

Arithmetic functions on low-power micro-controllers: an architectural comparison

Daniel Roggen
Wearable Computing Laboratory, Institut für Elektronik
ETH Zürich, Switzerland.
droggen@gmail.com
Version 1.1

Abstract

In fields such as wearable computing or robotics, dedicated micro-controllers are often used to collect and process sensor data (sensor nodes) or interface actuators. High execution speed is desirable on these nodes: it allows more complex processing algorithms to run on the node, e.g. to reduce the power consumption during wireless data transfer by compression, or to perform more sophisticated actuator control.

In this paper we make a comparison of implementations of operations typical of signal processing on three commonly used low-power micro-controller architectures (Texas Instruments MSP430, Atmel AVR8 and Microchip PIC18). This comparison highlights the strengths and weakness of these architectures. Its purpose is to help make informed decisions on the kind of algorithms that may be implemented on these architectures, in terms of available computational power.

This report compares multiplication and multiply-accumulate in integer arithmetics and multiplication in fixed-point arithmetics across these three architectures. These operations are typical in signal or image processing and they are time consuming in comparison to e.g. additions, comparisons, or other execution flow control instructions. Various operand size and result precision are considered to fit a wide range of applications and the execution time of these operations is reported.

1 Introduction

In fields such as wearable computing, robotics or wireless networked sensing, dedicated micro-controllers are often used to collect, process and transmit sensor data (*sensor nodes*) or to interface actuators.

High execution speed is desirable on these nodes: it allows more complex processing algorithms to run on the node, e.g. to reduce the power consumption during wireless data transfer by compression, or to perform more sophisticated actuator control.

The objective of this report is to compare architectures commonly used in sensor nodes in terms of their performance at executing operations typical of signal processing, thereby highlighting the strengths and weaknesses of various architectures. Its purpose is to help make informed decisions on the kind of algorithms that may be implemented, in terms of available computational power.

There exist a large number of micro-controller families. We consider here micro-controllers that are commonly used in wireless sensor nodes. Wireless sensor nodes have strong constraints from the power consumption point of view, and therefore they use low-power micro-controllers [4, 6, 10]. In [10, 6] a number of sensor nodes are reviewed. From this review it appears that the most common micro-controller architectures are the AVR8 from Atmel, the MSP430 from Texas Instruments, the PIC18 from Microchip, and micro-controllers based on the 8051 architecture. More powerful sensor

nodes use micro-controllers based on the ARM architecture¹ as well as normal ARM processors (XScale, StrongARM).

The micro-controllers selected for this review are the Texas Instruments MSP430, the Atmel AVR8 and the Microchip PIC18. They are all commonly used in sensor nodes and in other low-power systems. They have comparable instruction execution time, operating frequency, program and data memory size, pricing, and they all have a wide range of peripherals, and in particular analog to digital converters for signal acquisition. In addition, these micro-controllers all have hardware multiplication units which allows efficient implementation of signal processing algorithms. The MSP430 is a 16-bit micro-controller, while the other two are 8-bit.

Optimized implementations of arithmetic functions on these different micro-controllers are compared here. We assume that these arithmetic operations are the most time consuming part of the application running on the micro-controller. In other words, we assume that these operations are in innermost loops of programs and we optimize them for speed (e.g. by using fast register access instead of slower memory access).

This report compares across the three architectures the multiplication, multiply-accumulate (MAC) and vectorial multiply-accumulate² operations in integer arithmetics, and fixed-point multiplications. These operations are typical in filters and other signal processing algorithms and they are time consuming in comparison to e.g. additions, comparisons, or execution flow control instruction. MAC and vectorial MAC are typically used to implement filters or matrix multiplications. Fixed-point notation allows to represent fractional numbers between e.g. [-1;1). Division operations are not considered since their execution time is prohibitive. Multiplication by the inverse is preferred instead, using a fixed-point representation. Various operand size and result precision are considered to fit a wide range of applications. The following integer arithmetic functions are considered (the notation $AxB \rightarrow C$ indicates the bit size of the operands A and B and the bit size of the result C): signed and unsigned multiplication ($8x8 \rightarrow 16$, $16x16 \rightarrow 32$), signed and unsigned MAC ($8x8 \rightarrow 16$, $8x8 \rightarrow 32$, $16x16 \rightarrow 32$, $16x16 \rightarrow 48$), vectorial signed and unsigned MAC ($8x8 \rightarrow 32$, $16x16 \rightarrow 32$ and $16x16 \rightarrow 48$). The following fixed-point arithmetic functions are considered: signed multiplication in $7.1x7.1 \rightarrow 15.1$ and $1.15x1.15 \rightarrow 1.31$ format (the a.b notation indicates the number of bits before and after the decimal point in a fixed-point notation).

For each operation the assembler implementation is provided (in the native assembly language of each micro-controller) and the number of clock cycles required for execution is listed for various operand source (e.g. operands in memory or registers) and result destination (e.g. stored to memory or to registers). In the assembler code, the number of clock cycles required by each instruction is indicated in parenthesis as a comment next to the instructions.

In these implementations we assume registers are available to implement the desired operations and can be modified without saving. When loops are needed (e.g. vector MAC) we consider implementations with loop unrolling. Effort was made to optimize the arithmetic operations across all the micro-controllers based on the micro-controller data sheets and application notes and re-implementation when needed.

This document is organized as follows. First the implementation of arithmetic operations on the MSP430, AVR8 and PIC18 architectures is described in sections 2, 3 and 4 respectively. The implementations are discussed in section 5, and in section 6 we summarize the results obtained here.

2 MSP430

The MSP430 [9] is a 16-bit RISC CPU architecture with 16 16-bit register R0 to R15. Registers R0 to R3 are special registers (program counter, stack pointer, status registers, and constant

¹Micro-controllers based on the ARM architecture use a the “Thumb” version of the ARM7 which has an instruction set optimized for applications typical of micro-controllers

²Multiply-accumulate (MAC) is the multiplication of two operands and the accumulation of the product in an accumulator (the product is added to the accumulator). Vectorial MAC consists in executing MAC operations between two vectors. The vector elements are multiplied pairwise and the product is accumulated.

Address	Operation
0x130	Store first operand and select unsigned multiplication.
0x132	Store first operand and select signed multiplication.
0x134	Store first operand and select unsigned multiply-accumulate.
0x138	Store second operand and start operation.
0x13A	SumLo: lower 16-bit of result.
0x13C	SumHi: higher 16-bit of result.
0x13E	SumExt: extended sign of multiply operations, or carry of MAC operations.

Table 1: Memory mapping and functions of the hardware MAC unit in the TI MSP430 micro-controller.

generator). Registers R4 to R15 are general purpose registers and can be used with any of the micro-controller instructions and addressing modes (orthogonal instruction set). Flat memory addressing allows 64KB memory access with various addressing modes (e.g. absolute, indexed, register).

The MSP430 has a peripheral offering MAC functionalities³. The hardware MAC unit is capable of 16x16 bit multiplications with 32 bit result (both signed and unsigned source operands), and 16x16 bit multiplication and 32-bit accumulation (only unsigned operands). The operands must be both either unsigned or signed. A combination of signed and unsigned operands is not supported by the hardware. If a combination of signed and unsigned operands is needed, or if operands and results of other sizes are needed, then software must supplement the hardware.

Four memory addresses are reserved for the MAC unit: 0x130, 0x132, 0x134 and 0x138 (see table 1). Writing to one of the first three addresses stores the first operand and selects the type of multiplication to execute (unsigned multiplication, signed multiplication, or unsigned multiply-accumulate respectively). The operation itself starts once the second operand is written to address 0x138.

The result of the operation is stored in three memory-mapped registers: **SumLo** (address 0x13A), **SumHi** (0x13C) and **SumExt** (0x13E). **SumHi** and **SumLo** hold the 32-bit result of the operation. **SumExt** contains the extended sign of the multiplication result when signed or unsigned multiplications operations are executed. **SumExt** contains the carry bit of the sum operation when a multiply-accumulate instruction is executed.

Below the implementation of the arithmetic operations are described (some implementations are taken from [7]). Integer multiplication, integer MAC and fixed-point multiplication are described in this order. First operations with 16-bit operands are considered since they have hardware support on the MSP430. Afterward, 8-bit operations, that require software supplement, are described.

2.1 16x16-bit multiplication, 32-bit result

Assuming that the two 16-bit multiplication operands are in registers R14 and R15, unsigned multiplication is performed as follows:

```
MOV R15,&130h ; (4) Write the first operand and select unsigned multiplication
MOV R14,&138h ; (4) Write the second operand and start the operation
```

Signed multiplication or MAC is performed identically, changing the address to which the first operand is written to (0x132 instead of 0x130).

For comparison with other micro-controllers with multiply instructions (i.e. not a MAC peripheral), and since C compilers use registers to pass arguments to functions and retrieve function return values, it is reasonable to assume that the results of the operation have to be stored in registers. This would translate as follows:

```
MOV R15,&130h ; (4) Write the first operand and select unsigned multiplication
```

³This is a memory-mapped peripheral that is addressed by writing to specific memory locations; it is not a set of processor instructions.

Unsigned 16x16->32-bit MULT			
Load operands from registers and multiply		Load operands from memory and multiply	
MOV	R15,&130h ; (4)	MOV	&op1,&130h ; (6)
MOV	R14,&138h ; (4)	MOV	&op2,&138h ; (6)
Result in registers		Result in memory	Result in MAC unit
MOV	SumLo,R12 ; (3)	MOV	SumLo,RESULT ; (6) -
MOV	SumHi,R13 ; (3)	MOV	SumLo,RESULT+2 ; (6) -

Table 2: Implementation of 16-bit unsigned multiplication on the MSP430. The 16-bit signed multiplication is carried out in the same way changing operand address 0x130 by address 0x132.

```
MOV R14,&138h ; (4) Write the second operand and start the operation
MOV SumLo,R12 ; (3) Move the low 16 bits to result
MOV SumHi,R13 ; (3) Move the high 16 bits to result
```

Assuming that the two 16-bit multiplication operands are fetched from memory and the result stored again in memory (at address RESULT) the code would translate as follows:

```
MOV &op1,&130h ; (6) Write the first operand and select unsigned multiplication
MOV &op2,&138h ; (6) Write the second operand and start the operation
MOV SumLo,RESULT ; (6) Move the low 16 bits to result
MOV SumHi,RESULT+2 ; (6) Move the high 16 bits to result
```

Table 2 summarizes the number of clock cycles needed for 16-bit unsigned multiplication operation with various location of source operands and destination of results. The number of clock cycles needed for 16-bit signed multiplication (both operands signed) is identical.

