Application Report

SLAA205C–June 2006–Revised January 2009

MSP430 Competitive Benchmarking

William Goh, Kripasagar Venkat

MSP430 Applications

ABSTRACT

This application report contains the results from benchmarking the MSP430 against microcontrollers from other vendors. IAR Systems' 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 microcontroller's performance. In addition, FIR Filter, Dhrystone, and Whetstone analysis are included.

Contents

1 Embedded Benchmark Suite

This section shows results for simple and less intense math. Figure 1 compares the total code size in bytes for 8-bit and 16-bit microcontrollers with maximum optimization options for code size. The figure indicate the cumulative numbers for the entire simple math benchmarking suite. See Section A.2 for the individual numbers.

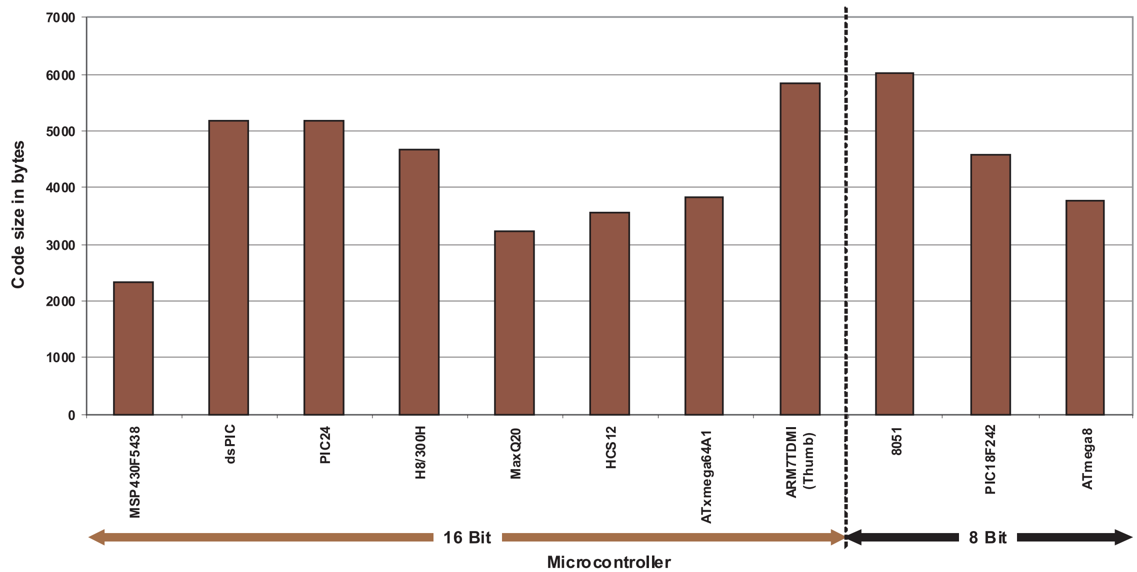

Figure 1. Total Code Size for Simple Math With 8-Bit and 16-Bit Microcontrollers

Figure 2 compares the total instruction cycle count for 8-bit and 16-bit microcontrollers with maximum optimization options for speed (cycle count).

Note: Some architectures use an internal CPU clock divider. In this case, the total execution time for the code is the clock divider multiplied by the total instruction cycle count. This clock divider is not reflected in the total instruction cycle count numbers presented here. See Section A.1 for more information regarding CPU clock dividers.

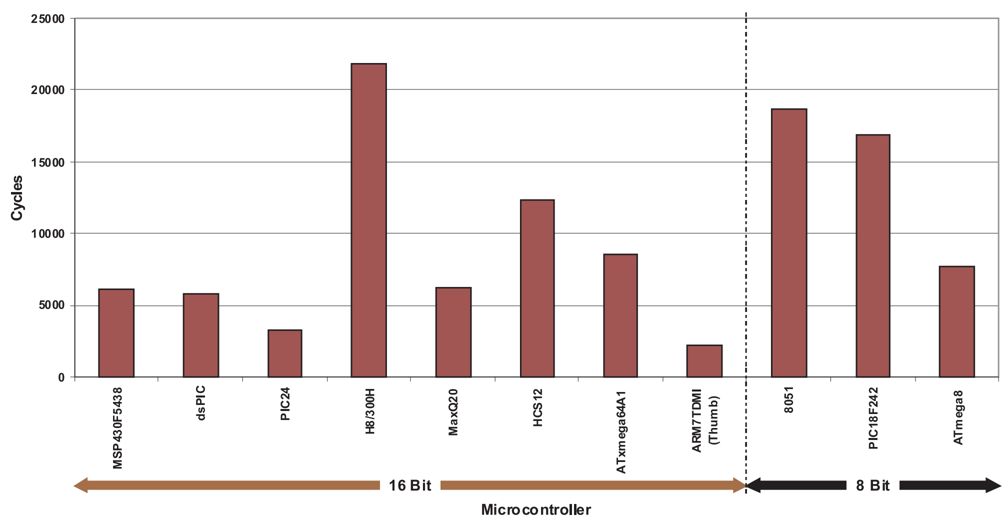

Figure 2. Total Cycle Count for Simple Math With 8-Bit and 16-bit Microcontrollers

