C54x-to-C55x Code Migration Reference Guide

Preliminary Draft

This document contains preliminary data current as of the publication date and is subject to change without notice.

Literature Number: SPRU429
March 2001

Texas Instruments

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

This manual describes techniques for migrating code from a TMS320C54x™ (C54x™) DSP to a TMS320C55x™ (C55x™) DSP. It also describes techniques for optimizing your code during and after the migration.

Notational Conventions

This document uses the following conventions.

- The device number TMS320C55x is often abbreviated as C55x.
- If an overbar is above the name of a signal (for example, $\overline{\text{BIO}}$), the signal is active low.
- Code examples are shown in a `special typeface`.
- In most cases, hexadecimal numbers are shown with the suffix h. For example, the following number is a hexadecimal 40 (decimal 64):

  40h

  Similarly, binary numbers usually are shown with the suffix b. For example, the following number is the decimal number 4 shown in binary form:

  0100b

- Bits and signals are sometimes referenced with the following notations:

Notation	Description	Example
Register(n–m)	Bits n through m of Register	AC0(15–0) represents the 16 least significant bits of the register AC0.
Bus[n:m]	Signals n through m of Bus	A[21:1] represents signals 21 through 1 of the external address bus.

❑ The following terms are used to name portions of data:

Term	Description	Example
LSB	Least significant bit	In AC0(15–0), bit 0 is the LSB.
MSB	Most significant bit	In AC0(15–0), bit 15 is the MSB.
LSByte	Least significant byte	In AC0(15–0), bits 7–0 are the LSByte.
MSByte	Most significant byte	In AC0(15–0), bits 15–8 are the MSByte.
LSW	Least significant word	In AC0(31–0), bits 15–0 are the LSW.
MSW	Most significant word	In AC0(31–0), bits 31–16 are the MSW.

Related Documentation From Texas Instruments

The following books describe the TMS320C55x™ (C55x™) DSP generation and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C55x Technical Overview (literature number SPRU393). This overview is an introduction to the TMS320C55x™ digital signal processor (DSP). The TMS320C55x is the latest generation of fixed-point DSPs in the TMS320C5000™ DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features of the TMS320C55x DSPs.

TMS320C55x DSP CPU Reference Guide (literature number SPRU371) describes the architecture, registers, and operation of the CPU for the TMS320C55x™ digital signal processors (DSPs).

TMS320C55x DSP Peripherals Reference Guide (literature number SPRU317) describes the peripherals, interfaces, and related hardware that are available on TMS320C55x™ (C55x™) DSPs. It also describes how you can use software (idle configurations) to turn on or off individual portions of the DSP, so that you can manage power consumption.

TMS320C55x DSP Mnemonic Instruction Set Reference Guide (literature number SPRU374) describes the TMS320C55x™ DSP mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the algebraic instruction set.

TMS320C55x DSP Algebraic Instruction Set Reference Guide (literature number SPRU375) describes the TMS320C55x™ DSP algebraic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

TMS320C55x Programmer's Guide (literature number SPRU376) describes ways to optimize C and assembly code for the TMS320C55x™ DSPs and explains how to write code that uses special features and instructions of the DSP.

TMS320C55x Assembly Language Tools User's Guide (literature number SPRU280) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for TMS320C55x™ devices.

TMS320C55x Optimizing C Compiler User's Guide (literature number SPRU281) describes the TMS320C55x™ C Compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for TMS320C55x devices.

TMS320 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that provide various products that serve the TMS320™ DSP family. A myriad of products and applications are offered—software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

Trademarks

Contents

Figures

Tables

Examples

Chapter 1

Introduction

Topic Page

1.1 About This Document

This document provides code migration examples that illustrate the recommended C54x-to-C55x code porting process. This document complements but does not replace the code migration information contained in the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280). SPRU280 includes the minimum steps required to run your C54x code on a C55x DSP and presents the software system-level issues that need to be addressed as part of the application code porting.

This document does not cover hardware migration or differences between the C54x and C55x peripherals and external memory interfaces. While performing a hardware system migration, you may want to consult the *TMS320C55x DSP CPU Reference Guide* (SPRU371) and the *TMS320VC5510 Fixed-Point Digital Signal Processor* data sheet (SPRS076).

1.2 The C54x-to-C55x Code Porting Process

The TMS320C55x™ (C55x™) DSPs are source compatible with the TMS320C54x™ (C54x™) DSPs and are able to run C54x code, producing bit-exact results. Migration of C54x mnemonic code to C55x mnemonic code can be achieved with minimal user intervention through the use of the C55x mnemonic assembler, MASM55. MASM55 assembles both C54x and the new C55x native instruction. To take full advantage of the C55x architecture, manual code modification of medium/high MIPS functions using C55x native instructions is suggested.

The recommended approach to migrate a C54x application code to C55x is **the partial native/code re-assembly approach** described below. *This should be seen as a recommendation not as the only possible code porting methodology.*

❑ **Phase 1: Code re-assembly using MASM55**

The mnemonic C55x assembler (MASM55) can be used to port a C54x algorithm to the C55x. MASM55 assembles the C54x mnemonic source and produces C55x object code. This approach saves time as it requires minimal user intervention. The effort involved is typically independent of the size of the C54x code. User intervention may be required to:

- Port system level code: C54x and C55x are source compatible in terms of the CPU instruction set. However, C54x and C55x are different in their memory maps, peripherals, stack management, and interrupts. Manual modification of system initialization code may be required.
- Remove non-portable code: MASM55 will port all C54x code with few exceptions related to hard-coding of addresses and open pipeline tricks

❑ **Phase 2: Selected code optimization of medium MIPS functions (Optional)**

The original C54x code does not exploit all the C55x architectural features. Limited code modification of the C54x medium MIPS functions is suggested to quickly get better cycle count and code size. For example, you can quickly make the C54x code more optimal by:

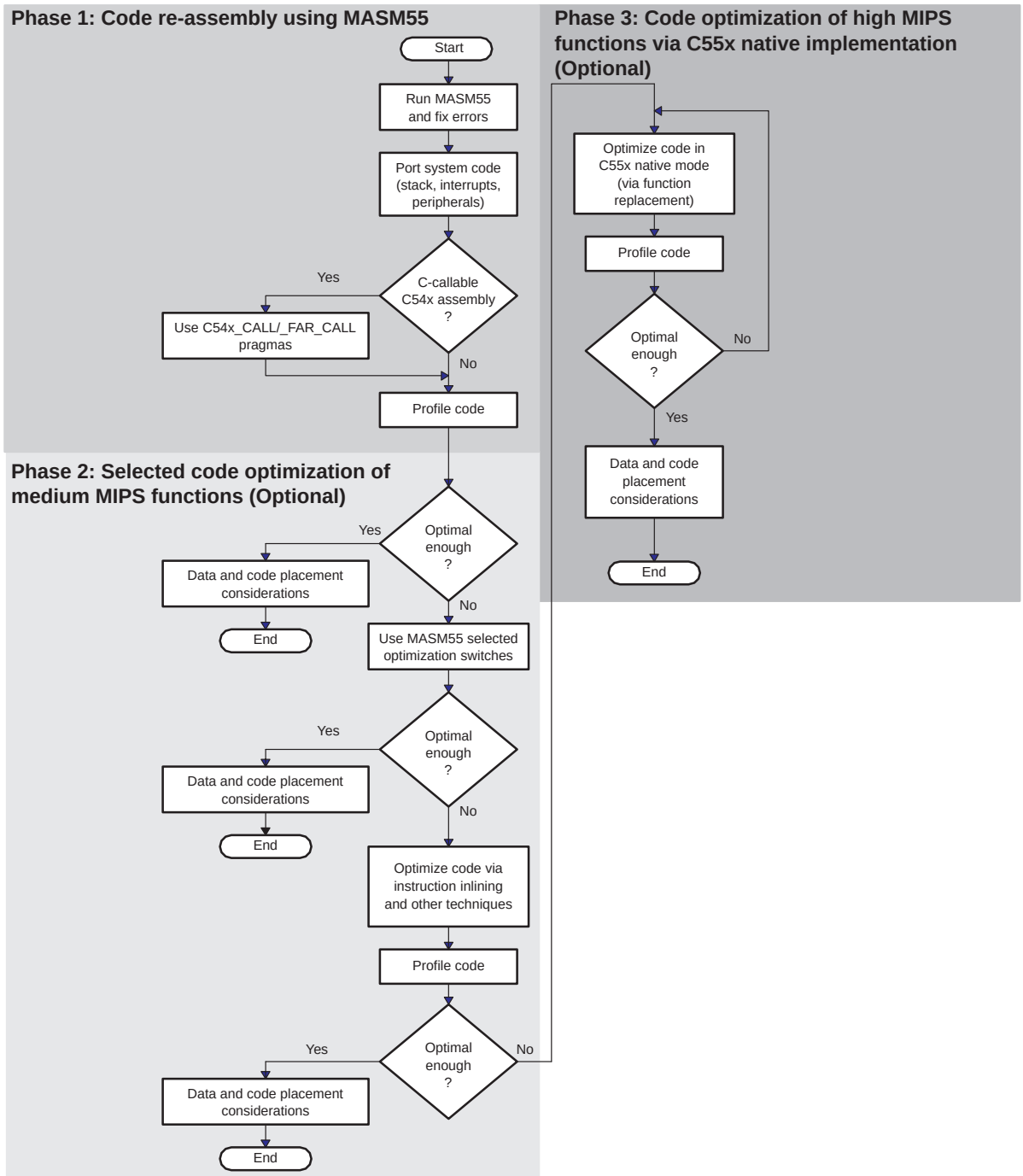
- Using MASM55 special optimization switches
- Using few C55x native instructions in your original C54x code

❑ **Phase 3: Code optimization of high MIPS functions via C55x native implementation (Optional)**

To optimize further, high MIPS consuming functions that could take advantage of the new C55x features should be implemented using C55x native instructions. The idea is to cycle profile all the functions and optimize a subset of the functions covering 80% of the algorithm MIPS. The reason is that in most applications, the high MIPS consuming modules represent only a small portion of the code size. Therefore, the effort is considerably less and the payback is substantial. However, the impact of new C55x architectural features, such as such as dual mac, dual store/load capability, nested block repeats and additional circular buffers, in those functions need to be analyzed to make a decision on native coding.

The C54x/C55x code porting process is illustrated in the flowchart presented in Figure 1–1. It is important to note that Phase 2 and Phase 3 steps are optional depending on your performance optimization goals. For example, you might be satisfied with the code performance that you obtain with just Phase 1, given the higher clock rate of C55x DSPs.

Figure 1–1. Flowchart of the C54x-to-C55x Code Porting Process



1.3 What Performance to Expect in C54x Ported Code

The performance of C54x code ported to the C55x depends on 3 main factors:

1) The approach selected when porting assembly code
2) The type of assembly code being ported
3) The mix of assembly code versus C code

1.3.1 The Approach Selected When Porting Assembly Code

There are three possible approaches when porting C54x assembly code to the C55x: code re-assembly using MASM55, a full C55x native coding approach, and a partial native/code re-assembly approach. *In this document we recommend and cover the partial native/code re-assembly approach that is a mix of the other two approaches.* However, it is important to decide on the approach with a clear understanding of the trade-offs involved.

- **Code re-assembly using MASM55 approach:** A C54x mnemonic assembly implementation can be ported to the C55x by using the mnemonic C55x assembler (MASM55). MASM55 assembles the C54x mnemonic source and produces C55x object code. This approach corresponds to the Phase 1 of the C54x-to-C55x code porting process presented in this document.

  Advantages

  It is not a time consuming process and the effort involved is independent of the size of the C54x code being ported in most cases.

