

Running 'C54x Code on 'C55x

In addition to accepting 'C55x source code, the 'C55x mnemonic assembler (masm55) also accepts 'C54x mnemonic assembly. The 'C54x instruction set contains 211 instructions; the 'C55x mnemonic instruction set is a superset of the 'C54x instruction set. The table below contains statistics on how the 'C54x instructions assemble with masm55:

original 'C54x instruction assembles as:	% of total 'C54x instruction set	% of commonly-used 'C54x instructions
one 'C55x instruction	85	95–99
two 'C55x instructions	10	1–3
more than two 'C55x instructions	5	0–2

The data in the second column characterizes the assembly of an imaginary file containing an instance of every 'C54x instruction. However, the instructions that assemble as more than two instructions are not commonly used. The data in the third column characterizes the assembly of a file containing the most commonly used 'C54x instructions. Exact percentages depend on the specific source file used.

Because of this compatibility, masm55 can assemble 'C54x code to generate 'C55x object code with bit-exact results. This assembler feature preserves your 'C54x source code investment as you transition to the 'C55x.

This chapter does not explain how to take advantage of the new architecture features of the 'C55x. For this type of information, see the *TMS320C55x DSP Programmer's Guide*.

Topic	Page
6.1 'C54x to 'C55x Development Flow	6-2
6.2 Understanding the Listing File	6-4
6.3 Handling 'C55x Reserved Names	6-6

6.1 'C54x to 'C55x Development Flow

To run a 'C54x application on the 'C55x, you must:

- ☐ Assemble each function with `masm55`. Your 'C54x application should already assemble without errors with the `asm500` assembler.
- ☐ Initialize the stack pointers `SP` and `SSP`. See Section 6.1.1.
- ☐ Handle differences in memory placement. See Section 6.1.2.

To use ported 'C54x functions along with native 'C55x functions, see Section 7.2, *Using Ported 'C54x Functions with Native 'C55x Functions*, on page 7-5.

6.1.1 Initializing the Stack Pointers

When you execute ported 'C54x code from reset, the appropriate runtime environment is already in place. However, it is still necessary to initialize the stack pointers `SP` (primary stack) and `SSP` (secondary system stack). For example:

```
stack_size    .set    0x400
stack:        .usect  "stack_section", stack_size
sysstack:     .usect  "stack_section", stack_size

              AMOV    #(stack+stack_size), XSP
              MOV     #(sysstack+stack_size), SSP
```

The stacks grow from high addresses to low addresses, so the stack pointers must be initialized to the highest address. The primary stack and the secondary system stack must be within the same 64K word page of memory.

Code that initializes the `SP` can be ported. However, the assembler cannot directly recognize the code as an `SP` initialization, and will not warn you that the `SSP` must also be initialized. Code that indirectly accesses the `SP` can also be ported. But, as above, the assembler will not warn you that the `SSP` must also be initialized.

6.1.2 Handling Differences in Memory Placement

This section describes the limitations on where you can place your code in memory.

For ported 'C54x code, a page of memory must be defined as a range of 64K (0x10000) bytes that begins on a 64K byte boundary. Edit your linker command file accordingly.

All data must be placed on page 0.

If your 'C54x code includes either of the following, all code must also be placed on page 0:

- ☐ Indirect calls with CALA
- ☐ Modification of the repeat block address registers REA or RSA

If your 'C54x code includes either of the following, it can be placed on any page, but it must fit within that page:

- ☐ Indirect branches with BACC
- ☐ Modification or use of the function return address on the stack in a non-standard way (stack unwinding)

Otherwise, code can be placed anywhere in memory.

6.2 Understanding the Listing File

The assembler's listing file (created when invoking `masm55` with the `-l` option) now provides additional information on how 'C54x instructions are mapped for the 'C55x.

Consider the following (contrived) 'C54x source file:

```
.global name

ADD      *AR2, A
STL      A, *AR3

RPT      #10
MVDK     *AR4+, name

subm     .macro mem1, mem2, reg
LD        mem1, reg
SUB       mem2, reg
.endm

subm     name, *AR6, B

MOV      T1, AC3      ; native 'C55x instruction
```

The listing file shown below has explanations inserted for clarification.

