XA benchmark versus the architectures 68000, 80C196, and 80C51

AN703

Author: Santanu Roy

BACKGROUND

A computer benchmark is a “program” that is used to determine relative computer core performance by evaluating benchmark execution time by that core. In the brainstorm on 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 gives worked out routines of the functions if they were implemented in assembler language of the compared controllers: Motorola 68000, Intel 80C196, Philips 80C552 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 is generated based on the report number DPE88187. It outlines code density and execution times of the XA, based on most recent information. The execution times are given in terms of both clock cycles and time units. Although XA can run at speed of 30 MHz @ 5.0 Volts, for sake of fairness, all cores are evaluated for running at 16.00 MHz. This is reasonable for comparing the cores at the same level of technology.

A separate section is included in this benchmark for “Bit manipulation” function benchmark results only. This (bit-test) routine is a stand alone one and should not be considered as a part of *engine management* routine.

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)

5,942 1,560 1089.24 402.6

PERFORMANCE RATIO	8051	68000	80C196	XA
8051	1.0	3.81	5.45	14.7
68000	0.34	1.0	1.43	3.85
80C196	0.18	0.7	1.0	2.7
XA	0.068	0.26	0.37	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	3.94	15.8	18
ADD/SUB	50	0.38	19	4
CMP 24b	13	1.06	13.78	9
CAN 16b	40	0.563	22.52	5
INTPLIN	20	1.98	41.3	14
INTERR	10	6.1	61	41
BRANCH	10		153.1	

XA totals : 335.5 µs
including 20% statistics : 402.6 µs

Table 2. 68000 instruction set execution times and bytes/function

FUNCTION	OC*	68000		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(μs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	4.4	52.8	2
FDIV	4	13.4	53.6	16
ADD/SUB	50	2.75	137.5	12
CMP 24b	13	3.2	41.6	14
CAN 16b	40	2.7	108	14
INTPLIN	20	7.5	150	14
INTERR	10	21.9	219	92
BRANCH	10		537.5	

68000 totals : 1,300 μs
including 20% statistics : 1,560 μs

Table 3. 80C196 instruction set execution times and bytes/function

FUNCTION	OC*	80C196		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(μs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	1.75	21	3
FDIV	4	9.5	38	19
ADD/SUB	50	1.25	62.5	7
CMP 24b	13	4.25	55.2	14
CAN 16b	40	2.5	100	6
INTPLIN	20	6.4	128	18
INTERR	10	12.8	128	58
BRANCH	10		375	

80C196 totals : 907.7 μs
including 20% statistics : 1,089.24 μs

Table 4. 8051 instruction set execution times and bytes/function

FUNCTION	OC*	8051		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(μs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	37.5	450	58
FDIV	4	451.5	1806	96
ADD/SUB	50	7.5	375	19
CMP 24b	13	9.98	129.74	22
CAN 16b	40	9	360	14
INTPLIN	20	25.8	516	20
INTERR	10	31.5	315	70
BRANCH	10		1000	

8051 totals : 4,951.74 μs
including 20% statistics : 5,942 μs

Table 5. Total benchmark execution time results

MICROCONTROLLER CORE	EXECUTION TIME (μs)
XA	402.6
68000	1560
80C196	1089.24
8051	5942

As the total activity has to be completed in one machine stroke of 2 ms, the XA, and the 80C196 will be able to meet the application requirements. The 80C552 originally was assumed to complete the functions over more than one stroke.

Best efficiency is of the XA and the 80C196. The 80C196 includes 3-parameter instructions that reduce the instruction count per function and it has JB/JBN instructions. It also uses half-word (1-byte) codes for frequently used instructions.

The lower code efficiency of the 8051 instruction set can mainly be explained by the “accumulator bottleneck” which is not present in XA: most data has to be transported to and from the accumulator be fore add/sub/cmp can be done, operations on words require 4 “MOV” instructions and 2 data execution instructions. The efficient JB and JBN instructions compensate this for a great part.

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. All 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.

Routines may contain assembler implementation errors

Assembler routine implementations are made after a short study of the micro specifications and are not checked by assembling or debugging in real hardware environment.

It can be rather safely said that a complete system setup and program debug to correct errors would not lead to considerable differences in performance results. Deviations in function occurrences and overheads may have a more significant effect on performance ratios.

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. It would be preferable to implement the complete control program in assembler and evaluate it in a real hardware environment, but this is not practical as every implementation requires many man-months to realize.

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	Interrupts	10
8	Program control branches	500
9	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.

Some core features a large register files (XA, 80C196) in which variables can be stored, others with few registers (68000) have to store all data in memory.

For the XA/80C196 processor, data 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 bye compare, is stored in “external memory”.

The 68000 assume data in memory (or memory mapped I/O) as not enough data registers are available. All 68000 memory data has to be accessed using long-absolute addressing: 68000 short addresses are relative to memory address 0000 and are therefore not useful.

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 =. For 68000, and 80C196 instruction set LT, EQ and GT are included in the cc after CMP.

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.

Interrupts

The average interrupt routine overhead includes the following stages:

a. Interrupt recognition and return

b. 1 * (long) branch

c. 2 * jump (short) on bit

d. 1 * call (long) and subroutine return

e. 2* set bit and 2 * clear bit

f. 5 * POP and 5 * PUSH (or move multiple)
[free 5 registers for local use]

g. 1 * mov #xxx, PST

Program Control Overheads

For a given algorithm, the Program Control Overheads 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 6. 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*

16×16 Signed Multiply

Parameters are assumed to be in registers, and the 32-bit result written into a register pair. The MUL.w R,R is encoded in the XA instruction set as a 2 byte instruction. The exact optimization for this instruction (such as skip over 1's and 0's) has not been concluded at this point, and the execution time may be data dependent and shorter than one outlined here.