  Disadvantages

  - Manual code modification may be required to address differences between C54x and C55x in system-level code (interrupt and peripheral register initialization mostly)
  - Masm55 does not exploit all the C55x architectural features and selected code modification may be required to get better cycle count and code size

  Expected Performance

  Table 1 shows a typical code size and cycle performance using the code re-assembly using MASM55 approach. After some of the C54x code optimization techniques presented in this document, **an average C54x/MASM55 ratio of 0.8 and 0.95 in cycle and code size respectively can be expected**. C55x performance have some degradation and for this reason this approach is typically recommended for less MIPS consuming functions.

- **full C55x native coding approach:** This approach involves a code re-write using C55x native instructions. This approach corresponds to the Phase 3 of the C54x-to-C55x code porting process presented in this document.

  Advantages

  - All architectural features of C55x can be exploited.
  - Developer has complete control over code size and cycles

  Disadvantages

  Time consuming process.

  Expected Performance

  Table 1–1 shows a typical code size and cycle performance using a full C55x native approach. ***With this method an average C54x/ C55x ratio of 1.4 to 1.8 can be achieved depending on the type of code.*** The more MAC intensive the code is the better the ratio would be. Refer to this document and the *TMS320C55x Programmer's Guide* (SPR376) for recommended C55x programming techniques.

- **Partial native/code re-assembly approach (Recommended Method):** In C55x, it is possible to mix re-assembled C54x code with code using new C55x native instructions. Hence, to port an algorithm to the C55x, a mix of migrating techniques using MASM55 and native implementation can be used which can ensure quick migration with a good memory and MIPS performance gain.

  Advantages

  Enables the developer to exploit C55x new architectural features for MIPS intensive functions and quickly port the remaining less MIPS consuming functions via MASM55.

  Expected Performance

  Memory and cycle performance with this approach is application code specific. It depends on the code characteristics and on the mix of MASM55 re-assembled code versus C55x native code used. ***Table 1–1 shows that an average C54x/ C55x cycle ratio of 1.35 to 1.6 can be achieved depending on the kind of code.*** The more MAC intensive the code is the better would be the ratio.

  The use of MASM55 re-assembled code versus C55x native code should be decided on a function by function basis. The following recommendations apply when selecting what functions to implement in C55x native code:

- Functions that consume 80% of MIPS of the algorithm are good candidates for C55x native coding. The reason is that in most applications, the high MIPS consuming modules represent only a small portion of the code size. Therefore, the effort is considerably less and the payback is substantial. However, as explained below the type of operations in these functions needs also to be considered.
- The impact of new C55x architectural features on the function needs to be analyzed to make a decision on native coding. MIPS consuming functions that could take advantage of new C55x architectural features such as dual mac, dual store/load capability, nested block repeats and additional circular buffers are candidates for native coding. You can quickly make this analysis from the fixed-point C model of the algorithm if available.

In summary,

Partial native/code re-assembly approach following the 80% MIPS rule is the recommended method. Non-critical functions of the algorithm (less MIPS consuming functions) can be ported using MASM55 (with selected code changes) while the MIPS consuming functions can be implemented using C55x native instructions.

Table 1–1. C54x vs. C55x Code Porting Performance Average

	C54x/C55x Cycle Ratio	C54x/C55x Code Size Ratio
MASM55	0.8	0.9–0.95
MASM55 with selected code changes	0.9	0.95
C55x Native	1.4–1.8	0.95–1.1 (for DSP loop-type code) 1.25–1.3 (for control-type code)
C	not available	1.3–1.4

Note: Larger means better performance for the C55x.

1.3.2 The Type of Assembly Code Being Ported

A typical application consists of a mix of control code and DSP-loop code. Control code tends to dominate in size, while DSP-loop code tends to dominate in cycles. Therefore an optimal DSP processor architecture should concentrate on decreasing code size for control code and decreasing cycle count for DSP code. The C55x architecture was designed to achieve just that.

C55x control code will shrink in size compared with C54x control code, as seen in Table 1–1. Assembly code can get larger by 10-25% with DSP loop code because the C55x DSP generation offers a more powerful and orthogonal instruction set than the C54x DSP generation, and this requires more instruction encoding bits. This increase in code size is balanced by a decrease in cycle count, which can be seen in Table 1–2.

Table 1–2. C54x Versus C55x Cycle Comparison in DSPLIB Code

Benchmark	C55x Cycle Count	C54x Cycle Count
Real block FIR	nx/2 (4+nh)	4+nx(4+nh)
complex block FIR	nx(2+ 2*nh)	nx(13 + 8*nh)
iircas4	nx(5+4*nbiq)	nx(11 + 4*nbiq)
dlms kernel	5+ 2*nh	12 + 2*nh
maxval	nx	2*nx

Notes: 1) nx = number of elements in the vector
2) nh = number of filter coefficients
3) nbiq = number of filter biquads
4) These kernels reflect a small subset of the C55x and C54x DSPLIB kernels. A complete set of C54x and C55x DSPLIB benchmarks can be found in the API descriptions in the *Optimized DSP Library for C Programmers on the TMS320C54x* (SPRA480) and the *TMS320C55x DSP Library Programmer's Reference* (SPRU422).

1.3.3 The Mix of Assembly Code Versus C Code

The C55x DSP generation has a more efficient compiler engine than the C54x DSP generation. Benchmarks on the C55x C compiler show a 30–40% code size reduction compared with those on the C54x C compiler (see Figure 1–2). The more C code your application has, the better code size and cycle improvements your ported code will see.

Combined C and assembly code usually gets smaller as shown in Figure 1–3, an example of a G.723 code porting. The analysis above assumes 50% C code, 50% assembly code and C54x code ported using MASM55 and the C55x compiler tools.

In Figure 1–3 (second bar), the G.723 assembly portion ported with MASM55 code re-assembly expanded by 22%, while the G.723 C portion shrunk by –21.67%, giving you an overall C-plus-assembly code growth of 6.38%.

Using C55x compiler optimization switches, the C code size can shrink to 30–40% of the C54x C code size. This is illustrated in Figure 1–3 (third bar).

Figure 1–2. C-Compiler Benchmarks

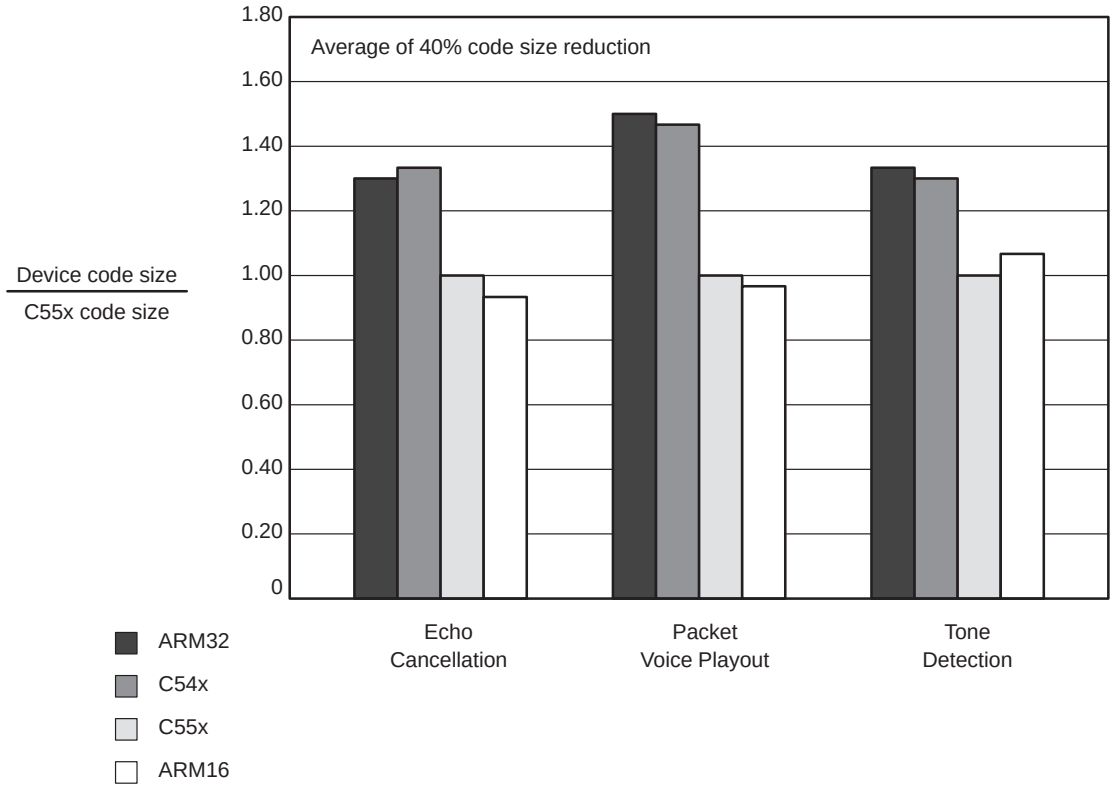
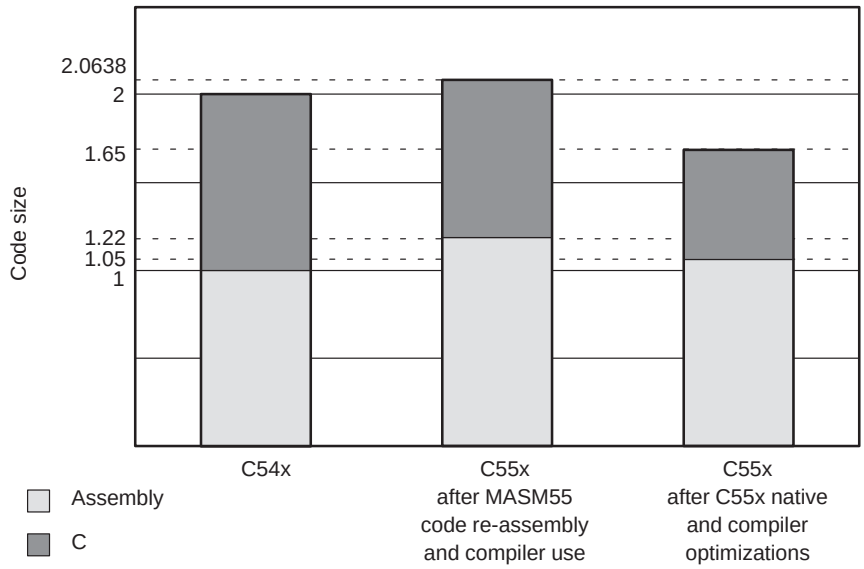


Figure 1–3. Effect of Migration on Code-Size in G.723 Code Porting



Chapter 2

Phase 1: Code Re-assembly Using MASM55

Topic Page

2.1 Overview

The first Phase of the C54x-to-C55x code porting process is to achieve functional code using the C55x mnemonic assembler MASM55. MASM55 can port C54x mnemonic code to C55x with minimal user intervention. User intervention is required to:

- ❑ Address system-level issues such as differences in memory maps, peripherals, stack management, and interrupts
- ❑ Replace code that might not be portable

This chapter covers, in detail, these two steps. The steps are summarized in Table 2–1.

