

MSP430 Competitive Benchmarking

Greg Morton, Kripasagar Venkat

MSP430 Products

ABSTRACT

This application report contains the results from benchmarking the MSP430 against microcontrollers from other vendors. IAR Embedded Workbench™ development platform was used to build and execute, in simulation mode, a set of simple math functions. These functions were executed on each microcontroller to benchmark different aspects of the microcontrollers' performance. In addition, both Dhrystone and Whetstone analyses have been included.

1 Embedded Benchmark Suite

This section has results for simple and less intense math functions. [Figure 1](#) shows the total code size in bytes for each microcontroller with no optimization and with full optimization.

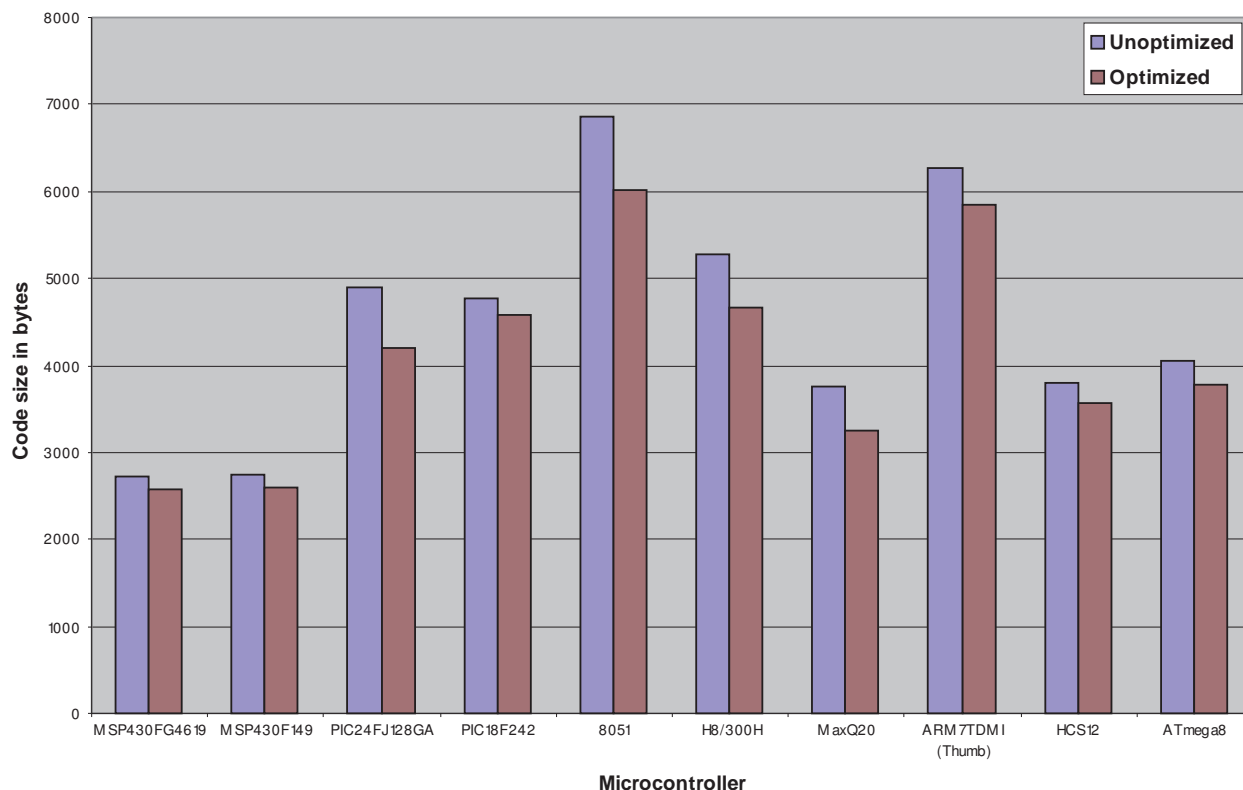


Figure 1. Total Code Size for Embedded Benchmark Suite

Figure 2 shows the total cycle count for each microcontroller with no optimization and with full optimization. Note that some architectures use an internal CPU clock divider. In these architectures, the total execution time for the code is the clock divider multiplied by the total instruction cycle count. This clock divider is not included in the total cycle count numbers presented here. See Appendix A.1 for more information regarding CPU clock dividers.

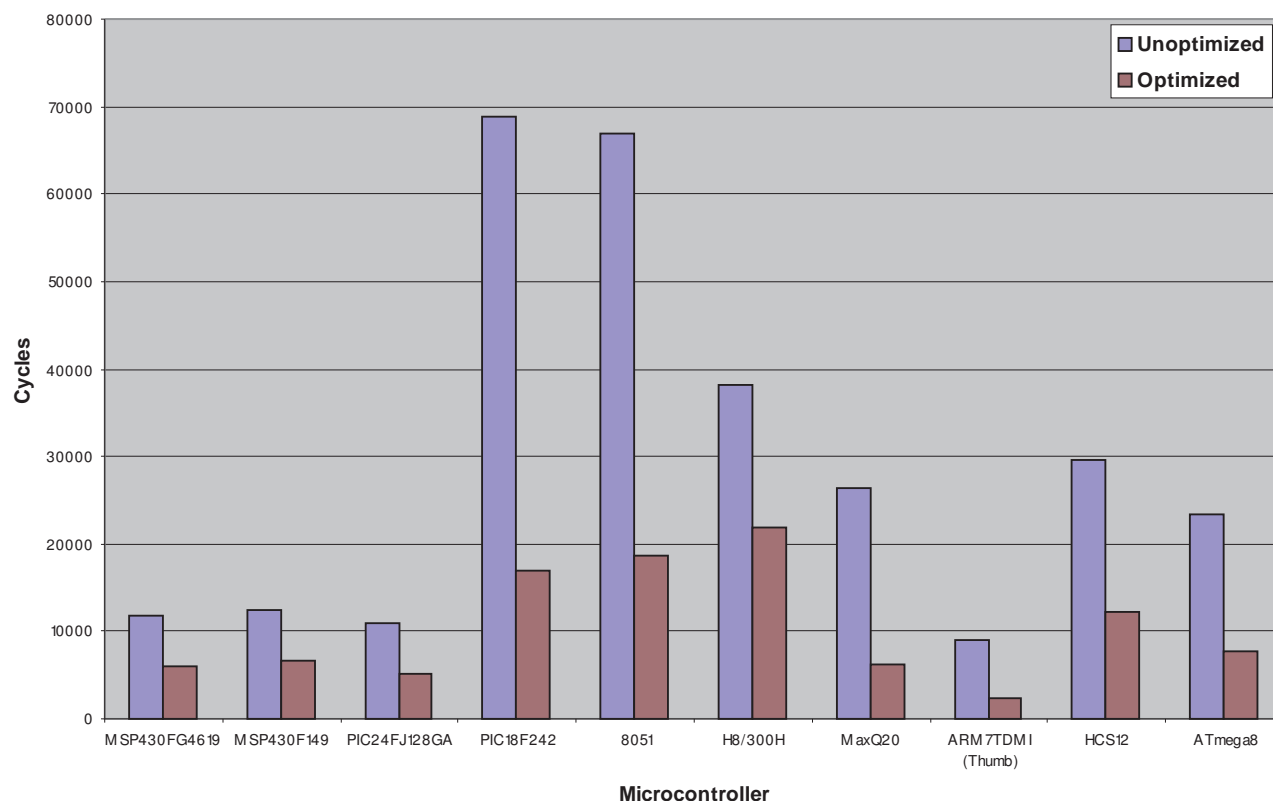


Figure 2. Total Instruction Cycles for Embedded Benchmark Suite

The MSP430FG4619 differs in architecture from the MSP430F149 and has the MSP430X CPU. The MSP430X CPU can address up to 1-MB address range without paging. In addition, the MSP430X CPU has fewer interrupt overhead cycles and fewer instruction cycles, in some cases, than the MSP430 CPU. The MSP430X CPU is completely backward compatible with the MSP430 CPU. Code size and cycle count values are shown in [Appendix A](#).

[Table 1](#) shows the total code size and the total instruction counts for each microcontroller, normalized against the MSP430FG4619, for the Embedded Benchmark Suite.

