Application Note, V1.1, October 2007

AP16113

XC2000/XE166/XC166 Families

DSP Optimization Guide for XC2000, XE166 and XC166 Microcontroller Families with MAC Unit

Microcontrollers

Never stop thinking

Edition 2007-10

Published by
Infineon Technologies AG
81726 Münich, Germany

© 2007 Infineon Technologies AG
All Rights Reserved.

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact the nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

AP16113

Revision History:	2007-10	V1.1
Previous Version:	none	

Page	Subjects (major changes since last revision)

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

Table of Contents **Page**

1 Introduction 5

2 The MAC Unit 5
2.1 Architecture of the MAC 5
2.2 Data Addressing Mode of the MAC Unit 7
2.3 MAC Instruction Set 12
2.4 Usage of the MAC Unit 14

3 Approaches of DSP Function Development 15
3.1 DSP Functions in Full Assembly 15
3.2 DSP Functions in Mixed C and Assembly 17
3.3 DSP Functions from Intrinsic Functions 20
3.4 DSP Function from Library 21
3.5 DSP Functions in C 21
3.6 Summary 23

4 DSP Optimization Techniques 24
4.1 Chip-independent Techniques 24
4.1.1 Packet Processing 24
4.1.2 Data Memory Interleaving 26
4.1.3 Loop Unrolling 26
4.1.4 Software Pipelining 28
4.2 Chip-dependent Techniques 28
4.2.1 Data Dependencies Removal 29
4.2.2 Memory Bandwidth Conflicts Removal 30
4.2.3 Instruction Re-ordering 31
4.2.4 Using Flash Waitstates 32
4.2.5 Using PSRAM 32

5 Conclusion 33

6 Reference 33

1 Introduction

XC2000 and XE166 are new Infineon microcontroller families based on the same silicon. In comparison with the XC166 family, both microcontrollers have a larger flash size, double the system frequency and more powerful peripherals. The XC2000, XE166 and XC166 all have the same C166S V2 CPU core. The C166S V2 core is a well-known 16 bit core, and has two instruction sets. A normal instruction set and a MAC (multiply and accumulate) instruction set. The normal instruction set is used for the general control tasks, and the MAC instruction set is dedicated for DSP and 32 bit mathematical operations, such as filter, FFT. The MAC unit enhances strongly the DSP performance and 32 bit data processing capability of XC2000 and XC166 microcontroller. With help of Tasking Viper compiler the performance of XC2000 microcontroller family is comparable with a 32 bit microcontroller.

The usage of the MAC instruction set is dependent on compilers. There are three commonly used compilers for the XC2000, XE166 and XC166 microcontrollers. The classic Tasking compiler, the Keil compiler and the Tasking Viper compiler. Currently, only the Tasking Viper compiler contains a MAC code generator that supports the MAC instruction set at C level, which means that with the Viper compiler we can get mixed assembly code with normal and MAC instructions after the C programs are compiled. Furthermore, the Viper compiler processes all possible data types including 64 bit long long data. Currently the classic Tasking and Keil compilers have no MAC code generator, and do not support MAC the instruction set at C level. Both compilers support the MAC unit only at assembly level. This means that we must write the program in assembly code if we want to use the MAC with the classic Tasking and Keil compilers. In the past, to resolve this problem Infineon provided a DSP library implemented with MAC instructions.

No matter with which programming languages (C or assembly) the DSP program is written, optimization is always an important issue that needs to be considered during DSP programming. In this application note we describe some useful DSP optimization techniques for XC2000, XE166 and XC166. These can be used as guide for developing DSP applications.

2 The MAC Unit

The MAC unit is a built-in unit in the CPU core shown in Figure 1, which means that the MAC can be considered as an additional arithmetic unit dedicated for DSP and 32 bit calculation. This is one of the important reasons, why the XC2000 microcontroller family is called a 16-/32 bit microcontroller. In the execution stage of the instruction processing pipeline the instruction will be put into the ALU or MAC unit to be executed according to which instruction set (normal or MAC) the code comes from. If the instruction comes from normal instruction set, the code execution will be done in the ALU unit. If the instruction is from the MAC instruction set with CoXXX prefix, this instruction will be processed in the MAC unit.

2.1 Architecture of the MAC

The MAC is designed to enhance DSP performance on microcontrollers. Therefore, the MAC unit has similar architecture to the common digital signal processor. Typically a MAC unit should provide single instruction-cycle, non-pipelined 32 bit additions, 32 bit subtraction, right and left shifts, 16 bit by 16 bit multiplication, and multiplication with cumulative subtraction/ addition. Figure 2 shows schematically the major components of the MAC unit on XC2000, XE166 and XC166 microcontrollers:

- 16 bit by 16 bit signed/unsigned multiplier with signed result
- Concatenation Unit
- Scaler (one-bit left shifter) for fractional computing
- 40-bit Adder/Subtracter
- 40-bit Signed/Accumulator
- Data Limiter
- Accumulator Shifter

· Repeat Counter

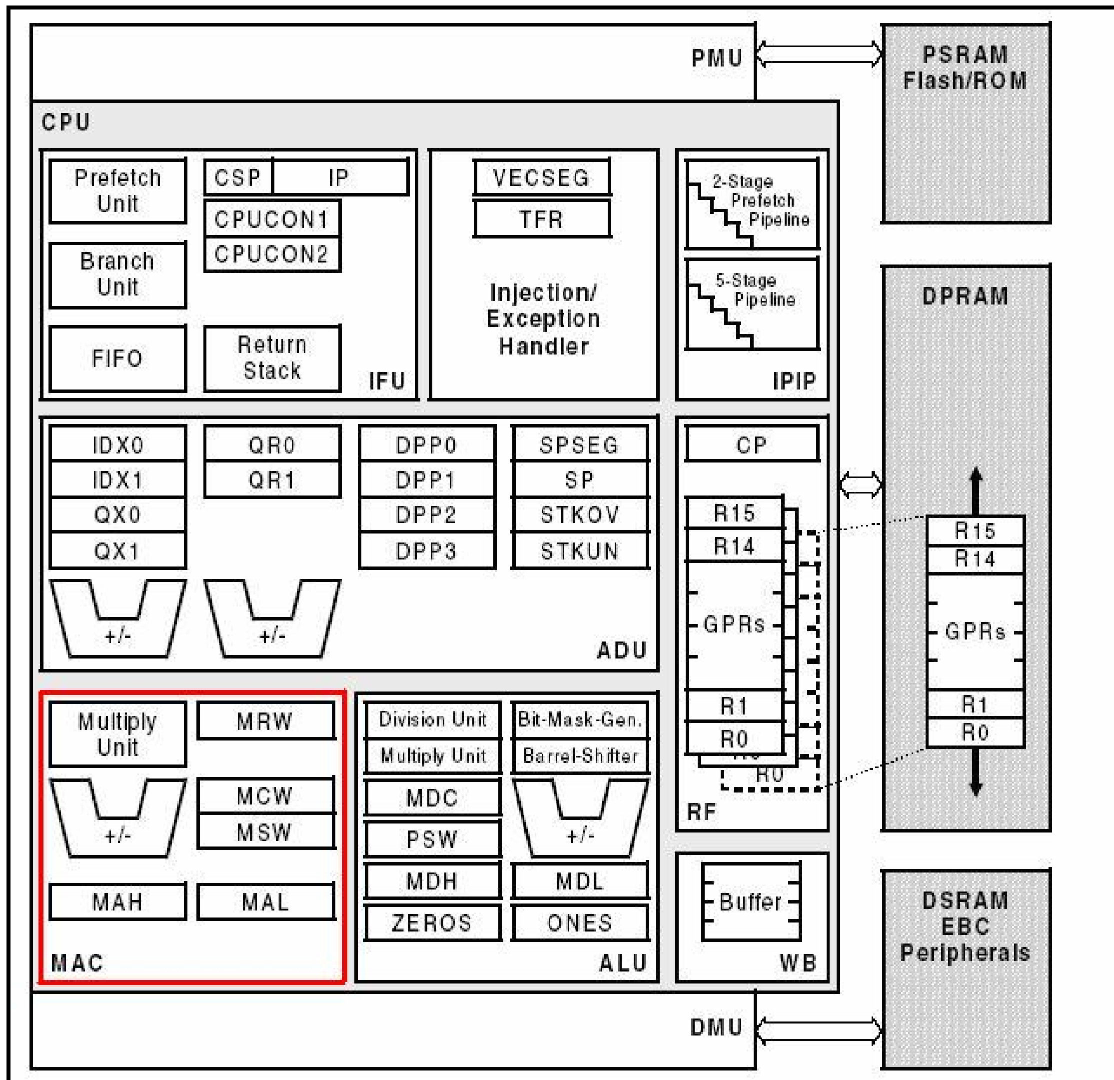

Figure 1 : CPU block diagram of XC2000, XE166 and XC166

The working register of the MAC Unit is a 40-bit wide accumulator register. A set of consistent flags can be automatically updated in the MAC Status Word (MSW) register after each MAC operation. These flags allow branching on specific conditions. Unlike the PSW flags, these flags are not preserved automatically by the CPU upon entry into an interrupter trap routine. So, all dedicated MAC registers must be saved on the stack which in the case the MAC unit is shared between different tasks and interrupts. For a detailed description of each component function please refer to the User Manuals of the XC2000 and XC166 microcontrollers.

The MAC unit on the XC2000, XE166 and XC166 microcontroller has a Harvard architecture implementation, which means that every CPU-cycle allows one opcode fetch, two operand reads and one optional operand write. The XC2000, XE166 and XC166 microcontrollers are designed with Von Neumann architecture, where the memory space is unified and the code and data share the same linear addressing space. In order to make it possible to fetch the code and data in one cycle a specific addressing mode is used in the MAC unit. When the MAC instruction needs two operands, one of them must be located in DPRAM.