2.2 8x8-bit multiplication, 16-bit result

The MAC unit can also be used for 8-bit multiplications. The high 8 bits of unsigned 8-bit operands are cleared automatically when using the MOV.B instruction. Unsigned 8x8->16 bit multiplication with source operands in registers and result stored in registers is implemented as follows:

```
MOV.B R15,&130h ; (4) Write 1st operand and select signed multiplication
MOV.B R14,&138h ; (4) Write 2nd operand and start the multiplication
MOV &SumLo,R12 ; (3) Move the low 16 bits to result
```

Signed 8-bit operands must be sign extended, requiring the use of extra instruction SXT. Here temporary register R10 is used to implement efficiently the sign extension before loading the sign extended operand into the MAC unit. Signed 8x8->16 bit multiplication with source operands in registers and result stored in registers is implemented as follows:

```
MOV.B R15,R10 ; (1) Sign extend 1st operand and select signed multiplication
SXT R10 ; (1)
MOV R10,&132h ; (4)
MOV.B R14,R10 ; (1) Sign extend 2nd operand and start multiplication
SXT R10 ; (1)
MOV R10,&138h ; (4)
MOV &SumLo,R12 ; (3) Move the low 16 bits to result
```

Shall the result be stored in memory instead of registers, the code for signed and unsigned 8x8->16 bit multiplication would be identical except for the MOV &SumLo,R12 which is replaced by MOV &SumLo,&ResultL with the corresponding instruction execution time.

In order to fetch the operands from memory, the instructions MOV.B Rx,&13xh are replaced by MOV.B &OpX,&13xh with the corresponding instruction execution time.

Table 3 summarizes the implementations of various 8-bit unsigned and signed multiplications.

Unsigned 8x8->16-bit MULT			
Load operands from registers and multiply		Load operands from memory and multiply	
MOV.B R15,&130h	; (4)	MOV.B &op1,&130h	; (6)
MOV.B R14,&138h	; (4)	MOV.B &op2,&138h	; (6)
Result in register		Result in memory	Result in MAC unit
MOV SumLo,R12	; (3)	MOV SumLo,RESULT	; (6) -
Signed 8x8->16-bit MULT			
Load operands from registers and multiply		Load operands from memory and multiply	
MOV.B R15,R10	; (1)	MOV.B &op1,R10	; (3)
SXT R10	; (1)	SXT R10	; (1)
MOV R10,&132h	; (4)	MOV R10,&132h	; (4)
MOV.B R14,R10	; (1)	MOV.B &op2,R10	; (3)
SXT R10	; (1)	SXT R10	; (1)
MOV R10,&138h	; (4)	MOV R10,&138h	; (4)
Result in register		Result in memory	Result in MAC unit
MOV SumLo,R12	; (3)	MOV SumLo,RESULT	; (6) -

Table 3: Implementation of 8-bit unsigned (top of table) and signed (bottom of table) multiplication on the MSP430.

Unsigned 16x16->32-bit MAC			
Load operands from registers and MAC		Load operands from memory and MAC	
MOV R15,&134h	; (4)	MOV &op1,&134h	; (6)
MOV R14,&138h	; (4)	MOV &op2,&138h	; (6)
Result in register		Result in memory	Result in MAC unit
MOV &SumLo,R12	; (3)	MOV &SumLo,RESULT	; (6) -
MOV &SumHi,R13	; (3)	MOV &SumHi,RESULT+2	; (6) -
Signed 16x16->32-bit MAC			
Load operands from registers and MAC		Load operands from memory and MAC	
MOV R15,&132h	; (4)	MOV &op1,&132h	; (6)
MOV R14,&138h	; (4)	MOV &op2,&138h	; (6)
Result in register		Result in memory	Result in MAC unit
ADD &SumLo,R12	; (3)	MOV &SumLo,RESULT	; (6) -
ADDC &SumHi,R13	; (3)	ADDC &SumHi,RESULT+2	; (6) -

Table 4: 16x16->32-bit unsigned (top) and signed (bottom) MAC operations on the MSP430 with various source of operands and destination of result.

2.3 16x16-bit MAC

Multiply-accumulate operations are used to perform pairwise multiplication and accumulation of operands. Vector MAC performs the pairwise multiplication and accumulation of operands located in two vectors. This is typically used in matrix multiplication or digital filters. The MAC unit provides 32-bit results for multiplication and MAC operations. Below the implementation of MAC and vector MAC operations are described. When implementing 16-bit vector MAC operations the result of the operation may overflow the 32-bits. For this reason 16-bit vector MAC with both 32-bit and 48-bit result are described below.

Table 4 shows 16x16->32-bit unsigned and signed MAC operations. Unsigned MAC operation with 32-bit result is directly executed by the MAC unit. Signed MAC operations with 32-bit result is implemented with hardware signed multiplication and software accumulation.

Unsigned or signed 16-bit MAC operations with 48-bit result must be supplemented by software. Table 5 shows unsigned 16x16->48-bit MAC. The signed implementation is identical with address 0x130 changed to address 0x132.

Vector MAC operations between two memory buffers (e.g. to multiply and accumulate two vectors, or apply a FIR filter to a signal) consists in a sequence of MAC operations for each pair of elements in the memory buffers. Here we consider that loops are unrolled. Furthermore we assume that initialization and finalization time of the vector MAC is negligible in comparison to

Unsigned 16x16->48-bit MAC			
Load operands from registers and MAC		Load operands from memory and MAC	
MOV	R15,&130h	;	(4)
MOV	R14,&138h	;	(4)
Result in register		Result in memory	
ADD	&SumLo,R11	;	(3)
ADDC	&SumHi,R12	;	(3)
ADDC	&SumExt,R13	;	(3)
MOV	&op1,&130h	;	(6)
MOV	&op2,&138h	;	(6)
ADD	&SumLo,RESULT	;	(6)
ADDC	&SumHi,RESULT+2	;	(6)
ADDC	&SumExt,RESULT+4	;	(6)

Table 5: 16x16->48-bit unsigned MAC operations on the MSP430 with various source of operands and destination of result. Signed operation is done in the same way with address 0x130 replaced by address 0x132.

the number of elements to process in the buffers. Therefore we describe only the vector MAC itself. Intermediate results are kept in registers or in the MAC unit for to avoid slower memory accesses. Indirect addressing with post-increment is used to efficiently load source operands from consecutive memory addresses. The memory buffers of length N are pointed to by R5 and R6.

The intermediate accumulation result of unsigned 16x16->32-bit vector MAC is kept in the MAC unit for speed reasons. Unsigned 16x16->32-bit vector MAC is implemented as follows:

```
; 16x16->32-bit unsigned vector MAC
MOV @R5+,&134h      ; (5) IDX=0. Load 1st operand from memory location R5,
                   ; increment R5
MOV @R6+,&138h      ; (5) IDX=0. Load 2nd operand from memory location R6,
                   ; increment R6, start the MAC
MOV @R5+,&134h      ; (5) IDX=1.
MOV @R6+,&138h      ; (5) IDX=1.
....
MOV @R5+,&134h      ; (5) IDX=N-1.
MOV @R6+,&138h      ; (5) IDX=N-1.
```

The signed MAC operation between two memory buffers is executed in the same way, except the accumulation is carried out in software in registers R13:R12:

```
; 16x16->32-bit signed vector MAC
MOV @R5+,&132h      ; (5) IDX=0. Load 1st operand from memory location R5,
                   ; increment R5
MOV @R6+,&138h      ; (5) IDX=0. Load 2nd operand from memory location R6,
                   ; increment R6, start the MAC
ADD &SumLo,R12      ; (3) IDX=0. Accumulate low word
ADC &SumHi,R13      ; (3) IDX=0. Accumulate high word
....
MOV @R5+,&132h      ; (5) IDX=N-1.
MOV @R6+,&138h      ; (5) IDX=N-1.
ADD &SumLo,R12      ; (3) IDX=N-1. Accumulate low word
ADC &SumHi,R13      ; (3) IDX=N-1. Accumulate high word
```

A vector MAC with 48 bit results must be supplemented by software; accumulation result is stored in registers R13:R12:R11. The unsigned vector MAC with 48-bit result is implemented as follows (signed vector MAC is similar with address 0x130 changed to 0x132):

```
; 16x16->48-bit unsigned vector MAC
MOV @R5+,&130h      ; (5) IDX=0. Load 1st operand from memory location R5,
                   ; increment R5
MOV @R6+,&138h      ; (5) IDX=0. Load 2nd operand from memory location R6,
                   ; increment R6, start the MAC
ADD SumLo,R11      ; (3) IDX=0. Accumulate word 0
```

Unsigned 8x8->32-bit MAC			
Load operands from registers and MAC		Load operands from memory and MAC	
MOV.B R15,&134h	; (4)	MOV.B &op1,&134h	; (6)
MOV.B R14,&138h	; (4)	MOV.B &op2,&138h	; (6)
Result in register		Result in memory	Result in MAC unit
MOV &SumLo,R12	; (3)	MOV &SumLo,RESULT	; (6)
MOV &SumHi,R13	; (3)	MOV &SumHi,RESULT+2	; (6)
			-
			-
Signed 8x8->32-bit MAC			
Load operands from registers and MAC		Load operands from memory and MAC	
MOV.B R15,R10	; (1)	MOV.B &op1,R10	; (3)
SXT R10	; (1)	SXT R10	; (1)
MOV R10,&132h	; (4)	MOV R10,&132h	; (4)
MOV.B R14,R10	; (1)	MOV.B &op2,R10	; (3)
SXT R10	; (1)	SXT R10	; (1)
MOV R10,&138h	; (4)	MOV R10,&138h	; (4)
Result in register		Result in memory	
ADD &SumLo,R12	; (3)	ADD &SumLo,&AccL	; (6)
ADDC &SumHi,R13	; (3)	ADDC &SumHi,&AccH	; (6)

Table 6: 8x8->32-bit unsigned (top) and signed (bottom) MAC operations on the MSP430 with various source of operands and destination of result.

```

ADDC SumHi,R12      ; (3) IDX=0. Accumulate word 1
ADDC SumExt,R13    ; (3) IDX=0. Accumulate word 2
....
MOV @R5+,&130h     ; (5) IDX=N-1.
MOV @R6+,&138h     ; (5) IDX=N-1.
ADD SumLo,R11      ; (3) IDX=N-1. Accumulate word 0
ADDC SumHi,R12    ; (3) IDX=N-1. Accumulate word 1
ADDC SumExt,R13   ; (3) IDX=N-1. Accumulate word 2

```

The number of clock cycles to carry out an unsigned 16x16->32-bit vector MAC of length N is $10 \cdot N$; a signed 16x16->32-bit vector MAC of length N requires $16 \cdot N$ clock cycles. A signed or unsigned 16x16->48-bit vector MAC of length N requires: $19 \cdot N$ clock cycles.