All graphs in this document compare MSP430F5438 against the other microcontrollers. Since the results of the other MSP430 families are almost similar, they are not displayed on the graphs. However, appendix A-3 contains the comparison simulation results of 'F5438, 'FG4619, 'F149, and 'F2274. Appendix A contains simulated numbers in which the compiler settings is set to both full optimization and no optimization. The unoptimized simulated numbers are not displayed on the graphs.

Table 1 shows the total code size and the total instruction cycle counts for each microcontroller normalized against the MSP430F5438 for the Embedded Benchmark Suite.

Table 1. Normalized Results for Simple Math Operations

Microcontroller		Total Code Size	Total Instruction Cycle Count
16-Bit	MSP430F5438	1.00	1.00
	dsPIC	2.22	0.96
	PIC24	2.21	0.54
	H8/300H	2.00	3.60
	MaxQ20	1.39	1.04
	HCS12	1.53	2.02
	ATxmega64A1	1.64	1.41
	ARM7TDMI (Thumb)	2.50	0.37
8-Bit	8051	2.58	3.07
	PIC18F242	1.96	2.79
	ATmega8	1.61	1.27

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

2 Math Intense Benchmark Suite

To show 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) operations has been shown. Also, this benchmark includes results for Dhrystone and Whetstone analysis. The actual values are included in Appendix A, and the code is in Appendix B.

2.1 FIR Filter Analysis

Figure 3 compares the code size for 8-bit and 16-bit microcontrollers with maximum optimization for code size in the implementation of a FIR filter.

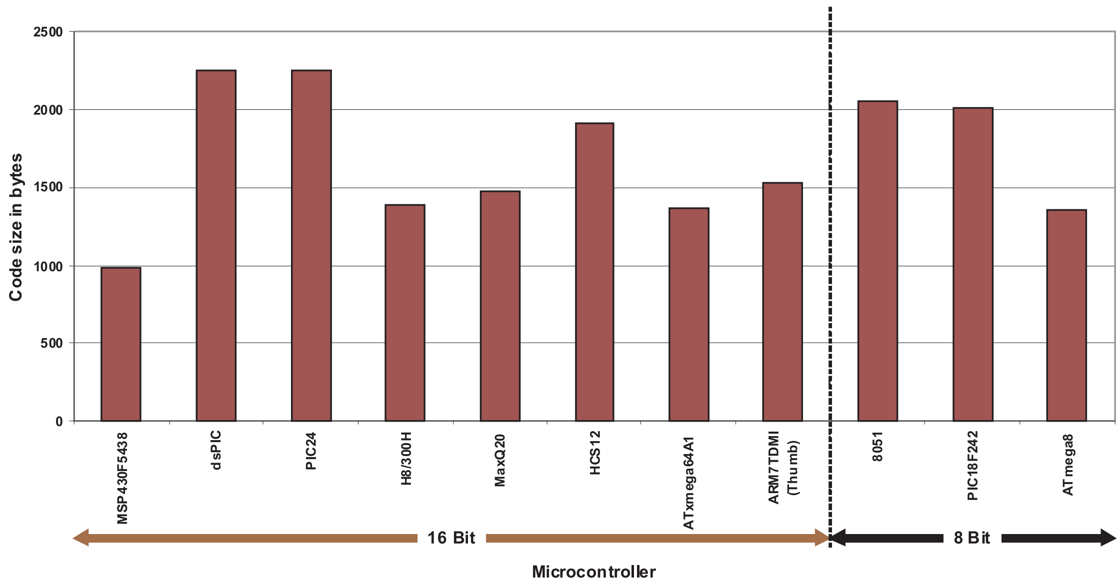

Figure 3. Code Size for FIR Filter With 8-Bit and 16-Bit Microcontrollers

Figure 4 compares the cycle count for 8-bit and 16-bit microcontrollers with maximum optimization for speed (cycle count) in the implementation of a FIR filter.