Table 1. Normalized Results for Embedded Benchmark Suite

MICROCONTROLLER	TOTAL CODE SIZE		TOTAL INSTRUCTION CYCLE COUNT	
	UNOPTIMIZED	FULLY OPTIMIZED	UNOPTIMIZED	FULLY OPTIMIZED
MSP430FG4619	1.00	1.00	1.00	1.00
MSP430F149	1.01	1.003	1.04	1.09
PIC24FJ128GA	1.80	1.62	0.93	0.84
PIC18F242	1.75	1.77	5.79	2.79
8051	2.53	2.33	5.64	3.08
H8/300H	1.94	1.81	3.22	3.60
MaxQ20	1.38	1.26	2.22	1.04
ARM7TDMI	2.30	2.26	0.76	0.37
HCS12	1.40	1.38	2.49	2.03
ATmega8	1.49	1.46	1.97	1.27

[Appendix B](#) includes the code names and a brief description of their functionality used for this benchmarking.

2 Math-Intense Benchmark Suite

In order to exhibit the performance of each of the microcontrollers under intense math operations, the benchmarking of a Finite Impulse Response (FIR) filter that requires multiply and accumulate (MAC) is included in this report. Also included are the results of the Dhrystone and Whetstone benchmarks. Code size and cycle count values are shown in [Appendix A](#).

[Figure 3](#) shows the code size for each microcontroller, with no optimization and full optimization, for the implementation of an FIR filter.

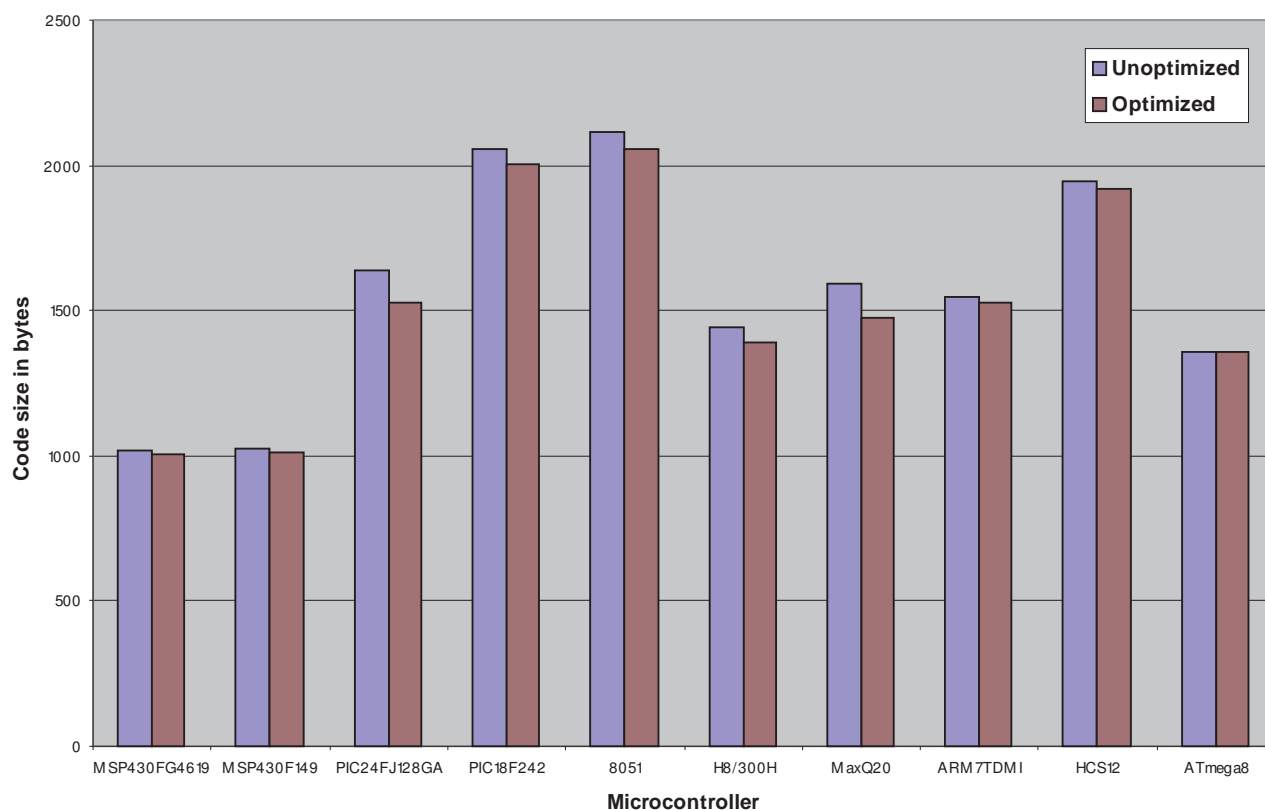


Figure 3. Code Size For FIR Filter Operation

Figure 4 shows the cycle count for each microcontroller, with no optimization and full optimization, for the implementation of an FIR filter.

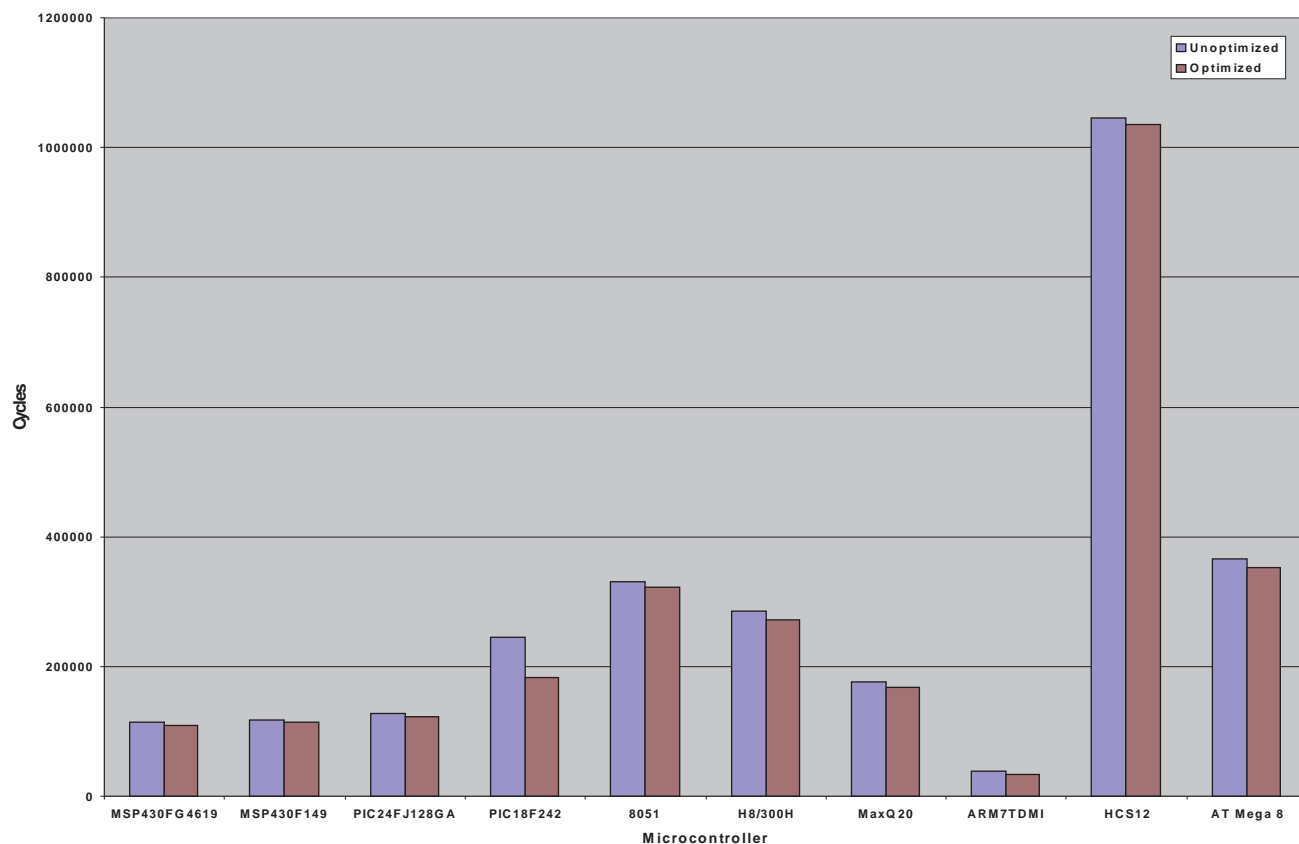


Figure 4. Cycle Count For FIR Filter Operation

Table 2 shows the total code size and the total instruction cycle count for each microcontroller, normalized against the MSP430FG4619, for the FIR filter operation.

