XA benchmark vs. the MCS251 AN705

BACKGROUND

A computer benchmark is a “program” that is used to determine relative computer core performance by evaluating benchmark execution time of the core. In a brainstorm sessionon microcontrollers for automotive applications, an assembler functional *benchmark for engine management*, which is a typical example of embedded high-end microcontrol was created. This report summarizes the functions implemented in assembler language of the compared controllers: Intel MCS251, and Philips XA. The total execution times of a program “engine cycle” (engine stroke) are calculated and the required program code is estimated for each controller.

Evaluation of performance in a High Level Language (HLL) like C would be preferable, but it is difficult to realize as “the best” compilers for all cores involved then should be used.

This document outlines code density and execution times of the XA, based on the most recent information. The execution times are given in terms of both clock cycles and time units. Although the XA can run at a much higher speed than the MCS251, for the sake of fairness, both cores are evaluated running at 16.00 MHz. This is a reasonable assumption for comparing the cores at the same level of technology.

Because of the pipeline architectures of the MCS251 and the XA, the benchmarks are run on actual silicon.

BENCHMARK RESULTS AND CONCLUSIONS

Relative performance on a line

The table below presents the most important result of the assembler benchmark evaluation. It pictures the relative performance of the compared core instruction set on a scale where XA=1.0. Also appended is the performance charts–execution and code density of all the processors.

Total exec.times/core(µs) for all routines (with *occurrences)
938.75 359.86

Performance ratio	MCS251	XA
MCS251	1.0	2.61
XA	0.383	1.0

Table 1. XA instruction set execution times and bytes/function

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.0	12	18
ADD/SUB	50	0.375	18.75	4
CMP 24b	13	1.25	16.25	9
CAN 16b	80	0.562	44.96	5
INTPLIN	20	2.04	40.8	42
BRANCH	1		158.13	

XA totals : 299.89 µs
including 20% statistics : 359.86 µs

Table 2. MCS251 instruction set execution times and bytes/function

FUNCTION	OC*	MCS251		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	1.53	18.36	2
FDIV	4	30.125	120.6	25
ADD/SUB	50	0.641	32.05	2
CMP 24b	13	3.375	43.88	12
CAN 16b	80	1.625	130	6
INTPLIN	20	6.12	122.4	60
BRANCH	1		315.0	

MCS251 totals : 782.29 µs
including 20% statistics : 938.75 µs

Table 3. Total benchmark execution time results

MICROCONTROLLER CORE	EXECUTION TIME (µs)
Philips XA-G3	359.86
Intel MCS251	938.75

Benchmark limitations

Like all benchmarks, the automotive engine management assembler functional benchmark has some weakness that limit validity of its results.

1. Control in a special (automotive, engine) environment is evaluated.
2. Occurrences of operation overheads are based on estimations.
3. Occurrences of functions are based on estimations.
4. Functions are implemented in assembler, not in a HLL like C.
5. Routines may contain assembler implementation errors.
6. Cores are evaluated at 16.0 MHz

Control in a special environment is evaluated (automotive, engine)

The core performance evaluation is based on a single specialized case. All benchmark implementations are fractions of the automotive engine management PCB83C552 demonstration program.

It can be advocated that the automotive engine control task gives a good example of a typical high demanding control environment, where many >= 16 bit calculations have to be done.

Occurrences of overheads are based on estimations

The assembler functional benchmark is not a full implementation of a program. Arbitrary choosing location for storage of parameters in register file or (external) memory, for instance, has for some instruction set a considerable effect on the total execution time.

For the different core parameter storage is chosen where possible using the core facilities to have minimum access overhead.

Occurrences of functions based on estimations

Occurrences is estimated on basis of experience of the automotive group. In a real implementation of an engine controller accents may shift. As most functions already include some “instruction mix”, the effect of changes in occurrences is limited.

Functions are implemented in assembler, not in a HLL like C.

Control programs for embedded systems get larger, have to provide more facilities and have to be realized in shorter development times. The only way to do this is to program in a HLL like C. Efficient C–language program implementation requires different features from microcontrollers than assembly programs. Results of this assembler benchmark evaluation therefore have a restricted value for ranking microcontroller performances for future HLL applications.

Benchmark ranking on basis of HLL like C requires good C–compilers of all the devices involved are needed. The quality of the C–compilers really has to be the best there is : HLL benchmarking measures not only the micro characteristics, but even more the compiler ability to use these qualities. As these are not available for all the micros evaluated, all routines are worked out only in assembly.

All cores are evaluated at 16.0 MHz

A 16.0 MHz internal clock frequency seems a reasonable choice for comparing the cores at the same level of technology:

Assembler functional benchmark for automotive engine management

This benchmark is a functional benchmark: it is a collection of functions to be executed in an automotive engine management program. To implement the assembly functional benchmark for automotive engine management correctly the “rules and details” described in this section have to be followed carefully.