'C54x instructions with the same syntax in 'C55x (such as the `ADD` instruction below) appear without any special notation:

```
1          .global name
2
3 000000D641    ADD *AR2,A
   00000200
```

Note that `A` in the example above is accepted even though it maps to `AC0` on the 'C55x.

'C54x instructions with a different syntax in 'C55x but a single-line mapping also appear without any special notation:

```
4 000003E961    STL A, *AR3
   000005 00
```

The `STL` instruction above could be written as:

```
MOV AC0, *AR3
```

The code below shows a one-to-many instruction mapping that requires the 'C55x instructions to be in a different order than the original source. A one-to-many mapping starts with a TRANS line that echoes the original source. The multiple lines that correspond to the mapping will begin and end with the original source line number (7, in this case).

```

7  ***** TRANS      MVDK *AR4+, name
7  000006 EC31         AMAR *(&(amp;name)), XCDP ; translation of
000008 7E00                                ; MVDK *AR4+, name
00000a 0000!
5
6  00000c 4C0A         RPT #10
7  00000e EF83         MOV *AR4+, coef(*CDP+) ; translation of
000010 05                                ; MVDK *AR4+, name

```

To summarize, in the example above, the original 'C54x code:

```

RPT #10
MVDK *AR4+, name

```

was mapped to be:

```

AMAR *(&(amp;name)), XCDP
RPT #10
MOV *AR4+, coef(*CDP+)

```

A macro definition is simply echoed:

```

8
9          subm .macro mem1, mem2, reg
10         LD    mem1, reg
11         SUB   mem2, reg
12         .endm

```

A macro invocation is marked with a MACRO line. Within the macro expansion, you may see any of the cases described above.

```

13
14 ***** MACRO      subm name, *AR6, B
14 000011 A100%        LD    name, B
14 000013 D7C1         SUB   *AR6, B
000015 11

```

Native 'C55x instructions appear without any special notation. For more information on using ported 'C54x code with native 'C55x code, see Section 7.2, *Using Ported 'C54x Functions with Native 'C55x Functions*, on page 7-5.

```

15
16 000016 2253         MOV T1, AC3 ; native 'C55x

```

6.3 Handling Reserved 'C55x Names

Note that new 'C55x mnemonics and registers are reserved words. Your 'C54x code should not contain symbol names that are now used as 'C55x mnemonics or registers. For example, you should not use "T3" as a symbol name.

Your 'C54x code also should not contain symbol names that are reserved words in the 'C55x algebraic syntax. For example, you should not have a label named "return".

The 'C55x mnemonic assembler issues an error message when it encounters a symbol name conflict.

Migrating a 'C54x System to a 'C55x System

After you've ported your 'C54x code as described in Chapter 6, you must consider various system-level issues when moving your 'C54x code to 'C55x. This chapter describes:

- ☐ how to handle differences related to interrupts
- ☐ how to use ported 'C54x functions with native 'C55x functions
- ☐ non-portable 'C54x coding practices

Topic	Page
7.1 Handling Interrupts	7-2
7.2 Using Ported 'C54x Functions with Native 'C55x Functions	7-5
7.3 Non-portable 'C54x Coding Practices	7-17
7.4 Additional 'C54x Issues	7-19

7.1 Handling Interrupts

This section describes issues related to interrupts.

7.1.1 Differences in the Interrupt Vector Table

The 'C54x interrupt table is composed of 32 vectors. Each vector contains 4 words of executable code. The 'C55x vector table is also composed of 32 vectors. The vectors in both tables are the same length, but on the 'C55x, the length is counted as 8 bytes.

The order of the vectors in the interrupt vector table is documented in the data sheet for the specific device in your system. Since the order of the vectors is device-specific, any access to the IMR or IFR register needs to be updated accordingly. Likewise, if you use the TRAP instruction, its operand may need to be updated.

'C54x and 'C55x handle the contents of their vectors in different ways. To handle these differences, you must modify the 'C54x vectors themselves.

In the 'C55x vector table, the first byte is ignored, and the next three bytes are interpreted as the address of the interrupt service routine (ISR). Use the `.ivec` assembler directive to initialize a 'C55x vector entry, as shown in the examples below. For more information on the `.ivec` directive, see the description on page 4-63.