Table 2. Normalized Results for FIR Filter Operation

MICROCONTROLLER	TOTAL CODE SIZE		TOTAL INSTRUCTION CYCLE COUNT	
	UNOPTIMIZED	FULLY OPTIMIZED	UNOPTIMIZED	FULLY OPTIMIZED
MSP430FG4619	1.00	1.00	1.00	1.00
MSP430F149	1.006	1.006	1.04	1.04
PIC24FJ128GA	1.61	1.52	1.12	1.13
PIC18F242	2.02	1.99	2.17	1.68
8051	2.08	2.04	2.92	2.97
H8/300H	1.41	1.38	2.52	2.51
MaxQ20	1.56	1.47	1.56	1.54
ARM7TDMI	1.52	1.52	0.33	0.31
HCS12	1.91	1.90	9.23	9.54
ATmega8	1.33	1.35	3.23	3.25

The Dhrystone benchmark gauges the performance of a microcontroller in handling pointers, structures, and strings. Figure 5 shows the code size for each microcontroller, with no optimization and full optimization, for the implementation of this code.

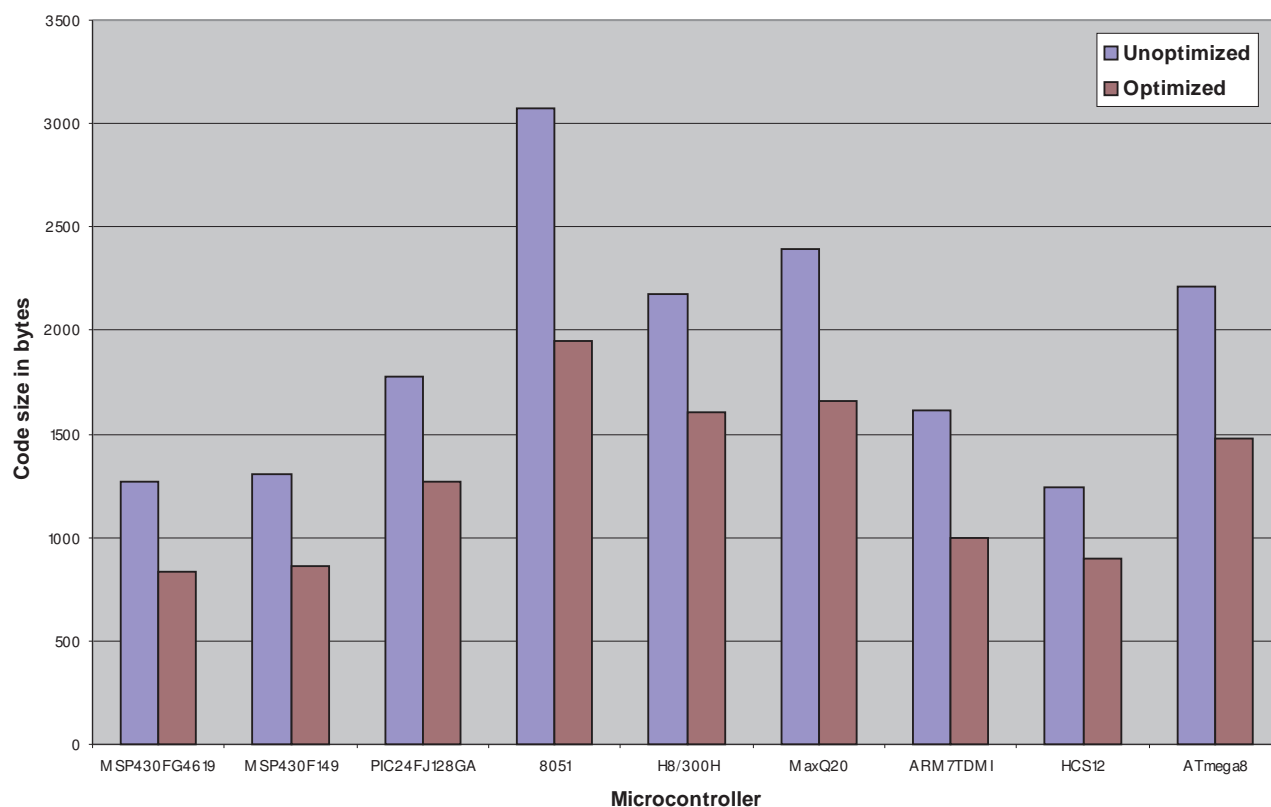


Figure 5. Code Size In Bytes For Dhrystone Analysis

Figure 6 shows the cycle count for each microcontroller, with no optimization and full optimization, for the Dhrystone analysis.

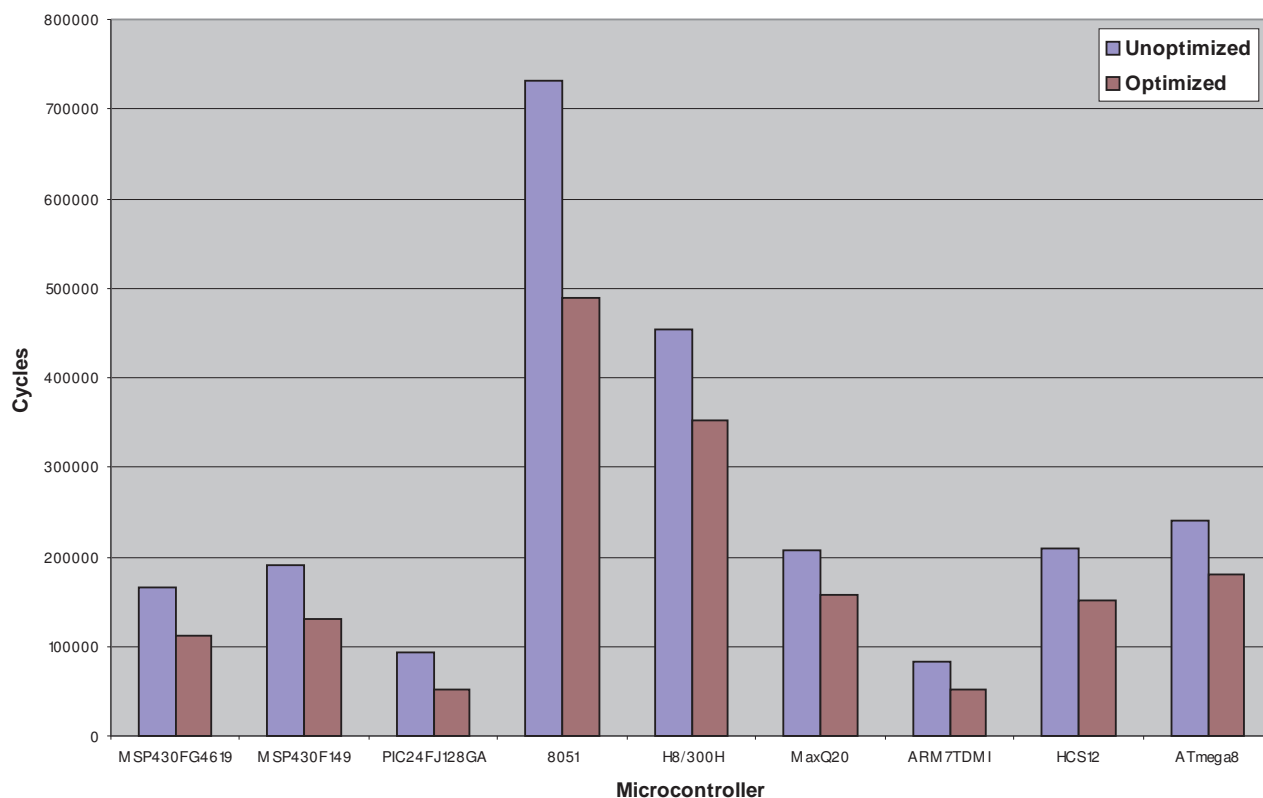


Figure 6. Cycle Count For Dhrystone Analysis

Table 3 shows the total code size and the total instruction cycle count for each microcontroller, normalized against the MSP430FG4619, for the Dhrystone analysis.

Table 3. Normalized Results for Dhrystone Analysis

MICROCONTROLLER	TOTAL CODE SIZE		TOTAL INSTRUCTION CYCLE COUNT	
	UNOPTIMIZED	FULLY OPTIMIZED	UNOPTIMIZED	FULLY OPTIMIZED
MSP430FG4619	1.00	1.00	1.00	1.00
MSP430F149	1.03	1.03	1.15	1.16
PIC24FJ128GA	1.40	1.51	0.56	0.47
PIC18F242 ⁽¹⁾	***	***	***	***
8051	2.42	2.32	4.40	4.36
H8/300H	1.71	1.92	2.73	3.15
MaxQ20	1.88	1.98	1.25	1.41
ARM7TDMI	1.27	1.19	0.50	0.47
HCS12	0.98	1.07	1.25	1.36
ATmega8	1.74	1.76	1.44	1.61

⁽¹⁾ The 30-day trial version of the IAR compiler did not support the memory model required for Dhrystone analysis.