The basic algorithm utilizes 2-bit Booth recoding. Instruction fetch and Decode time overlaps the execution of the preceding instruction (except when following a taken branch), so it is ignored. The total execution time is either 11 or 12 clocks, including operand fetch and write back (1 clock is dependent on critical path analysis).

A1.1: 16×16 Multiply

```
                                                          Bytes     Clocks
       MUL.w   R0, R1                                       2       12 (0.75 µS)
```

A1.2: Floating Point 16x16 Divide:

The algorithm here follows the one outlined for the 80C196.

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

                                                          Bytes   Clocks
FPDIV:
       ADDS    R6, # 0               ; Add short format       2       3
       BEQ     L1                    ; Check for DIVBY0       2       3  (not taken)
                                     ;
SGNXTD_AND_SHFT:                     ;
       SEXT    R5                    ; Sign extend into R5    2       3
       ASL     R4, R0                ; 13 position shifts     2       11
                                     ;
DIV:                                 ;
       DIV.d   R4, R6                ; Divide 32x16 signed    2       21
       BOV     L1                    ; Branch on Overflow     2       6  (taken)
       RET                           ; Normal termination     2       8
                                     ;
L1:                                  ;
       MOVS    R4, # –1              ; Overflow – Max Result  2       3  (not  executed)
       RET                           ;                        2       8

                                                             18      63 (3.94 µS)
```

A1.3: Extended 32-bit subtract

```
; R5:R4 = Minuend
; R3:R2 = Subtrahend
       SUB     R4, R2                                        2       3
       SUBB    R5, R3                                        2       3

                                                             4       6  (0.38 µS)
```

A1.4: Compare 24-bit Variables

Only minimum execution time is considered here. An average branch is included after compare. The table data, used for 3 byte compare, is stored in memory.

```
                                                          Bytes   Clocks
       CMP.b          R1L, R2L       ; direct addressing     2      4
       BNE            L1             ; average (6t/3nt)      2      4.5
       CMP.w          R0, mem2       ;                       3      4
L1:
       CMP.w          R0, mem1       ;                       3      4
       Bxx            LABEL1         ; average               2      4.5
LABEL1:
                                     ; xx –> GT or LT or EQ
                                                             9      17 (1.06 µS)
```

A1.5: CAN Move and Compare

Application:

For service of CAN (Controller Area Network) serial Interface it is estimated that 40* (2 byte compares + branch) have to be done. One parameter is in register, the other in internal memory. Again, minimum execution times are considered.

```
                                                          Bytes   Clocks
       CMP            R0, mem0       ;                       3      4
       Bxx            LABEL          ; average               2      4.5

                                                             5      9 (0.563 µS)
```

A1.6: Linear Interpolation

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

                                                          Bytes   Clocks
LIN_INT:
              MOV            R6, [R0+]            ;       2      4
              MOV            R1, [R0]             ;       2      3
              SUB            R1, R6               ;       2      3
              MULU.w         R6, R2               ;       2      12
              MOV.b          R1H, R1L             ;       2      3
              MOVS.b         R1L,#0               ;       2      3
              ADD            R6, R1               ;       2      3
              ADD            R0, #15              ;       2      3
              MOV            R1, [R0+]            ;       2      4
              MOV            R5, [R0]             ;       2      3
              SUB            R5, R1               ;       2      3
              MULU.w         R5, R2               ;       2      12
              MOV.b          R1H, R1L             ;       2      3
              MOVS.b         R1L,#0               ;       2      3
              ADD            R1, R5               ;       2      3
              SUB            R1, R6               ;       2      3
              MULU.w         R1, R4               ;       2      12
              MOV.b          R1H, R1L             ;       2      3
              MOVS.b         R1L,#0               ;       2      3
              ADD            R6, R1               ;       2      3
              RET                                 ;       2      6

                                                          42     95 (5.94 µS)
```

Linear Interpolation (2 dim. time / 3) = 14 bytes, 1.98 µS

A1.7: Interrupt Overhead

Note: Interrupt overhead, as defined in the benchmark, applies to performance calculations. It does not consider the interrupt latency associated with completing the current instruction.

All transfers are to / from internal memory, all addresses are 16-bit long.
{
Saves 2 words on stack = 4 clks
Prefetching ISR = 3 clks
Overhead through Interrupt Controller = 3 clks (allow synch + avoid metastability)
i.e., total = 10 clks
}

```
Interrupt Accept/Return                                  0/2     10+8
JMP     rel16               ; uncond. x 2                3x2     6x2
Bxx     bit, rel8           ; Branch on bit test x 2     2x2     4.5x2
CALL    rel16               ; Long Call (PZ assumed)     3       4
RET                         ; Subroutine return          2       6
SETB    bit                 ; Set bit x 2                3x2     4x2
CLR     bit                 ; Clear bit  x 2             3x2     4x2
PUSH    Rlist (5)           ; 5 PUSH Multiple            2       15
POP     Rlist (5)           ; 5 POP Multiple             2       12
MOV     PSWL, #data8        ; imm. byte to PSWL          4       3
MOV     PSWH, #data8        ; needs 2 for 8-bit sfr      4       3
                            ; bus
                                                         41      98 (6.1 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; based on the same approach taken for 80C196, code size is 1400 bytes.