The assembler functional benchmark embraces all activity to be completed in 1 program cycle that corresponds with 1 engine stroke of 2 ms. The benchmark execution time will be calculated as the sum of the products of functions and their occurrence rates in 1 calculation cycle.

Branches are evaluated separately as “branch penalties” have considerable effect of program execution efficiency. Estimated (branch count)*(average branch time) is added to the function execution times.

The relative estimated overhead for statistics does not contribute to the evaluation of speed performance ratios, but they have to be considered when looking at the total execution time required / engine stroke cycle. therefore the real total execution time is multiplied with the statistics overhead factor (1.2*).

NO.	FUNCTION DESCRIPTION	OCCURRENCES
1	16×16 Multiply	12
2	Floating Point divide (16:16)	4
3	Add/Subtract (24)	50
4	Compare (24)	13
5	CAN cmp/mov 10*8	80
6	Linear Interpolation (8*8)	20
7	Program control branches	500
8	Statistics (20%)	1.2 *

Function Parameter Allocation

Most functions are very short in exec. time, so that the function parameter data access method has great effect on the total time. Thus it is to be considered carefully. Both XA and MCS251SB have register files in which variables can be stored.

For the XA and 251SB processors, data is stored in the lower part of register file, or in sfrs for I/O, can be accessed using “direct”addressing, but table data, used e.g. for 3 byte compare, is stored in “external memory”. For more complex functions 16*16 multiply, Floating point division and interpolation, data is assumed to be already in registers.

16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32–bit result written into a register pair.

Divide (16:16) “floating point”

The floating point division is entered with parameters in registers:

a divisor, a dividend and an “exponent” that determines the position of the fraction point in the result.

Floating point binary 16/16 division is a function that is normally not included in HLL compilers as it requires separate algorithms for exponent control and accuracy is limited. For assembler control algorithms, floating point division can be quite efficient as it is much faster than normal “real” number calculations (where no “floating point accelerator” hardware is available).

Compare 24–bit variables

Note that 24–bit compare is very efficient for “real” 16–bit and 8–bit) controllers, but for automotive engine timers, 24–bit seems a good solution. Compare must give possibility to decide >, < or =. An average branch is included in the function.

CAN move and compares

For service of the CAN serial interface, it is estimated that 40* (2 byte compares + branch) have to be done. Devices with 16–bit bus assumes word access. An average branch is included in the CAN compare function.

Linear Interpolation (8*8)

The interpolation routine is entered with 3 register parameters:

1. Table position address
2. X fraction
3. Y fraction

The routine first interpolates using the X fraction the values of F(x.x, y) between F(x,y) ....V(x+1, y) and of F(x.x, y+1) between F(x, y+1) .... F(x+1, y+1). From F(x.x, y) and F(x.x, y+1) the value of F(x.x, y.y) is interpolated using the fraction of y.

The table is organized as 16 linear arrays of 16 x–values, so that an V(x,y) can be accessed with table origin address +x+16*y = “Table Position Address”. In x–direction the interpolation can be done between the “Table Position” value and next position (+1). Interpolation in y–direction is done by looking at “Table Position” + 16.

For linear interpolation time the 2–dimensional interpolation time and byte count are divided by 3 to include some “overhead” into linear interpolation.

Program Control Overheads

For a given algorithm, the “program control overhead” consisting of a number of decisions (=branches) and subroutine calls is independent of the instruction set used, except for cases where functions can be replaced by complex instructions. The most important exception cases, MPY words and Floating Point Division are handled in this benchmark separately.

Most 16–bit cores use more pipeline stages so that taken branches add branch time penalty for these CPU’s due to pipeline flush. This effect can be found in the branch execution time tables.

More efficient data operations and pipeline penalty of the more complex instruction set of 16–bit cores lead to considerable higher relative time used for branch instructions.

To incorporate the influence of branches in the benchmark the number of branches to be included must be estimated. For byte and bit routines, branches occur more frequent. Average branch time of 25% may be a good guess. For the automotive engine management benchmark that executes in approx. 5000/µS (on 8051) results in +/– 1250 /µS or 625 branches. As a part of the branches already taken account for in the compare functions the number of additional program control branches is estimated 500 branches.

To estimate the average branch execution time, an estimated relative occurrence of the branch types has to be made.

Table 4. Estimated relative occurrence of the branch types

	TYPE	RELATIVE	ABSOLUTE OCCURRENCE
Absolute Jumps	AJMP/JMP	20%	100
Subroutine calls	ACALL/JSR	20%	100
Jump on condition (rel)	Bcc/Jcc	40%	200
Jump on bit (rel)	JB/JBN	20%	100

Statistic Routine Overheads

Statistic routines are estimated as relative program overheads, only to get an indication of the required total processing time in a real engine management application. “Statistics” are mainly arithmetic routines to determine table corrections. They use about 20% of the total time.

XA BENCHMARK RESULTS

The following analysis assumes worst case operation. At any point in time, only 2 bytes are available in the instruction Queue. An instruction longer than 2 bytes requires additional code read cycle.

