

A Discussion on Code Density for the LPC1100, a Cortex-M0™ based Processor

By Rob Cosaro and Zeljko Stefanovic, NXP

The LPC1100 is a 32-bit Cortex-M0 based microcontroller that competes in the 8- and 16-bit space. There is a misconception that the memory requirements will be 4 times as large for a 32-bit machine that an 8-bit machine. This article discusses the various aspects of code size, comparing M0 code to various 8-bit processors. It shows that M0 code is significantly smaller in some cases and in others it is about the same size.

LPC1100 Code Density In early 2009 NXP introduced the first M0 based microcontroller family to the market called the LPC1100. The target markets for this family are customers currently using 8- and 16-bit architectures. One frequently asked question about the LPC1100 is how much code space will my application need. A simplistic approach is to assume that a 32-bit processor needs 4 bytes for the instruction, whereas an 8-bit processor takes 1 byte for an instruction, the code size for a 32-bit processor should be 4 times as large. This approach is not valid. Based on customer's experiences where 8-bit code was ported to the LPC1100, applications can be anywhere from a quarter to a half smaller than on an 8-bit processor. However, these kinds of gains are dependent on the nature of the code. In some cases, applications may not see a decrease, but may be a similar size to its 8-bit equivalent. This article will discuss cases where code will be smaller and similar in size, and explain where the code density comes from.

Instruction Set Architecture The Cortex-M0 is a RISC architecture whose instruction set is based on the ARM thumb concept. Thumb instructions are 16-bits in length, but still operated on 32-bit registers and data paths. Figure 1 shows the complete instruction set for the Cortex-M0. There are some 32-bit instructions but most of these are not commonly used in application code except for the BL instruction. (Branch with link). So for the LPC11000 family the instruction size is mostly 16 bits. But how does this compare to 8-bit instruction set architectures? An 8-bit architecture does not mean that the instruction length is 8-bits. Many instructions in 8-bit microcontrollers are 16-bit, 24-bit or other sizes larger than 8-bit. For example, the PIC18 and PIC16 instruction sizes are 18-bit and 16-bit respectively,

Thumb

User assembly code, compiler generated

ADC	ADD	ADR	AND	ASR	B
BIC	BL		BX	CMN	CMP
EOR	LDM	LDR	LDRB	LDRH	LDRSB
LDRSH	LSL	LSR	MOV	MUL	MVN
ORR	POP	PUSH	ROR	RSB	SBC
STM	STR	STRB	STRH	SUB	SVC
TST	BKPT	BLX	CPS	REV	REV16
REVSH	SXTB	SXTH	UXTB	UXTH	

Thumb-2

System, OS

NOP	
SEV	WFE
WFI	YIELD
DMB	
DSB	
ISB	
MRS	
MSR	

Figure 1: Cortex-M0 instruction set

and for the 8051 architecture some instructions are 1 byte long, but many are 2 or 3 bytes long. Likewise, compare the Cortex-M0 to that of a 16-bit architecture like the MSP430 where the instruction length can vary from 4 to 8 bytes, one cannot draw the conclusion that 16-bit architecture will produce half the code size. Figure 2 shows graphically how instruction length varies with the processor architecture. Since the instruction set for the Cortex-M0 is mostly 16 bits it's clear that the code size for the LPC1100 has a good chance of being smaller than most 8-bit processors.

Code size is not just about the instruction set width. It has much to do with the width of the peripheral data the processor has to deal with. These days it's not uncommon for microcontrollers to have integrated 10- and 12-bit ADC's. It takes many more instructions for an 8-bit micro to manipulate these data widths as opposed to a 32-bit machine. Figure 3 on the next page shows an example of a 16-bit multiplication. This could be part of a digital filter operating on incoming ADC data. For the 8051 it takes 48 bytes of code, for a 16-bit processor it takes 8 bytes of code and for the LPC1100 with a hardware multiply this takes only 2 bytes of code. This may seem like a narrow example but having larger bit widths to process data is an important part of saving code space. A more general example comes from the fact that when

that operate directly on the port registers. However, by understanding how address indexing works, the code overhead to toggle ports can be minimized.