```
JMP     rel16          ; Long branch  x 100          3x100   6 x 100
CALL    rel16          ; Call  x 100 (Page 0)        3x50    4 x 50
RET                    ; Subroutine return x 100     2x100   6 x 50
Bxx     rel8           ; Condl. short branch x 100   2x200   4.5 x 200
JB/JNBbit, rel8        ; Bit test & branch x 100     2x100   4.5 x 100

                                                     1400     2,450
                                                              (153.1 µS)
```

A1.9: XA TOTALS

FUNCTION	OC*	XA		BYTES/FUNCTION
		EXEC. TIME /FUNCT.(µs)	OCCURRENCE *TIME/FUNCT.	
MPY	12	0.75	9	2
FDIV	4	3.94	15.8	18
ADD/SUB	50	0.38	19	4
CMP 24b	13	1.06	13.78	9
CAN 16b	40	0.563	22.52	5
INTPLIN	20	5.94	118.8	42
INTERR	10	6.1	61	41
BRANCH	10		153.1	

Conclusion:

An assumption is made that XA code is in first 64K (PZ) as the 80196 has a 64K address space only.

APPENDIX 2

8051 Function Implementations

8051 reference: *Single chip 8-bit microcontrollers PCB83C552*
Users manual 1988

A2.1: 80C51 Multiply 16×16

The 80C51 core performs 8-bit multiply only. A 16×16 multiply has to be done by splitting X and Y into XH, XL and YH, YL so that:

P3..P0 = (XH*256+XL)*(YH*256+YL) =
XH*YH*65536+(XH*YL+XL*YH)*256+XL*YL

```
                          Clocks  Bytes
MPY:
        MOV     R1,XH       2       3
        MOV     R2,XL       2       3
        MOV     R3,YH       2       3
        MOV     R3,YL       2       3
        MOV     A,R2        1       1       ;XL
        MOV     B,R4        1       3       ;YL
        MUL     AB          4       1
        MOV     P0,A        1       2       ; Lowest multiply result byte
        MOV     A,R4        1       1       ;YL
        MOV     R4,B        2       3       ; XL*YL upper byte (*256)
        MOV     B,R1        2       3       ;XH
        MUL     A,B         4       1       ;XL*YL
        ADD     A,R4        1       1
        MOV     R4,A        1       1       ;upper (Xl*YL)+lower(XH*YL) in R2
        MOV     A,B         1       2
        ADDC    A,#0        1       2
        XCH     A,R2        1       1       ;XL upper (XH*YL) in R2
        MOV     B,R3        3       2       ;YH
        MUL     A,B         4       1       ;XL*YH
        ADD     A,R4        1       1
        MOV     P1,A        1       2
        MOV     A,B         1       2
        ADDC    A,R2        1       1
        MOV     R2,A        1       1
        MOV     A,R3        1       1
        MOV     B,R1        2       3
        MUL     AB          4       1
        ADD     A,R2        1       1
        MOV     P2,A        1       2
        MOV     A,B         1       2
        ADDC    A,#0        1       2
        MOV     P3,A        1       2

        Total              50      58
```

50 clocks = 50*12 = 600 clocks (37.5 µs @ 16.0 Mhz)

8051 MPY 16×16 (MPY Bytes) 50 clocks = 37.5 µs / 58 bytes

A2.2: 8051 Divide (16/16) “floating point”

Divide (R6, R7) (dividend) by (R4,R5) (divisor) with (R0) bits after the fraction point.

Alignment of MSBits of operand in R6.7 and R4.7 using R0 as bit counter.

```
                        Clocks  Bytes
FDV:
        INC     R0              1       1
        INC     R0              1       1
        MOV     R3,#0           1       2
        MOV     R2,#0           1       2
        CLR     C               1       1
        CLR     F0              1       2
        MOV     A,R4            1       1
        JB      ACC.7, L2       2       3
        JNZ     L1              2       2
        MOV     A,R5            1       1
        JZ      LX              2       2
L1:
        MOV     A,R5            1       1
        RCL     A               1       1
        MOV     R5,A            1       1
        MOV     A,R4            1       1
        RCL     A               1       1
        MOV     R4,A            1       1
        INC     R0              1       1
        JNB     ACC.7, L1       2       3
L2:
        MOV     A,R6            1       1
        JB      ACC.7, L6       2       3
L3:
        MOV     A,R7
        RLC     A               1       1
        MOV     R7,A            1       1
        MOV     A,R6            1       1
        RLC     A               1       1
        MOV     R6,A            1       1
        DJNZ    R0, $+4         2       2
        AJMP    LX              2=0     3
        JNB     ACC.7,L3        2       3
        AJMP    L6              2       3
L4:
        MOV     A,R3
        RLC     A               1       1
        MOV     R3,A            1       1
        MOV     A,R2            1       1
        RLC     A               1       1
        MOV     R2,A            1       1
        JNC     L5              2       2
        MOV     R2,#0FFH        1       1
        MOV     R3,#0FFH        1       1
        SJMP    LX              1       1
L5:
        CLR     C               1       1
        MOV     A, R7           1       1

        RLC     A               1       1
        MOV     R7,A            1       1
        MOV     A,R6            1       1
        RLC     A               1       1
        MOV     R6,A            1       1
        JNC     L5              1       1
        MOV     F0,C            1       2
```

