Depending on the setting of the CLAC bit, either a MAC (Multiply-and-Accumulate) or a MUL (Multiply) instruction is executed.

Operation code:



[bit14] **CLAC** (Clear ACC)

Setting this bit causes the instruction to act as a multiplication instruction.

"0": ACC = ACC + data@X-Addr x data@Y-Addr (multiply-and-accumulate instruction)

"1": ACC = 0 + data@X-Addr x data@Y-Addr (multiplication instruction)

[bit13] **STLY** (STore LY)

When this bit is set, the value in the LY-DLY register is written to the Y-RAM address defined by Y-Addr after the instruction was executed. The execution time increases by one cycle if this bit is set.

[bit12] **LDLY** (LoaD LY)

When this bit is set, first, the actual content of the DSP-LY register is moved to the LY-DLY register. Then, the content of the Y-RAM defined by Y-Addr is loaded to the DSP-LY register. These transfers have no influence on execution time of the MAC command.

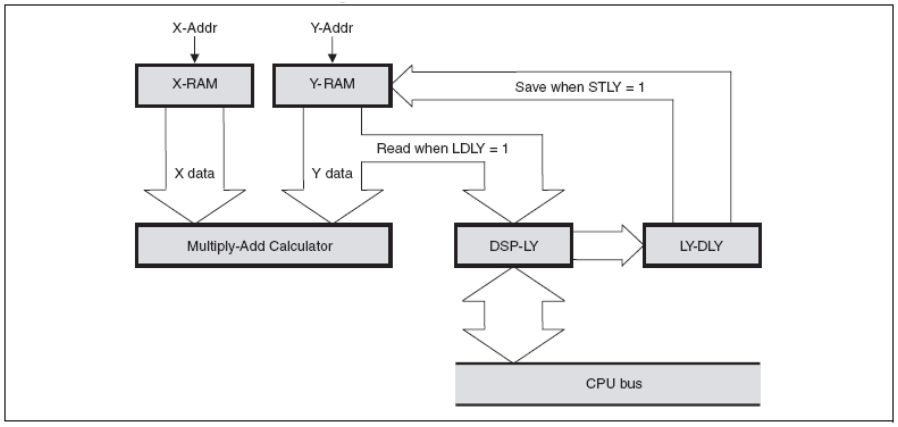
[bit11 to bit6] **X-Addr** (X-RAM Address)

These bits specify the X-RAM address for the actual MAC command (0-63)

[bit5 to bit0] **Y-Addr** (Y-RAM Address)

These bits specify the Y-RAM address for the actual MAC command (0-63)

Figure 2. Block diagram of the delay function



Note that X-RAM, Y-RAM and I-RAM addresses are word addresses counted from the beginning of the corresponding memory area; e.g. X-RAM address 0 = 0xC000, X-RAM address 1 = 0xC004, Y-RAM address 4 = 0xC110 etc. X- and Y-RAM data words are 32-bit wide, while the instructions are 16-bit wide.

Note:

To raise data throughput, the 32-bit MAC unit uses a seven-stage pipeline. Therefore, the actual write access to Y-RAM (when STLY bit is set) occurs as the sixth cycle after the delayed write instruction. Do not read from the written address before five instructions have been executed between the delayed write access and the read access, as the read value will not be valid before this time.

2.4.2 STR (Store) Instruction

The Store command converts the 72-bit accumulator value to a 32-bit value in accordance with the RND, CLP, and SLQ flags, and stores the result in the data RAM specified by the SLY flag and X/Y-Addr. Additionally, the entire 72-bit accumulator value can be read from the DSP-AC0…2 registers.

Operation code:

bit15	bit14	bit13	bit12	bit11	bit10	bit9.....bit7	bit6	bit5.....bit0
0	1	1	0	RND	CLP	SLQ	SLY	X/Y-Addr

[bit11] **RND** (Rounding)

This bit specifies whether to perform rounding for 32-bit data specified by the SLQ bits. Rounding rounds the 16-bit data based on the bit immediately below the LSB (add 1 if lower bit is 1, no action if 0).

[bit10] **CLP** (Clipping)

If this bit is set, the result value is either 0x7FFFFFFF (positive) or 0x80000000 (negative) if the result is not correctly represented by the current output format selection due to an overflow of the 32-bit value. This is done by comparing the MSB (bit71) of the accumulator with the MSB of the selected output word selected by the SLQ bits. If rounding is enabled, the result after rounding is used.