Table 2–1. Phase 1: Code Re-assembly Using MASM55

STEP 1: Dealing with Non-Portable Code	**STEP 2: Porting System-Level Code**
❑ Modify code that uses hard-coded addresses and offsets. ❑ Modify code that takes advantage of the C54x non-protected pipeline. ❑ Replace code that depends on the condition of the C54x BIO pin. ❑ Modify code that uses reserved symbols of the C55x code generation tools. ❑ Modify code that uses the ARP register.	**Stack** ❑ Add code to initialize the system stack. **Interrupts** ❑ Rewrite the interrupt vector table (IVT). ❑ Rewrite code that initializes the interrupt vector pointer (IPTR). ❑ Modify code that initializes the IMR and IFR registers. ❑ Modify the operands of the TRAP and INTR instructions. ❑ Preserve MASM55 temporary register values in nterrupt service routines. **Peripherals** ❑ Rewrite code that accesses peripheral and EMIF (external memory interface) registers. ❑ Replace code that accesses the C54x I/O space. **C-Callable Assembly** ❑ Use C54x_CALL/C54x_FAR_CALL pragmas for C54x C-callable assembly code. **Special Case** ❑ Inspect instructions that use the same auxiliary register (AR) for Xmem and Ymem and that modify the AR of the Ymem operand. **Linker Command File Changes** ❑ Allocate the data stack and the system stack in the same 64K-word page. ❑ Allocate all data in memory page 0 (the first 64K words of memory). ❑ Make sure DP addressing arrays maintain the same C54x 128-word page boundary. ❑ Keep code being called by CALA in memory page 0 (the first 64K words of memory). ❑ Keep code reached by BACC in the same 64K-word page as the calling code. ❑ Adjust for differences in the memory map and for byte addressing.

To execute C54x code ported with MASM55 correctly, a C55x DSP needs to be in the C54x-compatible mode (C54CM = 1) and must meet all the associated conditions shown in Table 2–2. All of the conditions except the stack mode condition are the defaults forced by a DSP hardware reset. The stack mode is determined by the first byte of the reset vector, as described in the *TMS320C55x DSP CPU Reference Guide* (SPRU371).

Table 2–2. MASM55 C54x Compatibility Context Requirements

Required Condition	Description
ST1(C54CM) = 1	The C54x-compatible mode is enabled.
ST1(M40) = 0	The D-unit of the CPU is in the 32-bit mode rather than the 40-bit mode.
ST2(ARMS) = 0	C54x-compatible options are available for indirect addressing.
ST2(RDM) = 0	C54x-compatible rounding is used.
ST2(7–0) = 0	Circular addressing is not forced for any of the auxiliary registers, AR0–AR7.
DPH = 0, SPH = 0, AR0H–AR7H = 0, CDPH = 0	All data is accessed in the first 64K words of memory.
BSA01 = 0, BSA23 = 0, BSA45 = 0, BSA67 = 0, BSAC = 0	The C55x circular buffer start address registers are cleared so that they do not affect circular addressing.
Stack mode = 32-bit	The data stack and the system stack act as a single 32-bit stack: When you access the data stack, the pointers for both stacks are modified by the same increment. The C55x fast-return registers (RETA and CFCT) are not used.

2.2 Step 1: Dealing with Non-Portable Code

Certain C54x instructions/code practices are not portable to the C55x DSPs. MASM55 flags most of these cases with “ERROR” or “REMARK” (indicating a weaker error), with the exception of open pipeline tricks. In this step, we must remove/replace those pieces of code to allow MASM55 to complete the C54x code assembly process.

Table 2–3 summarizes the cases in which C54x code is not portable, and the sections that follow the table describe how to handle such code.

Table 2–3. Cases in Which C54x Code is Not Portable

Case	Example	MASM55 Behavior
Hard-coded addresses/offsets (see section 2.2.1)	B #1233h	REMARK
Open-pipeline tricking (see section 2.2.2)	LD #1, A XC 1, AEQ	REMARK (for XC) (Does not detect other cases)
$\overline{\text{BIO}}$-pin-related code (see section 2.2.3)	XC 1, BIO	ERROR
New reserved symbols (see section 2.2.4)	B $	ERROR
ARP-related code (see section 2.2.5)	LD #3, ARP	ERROR

2.2.1 Modify Code That Uses Hard-Coded Addresses and Offsets

In C55x DSPs, addresses in program space are given in bytes, while those in C54x DSPs are given in words. Therefore, instructions that use hard-coded addresses/offsets (for example, #1000h) will not address the correct values, and for that reason, are flagged as MASM55 errors.

WHAT TO DO: It is good programming practice to avoid using hard-coded addresses or offsets. If hard-coded values have been used, replace them with labels so that the linker will resolve them at link time.

2.2.2 Modify Code That Takes Advantage of the C54x Non-Protected Pipeline

The C54x instruction pipeline was not protected, and therefore had the potential to cause some out-of-order instruction execution. Although rarely done in practice, it was possible to take advantage of the open pipeline to save cycles.

As an example, consider the following C54x code. When the C54x CPU executes the XC (conditional execute) instruction, the content of accumulator A is sampled two cycles before the XC instruction is executed. Therefore, the action taken by the XC instruction depends on the LD instruction, not on the ADD instruction. Because the LD instruction makes the content of A equal to 0, the SUB instruction is executed.

```
LD 0, A
NOP
ADD #1, A
XC 1, AEQ
SUB #5, B
```

To make programming easier, each C55x DSP offers a protected pipeline to ensure in-order execution of instructions. In a C55x pipeline, the action taken by the XC instruction above would depend on the ADD instruction. As a result, the SUB instruction would *not* be executed.

WHAT TO DO: MASM55 *does not* detect the usage of all nonprotected pipeline cycles. For this reason, we suggest you pre-process your C54x assembly file by running the C54x assembly with the pipeline open detection enabled (asm500 –pw). However, be aware that asm500 –pw can detect only some open pipeline cases (not all).

2.2.3 Replace Code That Depends on the Condition of the C54x $\overline{\text{BIO}}$ Pin

C54x devices have a $\overline{\text{BIO}}$ pin, but C55x devices do not. As a result, conditional instructions that test the condition of the $\overline{\text{BIO}}$ pin cannot be ported to C55x devices. MASM55 flags an error in those cases.

WHAT TO DO: Change the C54x code that tests the $\overline{\text{BIO}}$ pin to test something else, such as one of the pins of the C55x general-purpose I/O port (GPIO). Example 2–1 shows code that tests the C55x IO0 pin. Replacing the C54x $\overline{\text{BIO}}$ code also means modifying the hardware design to use the newly chosen pin.

Example 2–1. C55x Native Code to Test a GPIO Pin (IO0 Pin)

```
;...

IODIR  .set   0x3400

IODATA.set   0x3401

   .text

start:
   AND    #0FFFEh, port(#IODIR) ; Configure IO0 pin as input
   MOV    port(#IODATA), T3     ; Read GPIO data register into T3
   AND    #0001h, T3            ; Mask off all bits except the one for IO0
   BCC    start, T3 == 0        ; If T3=0 => IO0 is low, goto start

;...
```

2.2.4 Modify Code That Uses Reserved Symbols of the C55x Code Generation Tools

C55x code generation tools use some new reserved symbols. Appendix A provides a list of the symbols.

WHAT TO DO: If your C54x code uses any reserved symbols, you must rename them.

2.2.5 Modify Code That Uses the ARP Register

The ARP (auxiliary register pointer) of the TMS320C5x™ and C54x DSPs is not supported in C55x DSPs.

WHAT TO DO: Unless you ported code from a C5x™ DSP to the C54x DSP, you should not be encountering this problem. If this issue does arise, you can replace ARP addressing with ARn addressing. For example if the code makes ARP = 3 and then uses *+, you can remove the instruction that loads ARP and then replace *+ with *AR3+.

2.3 Step 2: Porting System-Level Code

C55x DSPs are instruction source compatible with C54x DSPs. However, device differences related to interrupts, stack operation, peripherals, and memory mapping make code modification required to achieve fully functional application code.

Table 2–4 lists the steps required to make your code functional on a C55x DSP that is running under the conditions listed in Table 2–2. When you run MASM55, you will get REMARKs for some (but not all) of the issues presented in Table 2–4. To correct any of the issues mentioned in the table, manual inspection and code modification are required.

Most of the steps in Table 2–4 are illustrated through the software counter code of Example 2–2 (page 2-8). This simple example increments a low resolution counter until a certain value is reached and then triggers an interrupt to increment a high resolution counter.

Table 2–4. Step 2: Porting System-Level Code

Category	MASM55 Behavior†	See ...
Stack		Section 2.3.1, page 2-9
❑ Add code to initialize the system stack.	REMARK	
Interrupts		
❑ Rewrite the interrupt vector table (IVT).	REMARK	Page 2-9
❑ Rewrite code that initializes the interrupt vector pointer (IPTR) in the PMST register.	REMARK	Page 2-12
❑ Modify code that initializes the IMR and IFR registers.	REMARK	Page 2-14
❑ Modify the operands of the TRAP and INTR instructions.	Does not detect	Page 2-15
❑ Preserve MASM55 temporary register values during interrupt service routines.	REMARK on RETE instruction	Page 2-16
Peripherals and I/O Space Accesses		Section 2.3.7, page 2-18
❑ Rewrite code that accesses peripheral and EMIF registers.	REMARK	
❑ Replace code that accesses the C54x I/O space.	REMARK	
C-Callable Assembly		Section 2.3.8, page 2-20
❑ Use the C54x_CALL or C54x_FAR_CALL pragma for C54x C-callable assembly code.	Does not detect	
Special Case		Section 2.3.9, page 2-21
❑ Inspect instructions that use the same AR for Xmem and Ymem operands and that modify the AR in the Ymem operand.	REMARK	

† MASM55 generates REMARK messages to flag most places in C54x code that require changes to make your code functional.

Table 2–4. Step 2: Porting System-Level Code (Continued)

Category	MASM55 Behavior†	See ...
Linker Command File Changes		Section 2.3.10, page 2-22
❑ Allocate the data stack and the system stack in same 64K-word page	–	
❑ Allocate all data sections in memory page 0 (the first 64K words of memory)	–	
❑ Allocate function code being called by CALA in memory page 0 (the first 64K words of memory)	–	
❑ Allocate code reached by using BACC in the same 64K-word page as the calling code	–	
❑ Modify the interrupt vector table alignment requirement	–	

† MASM55 generates REMARK messages to flag most places in C54x code that require changes to make your code functional.

Example 2–2. C54x Software Counter Code (Original)

```
;...
start:
   STM    #TOS, SP              ; Initialize stack pointer
   STM    #0FFFFh, IFR          ; Clear all pending interrupts
   SSBX   INTM                  ; Disable interrupts
   RSBX   SXM                   ; Turn sign extension mode off
   LD     #RSV, A               ; Load address of interrupt vector table
                                ;   into A
;...
   AND    #0FF80h, A            ; Clear lower 7 bits of address
                                ;   (only upper 9 bits are needed for IPTR)
;...
   OR     #0020h, A             ; Set overlay bit
   STLM   A, PMST               ; Load PMST with result in A
   LD     #lowcnt, DP           ; Load DP to point to lowcnt page
   STM    #0010h, IMR           ; Unmask SINT4 interrupt
   RSBX   INTM                  ; Enable interrupts

count_loop:
   LD     lowcnt, A             ; Load lowcnt into A
   ADD    #1, A                 ; Add 1 to A
   STL    A, lowcnt             ; Store A to lowcnt
   SUB    #0500h, A             ; Subtract 0x0500 from A
   BC     count_loop, ANEQ      ; If (A != 0) goto count_loop
   ST     #0, lowcnt            ;  else, reset lowcnt
   INTR   20                    ; Trigger SINT4
   B      count_loop            ; Goto count_loop
;...
```

2.3.1 Add Code to Initialize the System Stack

Background

- Each C54x DSP has a single stack that is referenced by the 16-bit SP register. Each C55x DSP has an additional stack (the system stack) that is referenced by the 16-bit SSP register.
- In a C55x DSP, SP holds the 16 least significant bits (LSBs) of the 23-bit extended data stack pointer (XSP), and SSP holds the 16 LSBs of the 23-bit extended system stack pointer (XSSP). A single register, SPH, provides the 7 most significant bits (MSBs) of XSP and the 7 MSBs of XSSP. For that reason, *both stacks must be located in the same 64K-word mepage of memory*.

