32-Bit Microcontroller Code Size Analysis

Draft 1.2.4. Joseph Yiu, Andrew Frame

Overview

Microcontroller application program code size can directly affect the cost and power consumption of products therefore it is almost always viewed as an important factor in the selection of a microcontroller for embedded projects. Since the release and availability of 32-bit processors such as the ARM Cortex-M3, more and more microcontroller users have discovered the benefits of switching to 32-bit products – lower power, greater energy efficiency, smaller code size and much better performance. Whilst most of the benefits of using 32-bit microcontrollers are widely known, the code size advantage of 32-bit microcontrollers is less obvious.

In this article we will explain why 32-bit microcontrollers can reduce application code size whilst still achieving high system performance and ease of use.

Typical myths of program size

Myth #1: 8-bit and 16-bit microcontrollers have smaller code size

There is a common misconception that switching from an 8-bit microcontroller to a 32-bit microcontroller will result in much bigger code size – why? Many people have the impression that 8-bit microcontrollers use 8-bit instructions and 32-bit microcontrollers use 32-bit instructions. This impression is often reinforced by slightly misleading marketing from the 8-bit and 16-bit microcontroller vendors.

In reality, many instructions in 8-bit microcontrollers are 16-bit, 24-bits or other sizes larger than 8-bit, for example, the PIC18 instruction sizes are 16-bit and, with the 8051 architecture, although some instructions are 1 byte long, many others are 2 or 3 bytes long.

So would code size be better moving to a 16-bit microcontroller? Not necessarily. Taking the MSP430 as an example, a single operand instruction can take 4 bytes (32 bits) and a double operand instruction can take 6 bytes (48 bits). In the worst case, an extended immediate/index instruction in MSP430X can take 8 bytes (64 bits).

So how about the code size for ARM Cortex microcontrollers? The ARM Cortex-M3 and Cortex-M0 processors are based on Thumb®-2 technology, which provides excellent code density. Thumb-2 microcontrollers have 16-bit instructions as well as 32-bit instructions, with the 32-bit instruction functionality a superset of the 16-bit version. In most cases a C compiler will use the 16-bit version of the instruction. The 32-bit version would only be used when the operation cannot be performed with a 16-bit instruction. As a result, most of the instructions in an ARM Cortex microcontroller program are 16-bits. That's even smaller than some of the instructions in 8-bit microcontrollers.

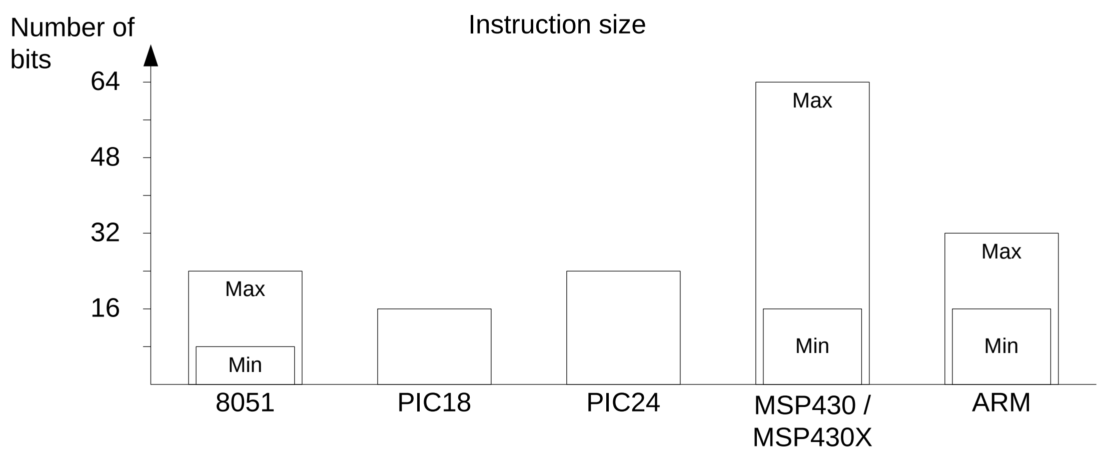

Figure 1: Size of a single instruction in various processors

Within a compiled program for Cortex-M processors, the number of 32-bit instructions can be only a small portion of the total instruction count. For example, the amount of 32-bit instructions in the Dhrystone program image is only 15.8% of the total instruction count (average instruction size is 18.53 bits) when compiled for the Cortex-M3. For the Cortex-M0 the ratio of 32-bit instructions is even lower at 5.4% (average instruction size 16.9 bits).