[bit9 to bit7] **SLQ**

These bits specify the output format of the 72-to-32 bit truncation, and thereby the position of the decimal point. Q0 (integer) as well as Q26 to Q32 fixed-point formats can be selected. See table below for details.

[bit6] **SLY**

This bit specifies the transfer destination RAM area:

0: X-RAM

1: Y-RAM

[bit5 to bit0] **X/Y Addr** (destination RAM Address)

These bits specify the address in X- or Y-RAM (0…63) to store the 32-bit calculation result. From there, the CPU can read the result after operation is halted. If one of the first eight Y-RAM addresses is used for the result, the CPU can also read the value using the variable monitor registers, and the calculation can continue directly after the store instruction.

Figure 3. Available fixed-point formats

SLQ bits	Overflow evaluation bits	32-bit data to be transferred	Rounded bit	Fixed-point format	Assembler code
000	bit71 to bit59	bit59 to bit28	bit27	Q28	Q12
001	bit71 to bit60	bit60 to bit29	bit28	Q29	Q13
010	bit71 to bit61	bit61 to bit30	bit29	Q30	Q14
011	bit71 to bit62	bit62 to bit31	bit30	Q31	Q15
100	bit71 to bit63	bit63 to bit32	bit31	Q32	Q8
101	bit71 to bit31	bit31 to bit0	(None)	Q0	Q9
110	bit71 to bit57	bit57 to bit26	bit25	Q26	Q10
111	bit71 to bit58	bit58 to bit27	bit26	Q27	Q11

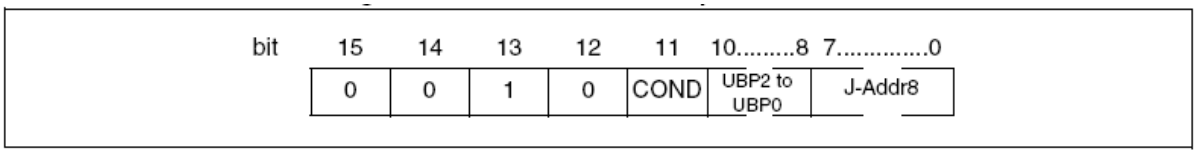
Note:

To raise data throughput, the 32-bit MAC unit features a seven-stage pipeline. Therefore, the actual write access to X- or Y-RAM occurs as the sixth cycle after the store instruction. Do not read from the written address before five instructions have been executed between the store command and the read access, as the read value will not be valid before this time.

2.4.3 JMP Instruction (Branch Instruction)

The JMP instruction loads the address specified by the J-Addr8 bits to the MAC unit program counter (DSP-PC). If the COND bit is set, the branch is only performed if UBP2…0 bits match the USR2…0 bits in the DSP-CSR register.

Operation code:



[bit11] **COND** (CONDition)

0: Always branch (Jump)

1: Branch to J-Addr8 if [UBP2…0] = [DSP_CSR_USR2…0]

[bit10 to bit8] **UBP2 to UBP0** (condition specification)

These bits set the condition to use for the conditional branch. The condition is established if these bits match the value of the USR2, USR1, and USR0 bits in the DSP-CSR register. These bits must be set to "000" if an unconditional branch is specified (when COND = 0).

[bit7 to bit0] **J-Addr8** (Jump Address)

These bits specify the branch target address in I-RAM. This value is written to the DSP-PC when the JMP is executed.

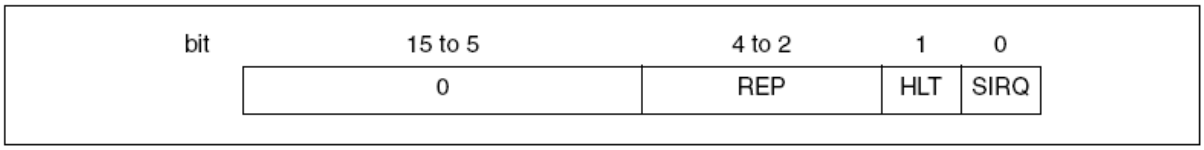
Note:

Due to the instruction pipeline, the actual branch occurs after the two instructions that follow the branch instruction are executed. Therefore, the JMP instruction should always be followed by two NOP instructions with REP = 001, HLT = 0, SIRQ = 0.

2.4.4 NOP Instruction (No Operation)

The NOP instruction can be used to wait a certain number of cycles, and to halt the MAC unit after calculation. Also an interrupt request can be issued.