The Whetstone benchmark attempts to measure the performance of both integer and floating-point arithmetic in a variety of scientific functions. The code has a mixture of C functions to calculate the sine, cosine, exponent, etc., of fixed-point and floating-point numbers. Figure 7 shows the code size for each microcontroller, with no optimization and full optimization, for the implementation of this code.

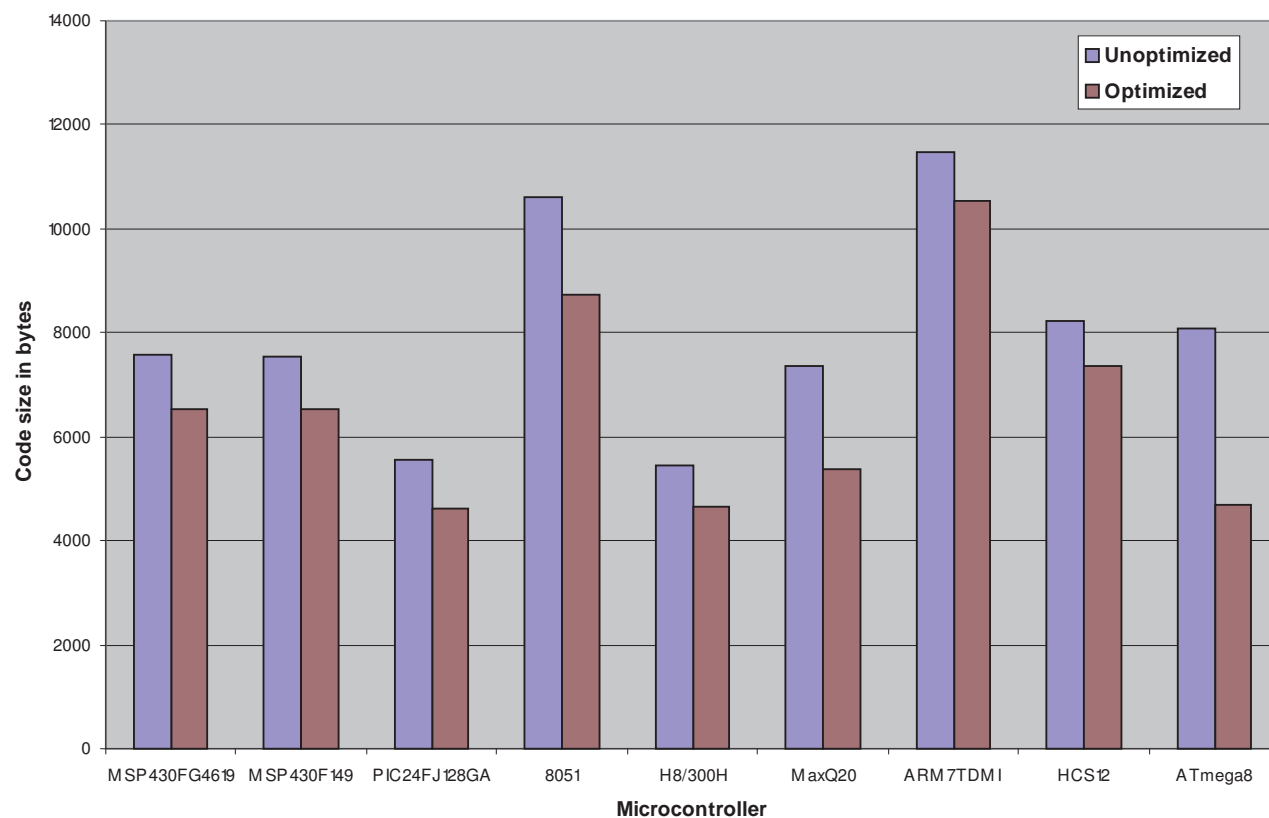


Figure 7. Code Size In Bytes For Whetstone Analysis

Figure 8 shows the cycle count for each microcontroller, with no optimization and full optimization, for the Whetstone analysis.

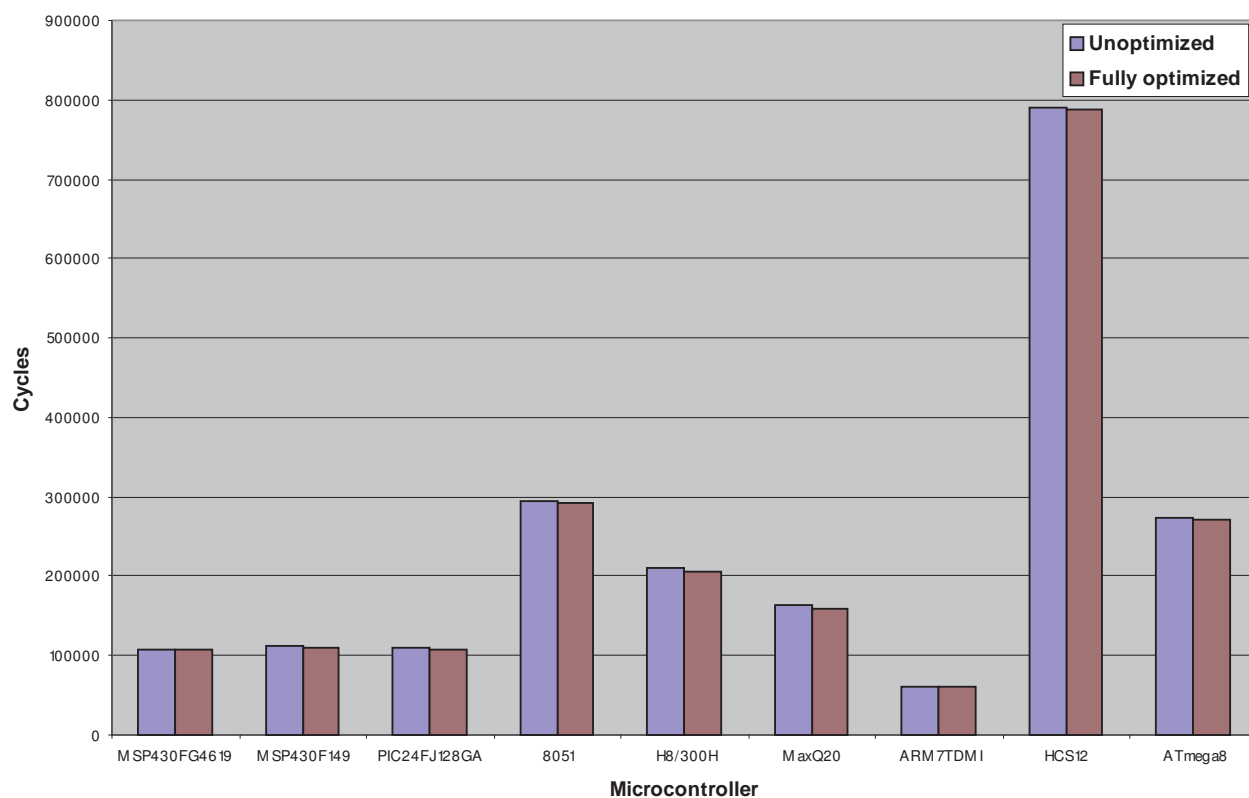


Figure 8. Cycle Count For Whetstone Analysis

Table 4 shows the total code size and the total instruction counts for each microcontroller, normalized against the MSP430FG4619, for the Whetstone analysis.

Table 4. Normalized Results for Whetstone Analysis

MICROCONTROLLER	TOTAL CODE SIZE		TOTAL INSTRUCTION CYCLE COUNT	
	UNOPTIMIZED	FULLY OPTIMIZED	UNOPTIMIZED	FULLY OPTIMIZED
MSP430FG4619	1.00	1.00	1.00	1.00
MSP430F149	1.00	0.99	1.03	1.03
PIC24FJ128GA	0.74	0.70	1.02	1.01
PIC18F242 ⁽¹⁾	***	***	***	***
8051	1.40	1.33	2.72	2.73
H8/300H	0.72	0.71	1.94	1.92
MaxQ20	0.98	0.82	1.50	1.48
ARM7TDMI	1.52	1.61	0.57	0.56
HCS12	1.09	1.13	7.30	7.36
ATmega8	1.07	0.72	2.54	2.53

⁽¹⁾ The 30-day trial version of the IAR compiler did not support the memory model required for Whetstone analysis.