Simple Branch to ISR

If the 'C54x vector contains:

```
B isr
```

Change the corresponding 'C55x vector to:

```
.ivec isr
```

Delayed Branch to ISR

If the 'C54x vector contains:

```
BD isr  
inst_1          ; two instruction words of code  
inst_2
```

The easiest solution is to write the vector as:

```
.ivec isr
```

and move the instructions *inst1* and *inst2* to the beginning of the ISR. If the conversion of *inst1* is a single 'C55x instruction that is 4 bytes or less, it can be placed in the vector. However, *inst2* must be moved to the ISR.

Vector Contains the Entire ISR

If the 'C54x vector contains the entire 4-word ISR, as in the examples shown below:

```
; example 1
inst1
inst2
inst3
RETF

; example 2
inst1
RETFD
inst2
inst3

; example 3
CALL routine1
RETE
nop
```

you have to create the 4-word ISR as a stand-alone routine. You must then provide the address of that routine in the 'C55x vector table:

```
.ivec new_isr
```

7.1.2 Handling Interrupt Service Routines

An interrupt service routine needs to be changed only if, when ported to 'C55x,

- ☐ it includes 'C54x instructions that map to more than one 'C55x instruction, *and*
- ☐ one of the 'C55x instructions requires the use of a 'C55x register or bit as a temporary.

In this case, the new 'C55x register needs to be preserved by the routine.

The registers need to be preserved in the ISRs as long as any ported 'C54x code remains in the application. When all code has been changed to native 'C55x code, it is no longer necessary to preserve the registers.

See Section 7.2.2, *'C55x Registers Used as Temporaries*, on page 7-6 for the list of 'C55x registers that can be used as temporaries in one-to-many instruction mappings.

To ensure that an interrupt will work, you can preserve the entire list of registers. Or, you can simply preserve the register(s) used:

- 1) Assemble the ISR using `masm55` with the `-l` option to generate a listing file.
- 2) Check the listing to see if it includes any one-to-many instruction mappings. These mappings are marked by a `TRANS` comment. For more information, see Section 6.2, *Understanding the Listing File*, on page 6-4.
- 3) Determine if the one-to-many mappings actually use any of the temporaries listed in Section 7.2.2. If so, the appropriate register or bit must be pushed on the stack at the beginning of the ISR, and popped off the stack at the end.

Note that you may refer to 'C55x register names within 'C54x instruction mnemonics. For example:

```
LD *AR2,AC3
```

7.1.3 Other Issues Related to Interrupts

You should be aware of the interrupt issues described below:

- ☐ When the assembler encounters `RETE`, `RETED`, `FRETE`, `FRETED`, `RETF`, or `RETFD`, a warning will be issued. With these instructions, the assembler is processing an interrupt service routine or the interrupt vector table itself and may not be able to port the instructions correctly.
- ☐ `INTR` has the same mnemonic syntax for both 'C54x and 'C55x. Consequently, the assembler cannot distinguish when an instruction is intended for a native 'C55x interrupt (which is acceptable) or for a 'C54x interrupt (for which the interrupt number would be wrong).
- ☐ If your code writes values to `IPTR`, a nine-bit field in the `PMST` indicating the location of the interrupt vector table, you will need to modify your code to reflect the changes in the 'C55x system.

7.2 Using Ported 'C54x Functions with Native 'C55x Functions

When rewriting a 'C54x application to be completely 'C55x, consider working on one function at a time, continually testing. If you encounter a problem, you can easily find it in the changes recently made. Throughout this process, you will be working with both ported 'C54x code and native 'C55x code. Keep the following in mind:

- ☐ Avoid mixing 'C54x and 'C55x instructions within the same function.
- ☐ Transitions between ported 'C54x instructions and native 'C55x instructions should occur only at function calls and returns.
- ☐ The C compiler provides an automatic solution when you are dealing with C code calling assembly. However, see the example in Section 7.2.6 for a detailed description of using a veneer function when calling a ported 'C54x assembly function from C code.

7.2.1 Runtime Environment for Ported 'C54x Code

A runtime environment is the set of presumptions and conventions that govern the use of machine resources such as registers, status register bit settings, and the stack. The runtime environment used by ported 'C54x code differs from the environment used by native 'C55x code. When you execute ported 'C54x code from reset, the appropriate runtime environment is already in place. However, when shifting from one kind of code to the other, it is important to be aware of the status bit and register settings that make up a particular environment.

The following CPU environment is expected upon entry to a ported 'C54x function.