APPENDIX 1

XA Function Implementations

XA reference: *XA User's Manual 1994*

A1.1: 16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32-bit result written into a register pair.

```
MUL.w                 R0, R1         ; result is in register pair R1:R0
```

2 Bytes, 12 clocks ==> 0.75 µs

A1.2: Floating Point 16x16 Divide:

```
;The floating point division is entered with parameters in registers:

;Arguments:    R4 = Dividend (extend into R5 for 32 bits)
;              R6 = Divisor Mantissa
;              R0 = Divisor  Exponent

FPDIV:
       ADDS           R6, # 0        ; Add short format
       BEQ            L1             ; divby 0 chk – if z=1, go to L1

SGNXTD_AND_SHFT:
       SEXT.W         R5             ; Sign extend into R5
       ASL            R4, R0L        ; 13 position shifts (average)

DIV:                                 ;
       DIV.d          R4, R6         ; Divide 32x16 signed
       BOV            L1             ; Branch on Overflow
       RET                           ; Normal termination

L1:
       MOVS           R4, # –1       ; Overflow – Max Result
       RET
```

18 Bytes, 48 clocks ==> 3.0 µs

A1.3: Extended 32-bit subtract

```
;      R5:R4 = Minuend
;      R3:R2 = Subtrahend

       SUB.w          R4, R2
       SUBB.w         R5, R3
```

4 Bytes, 6 clocks ==> 0.375 µs

A1.4: Compare 24-bit Variables

An average branch is included after compare.

The table data, used for 3 byte compare, is stored in “memory”.

```
CMP:
       CMP.B   R1L, R2L               ;
       BNE     L1                     ;
L1:
       CMP.W   R0, mem1               ;
       BGT     LABEL1                 ;
                                      ;
LABEL1:
;     xx –> GT or LT or EQ
```

9 Bytes, 20 clocks (average – branch always taken and not taken) ==> 1.25 µs

A1.5: CAN Compare and Move

Application: For service of CAN (Controller Area Network) serial Interface it is estimated that 80* (2 byte compares + branch) have to be done. One parameter is in register, the other in internal memory.

```
CAN:
       CMP           R0, mem0       ; mem0 = $10H          3
       BGT           LABEL          ;                      2
LABEL:
```

5 Bytes, 9 clocks (average) ==> 0.563 µs

A1.6: Linear Interpolation

```
Arguments:
               R0 = Table Base (assumed < 400 Hex)
               R2 = Fraction 1
               R4 = Fraction 2
               R6 = Result

LIN_INT:
               MOV            R2, [R5+]             ;                      2
               MOV            R0, [R5]              ;                      2
               SUB            R0, R2                ;                      2
               MULU.w         R2, R6                ;                      2
               MOV.b          R0H, R0L              ;                      2
               MOVS.b         R0L,#0                ;                      2
               ADD            R2, R1                ;                      2
               ADD            R5, #15               ;                      2
               MOV            R0, [R5+]             ;                      2
               MOV            R4, [R5]              ;                      2
               SUB            R4, R0                ;                      2
               MULU.w         R4, R6                ;                      2
               MOV.b          R0H, R0L              ;                      2
               MOVS.b         R0L,#0                ;                      2
               ADD            R0, R4                ;                      2
               SUB            R0, R2                ;                      2
               MULU.w         R0, R5                ;                      2
               MOV.b          R0H, R0L              ;                      2
               MOVS.b         R0L,#0                ;                      2
               ADD            R2, R0                ;                      2
               RET                                  ;                      2

                                                    ;                      42
```

42 Bytes, 98 clocks ==> 6.125 µs
Linear Interpolation (2 dim. time / 3) = 42 bytes, 2.04 µs

A1.8: Program Overhead

Branches are assumed taken 70% of the time, all addresses are external. Code is assumed a run–time trace, code size cannot be calculated.

TYPE	OCCURRENCE	XA		BYTES	
JMP rel16	100	6	600	3	300
CALL rel16	100	4	400	3	300
Bxx rel8	200	5.1	1020	2	400
JNB bit,rel8	100	5.1	510	2	200
total cylces µsec			2,530 158.13		1,200

A1.9: XA Totals

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.0	12	18
ADD/SUB	50	0.375	18.75	4
CMP 24b	13	1.25	16.25	16
CAN 16b	80	0.562	44.96	8
INTPLIN	20	2.04	40.8	14
BRANCH	1		158.3	1200

XA total/µs: 299.89 µs
including 20% statistics: 359.86 µs

Note:

An assumption is made that XA code is in first 64K (PZ), that is, only 64K address space is used.

APPENDIX 2

MCS251 Implementations

MCS251 reference: *“MCS251SB Embedded microcontroller users manual”*, February 1995.
All data are taken using the Kiel Development Board using a 251SB 16.0 MHz part.