Appendix B includes the code and a brief description of functionality used for this benchmarking.

Appendix A Background Information

This appendix includes the actual values for all of the benchmarking discussed in this report.

A.1 Processor Clock vs Instruction Cycle Clock Considerations

MCU architectures have different associations between the processor input clock frequency and the actual instruction cycle clock frequency. Ideally, one processor clock cycle fed into the CPU would result in one instruction being executed. However, in some cases, an additional CPU internal clock divider is used (Table 5). Then, multiple processor clock cycles are necessary to execute a single instruction. This is important to consider when determining the system clock frequency that is needed to achieve a given task. Note that higher clock frequencies generally also lead to a higher power consumption due to increased CMOS logic switching losses.

Table A-1. Table 5. CPU Clock Divider

Microcontroller	CPU Clock Divider
MSP430FG4619	1
MSP430F149	1
Microchip PIC24FJ128GA	2
Microchip PIC18F242	4
Generic 8051	1...12 ⁽¹⁾
Renesas H8/300H	2
MaxQ20	1
ARM7TDMI	1
Freescale HCS12	2
Atmel ATmega8	1

- ⁽¹⁾ 8051 architectures typically use a divider of 12. However, some improved architectures can execute a subset of instructions in as little as one clock cycle per instruction.

A.2 Compiler Information And Detailed Results

The C compiler bundled with IAR Embedded Workbench Integrated Development Environment (IDE) was used to build the benchmarking applications. Evaluation copies of the IDE were obtained for each microcontroller from IAR Systems' web site located at <http://www.iar.com>. Table A-2 lists the C compiler version used to build the benchmarking applications for each microcontroller.

All applications were built with compiler optimization set to "none" and to "full". This was done to utilize the compiler's ability to build efficient code, which has had a great impact on the results.

Table A-2. C Compiler Versions

Microcontroller	IAR C Compiler Version
MSP430FG4619	3.41A
MSP430F149	3.41A
Microchip PIC24FJ128GA	2.02 ⁽¹⁾
Microchip PIC18F242	3.10A
Generic 8051	7.20C
Renesas H8/300H	1.53I
MaxQ20	1.13C
ARM7TDMI	4.31A
Freescale HCS12	3.10A
Atmel ATmega8	4.12A

- ⁽¹⁾ For this device, the current Microchip MPLAB C30 compiler was used. An IAR compiler for the PIC24x was not available at the time of publishing this application note.

The following pages include the actual values for all of the benchmarking discussed in this report.

Table A-3 and Table A-4 show the code size in bytes for each of the microcontrollers for every math operation without optimization and with full optimization, respectively.

Table A-3. Code Size In Bytes Without Optimization For Simple Math Operations

APPLICATION	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	236	232	345	174	266	400	326	684	95	152
8-Bit Matrix	120	126	450	368	499	492	348	416	217	394
8-Bit Switch	220	200	486	238	305	498	200	532	197	378
16-Bit Math	176	172	333	266	478	398	240	684	107	210
16-Bit Matrix	136	166	558	834	693	572	460	432	301	532
16-Bit Switch	220	198	480	342	519	534	186	532	215	424
32-Bit Math	272	268	462	486	1050	646	316	644	324	352
Floating-Point Math	1134	1132	1266	1322	2346	1176	1200	1868	2082	1096
Matrix Multiplication	204	248	519	732	707	554	480	476	270	518
Total	2718	2742	4899	4762	6863	5270	3756	6268	3808	4056

Table A-4. Code Size In Bytes With Full Optimization For Simple Math Operations

APPLICATION	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242 ⁽¹⁾	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	214	210	297	170	233	344	230	636	83	134
8-Bit Matrix	106	106	366	324	398	412	252	392	188	354
8-Bit Switch	218	198	393	208	305	444	192	452	162	350
16-Bit Math	162	158	285	286	452	352	204	636	76	198
16-Bit Matrix	110	130	465	692	504	482	328	396	262	434
16-Bit Switch	218	196	390	282	493	478	184	452	174	382
32-Bit Math	258	254	375	542	909	574	288	620	323	342
Floating-Point Math	1122	1120	1194	1400	2190	1104	1172	1832	2082	1088
Matrix Multiplication	176	220	426	676	536	482	398	428	219	490
Total	2584	2592	4191	4580	6020	4672	3248	5844	3569	3772

⁽¹⁾ For some functions, the 30-day trial version of the IAR compiler produced larger code sizes with full optimization than it did with no optimization.

Table A-5 and Table A-6 show the cycle count for each of the microcontrollers for each math operation without optimization and with full optimization, respectively.

Table A-5. Cycle Count Without Optimization For Simple Math Operations

APPLICATION	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242 ⁽¹⁾	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	250	261	107	141	212	240	175	87	97	134
8-Bit Matrix	2370	2497	2927	7310	14898	10228	6196	2122	6858	2523
8-Bit Switch	33	32	76	49	112	96	42	51	51	39
16-Bit Math	223	233	108	332	542	254	201	102	108	288
16-Bit Matrix	3140	3270	3183	26533	23868	11252	9012	2890	8650	9506
16-Bit Switch	32	31	74	87	314	102	35	51	54	45
32-Bit Math	569	589	564	1259	3854	520	440	109	267	750
Floating-Point Math	771	795	789	1049	3339	1548	644	205	5508	1663
Matrix Multiplication	4500	4706	3203	32096	19856	14018	9624	3424	8034	8417
Total	11888	12414	11031	68856	66995	38258	26369	9041	29627	23365

⁽¹⁾ The 30-day trial version of IAR compiler for some functions did produce larger numbers with full optimization, as compared to no optimization.

Table A-6. Cycle Count With Full Optimization For Simple Math Operations

APPLICATION	MSP430FG4619	MSP430F149	PIC24FJ128GA	PIC18F242	8051	H8/300H	MaxQ20	ARM7TDMI	HCS12	ATmega8
8-Bit Math	233	243	75	136	176	152	130	64	68	110
8-Bit Matrix	875	1009	1051	2193	2590	4362	1140	475	1559	984
8-Bit Switch	32	31	61	49	112	62	38	20	46	38
16-Bit Math	210	219	73	339	526	172	183	79	60	266
16-Bit Matrix	811	945	1115	6461	4294	4746	1508	475	2073	1488
16-Bit Switch	31	30	60	87	318	66	34	20	41	44
32-Bit Math	556	575	510	1284	2622	388	425	97	235	731
Floating-Point Math	762	786	741	1085	2127	1416	629	187	5470	1654
Matrix Multiplication	2550	2762	1384	5283	5880	10468	2214	839	2732	2396
Total	6060	6600	5070	16917	18645	21832	6301	2256	12284	7711

Table A-7 shows the code size in bytes and cycle count and for each of the microcontrollers for every math operation without optimization and with full optimization.

Table A-7. FIR, Dhrystone, and Whetstone Code Size and Cycle Counts

MCU	FIR FILTER ⁽¹⁾				DHRYSTONE				WHETSTONE			
	CODE SIZE		CYCLES		CODE SIZE		CYCLES		CODE SIZE		CYCLES	
	UNOPT	OPT	UNOPT	OPT	UNOPT	OPT	UNOPT	OPT	UNOPT	OPT	UNOPT	OPT
MSP430FG4619	1020	1008	113308	108527	1272	838	166457	111929	7562	6544	108093	107040
MSP430F149	1026	1014	118170	113399	1310	864	190834	130104	7524	6524	111102	109837
PIC24FJ128GA	1638	1530	127125	122692	1779	1266	93920	52081	5571	4605	109807	108619
PIC18F242	2058	2006	245704	182210	***	***	***	***	***	***	***	***
8051	2116	2056	330640	321781	3075	1946	732532	488193	10613	8723	294309	291836
H8/300H	1440	1392	285580	271964	2173	1607	454518	352510	5432	4656	209370	205910
MaxQ20	1592	1478	176720	167583	2393	1661	207905	157965	7376	5392	162541	158945
ARM7TDMI	1548	1528	37827	33114	1616	1000	83798	52352	11488	10532	61600	60444
HCS12	1945	1917	1045982	1035394	1244	900	208648	152212	8238	7370	788966	787635
ATmega8	1356	1358	365837	352894	2210	1474	240320	179834	8090	4694	274586	270991

⁽¹⁾ The FIR filter code has been modified for correct operation from the previous version which lacked the MAC operations.