- ☐ 32-bit stack mode.
- ☐ The SP and SSP must be initialized to point into memory reserved for a stack. See Section 6.1.1, *Initializing the Stack Pointers*, on page 6-2.

- ☐ The status bits must be set as follows:

Status bit	Set to
C54CM	1
M40	0
ARMS	0
RDM	0
ST2[7:0] (circular addressing bits)	0

- ☐ The upper bits of addressing registers (DPH, CDPH, ARnH, SPH) must be set to 0.
- ☐ The BSAn registers must be set to 0.

7.2.2 'C55x Registers Used as Temporaries

The following 'C55x registers may be used as temporaries in one-to-many mappings generated by masn55:

- ☐ T0
- ☐ T1
- ☐ AC2
- ☐ CDP
- ☐ CSR
- ☐ ST0_55 (TC1 bit only)
- ☐ ST2_55

Interrupt routines using these registers must save and restore them. For more information, see Section 7.1.2, *Handling Interrupt Service Routines*, on page 7-3.

Native 'C55x code that calls ported 'C54x code must account for the possibility that ported code may overwrite these registers.

7.2.3 'C54x to 'C55x Register Mapping

The following 'C54x registers map to 'C55x registers as shown below:

'C54x register	'C55x register
T	T3
A	AC0
B	AC1
ARn	ARn
IMRn	IERn
ASM (status bit in ST1)	T2

7.2.4 Status Bit Field Mapping

The 'C55x status bit fields map to 'C54x status bit fields as shown below.

(a) *ST0*

Bit(s)	'C55x field	'C54x field
15	ACOV2	none
14	ACOV3	none
13	TC1	none
12	TC2	TC
11	CARRY	C
10	ACOV0	OVA
9	ACOV1	OVB
8–0	DP	DP

(b) *ST1*

Bit(s)	'C55x field	'C54x field
15	BRAF	BRAF
14	CPL	CPL
13	XF	XF
12	HM	HM
11	INTM	INTM
10	M40	none
9	SATD	OVM
8	SXMD	SXM
7	C16	C16
6	FRCT	FRCT
5	C54CM	none
4–0	ASM	ASM

(c) ST2

Bit(s)	'C55x field	'C54x field
15	ARMS	none
14–13	Reserved	none
12	DBGM	none
11	EALLOW	none
10	RDM	none
9	Reserved	none
8	CDPLC	none
7–0	ARnLC	none

(d) ST3

Bit(s)	'C55x field	'C54x field
15–8	Reserved	none
7	CBERR	none
6	MPNMC	MP/MC_
5	SATA	none
4	AVIS	AVIS
3	Reserved	none
2	CLKOFF	CLKOFF
1	SMUL	SMUL
0	SST	SST

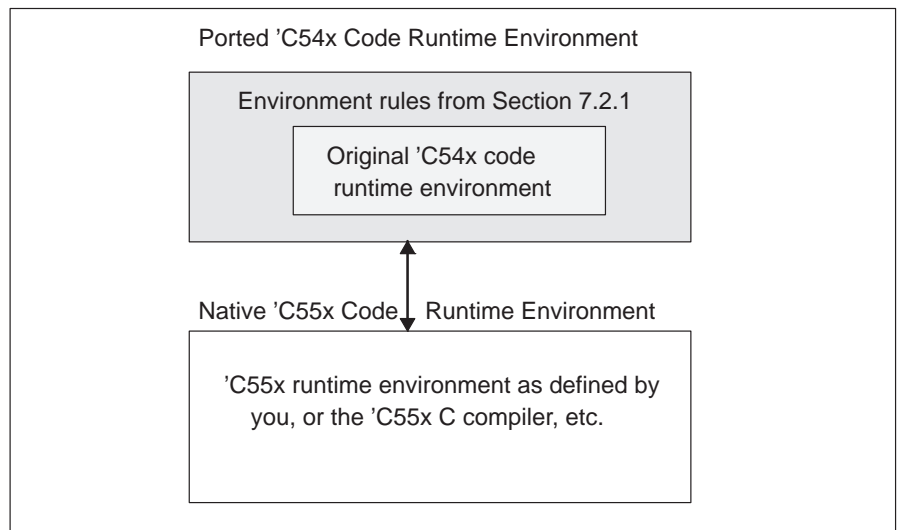
7.2.5 Switching Between Runtime Environments

The runtime environment defined in Section 7.2.1 is not complete because it only defines registers and status bits that are new with 'C55x. Registers and status bits that are not new with 'C55x inherit their conventions from the original 'C54x code. (As shown in Section 7.2.3, some registers have new names.)