2.4 8-bit MAC

MAC with 8-bit source operands and 32-bit results are considered, in particular because vector MAC operations results may overflow 16-bits. The simpler case with 16-bit result is not considered (only SumLo would be used).

Table 6 shows the implementation of 8x8->32-bit signed and unsigned MAC operations. Unsigned 8-bit MAC operation is directly executed by the MAC unit. Signed 8-bit MAC operation is executed by a signed multiply operation with software accumulation.

Vector unsigned 8x8->32-bit MAC is executed as follows (result in MAC unit):

```

; 8x8->32-bit unsigned vector MAC
MOV.B @R5+,&134h    ; (5) IDX=0. Load 1st operand from memory location R5,
                  ; increment R5
MOV.B @R6+,&138h    ; (5) IDX=0. Load 2nd operand from memory location R6,
                  ; increment R6, start the MAC
....
MOV.B @R5+,&134h    ; (5) IDX=N-1.
MOV.B @R6+,&138h    ; (5) IDX=N-1.

```

Vector signed 8x8->32-bit MAC is executed as follows (result in registers R13:R12):

```

; 8x8->32-bit signed vector MAC
MOV.B @R5+,&134h    ; (2) IDX=0. Load 1st operand from memory location R5,

```

```

                                ; increment R5
SXT  R10                        ; (1) IDX=0. Sign extend
MOV  R10,&132h                  ; (4) IDX=0.
MOV  @R6+,R10                   ; (2) IDX=0. Load 2nd operand from memory location R6,
                                ; increment R6, start the MAC
SXT  R10                        ; (1) IDX=0. Sign extend
MOV  R10,&138h                  ; (4) IDX=0.
ADD  SumLo,R12                  ; (3) IDX=0. Accumulate low word
ADC  SumHi,R13                  ; (3) IDX=0. Accumulate high word
....
MOV.B @R5+,R10                 ; (2) IDX=N-1.
SXT  R10                        ; (1) IDX=N-1.
MOV  R10,&132h                  ; (4) IDX=N-1.
MOV  @R6+,R10                   ; (2) IDX=N-1.
SXT  R10                        ; (1) IDX=N-1.
MOV  R10,&138h                  ; (4) IDX=N-1.
ADD  SumLo,R12                  ; (3) IDX=N-1. Accumulate low word
ADC  SumHi,R13                  ; (3) IDX=N-1. Accumulate high word

```

The number of clock cycles to carry out an unsigned $8 \times 8 \rightarrow 32$ -bit vector MAC of length N is $10 \cdot N$; a signed $8 \times 8 \rightarrow 32$ -bit vector MAC of length N requires $20 \cdot N$ clock cycles.

2.5 Fixed-point operations

The MSP430 has no specific instructions to deal with fixed-point numbers. This has to be supplemented in software.

Signed multiplication of two numbers in 1.7 format (registers R14 and R15) with result in 1.15 format in registers R12 looks as follows:

```

MOV.B R15,R10                 ; (1) Sign extend 1st operand and select signed multiplication
SXT  R10                       ; (1)
MOV  R10,&132h                 ; (4)
MOV.B R14,R10                 ; (1) Sign extend 2nd operand and start multiplication
SXT  R10                       ; (1)
MOV  R10,&138h                 ; (4)
MOV  &SumLo,R12               ; (3) Move the low 16 bits to result
ADD  R12,R12                   ; (1) Shift left by 1 bit
; Total: 16 cycles

```

Signed multiplication of two numbers in 1.15 format (registers R14 and R15) with result in 1.31 format in registers R13:R12 looks as follows:

```

MOV  R15,&132h                 ; (4) Write the first operand and select unsigned multiplication
MOV  R14,&138h                 ; (4) Write the second operand and start the operation
MOV  SumLo,R12                 ; (3) Move the low 16 bits to result
MOV  SumHi,R13                 ; (3) Move the high 16 bits to result
ADD  R12,R12                   ; (1) Shift left by 1
ADDC R13,R13                   ; (1) Shift left by 1
; Total: 16 cycles

```

If a result in format 1.7 is desired the same code applies but only register R13 is kept.

2.6 Summary

The number of clock cycles for a few multiplication and MAC operations are summarized in tables 7 and 8 respectively.

Operation	Clock cycles
8x8->16 unsigned MULT. Operands in registers, result in MAC unit	8
8x8->16 unsigned MULT. Operands and result in registers	11
8x8->16 unsigned MULT. Operands and result in memory	18
8x8->16 signed MULT. Operands and results in registers	15
8x8->16 signed MULT. Operands and results in memory	22
16x16->32 signed/unsigned MULT. Operands in registers, result in MAC peripheral	8
16x16->32 signed/unsigned MULT. Operands fetched from memory, results in registers	14
16x16->32 signed/unsigned MULT. Operands fetched from memory, result stored to memory	24
1.7x1.7->1.15 signed fixed-point multiply. Operands and results in registers	16
1.15x1.15->1.31 signed fixed-point multiply. Operands and results in registers	16

Table 7: Clock cycles for a few multiplication operations on the MSP430 micro-controller.

Operation	Clock cycles
8x8->16 unsigned MAC. Operands in registers, result in MAC unit	8
8x8->16 unsigned MAC. Operands and results in registers	11
8x8->16 unsigned MAC. Operands and results in memory	18
8x8->16 signed MAC. Operands and results in registers	15
8x8->16 signed MAC. Operands and results in memory	22
8x8->32 unsigned MAC. Operands in registers, result in MAC unit	8
8x8->32 unsigned MAC. Operands and results in registers	14
8x8->32 unsigned MAC. Operands and results in memory	24
8x8->32 signed MAC. Operands and results in registers	18
8x8->32 signed MAC. Operands and results in memory	28
16x16->32 unsigned MAC. Operands in registers, result in MAC unit	8
16x16->32 signed/unsigned MAC. Operands and results in registers	14
16x16->32 signed/unsigned MAC. Operands and results in memory	24
16x16->48 signed/unsigned MAC. Operands and results in registers	17
16x16->48 signed/unsigned MAC. Operands and results in memory	30
8x8->32 vector unsigned MAC. Operand in memory, result in MAC unit	$N \cdot 10$
8x8->32 vector signed MAC. Operand in memory, result in registers	$N \cdot 20$
16x16->32 vector unsigned MAC. Operand in memory, result in MAC unit	$N \cdot 10$
16x16->32 vector signed MAC. Operand in memory, result in registers	$N \cdot 16$
16x16->48 vector signed/unsigned MAC. Operand in memory, result in registers	$N \cdot 19$

Table 8: Clock cycles for a few isolated MAC operations (top) and vector MAC operations (bottom) on the MSP430 micro-controller. N represents the length of the vector used for vector MAC operations.

Operands in registers	-	Memory load (register addr.)	LD R14,X ; (2)	Memory load (direct addr.)	LDS R14,op1 ; (2)
	-		LD R15,Y ; (2)		LDS R15,op2 ; (2)
Multiply					
MUL	R14,R15 ; (2)				
Result in registers		Result in memory		Result in R1:R0	
MOVW	R17:R16,R1:R0 ; (1)	STS	RHigh,R1 ; (2)		
		STS	RLow,R0 ; (2)		

Table 9: Summary of various 8-bit unsigned multiplication operations on the AVR8. Signed multiplication is done by replacing MUL by MULS with the same number of clock cycles.

3 AVR8

The AVR8[3] is an 8-bit RISC CPU architecture with 32 general purpose 8-bit registers R0 to R31. Flat 16-bit memory addressing allows 64KB memory access with various addressing modes (e.g. direct, direct with displacement, indirect with pre-decrement/post-increment). Memory access is only possible with 3 pointer registers X, Y and Z. These 3 pointer registers are the made by pairs of the CPU general purpose registers: R26 and R27 form 16-bit register X; R28 and R29 form register Y, and R30 and R31 form register Z. Most CPU instructions apply to 8-bit memory elements and registers. Exceptions are MOVW, ADDIW and SBIW instructions. MOVW provides support to move 16-bit values between registers or register and memory. Two consecutive 8-bit registers are used as source or destination of the MOVW instruction. For instance MOVW R17:R16,R1:R0 copies the 16-bit R1:R0 to R17:R16. ADDIW and SBIW add and subtract an immediate value to a 16-bit pair of registers and they provide an efficient way to manipulate pointer registers.

The AVR8 micro-controller has instructions to perform various signed/unsigned 8x8 bit multiplications. Instructions MUL, MULS, MULSU perform an 8x8 bit multiplication with 16-bit result with respectively unsigned, signed, or signed and unsigned operands. All these instructions operate in two clock cycles. The result of the operations are always placed in registers R1:R0. Instruction MULSU allows to efficiently implement intermediate multiplications when operands are larger than 8-bit.

The AVR8 also has instructions to optimize the implementation of multiplications in fixed-point arithmetics in 1.7 format (FMUL, FMULS, FMULSU). These instructions are similar to integer multiplication instructions in terms of operands. The AVR8 however does not include hardware instructions for multiply-accumulate operations. This must be supplemented by software.

Below implementations of the arithmetic operations are described (some implementations are taken from [1, 2]). Integer multiplication, integer MAC and fixed-point multiplication are described in this order. Operations with 8-bit operands are described first since they have hardware support on the AVR8; 16-bit operand operations are described afterward.

3.1 8x8-bit multiplication, 16-bit result

Assuming that the two 8-bit multiplication operands are in registers R14 and R15, unsigned multiplication is performed as follows (result in R1:R0):

```
MUL R14,R15 ; (2) Unsigned multiplication of R14 and R15 with result in R1:R0
```

Signed multiplication is done with instruction MULS instead of MUL. Multiplication of unsigned and signed operand is done with instruction MULSU.