Appendix B Benchmarking Applications

B.1 Benchmarking Applications

In order to benchmark various aspects of a microcontroller's performance, the following set of simple applications was executed in simulation mode for each microcontroller.

8-bit_math.c — source file containing three math functions. One function performs addition of two 8-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

16-bit_math.c — source file containing three math functions. One function performs addition of two 16-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

32-bit_math.c — source file containing three math functions. One function performs addition of two 32-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

floating_point_math.c — source file containing three math functions. One function performs addition of two floating-point numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

8-bit_switch_case.c — source file with one function containing a switch statement having 16 cases. An 8-bit value is used to select a particular case. The "main()" function calls the "switch" function with an input parameter selecting the last case.

16-bit_switch_case.c — source file with one function containing a switch statement having 16 cases. A 16-bit value is used to select a particular case. The "main()" function calls the "switch" function with an input parameter selecting the last case.

8-bit_2-dim_matrix.c — source file containing 3 two-dimensional arrays containing 8-bit values—one of which is initialized. The "main()" function copies values from array 1 to array 2, then from array 2 to array 3.

16-bit_2-dim_matrix.c — source file containing 3 two-dimensional arrays containing 16-bit values—one of which is initialized. The "main()" function copies values from array 1 to array 2, then from array 2 to array 3.

matrix_multiplication.c — source file containing code, which multiplies a 3×4 matrix by a 4×5 matrix.

fir_filter.c — source file containing code that calculates the output from a 17-coefficient tap filter using simulated ADC input data.

dhry.c — source file containing code, which does the Dhrystone analysis.

whet.c — source file containing code, which does the Whetstone analysis.

B.2 Benchmarking Application Source Code

The following are the C source-code files for the benchmarking applications used in this document.

8-bit Math.c

```

/*****
*
*      Name      : 8-bit Math
*      Purpose   : Benchmark 8-bit math functions.
*
*****/

typedef unsigned char UInt8;

UInt8 add(UInt8 a, UInt8 b)
{
    return (a + b);
}

UInt8 mul(UInt8 a, UInt8 b)
{
    return (a * b);
}

UInt8 div(UInt8 a, UInt8 b)
{
    return (a / b);
}

void main(void)
{
    volatile UInt8 result[4];

    result[0] = 12;
    result[1] = 3;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}

```

8-bit 2-dim Matrix.c

```

/*****
*
*      Name      : 8-bit 2-dim Matrix
*      Purpose   : Benchmark copying 8-bit values.
*
*****/

typedef unsigned char UInt8;

const UInt8 m1[16][4] = {
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12},
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12},
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12},
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12}
};

void main (void)
{
    int i, j;
    volatile UInt8 m2[16][4], m3[16][4];

    for(i = 0; i < 16; i++)
    {
        for(j=0; j < 4; j++)
        {
            m2[i][j] = m1[i][j];
            m3[i][j] = m2[i][j];
        }
    }
    return;
}

```

8-bit Switch Case.c

```

/*****
*
*      Name      : 8-bit Switch Case
*      Purpose   : Benchmark accessing switch statement using 8-bit value.
*
*****/

typedef unsigned char UInt8;

UInt8 switch_case(UInt8 a)
{
    UInt8 output;

    switch (a)
    {
        case 0x01:
            output = 0x01;
            break;
        case 0x02:
            output = 0x02;
            break;
        case 0x03:
            output = 0x03;
            break;
        case 0x04:
            output = 0x04;
            break;
        case 0x05:
            output = 0x05;
            break;
        case 0x06:
            output = 0x06;
            break;
        case 0x07:
            output = 0x07;
            break;
        case 0x08:
            output = 0x08;
            break;
        case 0x09:
            output = 0x09;
            break;
        case 0x0a:
            output = 0x0a;
            break;
        case 0x0b:
            output = 0x0b;
            break;
        case 0x0c:
            output = 0x0c;
            break;

        case 0x0d:
            output = 0x0d;
            break;
        case 0x0e:
            output = 0x0e;
            break;
        case 0x0f:
            output = 0x0f;
            break;
        case 0x10:
            output = 0x10;
            break;
    } /* end switch*/
}

```

Benchmarking Application Source Code

```
        return (output);  
    }  
  
void main(void)  
{  
    volatile UInt8 result;  
  
    result = switch_case(0x10);  
    return;  
}
```

16-bit Math.c

```

/*****
*
*      Name      : 16-bit Math
*      Purpose   : Benchmark 16-bit math functions.
*
*****/

typedef unsigned short UInt16;

UInt16 add(UInt16 a, UInt16 b)
{
    return (a + b);
}

UInt16 mul(UInt16 a, UInt16 b)
{
    return (a * b);
}

UInt16 div(UInt16 a, UInt16 b)
{
    return (a / b);
}

void main(void)
{
    volatile UInt16 result[4];

    result[0] = 231;
    result[1] = 12;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}

```

16-bit 2-dim Matrix.c

```

/*****
*
*      Name      : 16-bit 2-dim Matrix
*      Purpose   : Benchmark copying 16-bit values.
*
*****/

typedef unsigned short UInt16;

const UInt16 m1[16][4] = {
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234},
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234},
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234},
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234}
};

void main(void)
{
    int i, j;
    volatile UInt16 m2[16][4], m3[16][4];

    for(i = 0; i < 16; i++)
    {
        for(j = 0; j < 4; j++)
        {
            m2[i][j] = m1[i][j];
            m3[i][j] = m2[i][j];
        }
    }
    return;
}

```

16-bit Switch Case.c

```

/*****
*
*      Name      : 16-bit Switch Case
*      Purpose   : Benchmark accessing switch statement using 16-bit value.
*
*****/

typedef unsigned short UInt16;

UInt16 switch_case(UInt16 a)
{
    UInt16 output;

    switch (a)
    {
        case 0x0001:
            output = 0x0001;
            break;
        case 0x0002:
            output = 0x0002;
            break;
        case 0x0003:
            output = 0x0003;
            break;
        case 0x0004:
            output = 0x0004;
            break;
        case 0x0005:
            output = 0x0005;
            break;
        case 0x0006:
            output = 0x0006;
            break;
        case 0x0007:
            output = 0x0007;
            break;
        case 0x0008:
            output = 0x0008;
            break;
        case 0x0009:
            output = 0x0009;
            break;
        case 0x000a:
            output = 0x000a;
            break;
        case 0x000b:
            output = 0x000b;
            break;

        case 0x000c:
            output = 0x000c;
            break;
        case 0x000d:
            output = 0x000d;
            break;
        case 0x000e:
            output = 0x000e;
            break;
        case 0x000f:
            output = 0x000f;
            break;
        case 0x0010:
            output = 0x0010;
    }
}

```

Benchmarking Application Source Code

```
        break;
    } /* end switch*/
    return (output);
}

void main(void)
{
    volatile UInt16 result;

    result = switch_case(0x0010);
    return;
}
```


32-bit Math.c

```

/*****
*
*      Name      : 32-bit Math
*      Purpose   : Benchmark 32-bit math functions.
*
*****/

#include <math.h>

typedef unsigned long UInt32;

UInt32 add(UInt32 a, UInt32 b)
{
    return (a + b);
}

UInt32 mul(UInt32 a, UInt32 b)
{
    return (a * b);
}

UInt32 div(UInt32 a, UInt32 b)
{
    return (a / b);
}

void main(void)
{
    volatile UInt32 result[4];

    result[0] = 43125;
    result[1] = 14567;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}

```

Floating-point Math.c

```

/*****
*
*      Name      : Floating-point Math
*      Purpose   : Benchmark floating-point math functions.
*
*****/

float add(float a, float b)
{
    return (a + b);
}

float mul(float a, float b)
{
    return (a * b);
}

float div(float a, float b)
{
    return (a / b);
}

void main(void)
{
    volatile float result[4];

    result[0] = 54.567;
    result[1] = 14346.67;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}

```

Matrix Multiplication.c

```

/*****
*
*      Name      : Matrix Multiplication
*      Purpose   : Benchmark multiplying a 3x4 matrix by a 4x5 matrix.
*                  Matrix contains 16-bit values.
*
*****/

typedef unsigned short UInt16;

const UInt16 m1[3][4] = {
    {0x01, 0x02, 0x03, 0x04},
    {0x05, 0x06, 0x07, 0x08},
    {0x09, 0x0A, 0x0B, 0x0C}
};

const UInt16 m2[4][5] = {
    {0x01, 0x02, 0x03, 0x04, 0x05},
    {0x06, 0x07, 0x08, 0x09, 0x0A},
    {0x0B, 0x0C, 0x0D, 0x0E, 0x0F},
    {0x10, 0x11, 0x12, 0x13, 0x14}
};

void main(void)
{
    int m, n, p;
    volatile UInt16 m3[3][5];

    for(m = 0; m < 3; m++)
    {
        for(p = 0; p < 5; p++)
        {
            m3[m][p] = 0;

            for(n = 0; n < 4; n++)
            {
                m3[m][p] += m1[m][n] * m2[n][p];
            }
        }
    }
    return;
}