```
L6:
       CLR     C               1       1
       MOV     A,R7            1       1
       SUBB    A,R4            1       1
       JNC     L7              2       2
       JNB     F0,L8           2       3
       CPL     C               1       1
L7:
       MOV     R6,A            1       1
       MOV     A, R1           1       1
       MOV     R7,A            1       1
L8:
       CPL     C               1       1
       DJNZ    R0,L4           2       2
       MOV     A,R3            1       1
       ADD     A,#0            1       1
       MOV     R3,A            1       1
       MOV     A,R2            1       1
       ADD     A,#0            1       2
       MOV     R2,A            1       1
LX:
       RET                     2       1
```

Total 96 bytes 13 branch instructions (=35 bytes== 36%)

Timing : 3 divide cases :		**subtracts**	**shifts**	**total**	**average**
1. R0=0E, 8–bit/14 bit	-->	15–8+2=9	8+2=9	32 subtracts	11
2. R0=08, 12–bit/14 bit	-->	8–4+4=8	4+4=8	17+11 shifts	6+4
3. R0=10, 11–bit/12 bit	-->	16–5+4=15	5+5		

17+4*9+6*10+(15.5+10*31.5)+8=451.5 clocks = 338.6 μS

8051 UFDIV 16/16 (sub/sft) : 338.6 clocks = 451.5 μs, 96 bytes.

A2.3: 8051 Add/Sub

```
                        Bytes  Clocks
ADS:
       CLR     C               1       1
       MOV     A,X0            1       2
       SUBB    A,Y0            1       2
       MOV     Z0,A            1       2
       MOV     A,X1            1       2
       SUBB    A,Y1            1       2
       MOV     Z0,A            1       2
       MOV     A,X2            1       2
       SUBB    A,#0            1       2
       MOV     Z2,A            1       2

                               10      19
```

8051 ADD/SUB in reg file 10 clocks = 7.5 μs, 19 bytes

8051 CMP enabling JZ JNZ JC JNC

The 8051 decisions made with branches are one of these three :

```
       JC      lt              2       2
       JC                      2       2
       JZ      eq              2       2
       JC                      2       2
       JNZ     gt              2       2
```

8051 compare decision branches take average : 10/3 clocks => 2.5 μs

A2.4: 8051 CMP 3 byte compare

```
                          Bytes  Clocks
CM3:
        CLR     C           1      1
        MOV     A,X2        1      2
        SUBB    A,Y2        1      2
        MOV     R0,A        1      2
        MOV     A,X1        1      2
        SUBB    A,Y1        1      2
        ORL     R0,A        1      2
        MOV     A,X2        1      2
        SUBB    AY2         1      2
        Orl     A,R0        1      2
        Jcc     xxxx        3.33   3.33

                            10     19
```

8051 CMP 3 byte data in reg file 13.3 clocks = 9.975 μs, 22.3 bytes

A2.5: 8051 2-byte CAN compares

```
                          Bytes  Clocks
CAN:
        MOV     DPTR,aX1    2      3          ; one compare src in X–RAM
        MOVX  A, @DPTR      1      2
        CJNE    A,Y1        1      2
        MOV     DPTR,aX2    1      2          ; one compare src in X–RAM
        MOVX  A,@DPTR       1      2
        CJNE    A,Y2        2      3

                            12     14
```

8051 CAN CMP XRAM/Direct 9 μs, 14 bytes

A2.6: 8051 2-dimensional interpolation

```
At the start registers are prepared
A       : position in table (x+16*y)
DPTR    : Start address of table (aligned at 256 byte boundary)
R0      : x–fraction   R1      : y–fraction
Result  : ACC registers used : ACC,R0,R1,R2,R4,R5,R6
```

		Clocks	Bytes	
INT:				
	MOV DPL,A	1	2	;POS X,Y
	ACALL GVAL	2	2	
	MOV R4,A	1	1	
	MOV A,DPL	1	2	
	ADD A,#15	1	2	
	MOV DPL,A	1	2	
	ACALL GVAL	2	2	
	MOV REG6,R4	1	2	
	MOV B,R1	1	2	
	ACALL INTP	1	2	
	RET	2	1	
GVAL:				
	MOVX A,@DPTR	2	1	
	MOV R6,A	1	1	
	INC DPL	2	1	
	MOVX A,@DPTR	2	1	
	MOV B,R0	1	2	
INTP:				
	CLR SF	1	2	
	CLR C	1	1	
	SUBB A,R6	1	1	
	JNC INT1	2	2	
	CPL A	1	1	
	INC A	1	1	
	SETB SF	1	2	
INT1:				
	MUL A,B	4	1	
	XCH A,B	1	2	
	CLR C	1	1	
	RRC A	1	1	
	XCH A,B	1	2	
	XCH A,B	1	2	
	CLR C	1	1	
	RRC A	1	1	
	XCH A,B	1	2	
	JB SF,INT2	2	3	
	ADDC A,R6	1	2	
	RET	2	1	
INT2:				
	XCH A,R6	1	2	
	SUBB A,R6	1	2	
	RET	2	1	

Total 2-dim. interpolation : 15+2*(8+24)+24=103 clocks = 77.25 μs, 59 bytes

8051 Linear interpolation : (2–dim. intp time /3) = 103/3 =25.75 μs, 20 bytes

A2.7: 8051 Interrupt Overhead

		Bytes	Clocks
a.	interrupt	2	2 (vector)
	RETI	2	1
b.	AJMP 2*	4	4
c.	JB 2*	4	6
d.	ACALL	2	2
	RET	2	1
e.	SETB 2*	2	4
	CLRB 2*	2	4
f.	POP 5*	10	10
	PUSH 5*	10	10
g.	MOV 1*	2	2
		42	**46**

8051 Interrupt Overhead 42 clocks = 31.5 μs

A2.8: 8051 Program Overhead

TYPE	OCCURRENCE	8051		BYTES	
LJMP/JMP	100	2	200	3	300
LCALL/JSR	100	2	200	3	300
Jcc/Bcc	200	2	400	3	600
JB/JBN	100	2	200	3	300
total cylces μsec			1000 750		1500

A2.9: 8051 Totals

FUNCTION	OC*	8051	
		EXEC	*OC
1. MPY	12	37.5	450
2. FDIV	4	338.6	1354.4
3. ADD/SUB	50	7.5	375
4. CMP 24b	13	9.98	129.74
5. CAN 16b	40	9	360
6. INTPLIN	20	25.8	516
7. INTERR	10	31.5	315
8. BRANCH	10		750

8051 totals : 4250.14 μs
including 20% statistics : 5,100.2 μs

APPENDIX 3

68000 implementations

68000 reference: *SC68000 microprocessor users manual*

(Motorola copyright; Philips edition 12NC: 4822 873 30116)

A3.1: 68000 16x16 Multiply

The 68000 can use 1 <ea> with MUL and move a long word result.