Operation code:



[bit4..2]: **REP** (Repeat count setting)

These bits can be used to set the number of wait cycles to be executed. REP = ‘001’ will execute a single wait cycle, while REP = ‘000’ will wait eight cycles.

[bit1] **HLT** (Halt)

If this bit is set, the MAC unit halts program execution after the NOP instruction, so that the MAC unit memory can be accessed by the CPU. To enable the HLT function, be sure to set REP to ‘001’. Also, because of the instruction pipeline, the actual halt occurs two instructions after the HLT instruction. Therefore, the HLT instruction should always be followed by two NOP instructions with REP = 001, HLT = 0, SIRQ = 0.

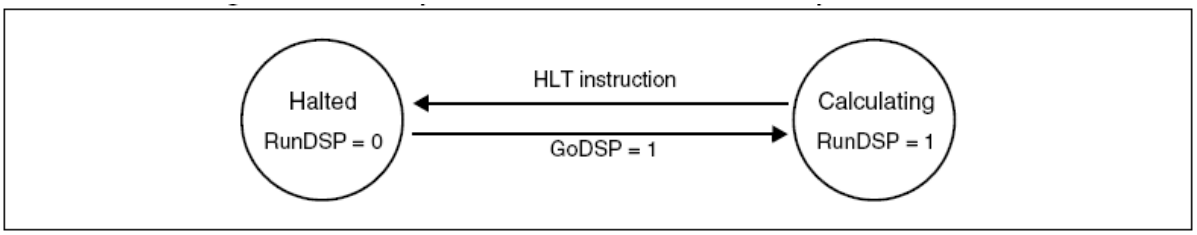
[bit0] **SIRQ** (Signal Interrupt request)

If this bit is set, the MAC unit sets the IrqDSP flag in the DSP-CSR register as soon as the calculation stops. If enabled (DSP_CSR:IeDSP = 1), an interrupt request can be generated. To enable the SIRQ function, be sure to set REP to ‘001’.

3 Operation of the MAC unit

This section explains the operation of the MAC unit.

The operation of the MAC unit is controlled by the DSP-CSR register and the instructions stored in I-RAM. The MAC unit starts program execution when ‘1’ is written to the GoDSP bit, and stops when a HLT command is executed (NOP with HLT=’1’). Note that some of the registers and memory areas of the MAC unit can only be accessed by the CPU while the MAC unit is stopped (RunDSP = 0).



- Stopped: The multiply-and-accumulate macro is stopped. The CPU can access instruction RAM (IRAM), data RAM (X-RAM, Y-RAM), and all registers of the MAC unit.
- Running: The multiply-and-accumulate unit is operating. When ‘1’ is written to the GoDSP bit while the MAC unit is stopped, it enters this state and starts program execution from the current DSP-PC (program counter) address. While in this state, not all registers and memory areas of the MAC unit can be accessed. Trying to read from a non-accessible area will return indeterminate results, while writing to it has no effect. A NOP command with set HLT = ‘1’ and REP = ‘001’ causes the MAC unit to halt and return to the ‘stopped’ state.

The general initialization of the MAC unit consists of the following steps:

1. Check that the MAC unit is halted (DSP_CSR:RunDSP = 0)
2. Transfer coefficients and data to X- and Y-RAM according to application
3. Transfer MAC unit instructions to I-RAM
4. Set the DSP-PC to the first instruction to be executed
5. Start the MAC unit (write ‘1’ to the GoDSP bit)

The MAC unit starts calculation and continues program execution until one of the following conditions occurs:

- A NOP command with HLT = ‘1’ and REP = ‘001’ is executed
- A reset occurs

Note that the 72-bit accumulator of the MAC unit is not initialized automatically. Therefore, the first MAC command of the program should clear the accumulator (CLAC = 1), except when this behavior is explicitly desired.

Since the MAC unit operates independently of the rest of the MCU, the CPU can process other tasks during calculation. Before accessing the MAC result, either bit0 (RunDSP) of the DSP-CSR register can be polled to check for MAC program end, or the MAC interrupt can be used. Both methods have their advantages; the polling loop will have less latency between the MAC calculation end and the usage of the result, while the ISR might be better suited if other tasks have to be scheduled. It also is often possible to give the CPU another task which has at least the same execution time as the MAC program (e.g. UART communication), and when the CPU completes this task, the MAC unit also has already completed calculation and the result can be used immediately.