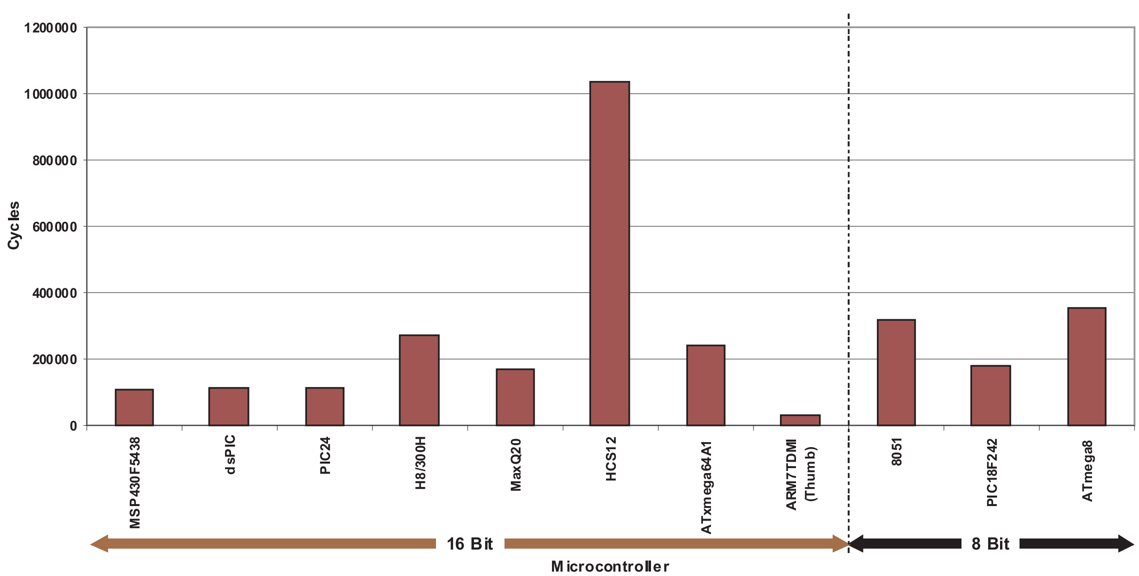

Figure 4. Cycle Count for FIR Filter With 8-Bit and 16-Bit Microcontrollers

Table 2 displays the FIR filter operation code size and the total instruction cycle count for each microcontroller normalized against the MSP430F5438.

Table 2. Normalized Results for FIR Filter Operation

Microcontroller		Total Code Size	Total Instruction Cycle Count
16-Bit	MSP430F5438	1.00	1.00
	dsPIC	2.30	1.09
	PIC24	2.30	1.07
	H8/300H	1.42	2.54
	MaxQ20	1.51	1.56
	HCS12	1.96	9.66
	ATxmega64A1	1.39	2.26
	ARM7TDMI (Thumb)	1.56	0.31
8-Bit	8051	2.10	3.00
	PIC18F242	2.05	1.70
	ATmega8	1.39	3.29

2.2 Dhrystone Analysis

Dhrystone benchmark is used to gauge the performance of the microcontroller in handling pointers, structures and strings. Figure 5 compares the code size for 8-bit and 16-bit microcontrollers with maximum optimization for code size.