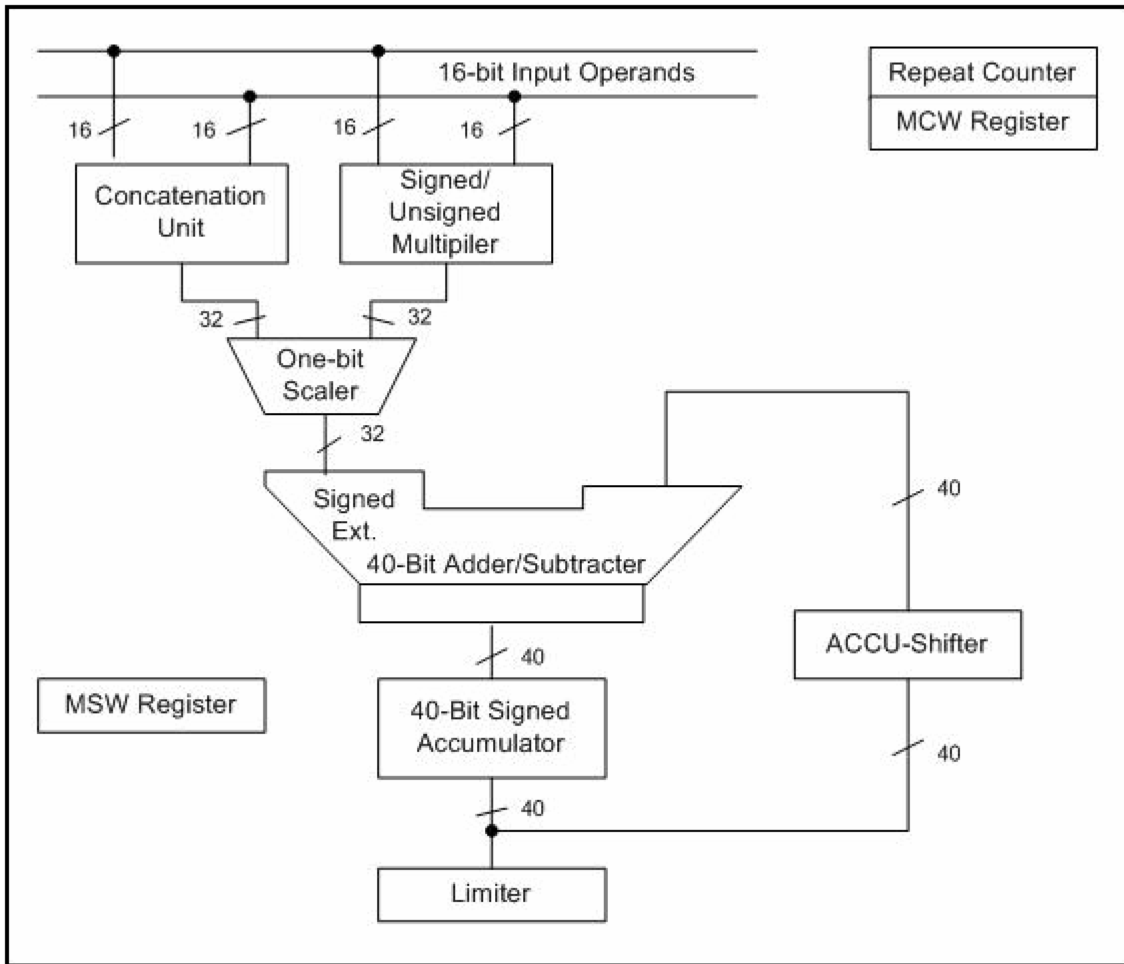

Figure 2: Functional MAC Unit Block Diagram

2.2 Data Addressing Mode of the MAC Unit

The Address Data Unit (ADU) of the C166S V2 CPU contains two independent arithmetic units to generate, calculate, and update addresses for data accesses. The ADU performs the following major tasks:

- Standard Address Generation (Standard Address Generation Unit)
- MAC Address Generation (MAC Address Unit)
- Data Paging (Standard Address Unit)
- Stack Handling (Standard Address Unit)

The Standard Address Unit supports linear arithmetic for the indirect addressing modes and also generates the address in case of all other short and long addressing modes. The MAC Address Generation Unit contains an additional set of address pointers and offset registers which are used in conjunction with the CoXXX instructions only.

The MAC data addressing mode

- provides two new pointers IDX0 and IDX1
- provides two pairs of offset registers QR0/QR1 and QX0/QX1, QR0/QR1 used with GPR pointer, QX0/QX1 used with IDX pointer together
- makes double indirect addressing possible: GPR pointers access to the entire memory space, IDXi limited to Dual-Port RAM (DPRAM) (0xF600h - 0xFD00h, 2Kbytes) except for the CoMOV instruction

Table 1 lists the various combinations of pointer post-modification for pointer Rw_n and IDXi.

Symbol	Mnemonic	Address Pointer Operation
[IDXi⊗] stands for	[IDXi]	(IDXi) <- (IDXi)
	[IDXi+]	(IDXi) <- (IDXi)+2 (i=0,1)
	[IDXi-]	(IDXi) <- (IDXi)-2 (i=0,1)
	[IDXi+QXj]	(IDXi) <- (IDXi)+(QXj) (i,j=0,1)
	[IDXi-QXj]	(IDXi) <- (IDXi)-(QXj) (i,j=0,1)
[RWn⊗] stands for	[Rwn]	(Rwn) <- (Rwn)
	[Rwn+]	(Rwn) <- (Rwn)+2 (n=0-15)
	[Rwn-]	(Rwn) <- (Rwn)-2 (n=0-15)
	[Rwn+QRj]	(Rwn) <- (Rwn)+QRj (n=0-15, j=0,1)
	[Rwn-QRj]	(Rwn) <- (Rwn)-QRj (n=0-15, j=0,1)

Table 1: Pointer post-modification for Rwn and IDXi

The address pointers IDXi can be used for arithmetic operations as well as for the special CoMOV instruction. But, the generation of the 24-bit memory address is different. In case of arithmetic CoXXX operations, the IDXi pointers are automatically zero extended to a 24-bit memory address in order to point the internal DPRAM area. The leading four bits of the IDX are not taken into account. Figure 3 shows the addressing mode with arithmetic CoXXX instructions.

For CoMOV operation, the IDXi pointers are concatenated with the Data Page Pointers, just like normal GPR-Pointers. In this case, the IDX pointer can address the entire C166S V2 memory area without any restrictions. Figure 4 shows CoMOV operation.

The instruction CoSTORE transfers a value from a MAC register to any location in memory. This instruction uses a specific addressing mode for the MAC registers, called **CoReg**. The following table gives the 5-bit addresses of the MAC registers corresponding to this CoReg addressing mode.

Register	Description	Coding of wwww:w bits [31:27]
MSW	MAC-Unit Status Word	00000
MAH	MAC-Unit Accumulator High	00001
MAS	“limited” MAH	00010
MAL	MAC-Unit Accumulator Low	00100
MCW	MAC-Unit Control Word	00101
MRW	MAC-Unit Repeat Word	00110

Table 2: Coding of the CoREG Addressing Mode

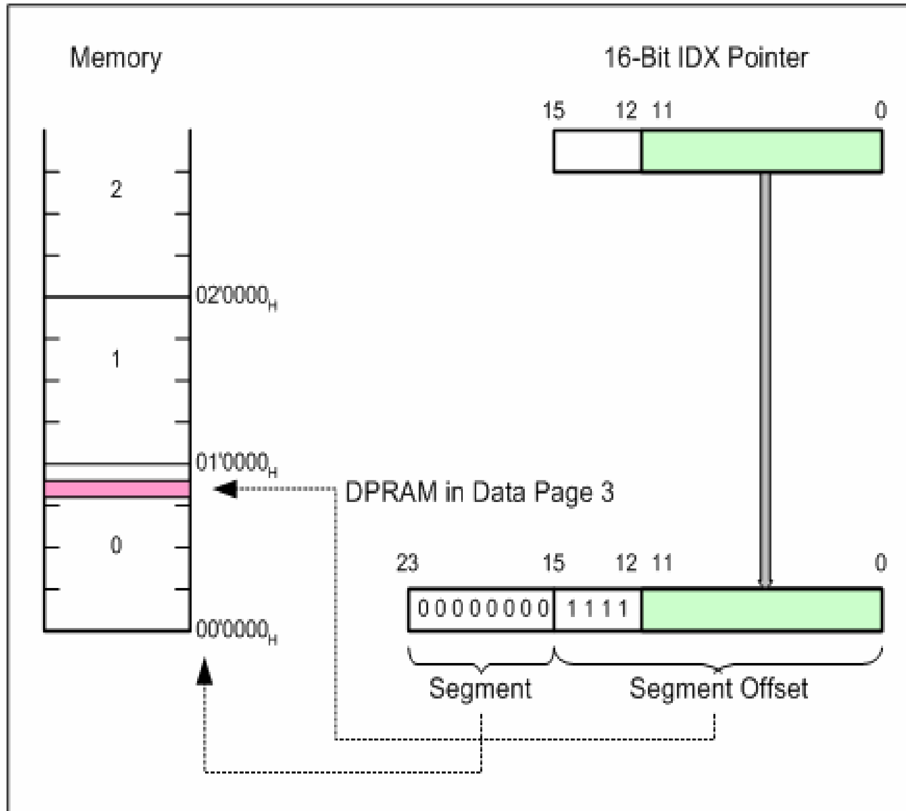

Figure 3: Arithmetic MAC Operation with addressing via IDX pointers

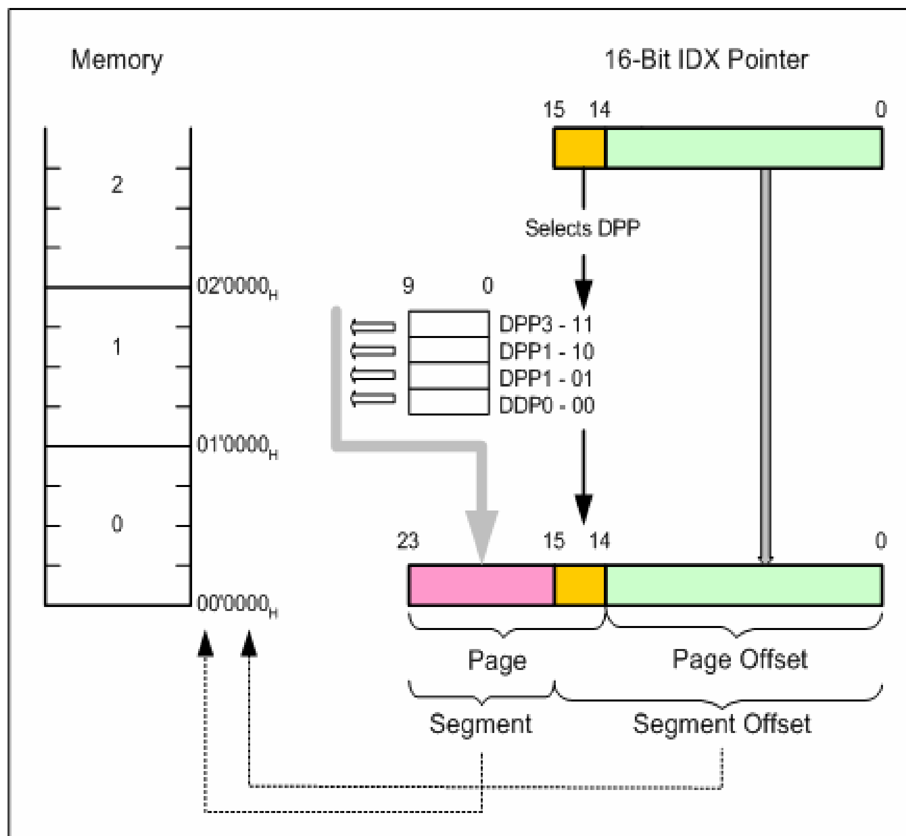

Figure 4: CoMOV Operation with addressing via IDX pointers

The following two examples show how the MAC data addressing modes work in the program execution.