4 Usage of the MAC unit

This chapter explains how to setup and use the MAC unit with some examples. Additional examples can be found in the MB91265 MAC unit application note listed at the end of this document.

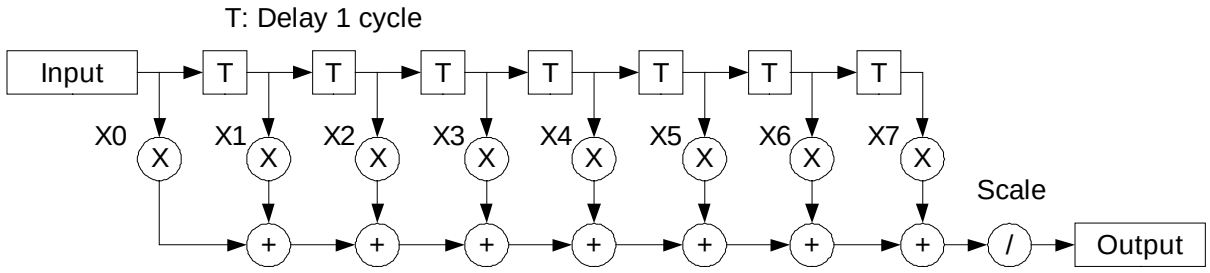
4.1 Using the MAC unit for FIR filters

4.1.1 Introduction

Multiplication-and-accumulate operations are typical for digital filters. Therefore, the functionality of the MAC unit enables high-speed filtering and other processing typical for DSP applications. Since the MAC unit operates completely independently of the CPU, it can process data separately and thereby reduce CPU load.

The typical structure of a digital FIR filter is shown below:

Figure 4. FIR filter



For every sampling point in time, the filter's output value is the sum of the *n* last input values; each one weighted with a coefficient X0…X7.

For example, in case of a moving average filter with equally weighted coefficients, the values X0…X7 are identical. The moving average filter is often used as simple low-pass filter, e.g. to reduce noise in analog measurements. Another common low-pass filter is the binomial filter, whose coefficients can be calculated with Pascal's triangle (e.g. 1, 7, 21, 35, 35, 21, 7, 1). The gain at 0 Hz (DC gain) is given by the sum of all coefficients, and since this is mostly set to unity for low-pass filters, the scaling stage divides the output by this value. For digital systems, usually a power of two is selected, so that the division can be performed by a simple bit shift. Every 'T' block in the above filter holds the delayed input signal from the last clock cycle, so the input values are 'daisy-chained' through the filter. The pulse response of a FIR filter of length N is max. N+1 samples long. The resulting output of the filter to an M-sample input signal therefore is M+N+1 samples long.

The MAC unit of the MB91470/480 Series can perform the hand-over of the input value from one filter stage to the next without CPU interaction. This is done by the DSP-LY and LY-DLY registers, when the LDLY and STLY bits in the MAC commands are set accordingly. Therefore, the complete processing of every input value, consisting of *(n+1)* multiplications, *n* additions, *n* register transfers and one division (or bit shift) is handled solely by the MAC unit. The CPU transfers the new input value to the MAC unit and starts it. 2n cycles later the result is ready-to-use; meanwhile other tasks can be performed.

The MAC unit also automatically performs the output scaling without additional cycles, simply by setting the appropriate format in the STR command. For example, to use the Q26 format, the X0…X7 for an 8-tap moving average filter would be set to 2^26 / 8 = 2^23 = 8.388.608 to achieve unity DC gain.

The following two scope figures (taken from the MB91265 MAC unit application note) show a 1 kHz rectangular signal, which is sampled by the ADC with 50 kHz, then passed through a 15-tap moving average filter (first figure) or a 15^{th} degree binomial filter (second figure). The filtered values are output by the PPG and filtered by a simple RC filter (and an additional averaging in the scope). The processing time for each sample is about 1.2 µs (@32MHz).