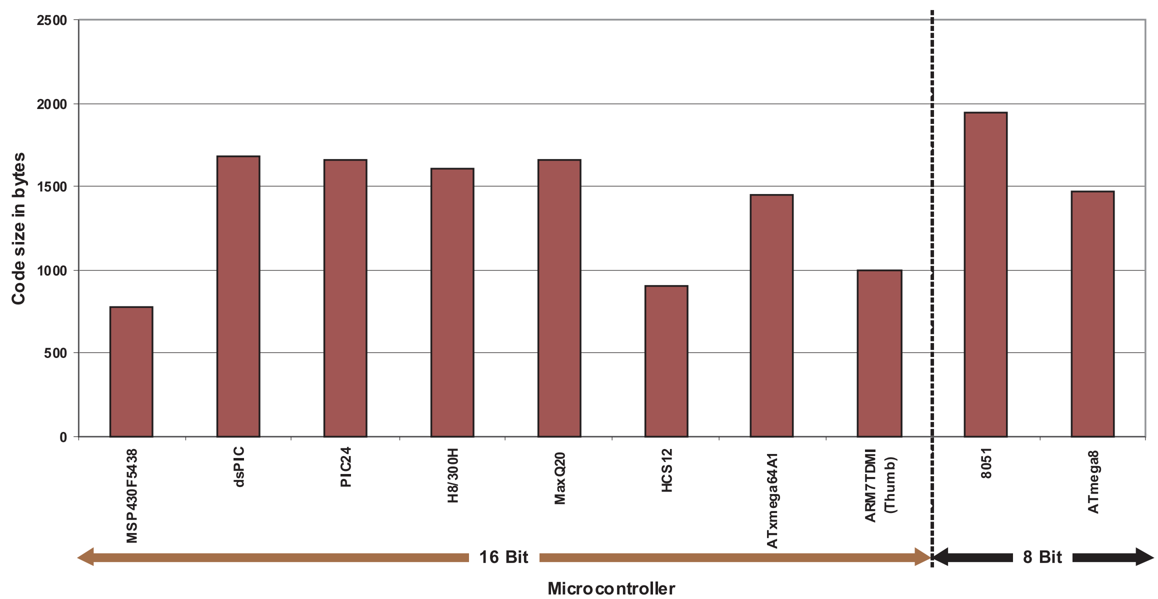

Figure 5. Code Size for Dhrystone Analysis With 8-Bit and 16-Bit Microcontrollers

Figure 6 compares the cycle count for 8-bit and 16-bit microcontrollers with maximum optimization for speed (cycle count) in the implementation for Dhrystone analysis.

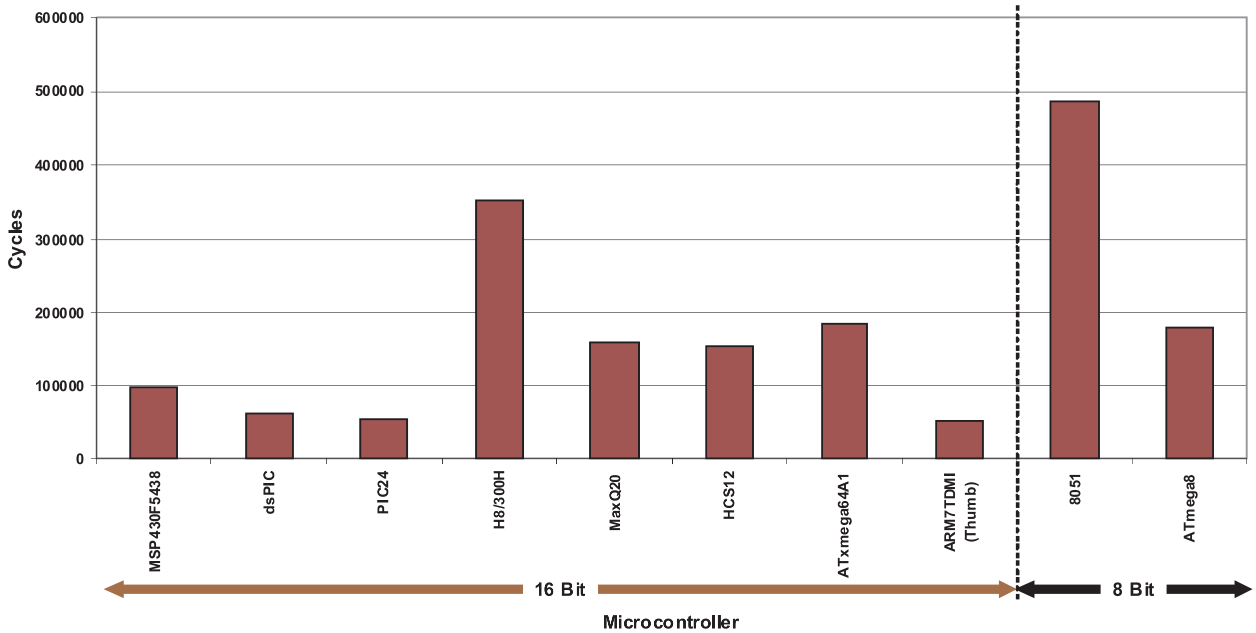

Figure 6. Cycle Count for Dhrystone Analysis With 8-Bit and 16-Bit Microcontrollers

Table 3 shows the Dhrystone analysis code size and the total instruction cycle count for each microcontroller normalized against the MSP430F5438.

Table 3. Normalized Results for Dhrystone Analysis

Microcontroller		Total Code Size	Total Instruction Cycle Count
16-Bit	MSP430F5438	1.00	1.00
	dsPIC	2.15	0.62
	PIC24	2.13	0.55
	H8/300H	2.06	3.60
	MaxQ20	2.13	1.61
	HCS12	1.15	1.55
	ATxmega64A1	1.86	1.89
	ARM7TDMI (Thumb)	1.28	0.53
8-Bit	8051	2.49	4.98
	PIC18F242 (1)	—	—
	ATmega8	1.89	1.83

(1) The available evaluation version of the IAR compiler did not support the memory model required for Dhrystone analysis.

2.3 Whetstone Analysis

The Whetstone type of 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 and floating point numbers. Figure 7 compares the code size for 8-bit and 16-bit microcontrollers with maximum optimization for code size implementation of the Whetstone analysis.

