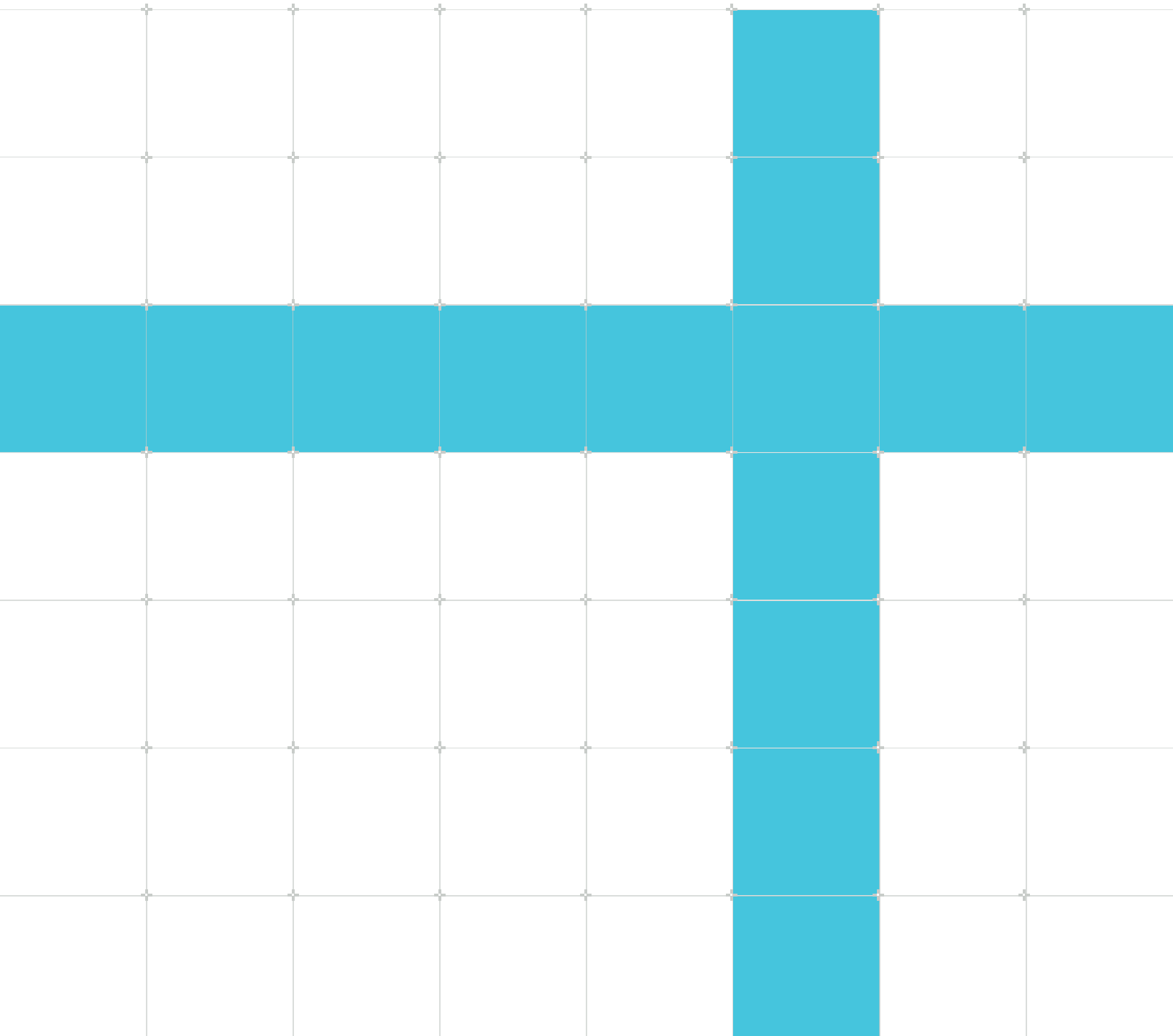
Using inline assembly to improve code efficiency

Version 1.0

Issue 01
102694_0100_01_en



Using inline assembly to improve code efficiency

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	11 May 2017	Non-Confidential	First release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

This tutorial for the Arm Compiler shows you how to use inline assembler to access features on a target processor not available with C and C++ alone.

The compiler provides an inline assembler that enables you to write optimized assembly language routines, and to access features of the target processor not available from C or C++.

This tutorial assumes you have installed and licensed Arm DS-5 Development Studio. For more information, see Getting Started with Arm DS-5 Development Studio.