Table 9 summarizes unsigned 8x8->16-bit multiplication with various source operands and result destination (signed multiplication is performed in the same way replacing MUL by MULS without change in clock cycles). Source operands can be fetched from memory in various ways. In indirect register addressing data is loaded from the memory location pointed by one of the registers X, Y or Z. In direct addressing data is loaded from an absolute memory location. When the operands

16x16->32-bit unsigned/signed MULT	
Operands in registers R23:R22 and R21:R20	Memory load (direct addr.)
-	LDS R23,op1h ; (2)
	LDS R22,op1l ; (2)
	LDS R21,op2h ; (2)
	LDS R20,op2l ; (2)
Unsigned multiplication	Signed multiplication
CLR R2 ; (1)	CLR R2 ; (1)
MUL R23,R21 ; (2) ah*bh	MULS R23,R21 ; (2) (signed)ah*(signed)bh
MOVW R19:R18,R1:R0 ; (1)	MOVW R19:R18,R1:R0 ; (1)
MUL R22,R20 ; (2) a1*b1	MUL R22,R20 ; (2) a1*b1
MOVW R17:R16,R1:R0 ; (1)	MOVW R17:R16,R1:R0 ; (1)
MUL R23,R20 ; (2) ah*b1	MULSU R23,R20 ; (2) (signed)ah*b1
	SBC R19,R2 ; (1)
ADD R17,R0 ; (1)	ADD R17,R0 ; (1)
ADC R18,R1 ; (1)	ADC R18,R1 ; (1)
ADC R19,R2 ; (1)	ADC R19,R2 ; (1)
MUL R21,R22 ; (2) a1*bh	MULSU R21,R22 ; (2) a1*(signed)bh
	SBC R19,R2 ; (1)
ADD R17,R0 ; (1)	ADD R17,R0 ; (1)
ADC R18,R1 ; (1)	ADC R18,R1 ; (1)
ADC R19,R2 ; (1)	ADC R19,R2 ; (1)
Result in memory	Result in R19:R18:R17:R16
STS Res3,R19 ; (2)	-
STS Res2,R18 ; (2)	
STS Res1,R17 ; (2)	
STS Res0,R16 ; (2)	

Table 10: Summary of 16-bit unsigned and signed multiplication operations on the AVR8 with operands in register or memory and result kept in registers or stored in memory. Indirect register addressing can be implemented in a similar way without change in clock cycle count.

to multiply are already in registers the multiplication instruction can directly be called. Result of the multiplication can be left in registers R1 and R0 at no cost or moved to another pair of registers with instructions MOVW. The result can be transferred to a memory location with absolute addressing (instruction STS illustrated in the table). Alternatively indirect register addressing can be used (not illustrated in the table). The result can be stored at memory location pointed by register Z with ST Z+,R0 and ST Z+,R1 (store at address Z with post-increment of register Z). The number of clock cycle is identical to absolute addressing (2 clock cycles per store instruction).

3.2 16x16-bit multiplication, 32-bit result

Multiplication of 16-bit operands is performed by decomposing the operation in 8-bit multiplications. These intermediate multiplications can be efficiently implemented with instruction MULSU when operands are of mixed signed and unsigned type (i.e. during signed multiplication).

Table 10 illustrates the 16x16->32 bit unsigned and signed multiplication. Operands are assumed to be in registers R23:R22 and R21:R20 and the result in registers R19:R18:R17:R16. Code may however be modified to handle operands and results in other register sets. If operands have to be fetched from memory (op1h:op1l and op2h:op2l) and the result stored to memory ((Res3:Res2:Res1:Res0), appropriate fetch and store code must be prepended/appended to the multiplication routine as indicated.

3.3 8x8-bit MAC

Multiply-Accumulate operations must be supplemented by software since there is no hardware MAC unit or instruction. The multiplication can however be done with the corresponding instruction.

8x8->32-bit unsigned MAC				8x8->32-bit signed MAC			
CLR	R2		; (1)	CLR	R2		; (1)
MUL	R22,R20		; (2) a*b	MULS	R22,R20		; (2) (signed)a*(signed)b
				SBC	R18,R2		; (1)
				SBC	R19,R2		; (1)
ADD	R16,R0		; (1)	ADD	R16,R0		; (1)
ADC	R17,R1		; (1)	ADC	R17,R1		; (1)
ADC	R18,R2		; (1)	ADC	R18,R2		; (1)
ADC	R19,R2		; (1)	ADC	R19,R2		; (1)
; Total: 7 cycles				; Total: 9 cycles			

Table 11: Implementation of 8x8->32-bit unsigned and signed MAC. Operands are in registers R22 and R20; result is in registers R19:R18:R17:R16.

8x8->32-bit vector unsigned MAC				8x8->32-bit vector signed MAC			
Initialization							
CLR	R2		; (1)				
...				...			
LD	R22,X+		; (2)	LD	R22,X+		; (2)
LD	R20,Y+		; (2)	LD	R20,Y+		; (2)
MUL	R22,R20		; (2) a*b	MULS	R22,R20		; (2) (signed)a*(signed)b
				SBC	R18,R2		; (1)
				SBC	R19,R2		; (1)
ADD	R16,R0		; (1)	ADD	R16,R0		; (1)
ADC	R17,R1		; (1)	ADC	R17,R1		; (1)
ADC	R18,R2		; (1)	ADC	R18,R2		; (1)
ADC	R19,R2		; (1)	ADC	R19,R2		; (1)
...				...			
; Total: 10*N cycles				; Total: 12*N cycles			

Table 12: Implementation of 8x8->32-bit unsigned and signed vector MAC of length N. Operands are in registers R22 and R20; result is in registers R19:R18:R17:R16.

The code for unsigned and signed 8x8->32-bit MAC implementation is illustrated in table 11. Source operands are either in registers or loaded from memory (register or direct addressing) as for the multiplication.

The 8x8->16 MAC is performed in the same way: instructions related to registers R2, R18 and R19 are removed from the code listed in table 11. The number of clock cycles for a signed and unsigned 8x8->16 MAC is 4.

Vector MAC operations can be unrolled; register initialization and finalization (storage of result) is neglected. Vector unsigned and signed 8x8->32-bit MAC operations are illustrated in table 12. Registers X and Y point to the two vectors. Register addressing with post-increment is used to efficiently fetch the operands from memory. The result of the vector MAC is stored in registers R19:R18:R17:R16. Execution time for a vector of length N is $10 \cdot N$ for unsigned vector MAC and $12 \cdot N$ for signed vector MAC.

3.4 16x16-bit MAC

Code for unsigned and signed 16x16->32 bit multiply-accumulate is illustrated in table 13. The register usage is identical as that of the 16 bit multiplication. Code for loading the operands from memory is not illustrated here; it is identical to that used with 16 bit multiplication.

Accumulation on 32-bit may overflow is several MAC operations follow. The result can thus be accumulated on 48 bits. The code is illustrated in table 14.

The unsigned and signed vector 16x16->32-bit MAC operation between two memory buffers pointed by registers X and Y is illustrated in table 15 (this code is repeated N times for a MAC between two vectors of length N).

Unsigned and signed vector 16x16->48-bit MAC is illustrated in table 16.

16x16->32-bit vector unsigned MAC			16x16->32-bit vector signed MAC		
Initialization					
CLR	R2	; (1)			
...			...		
LD	R22,X+	; (2)	LD	R22,X+	; (2)
LD	R23,X+	; (2)	LD	R23,X+	; (2)
LD	R20,Y+	; (2)	LD	R20,Y+	; (2)
LD	R21,Y+	; (2)	LD	R21,Y+	; (2)
MUL	R23,R21	; (2) ah*bh	MULS	R23,R21	; (2) ah*bh
MOVW	R5:R4,R1:R0	; (1)	MOVW	R5:R4,R1:R0	; (1)
MUL	R22,R20	; (2) al*bl	MUL	R22,R20	; (2) al*bl
ADD	R16,R0	; (1)	ADD	R16,R0	; (1)
ADD	R17,R1	; (1)	ADD	R17,R1	; (1)
ADD	R18,R4	; (1)	ADD	R18,R4	; (1)
ADD	R19,R5	; (1)	ADD	R19,R5	; (1)
MUL	R23,R20	; (2) ah*bl	MULSU	R23,R20	; (2) (signed)ah*bl
			SBC	R19,R2	; (1)
ADD	R17,R0	; (1)	ADD	R17,R0	; (1)
ADC	R18,R1	; (1)	ADC	R18,R1	; (1)
ADC	R19,R2	; (1)	ADC	R19,R2	; (1)
MUL	R21,R22	; (2) al*bh	MULSU	R21,R22	; (2) al*(signed)bh
			SBC	R19,R2	; (1)
ADD	R17,R0	; (1)	ADD	R17,R0	; (1)
ADC	R18,R1	; (1)	ADC	R18,R1	; (1)
ADC	R19,R2	; (1)	ADC	R19,R2	; (1)
...			...		
; Total: 27*N cycles			; Total: 29*N cycles		

Table 15: Implementation of 16x16->32-bit unsigned and signed vector MAC of length N. Operands are in registers R23:R22 and R21:R20; result is in registers R19:R18:R17:R16.

The number of clock cycles to carry out an unsigned 16x16->32-bit vector MAC of length N is $27 \cdot N$; a signed 16x16->32-bit vector MAC requires $29 \cdot N$ clock cycles. An unsigned 16x16->48-bit vector MAC requires: $33 \cdot N$ clock cycles; a signed 16x16->48-bit vector MAC requires $39 \cdot N$ clock cycles.

3.5 Fixed-point operations

The AVR8 has instructions to implement multiplications of fixed-point numbers in 1.7 format. Instruction FMUL, FMULS and FMULSU perform unsigned, signed and mixed signed/unsigned multiplications of two numbers in 1.7 format; they also include a left-shift operation to produce a result in the 1.15 format.