A2.1: MCS251SB 16×16 Multiply

```
;The MCS251 can do only unsigned multiply. So, there will be some overhead for testing
;the sign of the result.

MUL             R0,R1

;Total: 2 bytes, 24 clocks ==> 1.5 µs
```

A2.2: Floating point division 16:16

```
; Arguments:   WR4 = 16–bit Dividend
;              WR6 = 16–bit Divisor Mantissa
;              WR0 = Divisor Exponent

FPDIV:
       ADD     WR2,#0                 ;                     4
       JE      L1                     ;                     2
                                      ;
SGNXTD_AND_SHFT:
       MOVS    WR6,R5                 ;                     2

SHFT_LOOP:
       SLL     WR4                    ;NO ARITH SLL ?       2
       DJNZ       R0,SHFT_LOOP   ;DOES 1 BIT AT A TIME      3

DIVISION:
       DIV     WR4,WR2                ;                     2
       JB      OV,L1                  ;IF OVFLW BIT IS SET  4
       RET                            ;NORMAL TERMN.        1

L1:                                   ;
       MOV     WR4, #–1               ; OVFL – MAX RESULT   4 (not exc)
       RET                            ;                     1

; Totals: 25 bytes, 482 clocks ==> 30.125 µs
```

A2.3: Add/Sub

```
;      DR0 = Minuend
;      DR4 = Subtrahend

SUB    DR0,DR4                        ;

; Totals: 2 bytes, 10 clocks ==> 0.625 µs
```

A2.4: Compares 24 (=32) bit

```
COMPARE:
       MOV     WR0,60H                ;memory               3
       MOV     WR2,50H                ;memory               3
       CMP     DR0,DR4                ;                     2
       JE      CMP_EQUALS             ;                     2
       SJMP    CMP_APPROX             ;                     2
CMP_EQUALS:
CMP_APPROX:

; Totals: 12 bytes,   54 clocks (branch average)  ==> 2.375 µs
```

A2.5: CAN move and compares (16-bit)

```
COMPARE:
        CMP     WR0,mem0              ;mem0 = 40H           4 bytes, 6 clocks
        JNE     THERE                 ; 2 bytes 2t/8nt
THERE:

; Totals: 6 bytes, 10 clocks ==> 0.625 µs
```

A2.6: 2-dimensional interpolation

```
;Arguments:
;               XAR0 = Table Base (assumed < 400 Hex)
;               XAR2 = Fraction 1
;               XAR4 = Fraction 2
;               XAR6 = Result
;               XAR1 = temporary1
;               XAR0 = temporary2
;               XAR5 = temporary3
;
;               WR0 = Table Base (assumed < 400 Hex)
;               WR2 = Fraction 1
;               WR4 = Fraction 2
;               WR6 = Result
;               WR8 = temporary1 = XAR1
;               WR10 = temporary2 = XAR0
;               WR12 = temporary3 = XAR5

LIN_INT:
        MOV     WR6,@WR10             ; 3     6
        ADD     WR10,#2               ; 4     6
        MOV     WR8,@WR10             ; 3     6
        SUB     WR8,WR6               ; 2     4
        MUL     WR6,WR2               ; 2     22
        MOV     R2,R1                 ; 2     2
        MOV     R1,#0                 ; 3     4
        ADD     WR6,WR8               ; 2     4
        ADD     WR10,#15              ; 4     6
        MOV     WR8,@WR10             ; 3     6
        ADD     WR10,#2               ; 4     6
        MOV     WR12,@WR10            ; 3     6
        SUB     WR12,WR8              ; 2     4
        MUL     WR12,WR2              ; 2     22
        MOV     R2,R1                 ; 2     2
        MOV     R1,#0                 ; 3     4
        ADD     WR8,WR12              ; 2     4
        SUB     WR8,WR6               ; 2     4
        MUL     WR8,WR4               ; 2     22
        MOV     R2,R1                 ; 2     2
        MOV     R1,#0                 ; 3     4
        ADD     WR6,WR8               ; 2     4
        RET                           ; 1     12

; Totals:  58 bytes, 274 clocks ==> 17.125 µs
; Linear Interpolation (2 dim. time / 3) = 60 bytes, 5.71 µs
```

A2.7: MCS251 Program Overhead

TYPE		OCCURRENCE	MCS251		BYTES	
LJMP	addr16	100	8	800	4	400
LCALL	addr16	100	18	1800	3	300
JLE	rel	200	6.8	1360	2	400
JNB	rel	100	10.8	1080	4	400
total cylces µsec				5040 315.0		1500

A2.8: MCS251 Totals

FUNCTION	OC*	MCS251		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	1.53	18.36	2
FDIV	4	30.125	120.6	25
ADD/SUB	50	0.641	32.05	2
CMP 24b	13	3.375	43.88	12
CAN 16b	80	1.625	130	6
INTPLN	20	6.12	122.4	60
BRANCH	1		315.0	