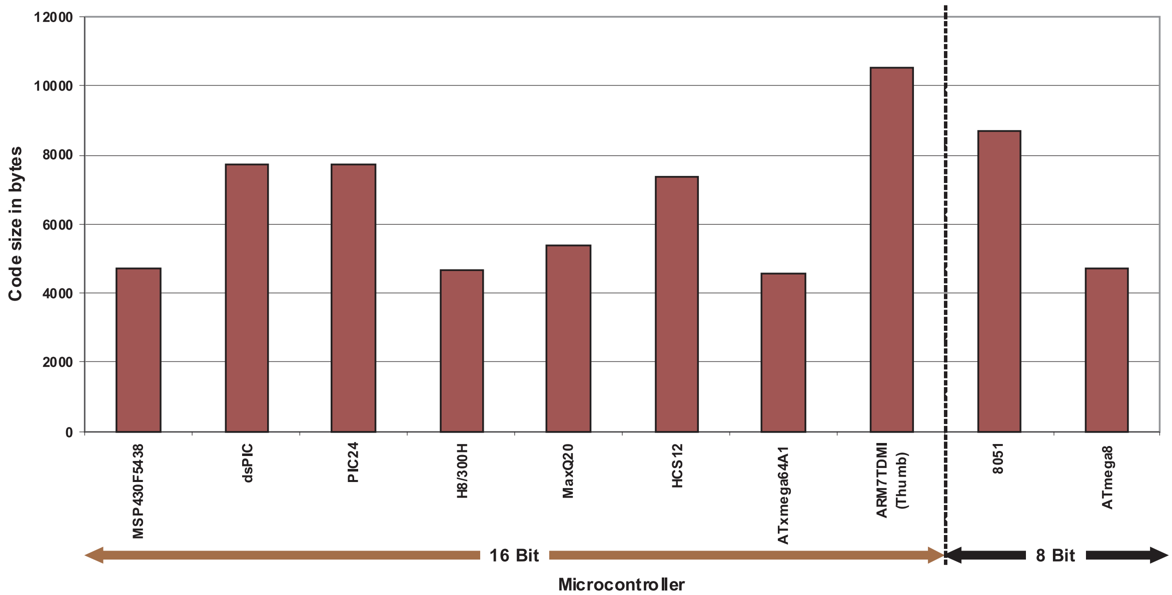

Figure 7. Code Size for Whetstone Analysis With 8-Bit and 16-Bit Microcontrollers

Figure 8 compares the cycle count for 8-bit and 16-bit microcontrollers with maximum optimization for speed (cycle count) in the implementation of the Whetstone analysis.

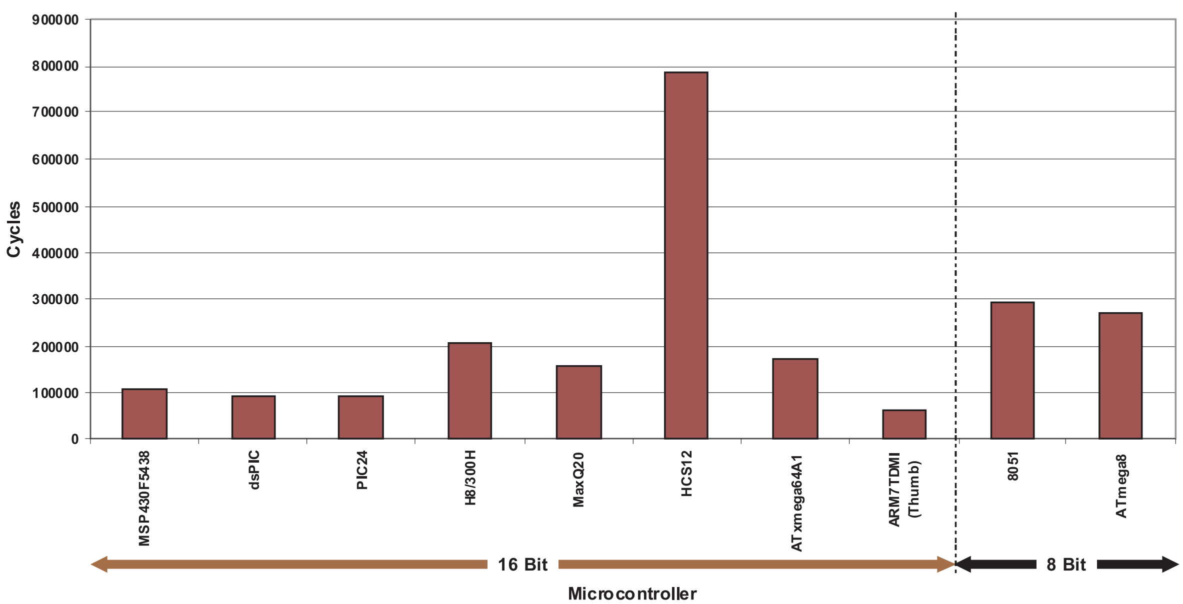

Figure 8. Cycle Count for Whetstone Analysis With 8-Bit and 16-Bit Microcontrollers

Table 4 shows the Whetstone analysis code size and the total instruction counts for each microcontroller normalized against the MSP430F5438.