Inline assembly code using the `__asm` keyword

The `__asm` keyword can incorporate inline Arm syntax assembly code into a function. The general form of an `__asm` inline assembly statement is:

```
__asm
        {
                ...
                instruction
                ...
        }
```

The following simple example uses inline assembly code to add two integers together:

```
#include  <stdio.h>

        int add_inline(int r5, int r6)
        {
          int res = 0;
          __asm
          {
                ADD res, r5, r6
          }
          return res;
        }

        int main(void)
        {
          int a = 12;
          int b = 2;
          int c = 0;

          c = add_inline(a,b);

          printf("Result of %d + %d = %d\n", a, b, c);
        }
```

To compile this code, save it to a file called `test.c` and use the **DS-5 Command Prompt** to enter the following command:

```
armcc -O0 test.c
```

The `-O0` option tells the compiler not to perform any optimization. This is required because our example is so simple that the compiler would otherwise inline the function call.

To see the resulting code, use the following command on the **DS-5 Command Prompt**:

```
armcc test.c -O0 -c -o-
```

This produces the following code:

```
; generated by Component: Arm Compiler 5.04 Tool: armcc [5040027]
        ; commandline armcc [-c -o- -O0 test.c]
                        ARM
                        REQUIRE8
                        PRESERVE8

                        AREA ||.text||, CODE, READONLY, ALIGN=2

                        REQUIRE _printf_percent
                        REQUIRE _printf_d
                        REQUIRE _printf_int_dec
        add_inline PROC
                        MOV      r2,r0
                        MOV      r0,#0
                        ADD      r0,r2,r1        ; Our inline assembly code
                        BX       lr
                        ENDP

        main PROC
                        PUSH     {r4-r6,lr}
                        MOV      r4,#0xc
                        MOV      r5,#2
                        MOV      r6,#0
                        MOV      r1,r5
                        MOV      r0,r4
                        BL       add_inline
                        MOV      r6,r0
                        ...
```

Register names in inline assembly code are treated as C or C++ variables. They do not necessarily relate to the physical register of the same name. In our C code, we use the variable names r5 and r6 for our operands, but the actual registers used are `r1` and `r2`.

2. Named register variables

You can use named register variables to access registers of an Arm architecture-based processor.

Named register variables are declared by combining the `register` keyword with the `__asm` keyword. The `__asm` keyword takes one parameter, a character string, that names the register. For example, the following declaration declares `R0` as a named register variable for the register `r0`:

```
register int R0 __asm("r0");
```

The following example uses inline assembly code to perform saturating addition of two integers. The code uses a named register variable `_apsr` to examine and clear the Sticky Overflow (Q) flag (bit 27) in the Application Program Status Register (APSR). The Q flag is set when saturating arithmetic operations, such as `QADD`, overflow.

Before compiling this code, you need to specify a target processor or architecture that supports the QADD instruction, for example, 7-A. Use the Target CPU (–cpu) option under project **Properties** > **C/C++ Build** > **Settings** > **Arm C Compiler 5** > **Code Generation** to specify it.

For information about the `QADD` instruction, see:

QADD - Signed saturating addition

For information about valid options for `Target CPU (--cpu)`, see:

–cpu=name compiler option

```
#include <stdio.h>

        register unsigned int _apsr __asm("apsr");

        // Clear the Q flag
        void clearQ(void)
        {
                _apsr = _apsr & ~0x08000000;
        }

        // Test the Q flag
        void testQ(void)
        {
          int armflag_Q = (_apsr>>27)&1;
          printf("  Q : %x\n\n", armflag_Q);
        }


        // Saturating addition of two operands
        void saturating_add_inline(unsigned int i, unsigned int j)
        {
          unsigned int res = 0;
          __asm
          {
                QADD res, i, j
          }
```

```
  printf("Result of %d + %d = %d\n", i, j, res);

}

int main(void)
{
  unsigned int a = 2147483645;
  int loop;

  clearQ();

  for (loop = 0; loop < 5; loop++)
  {
        saturating_add_inline(a,loop);
        testQ();
  }
}
```

You can view the following output of the code in the **Target Console** view in DS-5.

```
Result of 2147483645 + 0 = 2147483645
  Q : 0

Result of 2147483645 + 1 = 2147483646
  Q : 0

Result of 2147483645 + 2 = 2147483647
  Q : 0

Result of 2147483645 + 3 = 2147483647
  Q : 1

Result of 2147483645 + 4 = 2147483647
  Q : 1
```

3. Restrictions on inline assembly code

The inline assembler allows you to use most Arm and Thumb assembly language instructions in a C or C++ program, but there are some restrictions on the operations that you can perform.