MCS251 total/µs: 782.29 µs
including 20% statistics: 938.75 µs

EXECUTION TIME PERFORMANCE

Actual execution times/function

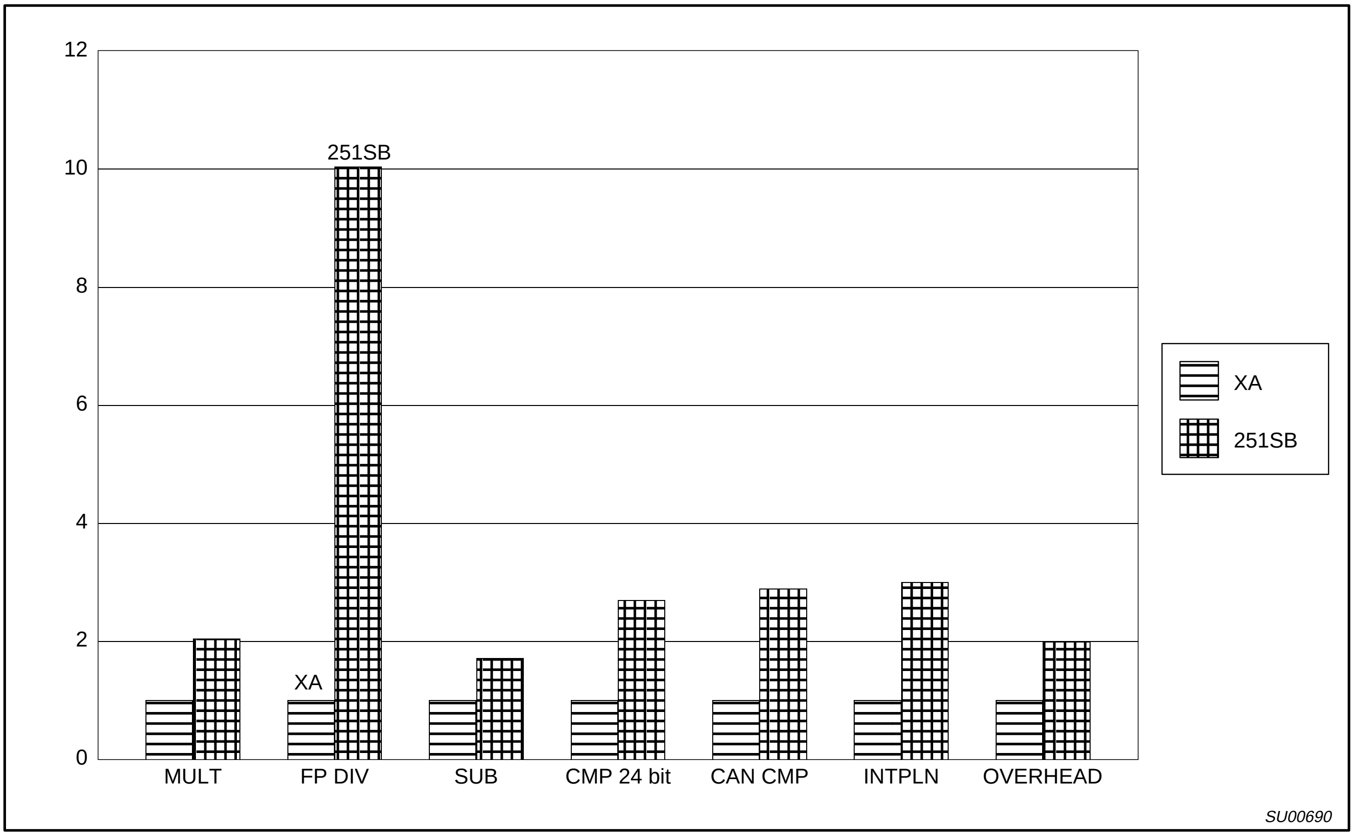
FUNCTIONS	XA	251SB
MULT	0.75	1.53 *
FP DIV	3	30.125
SUB	0.375	0.641
CMP 24 bIT	1.25	3.375
CAN CMP	0.562	1.625
INTPLN	2.04	6.12
OVERHEAD	158.13	315

* Only for unsigned, extra overhead for sign needs to be added.