```
MUL     R0,R1                   2       70
```

total: 4.375 µs, 2bytes

A3.2: Floating point division 16:16

```
(R0) Accuracy,          (R1)/(R2)   R1 result

                                Bytes  Clocks
FDV:
        EXT.l   R1                2       4
        TST     R2                2       4
        BEQ     L1                2       10/8
        ASL     R0,R1             2       32
        DIVU    R2,R1             2       140
        BVC     L2                2       10/8
L1:
        MOVI    #-1,R1            2       4
L2:
        RTS                       2       16
```

total : 214 clocks or 13.375 µs, 16 bytes

A3.3: Add/Sub

```
                                Bytes  Clocks
ADDS:
        MOV.l   A,R0              6       20
        ADD.l   R0,C              6       48
```

total : 44 clocks or 2.75 µs, 12 bytes

A3.4: Compares 24 (=32) bit

```
                                Bytes  Clocks
CMPl:
        MOV.l   X,R0              6       20
        CMP.l   Y,Rn              6       22
        BLT/EQ/GT (av) 2          9
```

total : 51 clocks or 3.19 µs, 14 bytes

A3.5: CAN move and compares (16-bit)

```
                                Bytes  Clocks
CMPw:
        MOV.w   X,R0              6       16
        CMP.w   Y,Rn              6       18
        BLT/EQ/GT (av)            2       9
```

total : 43 clocks or 2.69 µs, 14 bytes

A3.6: 2-dimensional interpolation

A0 : table position, R0 : fraction1, R1 : fraction 2 , R2 : result, R3, R4

```
                                    Bytes  Clocks
CMPw:
        MOV.w           (A0), R2      2      8
        ADDQ.l          #1,A0         2      8
        MOV.l           (A0), R3      2      8
        SUB.w           R2,R3         2      4
        MULu            R0,R3         2      74
        ASR.l           #8,R3         2      28
        ADD.w           R3,R2         2      4
        ADDI.l          #15,A0        4      8
        MOV.w           (A0),R3       2      8
        ADDQ.l          #1,A0         2      8
        MOV.w           (A0),R4       2      8
        SUB.w           R3,R4         2      4
        MULu            R0,R4         2      74
        ASR.l           #8,R4         2      28
        ADD.w           R4,R3         2      4
        SUB.w           R2,R3         2      4
        MULu            R1,R3         2      40
        ASR.l           #8,R3         2      22
        ADD.w           R3,R2         2      4
        RTS                           2      16
```

total : 362 clocks or 22.62 µs, 42 bytes

Linear interpolation is 2-dim. interpolation /3 :

1-dim. interpolation 7.54 µs, 14 bytes

A3.7: 68000 Interrupt Overhead

```
                                Clocks  Bytes
a.      interrupt                 44      4
        RETI                      20      2
b.      JMP 2*                    24      24
c.      BTST+BNE 2*               60      16
d.      BSR                       18      4
        RTS                       16      2
e.      BSET/BCLR 4*              96      24
f.      MOVEM 2* n=5              64      12
g.      MOVI    #xx,CCR           8       4
                                  350     92
```

68000 INTerrupt overhead 350 clocks = 21.87 µs, 92 bytes

A3.8: 68000 Program Overhead

For the 68000, the JB/JBN branches have to be constructed :

```
                                    Clocks  Bytes
MOV.w           ABS.l,Rn              12      6
ANDI.w          #bitmask,Rn           8       4
BEQ/BNE         rel.address           10      2
```

total JB/JNB execution : 34 clocks, 12 bytes

Now the absolute (estimated) branch time can be calculated, taking the core difference in account.

TYPE	OCCURRENCE	68000		BYTES	
LJMP/JMP	100	12	1200	6	600
LCALL/JSR	100	20	2000	8	800
Jcc/Bcc	200	10	2000	2	400
JB/JBN	100	34	3400	12	1200
total cycles µsec			8600 537.5		3000

A3.9: 68000 Totals

FUNCTION	OC*	68000	
		EXEC	*OC
1. MPY	12	4.4	52.8
2. FDIV	4	13.4	53.6
3. ADD/SUB	50	2.75	137.5
4. CMP 24b	13	3.2	41.6
5. CAN 16b	40	2.7	216
6. INTPLIN	20	7.5	150
7. INTERR	10	21.9	219
8. BRANCH	10		537.5

68000 totals : 1,300 µs
including 20% statistics : 1,560 µs

APPENDIX 4

80C196 function implementations

80C196 reference: *Embedded controller handbook* vol II-16 bit

A4.1: 80C196 Unsigned multiply P=X*Y (16x16)

```
                          Bytes  Clocks
MUL             R0,R1       3      28
```

total: 1.75 μs, 3 bytes

A4.2: Floating point division 16:16