Table 4. Normalized Results for Whetstone Analysis

Microcontroller		Total Code Size	Total Instruction Cycle Count
16-Bit	MSP430F5438	1.00	1.00
	dsPIC	1.63	0.88
	PIC24	1.63	0.88
	H8/300H	0.99	1.95
	MaxQ20	1.14	1.50
	HCS12	1.56	7.46
	ATxmega64A1	0.97	1.64
	ARM7TDMI (Thumb)	2.23	0.57
8-Bit	8051	1.85	2.76
	PIC18F242 (1)	—	—
	ATmega8	0.99	2.56

(1) The available evaluation version of the IAR compiler did not support the memory model required for Whetstone analysis.

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

Appendix A Background Information

A.1 CPU Clock vs Instruction Cycle Clock Considerations

MCU architectures have different associations between CPU clock frequency and the instruction cycle clock frequency. This is important in determining the actual instruction cycle clock frequency by dividing the CPU clock with the clock divider (see Table A-1). With the instruction cycle clock frequency, the total execution time of the system can be calculated. Note that higher clock frequencies generally also lead to higher power consumption due to increased CMOS logic switching losses.

Table A-1. CPU Clock Divider

Microcontroller		CPU Clock Divider
16-Bit	Texas Instruments MSP430	1
	Microchip dsPIC	2
	Microchip PIC24	2
	Renesas H8/300H	2
	MaxQ20	1
	Freescale HCS12	2
	Atmel ATxmega64A1	1
	ARM7TDMI (Thumb)	1
8-Bit	Generic 8051	1 to 12 [1]
	Microchip PIC18F242	4
	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 Systems 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. The library used in each compiler was CLIB. Table A-2 lists the "C" compiler version used to build the benchmarking applications for each microcontroller.

Table A-2. "C" Compiler Versions

Microcontroller		IAR C Compiler Version
16-Bit	Texas Instruments MSP430	4.11B
	Microchip dsPIC	1.4
	Microchip PIC24	1.4
	Renesas H8/300H	1.53I
	MaxQ20	1.13C
	Freescale HCS12	3.10A
	Atmel ATxmega64A1	5.11B
	ARM7TDMI (Thumb)	4.31A
8-Bit	Generic 8051	7.20C
	Microchip PIC18F242	3.10A
	Atmel ATmega8	4.12A

All applications were built with compiler optimization set to none and maximum, independently for code size and speed (cycle count). However, the graphs previously shown display only maximum optimization data. The optimized and unoptimized data are shown in the following tables.

Table A-3 and Table A-4 show the code size in bytes for each microcontroller for every math operation without optimization and with maximum optimization respectively.

Table A-3. Code Size in Bytes Without Optimization for Simple Math Operations

Microcontroller		8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
16-Bit	MSP430F5438	198	100	218	144	116	216	232	1114	164	2502
	dsPIC	312	476	504	308	620	504	480	2096	520	5820
	PIC24	304	476	496	300	620	496	472	2088	520	5772
	H8/300H	400	492	498	398	572	534	646	1176	554	5270
	MaxQ20	326	348	200	240	460	186	316	1200	480	3756
	HCS12	95	217	197	107	301	215	324	2082	270	3808
	ATxmega64A1	134	476	346	230	616	412	358	1132	592	4296
	ARM7TDMI (Thumb)	684	416	532	684	432	532	644	1868	476	6268
8-Bit	8051	266	499	305	478	693	519	1050	2346	707	6863
	PIC18F242	174	368	238	266	834	342	486	1322	732	4762
	ATmega8	152	394	378	210	532	424	352	1096	518	4056

Table A-4. Code Size in Bytes With Maximum Optimization for Simple Math Operations

Microcontroller		8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
16-Bit	MSP430F5438	178	86	198	126	90	198	222	1102	136	2336
	dsPIC	236	420	424	224	552	424	424	2020	464	5188
	PIC24	236	420	416	224	552	416	424	2020	464	5172
	H8/300H	344	412	444	352	482	478	574	1104	482	4672
	MaxQ20	230	252	192	204	328	184	288	1172	398	3248
	HCS12	83	188	162	76	262	174	323	2082	219	3569
	ATxmega64A1	118	398	338	174	490	350	300	1080	584	3832
	ARM7TDMI (Thumb)	636	392	452	636	396	452	620	1832	428	5844
8-Bit	8051	233	398	305	452	504	493	909	2190	536	6020
	PIC18F242	170	324	208	286	692	282	542	1400	676	4580
	ATmega8	134	354	350	198	434	382	342	1088	490	3772

Table A-5 and Table A-6 show the cycle count for each microcontroller for every math operation without optimization and with maximum optimization respectively.