Myth #2: My application only processes 8-bit data and 16-bit data

Many embedded developers think that if their application only processes 8-bit data then there is no benefit in switching to a 32-bit microcontroller. However, looking into the output from the C compiler carefully, in most cases the humble "integer" data type is actually 16-bits. So when you have a *for* loop with an integer as loop index, comparing a value to an integer value, or using a C library function that uses an integer (e.g. *memcpy*()), you are actually using 16-bit or larger data. This can affect code size and performance in various ways:

- For each integer computation, an 8-bit processor will need multiple instructions to carry out the operations. This directly increases the code size and the clock cycle count.
- If the integer value has to be saved into memory, or if you need to load an immediate value from program ROM to this integer, it will take multiple instructions and multiple clock cycles.
- Since an integer can take up two 8-bit registers, more registers are required to hold the same number of integer variables. When there are an insufficient number of registers in the register bank to hold local variables, some have to be stored in memory. Thus an 8-bit microcontroller might result in more memory accesses which increases code size and reduces performance and power efficiency. The same issue applies to the processing of 32-bit data on 16-bit microcontrollers.

- Since more registers are required to hold an integer in an 8-bit microcontroller when passing variables to a function via the stack, or saving register contents during context switching or interrupt servicing, the number of stack operations required is more than that of 32-bit microcontrollers. This increases the program size, and can also affect interrupt latency because an Interrupt Service Routine (ISR) must make sure that all registers used are saved at ISR entry and restored at ISR exit. The same issue applies to the processing of 32-bit data on 16-bit microcontrollers.

There is even more bad news for 8-bit microcontroller users: memory address pointers take multiple bytes so data processing involving the use of pointers can therefore be extremely inefficient.

Myth #3: A 32-bit processor is not efficient at handling 8-bit and 16-bit data

Most 32-bit processors are actually very efficient at handling 8-bit and 16-bit data. Compact memory access instructions for signed and unsigned 8-bit, 16-bit and 32-bit data are all available. There are also a number of instructions specially included for data type conversions. Overall the handling of 8-bit and 16-bit data in 32-bit processors such as the ARM Cortex microcontrollers is just as easy and efficient as handling 32-bit data.

Myth #4: C libraries for ARM processors are too big

There are various C library options for ARM processors. For microcontroller applications, a number of compiler vendors have developed C libraries with a much smaller footprint. For example, the ARM development tools have a smaller version of the C library called MicroLib. These C libraries are especially designed for microcontrollers and allow application code size to be small and efficient.

Myth #5: Interrupt handling on ARM microcontrollers is more complex

On the ARM Cortex microcontrollers the interrupt service routines are just normal C subroutines. Vectored or nested interrupts are supported by the Nested Vectored Interrupt Controller (NVIC) with no need for software intervention. In fact the setup process and processing of an interrupt request is much simpler than 8-bit and 16-bit microcontrollers, as generally you only need to program the priority level of an interrupt and then enable it.

The interrupt vectors are stored in a vector table in the beginning of the memory, normally within the flash, without the need for any software programming steps. When an interrupt request takes place the processor automatically fetches the corresponding interrupt vector and starts to execute the ISR. Some of the registers are pushed to the stack by a hardware sequence and restored automatically when the interrupt handler exits. The other registers that are not covered by the hardware stacking sequence are pushed onto the stack by C compiler-generated code only if the register is used and modified within the ISR.

What about moving to 16-bit microcontrollers?

16-bit microcontrollers can be efficient in handling 16-bit integers and 8-bit data (e.g. strings) however the code size is still not as optimal as using 32-bit processors:

- Handling of 32-bit data: if the application requires handling of any long integer (32-bit) or floating point types then the efficiency of 16-bit processors is greatly reduced because multiple instructions are required for each processing operation, as well as data transfers between the processor and the memory.

- Register usage: When processing 32-bit data, 16-bit processors requires two registers to hold each 32-bit variable. This reduces the number of variables that can be held in the register bank, hence reducing processing speed as well as increasing stack operations and memory accesses.

- Memory addressing mode: Many 16-bit architectures provide only basic addressing modes similar to 8-bit architectures. As a result, the code density is poor when they are used in applications that require processing of complex data sets.

- 64K bytes limitation: Many 16-bit processors are limited to 64K bytes of addressable memory reducing the functionality of the application. Some 16-bit architectures have extensions to allow more than 64K bytes of memory to be accessed, however, these extensions have an instruction code and clock cycle overhead, for example, a memory pointer would be larger than 16-bits and might require multiple instructions and multiple registers to process it.