```
(R0) Accuracy, (R4)/(R8)   R4 result
```

```
                          Bytes  Clocks
FDV:
        EXT     R4          2      4
        AND     R8,#FFFF    4      5
        JE      L1          2      8/4
        SHLL    R4,R0       3      20
        DIVU    R8,R4       3      24
        JNV     L2          2      4/8
L1:
        LD      R4,#FFFF    2      5
L2:
        RET                 1      11
```

total: 76 clocks or 9.5 μs, 19 bytes

A4.3: Add/Sub

```
                          Bytes  Clocks
ADDS:
        SUB     R5,R1,R3    3      5
        SUBB    R4,R0,R2    4      5
```

total : 10 clocks or 1.25 μs, 7 bytes

A4.4: 80C196 “3-byte compare”

```
                          Bytes  Clocks
        CMP     Rn,Y1       5      9
        BNE     L1          2      4/8
        CMP     Rm,Y2       5      9
L1:
        BLT/EQ/GT  (av)     2      4/8
```

Average total: 34 clocks or 4.25 μs, 14 bytes

A4.5: CAN move and compares (16-bit)

```
                          Bytes  Clocks
        CMP     Rx,Y        4      9
        BLT/EQ/GT (av)      2      6
```

total : 15 clocks or 2.5 μs, 6 bytes

A4.6: 80C196 2-dimensional interpolation using in-line linear interpolations

R0 : table position, R2=fraction1, R4=fraction2, R6=result, R8, R10

```
                                  Bytes  Clocks
        LD              R6,[R0]+    3      6
        LD              R8,[R0]+    3      5
        SUB             R8,R6       3      4
        MULU            R8,R2       3      14
        SHRAL           R8,#8       3      15
        ADD             R6,R8       3      4
        ADD             R0,#15      4      6
        LD              R8,[R0]+    3      6
        LD              R6,[R0]     3      5
        SUB             R10,R8      3      4
        MULU            R10,R2      3      14
        SHRAL           R10,#8      3      15
        ADD             R8,R10      3      4
        SUB             R8,R6       3      4
        MULU            R8,R4       3      14
        SHRAL           R8,#8       3      15
        ADD             R6,R8       3      4
        RET                         1      14
```

total : 153 clocks or 19.1 µs, 53 bytes

Linear interpolation is 2-dim. interpolation /3 :

1-dim. interpolation 6.4 µs, 18 bytes

A4.7 80C196 Interrupt Overhead

```
                              Clocks  Bytes
a.      interrupt /RTE          27      2
b.      LJMP 2*                 14      6
c.      JB      2*av.7          14      6
d.      CALL/RTS                22      4
e.      BSET/BCLR 4*            28      16
f.      POP 5*                  40      10
        PUSH    5*              55      10
g.      MOVI    #xx,CCR         5       4

                                205     58
```

80C196 INTerrupt overhead 205 clocks = 12.8 µs, 58 bytes

A4.8: 80C196 Program Overhead

TYPE	OCCURRENCE	68000		BYTES	
LJMP	100	7	700	3	300
LCALL/RET	100	22	2200	4	400
Jcc/Bcc	200	7	1400	2	400
JB/JBN	100	7	700	3	300
total cycles µsec			6000 375		1400

80C196 totals : 958.1 µs
including 20% statistics : 1150 µs

FUNCTION	OC*	80C196	
		EXEC	*OC
1. MPY	12	1.75	21
2. FDIV	4	9.5	38
3. ADD/SUB	50	1.25	62.5
4. CMP 24b	13	4.25	55.2
5. CAN 16b	40	1.88	150.4
6. INTPLIN	20	6.4	128
7. INTERR	10	12.8	128
8. BRANCH	10		375

BIT MANIPULATION

Copy a bit from one location to another in memory. Complement the bit in the new location

Note: Assumed that memory is on-chip and directly addressed.

Bit “x” of mem0 needs to be copied to bit “y” of mem1.

XA

```
        CLR     C               ; clear Carry                       3       4
        ORL     C, /bitm        ; compl. bit and save in C          3       4
        MOV     bitn, C         ; move mem0.x –> mem1.y             3       4

                                                                    9       12
                                                                            (0.75 µS)
```

Intel 80C196

Note : States = clock (period)/ 2

Move complement of bit “m” to “n” in memory

R3 = memory byte having bit “m”
R4 = memory byte having bit “n”
R0 = Used as bit-mask register
R1 = position of “m” in mem0
R2 = position of “n” in mem1

```
                                                                    Bytes   States
        LD      R0, 1                   ; Load 1 in Reg
        SHLB    R0, R2                  ; position of bit “n        3       16
                                        ; in R2
        NOTB    R0                      ; complement                2       4
        JBC     R3,bitm, L1             ; test bit “m” polarity     3       7 (av)
        ANDB    R4, R0                  ; reset “n” if “m” = 0      3       4
L1:
        ORB             R4, R0          ; set “m” otherwise         3       either/or

                                                                    14      31 (3.88 µS)
```

Motorola 68000

```
                                                                    Bytes   States
        BTST    bitm                    ; Test bit                  2       4
        BEQ     L1                      ; Branch if reset           2       6
        BCLR    bitn                    ; Test bit and clear (~m = 0) 2     4
        .......
        .......
L1:     BFSET   bitn                    ; Test bit and set (~m = 1) 2       either/or