Example 1: CoMAC [IDX0+],[R6+]

In Figure 5, the data vector X is stored in DPRAM and accessed by pointer IDX0. The vector h is located in data memory SRAM and pointed with GPR R6. After operation registers IDX0 and R6 are increased by 2 and point to another data elements. In one cycle following operations will be done with instruction CoMAC [IDX0+], [R6+]:

- Address operations:
    - address 1 = (IDX0)
    - address 2 = (R6)
    - (IDX0)new = (IDX0) + 2
    - (R6)new = (R6) + 2
- Data Operands
    - op1 = ((IDX0))
    - op2 = ((R6))
- Multiplication: op1 x op2
- Addition: (op1 x op2) + (ACC)

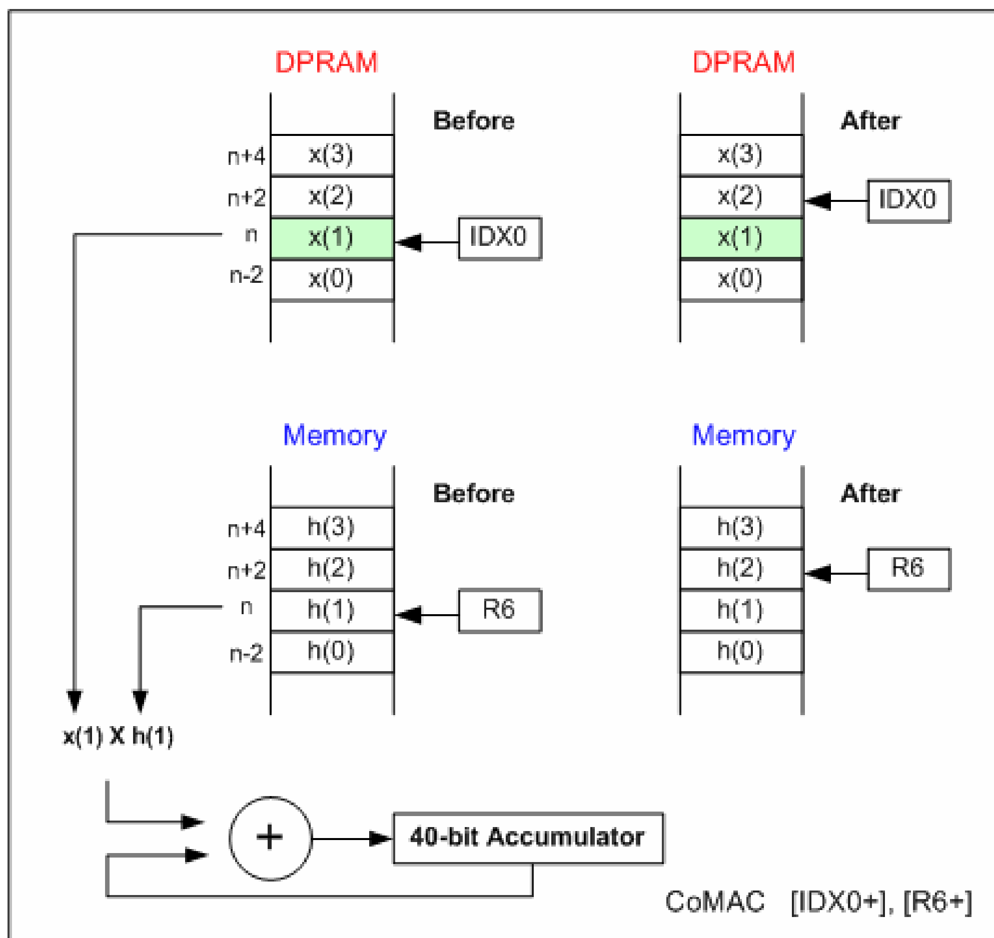

Figure 5: Addressing mode for CoMAC [IDX0+],[R6+]

Example 2: CoMACM [IDX0+], [R6+]

The MAC instruction CoMACM is a special instruction and can be used in cycle buffer operation. In addition to multiplication and addition the CoMACM can do at same time a parallel data moving operation. All operations just need one cycle as depicted in Figure 6. By executing CoMACM instruction the following operations are done:

- Address operation
    - address 1 = (IDX0)
    - address 2 = (R6)
    - intermediate address =(IDX0)-2
    - (IDX0)new = (IDX0) + 2
    - (R6)new = (R6)+2
- Read Operands
    - op1 = ((IDX0))
    - op2 = ((R6))
- Write Operand op1: (intermediate address) = op1
- Multiplication: op1 x op2
- Addition: (op1 x op2) + (ACC)

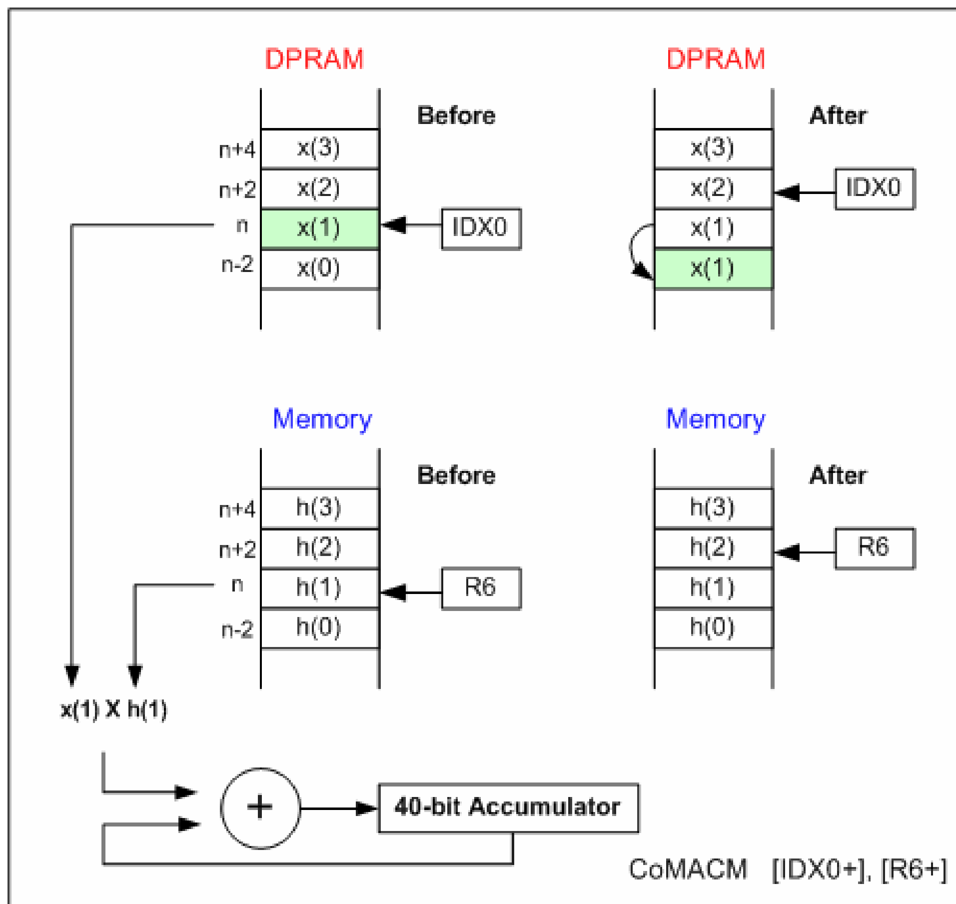

Figure 6: Addressing Model for CoMACM [IDX0+], [R6+] with parallel data moving

Note that the address for the parallel data moving is dependent on the IDX address post-modification used in the CoMACM instruction. In above example IDX0 is post-increased with 2 after operation. So, the data will be moved to pointer (IDX0-2). In following table we give the address calculation for the parallel data moving operation used in the CoMACM instruction based on IDX post-modification:

Instructions	**Write address for parallel data moving**
CoMACM [IDXi+],…	<IDXi-2>
CoMACM [IDXi-],…	<IDXi+2>
CoMACM [IDXi+QXj],…	<IDXi-QXj>
CoMACM [IDXi-QXj],…	<IDXi+QXj>

Table 3: Address calculation for parallel data moving in CoMACM instruction

2.3 MAC Instruction Set

The MAC Unit instruction set contains the following groups of instructions:

- multiply and multiply-accumulate instructions
- 32-bit arithmetic instructions
- shift instructions
- compare instructions
- transfer instructions

All MAC Unit instructions are 32-bit instructions with 4 bytes used to encode each instruction. The following table is a summarization of MAC instructions.

Table 4: Overview of MAC instruction set

Mnemonic	Address Modes	Rep	Mnemonic	Address Modes	Rep
CoMUL	Rwn,Rwm	Yes	CoMACM	[IDXi⊗],[Rwm⊗]	Yes
CoMULu	[IDXi⊗],[Rwm⊗]		CoMACMu		
CoMULus	Rwn,[Rwm⊗]		CoMACMus		
CoMULsu			CoMACMsu		
CoMUL-			CoMACM-		
CoMULu-			CoMACMu-		
CoMULus-			CoMACMus-		
CoMULsu-			CoMACMsu-		
CoMUL+rnd			CoMACM+rnd		
CoMULu+rnd			CoMACMu+rnd		
C0MULus+rnd			CoMACMus+rnd		
CoMULsu+rnd			CoMACMsu+rnd		
CoMAC	Rwn,Rwm	Yes	CoMACMR		
CoMACu	[IDXi⊗],[Rwm⊗]		CoMACMRu		
CoMACus	Rwn,[Rwm⊗]		CoMACMRus		
CoMACsu			CoMACMRsu		
CoMAC-			CoMACMR+rnd		
CoMACu-			CoMACMRu+rnd		
CoMACus-			CoMACMRus+rnd		
CoMACsu-			CoMACMRsu+rnd		
CoMAC+rnd			CoADD	Rwn,Rwm	Yes
CoMACu+rnd			CoADD2	[IDXi⊗],[Rwm⊗]	
CoMACus+rnd			CoSUB	Rwn,[Rwm⊗]	
CoMACsu+rnd			CoSUB2		
CoMACR			CoSUBR		
CoMACRu			CoSUB2R		
CoMACRus			CoMAX		
CoMACRsu			CoMIN		
CoMACR+rnd			CoLOAD	Rwn,Rwm	Yes
CoMACRu+rnd			CoLOAD-	[IDXi⊗],[Rwm⊗]	
CoMACRus+rnd			CoLOAD2	Rwn,[Rwm⊗]	
CoMACRsu+rnd			CoLOAD2-		
CoNOP	[Rwn⊗] [IDXi⊗],[Rwm⊗]	Yes	CoCMP		
			CoSHL	#data4 Rwm [Rwm⊗]	Yes
			CoSHR		
CoNEG	-	Yes	CoASHR		
CoNEG+rnd			CoASHR+rnd		
CoRND			CoABS	Rwn,Rwm [IDXi⊗],[Rwm⊗] Rwm,[Rwm⊗]	Yes
CoSTORE	Rwn,CoReg [Rwn⊗],CoReg	Yes			
CoMOV	[IDXi⊗],[Rwm⊗]	Yes			

2.4 Usage of the MAC Unit

The general usage of the MAC unit for DSP development will be described in the following sections in detail. Here we want to just provide two clues from the point of software development of view.