Instruction Set efficiency

When customers port their applications from 8-bit architecture to ARM Cortex microcontrollers, they very often find that the total code has dramatically decreased. For example, when Melfas (a leading company in capacitive sensing touch screen controllers) evaluated the Cortex-M0 processor, they found that the Cortex-M0 program size was less than half of that of the 8051 and, at the same time, delivered five times more performance at the same clock frequency. This, for example, could enable them to run the application at 1/5 clock speed of the equivalent 8051 product, reducing the power consumption, and lowering product cost at the same time due to a smaller program flash size requirements.

So how does ARM architecture provide such big advantages? The key factor is Thumb-2 technology which provides a highly efficient unified instruction set.

Powerful Addressing mode

The ARM Cortex microcontrollers support a number of addressing modes for memory transfer instructions. For example:

- Immediate offset (Address = Register value + offset)
- Register offset ((Address = Register value 1 + shifted(Register value 2))
- PC related (Address = Current PC value + offset)
- Stack pointer related (Address = SP + offset)
- Multiple register load and store, with optional automatic base address update
- PUSH/POP instructions with multiple registers

As a result of these various addressing modes, data transfer between registers and memory can be handled with fewer instructions. Since the PUSH and POP instructions support multiple registers, in most cases, saving and restoring of registers in a function call will only need one PUSH in the beginning of function and one POP at the end of the function. The POP can even be combined with the return instruction at the end of function to further reduce the instruction count.

Conditional branches

Almost all processors provide conditional branch instructions however ARM processors provide improved conditional branching by having separated branch conditions for signed and unsigned data operation results, and providing a good branch range.

For example, when comparing the conditional branches of the Cortex-M0 and MSP430, the Cortex-M0 has more branch conditions available, making it possible to generate more compact code no matter whether the data being process is signed or unsigned. The MSP430 conditional branches might require multiple instructions to get the same operations.

	Compare		Unsigned compare		Signed compare		Unsigned Overflow		Signed Overflow	
Cortex-M0 / M3	==	BEQ	>=	BCS	>=	BGE	Add overflow	BCS	Add overflow	BVS
	!=	BNE	>	BHI	>	BGT	No add overflow	BCC	No add overflow	BVC
	+ve	BPL	<=	BLS	<=	BLE	Sub overflow	BCC	Sub overflow	BVS
	-ve	BMI	<	BCC	<	BLT	No sub overflow	BCS	No sub overflow	BVC
MSP430	==	JEQ	>=	JC	>=	JGE	Add overflow	JC	Add overflow	-
	!=	JNE	>	-	>	-	No add overflow	JNC	No add overflow	-
	+ve	-	<=	-	<=	-	Sub overflow	JNC	Sub overflow	-
	-ve	JN	<	JNC	<	JL	No sub overflow	JC	No sub overflow	-

Generally the same situation applies to many 8-bit or 16-bit microcontrollers - when dealing with signed data, additional steps might also be required in the conditional branch.

In addition to the branch instructions available in the Cortex-M0, the Cortex-M3 processor also supports compare-and-branch instructions (CBZ and CBNZ). This further simplifies some of the conditional branch instruction sequence.

Conditional Execution

Another area that allows the ARM Cortex-M3 microcontrollers to have more compact code is the conditional execution feature. The Cortex-M3 supports an instruction called IT (IF-THEN). This instruction allows up to 4 subsequent instructions to be conditionally executed reducing the need for additional branches. For example,

```
if (xpos1 < xpos2) { x1 = xpos1;
              x2 = xpos2;
             } else {
              x1 = xpos2;
              x2 = xpos1;
```

This can be converted to the following assembly code (needs 12 bytes in the Cortex-M3):

```
    CMP   R0, R1
    ITTEE CC ; if unsigned “<”
    MOVCC  R2, R0
    MOVCC  R3, R1
    MOVCS  R3, R0
    MOVCS  R2, R1
```

Other architectures might need an additional branch (e.g. needs 14 bytes in MSP430):

```
    CMP.W  R14, R13
    JGE Label1 ; if unsigned “<”
    MOV.W  R11, R14
    MOV.W  R12, R13
    JMP Label2
Label1
    MOV.W  R11, R13
    MOV.W  R12, R14
Label2
```

This results in an extra two bytes for the MSP430 when compared to Cortex-M3.