Figure 5. Point moving average filter (ch1: input, ch2: output)

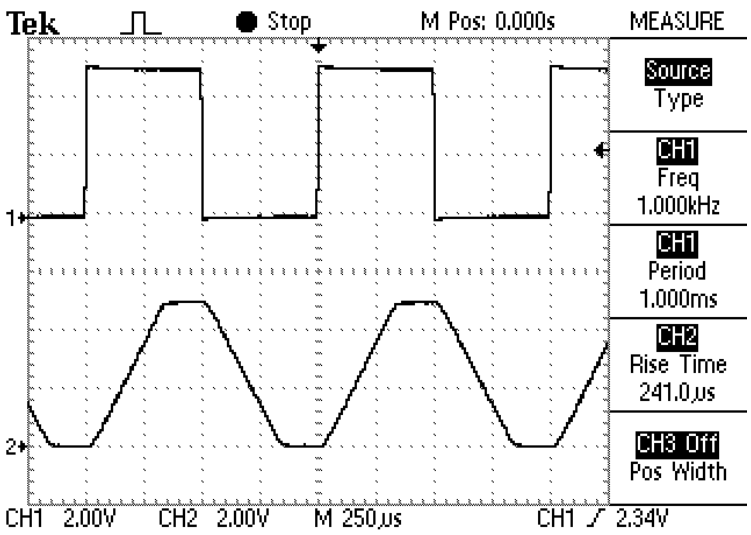
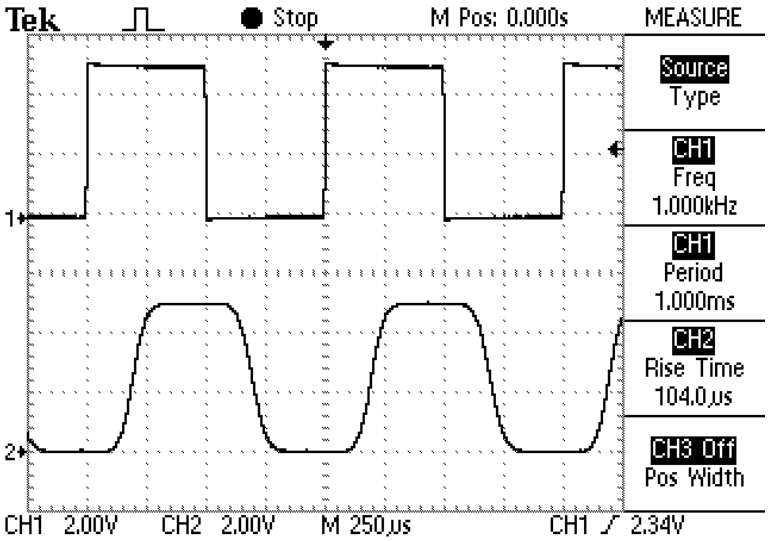


Figure 6. 15-degree binomial filter (ch1: input, ch2: output)



Please refer to the software example and the MB91265 MAC unit application note mentioned at the end of this document for more details about Filter implementation using the MAC unit.

4.2 Using the MAC unit for PID control loops

PID control loops are widely used in many applications. Especially in motor control, often fast response times are needed which demand fast processing and high sampling frequency. The MAC unit of the MB91470 series is well suited also for this task.

Without deepening in control theory, this chapter will only demonstrate how to implement a simple PID control algorithm on the MAC unit. Lots of information about control theory can be found in literature and on the web.

One standard equation for a (discrete) PID control loop is:

$$y_k = K_p \cdot e_k + K_i \cdot Ts \cdot \sum_{n=0}^{k} e_n + \frac{K_d}{Ts} \cdot (e_k - e_{k-1})$$

Kp is the proportional gain, Ki is the gain of the integral term, and Kd is the gain of the derivative term of the control loop.

After some transformations, the control loop can be described by another equation:

$$y_k = y_{k-1} + q_0 \cdot e_k + q_1 \cdot e_{k-1} + q_2 \cdot e_{k-2}$$

Wherein q0 = Kp + Ki + Kd, q1 = -Kp -2Kd, q2 = Kd

This form can easily be implemented using the MAC unit, using an initialization routine like the following:

```
...

#include "mac_91470.h"

long int * const ptr_speed_reg_out = YRAM_START; // Output of the speed loop is stored
here
long int * const ptr_speed_error = YRAM_START+1; // YRAM addr. 1 used as speed error
variable

void init_mac(void){
  unsigned short int * iptr;          // pointer used for IRAM access (16-bit words)
  long int * ptr;                     // pointer used for XRAM and YRAM access (32-
bit)

  while (DSP_ACTIVE) __wait_nop();    // ensure that no MAC calculation is ongoing

  RESET_uDSP_PC;                      // reset DSP Programm Counter

  ptr = XRAM_START;
  *(ptr++) = (1<<26);                 // constant scaling for old reg output memory in
Q26
  *(ptr++) = C0;                      // PID coefficients: C0=Kp+Ki*Ta+Kd/Ta
  *(ptr++) = C1;                      //                   C1=-Kp-2Kd/Ta
  *(ptr++) = C2;                      //                   C2=Kd/Ta

  ptr = YRAM_START;                   // initalize YRAM
  *(ptr++) = 0;                       // used for last control output memory
  *(ptr++) = 0;                       // stores current speed error
  *(ptr++) = 0;                       // stores error 1 step ago
  *(ptr++) = 0;                       // stores error 2 steps ago

  iptr = IRAM_START;                  // initialize uDSP programm memory
  *(iptr++) = DSP_MUL(NO_OPT,0,0);    // scale last control output value
  *(iptr++) = DSP_MAC(LDLY,1,1);      // + C0 * error(n); load error(n) to LY
  *(iptr++) = DSP_MAC(LDLY|STLY,2,2); // + C1 * error(n-1); load error(n-1) to LY;
                                      //               overwrite error(n-1) with
error(n);
```

```
  *(iptr++) = DSP_MAC(STLY,3,3);      // + C2 * error(n-2); overwrite error(n-2)
                                      // with error(n-1);
  *(iptr++) = DSP_STR(CLP|RND,Q26,YRAM,0); // store downscaled output value to YRAM
addr. 0;
                                      // it is re-used in every control loop cycle
  *(iptr++) = DSP_NOP(HLT,1);         // halt uDSP
  *(iptr++) = DSP_NOP(NO_OPT,1);
  *(iptr++) = DSP_NOP(NO_OPT,1);
}
```

Note that the above example makes use of the header file for the MAC unit, which is included in the SW examples mentioned at the end of this document.

The control loop itself now could look like this:

```
int reg_speed(int des_rpm, int act_rpm){

  while (DSP_ACTIVE) __wait_nop();              // ensure that no MAC calculation is
ongoing
                                                // (will normally not be necessary
here)
  *ptr_speed_error = (des_rpm - act_rpm);       // calculate actual speed error
  RESET_uDSP_PC;                                // set the uDSP Program Counter to
IRAM start
  START_uDSP;                                   // start uDSP (execute control loop)
  while (DSP_ACTIVE) __wait_nop();              // wait for calculation end
                              // also other operations could be done here
  if (*ptr_speed_reg_out < LOWER_MAX) * ptr_speed_reg_out = LOWER_MAX; // keep output
in range
  else if (*ptr_speed_reg_out > HIGHER_MAX) * ptr_speed_reg_out = HIGHER_MAX;
  return (*ptr_speed_reg_out);
}
```

In this example, the variables (control output and speed error) are placed directly in the Y-RAM of the MAC unit and accessed by pointers, so that no additional transfers are needed. But as mentioned earlier, this means that access only is possible when the MAC unit is stopped. Since the calculation time will usually be short compared to the sampling time of the control loop, this is no restriction. For the same reason, it often will not be necessary to check if the MAC unit is stopped before starting a new calculation.

The above control loop takes ten cycles for execution, which is far below 0.5µs, including overhead like the operations for the status pin used for time measurement. During the ten cycles which the MAC unit needs for the calculation, other tasks could be performed as well instead of the NOPs. As an example, some sensor information could be updated. But also without additional tweaking, the entire calculation time for the PID controller is strongly reduced compared to pure C code with the same functionality. As soon as multiple PID loops or additional functions are executed by the MAC unit, the amount of overhead becomes less and performance gain rises. One example could be vector control of three-phase motors, where the µDSP can handle all PID loops (speed + 2x current) as well as some additional filtering.

4.3 Using the MAC unit for function approximation

For some applications, math functions beyond basic arithmetic operations are required. Even though C compilers usually bring libraries with plenty of math functions with them, these are often standard c library floating point functions which are handled by fixed-point operations. Usually this will not perform as well as natively using fixed-point arithmetic. As an example, the arcus tangent function can be used by including math.h (double atan(double x)), as well as the atan2 function (double atan2(double y, double x)). The atan2 function also is known as the arcus tangent function for two input values and is commonly used to determine the angle when converting from two-axis Cartesian coordinates to polar coordinates.