- Protection of MAC registers
- The MAC working registers, such as MSW, MCW, DX0, DX1 an so on, are not automatically saved into stack if an interrupt occurs. So, the programmer must save the MAC working registers at the beginning of the program if the program could be interrupted by other programs or tasks that have higher priority and also use the same MAC registers.
- Flag USRx
- USR0 and USR1 are two general purpose flags that can be automatically set by the MAC operation –USRx CoXXX combined with the MAC Repeat Counter (MRW) register. Using USRx flags and the MRW register we can repeat a MAC instruction many times. For example:

```
  MOV MRW, #19                ;Pre-load loop counter
loop1:
  -USR1 CoMACM [IDX0+],[R1+]  ;Calculate and decrement MSW
  ADD R2,#2h
  JMPA cc_nusr1, loop1        ;Repeat loop until USR1 is set
```

The MAC operation –USRx CoXXX decrements the value in the MRW register. If MRW=0, the flag USR1 is set, and the loop ends. Note: the flag bit USRx can be automatically cleared by the first –USRx CoXXX operation, and kept until last operation in the loop. So, it is not necessary to clear USRx bit using instructions, such as bclr USRx after the loop. All CoXXX instruction can be executed repeatedly using the above code.

Here is another example to showing how to use USRx flag to build a loop with MAC instructions:

```
  MOV     MRW,#20
  EXTR    #1              ;IDX0 is ESFR
  MOV     IDX0,R12
Loop:
  CoLOAD  [IDX0+],[R13]
  CoADD   [IDX0],[R13]
  CoSHR   #2
  -usr1   CoSTORE [R12],MAL  ;decrement MRW register
  ADD     R12,#2
  JMPA    cc_nusr1,Loop     ;if MRW=0, set USR1, end of loop
```

Note: USR0 could be used by compiler. So, it is at best not to use USR0 flag in programming.

3 Approaches of DSP Function Development

There are different ways to develop DSP functions with the MAC unit in XC2000, XE166 and XC166 microcontroller families according to the algorithms, the requirements and the development tool chains available. Basically, independent from tool chains, we have the following approaches to develop DSP functions using MAC unit: in full assembly, in mixed C and Assembly, using MAC intrinsic functions provided by tools and using the DSP library. Each approach has advantages and disadvantages. In the following we describe each approach with examples, separately.

3.1 DSP Functions in Full Assembly

The common programming language for implementation of DSP algorithms is C. However, the C compiler must support the MAC unit. We have said, the MAC unit has own instruction set that is different from normal instruction set for microcontrollers. It is not easy for tool vendor to develop a C compiler supporting both instruction sets. Currently only Tasking Viper compiler supports the MAC instruction set. This means that we must use the Viper compiler if we want to write DSP functions for the MAC unit in full C. If you use other compilers, such as classic Tasking and Keil compilers you need to write the program for the MAC unit in assembly.

The following example shows how to port DSP algorithm at assembly level using MAC unit. The used tool chain is classic Tasking. The argument transform should conform to the tool’s convention. The Classic Tasking tool chain defines the parameter transform as follows: the first four arguments of the function will be transferred through general registers R12 to R15, and the rest will be transferred through the stack. So, the first part of a C-callable assembly DSP function consists of always the parameter reads. If we call the library function from an assembly program the convention must be also conformed to.

Theoretically there are 16 general registers on XC2000, XE166 and XC166 microcontrollers that can be used for programming. But R0 is used by classic Tasking tools as stack pointer. R12 to R15 are used to transfer parameters. If we want to use these registers in the interrupt service routine we need to store the states before and restore them after leaving the subroutine. In the following we take the Fir filter function implemented in full assembly as an example:

```
$EXTEND
$CASE

NAME   Fir_16

;============ Define registers ===================================

h_ptr       LIT    'R12'  ; pointer of coefficient vector h
X_in        LIT    'R13'  ; pointer of new input sample
N_h         LIT    'R14'  ; filter order
buffer_ptr  LIT    'R15'  ; pointer of delay buffer

;-------------- Section for Code Segment Declaration --------------

CODE_FILTER SECTION CODE WORD GLOBAL

GLOBAL _Fir_16

;====================Procedure Declaration ========================
_Fir_16 PROC FAR

  ASSUME DPP3:SYSTEM
;MAC registers initialization
  MOV    MCW,#0600h     ; MP=1, MS=1
  SUB    N_h,#2h        ;(R14)=N_h-2
```

```
   MOV     MRW,N_h        ;(MRW)=N_h-2, repeat counter

;ESFR register initialization
   ADD     N_h,#1           ;(R14)=N_h-1
   SHL     N_h,#1           ;(R14)=2*(N_h-1)
   EXTR #3                  ; Next 2 instructions use ESFR space
   MOV     IDX0,buffer_ptr  ;(IDX0)=DPRAM_add
   MOV     QX0,N_h          ;(QX0)=2*(N_h-1)
   MOV     QR0,N_h

;Read the new input sample and move it
;at x(n-1) address overwriting x(n-1)
   CoMOV   [IDX0],[X_in]        ;x(n-1) <- IN
   CoNOP   [IDX0+QX0],[h_ptr+QR0]      ;return the pointers to beginning
;FIR: first multiplication, h(N_h-1)*x((n-N_h+1)
 CoMUL [IDX0-],[h_ptr-]                ;(ACC)=h(N_h-1)*x(n-N_h+1)<<1
                                       ;(IDX0)=(IDX0)-2
                                       ;(R12)=(R12)-2
;FIR loop: repeat N_h-1 times the same MAC instruction
loop1:                              ;i=(N_h-2):(-1):0
-USR1   CoMACM [IDX0-],[h_ptr-]     ;(ACC)=(ACC)+h(i)*x(n-i)<<1
   JMPA cc_nusr1,loop1

;return the pointers
   CoNOP   [IDX0+],[h_ptr+]

;Rounding
   CoRND

;write the 16-bit filter output y(n) into registers R4
   CoSTORER4,MAH       ;(R4)=(MAH)

   RET
_Fir_16ENDP

CODE_FILTER   ENDS

   REGDEF R0-R15
   END
```

In the above example, four parameters need to be passed to the function; the pointer to coefficient vector, the pointer to the input sample, the filter size and the pointer of buffer. As usual with classic Tasking tool chains, the registers R12, R13, R14 and R15 are used to transfer parameters. At the beginning of the function the size of the filter is read and used for initialization of MAC repeat counter, the pointer register IDX0 and the offset register QX0. After reading the current input sample the routine executes the first multiplication of the Fir filter. The rest N-1 multiplications will be executed by repeating N-1 times the same CoMACM instruction. The filter result will be rounded and then outputted through register R4. To distinguish the normal instruction set, the MAC instruction uses the prefix CoXXX. Each CoXXX instruction costs only one CPU cycle.
The cycles needed to execute the above example read N+12, where N is the filter size. If we implement the same Fir filter using normal instructions we need at lest 4N+16 cycles. So, implementation of the DSP algorithm using the MAC unit is much more efficient than using the ALU of the microcontroller.

3.2 DSP Functions in Mixed C and Assembly

By means of inline assembly functions we have opportunity to develop the DSP routine with assembly in the C function environment. The inline assembly function has similar definitions to the C function. The difference is that the contents of inline assembly function are written in assembly while the contents in the C function are written in C. So, we call this approach mixed C and Assembly. This approach is especially suitable for short DSP functions. By inline assembly function the pseudo register allocation is done by the compiler depending on the available registers. The code is independent of memory models.

An inline assembly function is actually a C function with the function body consisting of assembly instructions. How to use assembly instructions in C functions depends on the compiler. Each compiler has its own method. In the following sections we show the different definitions of the inline assembly function for classic Tasking, Viper and Keil compilers.

Note: There are differences between an inline assembly function and an inline function. An inline function means that this function will be copied to the place where it is called. Here we define an inline assembly function as a normal C function with assembly instruction body. A function is identified as an inline function if the key word *_inline* (for classic Tasking and Keil compiler) or *inline* is attached before the function definition. An inline assembly function can be inlined or noinlined depending on if there is the key word _inline in the function definition.

The classic Tasking compiler uses pragma asm and pragma endasm to insert the assembly instructions in the C function. Here we take the Fir filter as an example to show how to write an inline assembly DSP function using the classic Tasking compiler.

```
short Fir(short *h, short *in, short N, short *D_buffer)
{
      short y;

#pragma  asm(@1=h, @2=in, @3=N, @4=D_buffer)
;MAC registers initialization
   MOV     MCW,#0600h     ; MP=1, MS=1
   SUB     @3,#2h         ;(R14)=N_h-2
   MOV     MRW,@3         ;(MRW)=N_h-2, repeat counter

;ESFR register initialization
   ADD     @3,#1            ;(R14)=N_h-1
   SHL     @3,#1            ;(R14)=2*(N_h-1)
   EXTR #3                  ; Next 2 instructions use ESFR space
   MOV     IDX0,@4          ;(IDX0)=DPRAM_add
   MOV     QX0,@3           ;(QX0)=2*(N_h-1)
   MOV     QR0,@3

;Read the new input sample and move it
;at x(n-1) address overwriting x(n-1)
   CoMOV   [IDX0],[@2]               ;x(n-1)  <- IN
   CoNOP   [IDX0+QX0],[@1+QR0]      ;return the pointers to beginning
;FIR: first multiplication, h(N_h-1)*x((n-N_h+1)
 CoMUL  [IDX0-],[@1-]                ;(ACC)=h(N_h-1)*x(n-N_h+1)<<1
                                     ;(IDX0)=(IDX0)-2
                                     ;(R12)=(R12)-2
;FIR loop: repeat N_h-1 times the same MAC instruction
loop1:                               ;i=(N_h-2):(-1):0
-USR1   CoMACM [IDX0-],[@1-]         ;(ACC)=(ACC)+h(i)*x(n-i)<<1
   JMPA cc_nusr1,loop1

;return the pointers
   CoNOP   [IDX0+],[@1+]

;Rounding
   CoRND
```

```
;write the 16-bit filter output y(n) into registers R4
  CoSTORE   @3,MAH      ;(R4)=(MAH)

#pragma endasm (y = @3)

;return the 16-bit filter output y
  return y;
}
```

In the above function the pragma environment indicates that the program body is written in assembly instructions. Unlike the implementation in full assembly the parameter passing will be done by the compiler automatically, so the convention for parameter passing doesn't need to be conformed to. Here the pseudo registers @1, @2 and @3 are used in program to present the pointer of the coefficient vector, the pointer of the input sample and the size of the filter, respectively. Which real registers will replace the three pseudo registers is determined by compiler according to the available registers. The filter result will be returned to calling function through the variable y. To call the inline assembly function we can write a main function as follows:

```
#define N  5
_near short D_buffer[N_h] _at(0x0f600) ;variable in located in DPRAM
short in = 0;

void main(void)
{
   short h[N];
   short y;

   y = Fir(h,&in,N, D_buffer);

}
```

Note: if the above inline assembly function is defines as

```
      _inline short Fir(short *h, short *in, short N)
```

Then we get an inlined inline assembly function.