Multiply and Divide

Both the Cortex-M0 and Cortex-M3 processors support single cycle multiply operations. The Cortex-M3 also has multiply and multiply-accumulate instructions for 32-bit or 64-bit results. These instructions greatly reduce the code size required when handling multiplication of large variables.

Most other 8-bit and 16-bit microcontrollers also have multiply instructions however the limitation of the register size often means that the multiplication requires multiple steps, if the result needs to be more than 8 or 16 bits.

The MSP430 does not have multiply instruction (MSP430 document slaa329, reference 1). To carry out multiplication either a memory mapped hardware multiplier is used, or the multiply operation has to be handled by software using add and shift. Even if a hardware multiplier is present the memory mapped nature of the multiplier results in the additional overhead of transferring data to and from the external hardware. In addition, using the multiplier within an interrupt handler could cause existing data in the multiplier to be lost. As a result, interrupts are usually disabled before a multiply operation and the interrupt is re-enabled after multiplication is completed. This adds additional software overhead and affects interrupt latency and determinism.

The Cortex-M3 processor also has unsigned and signed integer divide instructions. This reduces the code size required in applications that need to perform integer division because there is no need for the C library to include a function for handling divide operations.

Powerful instruction set

In additional to the standard data processing, memory access and program control instructions, the Cortex microcontrollers also support a number of other instructions to help data type conversion. The Cortex-M3 processor also supports a number of bit field operations reducing the software overhead in, for example, peripheral control and communication data processing.

Breaking the 64K byte memory barrier

As already mentioned, many 8-bit and 16-bit microcontrollers are limited to 64k bytes addressable memory. Due to the nature of 8-bit and 16-bit microcontroller architecture, the coding efficiency of these microcontrollers often decreases dramatically when the application exceeds the 64k byte memory barrier. In 8-bit and 16-bit microcontrollers (e.g. 8051, PIC24, C166) this is often handled by memory bank switching or memory segmentation with the switching code generated automatically by the C compilers. Every time a function or data in a different memory page is required bank switching code would be needed and hence further increases the program size.

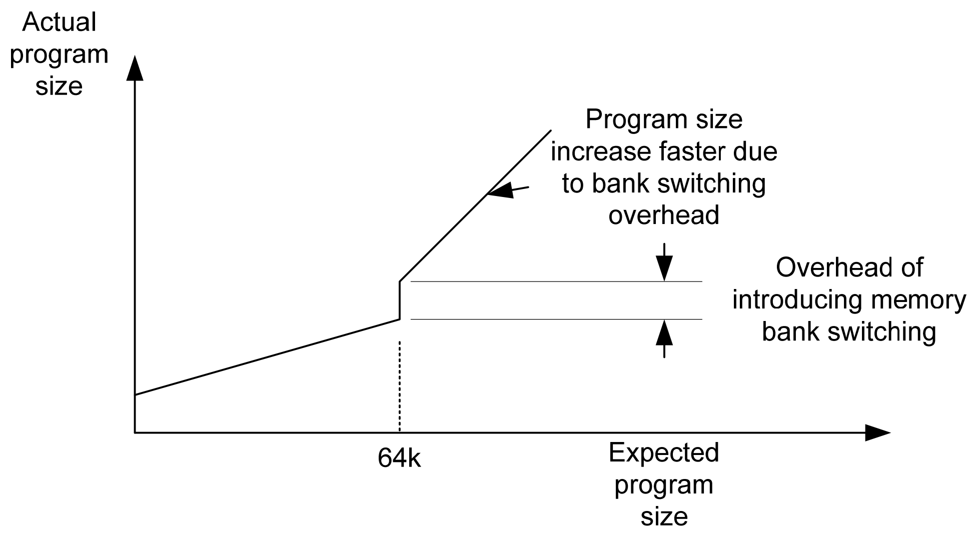

Figure 2: Increase code size overhead of memory bank switching or segmentation in 8-bit and 16-bit systems

The memory bank switching not only creates larger code but it also greatly reduces the performance of a system. This is especially the case if the data being processed is on different memory bank (e.g. copying a block of data from one page to another page can be very costly in terms of performance.) This is particularly inefficient for 8-bit microcontrollers like the 8051 because the MCS-51 architecture does not have proper support for such a memory bank switching feature. Therefore memory switching has to be carried out by saving and updating memory bank control like I/O port registers. In addition, the memory page switching code usually has to be carried out in a congested shared memory space with limited size. At the same time some of the memory pages might not be fully utilized and memory space is wasted.