If the runtime environment for your native 'C55x code differs from the environment defined for ported 'C54x code, you must ensure that, when switching between environments, the proper adjustments are made for:

- ☐ preserving status bit field values
- ☐ preserving registers
- ☐ how arguments are passed
- ☐ how results are returned

Figure 7–1. Runtime Environments for Ported 'C54x Code and Native 'C55x Code



7.2.6 Example of C Code Calling 'C54x Assembly

This example describes a technique for handling a call from compiled C code to a 'C54x assembly routine. In this example, an additional function is inserted between the native 'C55x code and the ported 'C54x code. This function, referred to as a *veneer function*, provides code to transition between the two runtime environments.

The compiler provides an automatic solution for the case of C code calling assembly. This example assumes that an automatic solution does not exist. Both the 'C54x and 'C55x C compiler runtime environments are well-defined, which makes the techniques shown in this example more concrete and easier to apply to your own situation.

Example 7–1. C Prototype of Called Function

```
short firlat(short *x, short *k, short *r, short *dbuffer,  
            unsigned short nx, unsigned short nk);
```

Example 7–2. Assembly Function _firlat_veneer

```

        .def    _firlat_veneer
        .ref    _firlat

_firlat_veneer:
; Saving Registers -----
    PSH    AR5
    ; PSH AR6    ; saved in ported C54x environment
    ; PSH AR7    ; ditto
    PSH    T2
    PSH    T3

; Passing Arguments -----
    PSH    T1    ; push rightmost argument first
    PSH    T0    ; then the next rightmost
    PSH    AR3    ; and so on
    PSH    AR2
    PSH    AR1

    MOV    AR0, AC0 ; leftmost argument goes in AC0

; Change Status Bits -----
    BSET    C54CM
    BCLR    ARMS

; Call -----
    CALL    _firlat

; Restore Status Bits -----
    BCLR    C54CM
    BSET    ARMS

; Capture Result -----
    MOV    AC0, T0

; Clear Arguments From the Stack -----
    AADD    #5, SP

; Restore Registers and Return -----
    POP    T3
    POP    T2
    ; POP AR7
    ; POP AR6
    POP    AR5

    RET

```

The veneer function is described below. It is separated into several parts to allow for a description of each segment.

Example 7–2. Assembly Function `_firlat_veneer` (Continued)*(a) Saving registers*

```

PSH    AR5
; PSH  AR6    ; saved in ported C54x environment
; PSH  AR7    ; ditto
PSH    T2
PSH    T3

```

If the 'C55x runtime environment expects that certain registers will not be modified by a function call, these registers must be saved. In the case of the 'C55x C compiler environment, registers XAR5–XAR7, T2, and T3 must be saved. Because 'C54x code cannot modify the upper bits of the XAR n registers, only the lower bits need to be preserved. The instructions that push AR6 and AR7 are commented out because the runtime environment of the 'C54x ported code (as defined by the 'C54x C compiler) presumably saves these registers. A more conservative approach would be to save these registers anyway.

(b) Passing arguments

```

PSH    T1      ; push right-most argument first
PSH    T0      ; then the next argument
PSH    AR3     ; and so on
PSH    AR2
PSH    AR1

MOV     AR0, AC0 ; left-most argument goes in AC0

```

Arguments passed from native 'C55x code must be placed where the ported 'C54x code expects them. In this case, all arguments are passed in registers. According to the calling conventions of the 'C55x C compiler, the arguments to the `firlat()` function will be passed, and the result returned, in the registers shown below.

```

T0          AR0      AR1      AR2      AR3
short firlat(short *x, short *k, short *r, short *dbuffer,
              T0          T1
              unsigned short nx, unsigned short nk);

```

For more information on the C compiler's calling conventions, see the *Runtime Environment* chapter of the TMS320C55x *Optimizing C Compiler User's Guide*.

The ported 'C54x environment expects the first argument to be in A (AC0 on 'C55x) and the remaining arguments to be placed on the stack, in reverse order of appearance in the argument list. The right-most argument (T1) is pushed onto the stack first. The next argument (T0) is then pushed onto the

stack. The argument placement continues until the left-most argument (AR0) is reached. This argument is copied to AC0.