                                                                    8       14  (0.88 µS)
```

8051 Bit-test

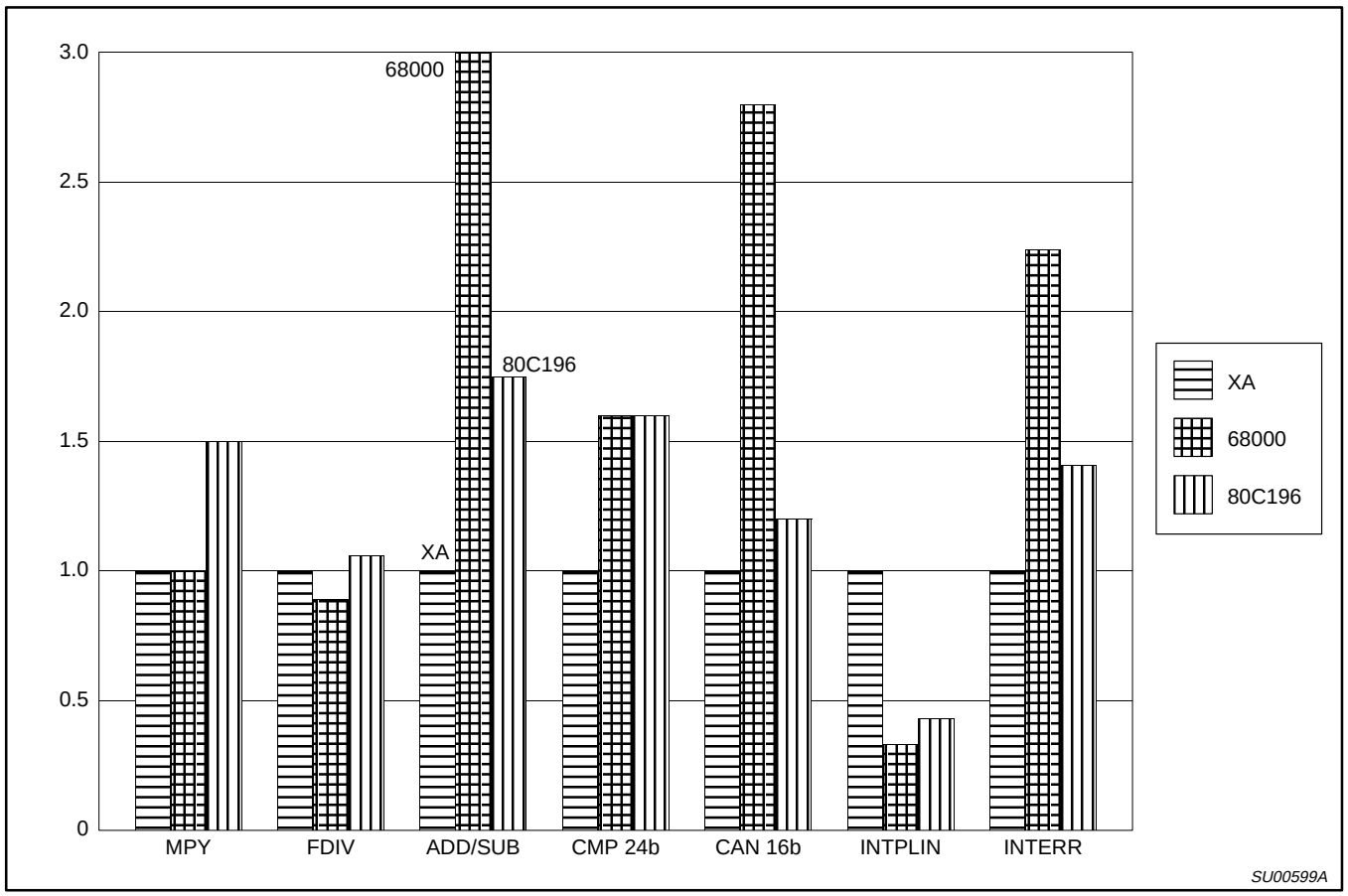
```
        MOV     C, bitm                                             2       12
        CPL     C                                                   1       12
        MOV     bitn, C                                             2       24