Signed multiplication of two numbers in 1.7 format (registers R23 and R22) with result in 1.15 format in registers R21:R20 looks as follows:

```
FMULS  R23,R22          ; (2) Multiplication
MOVW   R1:R0,R21:R20   ; (2) Move the result to the destination registers
; Total: 4 cycles
```

Signed multiplication of two numbers in 1.15 format (registers R23:R22 and R21:R20) with result in 1.31 format in registers R19:R18:R17:R16 looks as follows:

```
CLR    R2              ; (1)
FMULS  R23,R21         ; (2) ((signed)ah*(signed)bh)<<1
MOVW   R19:R18,R1:R0   ; (1)
FMUL   R22,R20         ; (2) (al*bl)<<1
ADC    R18,R2          ; (1)
MOVW   R17:R16,R1:R0   ; (1)
FMULSU R23,R20         ; (2) ((signed)ah*bl)<<1
```

16x16->48-bit unsigned vector MAC			16x16->48-bit signed vector MAC		
Initialization					
CLR	R2	; (1)			
...			...		
LD	R22,X+	; (2)	LD	R22,X+	; (2)
LD	R23,X+	; (2)	LD	R23,X+	; (2)
LD	R20,Y+	; (2)	LD	R20,Y+	; (2)
LD	R21,Y+	; (2)	LD	R21,Y+	; (2)
MUL	R23,R21	; (2) ah*bh	MULS	R23,R21	; (2) (signed)ah*(signed)bh
			SBC	R18,R2	; (1)
			SBC	R19,R2	; (1)
MOVW	R5:R4,R1:R0	; (1)	MOVW	R5:R4,R1:R0	; (1)
MUL	R22,R20	; (2) al*bl	MUL	R22,R20	; (2) al*bl
ADD	R14,R0	; (1)	ADD	R14,R0	; (1)
ADC	R15,R1	; (1)	ADC	R15,R1	; (1)
ADC	R16,R4	; (1)	ADC	R16,R4	; (1)
ADC	R17,R5	; (1)	ADC	R17,R5	; (1)
ADC	R18,R2	; (1)	ADC	R18,R2	; (1)
ADC	R19,R2	; (1)	ADC	R19,R2	; (1)
MUL	R23,R20	; (2) ah*bl	MULSU	R23,R20	; (2) (signed)ah*bl
			SBC	R17,R2	; (1)
			SBC	R18,R2	; (1)
			SBC	R19,R2	; (1)
ADD	R15,R0	; (1)	ADD	R15,R0	; (1)
ADC	R16,R1	; (1)	ADC	R16,R1	; (1)
ADC	R17,R2	; (1)	ADC	R17,R2	; (1)
ADC	R18,R2	; (1)	ADC	R18,R2	; (1)
ADC	R19,R2	; (1)	ADC	R19,R2	; (1)
MUL	R21,R22	; (2) al*bh	MULSU	R21,R22	; (2) al*(signed)bh
			SBC	R17,R2	; (1)
			SBC	R18,R2	; (1)
			SBC	R19,R2	; (1)
ADD	R15,R0	; (1)	ADD	R15,R0	; (1)
ADC	R16,R1	; (1)	ADC	R16,R1	; (1)
ADC	R17,R2	; (1)	ADC	R17,R2	; (1)
ADC	R18,R2	; (1)	ADC	R18,R2	; (1)
ADC	R19,R2	; (1)	ADC	R19,R2	; (1)
...			...		
; Total: 33*N cycles			; Total: 41*N cycles		

Table 16: Implementation of 16x16->48-bit unsigned and signed vector MAC. Operands are in registers R23:R22 and R21:R20; result is in registers R19:R18:R17:R16:R15:R14.

```

SBC    R19,R2        ; (1)
ADD    R17,R0        ; (1)
ADC    R18,R1        ; (1)
ADC    R19,R2        ; (1)
FMULSU R21,R22      ; (2) (al*(signed)bh)<<1
SBC    R19,R2        ; (1)
ADD    R17,R0        ; (1)
ADC    R18,R1        ; (1)
ADC    R19,R2        ; (1)
; Total: 20 cycles

```

If a result in format 1.7 is desired the same code applies but only registers R19:R18 are kept.

3.6 Summary

The number of clock cycles for a few multiplication and MAC operations are summarized in tables 17 and 18 respectively.

In the implementations presented here register saving is not taken into account. For instance

Operation	Clock cycles
8x8->16 signed/unsigned MULT. Operands and result in registers	3
8x8->16 signed/unsigned MULT. Operands and result in memory	10
16x16->32 unsigned MULT. Operands and results in registers	17
16x16->32 unsigned MULT. Operands and results in memory	33
16x16->32 signed MULT. Operands and results in registers	19
16x16->32 signed MULT. Operands and results in memory	35
1.7x1.7->1.15 signed fixed-point multiply. Operands and results in registers	4
1.15x1.15->1.31 signed fixed-point multiply. Operands and results in registers	20

Table 17: Clock cycles for a few multiplication operations on the AVR8 micro-controller.

Operation	Clock cycles
8x8->16 signed/unsigned MAC. Operands and results in registers	4
8x8->16 signed/unsigned MAC. Operands and results in memory	12
8x8->32 unsigned MAC. Operands and results in registers	7
8x8->32 unsigned MAC. Operands and results in memory	19
8x8->32 signed MAC. Operands and results in registers	9
8x8->32 signed MAC. Operands and results in memory	21
16x16->32 unsigned MAC. Operands and results in registers	20
16x16->32 unsigned MAC. Operands and results in memory	36
16x16->32 signed MAC. Operands and results in registers	22
16x16->32 signed MAC. Operands and results in memory	38
16x16->48 unsigned MAC. Operands and results in registers	26
16x16->48 unsigned MAC. Operands and results in memory	46
16x16->48 signed MAC. Operands and results in registers	34
16x16->48 signed MAC. Operands and results in memory	54
8x8->32 vector unsigned MAC. Operand in memory, result registers	$N \cdot 10$
8x8->32 vector signed MAC. Operand in memory, result in registers	$N \cdot 12$
16x16->32 vector unsigned MAC. Operand in memory, result in registers	$N \cdot 27$
16x16->32 vector signed MAC. Operand in memory, result in registers	$N \cdot 29$
16x16->48 vector unsigned MAC. Operand in memory, result in registers	$N \cdot 33$
16x16->48 vector unsigned MAC. Operand in memory, result in registers	$N \cdot 41$

Table 18: Clock cycles for a few isolated MAC operations (top) and vector MAC operations (bottom) on the AVR8 micro-controller. N represents the length of the vector used for vector MAC operations.

registers $R1:R0$ are modified by the multiplication instructions. Register $R2$ is cleared and registers $R5:R4$ are used as temporary variables. We assumed that the time to save these registers is negligible in comparison to the time spent performing arithmetic operations.

Loop unrolling is used to implement vector MAC. In comparison to the time spent performing the MAC operations, in particular with 16-bit operands, an implementation using a loop would introduce a comparatively small overhead.

4 PIC18

The PIC18 [5] is an 8-bit micro-controller with a single accumulator register. As such it differs significantly from the MSP430 or AVR8 architectures which have a register file. Results of instructions are stored in the accumulator, or in memory, depending on the opcode. Access to memory with PIC18 instructions is taking the same number of clock cycles as access to the accumulator register. As such, from the instruction execution time, the memory can be seen as a large register file.

The PIC18 memory contains General Purpose Registers (in other words general purpose RAM) and Special Function Registers (SFRs). SFRs include memory mapped peripherals and special registers that affect memory addressing. Memory access is partitioned in banks of 256 bytes and a maximum of 16 banks is available (RAM is in bank 0 to 14, SFRs are in bank 15). The PIC18 ALU is only capable of direct memory addressing within a bank (8-bit address). Therefore, to directly access the whole memory, the corresponding memory bank must first be selected by writing to the bank select SFR. Other type of memory addressing (e.g. indirect addressing) is done using SFRs referred to as file select registers. Three file select registers are available. They contain a 12-bit memory address and therefore can address the whole CPU address space. By then reading or writing a specific SFR the content of the memory pointed by the file select register is accessed.

The *access bank* is a “virtual” bank that is convenient to access both data memory and the special function registers. Selection of the access bank is done by a specific bit in the opcode. The lower partition (96 bytes) of the the access bank map to the first bytes of data memory. The higher partition (160 bytes) maps to the special function registers. Using the access bank allows both to access data memory and the SFR. If the access bank is not used, the bank corresponding to the desired memory range or SFRs must be selected beforehand with the instruction `MOVLB` (instruction `MOVLB` moves an immediate value to the bank select register).

Two instructions allow to move data between the memory and the work registers. Instruction `MOVWF` provides 256 bytes addressing within the currently selected memory bank and operates in one clock cycle. Instruction `MOVFF` provides 12-bit flat memory addressing, and can be used to copy any memory location to any other, or to the work register. It executes in two clock cycles.

In this paper we look at optimized implementations of arithmetic functions. Operands and result are stored in the access bank so that they can be read with the single cycle `MOVWF` instruction. If the operands are in different banks then bank switching must be implemented, or the slower `MOVFF` instruction must be used to read data with flat memory addressing, with the corresponding increase in execution time.

The PIC18 memory is also referred to as a “register file” since all access to those memory locations is done in 1 clock cycle. In this document we refer to it as “memory” and leave the term “register” for the work register of the CPU. For comparison purposes, this memory can however be considered as a large register set. This may be the case when comparing PIC18 arithmetic functions with those implemented on architectures that do use registers e.g. to store results.

The PIC18 has a hardware 8x8->16-bit unsigned multiplier that operates in a single cycle (instruction `MULWF`). The result of hardware multiplication is stored in product registers (`PRODH:PRODL`) which are mapped to memory (i.e. in SFRs).

The PIC18 architecture distinguishes the system clock from the instruction clock. The system clock corresponds to the signal that is applied to the clock input pin of the device. For architectural reasons, the instruction clock is four times slower than the system clock. In this document we report the number of clock cycles in terms of “instruction clock cycles”. The number of system clock cycles is four times larger. The PIC18 however offers various clocking schemes (e.g. crystal, oscillator, RC oscillator) and in particular it has a crystal/resonator mode with a 4x PLL. When this clocking scheme is selected, the system clock frequency is internally multiplied by 4. As a consequence, the frequency of the system clock and instruction clocks are identical when this mode is selected.

Below the implementation of the arithmetic operations are described (some implementations are taken from [5]). Integer multiplication, integer MAC and fixed-point multiplication are described in this order. First operations with 8-bit operands are considered since they have hardware support

8x8->16-bit signed/unsigned MULT	
Unsigned multiplication	Signed multiplication
MOVF op1,W ; (1)	MOVF op1,W ; (1)
MULWF op2 ; (1) PRODH:PRODL=op1*op2	MULWF op2 ; (1) PRODH:PRODL=op1*op2
	BTFSC op2,7 ; (1-2) Test sign
	SUBWF PRODH,F ; (1) PRODH=PRODH-op1
	MOVF op2,W ; (1)
	BTFSC op1,7 ; (1-2) Test sign
	SUBWF PRODH,F ; (1) PRODH=PRODH-op2
Result in PRODH:PRODL	Result in memory
-	MOVFF PRODH,ResH ; (2)
	MOVFF PRODL,ResL ; (2)

Table 19: Summary of 8-bit unsigned and signed multiplication operations on the PIC18.

on the PIC18; operations on 16-bit operands are considered afterward.