Instead of using the original function, many functions can be approximated by other, simpler ones, mostly polynomials, for a certain input interval. The Taylor series is a very popular example for this. It uses an infinite power series to approximate the function, but usually only the first few terms are calculated to limit calculation time. The Taylor series for the arcus tangent function for |x| <= 1 is given by

$$\arctan x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \frac{1}{9}x^9 - \ldots$$

The values for |x| > 1 can be obtained by using the following formula:

$$\arctan x = sgn(x) - \arctan \frac{1}{x}$$

Now, this series shall be implemented using the uDSP.

First of all, the X-RAM addresses have to be preloaded once with the coefficients of the Taylor series in an appropriate fixed-point format, in a similar way as in the example above. Then, the I-RAM has to be programmed so that the uDSP will first calculate the required powers of the input variable x, then multiply and sum up every power with its corresponding coefficient, and finally store the result. In the following, a part of this initialization is shown. Please refer to the software example listed at the end of this document for the full c source code.

```
#include "mb91f479.h"
#define SCALED_ONE_Q30 (1<<30)              // unity in Q30 format

void dsp_init(void) {                       // set up uDSP for a taylow series
  long int * ptr;
  unsigned short int * iptr;

  while (DSP_ACTIVE) __wait_nop();          // wait in case there are calculations
ongoing

  RESET_uDSP_PC;                            // reset DSP Programm Counter
  ptr = XRAM_START;                         // set up the coefficients of the
polinomial
  *(ptr++) = 0;                             // XRAM0 = input value, also used for
temporary storage of x^2 and for output
  *(ptr++) = SCALED_ONE_Q30;                // XRAM1 = 1
  *(ptr++) = -(SCALED_ONE_Q30/3);           // XRAM2 = -1/3
  *(ptr++) = SCALED_ONE_Q30/5;              // XRAM3 = 1/5
  *(ptr++) = -(SCALED_ONE_Q30/7);           // XRAM4 = -1/7
. . .
  ptr = YRAM_START;
  *ptr = SCALED_ONE_Q30;                    // needed for initial calculations
  iptr = IRAM_START;                        // program DSP memory

  *(iptr++) = DSP_MUL(NO_OPT,0,0);          // x * 1
  *(iptr++) = DSP_STR(CLP|RND,Q30,YRAM,1); // YRAM[1] = x^1, needed to calculate x^2
  *(iptr++) = DSP_NOP(NO_OPT,5);            // wait for value to be written (pipeline)
    *(iptr++) = DSP_MUL(NO_OPT,0,1);          // x * x
  *(iptr++) = DSP_STR(CLP|RND,Q30,XRAM,0); // XRAM[0] = x^2, used for calculation of
x^3, x^5…
  *(iptr++) = DSP_NOP(NO_OPT,5);
  // now that x and x^2 are available, building up the needed powers of x is
straightforward:
  *(iptr++) = DSP_MUL(NO_OPT,0,1);          // x^2 * x
  *(iptr++) = DSP_STR(CLP|RND,Q30,YRAM,3); // YRAM[3] = x^3
  *(iptr++) = DSP_NOP(NO_OPT,5);
  *(iptr++) = DSP_MUL(NO_OPT,0,3);          // x^2 * x^3
  *(iptr++) = DSP_STR(CLP|RND,Q30,YRAM,5); // YRAM[5] = x^5
  *(iptr++) = DSP_NOP(NO_OPT,5);
. . .

// now, every power of x has to be multiplied with the corresponding coefficient and
summed up
```

```
  *(iptr++) = DSP_MUL(NO_OPT,1,1);          //    1 * x
  *(iptr++) = DSP_MAC(NO_OPT,2,3);          // -1/3 * x^3
  *(iptr++) = DSP_MAC(NO_OPT,3,5);          // +1/5 * x^5
  *(iptr++) = DSP_MAC(NO_OPT,4,7);          // -1/7 * x^7
. . .
  *(iptr++) = DSP_STR(CLP|RND,Q30,XRAM,0);  // store result in XRAM[0]
  *(iptr++) = DSP_NOP(HLT,1);               // stop uDSP
```

The above example uses XRAM1 to store x^1, XRAM3 to store x^3 etc. This was arbitrarily chosen to improve readability.