Normalized timings/function

FUNCTIONS	XA-G3	251SB
MULT	1	2.04
FP DIV	1	10.04
SUB	1	1.71
CMP 24 bIT	1	2.7
CAN CMP	1	2.89
INTPLN	1	3
OVERHEAD	1	1.99

EXECUTION BENCHMARK



BENCHMARK OF CODE DENSITY

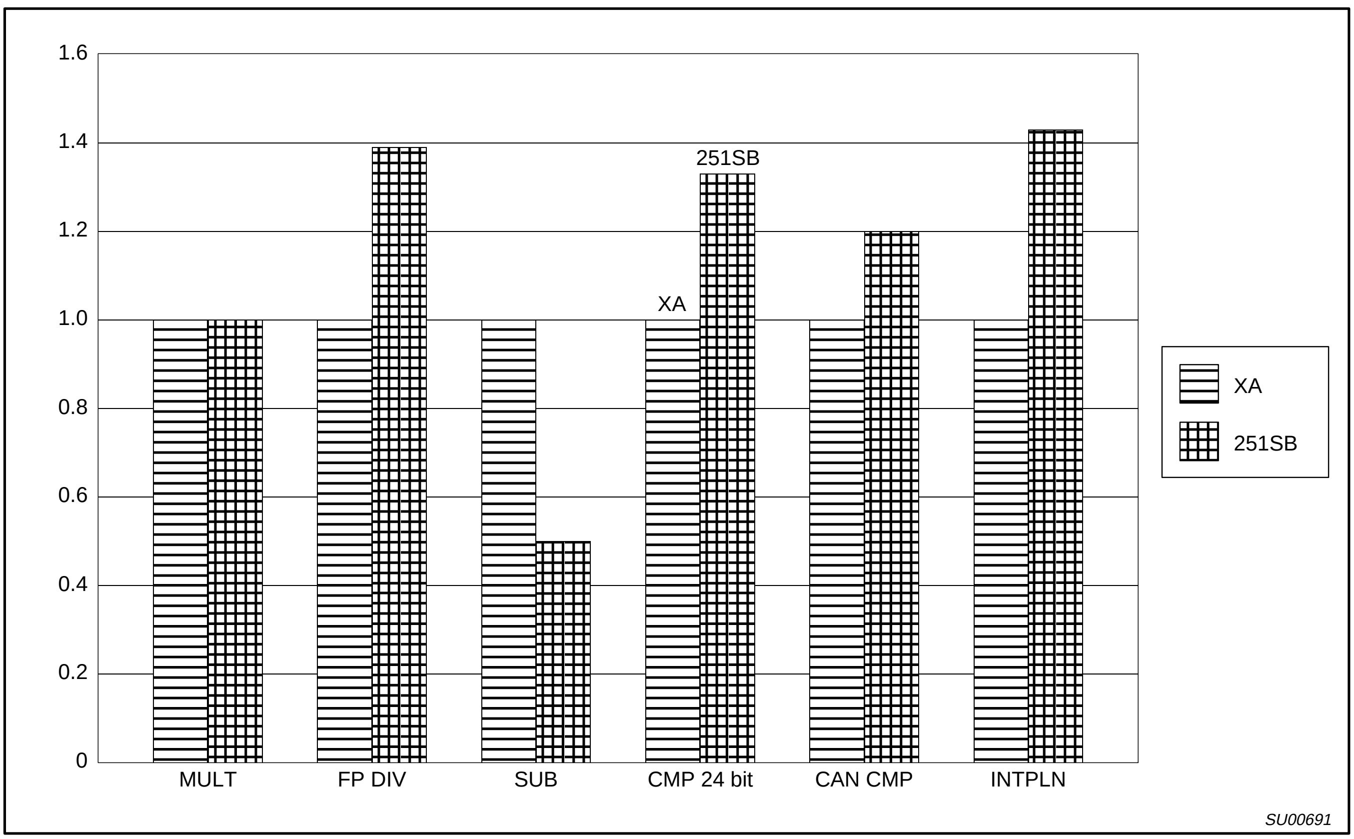
Actual code density performance

FUNCTIONS	XA-G3	251SB
MULT	2	2
FP DIV	18	25
SUB	4	2
CMP 24 bIT	9	12
CAN CMP	5	6
INTPLN	42	60

Normalized w.r.t. XA

FUNCTIONS	XA-G3	251SB
MULT	1	1
FP DIV	1	1.39
SUB	1	0.5
CMP 24 bIT	1	1.33
CAN CMP	1	1.2
INTPLN	1	1.43

CODE DENSITY BENCHMARK



BM1.ASM

```
$include xa–g3.equ
$include bm.inc

 ;16x16 signed multiply



        org $0
        dw $8f00,start
;

        org $200

start:

         setp_15
         MUL.w   R0, R1   ;     2
         rstp_15
         br      start


;Totals = 2 Bytes, 12 clocks (0.75 uS)
```

BM2.ASM

```
;$listing_min
$include xa–g3.equ
$include bm.inc


        org $0
        dw $8f00,start                          Bytes      Clocks
;

        org $200
;r6= divisor mantissa
;r0=divisor exponent
;r4=dividend (extended to r5 for 32–bits)

start:
        movs.b  r61,#2          ; some value > 0
        mov.b   r01,#13         ;
        mov.w   r4,#$200        ;
        mov.w   r6,#$100        ;
        call    FPDIV           ;
        br      start


FPDIV:
        setp_15
        ADDS    R6, # 0         ; Add short format              2
        BEQ     L1              ; divby 0 chk                   2
                                ;– if z=1, go to L1
                                ;
SGNXTD_AND_SHFT:                ;
        SEXT.W  R5              ; Sign extend into R5           2
        ASL     R4, R0L         ; 13 position shifts (average)  2
                                ;
DIV:                            ;
        DIV.d   R4, R6          ; Divide 32x16 signed           2
        BOV             L1      ; Branch on Overflow            2
        rstp_15                 ;

        RET                     ;                               2
                                ;
L1:                             ;
        MOVS    R4, # –1        ; Overflow – Max Result         2
        rstp_15
        RET                                                     2

;Totals = 18 Bytes, 48 clocks (averages for branches) i.e 3.0 uS at 16.0 MHz
```