4.1 8x8-bit multiplication, 16-bit result

Table 19 show unsigned and signed 8x8->16-bit multiplication. Operands are fetched from memory locations `op1` and `op2`. The result is either left in the hardware multiplier (`PRODH:PRODL`) or stored to memory locations `ResH:ResL`. The signed multiplication requires conditional subtraction when an operand is negative. The resulting number of clock cycles is however identical regardless of the sign of the operands because the “bit test and skip if clear instruction” (`BTFSC`) executes in 1 clock cycles when the single clock cycle subtraction `SUBWF` is executed, and it executes in 2 clock cycles when `SUBWF` is skipped. The number of clock cycles required for the sequence of these two instructions is therefore always 2.

The `MOVF` instruction is used to fetch the first operand from the current memory bank. To fetch this operand from any memory location the instruction `MOVFF` may be used instead (2 clock cycles), or the appropriate memory bank needs to be selected beforehand (instruction `MOVLB`). The second operand is fetched by instruction `MULWF`; it must reside in the currently selected bank.

4.2 16x16-bit multiplication, 32-bit result

The 16x16->32-bit unsigned and signed multiplication of operands `op1h:op1l` and `op2h:op2l` with result in memory locations `RES3:RES2:RES1:RES0` is illustrated in table 20. The multiplication is decomposed in elementary 8x8-bit multiplications that can be computed with `MULWF`. The number of clock cycles for the signed operation depends on the sign of the operands and thus the minimum and maximum number of clock cycles that the multiplication may take is indicated.

4.3 8-bit MAC

Table 21 illustrates unsigned and signed 8x8->32-bit MAC on the PIC18. The operands and the result are assumed to be in the access bank. The signed and unsigned 8x8->16-bit MAC is similar with the exception that instructions doing the accumulation in `RES3:RES2` (together with 32-bit sign extension for the signed case) are removed. The number of clock cycles for the unsigned and signed 8x8->16-bit MAC are respectively 6 and 11 clock cycles.

Vector MAC is implemented with indirect memory addressing to load the operands of the MAC operation. Indirect memory access with absolute addresses is done using the file select registers. We assume that the access bank is used, which is partitioned in data memory and special function registers, so that data memory (for temporary variables) and special functions registers (for indirect memory access) can be both accessed without costly bank switching operations.

The special function registers `FSR0` and `FSR1` (file select registers) point to the start of the two vectors to multiply-accumulate. Data is accessed by reading `INDF0` and `INDF1`, which contain the content of the memory locations pointed by `FSR0` and `FSR1` respectively. `FSRx` is incremented

8x8->32-bit unsigned MAC	8x8->32-bit signed MAC
<pre> ; Multiply MOVWF op0,W ; (1) MULWF op1 ; (1) PRODH:PRODL=op1*op2 ; Accumulate MOVWF PRODL,W ; (1) ADDWF RES0,F ; (1) MOVWF PRODH,F ; (1) ADDWFC RES1,F ; (1) CLRF WREG ; (1) ADDWFC RES2,F ; (1) ADDWFC RES3,F ; (1) ; Total: 9 clock cycles </pre>	<pre> ; Multiply MOVWF op0,W ; (1) MULWF op1 ; (1) PRODH:PRODL=op1*op2 BTFSC op1,7 ; (1-2) Test sign SUBWF PRODH,F ; (1) PRODH=PRODH-op1 MOVWF op1,W ; (1) BTFSC op0,7 ; (1-2) Test sign SUBWF PRODH,F ; (1) PRODH=PRODH-op2 ; Sign extend and accumulate SETF STATUS ; (1) Set carry BTFSC PRODH,7 ; (1-2) Test sign DECF RES2,F ; (1) RES2=RES2-1 BTFSS STATUS,C ; (1-2) Test carry DECF RES3,F ; (1) RES3=RES3-1 MOVWF PRODL,W ; (1) ADDWF RES0,F ; (1) MOVWF PRODH,F ; (1) ADDWFC RES1,F ; (1) CLRF WREG ; (1) ADDWFC RES2,F ; (1) ADDWFC RES3,F ; (1) ; Total: 19 clock cycles </pre>

Table 21: 8x8->32-bit MAC operations on the PIC18.

8x8->32-bit unsigned vector MAC	8x8->32-bit signed vector MAC
<pre> ... ; Load operands and multiply MOVWF INDF0,W ; (1) Operand 1 in W MULWF INDF1 ; (1) ; Accumulate MOVWF PRODL,W ; (1) ADDWF RES0,F ; (1) MOVWF PRODH,W ; (1) ADDWFC RES1,F ; (1) CLRF WREG ; (1) ADDWFC RES2,F ; (1) ADDWFC RES3,F ; (1) ; Increment pointers MOVWF POSTINC0,W ; (1) MOVWF POSTINC1,W ; (1) ... ; Total: 14*N clock cycles </pre>	<pre> ... ; Load operands and multiply MOVWF INDF0,W ; (1) Operand 1 in W MULWF INDF1 ; (1) BTFSC INDF1,7 ; (1-2) Test sign SUBWF PRODH,F ; (1) PRODH=PRODH-op1 MOVWF INDF1,W ; (1) BTFSC INDF0,7 ; (1-2) Test sign SUBWF PRODH,F ; (1) PRODH=PRODH-op2 ; Sign extend and accumulate SETF STATUS ; (1) Set carry BTFSC PRODH,7 ; (1-2) Test sign DECF RES2,F ; (1) RES2=RES2-1 BTFSS STATUS,C ; (1-2) Test carry DECF RES3,F ; (1) RES3=RES3-1 MOVWF PRODL,W ; (1) ADDWF RES0,F ; (1) MOVWF PRODH,W ; (1) ADDWFC RES1,F ; (1) CLRF WREG ; (1) ADDWFC RES2,F ; (1) ADDWFC RES3,F ; (1) ; Increment pointers MOVWF POSTINC0,W ; (1) MOVWF POSTINC1,W ; (1) ... ; Total: 24*N clock cycles </pre>

Table 22: 8x8->32-bit vector MAC operations on the PIC18.

16x16->48-bit unsigned MAC	16x16->48-bit signed MAC
<pre> ; Multiply in temporary MOVWF op1l,W ; (1) MULWF op2l ; (1) op1l*op2l MOVFF PRODH,TMP1 ; (2) MOVFF PRODL,TMP0 ; (2) MOVF op1h,W ; (1) MULWF op2h ; (1) op1h*op2h MOVFF PRODH,TMP3 ; (2) MOVFF PRODL,TMP2 ; (2) MOVF op1l,W ; (1) MULWF op2h ; (1) op1l*op2h MOVF PRODL,W ; (1) ADDFW TMP1,F ; (1) MOVF PRODH,W ; (1) ADDFWC TMP2,F ; (1) CLRF WREG ; (1) ADDFWC TMP3,F ; (1) MOVF op1h,W ; (1) MULWF op2l ; (1) op1h*op2l MOVF PRODL,W ; (1) ADDFW TMP1,F ; (1) MOVF PRODH,W ; (1) ADDFWC TMP2,F ; (1) CLRF WREG ; (1) ADDFWC TMP3,F ; (1) ; Accumulate MOVF TMP0,W ; (1) ADDFW RES0,F ; (1) MOVF TMP1,W ; (1) ADDFWC RES1,F ; (1) MOVF TMP2,W ; (1) ADDFWC RES2,F ; (1) MOVF TMP3,W ; (1) ADDFWC RES3,F ; (1) CLRF WREG ; (1) ADDFWC RES4,F ; (1) ADDFWC RES5,F ; (1) ; Total: 39 clock cycles </pre>	<pre> ; Multiply in temporary MOVF op1l,W ; (1) MULWF op2l ; (1) op1l*op2l MOVFF PRODH,TMP1 ; (2) MOVFF PRODL,TMP0 ; (2) MOVF op1h,W ; (1) MULWF op2h ; (1) op1h*op2h MOVFF PRODH,TMP3 ; (2) MOVFF PRODL,TMP2 ; (2) MOVF op1l,W ; (1) MULWF op2h ; (1) op1l*op2h MOVF PRODL,W ; (1) ADDFW TMP1,F ; (1) MOVF PRODH,W ; (1) ADDFWC TMP2,F ; (1) CLRF WREG ; (1) ADDFWC TMP3,F ; (1) ; Sign correction BTFSS op2h,7 ; (1-2) GOTO sgn1 ; (2) MOVF op1l,W ; (1) SUBWF TMP2 ; (1) MOVF op1h,W ; (1) SUBWFB TMP3 ; (1) sgn1 BTFSS op1h,7 ; (1-2) GOTO sgn2 ; (2) MOVF op2l,W ; (1) SUBWF TMP2 ; (1) MOVF op2h,W ; (1) SUBWFB TMP3 ; (1) sgn2 ; Sign extend for accumulation BTFSS TMP3,7 ; (1-2) Test sign GOTO pos ; (2) CLRF WREG ; (1) DECF RES4,F ; (1) SUBWFB RES5,F ; (1) pos ; Accumulate MOVF TMP0,W ; (1) ADDFW RES0,F ; (1) MOVF TMP1,W ; (1) ADDFWC RES1,F ; (1) MOVF TMP2,W ; (1) ADDFWC RES2,F ; (1) MOVF TMP3,W ; (1) ADDFWC RES3,F ; (1) CLRF WREG ; (1) ADDFWC RES4,F ; (1) ADDFWC RES5,F ; (1) ; Minimum/Maximum: 48/54 clock cycles </pre>

Table 24: 16x16->48-bit MAC operations on the PIC18.

Vector 16x16->32-bit MAC is illustrated in table 25. Indirect memory addressing is done in the same way as the 8x8->16-bit MAC, with source operands pointed to by `FSR0` and `FSR1`. The higher 8-bits of the operands are copied in `op1h` and `op2h` with indirect addressing with post-increment. The lower 8-bits are directly accessed via `INFO` and `INDF1` during the MAC operation. At the end of the MAC operation the pointers are incremented.⁵

Vector 16x16->48-bit MAC is illustrated in table 26.

4.5 Fixed-point operations

Fixed-point signed multiplication is identical to the integer signed multiplication with the addition of a left-shift operation to align the decimal point. Fixed-point signed 1.7x1.7->1.15 and 1.15x1.15->1.31 multiplication is illustrated in table 27. Operands are loaded from memory and the result is stored to memory.

4.6 Summary

The number of clock cycles for a few multiplication and MAC operations are summarized in tables 28 and 29 respectively.

⁵The whole 16-bit operands could be copied into local variables `op1h:op1l` and `op2h:op2l` at the start of the MAC operation with indirect addressing with post-increment. However this implementation would require two additional clock-cycles.