Example: XSSP Initialization in the Software Counter Code

It is important to initialize the system stack pointer in the software counter code (Example 2–2 on page 2-8) because the code calls an interrupt service routine (ISR) and thus will need to use the stacks to store and retreive data. The linker command file in Example 2–6 on page 2-14 shows how a section is allocated for the system stack on the same 64K-word page as the data stack. XSSP must be initialized to hold the address of this section. The modification for Example 2–2 is shown in the code below. Note that the write to XSSP loads a memory page into SPH, which is used for both stacks. Both stack memory sections can be placed on any page of data memory.

Before (C54x)	After manual modification (C55x)
`STM  #TOS, SP`	`STM  #TOS, SP` `AMOV  #TOSS, mmap(XSSP)`

Note: TOS = Top of (data) stack; TOSS = Top of system stack

2.3.2 Rewrite the Interrupt Vector Table (IVT)

Background

- **Interrupt vector locations:** Both C54x and C55x devices have 32 interrupt vector locations, each consisting of eight bytes. In C54x DSPs the interrupt vector location includes a branch instruction to lead to the ISR. In C55x DSPs, the interrupt vector location includes the ISR address, and the CPU uses that address to branch to the ISR.

A C55x DSP ignores the first byte of each interrupt vector location except the reset vector location, whose first byte defines the stack mode. Use the .ivec directive supported by MASM55 to select the appropriate value for that first byte and to insert the 3-byte ISR address in the second, third, and fourth byte locations. If no instruction follows this, MASM55 inserts NOPs to fill the remaining bytes. However, you can insert one instruction within the fifth through eighth byte locations. Any second instruction in bytes 5 through 8 is ignored.

- **Rearranging interrupt vector locations:** The interrupt vector offsets within the C55x interrupt vector table are different from those in a C54x DSP. For example, the INT0 vector is at word-offset 40h (byte-offset 0x80) in a C54x DSP but is at byte-offset 10h in a C55x DSP.

  Also, a C55x DSP dedicates one interrupt vector to each peripheral event. There is no multiplexing of peripheral interrupt vectors like there is in C54x DSPs.

- **Relative priorities of interrupts:** There is a difference between C54x and C55x interrupt priorities in some cases. For example in a C54x DSP, INT3 has a higher priority than the interrupt for McBSP 1. In a C55x DSP, the opposite is true. If ignored, the priority differences could have an unexpected impact on real-time systems.

Example: Interrupt Vector Table for the Software Counter Code

Example 2–3 and Example 2–4 show the C54x and C55x interrupt vector tables, respectively, for the software counter code (Example 2–2 on page 2-8). Note the use of the *.ivec* directive in the C55x version. Also, the size of the *.space* allocation and the parameters for *.loop* are different because of differences in the mapping of software interrupts within the vector table. The C54x interrupt vector for the RINT0/SINT4 ISR is at byte-offset A0h, but the corresponding VC5510 vector (for RINT0/SINT5) is at byte-offset 28h.

Example 2–3. C54x Interrupt Vector Table

```
;...
       .sect ”vectors”
;...
RSV:   BD  start      ; Reset
       NOP
       NOP
;...
; Method 1: using .space
    .space  16*4*19    ; 16 bits/word * 4 word/interrupt vector * 19 vector spaces

; Method 2: Using .loop and a dummy isr
;   .loop 19      ; 19 ”empty” vectors
;      bd  dummy_isr
;      nop
;      nop
;   .endloop
;
; where dummy_isr could be defined as
;                .text
;  dummy_isr     b   dummy_isr
; Note:   Use this dummy isr to detect and trap in code errors produced by
;         an unexpected interrupt vector fetch.

SINT4  BD  SWI_4_isr  ; Software Interrupt #4
       NOP
       NOP
;...
```

Example 2–4. C55x Interrupt Vector Table (After Manual Modification)

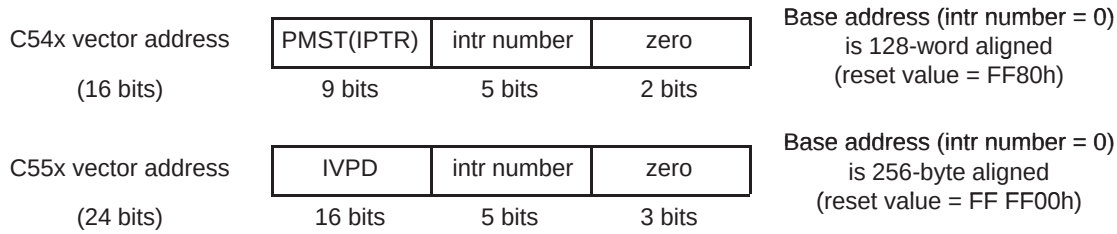
```
;...
       .sect ”vectors”
;...

RSV:    .ivec   start       ; Reset
;...
; Method 1: using .space
    .space 8*8*4           ; 8 bits/byte * 8 bytes/interrupt vector * 4 vector spaces

; Method 2: Using .loop and a dummy isr
;   .loop 4                ; 4 ”empty” vectors
;      .ivec  dummy_isr
;   .endloop
;
; where dummy_isr could be defined as
;                 .text
;  dummy_isr     b   dummy_isr
; Note:   Use this dummy isr to detect and trap in code errors produced by
;         an unexpected interrupt vector fetch.

SINT5   .ivec   SWI_5_isr   ; Software Interrupt #5
;...
```

2.3.3 Rewrite Code That Initializes the Interrupt Vector Pointer (IPTR) in the PMST Register

Background

- The location of the interrupt vector table (IVT) after reset is word address FF80h in a C54x DSP and byte address FF FF00h in a C55x DSP. As shown in Figure 2–1, relocation of the IVT from its reset location is achieved by changing the value of the 9-bit PMST(IPTR) field in a C54x DSP and by changing the 16-bit IVPD (interrupt vector pointer-DSP) register in a C55x DSP. The “intr number” mentioned in the figure is the 5-bit binary number for the sequential position of the vector in the IVT. For example, the first vector in the IVT, the reset vector, is vector 0. Thus, for the reset vector, intr number = 0000b.

Figure 2–1. Formation of an Interrupt Vector Address



- The IVT alignment requirement stays the same: 128 16-bit words in a C54x DSP and the equivalent, 256 bytes, in a C55x DSP. However, because the C55x .align directive is given in bytes, we need to change the number used for the directive.
- A C55x device has separate host and DSP interrupt vector pointers (IVPH and IVPD) for the host and DSP interrupt vector tables, respectively, and they both need to be initialized if they are used. Special treatment is required when decoding with HPI-related interrupts.

Example: Initializing an Interrupt Vector Pointer in C55x Code

The C54x code that initializes PMST(IPTR) must be modified for C55x DSPs as shown below. Notice that in both the C54x case and the C55x case, a label (RSV) has been used for the reset vector address to avoid a hard-coded address. In the C55x code, the 24-bit address is shifted right by 8 bits so that the 16 MSBs are isolated and stored in IVPD (look back to Figure 2–1).

Before (C54x)	**After (C55x)**
`STM  #(RSV & 0FF80h | 20h), PMST`	`MOV  #(RSV >> 8), mmap(IVPD)`

Compare the “vectors” section declarations in Example 2–5 (C54x code) and Example 2–6 (C55x code). Because C55x code is byte addressed, the interrupt vector table should be aligned on a 256-byte page (the 8 LSBs of the base address should be 0s). The linker command file in Example 2–6 reflects this fact by using the *.align 256* directive rather than the *.align 128* directive shown in Example 2–5.

Example 2–5. C54x Linker Command File

```
/*************************************************************************/
/* C54x Linker Command File                                              */
/*************************************************************************/
;...
SECTIONS
{
    .text     :> PRAM    PAGE 0
    vectors   :> VECS    PAGE 0, align 128     /* align given in words */
    .bss      :> SPRAM   PAGE 1
    .data     :> DARAM   PAGE 1
    stack     :> DARAM   PAGE 1
}
;...
```

Example 2–6. C55x Linker Command File

```
/*************************************************************************/
/* C55x Linker Command File                                              */
/*************************************************************************/
;...
SECTIONS
{
    .text     :> PRAM
    vectors   :> VECS, align 256     /* align now given in bytes */
    .bss      :> SPRAM
    .data     :> DARAM
    stack     :> DARAM
    sysstack  :> DARAM
}
;...
```

2.3.4 Modify Code That Initializes the IMR and IFR Registers

Background

A C54x DSP has two registers to flag and mask interrupts (IFR and IMR). To avoid the interrupt multiplexing done for peripherals in C54x DSPs, the C55x architecture expands the number of registers to four: two for flag bits (IFR0 and IFR1) and two for mask/enable bits (IER0 and IER1). In addition, relative bit locations for interrupts in the flag registers and mask/enable registers are different in C54x and C55x DSPs. Due to these differences, C54x code that initializes the interrupt registers must be rewritten.

Example: Initializing Interrupt Registers in the Software Counter Code

In the software counter code (Example 2–2 on page 2-8), all interrupt flags are cleared and only the interrupt SINT4 is left unmasked. The code below shows the modifications required for porting to a C55x device. The flags in both flag registers, IFR0 and IFR1, are cleared. The interrupt to be unmasked (corresponding to SINT4) is in IER0. Note that in Example 2–2, there was no need to unmask the interrupt because the IER0 bits do not affect the INTR instruction; it was done to show the changes required for a C55x DSP.

Note also that the value to be written to the IERs depends on how the interrupts are assigned to the bits in the IERs. Writing 0010h to IER0, as in the example, might not have the same effect as writing 0010h to IMR.

Before (C54x)	After Manual Modification (C55x)
`STM #0FFFFh, IFR`	`MOV #0FFFFh, mmap(@IFR0)`
	`MOV #0FFFFh, mmap(@IFR1)`
`STM #0010h, IMR`	`MOV #0010h, mmap(@IER0)`
	`MOV #0000h, mmap(@IER1)`

2.3.5 Modify the Operands of the TRAP and INTR Instructions

C54x and C55x DSPs offer the same TRAP and INTR insruction mechanisms. However, due to the differences in the relative positions of the hardware interrupt vectors, you may need to modify the “K” operand value to avoid clashes between hardware and software interrupts (hardware and software interrupt tables overlap in both C54x and C55x devices). In the case of the software counter code (Example 2–2 on page 2-8), INTR 20 must be changed to INTR 5, as they both share the vector with RINT0.

Note:

MASM55 does not warn you that INTR/TRAP values may need to change. Manual inspection and code change is required.

2.3.6 Preserve MASM55 Temporary Register Values During Interrupt Service Routines

Background

- A ported C54x interrupt service routine (ISR) needs to preserve on entry the following C55x registers, if they are used as temporary registers by MASM55 during the ISR assembly process. MASM55 uses temporary registers when one C54x instruction ports to more than one C55x instruction.

  T1
  AC2
  AC3
  XCDP
  CSR
  ST0_55 (TC1 bit only)
  ST2_55

  You must manually add code to your C54x ISR to preserve in the stack the C55x temporary registers used. Version 1.5 and later versions of MASM55 identify the temporary registers used by listing them at the top of the listing file. For example the listing file may have the following statement, indicating that you must preserve T1, AC2, AC3:

  ```
  ; Temporary Registers Used: AC2, AC3, T1.
  ```

  In addition, the specific line of code that uses temporary registers is marked by ***!REG!***. If looking in the listing files is troublesome, simply add the TEMP_SAVE and TEMP_RESTORE macros shown in Figure 2–2 at the beginning and at the end of the C54x ISR, to save and restore all of the possible temporary registers.