For the 8-bit and 16-bit microcontrollers that support memory of over 64k this often comes at a price. The MSP430X design overcomes the 64K bytes memory barrier by increasing the Program Counter (PC) and register width to 20-bits. Despite no memory paging being involved, the sizes of some MSP430X instructions are considerably larger than the original MSP430. For example, when the large memory model is used, a double operand formatted instruction can take 8 bytes rather than 6 (a 33% increases):

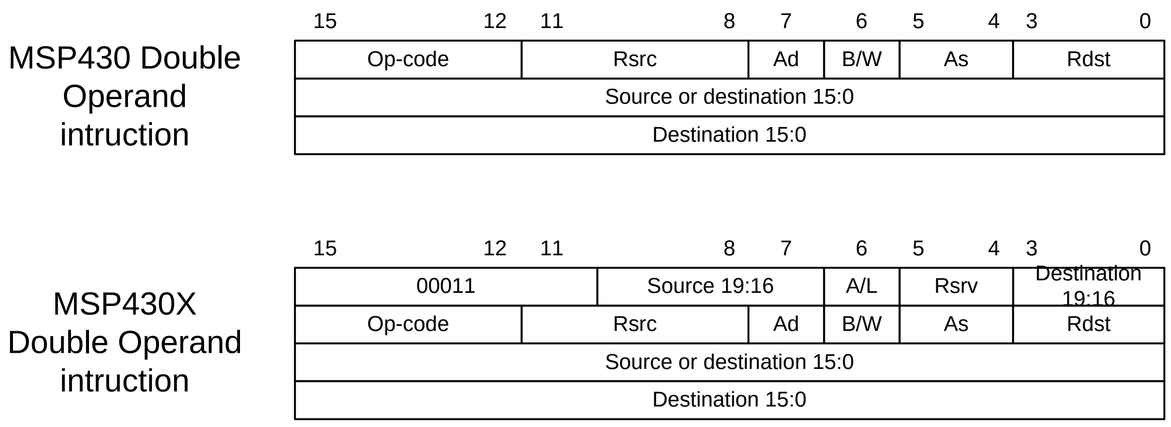

Figure 3: Support of larger memory system increases the size of some instructions in MSP430X

Apart from the size of the instruction itself, the use of the 20-bit addressing also increases the number of stack operations required. Since the memory is only 16-bit, the saving of a 20-bit address pointer will need two stack push operations, resulting in extra instructions and poor utilization of the stack memory.

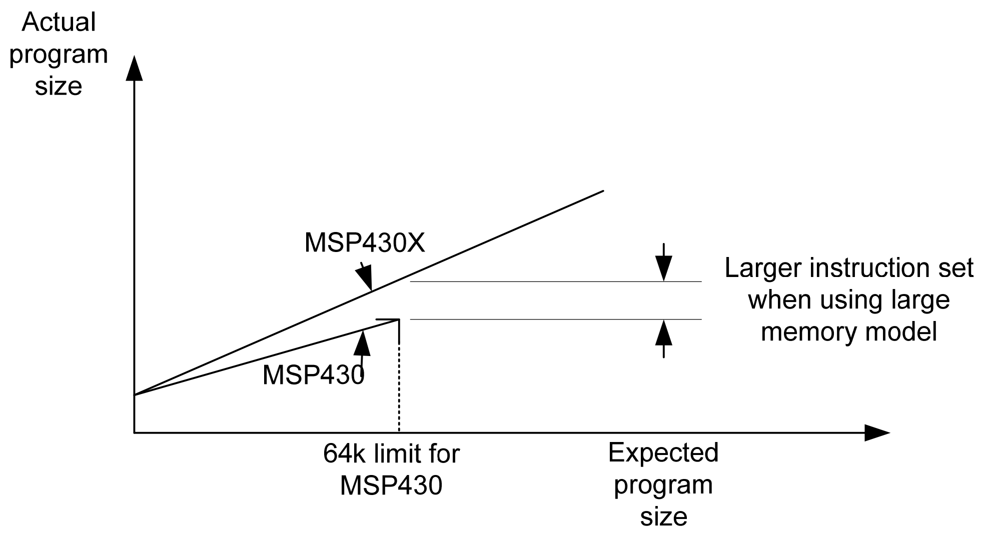

Figure 4: Use of large memory data model in MSP430X increases code size

As a result, an MSP430X application has a lower code density when the large memory model is used, which is required when the address range exceeds the 64k range.