16x16->32-bit unsigned vector MAC	16x16->32-bit signed vector MAC
<pre> ... ; Load the higher byte of operands MOVFF POSTINC0,op1h ; (2) MOVFF POSTINC1,op2h ; (2) ; Multiply MOVF INDF0,W ; (1) MULWF INDF1 ; (1) op1*op2l MOVF PRODL,W ; (1) Accumulate op1*op2l ADDWF RES0 ; (1) MOVF PRODH,W ; (1) ADDWFC RES1 ; (1) CLRF WREG ; (1) ADDWFC RES2 ; (1) ADDWFC RES3 ; (1) MOVF op1h,W ; (1) MULWF op2h ; (1) op1*op2h MOVF PRODL,W ; (1) Accumulate op1*op2h ADDWF RES2 ; (1) MOVF PRODH,W ; (1) ADDWFC RES3 ; (1) MOVF INDF0,W ; (1) MULWF op2h ; (1) op1*op2h MOVF PRODL,W ; (1) Accumulate op1*op2h ADDWF RES1,F ; (1) MOVF PRODH,W ; (1) ADDWFC RES2,F ; (1) CLRF WREG ; (1) ADDWFC RES3,F ; (1) MOVF op1h,W ; (1) MULWF INDF1 ; (1) op1*op2l MOVF PRODL,W ; (1) Accumulate op1*op2l ADDWF RES1,F ; (1) MOVF PRODH,W ; (1) ADDWFC RES2,F ; (1) CLRF WREG ; (1) ADDWFC RES3,F ; (1) ; Increment pointers MOVF POSTINC0,W ; (1) MOVF POSTINC1,W ; (1) ... ; Total: 37*N clock cycles </pre>	<pre> ... ; Load the higher byte of operands MOVFF POSTINC0,op1h ; (2) MOVFF POSTINC1,op2h ; (2) ; Multiply MOVF INDF0,W ; (1) MULWF INDF1 ; (1) op1*op2l MOVF PRODL,W ; (1) Accumulate op1*op2l ADDWF RES0 ; (1) MOVF PRODH,W ; (1) ADDWFC RES1 ; (1) CLRF WREG ; (1) ADDWFC RES2 ; (1) ADDWFC RES3 ; (1) MOVF op1h,W ; (1) MULWF op2h ; (1) op1*op2h MOVF PRODL,W ; (1) Accumulate op1*op2h ADDWF RES2 ; (1) MOVF PRODH,W ; (1) ADDWFC RES3 ; (1) MOVF INDF0,W ; (1) MULWF op2h ; (1) op1*op2h MOVF PRODL,W ; (1) Accumulate op1*op2h ADDWF RES1,F ; (1) MOVF PRODH,W ; (1) ADDWFC RES2,F ; (1) CLRF WREG ; (1) ADDWFC RES3,F ; (1) MOVF op1h,W ; (1) MULWF INDF1 ; (1) op1*op2l MOVF PRODL,W ; (1) Accumulate op1*op2l ADDWF RES1,F ; (1) MOVF PRODH,W ; (1) ADDWFC RES2,F ; (1) CLRF WREG ; (1) ADDWFC RES3,F ; (1) ; Sign correction BTFSS op2h,7 ; (1-2) GOTO sgn1 ; (2) MOVF INDF0,W ; (1) SUBWF RES2 ; (1) MOVF op1h,W ; (1) SUBWFB RES3 ; (1) sgn1 BTFSS op1h,7 ; (1-2) GOTO sgn2 ; (2) MOVF INDF1,W ; (1) SUBWF RES2 ; (1) MOVF op2h,W ; (1) SUBWFB RES3 ; (1) sgn2 ; Increment pointers MOVF POSTINC0,W ; (1) MOVF POSTINC1,W ; (1) ... ; Minimum/Maximum: 43*N/49*N clock cycles </pre>

Table 25: 16x16->32-bit vector MAC operations on the PIC18.

16x16->48-bit vector unsigned MAC	16x16->48-bit vector signed MAC
<pre> ... ; Load the higher byte of operands MOVFF POSTINCO,opih ; (2) MOVFF POSTINC1,op2h ; (2) ; Multiply in temporary MOVF INDF0,W ; (1) MULWF INDF1 ; (1) op11*op21 MOVFF PRODH,TMP1 ; (2) MOVFF PRODL,TMP0 ; (2) MOVF opih,W ; (1) MULWF op2h ; (1) op1h*op2h MOVFF PRODH,TMP3 ; (2) MOVFF PRODL,TMP2 ; (2) MOVF INDF0,W ; (1) MULWF op2h ; (1) op11*op2h MOVF PRODL,W ; (1) ADDWF TMP1,F ; (1) MOVF PRODH,W ; (1) ADDWFC TMP2,F ; (1) CLRF WREG ; (1) ADDWFC TMP3,F ; (1) MOVF opih,W ; (1) MULWF INDF1 ; (1) op1h*op21 MOVF PRODL,W ; (1) ADDWF TMP1,F ; (1) MOVF PRODH,W ; (1) ADDWFC TMP2,F ; (1) CLRF WREG ; (1) ADDWFC TMP3,F ; (1) ; Accumulate MOVF TMP0,W ; (1) ADDWF RES0,F ; (1) MOVF TMP1,W ; (1) ADDWFC RES1,F ; (1) MOVF TMP2,W ; (1) ADDWFC RES2,F ; (1) MOVF TMP3,W ; (1) ADDWFC RES3,F ; (1) CLRF WREG ; (1) ADDWFC RES4,F ; (1) ADDWFC RES5,F ; (1) ; Increment pointers MOVF POSTINCO,W ; (1) MOVF POSTINC1,W ; (1) ... ; Total: 45*N clock cycles </pre>	<pre> ... ; Load the higher byte of operands MOVFF POSTINCO,opih ; (2) MOVFF POSTINC1,op2h ; (2) ; Multiply in temporary MOVF INDF0,W ; (1) MULWF INDF1 ; (1) op11*op21 MOVFF PRODH,TMP1 ; (2) MOVFF PRODL,TMP0 ; (2) MOVF opih,W ; (1) MULWF op2h ; (1) op1h*op2h MOVFF PRODH,TMP3 ; (2) MOVFF PRODL,TMP2 ; (2) MOVF INDF0,W ; (1) MULWF op2h ; (1) op11*op2h MOVF PRODL,W ; (1) ADDWF TMP1,F ; (1) MOVF PRODH,W ; (1) ADDWFC TMP2,F ; (1) CLRF WREG ; (1) ADDWFC TMP3,F ; (1) MOVF opih,W ; (1) MULWF INDF1 ; (1) op1h*op21 MOVF PRODL,W ; (1) ADDWF TMP1,F ; (1) MOVF PRODH,W ; (1) ADDWFC TMP2,F ; (1) CLRF WREG ; (1) ADDWFC TMP3,F ; (1) ; Sign correction BTFSF op2h,7 ; (1-2) GOTO sgn1 ; (2) MOVF INDF0,W ; (1) SUBWF TMP2 ; (1) MOVF opih,W ; (1) SUBWFB TMP3 ; (1) sgn1 BTFSF op1h,7 ; (1-2) GOTO sgn2 ; (2) MOVF INDF1,W ; (1) SUBWF TMP2 ; (1) MOVF op2h,W ; (1) SUBWFB TMP3 ; (1) sgn2 ; Sign extend for accumulation BTFSF TMP3,7 ; (1-2) Test sign GOTO pos ; (2) CLRF WREG ; (1) DECF RES4,F ; (1) SUBWFB RES5,F ; (1) pos ; Accumulate MOVF TMP0,W ; (1) ADDWF RES0,F ; (1) MOVF TMP1,W ; (1) ADDWFC RES1,F ; (1) MOVF TMP2,W ; (1) ADDWFC RES2,F ; (1) MOVF TMP3,W ; (1) ADDWFC RES3,F ; (1) CLRF WREG ; (1) ADDWFC RES4,F ; (1) ADDWFC RES5,F ; (1) ; Increment pointers MOVF POSTINCO,W ; (1) MOVF POSTINC1,W ; (1) ... ; Minimum/Maximum: 54/60 clock cycles </pre>

Table 26: 16x16->48-bit vector MAC operations on the PIC18.

Operation	Clock cycles
8x8->16 unsigned MAC. Operands and result in memory	6
8x8->16 signed MAC. Operands and result in memory	11
8x8->32 unsigned MAC. Operands and results in memory	9
8x8->32 signed MAC. Operands and results in memory	19
16x16->32 unsigned MAC. Operands and result in memory	31
16x16->32 signed MAC. Operands and result in memory	37/43
16x16->48 unsigned MAC. Operands and results in memory	39
16x16->48 signed MAC. Operands and results in memory	48/54
8x8->32 vector unsigned MAC. Operand in memory, result in memory	$N \cdot 11$
8x8->32 vector signed MAC. Operand in memory, result in memory	$N \cdot 21$
16x16->32 vector unsigned MAC. Operands and result in memory	$N \cdot 37$
16x16->32 vector signed MAC. Operands and result in memory	$N \cdot 43/49$
16x16->48 vector unsigned MAC. Operands and results in memory	$N \cdot 47$
16x16->48 vector signed MAC. Operands and results in memory	$N \cdot 56/62$

Table 29: Clock cycles for a few isolated MAC operations (top) and vector MAC operations (bottom) on the PIC18 micro-controller. N represents the length of the vector used for vector MAC operations.

5 Discussion

In this comparison we selected micro-controllers disposing of hardware multiplication unit to efficiently implement operations commonly used in signal processing. Micro-controllers that do not have hardware multiplication must rely on a software implementation, typically carried out with shift and add operations. As a consequence the execution time for the operations considered in this document would be significantly longer. Micro-controllers not disposing of hardware multiplication may only be suited when the computational requirements are limited (simple algorithms or no need for fast execution).

This review is far from exhaustive since there are other architectures with similar features and that could be included for comparison. However, we limited this review to the most commonly used architectures in sensor nodes and embedded systems. Among other architectures, the 8051 architecture is also found in sensor nodes⁶ and also disposes of hardware support for multiplications.

Power consumption aspects have not been included here. This aspect was investigated in [10], where a comparison of the energy used to execute an instruction was made across different processors.

In this document we assumed that the arithmetic operations that we compared were in innermost loops of programs (i.e. we assumed that they were the time-critical part of the program). This is true e.g. for filtering applications (a finite impulse filter is implemented as a vector MAC), but may not be case for general purpose programs. Therefore, this document may not allow to compare micro-controller performance at executing general purpose programs or programs that make only moderate use of the arithmetic functions considered here. In the latter case, operations such as execution flow control, function calls, register allocation, and interrupts must also be taken into account. Comparing micro-controller performance in this case may be best done by running a common benchmark on all the systems. Furthermore, if there are no time-critical routines in the program (i.e. if the time spent in the various routines of the program is similar) then a compiler may be better at optimizing the program performance than manual assembler optimizations of hand-picked routines.