BM3.ASM

```
;$listing_min
$include xa–g3.equ
$include bm.inc


        org $0
        dw $8f00,start
;                                                Bytes      Clocks

        org $200

start:
        MOV             R4,#$200
        MOV             R5,#$210
        MOV             R2,#$100
        MOV             R3,#$110

        setp_15

;Extended 32-bit subtract

        SUB     R4, R2          ;       2
        SUBB    R5, R3          ;       2

        rstp_15
        br      start

;Totals= 4 Bytes and 6 clocks  (0.375 uS) at 16.00 MHz
```

BM4.ASM

```
$include xa–g3.equ
$include bm.inc

mem1    equ             $20

        org $0
        dw $8f00,start
;;Compare 24–bit Variables                                  Bytes      Clocks

        org $200

start:
        mov    R2L,#$40         ;  one parameter is register
        mov    mem1,#$1000      ;  and one in memory
        mov    R1L,#$50         ;
        mov    R0,#$5000        ;

CMP:
        setp_15
        CMP.B  R1L, R2L         ;                2
        BNE    L1               ;                2
                                ;


L1:
        CMP.W  R0, mem1         ;                3
        BGT    LABEL1           ;  average       2

LABEL1:
;     xx –> GT or LT or EQ
        rstp_15
        br     start

;Totals= 9 Bytes and 20 clocks i.e 1.25 uS at 16.00 MHz
```

BM5.ASM

```
$include xa–g3.equ
$include bm.inc


;A1.5
;CAN Move and Compare
;one parameter in register, the other in memory

mem0            equ             $10


        org $0
        dw $8f00,start
;
  Bytes Clocks

        org $200
start:
        mov     mem0,#$100
        mov     R0,#$50

CMPR:
        setp_15
        CMP     R0, mem0         ;       3
        BGT     LABEL            ;       2

LABEL:

        rstp_15
        br      start


;Totals = 5 Bytes and 9 clocks (average for branches)
;or 0.563 uS at 16.00 MHz
```

BM6.ASM

```
$include xa–g3.equ
$include bm.inc

mem1    equ             $20

        org $0
        dw $8f00,start


;Linear Interpolation

;Arguments:
;               R4 = Table Base (assumed < 400 Hex)
;               R6 = Fraction 1
;               R5 = Fraction 2
;               R2 = Result




        org $200
start:
        mov     r7,#$100        ;safe
        movs    scr,#1          ;page 0
        mov     R5,#$120
        mov     R2,#$12F
        mov     R4,#$80
        mov.w   $120,#$45
        call    LIN_INT
        rstp_15
        br      start

LIN_INT:
        setp_15                         ;
        MOV     R2, [R5+]               ;       2
        MOV     R0, [R5]                ;       2
        SUB     R0, R2                  ;       2
        MULU.w  R2, R6                  ;       2
        MOV.b   R0H, R0L                ;       2
        MOVS.b  R0L,#0                  ;       2
        ADD     R2, R1                  ;       2
        ADD     R5, #15                 ;       2
        MOV     R0, [R5+]               ;       2
        MOV     R4, [R5]                ;       2
        SUB     R4, R0                  ;       2
        MULU.w  R4, R6                  ;       2
        MOV.b   R0H, R0L                ;       2
        MOVS.b  R0L,#0                  ;       2
        ADD     R0, R4                  ;       2
        SUB     R0, R2                  ;       2
        MULU.w  R0, R5                  ;       2
        MOV.b   R0H, R0L                ;       2
        MOVS.b  R0L,#0                  ;       2
        ADD     R2, R0                  ;       2
        RET                             ;       2
;Totals = 42 bytes and 98 clocks i.e 6.125 us at 16.00 MHz
;For 2–dim interpolation, exec. time = 6.13/3 = 2.04 us
```

BM1.A51

```
$TITLE(bm1.a51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM1 SEGMENT CODE
        RSEG ?PR?BM1

; 16x16 '251 Multiply
;
;
;
test:
        T_START
        MUL     WR0,WR2         ;   2
        T_END
;stall:
        sjmp     test

;Totals:  2 bytes, 24.5 clocks ==> 1.53 uS

        END
```