Example 7–2. Assembly Function `_firlat_veneer` (Continued)

(c) Changing status bits

BSET	C54CM
BCLR	ARMS

It is necessary to change the status settings of the native 'C55x code to the settings required by ported 'C54x code. These settings are shown in Section 7.2.1 on page 7-5. In this case, only the C54CM and ARMS bits need to be changed.

(d) Function call

CALL	<code>_firlat</code>
------	----------------------

Now that registers have been saved and status bits set, the call to ported 'C54x code can be made.

(e) Restoring status bits

BCLR	C54CM
BSET	ARMS

After the call, restore the status bits to the settings required by the native 'C55x environment.

(f) Capturing results

MOV	AC0, T0
-----	---------

The ported 'C54x environment returns the result in AC0, while the native 'C55x environment expects the result to be returned in T0. Consequently, the result must be copied from AC0 to T0.

(g) Clearing arguments from the stack

AADD	#5, SP
------	--------

At this point, you should decrease the stack by the number of words originally needed to push the function's passed arguments. In this case, the amount is 5 words. Because the stack grows from high addresses to low addresses, addition is used to change the stack pointer from a low address to a higher one.

Example 7–2. Assembly Function `_firlat_veneer` (Continued)*(h) Restoring registers and returning*

```
POP     T3
POP     T2
; POP   AR7
; POP   AR6
POP     AR5

RET
```

Restore the registers saved at the beginning of the function, and return.

7.2.7 Example of 'C54x Assembly Calling C Code

This example contains a 'C54x assembly routine calling a compiled C routine. Because the C routine is recompiled with the 'C55x C compiler, the assembly routine must handle the differences between the ported 'C54x runtime environment and the runtime environment used by the 'C55x compiler.

If you use a different runtime environment for your 'C55x code, your code changes will differ slightly from those in this example. However, you must still consider the issues addressed here.

Example 7–3. Prototype of Called C Function

```
int C_func(int *buffer, int length);
...
```

The assembly function performs some calculations not shown in this example and calls the C function. The returned result is copied to the C global variable named `result`. Further calculations, also not shown here, are then performed.

Example 7–4. Original 'C54x Assembly Function

```

; Declare some data -----
                .data
buffer:         .word 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
BUFLen         .set 11
                .text

; Assembly routine starts -----
callsc:
; original 'C54x code ...

; Call C function (original 'C54x code) -----
                ST #BUFLen, *SP(0) ; pass 2nd arg on stack
                CALLD #_C_func
                LD #buffer, A      ; pass 1st arg in A

; Effects of calling C:
; May modify A, B, AR0, AR2-AR5, T, BRC
; Will not modify AR1, AR6, AR7
; May modify ASM, BRAF, C, OVA, OVB, SXM, TC
; Will not modify other status bits
; Presume CMPT = 0, CPL = 1

                STL A, *(_result) ; Result is in accumulator A

; original 'C54x code ...

                RET

```

To use this assembly function on 'C55x, it is necessary to change the call to the C function.

Example 7–5. Modified Assembly Function

```

; declare data as shown previously
; Assembly routine starts -----
callsc:
;   ported 'C54x code ...

; Call C function (Change to 'C55x compiler environment)
        AMOV #buffer,AR0    ; pass 1st ptr arg in AR0
        MOV #BUFLN,T0       ; pass 1st int arg in T0
;   compiler code needs C54CM=0, ARMS=1
        BCLR C54CM    ; clear 'C54x compatibility mode
        BSET ARMS     ; set AR mode
        CALL _C_func   ; no delayed call instruction

; Effects of calling C:
;   May modify AC0-AC3, XAR0-XAR4, T0-T1
;   May modify RPTC,CSR,BRCx,BRS1,RSAX,REAX
;   Will not modify XAR5-XAR7,T2-T3,RETA
;   May modify ACOV[0-3],CARRY,TC1,TC2,SATD,FRCT,ASM,
;   SATA,SMUL
;   Will not modify other status bits

        MOV T0, *(_result) ; Result is in T0

; could use *abs16(_result) if all globals are in the
; same 64K word page of data

; Change back to ported 'C54x environment -----
        BSET C54CM    ; reset 'C54x compatibility mode
        BCLR ARMS     ; disable AR mode

;   ported 'C54x code ...

        RET

```

The arguments are passed according the calling conventions described in the *Runtime Environment* chapter of the TMS320C55x *Optimizing C Compiler User's Guide*. The status bits modified are the only ones that differ between the 'C54x ported runtime environment and the native 'C55x environment (in this case, as defined by the 'C55x C compiler).