                                                                    5       48 (3.0 µS)
```

XA CODE DENSITY RESULTS

Graph showing performance with respect to 68000, and 80C196 cores normalized with respect to XA. The 80C51 is included just for reference.

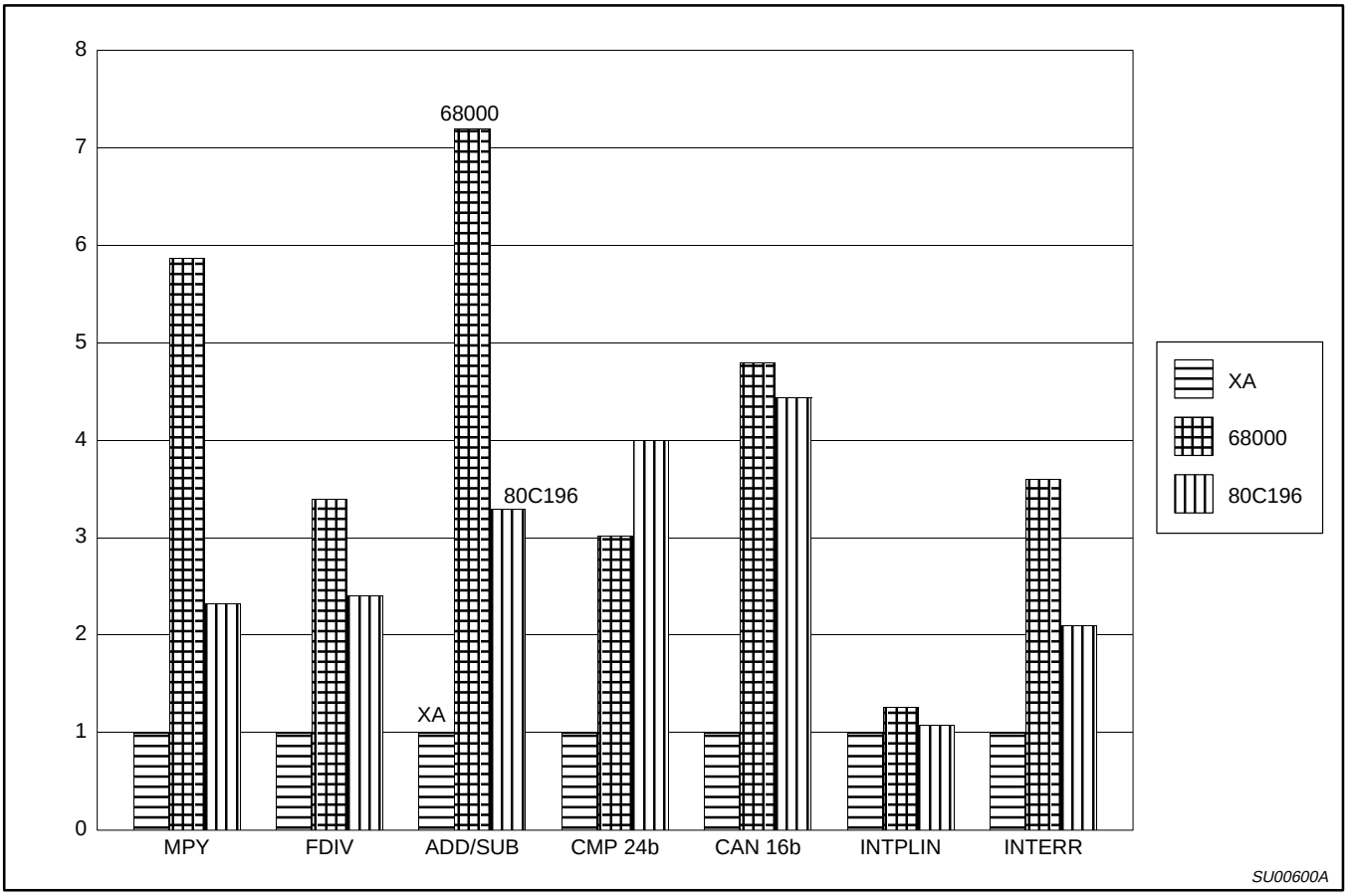
	XA	68000	80C196	8051
MPY	1	1	1.5	1
FDIV	1	0.89	1.06	5.33
ADD/SUB	1	3	1.75	2.5
CMP 24b	1	1.6	1.6	1
CAN 16b	1	2.8	1.2	1.5
INTPLIN	1	0.33	0.43	0.33
INTERR	1	2.24	1.41	1.71



XA EXECUTION TIME RESULTS

Graph showing performance with respect to 68000, and 80C196 cores normalized with respect to XA. The 80C51 is included just for reference.

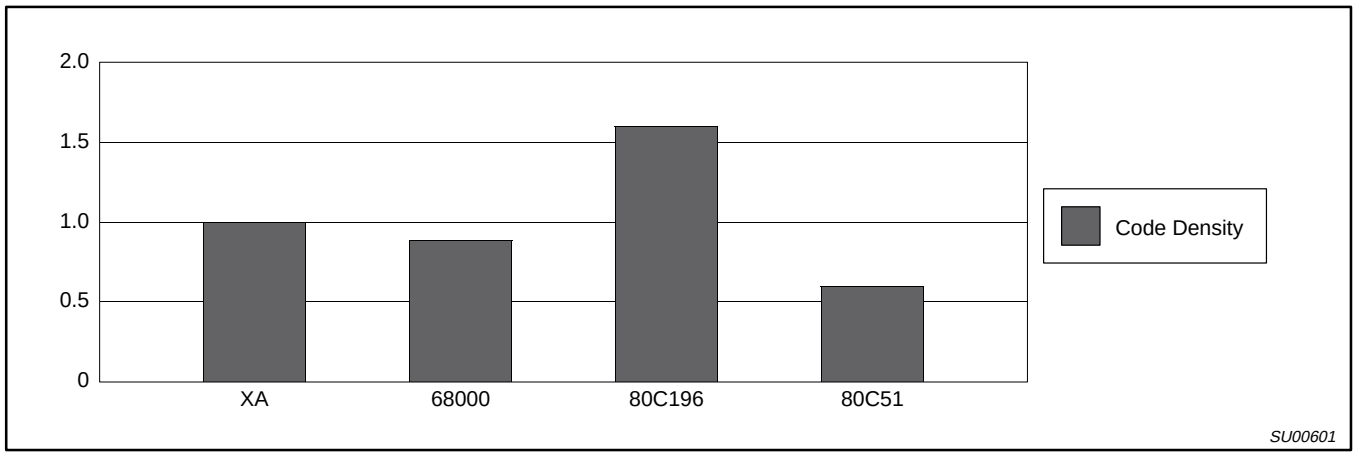
	XA	68000	80C196	8051
MPY	1	5.87	2.33	50
FDIV	1	3.4	2.41	86
ADD/SUB	1	7.2	3.3	19.74
CMP 24b	1	3.02	4	9.41
CAN 16b	1	4.8	4.44	15.98
INTPLIN	1	1.26	1.08	4.34
INTERR	1	3.6	2.1	5.16



BIT TEST BENCHMARK: CODE DENSITY NORMALIZED WITH XA (=1.0)

The 80C51 is shown here only for reference.

	XA	68000	80C196	8051
Code Density	1	0.89	1.6	0.6



BIT TEST BENCHMARK: EXECUTION TIME NORMALIZED WITH XA (=1.0)

The 80C51 is shown here only for reference.

	XA	68000	80C196	8051
Execution Time	1	1.2	5.2	4