BM2.A51

```
$TITLE(bm2.a51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM2 SEGMENT CODE
        RSEG ?PR?BM2

; 251 Floating Point 16x16 Divide, 16:16
;
; Note: the '251 may have a shift-by-n, but I can;t seem to find it!
; If there is one, the '251 results would likely improve.
;
; Arguments:     WR4 = 16-bit Dividend
;                WR2 = 16-bit Divisor Mantissa
;                WR0 = Divisor Exponent

test:
        mov  r0,#13
        mov  wr4,#200H
        mov  wr2,#100H
        call FPDIV
                                        ; return here
stall:
        jmp test


FPDIV:
        T_START
        add     wr2,#0                  ;                        4
        je      l1                      ;                        2
                                        ;
SGNXTD_AND_SHFT:
        movs    wr6,r5                  ;                        2
SHFT_LOOP:
        sll     wr4                     ;No arith sll ?          2
        djnz    r0,SHFT_LOOP            ;does 1 bit at a time    3

DIVISION:
        div     wr4,wr2                 ;                        2
        jb      OV,L1                   ;if ovflw bit is set     4
        T_END
        ret                             ; Normal termination     1
L1:                                     ;
        mov     wr4, #-1                ; Overflow - Max Result  4

        T_END
        ret

        END


;Totals: 25 bytes, 482 clocks ==>  20.125 uS

;Note : The shift instructions are taking 10 clocks in the MCS251 part
;instead of 2 clocks as specified in the manual. No idea why !!!
;For sign divide in MCS 251, there will be a considerable overhead involved
```

BM3.A51

```
$TITLE (BM3.A51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM3 SEGMENT CODE
        RSEG ?PR?BM3

;; Extended 32–bit subtract
; Z = X – Y
;
; entry: DW(X) in DR0
;        DW(Y) in DR4
; exit:  DW(Z) in DR0
;
;
SUBTR:
        T_START
        SUB     DR0,DR4                 ;    2
        T_END
        sjmp    SUBTR
        END

; Totals: 2 bytes, 10.25 clocks ==> 0.641 uS at 16.00 MHz
```

BM4.A51

```
$TITLE (BM4.A51)
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM4 SEGMENT CODE
        RSEG ?PR?BM4

; Compare 24–bit Variables

; The ’251 really uses fewer instruction for a 3 byte compare because it
;
test:
        mov     wr4,#4000H
        mov     wr6,#2000H
        mov     60H,wr6
        mov     50H,wr4

compare:
        T_START
        MOV     WR0,60H         ;              3
        MOV     WR2,50H         ;              3
        CMP     DR0,DR4         ;              2
        JE      CMP_EQUALS      ;              2
        SJMP    CMP_APPROX      ;              2

; Totals: 12 bytes, 54 clocks (average) ==> 3.375 uS
CMP_EQUALS:
CMP_APPROX:
        T_END
        sjmp    compare
        END
```

BM5.A51

```
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM5 SEGMENT CODE
        RSEG ?PR?BM5

; CAN COMPARE
;1 parameter in register, the other in memory

test:
         MOV     WR0,#2000H
         MOV     WR4,#3000H
         MOV     40H,WR4
compare:
        T_START
        CMP     WR0,40H         ;    4
        JNE     THERE           ;    2

THERE:
        T_END
        jmp test

        end


;
; Totals: 6 bytes, 26 clocks (average branches) ==> 1.625 uS at 16 MHz
;
```

BM6.A51

```
$INCLUDE (reg251sb.inc)
$INCLUDE (bm.inc)

?PR?BM6 SEGMENT CODE
        RSEG ?PR?BM6

;;Linear Interpolation

;Arguments:
;               XAR0 = Table Base (assumed < 400 Hex)
;               XAR2 = Fraction 1
;               XAR4 = Fraction 2
;               XAR6 = Result
;               XAR1 = temporary1
;               XAR0 = temporary2
;               XAR5 = temporary3
;
;               WR0 = Table Base (assumed < 400 Hex)
;               WR2 = Fraction 1
;               WR4 = Fraction 2
;               WR6 = Result
;               WR8 = temporary1 = XAR1
;               WR10 = temporary2 = XAR0
;               WR12 = temporary3 = XAR5

;
test:
        call    LIN_INT
        T_END                           ; return here
stall:
        jmp test

LIN_INT:
        T_START

        MOV             WR6,@WR10       ;                          3
        ADD             WR10,#2         ;                          4

        MOV             WR8,@WR10       ;;                         3
        SUB             WR8,WR6         ;;                         2
        MUL             WR6,WR2         ;                          2

        MOV             R2,R1           ;                          2
        MOV             R1,#0           ;                          3

        ADD             WR6,WR8         ;;                         2
        ADD             WR10,#15        ;                          4
        MOV             WR8,@WR10       ;;                         3

        ADD             WR10,#2         ;                          4

        MOV             WR12,@WR10      ;;                         3
        SUB             WR12,WR8        ;;                         2
        MUL             WR12,WR2        ;;                         2
        MOV             R2,R1           ;                          2
        MOV             R1,#0           ;                          4
        ADD             WR8,WR12        ;;                         2
        SUB             WR8,WR6         ;;                         2
        MUL             WR8,WR4         ;;                         2
        MOV             R2,R1           ;                          2
        MOV             R1,#0           ;                          4
        ADD             WR6,WR8         ;;                         2
        RET                             ;

        END

; Totals:  60 bytes, 294 clocks ==>18.36 uS at 16.00 MHz
```