This example uses inline assembly code to enable or disable interrupts by reading from and writing to the CPSR. There are three versions. The first version is deliberately incorrect, to show some restrictions when using the inline assembler. The errors are corrected in the second version and the third version is more efficient.

Version 1

```
        void ChangeIRQ(unsigned char NewState)
        /* NewState=1 enables IRQ interrupts, NewState=0 disables them.*/
        {
          NewState=(~NewState)<<7; /* Invert and shift to bit 7. */
          __asm                    /* Invoke the inline assembler. */
          {
                /* This code is deliberately incorrect. It is for illustration only.
*/
                STMDB SP!, {R1}         /* Save working register. */
                MRS R1, CPSR            /* Get current program status. */
                BIC R1, R1, #0x80        /* Clear IRQ disable bit flag. */
                ORR R1, R1, R0          /* OR with new value (NewState is in R0) */
                MSR CPSR_c, R1          /* Store updated program status. */
                LDMIA SP!,{R1}          /* Restore working register. */
          }
        }
```

- The processor must be in a privileged mode to execute this code.
- Using `CPSR_c` instead of `CPSR` in the `MSR` instruction ensures that you can only write to the bottom 8 bits of the CPSR. This prevents you from accidentally altering any other bits.

Building this code gives multiple errors. This is because the capabilities of the inline assembler are more limited than those of `armasm`. For example:

- You cannot directly modify the stack pointer, link register, or program counter in inline assembly code. This example tries to explicitly stack and restore R1. This is not allowed, but also is not necessary, because as shown in Version 2, the inline assembler automatically stacks and restores any working registers as required.
- Register names R0 to R12 in inline assembly code are treated as local variables. As mentioned earlier in this tutorial, they do not necessarily relate to the physical registers of the same name. When the inline assembler tries to read R1 to put it onto the stack, this causes an error because it is treated as an uninitialized variable.

Version 2

Instead of trying to stack the working registers at the beginning of an `__asm` block, use C variables in the block to hold the working data. The compiler selects the most appropriate registers to use, and the inline assembler stacks and restores them automatically, as required.

```
        void ChangeIRQ(unsigned int NewState)
        /* NewState=1 enables IRQ interrupts, NewState=0 disables them.*/
        {
          int my_cpsr;                      /* To be used by inline assembler. */

          NewState=(~NewState)<<7;           /* Invert and shift to bit 7. */
          __asm                             /* Invoke the inline assembler. */
          {
                MRS my_cpsr, CPSR               /* Get current program status. */
                BIC my_cpsr, my_cpsr, #0x80      /* Clear IRQ disable bit flag. */
                ORR my_cpsr, my_cpsr, NewState  /* OR with new value. */
                MSR CPSR_c, my_cpsr             /* Store updated program status. */
          }
        }
```

The type of the variables used in place of registers in inline assembly code must be integer-assignable because Arm registers can only hold integers.

You can see the instructions that the code compiles to in the Disassembly view in the DS-5 Debug perspective when debugging the code. Using optimization level `-O1`, this code compiles to the following instructions:

```
                MVN      r0,r0
                LSL      r0,r0,#7
                MRS      r1,APSR ; formerly CPSR
                BIC      r1,r1,#0x80
                ORR      r0,r1,r0
                MSR      CPSR_c,r0
                BX       lr
```

Version 3

```
        void ChangeIRQ(unsigned int NewState)
        /* NewState=1 enables IRQ interrupts, NewState=0 disables them.*/
        {
          int my_cpsr;
          __asm
          {
                MRS my_cpsr, CPSR                       /* Get current program
status. */
                ORR my_cpsr, my_cpsr, #0x80              /* Set IRQ disable bit
flag. */
                BIC my_cpsr, my_cpsr, NewState, LSL #7  /* Reset IRQ bit with new
value. */
                MSR CPSR_c, my_cpsr                     /* Store updated program
status. */
          }
        }
```

This version of the example is more efficient than Version 2 because it compiles to two fewer instructions:

```
MRS      r1,APSR ; formerly CPSR
ORR      r1,r1,#0x80
BIC      r0,r1,r0,LSL #7
MSR      CPSR_c,r0
BX       lr
```

4. Related information

Here are some resources related to material in this guide:

- Arm Compiler armcc User Guide
- Using the Inline and Embedded Assemblers of the Arm Compiler
- Compiler support for accessing registers using named register variables
- Restrictions on inline assembly operations in C and C++ code
- MSR (general-purpose register to PSR)