In ARM Cortex microcontrollers, 32-bit linear addressing is used to provide 4GB of memory space for embedded applications. Therefore there is no paging overhead and the programming model is easy to use.

Examples

To demonstrate the code size compared to 8-bit and 16-bit processors, a number of test cases are compiled and illustrated here. The tests are based on "MSP430 Competitive Benchmark" document from Texas instruments (SLAA205C, reference 2). The results listed here show total program memory size in bytes.

MSP430 results:

The tests listed are compiled using IAR Embedded Workbench 4.20.1 with hardware multipler enabled, optimization level set to "High" with "Size" optimization. Unless specified, the "Small" data model is used and type double is 32-bit. The results are obtained at linker output report (CODE+CONST).

ARM Cortex processor results:

The tests listed are compiled using RealView Development Suite 4.0-SP2. Optimization level is 3 for size, minimal vector table, and MicroLIB is used. The results are obtained at linker output report (VECTORS + CODE).

Test	**Generic MSP430**	**MSP430F5438**	**MSP430F5438 large data model**	**Cortex-M3**
Math8bit	198	198	202	144
Math16bit	144	144	144	144
Math32bit	256	244	256	120
MathFloat	1122	1122	1162	600
Matrix2dim8bit	180	178	196	184
Matrix2dim16bit	268	246	290	256
Matrixmult	276	228	(linker error)	228
Switch8bit	200	218	218	160
Switch16bit	198	218	218	160
Firfilter(Note 1)	1202	1170	1222	716 (820 without modification)
Dhry	923	893	1079	900
Whet(Note 2)	6434	6308	6614	4384(8496 without modification)

Note 1: The constant data array in the Firfilter test is modified to use 16-bit data type on the Cortex-M processor (const unsigned short int INPUT[]).

Note 2: When certain math functions are used (*sin, cos, atan,sqrt, exp, log*) in the ARM C standard the double precision libraries are used by default. This can result in significantly larger program size unless adjustments are made. In order to achieve an equivalent comparison, the program code is edited so that single precision versions are used (*sinf, cosf, atanf, sqrtd, expf, logf*). Also, some of the constant definitions have been adjusted to single precision (e.g. 1.0 becomes 1.0F).

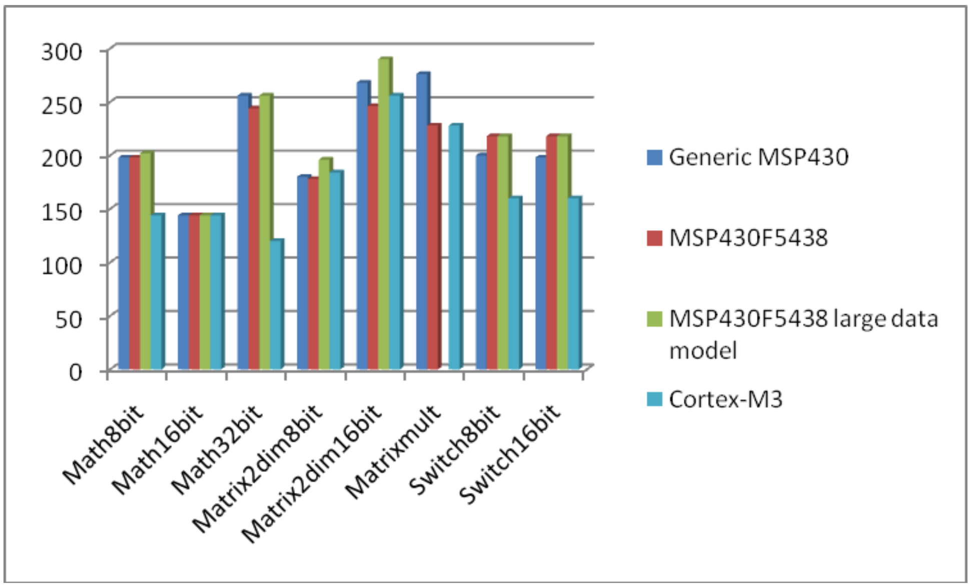

Figure 5 : Code size comparison for basic operations

The total size for simple tests (integer math, matrix and switch tests) are:

Summary for simple tests	Generic MSP430	MSP430F5438	Cortex-M3
Total size (bytes)	1720	1674	1396
Advantage (% smaller)	-	2.6%	18.8%

For applications using floating point, there us a signicant advantage for Cortex microcontrollers., whereas Dhrystone program size is closer.