The NXP LPC1113 was used as a test vehicle to show how much code it takes to write to a port. The LPC1113 ports are set up to be accessible as a bit, byte, or a half-word.

The fact that they can be written to in a half word does save code, and more than 8-bits can be written to a port at the same time. The tests were performed to show how much code is required to change a single bit (either to set it high or low) and then to set a bit pattern. The code size results are shown in Table 2.

LPC1113								
Armcc.exe V4.0.0.524 optimization								
	size				time			
	0	1	2	3	0	1	2	3
iterations	200	200	200	200	200	200	200	300
time	19.10	15.87	15.68	15.68	19.10	14.35	13.05	17.73
CM score	0.87	1.05	1.06	1.06	0.87	1.16	1.28	1.41
Code	9120	7348	7052	7072	9120	7734	8132	9648
RO data	748	640	592	592	748	586	628	588
RW data	148	136	136	136	148	136	136	136
ZI data	2612	2608	2608	2608	2612	2608	2608	2608
Code+RO	9868	7988	7644	7664	9868	8320	8760	10236
RW+ZI	2760	2744	2744	2744	2760	2744	2744	2744
ROM (Code+RO+ZI)	10016	8124	7780	7800	10016	8456	8896	10372

Table 1: CoreMark results for the LPC1113

Four different types of coding styles were used to access the port of the LPC1113.

Access via the DATA register

```
LPC_GPIO0->DATA &= ~(1<<clr_pin_position);
LPC_GPIO0->DATA |= (1<<set_pin_position);
LPC_GPIO0->DATA = (LPC_GPIO0->DATA & ~controlled_pins) | pin_pattern;
```

Access via the DATA register and reserved CPU register

```
register volatile unsigned long int *ptr_port_data;
ptr_port_data = (volatile unsigned long int *) &(LPC_GPIO1->DATA);
*ptr_port_data &= ~(1<<clr_pin_position);
*ptr_port_data |= (1<<set_pin_position);
*ptr_port_data = (*ptr_port_data & ~controlled_pins) | pin_pattern;
```

Access via a macro

```
#define RESET_PORT2_PINx (*(volatile unsigned long int *)
    (LPC_GPIO2_BASE + ((1<<x)<<2))) = 0)
#define SET_PORT2_PINx (*(volatile unsigned long int *)
    (LPC_GPIO2_BASE + ((1<<x)<<2))) = 1<<x)
#define PATTERN_PORT2 (*(volatile unsigned long int *)
    (LPC_GPIO2_BASE + (controlled_pins<<2))) = bit_pattern)
```

Access via a CPU register

```
register volatile unsigned long int *ptr_port_pattern;
```

```
ptr_port_pattern = (volatile unsigned long int *) (LPC_GPIO3_BASE +
    (controlled_pins<<2));
*ptr_port_pattern = 0;
*ptr_port_pattern = 1<<set_bit_position;
*ptr_port_pattern = pin_pattern;
```

The smallest code size for setting a pin is accomplished by reserving a CPU register for the address. In this case it takes a total of 4 bytes. Below is the disassembled code that is driving a port pin low and then setting it high.

```
*ptr_port_pin = 0; //P3_02 = 0
0x000004B8 2000 MOVS r0,#0x00
0x000004BA 6028 STR r0,[r5,#0x00]

*ptr_port_pin = 1<<2; //P3_02 = 1; access via a reserved CPU register
```

```
0x000004B8 2004 MOVS r0,#0x04
0x000004BA 6028 STR r0,[r5,#0x00]
```

These code snippets show there is no difference between driving a pin low or high, as long as the bit position is 0 to 7. This is because the position is encoded as the power of n and exist in the 256 bit opcode for MOVS. As from the above example the constant went from x00 to x04. If the constant exceeds 8-bits as in the case when the bit position is in the 8:11 place, then the compiler must insert another instruction as shown below.

```
*ptr_port_pin = 1<<10; //P3_10 = 1; access via a reserved CPU register
```

```
0x000004B4 2001 MOVS r0,#0x01
0x000004B6 0280 LSLS r0,r0,#10
0x000004B8 6020 STR r0,[r4,#0x00]
```