The comments about the effects of calling C (the registers and status bits that may or may not be modified) do not impact the code shown. But these effects can impact the code around such a call.

For example, consider the XAR1 register. In the 'C54x compiler environment, AR1 will not be modified by the call. In the 'C55x compiler environment, XAR1 may be modified. If code before the call to C_func loads a value into AR1, and code after the call reads AR1 for that value, then the code, as written, will not work on 'C55x. The best alternative is to use an XARn register that is saved by C routines, such as XAR5.

7.3 Non-Portable 'C54x Coding Practices

Some 'C54x coding practices cannot be ported to the 'C55x. The assembler will warn you of certain detectable issues, but it cannot detect every issue. The following coding practices are not portable:

- ☐ Any use of a constant as a memory address. For example:

```
B 42
ADD @42,A
SUB @symbol+10,b
```

- ☐ Memory initialized with constants that are later interpreted as code addresses. For example:

```
table: .word 10, 20, 30
...
LD      @table,A
CALA
```

- ☐ Using data as instructions. For example:

```
function:
    .word 0xabcd ; opcode for ???
    .word 0xdef0 ; opcode for ???
...
    CALL function
```

- ☐ Out of order execution, also known as pipeline tricking.
- ☐ Code that creates or modifies code.
- ☐ Repeat blocks spanning more than one file.
- ☐ Branching/calling unlabeled locations. Or, modifying the return address to return to unlabeled location. This includes instructions such as:

```
B $+10
```

- ☐ Using READA and WRITEA instructions to access instructions and not data.

- ❑ Using READA/WRITA with an accumulator whose upper bits are not zero.

The READA/WRITA instruction on 'C54x devices (other than 'C548 or later) uses the lower 16 bits of the accumulator and ignores the upper 16 bits. 'C548 and later devices, however, use the lower 23 bits. The assembler cannot easily know the device for which the code is targeted. It assumes 'C548 or later. Consequently, code for 'C548 and later devices will map with no problems. Code for devices other than these will not run.

7.4 Additional 'C54x Issues

This section contains some additional system issues.

If your 'C54x code:

- ☐ uses a *SP(offset) operand in the MMR slot of MMR instructions like LDM
- ☐ copies blocks of code, usually from off-chip memory to on-chip memory
- ☐ uses memory-mapped access to peripherals
- ☐ uses repeat blocks larger than 32K after mapping to 'C55x
- ☐ uses the branch conditions BIO/NBIO

you may need to modify this code to use native 'C55x instructions.

You should also be aware of the following issues:

- ☐ The 'C5x-compatibility features of the 'C54x are not supported on 'C55x.
- ☐ RPT instructions, non-interruptible on 'C54x, can be interrupted on 'C55x.
- ☐ When an operation overflows into the guard bits, and then a left-shift clears the guard bits, the 'C54x has the value of zero while the 'C55x has a saturated value.
- ☐ The 'C54x and 'C55x mnemonic assembly languages differ significantly in the representation of instruction parallelism.

The 'C55x implements two types of parallelism: implied parallelism within a single instruction (using the :: operator), and user-defined parallelism between two instructions (using the || operator). The 'C54x implements only one type of parallelism, which is analogous to implied parallelism on the 'C55x. However, 'C54x parallelism uses parallel bars (||) as its operator. 'C55x parallelism is documented in the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*.

- When using indirect access with memory-mapped access instructions, such as:

```
STM #0x1234, *AR2+
```

the 'C54x masks the upper 9 bits of the AR_n register. This masking effectively occurs both before and after the post-increment to AR2. For example:

```
; AR2 = 0x127f
STM #0x1234, *AR2+ ; access location 0x7f
; AR2 = (0x7f + 1) & ~7f ==> 0
```

However, the 'C55x assembler maps this as:

```
AND #0x7f, AR2
MOV #0x1234, *AR2+ ; note no masking afterward
```

to account for the possibility of a memory-mapped address for AR2.