Proof that C can match or beat assembly

Byte Craft Limited

Introduction

We've often boasted that C can equal, and usually beat, macro-assembly language programming. When assembly programmers can convince themselves to let the compiler do the work, a C compiler can match or beat their best assembly efforts. After all, what is a compiler but the distilled wisdom of years of programming experience?

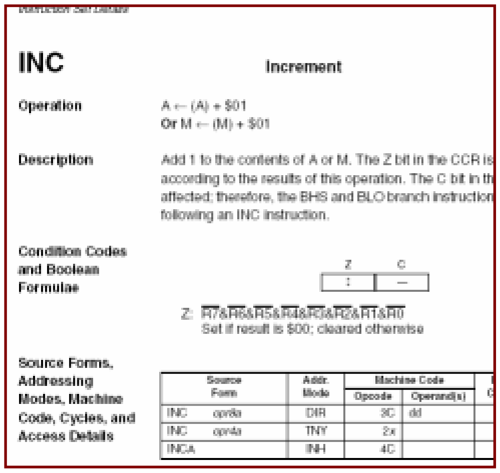
Now we have some concrete proof to show this. This document demonstrates a way to prove that a compiler can match handwritten assembly, and even beat it without trying.

In the process of verifying our latest product, C6808 with support for Freescale's RS08 core, we went through an exercise that tests our compiler's ability to match assembly. The exercise is simple: we wrote C code that implemented the semantics of every assembly instruction exactly, and then we compiled the program. For each C operation, we expected to see its equivalent machine language code, but the results showed us something more.

The process

Creating the test is simple:

1. Declare a few choice C variables. There's more on this below, but the key is to have one or two variables from each memory space within the microcontroller.
2. Look at the algebraic description of the operation of each instruction, and simply perform that operation in C. Compose short C statements that implement each assembly instruction. Do the same for each addressing mode.
3. For each instruction, where the instruction description changes the program counter, use **`goto`**. Where the instruction changes the accumulator, assign a new value to `reg_ac` or another variable declared as **`#pragma rega`**.



Instruction Set Details

INC Increment

Operation A ← (A) + $01
Or M ← (M) + $01

Description Add 1 to the contents of A or M. The Z bit in the CCR is according to the results of this operation. The C bit in th affected; therefore, the BHS and BLO branch instruction following an INC instruction.

Condition Codes and Boolean Formulae

Z	C
↕	—

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$
Set if result is $00; cleared otherwise

Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Addr. Mode	Machine Code	
		Opcode	Operand(s)
INC opr8a	DIR	3C	dd
INC opr4a	TNY	2x	
INCA	INH	4C	

Byte Craft Limited

A2-490 Dutton Drive
Waterloo, Ontario
Canada N2L 6H7
Phone: 519 888 6911
Fax: 519 746 6751
Email: info@bytecraft.com
http://www.bytecraft.com

4. Compile the resulting source file and look at the generated code in the listing file. For each C translation, the correct opcode should appear.

Here's an example:

```
0100 4C         INCA                    AC++;
0101 2F         INC   X                 X++;
0102 26         INC   $06               tiny++;
0103 3C 15      INC   $15               small++;
0105 3C 34      INC   $34               page0++;

0107 4A         DECA                    AC--;
0108 5F         DEC   X                 X--;
0109 56         DEC   $06               tiny--;
010A 3A 15      DEC   $15               small--;
010C 3A 34      DEC   $34               page0--;
```

A cursory check of the code shows the compiler is matching C correctly. Here's a little more involved test:

```
0160 3B 34 9D   DBNZ  $34,$0100              if (--page0 !=0) goto rel;  // DBNZ page0,rel
0163 3B 34 01   DBNZ  $34,$0167              if (--page0 == 0) NOP();
0166 AC         NOP
0167 31 34 96   CBEQ  $34,$0100              if (AC == page0) goto rel;  // CBEQ page0,rel
016A 31 34 01   CBEQ  $34,$016E              if (AC != page0) NOP();
016D AC         NOP
```

The C implements two common uses of **`DBNZ`** and **`CBEQ`**, and the compiler optimizes accordingly, jumping directly to a target or simply skipping an implied branch.

An entire compiled file of these tests runs about 400 bytes, for a regular instruction set with 1 byte opcodes, and with 1, 2, and (rarely) 3 byte instructions.

The variables are declared in locations that correspond to the different address modes. `AC` and `X` are declared as `registera` and `registerx`, built-in types that correspond to the accumulator and 0x0F (the index register), respectively. Other variables are declared in named address spaces specified in the device header file, or simply assigned to known locations using the @ notation:

```
char small @ 0x15;
```

The Results

We can see the compiler's intelligence appear when it chooses the right instructions, of course. For instance, at the beginning of the test suite, the compiler loads `PAGE` once for several extended address instructions. Unless the test uses variables outside the 0x200 64-byte extended page (`PAGE` == 0x08), this only needs to happen once.

But we can also see the compiler shine when it chooses unintuitive instructions that, on later analysis, are completely correct. Here's an example:

```
010E B0 34      SUB   $34               AC = AC - page0;
0110 B1 34      CMP   $34               AC - page0;
0112 B1 34      CMP   $34               (void)AC - page0;
```

These three C statements were intended to test the **`SUB`** instruction. In the second and third cases, the compiler has caught a subtle distinction. When the result of an expression is to be discarded (implicitly or explicitly), the registers involved must not be mutated (`AC` doesn't change value, after all). The compiler chooses an optimization to determine the implied result, and the compiler can avoid reloading `AC`.

Frequently-Asked Questions

But doesn't this prove C is no more than elaborate assembly?

The first aim of this exercise is to match assembly, a kind of identity transformation. To do that, we followed the assembly pretty closely in C. True, we did use some specially-located variables that you might not otherwise use on a regular basis.

But this exercise does go beyond simple freeform assembly. It invokes some optimizations regarding simplification of operations, as seen above. It demonstrates that all optimizations are always considered, something that a human assembly programmer can find difficult to do.

What doesn't this example test?

This exercise doesn't test optimizations related to higher math functions or program flow (dead code removal and so on).

Also, it doesn't invoke higher memory allocation functions. You can move declarations for LOCAL memory, for instance, between different areas within RS08's RAM. If you'll be using lots of local variables and using them intensively, consider moving LOCAL to the *Tiny* address space:

```
#pragma memory LOCAL [14] @ 0x0000;
```

The frequent accesses to function-local variables will enjoy reduced code size.

Why do you declare variables where no RAM exists?

Some of the variables are declared in the frequently-used registers area (0x10-0x1F). This tests the *Small* addressing mode selection, whether addressing a byte of RAM or a register. To the compiler, the accesses are much the same. The only difference is that port registers are deemed **`volatile`**.

Conclusions

Compilers are very complex software systems. This means that, within their rules, they can sometimes offer results we didn't think to expect. We were surprised to see the optimizer use strategies we knew were possible but that were in fact novel combinations in the compiler's repertoire.

Ultimately, this test was a shakedown run for the compiler: once around the instruction set, with a detailed inspection afterward. Since we can match each basic assembly operation in C, compiler performance will only improve from that baseline.

Document Revision history:

Version	Date	Initials	Description
1	9 June 2006	KZ	Initial version.
1.1	26 June 2006	KZ	Revision.
1.2	10 July 2006	KZ	Revision; public release.
1.3	4 October 2006	KZ	Formatting, corrections.