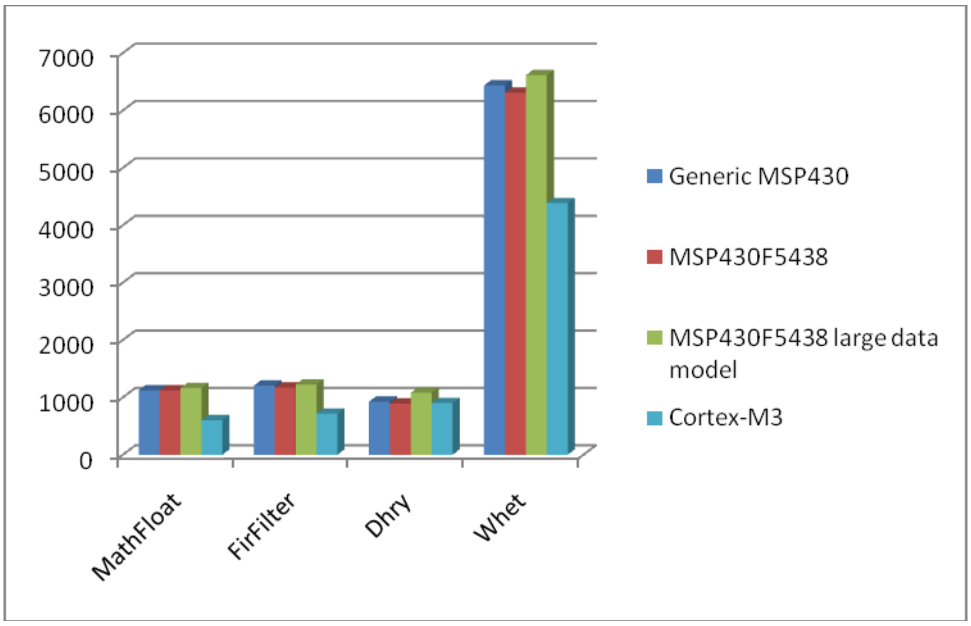

Figure 6: Code size comparison for floating point operations and benchmark suites

The total size for benchmark and floating point tests (Dhrystone, Whetstone, Firfilter and MathFloat) are:

Summary for simple tests	Generic MSP430	MSP430F5438	Cortex-M3
Total size (bytes)	9681	9493	6600
Advantage (% smaller)	-	1.9%	31.8%

Observations:

1. From the results, we can see that the Cortex microcontrollers have better code density compared to MSP430 in most cases. The remaining tests show similar code density when compared to MSP430.
2. One of the tests (firfilter) uses an integer data type for a constant array. Since an integer is 32-bit in the ARM processor and is 16-bit on MSP 430, the program has been modified to allow a direct comparison.
3. When the large data memory model is used with MSP430, the code size increases by up to 20% (dhrystone).
4. We are unable to reproduce all of the claimed results in the Texas Instruments document. This may be because the storage of constant data in ROM might have been omitted from their code size calculations.

Additional investigation on floating point

When analysing the results of the whetstone benchmark it became apparent that the MSP430 C compiler only generated single precision floating operations, while the ARM C compiler generated double precision operations for some of the math functions used.

After changing the code to use only single precision floating points the code size reduced dramatically and resulted in much smaller code size than the MSP430 code size.

The IAR MSP430 compiler has an option to define floating point : “Size of type double” which is by default set to 32-bit (single precision). If it is set to 64-bit (as in ARM C compiler), the code size increased significantly.

Program size	Generic MSP430	MSP430430F5438
Type Double is 32-bit	6434	6308
Type Double is 64-bit	11510	11798

These results match those seen for the ARM Cortex-M3 processor.

Program size	Cortex-M3
Whetstone modified to use single precision only	4384
Out of box compile for whetstone (use double precision for math functions)	8496

The option of setting type double to 32-bit is quite sensible for small microcontroller applications where the C code might only need to process source data generated from 12-bit/14-bit ADC. Benchmarking using different default types can make a very big difference and not show accurate comparative results.

Recommendations on how to get the smallest code size with Cortex-M microcontrollers

Use MicroLib

In the ARM development tools there is an option to use the area optimized MicroLIB rather than the standard C libraries. The MicroLIB is suitable for most embedded applications and has a much smaller code size when compared to the standard C library.