Figure 2–2. TEMP_SAVE and TEMP_RESTORE Macros

```
TEMP_SAVE     .macro
              PSH  mmap(@T1)
              PSH  dbl(AC2)
              PSH  dbl(AC3)
              PSH  mmap(@AC2G)
              PSH  mmap(@AC3G)
              PSH  mmap(@CDP)
              PSH  mmap(@CDPH)
              PSH  mmap(@CSR)
              PSH  mmap(@ST0_55)
              PSH  mmap(@ST2_55)
              .endm
```

```
TEMP_RESTORE  .macro
              POP  map(@ST2_55)
              POP  mmap(@ST0_55)
              POP  mmap(@CSR)
              POP  mmap(@CDPH)
              POP  mmap(@CDP)
              POP  mmap (@AC3G)
              POP  mmap(@AC2G)
              POP  dbl(AC3)
              POP  dbl(AC2)
              POP  mmap(@T1)
              .endm
```

- In applications containing a mix of C54x and C55x instructions, if a C55x-native ISR uses MASM55 temporary registers, then the ISR must preserve on entry those registers. This would normally be done by the ISR, anyway, except in cases where one or more of these registers has been reserved for exclusive use by the ISR (not used in any other part of your code).

Example: XCDP Used as a Temporary Register by MASM55

In the software counter code (Example 2–2 on page 2-8), an INTR 20 instruction forces a branch to an ISR. The code for the ISR is shown in Example 2–7. When MASM55 assembles this ISR, there is one case of a multi-instruction porting that uses the XCDP register. An inspection of the ISR listing file reveals that XCDP is used as a temporary register in the porting of an MVDK instruction (XCDP is used because C55x DSPs do not support the C54x dmad type of addressing):

Before (C54x)	**After MASM55 – Listing File**
`MVDK *(highcnt), #globcnt`	`AMOV #globcnt, XCDP` `MOV *(highcnt), *CDP`

The original XCDP value should be saved before XCDP is used in the ISR, and the original XCDP value should be restored before the CPU returns from the ISR. In preparation for another pass through MASM55, the ISR can be modified as shown in Example 2–8:

- ❏ Before the MVDK instruction, the modified ISR saves XCDP, by pushing its high part (CDPH) and its low part (CDP) onto the stack.
- ❏ Before ending, the modified ISR restores XCDP, by popping the two parts of XCDP off the stack.

Example 2–7. Original C54x ISR

```
SWI_4_isr:
;...
          LD     highcnt, A                ; load highcnt into A
          ADD    #1, A                     ; add 1 to A
          STL    A, highcnt                ; store A to highcnt
          MVDK   highcnt, #globcnt         ; copy highcnt to globcnt
;...
          RETE                             ; return with interrupts enabled
```

Example 2–8. ISR of Example 2–7 (After Manual Modification)

```
SWI_5_isr:
;...
          PSH    mmap(CDPH)                ; save CDPH (upper 7 bits)
          PSH    mmap(CDP)                 ; save CDP (lower 16 bits)
          LD     highcnt, A                ; load highcnt into A
          ADD    #1, A                     ; add 1 to A
          STL    A, highcnt                ; store A to highcnt
          MVDK   highcnt, #globcnt         ; copy highcnt to globcnt
;...
          POP    mmap(CDP)                 ; restore CDP (lower 16 bits)
          POP    mmap(CDPH)                ; restore CDPH (upper 7 bits)

          RETE                             ; return with interrupts enabled
```

2.3.7 Porting of C54x Peripheral and I/O Code to C55x DSPs

Background

C54x code that initializes or controls peripherals cannot be ported correctly to a C55x device by MASM55.The same is true for C54x code that accesses I/O space. You must re-write the code because:

- ❏ The peripheral registers in a C55x device are located in the I/O space, while in a C54x device, they are located in the data space.
- ❏ Some C54x DSPs use sub-addressing schemes for peripherals, while in C55x DSPs, each peripheral register has its own address.

- C55x and C54x peripheral features are different.

- The C54x I/O space is dedicated to external peripherals, while the C55x I/O space is dedicated to internal peripherals. A C55x DSP does not offer an external I/O space; it offers only an internal I/O space. C54x PORTR and PORTW instructions are ported by MASM55 to access the new C55x internal I/O space, and they require manual modification. This also means that the hardware design must be modified to use, for example, C55x data memory space instead of C54x I/O space.

Example

Example 2–9 shows generic C54x code that initializes McBSP 0. The equivalent code to initialize McBSP 0 of the TMS320VC5510 (VC5510) DSP is shown in Example 2–10.

Example 2–9. C54x McBSP 0 Initialization

```
;...
SPSA1      .set 0x0038                  ; Address of subbank address register
SPSD1      .set 0x0039                  ; Address of subbank data register
;...
_mcbsp_54_init:
;...
    STM    #SPCR1_0, SPSA1              ; Set McBSP 0 Port Control Register 1 (SPCR1)
    STM    #0010000000000000b, SPSD1

    STM    #SPCR2_0, SPSA1              ; Set McBSP 0 Port Control Register 2 (SPCR2)
    STM    #0000001000000000b, SPSD1

    STM    #RCR1_0, SPSA1               ; Set McBSP 0 Receive Control Register 1 (RCR1)
    STM    #0000000001000000b, SPSD1

    STM    #RCR2_0, SPSA1               ; Set McBSP 0 Receive Control Register 2 (RCR2)
    STM    #0000000001000001b, SPSD1

    STM    #XCR1_0, SPSA1               ; Set McBSP 0 Transmit Control Register 1 (XCR1)
    STM    #0000000001000000b, SPSD1

    STM    #XCR2_0, SPSA1               ; Set McBSP 0 Transmit Control Register 2 (XCR2)
    STM    #0000000001000001b, SPSD1

    STM    #PCR_0, SPSA1                ; Set McBSP 0 Pin Control Register (PCR)
    STM    #0000000000000000b, SPSD1

    STM    #SPCR1_0, SPSA1              ; Enable/Unreset McBSP 0 Receiver
    ORM    #0000000000000001b, *(SPSD1)
```

Example 2–10. C55x McBSP 0 Initialization

```
SPCR1_0   .set   0x0000
SPCR2_0   .set   0x0001
RCR1_0    .set   0x0002
RCR2_0    .set   0x0003
XCR1_0    .set   0x0004
XCR2_0    .set   0x0005
PCR_0     .set   0x000E
;...
_mcbsp_55_init:
;...
    MOV    #0010000000000000b, port(#SPCR1_0); Set McBSP 0 Port Control Register 1

    MOV    #0000001000000000b, port(#SPCR2_0); Set McBSP 0 Port Control Register 2

    MOV    #0000000001000000b, port(#RCR1_0) ; Set McBSP 0 Receive Control Register 1

    MOV    #0000000001000001b, port(#RCR2_0) ; Set McBSP 0 Receive Control Register 2

    MOV    #0000000001000000b, port(#XCR1_0) ; Set McBSP 0 Transmit Control Register 1

    MOV    #0000000001000001b, port(#XCR2_0) ; Set McBSP 0 Transmit Control Register 2

    MOV    #0000000000000000b, port(#PCR_0)  ; Set McBSP 0 Pin Control Register

    OR     #0000000000000001b, port(#SPCR1_0); Enable/Unreset McBSP 0 Receiver
```

2.3.8 Use the C54x_CALL or C54x_FAR_CALL Pragma for C54x C-Callable Assembly Code

Calling conventions for the C54x and C55x compilers are different. Parameters are passed mainly through the stack in C54x compiling and mainly through registers in C55x compiling. If you are porting C54x C-callable assembly code, use the C54x_CALL () pragma (for near calls) or the C54x_FAR_CALL () pragma (for far calls) to identify C54x functions in your C code:

```
#pragma C54X_CALL (function)
            or
#pragma C54X_FAR_CALL (function)
```

The use of one of these pragmas allows the compiler to set up the C54x-compatible mode environment before the call as well as using the C54x calling convention (the C55x C compiler assumes the C55x native mode by default). In this way, no changes are required in the C54x assembly code to adjust for the different parameter passing.

To determine when to use the C54X_CALL () pragma and when to use the C54X_FAR_CALL () pragma, use the following rules:

- Ported C54x assembly code that supports both near and far modes via conditional compilation, using the __far_mode symbol, will default to near mode operation. This is because the C55x code generation tools define the __far_mode symbol to equal 0. The C54X_CALL () pragma should be used in this case.

- Ported C54x assembly code that supports *only* far mode, should use the C54x_FAR_CALL () pragma.

The usage of the C54X_CALL () pragma to call a C54x assembly function is illustrated in the block FIR code shown in Example 3–1 and Example 3–2 (see page 3-3). Notice the use of the __far_mode symbol in Example 3–2.

2.3.9 Special Case: Instructions that use the Same AR for Xmem and Ymem and that Modify the AR in the Ymem operand.

Background

You must inspect the execution of instructions that use the **same** auxiliary register (AR) for Xmem and Ymem operands and also include a post-increment/decrement for the AR in the Ymem operand. In C54x devices, the Ymem post-modification does not occur. In C55x devices, the post-modification does occur.

Example

Syntax: ADD Xmem, Ymem, A
Instruction: ADD *AR3, *AR3+, A

When this instruction is executed in C54x devices, the CPU does not increment AR3. In C55x devices, the CPU *does* increment AR3. MASM55 cannot differentiate between C54x or C55x behavior because the same mnemonic, ADD, is used for both C54x and C55x instruction syntaxes. **MASM55 defaults to C55x behavior and as a result, AR3 is incremented by 1.**

This special case is not a problem for C54x MAC instructions because the equivalent C55x mnemonic is MACM. The different mnemonics allow MASM55 to treat the instructions differently.

WHAT TO DO: Although though this case is rare, MASM55 detects it and flags it with a REMARK. If you want the original C54x behavior, simply remove the + sign from the Ymem operand.

2.3.10 Change the Linker Command File

Background

Changes in the linker command file are required because:

- C54x devices offer a modified Harvard architecture with separate code and data memory spaces. C55x devices offer a unified memory architecture in which code and data share the same address space.
- C54x devices uses 16-bit word addresses and lengths in the linker command file. A C55x linker command file uses byte addresses and byte lengths for both program and data sections. However, notice that the C55x map file lists program addresses in bytes and data addresses in 16-bit words. Also, be aware that a section is categorized as a program section if it contains any instruction.
- The C54x and C55x memory maps are different (see section 2.3.10.1).
- The porting of C54x code imposes special restrictions on where you map data and program sections (see Table 2–5). These restrictions can be removed by changing the C54x code, as covered in section 3.6, Code and *Data Placement Considerations*.
- The C55x linker directives *–heap* and *–stack* specify size in bytes. C54x directives use 16-bit words for size. For the *.sysmem* and *.stack* sections to have the original C54x size, sizes will have to be multiplied by 2. Also, a *.sysstack* section should now be included. Assuming the use of the 32-bit stack mode, you should use the same size for both *.stack* and *.sysstack* sections.

Table 2–5. Linker Section Mapping Restrictions for C54x Ported Code

Restrictions	Explanation
Stack and system stack must be in same 64K-word page	The stack pointers (XSP and XSSP) share the same upper 7 bits (SPH) address space. This is not only a C54x ported code restriction; C55x native code requires the same.
All data sections must be in memory page 0 (the first 64K words of memory)	MASM55 does not initialize the upper 7 bits of the ARn registers because C54x data existed only in the first 64K-word data page.
DP addressing arrays must maintain the same C54x 128-word page boundary	MASM55 does not initialize the upper 7 bits of the DP register because C54x data existed only in the first 64K-word data page.