Viper compiler uses the key word __asm( ) to insert the assembly instructions into C function body. For Viper compiler the Fir inline assembly function looks like:

```
short Fir(short *h, short *in, short N, short *D_buffer)
{
      short y;

__asm (  ;0%=h, 1%=in, 2%=N, 3%=D_buffer, 4%=y

;MAC registers initialization
  "MOV   MCW,#0600h\n"    ; MP=1, MS=1
  "SUB   2%,#2h\n"        ;(R14)=N_h-2
  "MOV   MRW,2%\n"        ;(MRW)=N_h-2, repeat counter

;ESFR register initialization
  "ADD   2%,#1\n"           ;(R14)=N_h-1
  "SHL   2%,#1\n"           ;(R14)=2*(N_h-1)
  "EXTR  #3\n"              ; Next 2 instructions use ESFR space
  "MOV   IDX0,3%\n"         ;(IDX0)=DPRAM_add
  "MOV   QX0,3%\n"          ;(QX0)=2*(N_h-1)
  "MOV   QR0,3%\n"

;Read the new input sample and move it
;at x(n-1) address overwriting x(n-1)
  "CoMOV [IDX0],[1%]\n"              ;x(n-1) <- IN
  "CoNOP [IDX0+QX0],[0%+QR0]\n"      ;return the pointers to beginning
;FIR: first multiplication, h(N_h-1)*x((n-N_h+1)
  "CoMUL [IDX0-],[0%-]\n"                 ;(ACC)=h(N_h-1)*x(n-N_h+1)<<1
                                     ;(IDX0)=(IDX0)-2
                                     ;(R12)=(R12)-2
```

```
;FIR loop: repeat N_h-1 times the same MAC instruction
"loop1:\n"                              ;i=(N_h-2):(-1):0
"-USR1    CoMACM [IDX0-],[0%-]\n"       ;(ACC)=(ACC)+h(i)*x(n-i)<<1
  "JMPA   cc_nusr1,loop1\n"

;return the pointers
  "CoNOP  [IDX0+],[0%+]\n"

;Rounding
  "CoRND\n"

;write the 16-bit filter output y(n) into registers R4
  "CoSTORE    2%,MAH\n"      ;(R4)=(MAH)

  : "=w" (y)
  : "w" (h), "w" (in), "w" (N), "w" (D_buffer)
     );
}
```

Keil compiler uses also key word __asm{ } with brace to indicate that the codes in this brace are assembly instructions. Keil compiler supports direct usage of variable names in assembly code. With Keil compiler the inline assembly Fir filter function can be implemented:

```
short  Fir(short *h, short *in, short N, short *D_buffer)
{
      short y;

__asm {
  ;MAC registers initialization
     MOV     MCW,#0600h     ; MP=1, MS=1
     SUB     N_h,#2h        ; N_h-2
     MOV     MRW,N_h        ;(MRW)=N_h-2, repeat counter

  ;ESFR register initialization
     ADD     N_h,#1           ;(R14)=N_h-1
     SHL     N_h,#1         ;(R14)=2*(N_h-1)
     EXTR #3                ; Next 3 instructions use ESFR space
     MOV     IDX0,D_buffer  ;(IDX0)=DPRAM_add
     MOV     QX0,N_h        ;(QX0)=2*(N_h-1)
     MOV     QR0,N_h

  ;Read the new input sample and move it
  ;at x(n-1) address overwriting x(n-1)
     CoMOV   [IDX0],[IN]              ;x(n-1) <- IN
     CoNOP   [IDX0+QX0],[h+QR0]       ;return the pointers to beginning
  ;FIR: first multiplication, h(N_h-1)*x((n-N_h+1)
     CoMUL   [IDX0-],[h-]             ;(ACC)=h(N_h-1)*x(n-N_h+1)<<1
                                      ;(IDX0)=(IDX0)-2
                                      ;(R12)=(R12)-2
  ;FIR loop: repeat N_h-1 times the same MAC instruction
     loop1:                           ;i=(N_h-2):(-1):0
     -USR1   CoMACM [IDX0-],[h-]      ;(ACC)=(ACC)+h(i)*x(n-i)<<1
             JMPA cc_nusr1,loop1

  ;return the pointers
     CoNOP   [IDX0+],[h+]

  ;Rounding
     CoRND

  ;write the 16-bit filter output y(n) into registers R4
     CoSTORER4,MAH    ;(R4)=(MAH)
    }

    return  y;
}
```

3.3 DSP Functions from Intrinsic Functions

Every microcontroller has always some assembly instructions that have no equivalence in C language. In order to get the equivalence in C code, usually the C compiler provides a special library function that is implemented with those special assembly instructions. Such functions are called intrinsic functions. MAC assembly instructions belong to such specific instructions that have no equivalence in C. To make it possible to use MAC instructions in C every compiler that supports XC2000, XE166 and XC166 provides a lot of MAC intrinsic functions. Following is an example of using MAC intrinsic function to implement a multiplication:

```
short  mul(short x, short y)
{
   short  r;

   _CoMUL(x,y);
   r = CoSTOREMAH();

   return  r;
}
```

The function operates a multiplication of the 16-bit integer x with integer y. The multiplication operation is executed by intrinsic function _CoMUL. The result is passed to variable r through the intrinsic function CoSTOREMAH. After compiling we have the following assembly code:

```
......

CoMUL     R12,R13
CoSTORE   R12,MAH

......
```

The usage of intrinsic functions is very limited. Firstly, the MAC intrinsic functions are not complete because some MAC assembly functions are difficult to implement in intrinsic functions. Secondly, due to the incomplete MAC intrinsic functions some DSP functions cannot be implemented using MAC intrinsic functions. Thirdly, the DSP function implemented with MAC intrinsic functions is not efficient in comparison with the assembly implementation. So, it is not good way to implement DSP functions for MAC unit using intrinsic functions.

3.4 DSP Function from Library

In addition to using inline assembly functions and the MAC intrinsic functions we can also develop DSP functions at C level by calling an established DSP library. Usually the inline assembly functions and the intrinsic functions are limited to porting short DSP functions due to the performance loss and possible register conflict. For complicated algorithms, e.g. adaptive filter, FFT, using a DSP library is a good choice.

In order to make the MAC instructions easy to use we provide customers with a free of charge DSP library which is hand optimized with MAC assembly instructions implemented. The DSP library contains many basic DSP functions, e.g. Fir filter, IIR filter, adaptive filter, FFT, static functions and many basic mathematic functions. The DSP library can be downloaded from http://www.infineon.com/C166DSPLIB .

In the following we give an example to show how to use the DSP library to realize a Fir filter:

```
#include "DspLib.h"        //include header
#include <stdio.h>
#include <stdlib.h>

#define  N_h  4             //Filter Order
#define  N_x  8             //length of input data

_near short D_buffer[N_h] _at(0x0f600);  //define the delay buffer in DPRAM area
short X[N_x]={-10408,2233,14884,12499,22183, //input data vector in 1Q15 format
               4461,-8493,13286};
short H[N_h]={9519,15268,-4326,3654};         //filter coefficients in 1Q15

void main()
{
    short y[N_x];       //output vector
    short i;

  //clear the buffer
     for(i=0; i<N_h; i++)
        D_buffer[i] = 0;

  // Fir filter, call library function
     for(i=0; i<N_x; i++)
        y[i] = Fir_16(H, &X[i], N_h, D_buffer);

}
```

3.5 DSP Functions in C

C language is the most used programming language in embedded applications. However, programming with C needs C compiler support. If the C compiler does not support the MAC instruction set, we cannot write the program in C for the MAC unit. So far, only the Tasking Viper compiler contains an automatic MAC code generator and support for MAC DSP programming in ANSI C and ISO C99. In the following we provide an example to implement the same fixed point Fir filter in C with the Viper compiler. The program consists of the main function and the Fir filter function Fir_16:

```
#include <stdio.h>
#include <stdlib.h>

#define ABSVAL(A)  ((A) >= 0 ? (A) : (-A))
#define MAXERROR 10

#define   N_h     4          //Filter Order
#define   N_x     8          //length of input data used in the example
```

```
//input data vector in 1Q15 format
short  x[N_x] ={-10408,2233,14884,-12499,22183,4461,-8493,13286};

//filter coefficients in normal order and 1Q15 format
short  h[N_h] ={3654, -4326,15268,9519,}; // ={h(0), h(1), h(2), h(3)}

//reference result vector
short  rtest[N_x] = {-1161,1623,-3484,-5342,11708,-3930,5170,11126};

short Fir_16(short *h, short IN, short n_h);

short Fir_16(short *h, short IN, short n_h)
{
  short i;
  long y;
  static short D_buffer[N_h]={0,0,0,0} ;

  y =0;

  for (i=n_h-1;i>0;i--)
    D_buffer[i] = D_buffer[i-1];
  D_buffer[0] = IN;

  for(i=0; i<n_h; i++)
  {
    y = y + ((long)h[i] * (long)D_buffer[i]);
  }

  return (short)(y>>15);
}
void main()
{
  short  y[N_x];             //output vector
  short  i, diff, errflag=0;

//Fir filter
  for(i=0; i<N_x; i++)
    y[i] = Fir_16(h, x[i], N_h);

/*------ test the function ------*/
//print the results, reference results and their difference
//the difference should be less than 10.
  for(i=0; i<N_x; i++)
  {
    diff = rtest[i] - y[i];
    printf("y[%d] = %5d; %5d; %5d\n", i, y[i], rtest[i], ABSVAL(diff));

    if(ABSVAL(diff) > MAXERROR)
      errflag = -1;
  }

  if(errflag == -1)
    printf("\nTest result: error \n");
  else
     printf("\nTest result: ok\n")

}
```

Fig. 7 shows the MAC codes generated by the Viper compiler. The left side in Fig. 7 lists the C code, and on the right side the corresponding MAC assembly codes are displayed. Note: if we want to generate MAC codes, we need to select the MAC generator in the Viper compiler through choosing option *project->properties->C/C++ Compiler->Code Generation->Automatic MAC code generation*.