```

Benchmarking Application Source Code

FIR Filter.c

```

/*****
*
*      Name      : FIR Filter
*      Purpose   : Benchmark an FIR filter. The input values for the filter
*                  is an array of 51 16-bit values. The order of the filter
*                  17.
*
*****/

#ifdef MSP430
#include "msp430x14x.h"
#endif
#include <math.h>
#define FIR_LENGTH 17

const float COEFF[FIR_LENGTH] =
{
-0.000091552734, 0.000305175781, 0.004608154297, 0.003356933594, -0.025939941406,
-0.044006347656, 0.063079833984, 0.290313720703, 0.416748046875, 0.290313720703,
0.063079833984, -0.044006347656, -0.025939941406, 0.003356933594, 0.004608154297,
0.000305175781, -0.000091552734};

/* The following array simulates input A/D converted values */

const unsigned int INPUT[] =
{
0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000,0x0000, 0x0000, 0x0000, 0x0000,
0x0400, 0x0800, 0x0C00, 0x1000, 0x1400, 0x1800, 0x1C00, 0x2000,
0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000, 0x0C00, 0x0800,
0x0400, 0x0400, 0x0800, 0x0C00, 0x1000, 0x1400, 0x1800, 0x1C00,
0x2000, 0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000, 0x0C00,
0x0800, 0x0400, 0x0400, 0x0800, 0x0C00, 0x1000, 0x1400, 0x1800,
0x1C00, 0x2000, 0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000,
0x0C00, 0x0800, 0x0400};

void main(void)
{
    int i, y; /* Loop counters */
    volatile float OUTPUT[36],sum;

#ifdef MSP430
    WDTCTL = WDTPW + WDTHOLD; /* Stop watchdog timer */
#endif

    for(y = 0; y < 36; y++)
    {
        sum=0;
        for(i = 0; i < FIR_LENGTH/2; i++)
        {
            sum = sum+COEFF[i] * ( INPUT[y + 16 - i] + INPUT[y + i] );
        }
        OUTPUT[y] = sum + (INPUT[y + FIR_LENGTH/2] * COEFF[FIR_LENGTH/2] );
    }
    return;
}

```

Dhry.c

```

/*****
*
*      Name      : Dhrystone
*      Purpose   : Benchmark the Dhrystone code. This benchmark is used to gauge
*                  the performance of the microcontroller in handling pointers,
*                  structures and strings.
*
*****/
#include <stdio.h>
#include <string.h>
#define LOOPS    100    /* Use this for slow or 16 bit machines */
#define structassign(d, s)    d = s

typedef enum    {Ident1, Ident2, Ident3, Ident4, Ident5} Enumeration;
typedef int     OneToThirty;
typedef int     OneToFifty;
typedef unsigned char    CapitalLetter;
typedef unsigned char    String30[31];
typedef int     Array1Dim[51];
typedef int     Array2Dim[51][51];

struct Record
{
    struct Record    *PtrComp;
    Enumeration      Discr;
    Enumeration      EnumComp;
    OneToFifty       IntComp;
    String30         StringComp;
};

typedef struct Record    RecordType;
typedef RecordType *    RecordPtr;
typedef int              boolean;

#define NULL            0
#define TRUE            1
#define FALSE          0
#define REG register

int            IntGlob;
boolean        BoolGlob;
unsigned char  Char1Glob;
unsigned char  Char2Glob;
Array1Dim      Array1Glob;
Array2Dim      Array2Glob;
RecordPtr      PtrGlb;
RecordPtr      PtrGlbNext;
RecordType     rec1, rec2;

Enumeration Func1(CapitalLetter CharPar1, CapitalLetter CharPar2)
{
    REG CapitalLetter    CharLoc1;
    REG CapitalLetter    CharLoc2;
    CharLoc1 = CharPar1;
    CharLoc2 = CharLoc1;
    if (CharLoc2 != CharPar2)
        return (Ident1);
    else
        return (Ident2);
}

boolean Func2(String30 StrParI1, String30 StrParI2)
{
    REG OneToThirty      IntLoc;
    REG CapitalLetter    CharLoc;

```

Benchmarking Application Source Code

```

        IntLoc = 1;
        while (IntLoc <= 1)
            if (Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) == Ident1)
            {
                CharLoc = 'A';
                ++IntLoc;
            }
        if (CharLoc >= 'W' && CharLoc <= 'Z')
            IntLoc = 7;
        if (CharLoc == 'X')
            return(TRUE);
        else
        {
            if (strcmp(StrParI1, StrParI2) > 0)
            {
                IntLoc += 7;
                return (TRUE);
            }
            else
                return (FALSE);
        }
    }

boolean Func3(Enumeration EnumParIn)
{
    REG Enumeration EnumLoc;
    EnumLoc = EnumParIn;
    if (EnumLoc == Ident3) return (TRUE);
    return (FALSE);
}

void Proc7(OneToFifty IntParI1, OneToFifty IntParI2, OneToFifty *IntParOut)
{
    REG OneToFifty IntLoc;
    IntLoc = IntParI1 + 2;
    *IntParOut = IntParI2 + IntLoc;
}

void Proc4(void)
{
    REG boolean BoolLoc;
    BoolLoc = Char1Glob == 'A';
    BoolLoc |= BoolGlob;
    Char2Glob = 'B';
}

void Proc5(void)
{
    Char1Glob = 'A';
    BoolGlob = FALSE;
}

void Proc6(Enumeration EnumParIn, Enumeration *EnumParOut)
{
    *EnumParOut = EnumParIn;
    if (! Func3(EnumParIn) )
        *EnumParOut = Ident4;
    switch (EnumParIn)
    {
        case Ident1:
            *EnumParOut = Ident1; break;
        case Ident2:
            if (IntGlob > 100) *EnumParOut = Ident1;
            else *EnumParOut = Ident4;
            break;
        case Ident3:
            *EnumParOut = Ident2; break;
        case Ident4:
            break;
    }
}

```

```

        case Ident5:    *EnumParOut = Ident3;
        }
    }

void Proc3(RecordPtr *PtrParOut)
{
    if (PtrGlb != NULL)
        *PtrParOut = PtrGlb->PtrComp;
    else
        IntGlob = 100;
    Proc7(10, IntGlob, &PtrGlb->IntComp);
}

void Proc1(RecordPtr PtrParIn)
{
    #define NextRecord (*(PtrParIn->PtrComp))
    structassign(NextRecord, *PtrGlb);
    PtrParIn->IntComp = 5;
    NextRecord.IntComp = PtrParIn->IntComp;
    NextRecord.PtrComp = PtrParIn->PtrComp;
    Proc3(&NextRecord.PtrComp);
    if (NextRecord.Discr == Ident1)
    {
        NextRecord.IntComp = 6;
        Proc6(PtrParIn->EnumComp, &NextRecord.EnumComp);
        NextRecord.PtrComp = PtrGlb->PtrComp;
        Proc7(NextRecord.IntComp, 10, &NextRecord.IntComp);
    }
    else
        structassign(*PtrParIn, NextRecord);

#undef NextRecord
}

void Proc2(OneToFifty *IntParIO)
{
    REG OneToFifty      IntLoc;
    REG Enumeration     EnumLoc;
    IntLoc = *IntParIO + 10;
    for(;;)
    {
        if (Char1Glob == 'A')
        {
            --IntLoc;
            *IntParIO = IntLoc - IntGlob;
            EnumLoc = Ident1;
        }
        if (EnumLoc == Ident1)
            break;
    }
}

void Proc8 (Array1Dim Array1Par, Array2Dim Array2Par, OneToFifty IntParI1, OneToFifty IntParI2)
{
    REG OneToFifty  IntLoc;
    REG OneToFifty  IntIndex;

    IntLoc = IntParI1 + 5;
    Array1Par[IntLoc] = IntParI2;
    Array1Par[IntLoc+1] = Array1Par[IntLoc];
    Array1Par[IntLoc+30] = IntLoc;
    for (IntIndex = IntLoc; IntIndex <= (IntLoc+1); ++IntIndex)
        Array2Par[IntLoc][IntIndex] = IntLoc;
    ++Array2Par[IntLoc][IntLoc-1];
    Array2Par[IntLoc+20][IntLoc] = Array1Par[IntLoc];
    IntGlob = 5;
}