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

Microcontroller		8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
16-Bit	MSP430F5438	239	2106	31	216	2875	30	552	769	3514	10332
	dsPIC	73	2811	63	72	3803	63	543	810	2644	10882
	PIC24	54	2055	44	53	2311	44	524	791	1888	7764
	H8/300H	240	10228	96	254	11252	102	520	1548	14018	38258
	MaxQ20	175	6196	42	201	9012	35	440	644	9624	26369
	HCS12	97	6858	51	108	8650	54	267	5508	8034	29627
	ATxmega64A1	128	4301	90	307	10289	119	765	1245	9344	26588
	ARM7TDMI (Thumb)	87	2122	51	102	2890	51	109	205	3424	9041
8-Bit	8051	212	14898	112	542	23868	314	3854	3339	19856	66995
	PIC18F242	141	7310	49	332	26533	87	1259	1049	32096	68856
	ATmega8	134	2523	39	288	9506	45	750	1663	8417	23365

Table A-6. Cycle Count With Maximum Optimization for Simple Math Operations

Microcontroller		8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
16-Bit	MSP430F5438	218	864	30	196	800	29	533	760	2637	6067
	dsPIC	60	1130	58	57	1866	58	535	797	1278	5839
	PIC24	43	550	39	40	550	39	518	780	687	3246
	H8/300H	152	4362	62	172	4746	66	388	1416	10468	21832
	MaxQ20	130	1140	38	183	1508	34	425	629	2214	6301
	HCS12	68	1559	46	60	2073	41	235	5470	2732	12284
	ATxmega64A1	105	1423	35	257	1929	41	716	1208	2820	8534
	ARM7TDMI (Thumb)	64	475	20	79	475	20	97	187	839	2256
8-Bit	8051	176	2590	112	526	4294	318	2622	2127	5880	18645
	PIC18F242	136	2193	49	339	6461	87	1284	1085	5283	16917
	ATmega8	110	984	38	266	1488	44	731	1654	2396	7711

Table A-7 shows the code size in bytes and cycle count for each microcontroller for math intensive operations without optimization and with maximum optimization selected individually for code size and speed (cycle count).

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

Microcontroller		FIR Filter				Dhrystone				Whetstone			
		Code Size		Cycles		Code Size		Cycles		Code Size		Cycles	
		Unopt	Opt	Unopt	Opt	Unopt	Opt	Unopt	Opt	Unopt	Opt	Unopt	Opt
16-Bit	MSP430F5438	988	980	111607	107146	1194	780	160672	98039	5776	4726	106451	105651
	dsPIC	2292	2256	120286	116718	2378	1678	86175	60651	8780	7716	93292	92965
	PIC24	2292	2256	117943	114375	2358	1662	79268	53944	8772	7708	93212	92885
	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
	HCS12	1945	1917	1045982	1035394	1244	900	208648	152212	8238	7370	788966	787635
	ATxmega64A1	1422	1362	250732	242008	2117	1453	269463	185295	7288	4594	114084	173354
	ARM7TDMI (Thumb)	1548	1528	37827	33114	1616	1000	83798	52352	11488	10532	61600	60444
8-Bit	8051	2116	2056	330640	321781	3075	1946	732532	488193	10613	8723	294309	291836
	PIC18F242 [(1)]	2058	2006	245704	182210	—	—	—	—	—	—	—	—
	ATmega8	1356	1358	365837	352894	2210	1474	240320	179834	8090	4694	274586	270991

(1) The available evaluation version of the IAR compiler did not support the memory model required for Dhrystone or Whetstone analysis.

A.3 Other MSP430 Families of Code Size and Cycle Count Simulations

Table A-8 shows how the MSP430 devices benchmarked in this application report are related to the rest of the devices in the MSP430 product line that are not explicitly listed.

Table A-8. Summary of Architectural Differences of All MSP430 Devices Not Explicitly Listed

Microcontroller	CPU	CPUX	CPUX (5xx)	MPY
MSP430F5438			•	•
MSP430FG4619		•		•
MSP430F2274	•			
MSP430F149	•			•

Table A-9 to Table A-13 show simulation results of the other MSP430 family of processors (‘F5438, ‘FG4619, ‘F2274, ‘F149). The MSP430F5438 and MSP430FG4619 differ slightly in architecture from the MSP430F2274 and MSP430F149 and integrate the MSP430X CPU (see user's guides for more information). 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.