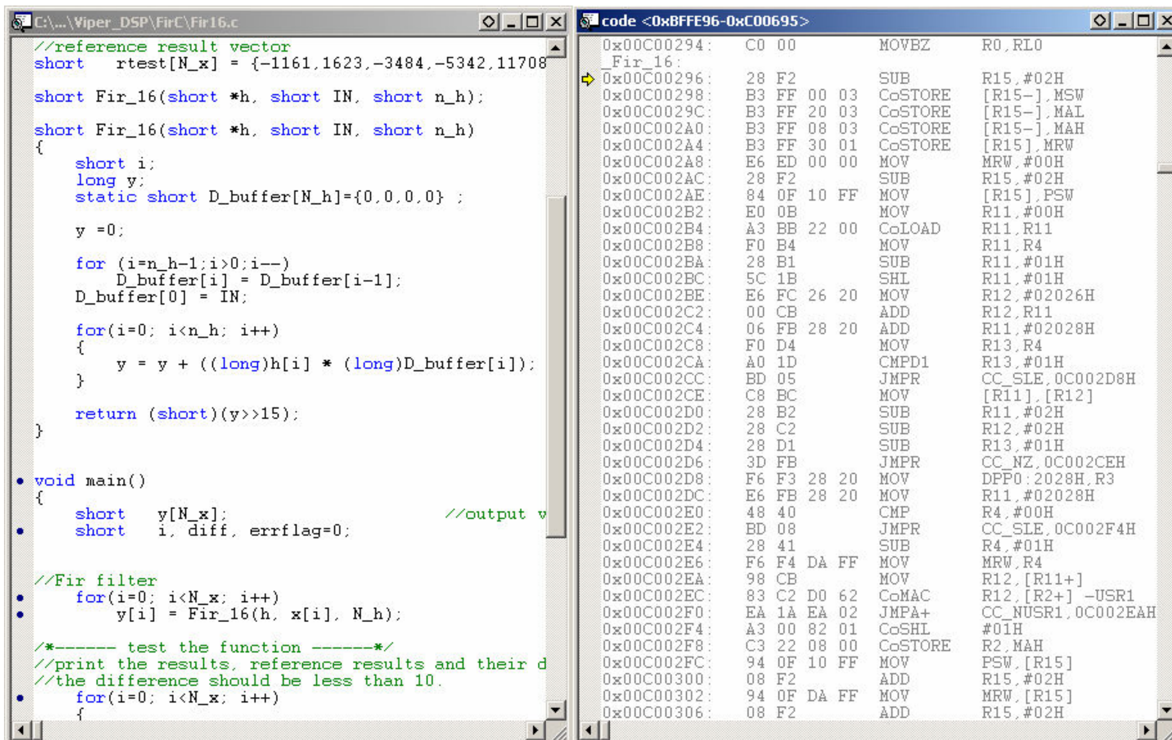

Figure 7: MAC instructions generated by Viper compiler

3.6 Summary

In this section we introduced some useful methods of DSP programming with MAC unit.

For users of Keil and the classic Tasking compiler:

- Infineon provided DSP library is always the first choice, if your application uses the standard basic DSP functions and filters.
- If the DSP function in you application is small, the inline assembly function is also a good method.

For users of the Viper compiler:

- The DSP library is the first choice if the DSP functions in the library can be integrated in your application, because the library functions are already optimized.
- If there are none of your required functions in the DSP library, or the performance of the DSP function is not critical, you can write the DSP function directly in C, and let Viper generate the MAC code automatically.
- For some small DSP routines the inline assembly method is also a good choice.

4 DSP Optimization Techniques

In the last section we are focused on DSP development approaches. The commonly used DSP development methods for XC2000, XE166 and XC166 microcontrollers are described in detail. In this section we introduce some DSP program optimization methods based on XC2000, XE166 and XC166 core architecture.

We know that in the XC2000, XE166 and XC166 microcontrollers two different addressing models are used for normal instruction set and MAC instruction set. Due to these different data memory and addressing modes the instruction pipeline conflicts are unavoidable in some cases, which can dramatically decrease the DSP performance through delays. From the DSP optimization point of view such hardware disadvantages can be used by us to optimize the DSP implementation because we can remove the delays caused by instruction pipeline conflicts through software pipelining.

Basically, there are two classes of DSP optimization techniques, chip-independent and chip-dependent. The chip-dependent optimization techniques depend on the microcontrollers. Different types of microcontroller have different optimization techniques. However, the chip-independent optimization techniques can be used generally. We know that the purpose of DSP optimization is to implement DSP algorithms in minimal cycles. The tricks of optimization are endless. Here, according to our experiences we summarize some useful DSP optimization principles. First, we introduce the chip-independent optimization techniques which can be used in different microcontrollers and DSP processors. Then, the optimization techniques used in the XC2000, XE166 and XC166 microcontrollers are given.

4.1 Chip-independent Techniques

4.1.1 Packet Processing

The idea of packet processing is to process the data in packet. This can be done at C-level and assembly level. For developing embedded DSP software C and Assembly are the most frequently used program languages. Using packet processing can save many cycles compared to using separate processing because some data processions do not need to be executed repeatedly by packet processing.

- ***At C-Level***

The most frequently used approach for developing embedded DSP software is the mixed C and assembly, which means that the C and assembly languages are used in parallel during programming. In such methods the kernel DSP algorithms must be implemented in assembly because assembly is more efficient than C for optimization goal. The assembly kernel routines will be called as subroutines by the main C program.

Basically there are two ways to call the assembly kernel routines according to the input data format. One of them is with loop. Another is without loop. If we have K samples of input data waiting for processing and the data will be inputted sample by sample, then the kernel routine needs to be called K times with loop. This means all instructions in the kernel assembly routine have to be repeated K times. Usually the first and last parts of instructions in the assembly routine are used for argument transfer from C to assembly, storing and restoring register states. Suppose that they need (M+N) cycles. Actually it is only necessary to repeat the instructions for the algorithm itself. From the optimization point of view the repetition of those instructions is unnecessary. In order to save the executing cycles and avoid repeating those parts of instructions we can move the loop at C-level into assembly routine between first and last part. Then, the instructions for argument transfer, storing and restoring register states need only to be executed one time and we can save (K+1)(M+N) cycles. In this case the kernel DSP assembly routine is called without loop. The input data should be defined as vector. Fig. 8 shows the packet processing at C-level schematically.

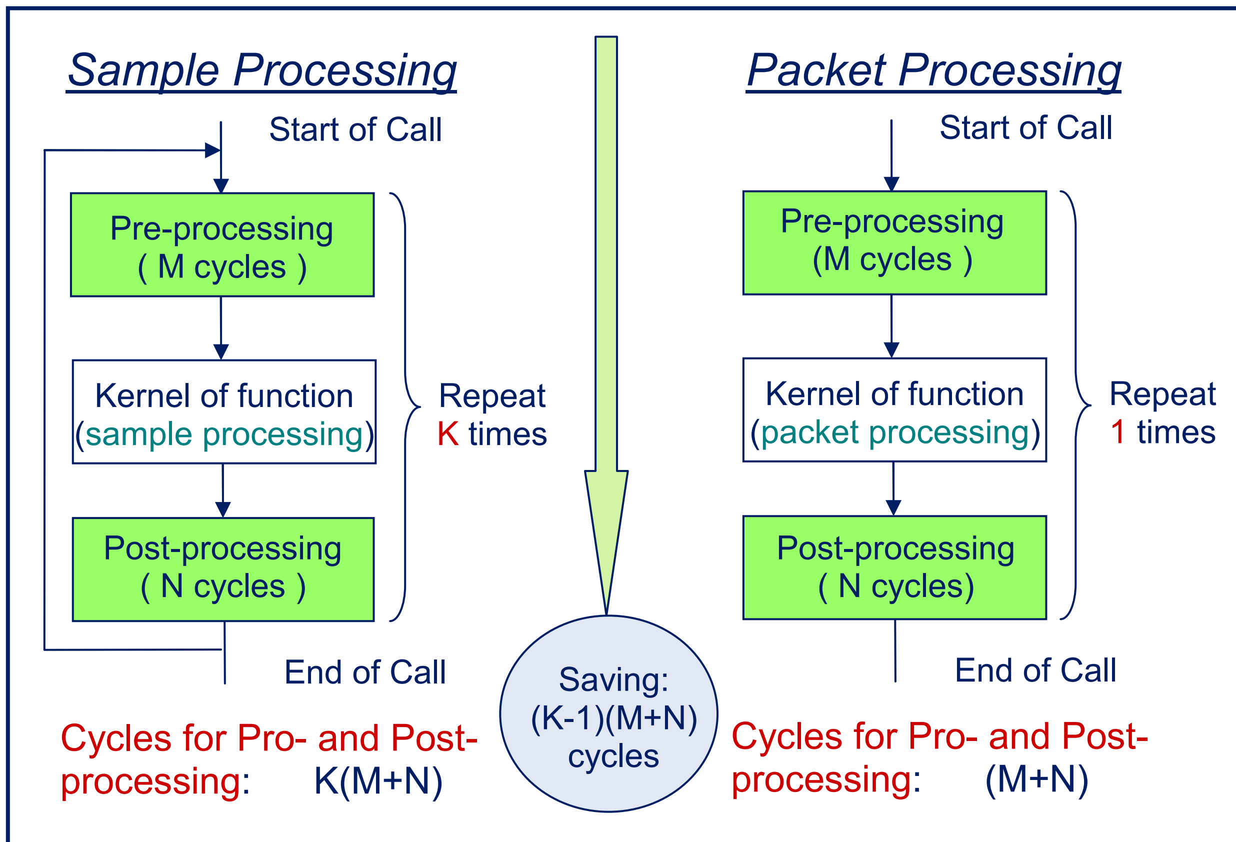

Figure 8: Schematic representation of packet processing

- ***At Assembly Level***

The idea of packet processing can be realized at assembly level with different forms, e.g. Packed Operation, Packed Load/Store, etc.