The first step the compiler takes is to load 1 into a register and then shift it where the bit should go. The additional instruction takes an extra 2 bytes. There is also a difference in driving patterns to 0:7 or 0:11. It takes 4 bytes when driving ports 0:7 and 8 bytes when driving a pattern to 0:11. This example shows the code required to set a pattern to 0:11, is 4 bytes, but there is a constant for the bit pattern that is stored in flash. In the next example the @0x00000530 is where the address of the constant is located. The total code space required is 4 bytes of code and 4 bytes for the constant.

```
*ptr_port_pattern = 0x329;
0x000004DA 4815 LDR r0,[pc,#84] ; @0x00000530
0x000004DC 6028 STR r0,[r5,#0x00]
```

Table 2 on the next page shows examples for other ways to write to ports. The worst case is 28 bytes to write a pattern using data registers and writes to a 12-bit port. Below is this example.

```
LPC_GPIO0->DATA = (LPC_GPIO0->DATA & ~0xE13) | 0x611;
0x000004B8 4816 LDR r0,[pc,#88] ; @0x00000514
0x000004BA 6BC0 LDR r0,[r0,#0x3C]
0x000004BC 4916 LDR r1,[pc,#88] ; @0x00000518
0x000004BE 4008 ANDS r0,r0,r1
```

```

0x000004C0 4916 LDR    r1,[pc,#88] ; @0x0000051C
0x000004C2 1840 ADDS  r0,r0,r1
0x000004C4 4913 LDR    r1,[pc,#76] ; @0x00000514
0x000004C6 63C8 STR    r0,[r1,#0x3C]
    
```

```

0x00804    E5A0    MOV    A,P2(0xA0)
0x00806    54E8    ANL   A,#IEN1(0xE8)
0x00808    4414    ORL   A,#0x14
0x0080A    F5A0    MOV   P2(0xA0),A
    
```

This is the typical read-modify-write approach. The data register is being masked with the control bit patterns and or'ed with the desired content. It may seem like a reasonable approach but leads to numerous constants that need to be stored in flash and makes for code that is larger than necessary.

For single bit writes the 8051 is good, but equivalent patterns to ports the M0 core wins.

Summary Code density for the LPC1100 is not 4 times as big as an 8-bit processor. It may seem counter intuitive, but in

M0 port pin control approach vs code size [bytes]	drive a pin low						drive a pin high						drive a pattern					
	0:7			8:11			0:7			8:11			0:7			0:11		
	code	const	total	code	const	total	code	const	total	code	const	total	code	const	total	code	const	total
access via the DATA register	12	4	16	14	4	18	12	4	16	14	4	18	14	4	18	16	12	28
acc via the DATA and reserved CPU reg	8	0	8	10	0	10	8	0	8	10	0	10	12	0	12	12	8	20
access via the macro	6	4	10	6	4	10	6	4	10	8	4	12	6	4	10	6	8	14
access via a reserved CPU register	4	0	4	4	0	4	4	0	4	6	0	6	4	0	4	4	4	8

Table 2: Code size summary for port writing

How does this compare with an 8051 that has the “bit set” and “bit clear” instruction? It takes 2 bytes to set any port pin high or low and 8 bytes to set any pattern to an 8-bit port, and it can't go beyond 8-bits for one write. It would take another 8 bytes to accomplish that. Below is the disassembled code for an 8051 for clearing and setting bits and writing a pattern.

```

14:                P2_D0 = 0;
0x00800    C2A0    CLR  P2_D0(0xA0.0)
15:                P2_D1 = 1;
0x00802    D2A1    SETB P2_D1(0xA0.1)
16:                P2 = (P2 & ~0x17) | 0x14;
    
```

most cases the code will be smaller. There are cases where applications are doing mostly port manipulations that could lead to slightly larger but with some understanding of the architecture and the compiler there will not be any significant differences. There are even advantages over 8-bit processors in this case as well, since in one single write up to 32 bits can be changed at the same time. The Cortex-M0 not only offers significant savings in code size compared to 8- and 16-bit architectures, it offers a dramatic performance advantage.

END