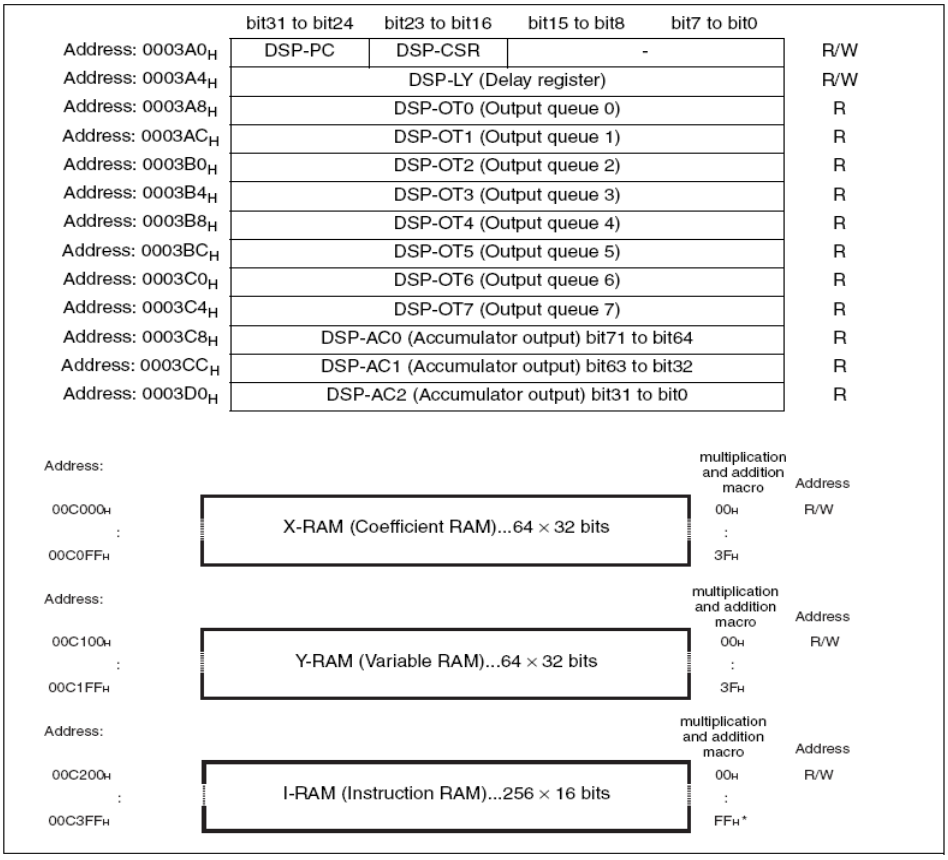
Please refer to the mentioned software example for usage of the above functions. It also contains wrapper functions, which allows using floating-point input and output values for easy replacement of the library functions.

In the software example, the approximation is calculated up to the 21st term of the Taylor series. This gives very good result with errors far below one percent, except a small peak around x=1 (~2.8%). Also, the software example shows how to extend the above function to cover the atan2 function with two input arguments.

5 Appendix

5.1 Register list and memory map of the MAC unit

Figure 7. Register list and memory map



5.2 Related documents

hm91470_480-cm71-10134-4e.pdf	Hardware manual of the MB91470/480 series
mcu-an-300030-e-mb91265_mac_unit	Application note describing the 16-bit MAC unit included in the MB91265 Series.

5.3 Related software examples

91470_mac_filter-v11	FIR filter with the µDSP
91470_mac_matrix-v10	Matrix multiplication with the µDSP
91470_mac_atan_taylor-v10	Atan() / atan2() function approx. using the µDSP

5.4 Glossary

ADC	Analog-Digital-Converter
CPU	Central Processing Unit
(µ)DSP	(micro) Digital Signal Processor, Digital Signal Processing
FIR	Finite Impulse Response (Filter)
IIR	Infinite Impulse Response (Filter)
IRQ	Interrupt ReQuest
ISR	Interrupt Service Routine
MAC	Multiply-and-ACumulate
MCU	Micro-Controller Unit
PID	Proportional-Integral-Derivative (Control loop)

Document History

Document Title: AN205433 - FR, MB91470/480, 32-BIT MAC unit

Document Number:002-05433

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	WOFR	09/22/2008	CHa First draft, based on MB91265 MAC appnote
*A	5092220	WOFR	01/19/2016	Converted Spansion Application Note “MCU-AN-300107-E-V10” to Cypress format

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless
Spansion Products	cypress.com/spansionproducts

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums |Blogs | Video |Training

Technical Support

cypress.com/go/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2008-2016. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.
Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
Use may be limited by and subject to the applicable Cypress software license agreement.