Packed Operation: Packed operation means that we pack two different data in the same register for the packed load/store purpose. For example, a 16-bit register can pack two 8-bit data coming from an 8-bit A/D converter. In the following we take Infineon's XC166 microcontroller as an example to show the differences. Without Packet Operation the read and write operations need 4 cycles:

```
;read inputs in byte
   MOVB R2L,[R1+] ;first input, 1 cycle
   MOVB R2H,[R1+] ;second input, 1 cycle
   .
   .
;processing
   .
   .
;output in byte
   MOVB [-R1],R2L ;first output, 1 cycle
   MOVB [-R1],R2H ;second output, 1 cycle

With Packed Operation the read and write operations need only 2 cycles:

;read inputs in word
   MOV  R2,[R1+]  ;input, 1 cycle
   .
   .
;processing
   .
```

```
    .
;output in word
  MOV  [-R1],R2  ;output, 1 cycle
```

Packed Load/Store: With packed load/store at least two data are loaded or stored in the same instruction. Refer to the example by packed operation.

4.1.2 Data Memory Interleaving

The goal of data memory interleaving is that the data are ordered so that the program can read or write them at minimal cycles. Data memory interleaving is usually done at C-level by data defining. With memory interleaving the data are mixed with other type of data in memory.

For example we have two 8-bit complex data x and y. Usually the complex data will be stored in the data memory in the order: real(x) image(x) real(y) image(y). If we want to make reading the real parts easy we can interleave the data memory as real(x) real(y) image(x) image(y) shown in Fig. 9.

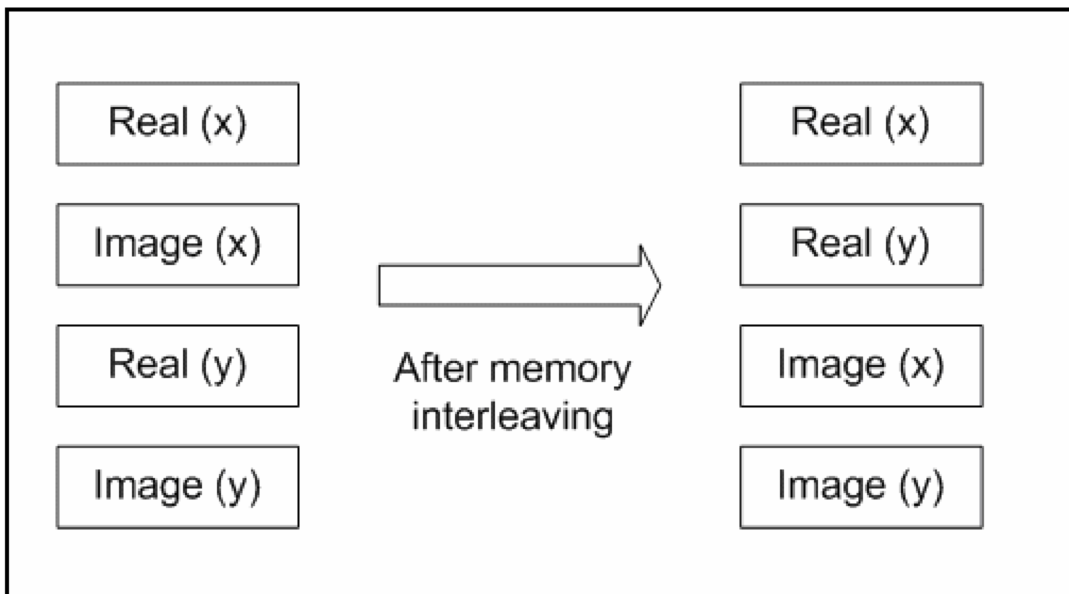

Figure 9: Example of data memory interleaving

4.1.3 Loop Unrolling

Loop unrolling is a well-known optimization method for developing embedded software, and can be used generally. The idea is to repeat the same operations two or more times inside a loop in order to reduce the loop counter and then the cycles used for loop checking. Here is an example using assembly of Infineon's XC166 microcontrollers. The program flow diagram is showed in Fig. 10.

Example (unrolling factor 2):

Assembly code without loop unrolling (17 cycles):

```
        ............

        MOV    R3, #3
loop:
        CoMAC  [IDX0+], [R1+]
        CoMAC  [IDX0+], [R1+]
        CMPD1  R3,#0h
        JMPR   cc_NZ,loop

        ............
```

Assembly code with loop unrolling (13 cycles):

```
        ............
        MOV     R3, #1
loop:
        CoMAC   [IDX0+], [R1+]
        CoMAC   [IDX0+], [R1+]
        CoMAC   [IDX0+], [R1+]
        CoMAC   [IDX0+], [R1+]
        CMPD1   R3,#0h
        JMPR    cc_NZ,loop

        ............
```

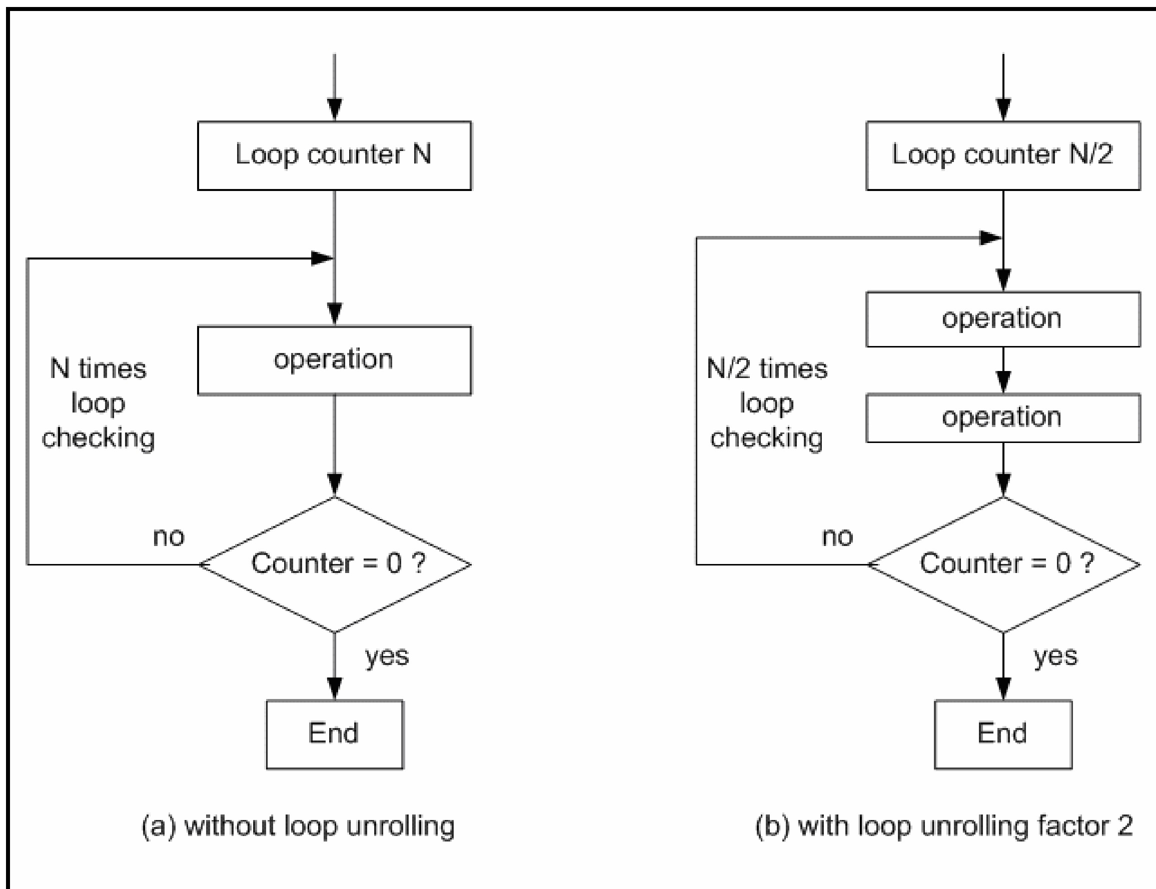

Figure 10: Program flow diagram of loop unrolling

4.1.4 Software Pipelining

Software pipelining is a very general optimization principle and can be used on different types of chips. Usually the software pipelining is implemented at assembly level. With software pipelining the instructions must be reordered in order to delete the possible instruction pipeline conflicts caused by hardware and then get the minimal cycles for executing the instructions.

Different microcontrollers have different instruction pipelines. Therefore, the realization of this optimization principle is chip-dependent. We will take Infineon's XC166 microcontroller family as an example to show the application of the software pipeline optimization principle. The same principles can also be applied to the XC2000 and XE166 families.

4.2 Chip-dependent Techniques

We first briefly introduce the instruction pipeline of XC166 shown in Fig. 11, then give some chip-dependent optimization techniques for XC166 based on software pipeline optimization principles.