Table 2–5. Linker Section Mapping Restrictions for C54x Ported Code (Continued)

Restrictions	Explanation
Function code being called by CALA should be in memory page 0 (the first 64K words of memory)	In C54x devices, the CALA instruction only uses the lower 16-bits of the accumulator (src) to make the call. This means that the called function must be within the first 64K words of memory. When ported to a C55x DSP by MASM55, this instruction emulates C54x behavior exactly. However, we run into the constraint of having to limit called functions to the first 64K bytes of memory.
Code reached by using BACC must be in the same page as the calling code	The BACC instruction suffers a slightly different limitation. The ported BACC[D] instruction maintains the value of the XPC (upper 8 bits of the program counter). Thus, the branch is always to 16-bit address within the same 64K memory page where the BACC[D] instruction is being executed.

Example: The Linker Command File Used to Port the Software Counter Code

Example 2–5 (page 2-13) shows the original C54x linker command file for the C54x software counter code (Example 2–2 on page 2-8). Example 2–6 (page 2-14) shows the C55x linker command file that is used to port the C54x software counter code to a C55x DSP. Notice that:

- The *.align 128* directive in the C54x code was changed to the *.align 256* in the C55x code. In the C55x environment, the *.align* specifies bytes, not 16-bit words.
- In the C55x code, the data stack and the system stack are assigned to the same memory section (DARAM). You must also initialize the stack pointers such that both stacks are in the same 64K-word page.

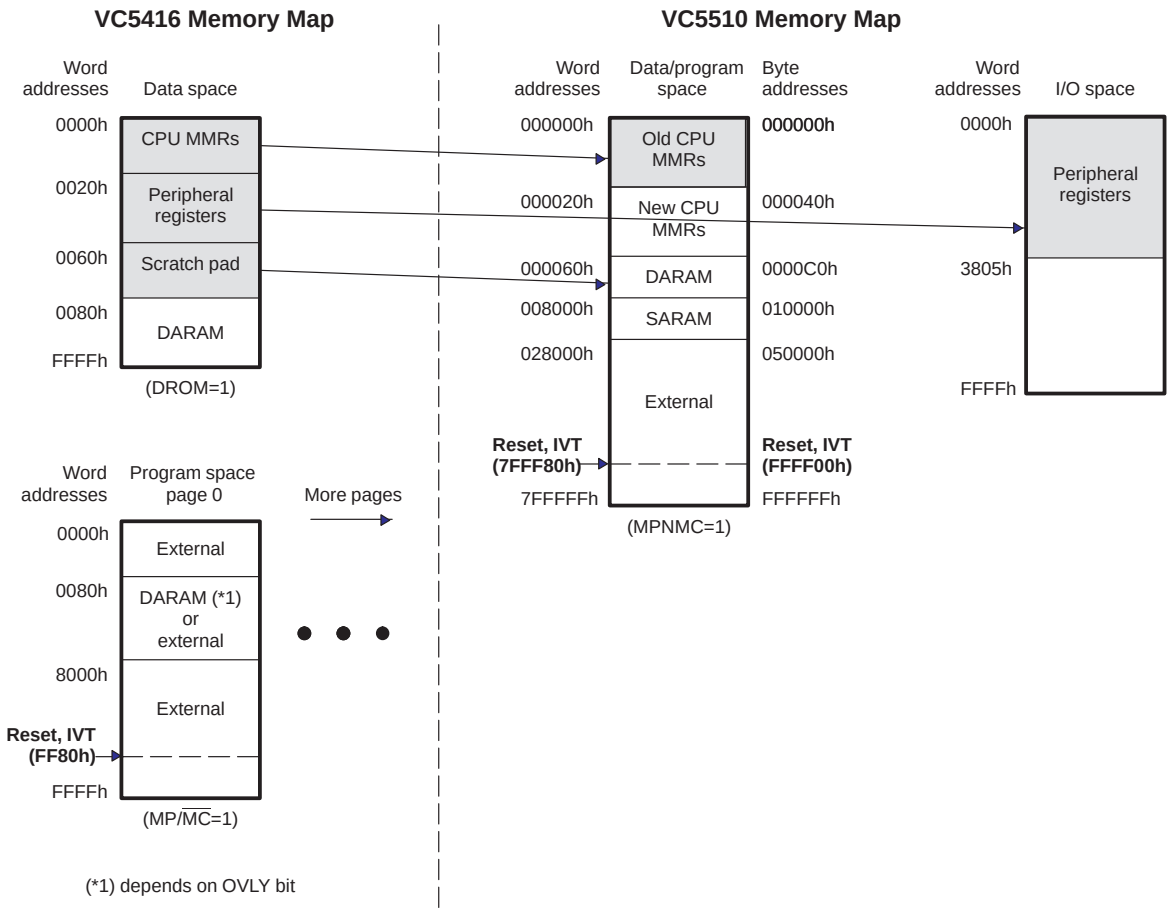
2.3.10.1 Comparison of the C54x and C55x Memory Maps

The C54x and C55x memory maps have differences that have a direct effect on the linker command file. Consider Figure 2–3, which compares the TMS320VC5416 (VC5416) and TMS320VC5510 (VC5510) memory maps when the microprocessor mode is selected (the MPNMC bit is 1: on-chip ROM is not accessible in the memory map). Notice that:

- The location of the CPU memory-mapped registers (MMRs) was kept the same to ease code porting.
- The C54x scratch pad DARAM corresponds to the C55x DARAM area to ease code porting.

- C54x peripheral registers are located in data space. C55x peripheral registers are located in I/O space, and an instruction that accesses them should include the port() instruction qualifier.
- The VC5510 device offers more DARAM and SARAM than most of the C54x devices. C54x block sizes are 8K 16-bit words, while C55x block sizes are smaller (8K bytes), giving you extra parallelism in data accesses. Changes in memory section mapping should not create any problem.
- The default location of the reset vector and the rest of the interrupt vector table (IVT) is different. In C54x DSPs, the default address of the reset vector is the word address FF80h, while in C55x DSPs, the default address is the byte address FF FF00h (word address 7F FF80h). In the microprocessor mode (on-chip ROM disabled), this could imply changes in the location of external flash memory or ROM.

Figure 2–3. Comparison of the VC5416 and VC5510 Memory Maps



Chapter 3

Phase 2: Selected Code Optimization of Medium MIPS Functions (Optional)

Topic **Page**

3.1 Overview

After achieving functional code, the next phase is to optimize selected pieces of code (medium MIPS functions). This optimization phase is optional but can be implemented to:

- ❑ Correct some MASM55 code porting inefficiencies
- ❑ Take advantage of C55x instruction-level parallelism
- ❑ Improve stack memory usage

The possible optimizations are shown in Table 3–1.

Table 3–1. Phase 2: Selected Code Optimization of Medium MIPS Functions (Optional)

Steps	See ...
Step 1: Use MASM55 Selected Optimization Switches ❑ –mh (optimize for speed instead of size) ❑ –mt (save code space and cycles if SST=0 throughout your code) ❑ –mn (remove NOPs in branch and call delay slots) ❑ – –purecirc (optimize porting of circular addressing code)	Page 3-6
Step 2: Use native C55x Instructions and Code Modification	
❑ Replace RPT with RPTBLOCAL if MASM55 translates the repeated C54x instruction into multiple C55x instructions.	Page 3-7
❑ Replace RPTB or RPTBD with RPTBLOCAL when possible.	Page 3-8
❑ Rearrange code to reduce C55x pipeline stalls.	Page 3-9
❑ Replace C54x ASM load instructions with equivalent C55x instructions.	Page 3-9
❑ Delete useless NOPs (used for pipeline latency in C54x DSPs).	Page 3-10
❑ Remove the circular addressing % when BK = 0.	Page 3-10
Step 3: Use C55x Instruction-Level Parallelism with C54x Instructions	Page 3-11
Step 4: Evaluate Whether the 32-Bit Stack Mode is Required	Page 3-13
Step 5: Code and Data Placement Considerations	Page 3-14

This section uses code for a block FIR to illustrate the Phase 2 optimization process. This code is a mix of C and assembly that computes an FIR filter on a block of data stored in memory. The calling C function (after Phase 1 adjustments) and the C54x assembly code are shown in Example 3–1 and Example 3–2, respectively. Notice the use of the C54x_CALL pragma introduced as part of the Phase 1 code porting process. Example 3–3 shows the FIR assembly code after Phase 2 adjustments.

Example 3–1. C54x Calling C Function (After Phase 1 Changes)

```
//...
ushort fir(DATA *x, DATA *h, DATA *r,DATA **d, ushort nh, ushort nx);

#pragma C54X_CALL(fir);
;...
void main()
{
//...
   /* compute FIR */
   fir(x, h, r, &dbptr, NH, NX);
//...
}
```

Note: "ushort" has been previously defined as "unsigned short".

Example 3–2. Original C54x FIR Assembly Function

```
_fir:
;...
       STLM   A, x_ptr
       MVDK   *sp(h), h_ptr
       MVDK   *sp(r), r_ptr
       MVDK   *sp(db), db_ptr
;...
       RPTBD  END_LOOP – 1
       STM    #1, AR0

       MVDD   *x_ptr+, *db_ptr
       MPY    *h_ptr+0%, *db_ptr+0%, A
       RPT    *sp(nc)
       MAC    *h_ptr+0% , *db_ptr+0%, A
       MACR   *h_ptr+0% , *db_ptr, A

       STH    A, *r_ptr+
END_LOOP:


RETURN_FXN:

       LDM    db_ptr, B
       MVDK   *sp(db), db_ptr

       LD     #0, A
       XC     1, AOV
       LD     #1, A
       FRAME  #(FRAME_SZ)

       POPM   ST1
       POPM   ST0

       .if __far_mode                ; __far_mode = 0 in C55x assembler
          FRETD
       .else
          RETD
       .endif
       NOP
       STL    B, *db_ptr
```

Example 3–3. Optimized C54x FIR Assembly Function (After Phase 2 Changes)

```
_fir:
;...
       STLM          A, x_ptr
       ||MVDK        *sp(h), h_ptr          ; use instruction–level parallelism

       MVDK          *sp(r), r_ptr
       MVDK          *sp(db), db_ptr
;...
       STM           #1, AR0                ; replace RPTBD with RPTBLOCAL and
       ||RPTBLOCAL   END_LOOP–1             ;  use instruction–level parallelism

       MVDD          *x_ptr+, *db_ptr
       MPY           *h_ptr+0%, *db_ptr+0%, A
       RPT           *sp(nc)
       MAC           *h_ptr+0% , *db_ptr+0%, A
       MACR          *h_ptr+0% , *db_ptr, A

       STH           A, *r_ptr+
END_LOOP:


RETURN_FXN:

       LDM           db_ptr, B

       MVDK          *sp(db), db_ptr        ; use instruction-level paralellism
       ||LD          #0, A

       XC            1, AOV                 ; use instruction-level paralellism
       ||LD          #1, A

       FRAME         #(FRAME_SZ)

       POPM          ST1
       POPM          ST0

       STL           B, *db_ptr             ; instruction moved from delay slot

       .if __far_mode                       ; __far_mode = 0 in C55x assembler
          FRET                              ; Not delayed
       .else
          RET                               ; Not delayed
       .endif
```

3.2 Step 1: Use MASM55 Selected Optimization Switches

Version 1.5 and later versions of MASM55 implement additional code optimization switches that in certain code-specific situations can improve the performance of your code. Refer to the MASM55 readme.1st file and *TMS320C55x Assembly Language Tools User's Guide* (SPRU280) for a detailed description of these switches.