Ensure the use of area optimizations

The performance of Cortex-M microcontrollers is much higher than that of 16-bit and 8-bit microcontrollers so when porting applications from these microcontrollers you can generally select the highest area optimization rather than selecting optimizations for speed. The resulting performance will still be much higher than that of a 16-bit or 8-bit system running at the same clock frequency.

Use the right data type

When porting applications from 8-bit or 16-bit microcontrollers, you might need to modify the data type for constant arrays to achieve the most optimal program size. For example, an integer is normally 16-bit in 8-bit and 16-bit microcontrollers, while in ARM microcontrollers integers are 32-bit.

Type	Number of bits in 8051	Number of bits in MSP430	Number of bits in ARM
"char", "unsigned char"	8	8	8
"enum"	8/16	16	8/16/32 (smallest is chosen)
"short", "unsigned short"	16	16	16
"int", "unsigned int"	16	16	32
"long", "unsigned long"	32	32	32
float	32	32	32
double	32	32	64

When porting a constant array of integers from an 8-bit or 16-bit architecture, you should modify the data type from "int" to "short int" to make sure the constant array remains the same size. For example,

const int mydata = { 1234, 5678, …};

This should be changed to:

const ***short*** int mydata = { 1234, 5678, …};

For an array of integer variables (non-constant data), changing from an integer to a short integer might also prevent an increase in memory usage during software porting. Most other data (e.g. variables) does not require modification.

Floating point functions

Some floating point functions are defined as single precision in 8-bit or 16-bit microcontrollers and are by default defined as double precision in ARM microcontrollers, as we have found out with the whetstone test analysis. When porting application code from 8-bit or 16-bit microcontrollers to an ARM microcontroller, you might have to adjust math functions to single precision versions and modify constant definitions to ensure that the program behaves in the same way. For example, in the whetstone program code, a section of code uses some math functions that are double precision in ARM compilers:

```
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));
```

If we want to use single precision only, the program code has to be changed to

```
X=T*atanf(T2*sinf(X)*cosf(X)/(cosf(X+Y)+cosf(X-Y)-1.0F));

Y=T*atanf(T2*sinf(Y)*cosf(Y)/(cosf(X+Y)+cosf(X-Y)-1.0F));
```

Other constant definitions such as:

```
/* Module 7: Procedure calls */
X = 1.0;
Y = 1.0;
Z = 1.0;
```

should to be changed to the following for single precision representation:

```
/* Module 7: Procedure calls */
X = 1.0F;
Y = 1.0F;
Z = 1.0F;
```

Define peripherals as data structure

You can also reduce program size by defining registers in peripherals as a data structure. For example, instead of representing the SysTick timer registers as

```
#define SYSTICK_CTRL   (*((volatile unsigned long *)(0xE000E010)))
#define SYSTICK_LOAD   (*((volatile unsigned long *)(0xE000E014)))
#define SYSTICK_VAL    (*((volatile unsigned long *)(0xE000E018)))
#define SYSTICK_CALIB  (*((volatile unsigned long *)(0xE000E01C)))
```

you can define the SysTick registers as:

```
typedef struct
{
  volatile unsigned int CTRL;
  volatile unsigned int LOAD;
  volatile unsigned int VAL;
  unsigned int CALIB;
} SysTick_Type;

#define SysTick ((SysTick_Type *) 0xE000E010)
```

By doing this, you only need one address constant to be stored in the program ROM. The register accesses will be using this address constant with different address offsets for different registers. If a sequence of hardware register accesses is required for a peripheral, using a data structure can reduce code size as well as improve performance. Most 8-bit microcontrollers do not have the same addressing mode feature which can result in a much larger code size for the same task.

Conclusions

32-bit processors provide equal or more often better code size than 8-bit and 16-bit architectures whilst at the same time delivering much better performance.

For users of 8-bit microcontrollers, moving to a 16-bit architecture can solve some of the inherent problems with 8-bit architectures, however, the overall benefits of migrating from 8-bit to 16-bit is much less than that achieved by migrating to the 32-bit Cortex processors.

As the power consumption and cost of 32-bit microcontrollers has reduced dramatically over last few years, 32-bit processors have become the best choice for many embedded projects.

Reference

The following articles on MSP430 are referenced:

	Reference
1	MSP430 Competitive Benchmarking *http:// focus.ti.com/lit/an/slaa205c/slaa205c.pdf*
2	Efficient Multiplication and Division Using MSP430 *http://focus.ti.com/lit/an/slaa329/slaa329.pdf*