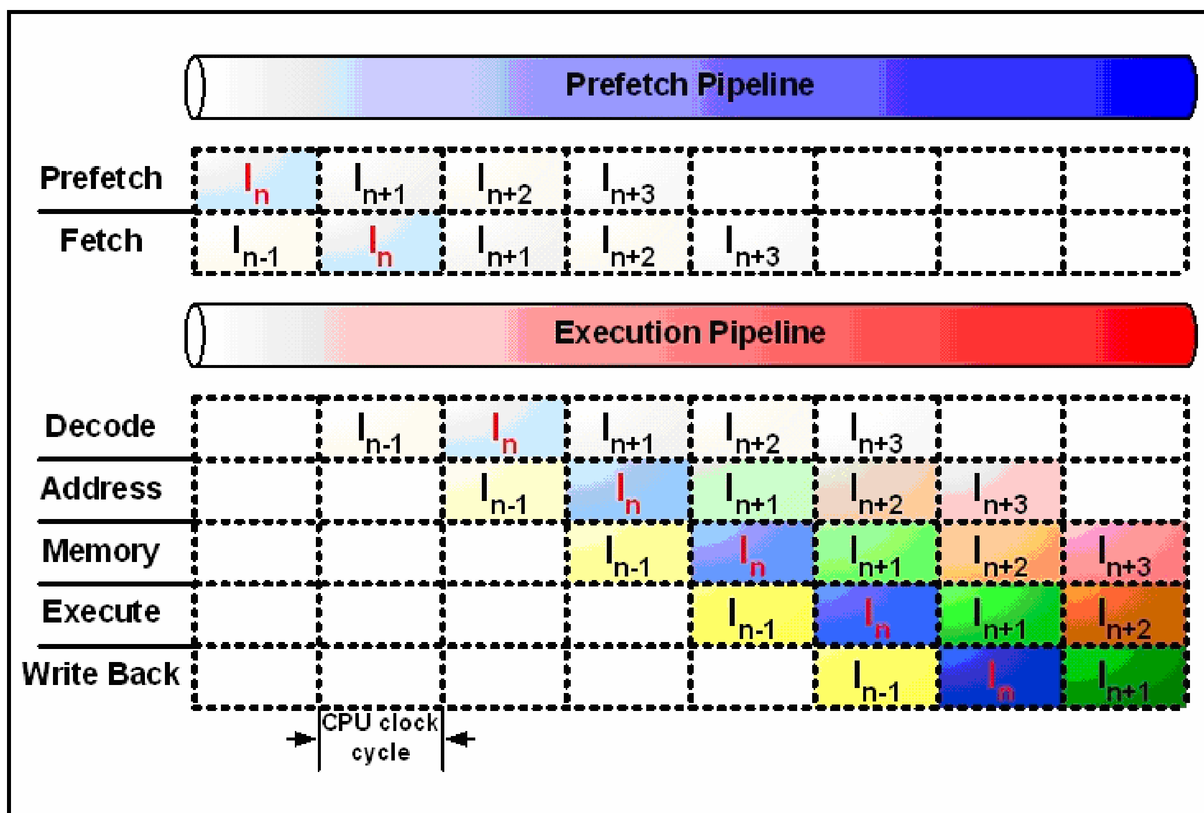

Figure 11: Schematic representation of XC166 instruction pipeline

The pipeline of the XC166 CPU has seven stages. Each stage processes its individual task. The first two stages make up the instruction fetch pipeline and the remaining five stages form the instruction processing pipeline. The instruction fetch pipeline is used to pre-fetch instructions and to store them into an instruction FIFO. The preprocessing of branch instructions in combination with the instruction FIFO allows filling of the execution pipeline with a continuous flow of instructions. In the case of an incorrectly predicted instruction flow, the instruction fetch pipeline is bypassed to reduce the number of dead cycles. All instructions must pass through each of the five stages of the instruction processing pipeline regardless of the need of some stages to complete an execution of a certain instruction. In following sections, the pipeline operations of each stage are described.

Fetch Pipeline

- Prefetch: *This stage pre-fetches instructions from the Program Memory Unit (PMU) in the predicted order. The instructions are pre-processed in the branch detection unit to detect branches. The prediction logic decides if the branches are assumed to be taken or not.*
- Fetch: *The instruction pointer of the next instruction to be fetched is calculated according to the branch prediction rules. For zero-cycle branch execution, the Branch Folding Unit pre-processes and combines detected branches with the proceeding instructions. Pre-fetched instructions are stored in the instruction FIFO. At the same time, instructions are transported out of the instruction FIFO to be executed in the instruction processing pipeline.*

Execution Pipeline

- Decode: *The instructions are decoded and, if required, the register file is accessed to read the GPR used in indirect addressing modes.*
- Address: *All the operand addresses are calculated. The SP register is de/incremented for all instructions which implicitly access the system stack.*
- Memory: *All the required operands are fetched.*
- Execute: *An Arithmetic and Logic Unit (ALU) or MAC-Unit operation is performed on the previously fetched operands. The condition flags are updated. All explicit write operations to CPU special function registers (SFRs) and all auto-in/decrement operations of general purpose registers (GPRs) used as indirect address pointers are performed.*
- Write Back: *All external operands and the remaining operands within the internal DPRAM space are written back. Operands located in the internal SRAM are buffered in the Write Back Buffer.*

Because up to five different instructions are processed simultaneously on the XC166 CPU, the bandwidth limitations and data dependencies may exist between instructions in different pipeline stages. The delay caused by the pipeline conflicts can be used by other instructions to optimize performance. In the following we apply software pipelining optimization principle on XC166 and introduce some optimization techniques for DSP software development on XC166 according to its instruction pipeline.
Although XC166 has dedicated hardware to detect and to resolve different kinds of instruction conflicts, there are also some cases in which the pipeline requires attention by the programmer.

4.2.1 Data Dependencies Removal

Due to the pipeline requirement of the XC166 CPU there are a lot of possible data dependencies between instructions using GPRs. In the XC166 the dedicated hardware is added to detect and resolve the data dependencies. However, in the XC166 CPU none of the instructions using indirect addressing modes are capable of using a GPR, which is to be updated by one of the two immediately preceding instructions. This means that the instruction using indirect addressing modes will lead to a two cycle stall. To use these two cycles for the optimization we can insert before this instruction a multicycle or two single cycle instructions that must not update the GPR used for indirect addressing.

Example:

```
Assembly without optimization  ( 6 cycles )

     .............

     ADD    R1, R2
     MOV    R8, [R1]     ; instruction using indirect addressing mode
     ADD    R5, R1
     ADD    R6, R1

     .............
Assembly with optimization  ( 4 cycles )
```

```
    .............

    ADD    R1, R2
    ADD    R5, R1       ; inserted one cycle instruction
    ADD    R6, R1       ; inserted one cycle instruction
    MOV    R8, [R1]     ; instruction using indirect addressing mode

    .............
```

In the case of read accesses using indirect addressing modes, the address generation unit on the XC166 uses a speculative addressing mechanism. The read data path to one of the different memory areas is selected according to a history table before the address is decoded. This history table has one entry for each of the GPRs. The entries store the information of the last accessed memory area using the corresponding GPR. In the case of an incorrect prediction of the memory area, the read access must be restarted.
It is recommended that the GPR used for indirect addressing points to the same memory. If an updated GPR points to a different memory area, the next read operation will access the wrong memory area. The read access must be repeated, which leads to pipeline stalls.

Example:

```
Assembly without optimization  ( 6 cycles )

    .............

    CoMUL  R3, [R0]  ; R0 points to DPRAM area
    MOV    R0, R4    ; update R0

    .............

    MOV    DPPX, ... ; change data page

    .............

    CoADD  R6, [R0]  ; R0 points to now SRAM area
    CoADD  R6, R1

    .............
Assembly with optimization  ( 4 cycles )
    .............
    CoMUL  R3, [R0]  ; R0 points to DPRAM area
    MOV    R0, R4    ; update R0

    .............

    CoADD  R6, [R0]  ; R0 points still to DPRAM area
    CoADD  R6, R1

    .............
```

4.2.2 Memory Bandwidth Conflicts Removal

Memory bandwidth conflicts can occur if instructions in the pipeline access the same memory area at the same time. The CoXXX instructions on XC166 MAC unit are specially designed for DSP implementation. To avoid the memory bandwidth conflicts in the DPRAM area, one of the operands should be located in the internal SRAM to guarantee a single cycle execution time of the CoXXX instructions.

Example:

```
Assembly without optimization  ( 4 cycles )
```

```
    .............

    ADD    R6,R1
    CoMAC  [IDX0], [R0]   ; IDX0 and R0 both point to DPRAM area
    MOV    R3, [R0]       ; one cycle stall

    .............
```

```
Assembly with optimization  ( 3 cycles )
    .............

    ADD    R6,R1
    CoMAC  [IDX0], [R0]   ; IDX0 points to DPRAM area
    MOV    R3, [R0]       ; R0 points to SRAM area

    .............
```

4.2.3 Instruction Re-ordering

When writing DSP routines with MAC instructions, the Special Function Registers (SFRs) often need to change and update, such as IDX0, IDX1, QX0, QX1, QR0, QR1, and so on. CPU-SFRs control the CPU functionality and behaviour. All instructions in the following list will be held in the decode stage until the CPU-SFRs are updated, if they follow a CPU-SFR modification instruction. In this case, a delay with three cycles will occur.

- Instructions using long addressing mode
- Instructions using indirect addressing modes, except JMPI and CALLI
- All MAC instructions (CoXXX instructions)

With instruction re-ordering the flow of instructions through the pipeline can be improved to delete the delays.

Example:

```
Assembly without optimization  ( 8 cycles )

    .............

    MOV    IDX0,R1
    CoMAC  [IDX0], [R0]   ; 3 cycle stall
    MOV    R3, R1
    ADD    R2, R3
    MOV    R1, R2

    .............
```

```
Assembly with optimization  ( 5 cycles )
    .............

    MOV    IDX0,R1
    MOV    R3, R1
    ADD    R2, R3
    MOV    R1, R2
    CoMAC  [IDX0], [R0]

    .............
```

4.2.4 Using Flash Waitstates

The flash memory in XC2000 has typically 4 wait states over 20 MHz system frequencies. The bus wide of XC2000 flash is 64 bits, which means that in each time 64 bits data can be read without wait states. For MAC instruction 64 bits correspond to 2 MAC instructions and 4 normal instructions because each MAC instruction has 4 bytes code size, and the normal instruction has only 2 bytes.

If the code is executed from flash in XC2000 microcontroller, every 2 CoXXX instructions or every 4 normal instructions could have 4 cycles delay. We can use these 4 cycles for the delay caused by pipeline conflict.

Example:

```
Assembly without optimization  ( 12 cycles )

     .............

     ADD     R1,#2
     MOV     IDX0,R1
     CoMAC   [IDX0], [R0]   ; 3 cycle stall

     MOV     R3, R1         ; 4 cycle waitstates
     ADD     R2, R3

     .............
Assembly with optimization  ( 5 cycles )

     .............

     MOV     R3, R1
     ADD     R2, R3
     MOV     IDX0,R1        ; 3 cycle stall in 4 cycle waitstates

     CoMAC   [IDX0], [R0]   ; 1 cycle delay

     .............
```

4.2.5 Using PSRAM

The XC2000 and XE166 families have up to 64 Kbytes PSRAM that can be used to store codes and dates. The PSRAM has no waitstates. So we can locate the DSP routines in PSRAM to get better performance in comparison with flash. In general, the code in PSRAM is two times faster than in flash for the XC2000 and XE166 families.

5 Conclusion

In this application note we introduced many useful DSP software development methods and their optimization techniques based on the XC2000, XE166 and XC166 microcontroller families. The MAC architecture in C166S V2 CPU code, in particular the MAC address mode, is described from the point of software development of view in detail. The provided information in this application note can be considered as a user guide for DSP programming the XC2000, XE166 and XC166 microcontroller families.

6 Reference

[1] User's Manual: C166S V2 Core, v1.7, January 2001.
[2] XC2200 Derivatives: User's Manual, V1.0 June 2007.
[3] Guangyu Wang, "Implementation of DSP Algorithms on Microcontrollers with MAC Unit", Elektronik, March 2005, Page 80-85.
[4] Guangyu Wang, "DSP Optimization Based on XC166 Microcontroller Architecture", Electronik, June 2006, Page 50-55.

http://www.infineon.com

Published by Infineon Technologies AG