While a lot of effort has been put to compare optimized code on the various architectures, it is not excluded that there are still further optimizations possible of the individual operations. Also, further improvements may be possible depending on the particular application specificities (e.g. register reuse to avoid storing results to memory).

6 Summary and conclusion

Table 30 reports the execution time for the multiplication operations on the three architectures considered in this document; table 31 reports the execution time for the MAC operations. Since the architecture of these micro-controllers differs significantly (number of registers, multiplication instruction or memory mapped peripheral), only operations that can be meaningfully compared are reported in these tables.

The operands may come from memory or registers and the results may be left in the dedicated multiplication hardware⁷, or stored in registers or in memory. The tables report typically the fastest implementation where operands and results are in registers, and the slowest implementation where operands and results are in memory. The exception is the PIC18 architecture which has no registers. In this case operands and results are in memory. The PIC18 memory is however accessed in a single cycle and for comparison purposes it may be considered as if it was the equivalent of a register in other architectures.

Vector MAC requires fast MAC execution since one MAC operation is executed for each element of the vectors. Therefore timings for the fastest implementations are indicated, which often use

⁶The CC2430 [8] radio chip from Chipcon/Texas Instruments includes a 8051 micro-controller with flash and RAM which makes it suited for single-chip sensor nodes.

⁷On the MSP430 results (typically accumulation results) may be left in the memory-mapped hardware MAC unit. On the PIC18 multiplication results can be left or accumulated in memory-mapped product registers.

	MSP430	AVR8	PIC18
Operation	Clock cycles		
8x8->16 unsigned MULT. Operands and result in registers	11	3	-
8x8->16 unsigned MULT. Operands and result in memory	18	10	6
8x8->16 signed MULT. Operands and result in registers	15	3	-
8x8->16 signed MULT. Operands and result in memory	22	10	11
16x16->32 unsigned MULT. Operands and results in registers	11	17	-
16x16->32 unsigned MULT. Operands and results in memory	24	33	28
16x16->32 signed MULT. Operands and results in registers	22	19	-
16x16->32 signed MULT. Operands and results in memory	32	35	34/40
1.7x1.7->1.15 signed fixed-point multiply. Operands and results in registers (PIC18: memory)	16	4	12
1.15x1.15->1.31 signed fixed-point multiply. Operands and results in registers (PIC18: memory)	16	20	39/45

Table 30: Comparison of multiplication operations between MSP430, AVR8 and PIC18 micro-controllers. The operand and results of the fixed-point multiplication on the PIC18 are actually in memory.

	MSP430	AVR8	PIC18
Operation	Clock cycles		
8x8->16 unsigned MAC. Operands and results in registers	11	4	-
8x8->16 unsigned MAC. Operands and results in memory	18	4	6
8x8->16 signed MAC. Operands and results in registers	15	12	-
8x8->16 signed MAC. Operands and results in memory	22	12	11
8x8->32 unsigned MAC. Operands and results in registers	14	7	-
8x8->32 unsigned MAC. Operands and results in memory	24	19	9
8x8->32 signed MAC. Operands and results in registers	18	9	-
8x8->32 signed MAC. Operands and results in memory	28	21	19
16x16->32 unsigned MAC. Operands and results in registers	14	20	-
16x16->32 unsigned MAC. Operands and results in memory	18	36	31
16x16->32 signed MAC. Operands and results in registers	14	22	-
16x16->32 signed MAC. Operands and results in memory	18	38	37/43
16x16->48 unsigned MAC. Operands and results in registers	17	26	-
16x16->48 unsigned MAC. Operands and results in memory	30	46	39
16x16->48 signed MAC. Operands and results in registers	17	34	-
16x16->48 signed MAC. Operands and results in memory	30	54	48/54
8x8->32 vector unsigned MAC. Operands in memory, result in MAC unit / registers / memory	$N \cdot 10$	$N \cdot 10$	$N \cdot 11$
8x8->32 vector signed MAC. Operands in memory, result in registers / memory	$N \cdot 20$	$N \cdot 12$	$N \cdot 21$
16x16->32 vector unsigned MAC. Operands in memory, result in MAC unit / registers / memory	$N \cdot 10$	$N \cdot 27$	$N \cdot 37$
16x16->32 vector signed MAC. Operands in memory, result in registers / memory	$N \cdot 16$	$N \cdot 29$	$N \cdot 43/49$
16x16->48 vector unsigned MAC. Operands in memory, result in registers / memory	$N \cdot 19$	$N \cdot 33$	$N \cdot 45$
16x16->48 vector signed MAC. Operands in memory, result in registers / memory	$N \cdot 19$	$N \cdot 41$	$N \cdot 54/60$

Table 31: Comparison of MAC operations between MSP430, AVR8 and PIC18 micro-controllers. Result of vector MAC on the MSP430 is in the MAC unit (unsigned) or in registers (signed). Results of vector MAC on the AVR8 are in registers. Result of vector MAC on PIC18 is in memory.

registers or even directly the multiplication hardware to accumulate the result (i.e. the time necessary to move the result from the multiplication hardware to registers or memory is considered negligible in comparison to the vector MAC and it is not reported).

The number of clock cycles indicated in the tables directly corresponds to the clock signal applied to the clock pin of the micro-controller, with the exception of the PIC18. The clock cycles reported for the PIC18 are the number of “instruction clock cycles”. Those correspond to clock signal applied to the clock pin only when the PIC18 is clocked with its internal 4x PLL. If the 4x PLL mode is not used, then the number of clock cycles reported here must be multiplied by 4 in order to compare the results with the other architectures.

The three architectures have various advantages and drawbacks. The MSP430 benefits from its 16-bit registers and its hardware MAC unit which can handle 16-bit multiplications with 32-bit results. In addition the hardware MAC provides a “sign extend” register that allows to easily obtain 48-bit results. The hardware MAC unit however does not handle mixed signed/unsigned operands, and signed accumulation. This must be supplemented in software. The MSP430 has also a relatively slow memory access. This affects the performance of the arithmetic operations since the MAC unit is memory mapped. It also affects operations that fetch many operands from memory (e.g. vector MAC).

The AVR8 benefits from a large number of registers and an efficient instruction set with most instructions that execute in a single clock cycle. It has an instruction set that is efficiently designed to handle unsigned, signed and mixed signed/unsigned multiplications. This allows efficient implementation of multiplications with operands larger than 8-bits by decomposing the operation in intermediate 8-bit multiplications. In comparison to the MSP430, the AVR8 has faster memory access. This allows it to fare well compared to the MSP430 when operand fetching and result storage take a significant portion of the arithmetic operation execution time (e.g. in 8x8->16-bit multiplications). The AVR8 is also the only architecture with instructions dedicated to fixed-point multiplications (multiplication and implicit left shift), although these instructions only slightly decrease the multiplication time in comparison a standard multiplication followed by an explicit left shift.

The PIC18 benefits from fast instruction execution time (assuming that the clocking scheme with the internal 4x PLL is used): except branching all instructions execute in a single cycle. The PIC18 multiplication instruction only handles unsigned operands. Signed multiplications require the use conditional execution in the routine which translates in significantly longer execution time in comparison to unsigned multiplications. In addition, the execution time varies depending on the sign of the operands. Therefore the worst-case timing must be taken into account when devising time-critical algorithms. The fast instruction execution time and memory access time allow the PIC18 allow the PIC18 to fare well in some operations compared to the other architectures (e.g. 8x8->32-bit MAC), but the unsigned-only hardware multiplication tends to limit its performance when handling signed arithmetics.

There are two ways to interpret the result that we shown here. On the one hand, the three architectures that were compared are very different (8 and 16 bits, from a single accumulator register up to 32 registers, hardware MAC or multiplication instruction), yet differences in execution time remain below an order of magnitude⁸. These differences may be considered small for such diverse architectures. Algorithmic optimizations may therefore be more efficient than optimizing the implementation itself. An algorithmic optimization allowing a reduction of a factor 4 in the number of arithmetic operations on the slowest architecture may bring it to the level of the fastest architecture for most operations.

On the other hand, algorithmic optimizations may be carried out on all the platforms and this comparison highlights significant differences across the architectures. This review may help devise appropriate algorithms (e.g. selecting the precision of the fixed-point representation) according to the hardware that is available (e.g. commercial wireless sensor nodes). It may also allow to select

⁸Execution time of the slowest architecture is in most cases less than 4 times slower than the fastest one. The largest differences in execution speed are found between the MSP430 and AVR8 in the signed 8x8->16-bit multiplication.

the most suited architecture for custom hardware if the algorithms are already devised.

Signal processing tasks running on low-power micro-controllers are likely to get more complex in the future. In this review we selected micro-controllers that are commonly used in wireless sensor nodes. However new generation on micro-controllers are continuously introduced. Some of them target specifically signal processing applications, yet they remain in the low-power range. Among them, the dsPIC of Microchip and the ARM7 Thumb are very interesting architectures to implement signal processing applications on low-power systems and they may be worth considering for newer systems.

References

- [1] Atmel. *AVR201: Using the AVR Hardware Multiplier*. Atmel, San Jose, CA, 2002.
- [2] Atmel. *AVR202: 16-bit Arithmetics*. Atmel, San Jose, CA, 2002.
- [3] Atmel. *AVR Instruction Set*. Atmel, San Jose, CA, 2005.
- [4] D. Culler, D. Estrin, and M. Srivastava. Overview of sensor networks. *Computer*, 37(8):41–49, 2004.
- [5] Microchip. *PICmicro 18C MCU Family Reference Manual*. Microchip, Chandler, AZ, 2000.
- [6] D. Puccinelli and M. Haenggi. Wireless sensor networks: applications and challenges of ubiquitous sensing. *IEEE Circuits and Systems Magazine*, 5(3):19–31, 2005.
- [7] Texas Instruments. *Application Report: The MSP430 Hardware Multiplier*. Texas Instruments, Dallas, TX, 1999.
- [8] Texas Instruments. *CC2430 Data Sheet*. Texas Instrument, Dallas, TX, 2006.
- [9] Texas Instruments. *MSP430x2xx Family User's Guide (Rev. B)*. Texas Instruments, Dallas, TX, 2006.
- [10] M. A. M. Vieira, C. N. Coelho, D. C. da Silva, and J. M. da Mata. Survey on wireless sensor network devices. In *Proceedings of IEEE Conference on Emerging Technologies and Factory Automation*, pages 537–544, 2003.