```

Benchmarking Application Source Code

```

}

void Proc0 (void)
{
    OneToFifty          IntLoc1;
    REG OneToFifty       IntLoc2;
    OneToFifty          IntLoc3;
    REG unsigned char    CharLoc;
    REG unsigned char    CharIndex;
    Enumeration          EnumLoc;
    String30             String1Loc;
    String30             String2Loc;
    extern unsigned char  *malloc();

    long                time(long *);
    long                starttime;
    long                benchtime;
    long                nulltime;
    register unsigned int i;

    for (i = 0; i < LOOPS; ++i);
    PtrGlbNext = &rec1; /* (RecordPtr) malloc(sizeof(RecordType)); */
    PtrGlb      = &rec2; /* (RecordPtr) malloc(sizeof(RecordType)); */
    PtrGlb->PtrComp = PtrGlbNext;
    PtrGlb->Discr = Ident1;
    PtrGlb->EnumComp = Ident3;
    PtrGlb->IntComp = 40;
    strcpy(PtrGlb->StringComp, "DHRYSTONE PROGRAM, SOME STRING");
    strcpy(String1Loc, "DHRYSTONE PROGRAM, 1'ST STRING"); /*GOOF*/
    Array2Glob[8][7] = 10; /* Was missing in published program */
    for (i = 0; i < LOOPS; ++i)
    {
        Proc5();
        Proc4();
        IntLoc1 = 2;
        IntLoc2 = 3;
        strcpy(String2Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
        EnumLoc = Ident2;
        BoolGlob = ! Func2(String1Loc, String2Loc);
        while (IntLoc1 < IntLoc2)
        {
            IntLoc3 = 5 * IntLoc1 - IntLoc2;
            Proc7(IntLoc1, IntLoc2, &IntLoc3);
            ++IntLoc1;
        }
        Proc8(Array1Glob, Array2Glob, IntLoc1, IntLoc3);
        Procl(PtrGlb);
        for (CharIndex = 'A'; CharIndex <= Char2Glob; ++CharIndex)
            if (EnumLoc == Func1(CharIndex, 'C'))
                Proc6(Ident1, &EnumLoc);
        IntLoc3 = IntLoc2 * IntLoc1;
        IntLoc2 = IntLoc3 / IntLoc1;
        IntLoc2 = 7 * (IntLoc3 - IntLoc2) - IntLoc1;
        Proc2(&IntLoc1);
    }
}

void main(void)
{
    Proc0();
}

```


Whet.c

```

/*****
*
*      Name      : Whetstone
*      Purpose   : Benchmark the Whetstone code. The code focuses on scientific
*                  functions such as sine, cosine, exponents and logarithm on
*                  fixed and floating point numbers.
*
*****/
#include <math.h>
#include <stdio.h>

PA(float E[5]);
P0(void);
P3(float *X, float *Y, float *Z);

float T,T1,T2,E1[5];
int J,K,L;
float X1,X2,X3,X4;
long ptime,time0;

main ()
{
    int LOOP,I,II,JJ,N1,N2,N3,N4,N5,N6,N7,N8,N9,N10,N11;
    float X,Y,Z;
    T = .499975;
    T1 = 0.50025;
    T2 = 2.0;
    LOOP = 1;
    II = 1;
    for (JJ=1;JJ<=II;JJ++)
    {
        N1 = 0;
        N2 = 2 * LOOP;
        N3 = 2 * LOOP;
        N4 = 2 * LOOP;
        N5 = 0;
        N6 = 2 * LOOP;
        N7 = 2 * LOOP;
        N8 = 2 * LOOP;
        N9 = 2 * LOOP;
        N10 = 0;
        N11 = 2 * LOOP;

        /*      Module 1: Simple identifiers */
        X1 = 1.0;
        X2 = -1.0;
        X3 = -1.0;
        X4 = -1.0;
        if (N1!=0)
        {
            for(I=1;I<=N1;I++)
            {
                X1 = (X1 + X2 + X3 - X4)*T;
                X2 = (X1 + X2 - X3 + X4)*T;
                X3 = (X1 - X2 + X3 + X4)*T;
                X4 = (-X1 + X2 + X3 + X4)*T;
            }
        };

        /*      Module 2: Array elements */
        E1[1] = 1.0;
        E1[2] = -1.0;
        E1[3] = -1.0;
        E1[4] = -1.0;
        if (N2!=0)

```

Benchmarking Application Source Code

```

        {
            for (I=1;I<=N2;I++)
            {
                E1[1] = (E1[1] + E1[2] + E1[3] - E1[4])*T;
                E1[2] = (E1[1] + E1[2] - E1[3] + E1[4])*T;
                E1[3] = (E1[1] - E1[2] + E1[3] + E1[4])*T;
                E1[4] = (-E1[1] + E1[2] + E1[3] + E1[4])*T;
            }
        }

/*      Module 3: Array as parameter */
if (N3!=0)
{
    for (I=1;I<=N3;I++)
    {
        PA(E1);
    }
}

/*      Module 4: Conditional jumps */
J = 1;
if (N4!=0)
{
    for (I=1;I<=N4;I++)
    {
        if (J==1) goto L51;
        J = 3;
        goto L52;
L51:      J = 2;
L52:      if (J > 2) goto L53;
        J = 1;
        goto L54;
L53:      J = 0;
L54:      if (J < 1) goto L55;
        J = 0;
        goto L60;
L55:      J = 1;
L60:    }
}

/*      Module 5: Integer arithmetic */
J = 1;
K = 2;
L = 3;

if (N6!=0)
{
    for (I=1;I<=N6;I++)
    {
        J = J * (K-J) * (L-K);
        K = L * K - (L-J) * K;
        L = (L - K) * (K + J);
        E1[L-1] = J + K + L;
        E1[K-1] = J * K * L;
    }
}

/*      Module 6: Trigonometric functions */
X = 0.5;
Y = 0.5;
if (N7!=0)
{
    for (I=1;I<=N7;I++)
    {
        X=T*atan(T2*sin(X)*cos(X)/(cos(X+Y)+cos(X-Y)-1.0));
        Y=T*atan(T2*sin(Y)*cos(Y)/(cos(X+Y)+cos(X-Y)-1.0));
    }
}

```

```

    }
}

/*      Module 7: Procedure calls */
X = 1.0;
Y = 1.0;
Z = 1.0;
if (N8!=0)
{
    for (I=1;I<=N8;I++)
    {
        P3(&X,&Y,&Z);
    }
}

/*      Module 8: Array references */
J = 1;
K = 2;
L = 3;
E1[1] = 1.0;
E1[2] = 2.0;
E1[3] = 3.0;
if (N9!=0)
{
    for (I=1;I<=N9;I++)
    {
        P0();
    }
}

/*      Module 9: Integer arithmetic */
J = 2;
K = 3;
if (N10!=0)
{
    for (I=1;I<=N10;I++)
    {
        J = J + K;
        K = J + K;
        J = K - J;
        K = K - J - J;
    }
}

/*      Module 10: Standard functions */
X = 0.75;
if (N11!=0)
{
    for (I=1;I<=N11;I++)
    {
        X = sqrt(exp(log(X)/T1));
    }
}
}

PA(E) float E[5];
{
    int J1;
    J1 = 0;
L10:  E[1] = (E[1] + E[2] + E[3] - E[4]) * T;
      E[2] = (E[1] + E[2] - E[3] + E[4]) * T;
      E[3] = (E[1] - E[2] + E[3] + E[4]) * T;
      E[4] = (-E[1] + E[2] + E[3] + E[4]) / T2;
      J1 = J1 + 1;
      if ((J1 - 6) < 0) goto L10;
}

```

Benchmarking Application Source Code

```
        return;
    }

    P0()
    {
        E1[J] = E1[K];
        E1[K] = E1[L];
        E1[L] = E1[J];
        return;
    }

    P3(X,Y,Z) float *X,*Y,*Z;
    {
        float Y1;
        X1 = *X;
        Y1 = *Y;
        X1 = T * (X1 + Y1);
        Y1 = T * (X1 + Y1);
        *Z = (X1 + Y1) / T2;
        return;
    }
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
Low Power Wireless	www.ti.com/lpw	Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Copyright © 2006, Texas Instruments Incorporated