In this step, you should evaluate if the following switches can be used to benefit the performance of your ported code:

- **–mh** (speed over size): MASM55 by default optimizes for size over speed. When you use the –ms switch, MASM55 uses certain instructions that execute in the address phase of the pipeline instead of the execute phase, giving you the possibility of faster code, but also potentially larger code.
- **–mt** (if PMST(SST)=0): This switch could save code space and cycles if your code always keeps SST=0. SST is the saturate-on-store bit in PMST. When SST=0, the CPU does not perform automatic saturation during accumulator store operations. C54x instructions that are potentially affected by SST include STL, STH, STLM, DST, ST||ADD, ST||LD, ST||MACR[R], ST||MAS[R], ST||MPY and ST||SUB. If you cannot verify that SST is always 0, proceed with caution, verifying correct code functionality after applying the switch.
- **–mn**: This switch tells MASM55 to remove NOP in delay slots.
- **– –purecirc**: The C55x and C54x implementations of circular addressing are different. Use this switch for additional cycle and code size savings **if only** the C54x circular mode is used in the file (that is, no C55x linear/circular mode bits are used).

The optimization of these switches should be explored typically on a file-by-file basis. However, MASM55 offers equivalent MASM55 directives (for example: .sst_on and .sst_off for the case of the –mt switch) that you could embed in your code to achieve a finer degree of granularity within each file. Refer to the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280) for details.

3.3 Step 2: Use C55x Instructions Selectively in C54x Source Code

It is possible to further optimize the performance of your C54x code running on a C55x DSP by selectively using C55x instructions. Mixing of C55x and C54x instruction syntaxes in the same file is allowed. MASM55 understands both C54x and C55x assembly source instruction sets. The following general rules apply:

- Code should run in the C54x-compatible mode (C54CM=1). You can use the C55x instruction set and have full access to new C55x registers and resources, even in the C54x-compatible mode. Specific differences between the C54x-compatible mode and the C55x native mode are covered in section 4.1.
- C55x native instructions should not use MASM55 temporary registers and should not use C55x registers that are used in the C54x code, such as AC0 (C54x A) and T3 (C54x T). Otherwise, corruption of temporary registers could occur when a C54x interrupt service routine (ISR) executes.
- Make sure not to disturb the original behavior of your C54x code.

This step is considered optional because even though you can achieve very quick optimizations through Step 2, the mixing of C54x and C55x instruction syntaxes could create confusion.

In the following subsections, we provide examples of some of the techniques.

3.3.1 Replace RPT with RPTBLOCAL if MASM55 Translates the Repeated C54x Instruction into Multiple C55x Instructions

Background

- When MASM55 encounters a C54x RPT single instruction and the C54x instruction to be repeated ports into multiple C55x instructions, MASM55 replaces the RPT with a conditional branch and decrement. This results in a considerable code size and cycle count increase.
- A C55x DSP provides registers to automatically service two levels of repeat loops, one loop nested inside another in the C55x native mode (C54CM = 0). **Only one level is supported by dedicated registers in the C54x-compatible mode (C54CM = 1).**

Recommendation

If code porting results are not optimal enough, inspect key code kernels for the occurrence of this RPT expansion to multiple instructions. You can do this by manually inspecting the corresponding listing file for a !REG! associated with a RPT instruction porting. When you find cases of RPT expansion, the solution depends on whether C55x native mode or C54x-compatible mode is selected:

- In the C55x native mode (C54CM = 0), you can manually replace the RPT instruction with a C55x RPTBLOCAL instruction if there is an available block repeat counter (that is, if BRC0 or BRC1 is unused). If you are only dealing with C54x ported code, and if no nested C54x RPTB loops are used, then BRC1 should be available.
- In the C54x-compatible mode (C54CM = 1), you can manually replace the RPT instruction with a C55x RPTBLOCAL instruction if this will not create a nested block-repeat loop.

Example: Replacing RPT with RPTBLOCAL to Prevent Code Size Growth

In the following example, the Original C54x Code repeats an instruction that MASM5 translates into multiple C55x instructions. The MASM55 code shows that the RPT instruction has been replaced by a conditional branch and decrement. To fix this, you can replace the RPT instruction with a RPTBLOCAL instruction (as shown in the Modified C54x Code) and then run the code through MASM55 again.

Origial C54x Code

```
    RPT     #4
    LDR     *AR3+, A
```

MASM55 Code (listing file)

```
    MOV #5, T0
P4_4:
    MOV *AR3+ << #16, AC0
    ADD #1 << #15, AC0, AC0
    SUB #1, T0
    BCC P4_4, T0 != #0
```

Modified C54x Code

```
    MOV         #4, BRC1
    RPTBLOCAL   endloop
endloop: LDR *AR3+, A
```

3.3.2 Replace RPTB or RPTBD with RPTBLOCAL When Possible

Background

- MASM55 always replaces a C54x RPTB/RPTBD with a RPTB instruction regardless if the loop contains less than 56 bytes. RPTBLOCAL instructions are not generated. This will be improved in upcoming MASM55 releases.
- RPTBLOCAL can be used in C55x loop code when the loop code is smaller than 56 bytes **and** there is no backward jumping inside the loop.

Recommendation

Replace the RPTB or RPTBD instruction with a RPTBLOCAL instruction in loops that contain less than 56 bytes. The size of the ported C54x loop can be determined by looking at the MASM55 listing file. If you are replacing a RPTBD loop, the instructions in the delay slot need to be placed above the RPTBLOCAL instruction.

3.3.3 Rearrange Code to Reduce C55x Pipeline Stalls

Background

The C54x and C55x execution pipelines are different. The C55x execution pipeline has seven stages (D, AD, AC1, AC2, R, X, W), while the C54x execution pipeline has four stages (D, AC, R, X). As a consequence, pipeline delay conflicts in the two devices will be different.

Recommendation

Single step with the simulator through the ported C55x code to identify potential pipeline stalls. *Remove the stalls by rearranging the instructions carefully.*

As an example, consider one of the most common pipeline conflicts related to the auxiliary registers (AR0–AR7): ARn (or any address generation register) post-modification during indirect addressing takes place in the access (AC) phase in a C54x device and in the address (AD) phase in a C55x device. Consider the following C54x code:

```
STLM   A, AR2
LD     *AR2+, 16, B
LD     #8, A
ADD    *AR5, A
STL    A, 8, *AR7+
STL    A, 7, *AR4–
```

A C55x device introduces four stalls during the execution of the first LD instruction, so that AR2 is loaded by the STLM instruction before being used by the LD instruction. We can remove these stalls by rearranging the instructions:

```
STLM   A, AR2
LD     #8, A
ADD    *AR5, A
STL    A, 8, *AR7+
STL    A, 7, *AR4–
LD     *AR2+, 16, B
```

3.3.4 Replace C54x ASM Load Instructions with Equivalent C55x Instructions

Background

C55x devices use T2 instead of ASM to implement accumulator shifting. To support C54x code, a C55x DSP has special hardware logic that automatically copies the ASM field value to the T2 register.

To make sure that C54x code ports correctly to C55x code, MASM55 ports ASM load instructions (LD #k5/Smem, ASM) to multiple C55x instructions that initialize T2 and ASM to the same value.

Recommendation

Under most conditions, you may be able to avoid those extra C55x instructions by replacing the C54x ASM instruction with a C55x native instruction as shown below.

Before (C54x)	Recommended C55x Replacement
`LD *AR2,ASM`	`MOV *AR2,T2`

Before making this replacement you should check whether a direct modification of the ST1 register is changing the value of ASM. For example, in the following code, applying the instruction replacement above will produce wrong code:

```
                ; Assume initial condition ASM = 0, T2 = 0
LD *AR2,ASM     ; This instruction changes both ASM and T2
                ;   For example to ASM = T2 = 2
                ; If replaced with ″mov *ar2, T2″, only T2
                ;   is initialized (i.e., ASM = 0, T2 = 2)

OR #8000h, ST1  ; This instruction changes BOTH T2 and ASM
                ; to ASM = T2 = 0  (C55x hardware logic)
                ; end-result ==> T2 = 0 (wrong)
```

3.3.5 Delete Useless NOPs

In C54x, NOPs were used as place holders to prevent incorrect functionality due to open pipeline cases. Because C55x offers now a fully protected pipeline, there is no reason to keep those NOPs. Manual removal of NOPs is required.

3.3.6 Remove Circular Addressing Symbol (%) When it is Not Necessary

When the circular addressing symbol (%) is not necessary—that is, when BK = 0—remove it from your code along with the code that initializes the BK register. In C54x code, due to Xmem/Ymem restrictions, the % modifier was used with BK=0 even if there was no need for circular addressing. In C55x code, this is not required.

3.4 Step 3: Use C55x Instruction-Level Parallelism with C54x Instructions

Background

The new C55x instruction-level parallelism can be applied to C54x instruction code before it is ported to the C55x environment. Please refer to the *TMS320C55x DSP Programmer's Guide* (SPRU376) for the rules and guidelines for instruction-level parallelism.

Example: Adding Parallelism to the Block FIR Code

Consider the block FIR code in Example 3–2 (page 3-4) and the parallelism applied in Example 3–3 (page 3-5). One pair of instructions that was placed in parallel is:

```
STLM     A, x_ptr
||MVDK   *sp(h), h_ptr
```

It was possible to place these two C54x instructions in parallel because they each mapped into a single C55x instruction. Another pair of instructions in Example 3–2 are:

```
RPTBD    END_LOOP – 1
STM      #1, AR0
```

Because there are no delayed instructions in the C55 instruction set, MASM55 would create a RPTB instruction and would automatically move the STM instruction ahead of it. In Example 3–3, the STM instruction is moved manually and, in addition, RPTB is changed to RPTBLOCAL to take advantage of instruction-level parallelism:

```
STM          #1, AR0
||RPTBLOCAL  END_LOOP–1
```

This saves one cycle. The change from RPTB to RPTBLOCAL is required in this case because two instructions in parallel cannot be longer than six bytes. With a length of three bytes, RPTB cannot be placed in parallel with the four bytes of the STM instruction. RPTLOCAL is used because it is has only two bytes.

Example 3–3 shows three more optimizations. Two were achieved by placing the following instruction pairs in parallel:

```
MVDK   *sp(db), db_ptr
||LD   #0, A

XC     1, AOV
||LD   #1, A
```

The last change implemented (FRETD and RETD replacement by FRET and RET) was due to the fact that delayed instructions are not supported in the C55x environment. The STL instruction in the delay slot of the return instruction was moved ahead of the return instruction. MASM55 normally moves delay-slot instructions automatically; however, MASM55 will not remove the NOP unless the –mn option is used. In the C54x code, one NOP was used to fill the delay slot. This NOP is now manually removed because it is of no use.

3.5 Step 4: Evaluate Whether the 32-Bit Stack Mode is Required

Background

The C55x 32-bit stack mode is required **only** when you have a C54x C-callable FAR mode assembly function that passes arguments through the stack, or in cases of stack unwinding (typical stack manipulation in a multi-tasking operating system). However, in the 32-bit stack mode, system-stack memory requirements increase because SSP is kept aligned with SP.

Example

In the block FIR code of Example 3–1 through Example 3–3 (pages 3-3 through 3-5), there is no reason to use the 32-bit stack mode because the C55x assembler makes the __far_mode symbol equal to 0 as explained in section 2.3.8 (page 2-20). For this reason, the 16-bit fast-return mode can be selected by specifying USE_RETA in the *.ivec* directive that declares the reset vector:

```
RSV: .ivec _c_int00, USE_RETA
```

This directive places a code in a portion the reset vector location, and during a DSP reset, the CPU reads the code to determine the chosen stack mode. Refer to the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280) for a complete description of *.ivec*.