Furthermore, in comparison with the 'F5438, 'FG4619, and 'F149, the 'F2274 does not have a built-in hardware multiplier. Hence, the ‘F2274 requires slightly larger code size and clock cycles. When using the IAR library, there are instances in which the floating-point library without the hardware multiplier for the MSP430 is a little more code efficient compared to the use of the hardware multiplier. However, the use of the hardware multiplier produced better clock cycle efficiency. The reason for this efficiency is because the IAR library without the hardware multiplier uses code loops that reduce the code size but increase the clock cycles.

Table A-9. MSP430 Families Code Size Without Optimization for Simple Math Operations

Microcontroller	8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
MSP430F5438	198	100	218	144	116	216	232	1114	164	2502
MSP430FG4619	198	100	218	144	116	216	246	1114	164	2516
MSP430F2274	234	108	182	180	136	180	252	1080	238	2590
MSP430F149	198	108	182	144	136	180	246	1114	198	2506

Table A-10. MSP430 Families Code Size With Optimization for Simple Math Operations

Microcontroller	8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
MSP430F5438	178	86	198	126	90	198	222	1102	136	2336
MSP430FG4619	178	86	198	126	90	198	236	1102	136	2350
MSP430F2274	214	88	180	162	112	178	238	1068	224	2464
MSP430F149	178	88	180	126	112	178	236	1102	182	2382

Table A-11. MSP430 Families Cycle Count Without Optimization for Simple Math Operations

Microcontroller	8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
MSP430F5438	239	2106	31	216	2875	30	552	769	3514	10332
MSP430FG4619	235	2106	28	213	2875	27	560	766	3514	10324
MSP430F2274	275	2490	28	286	3261	27	769	945	5411	13492
MSP430F149	246	2490	28	223	3261	27	580	790	3850	11495

Table A-12. MSP430 Families Cycle Count With Optimization for Simple Math Operations

Microcontroller	8-Bit Math	8-Bit Matrix	8-Bit Switch	16-Bit Math	16-Bit Matrix	16-Bit Switch	32-Bit Math	Floating-Point Math	Matrix Multi-plication	Total
MSP430F5438	218	864	30	196	800	29	533	760	2637	6067
MSP430FG4619	216	864	27	195	800	26	543	757	2637	6065
MSP430F2274	258	995	27	270	931	26	755	936	3406	7604
MSP430F149	225	995	27	203	931	26	561	781	3393	7142

Table A-13. MSP430 Families Code Size and Cycle Counts for FIR, Dhrystone, and Whetstone

Microcontroller	FIR Filter				Dhrystone				Whetstone			
	Code Size		Cycles		Code Size		Cycles		Code Size		Cycles	
	Unopt	Opt	Unopt	Opt	Unopt	Opt	Unopt	Opt	Unopt	Opt	Unopt	Opt
MSP430F5438	988	980	111607	107146	1194	780	160672	98039	5776	4726	106451	105651
MSP430FG4619	988	980	110635	106174	1194	780	158669	97436	5776	4726	105500	104702
MSP430F2274	962	954	128885	124105	1284	810	177547	109008	5828	4858	144901	143948
MSP430F149	996	988	115821	111041	1246	810	175447	109008	5888	4898	109698	108766

Appendix B Benchmarking Applications

B.1 Benchmarking Applications

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.

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.

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_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.

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.

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.

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.

matrix_multiplication.c
: Source file containing code that multiplies a 3x4 matrix by a 4x5 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 that performs the Dhrystone analysis.

whet.c
: Source file containing code that performs the Whetstone analysis.

B.2 Benchmarking Application Source Code

The following sections show the "C" source code files for the benchmarking applications used in this document.

B.2.1 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;
}
```

B.2.2 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;
}
```

B.2.3 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*/
  return (output);
}

void main(void)
{
  volatile UInt8 result;

  result = switch_case(0x10);
  return;
}
```

B.2.4 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;
}
```

B.2.5 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;
}
```

B.2.6 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;
        break;
 } /* end switch*/
 return (output);
}

void main(void)
{
  volatile UInt16 result;

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

B.2.7 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;
}
```

B.2.8 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;
}
```

B.2.9 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;
}
```

B.2.10 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 is
*                     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;

  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;
}
```

B.2.11 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[10][10];

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;

        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;
        }

        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);
                        Proc1(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();
}
```

B.2.12 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)
                         {
          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;
        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;
}
```