3.6 Step 5: Code and Data Placement Considerations

Table 2–5 (page 2-22) lists section mapping restrictions for C54x ported code. There are ways to remove some of those restrictions, so that you can take advantage of the larger data addressing range of the C55x DSPs. The main restriction is that MASM55 assumes that all data sections reside in the first 64K words (page 0) of memory. This limits the range of memory in which data sections of migrated code can be placed. The following discussion addresses this issue and offers some workarounds.

3.6.1 Indirect Addressing Considerations

Background

- C54x auxiliary registers are 16 bits wide. Data addressed via indirect addressing had to reside within the first 64K words of memory (page 0).
- C55x auxiliary registers are 23 bits wide. Data addressed via indirect addressing can be located in any 64K-word page but must reside entirely within the page boundary. A C55x DSP has a 23-bit address space that is segmented into 64K-word pages. Auxiliary registers incremented beyond the current page will wrap around.
- MASM55 does not initialize the upper 7 bits of the 23-bit extended auxiliary registers (XAR0–XAR7). MASM55 assumes data resides in memory page 0, and for this reason, all auxiliary register loads are 16-bit loads as opposed to 23-bit loads. This means that indirect accesses that use the auxiliary registers are limited to page 0.

Recommendation

It is possible to move data sections outside of the first 64K words of memory by changing the C54x 16-bit load instructions to native C55x instructions that load an effective 23-bit address. This is illustrated in Case 1 below. The C54x Code loads 16 bits into AR1, leaving the 7 MSBs of XAR1 unaffected. The Recommended C55x Code loads all 23 bits of XAR1.

Case 1. Simple initialization of ARn

C54x Code	Modified C54x Code
`STM  #myVar, AR1`	`AMOV  #myVar, XAR1`

It is important to note that these changes need to be applied for all loads of auxiliary registers. Once this is done, data located in any section beyond the first 64K will be correctly accessed via indirect addressing.

Case 1 is a simple case in which only a direct load with a data label is required. The assembler/linker will simply resolve the now 23-bit value of myVar at link time, and the result will be correct.

However, when the initial ARn value is computed at run time, the computation now must involve 23-bit math operations. In this case, achieving 23-bit XARn initialization is not the straight-forward instruction-replacement process shown in Case 1. The following cases involve doing auxiliary-register computations at run time.

In Case 2, imm is now a 23-bit base of an array and @var is a 16-bit offset taken from memory. Because a C55x device does not have a “MOV #k23, A” instruction to replace “LD #imm, A”, one solution is to take advantage of the C55x 16-bit A-unit ALU as shown.

Case 2. ARn computation via accumulator

C54x Code

```
LD     #imm, A   ; imm = potential 23-bit value
ADD    @var, A   ; @var provides a 16-bit value from memory
STLM   A, AR2
ADD    *AR2, B
```

One Solution

```
AMOV   #imm, XAR2    ; load 23-bit value
ADD    @var, AR2     ; add only in the lower 16-bit
```

Now consider Case 3 below. In a C54x DSP, base_of_an_array could be a 16-bit constant value. Unfortunately, you *cannot* relocate the base of a C54x array beyond memory page 0 (the first 64K words of memory) because the C55x DSPs have no addressing modes that allow the base of the array to be 23 bits wide.

Case 3. A case with no solution

```
    .global   base_of_an_array

    LD        *AR3(base_of_an_array), A
```

In summary:

- ❑ It is possible to relocate data accessed via indirect addressing beyond the first 64K-word page, but the required code changes should be analyzed on a case-by-case basis. Special considerations need to be observed when the initial ARn value is computed at run time.
- ❑ Data can be relocated to any page, but the entire data array must fit within a 64K-word page boundary. The C55x architecture offers 23-bit addresses for a data space that is segmented into 64K-word pages.

3.6.2 DP Direct Addressing Considerations

Background

- ❑ The C54x DP (data page register) is a 9-bit field in status register ST1. An array accessed via direct addressing must start on a 7-bit (128-word) memory boundary (in other words, the lower 7 bits of the base address must be 0s). Each data page that is referenced by DP contains 128 16-bit words.
- ❑ The C55x XDP (extended data page register) is a 23-bit register. An array accessed via direct addressing can start in any memory location; the 128-word boundary is not required. Each data page that is referenced by DP contains 128 16-bit words (the dma addressing field still has 7 bits).
- ❑ MASM55 does not initialize the upper 7-bits of the XDP register. MASM55 assumes that data resides in memory page 0. In addition, MASM55 assumes that the DP data sections in C54x ported code keep their 128-word alignment. For this reason, no .dp directive is added by MASM55 on a DP load. (The .dp directive tells the assembler what value the DP register has at that point in the code, so that the correct *dma* offset can be encoded in the instruction.)

Recommendation to Remove the Page 0 Requirement

It is possible to remove the page 0 requirement for C54x ported code by replacing the typical C54x 9-bit DP load instruction (LD #var, DP) instruction with a C55x native 23-bit XDP load instruction:

C54x Code	Recommended C55x Code
`LD  #var1, DP`	`AMOV  #var1, XDP`

You need to make this change to every DP load instruction in your code. The 128-word alignment requirement is maintained (no .dp directive is used).

Recommendation to Remove the 128-Word Alignment Requirement

The changes presented above remove the page 0 requirement but not the 128-word page alignment requirement. Removal of the 128-word page alignment is more difficult, as it implies:

- ❑ Adding the .dp directive next to every load DP instruction (See Case 1 below)
- ❑ Adding the .dp directive at the top of every file that uses the same DP reference value (See Case 2 below)
- ❑ Making sure that the variable used to load DP references the first variable in the 128-word DP page (See Case 3 and Case 4 and the paragraph following each)

Case 1. Removal of 128-Word and Page 0 Requirements (Simplest Case)

C54x Code

```
.global base

LD  #base, DP
```

Modified Code

```
.global    base
.dp        base    ; removes 128-word requirement
AMOV #base, XDP    ; removes Page 0 requirement
```

Case 2. Handling of Same DP Reference in Multiple Files

C54x Code

file 1

```
    .global   base, func_in_file_2

    LD        #base, DP
    CALL      func_in_file_2
```

file 2

```
    .global   base, func_in_file_2

func_in_file_2:
    ADD       @symbol, A
```

Modified Code

file 1

```
    .global   base, func_in_file_2
    .dp       base
    LD        #base, DP
    CALL      func_in_file_2
```

file 2

```
    .global   base, func_in_file_2
    .dp       base
func_in_file_2:
    ADD       @symbol, A
```

Case 3. Initialization of the DP Using a Variable That Does Not Point to the Array Base

C54x Code

```
   .sect  "my_array"
var1      .word  1
var2      .word  2
var3      .word  3
var4      .word  4

;...


   LD     #var1, DP
   LD     @var4, A
;...

   LD     #var3, DP
   LD     @var2, A
```

Modified Code

```
   .sect  "my_array"
var1      .word  1
var2      .word  2
var3      .word  3
var4      .word  4

;...

   .dp    var1
   AMOV   #var1, XDP
   LD     @var4, A
;...
   .dp    var1
   AMOV   #var1, XDP
   LD     @var2, A
   ; AMOV #var3, XDP would make
   ;  @var2 fail.
```

In the modified code of Case 3, DP must be initialized with var1, so that DP points to the base (first word) of the array. Otherwise, there is the risk of producing invalid offsets. The access made with @var2 will fail if AMOV #var3, XDP is used instead of AMOV #var1, XDP. The @var2 works in the C54x code because of the 128-word alignment requirement. In a C54x device, the DP will get initialized to the correct value regardless of which var_x we use in LD #var_x, DP.

Case 4. Initialization of the DP Using a Variable That Does Not Point to the Array Base

C54x Code

```
   .bss   local, 5
;...
   .bss   temp, 4
;...

   LD     #temp, DP
   ADD    @local, B
```

Modified Code

```
   .bss   local, 5
;...
   .bss   temp, 4
;...
   .dp    local
   AMOV   #local, XDP
   ADD    @local, B
```

The code in Case 4 works in a C54x device if the address labeled “temp” is no greater than 127 words from the address labeled “local” and the entire .bss section is contained within an aligned DP page. As in Case 3, DP must initialized with a value that points the DP to the top of the reference memory section. Therefore, #temp must be replaced with #local to avoid negative offsets. In addition, the .dp directive must be used.

3.6.3 SP (Stack) Direct Addressing

Stack memory (*.stack* or *.sysstack*) or any data section being accessed via stack direct addressing can be located anywhere in the 23-bit C55x data space memory range as long as the changes recommended in section 2.3.1 (page 2-9) are implemented. Remember that *.stack* and *.systack* must be in the same 64K-word page of memory because they share the upper 7 bits.

3.6.4 Dmad, Pmad, and *(lk) Addressing Considerations

These types of memory addressing are ported by MASM55 to support the full data memory reach of the C55x device, and no extra work is required to relocate data or program sections.

3.6.5 Indirect Call/ Branch Considerations

If you want to create code sections that are fully portable across pages of memory, you must manually replace the following two instructions:

```
CALA[D]  src
BACC[D]  src
```

Replacing the CALA[D] Instruction

In C54x devices, the CALA instruction only uses the lower 16-bits if the accumulator (src) to make the call. This means that the called function must be within the first 64K words of memory. When ported to a C55x DSP by MASM55, this instruction emulates the C54x behavior, but because C55x program address are byte addresses, the limitation on a C55x DSP is the first 64K bytes. To overcome this limitation, it is recommended that all CALA instructions be replaced with the CALL ACx instruction of the C55x instruction set. This ensures that the full 24-bit address in the ACx accumulator is used.

Important: Bits 23–16 of ACx must contain a valid value; otherwise, the call will be incorrect. This means that if the original C54x code relied on the fact that the bits in the high part of the accumulator were ignored, it is necessary to ensure that those bits are correctly set to 0. The following example shows the use of CALL ACx.

C54x Code	MASM55 (listing file)	Recommended C55x Code

`CALA  A`	`MOV   AC0, AC2`
	`AND   #65535, AC2, AC2`
	`CALL  AC2`
		`CALL  AC0`

In the case of CALAD (delayed), you must also move the instructions that are in the delay slot to a new position preceding the CALL AC0 instruction.

Replacing the BACC[D] Instruction

In a C55x DSP, the ported BACC[D] instruction maintains the value of the XPC (upper 8 bits of the program counter). Thus, the branch is always to a 16-bit address within the same 64K-byte memory page where the BACC[D] instruction is being executed.

To overcome this limitation, it is recommended that all BACC instructions be replaced with the B ACx instruction of the C55x instruction set. **Important:** Bits 23–16 of ACx must have correct address values or the branch will be incorrect.

In the case of BACCD (delayed), you must also move the instructions that are in the delay slot to a new position preceding the B ACx instruction.

Chapter 4

Phase 3: Code Optimization of High MIPS Functions via C55x Native Implementation (Optional)

Topic **Page**

4.1 Overview

In order to take advantage of all C55x architectural features, *MIPS intensive functions of the algorithm should use* C55x native instructions and **if required**, should run under C55x native mode (C54CM=0).

As explained in section 1.2 *The C54x-to-C55x Code Porting Process*, functions that consume 80% of MIPS of the algorithm and that could take advantage of C55x new features are good candidates for native coding. These functions are typically *leaf functions*. Leaf functions are defined as functions that call no other functions (like the end-terminals or leaves in a tree). It is recommended that you start with leaf functions because they are the inner functions with a higher potential to impact your cycle benchmarks.

This section shows how you could take advantage of some of the new C55x architectural features listed in Table 4–2. Because native code optimization is to be implemented in a function-by-function basis, C54x native functions must coexist with C55x native functions. Special C54x/C55x context switching between functions might be required. This is the focus of section 4.2.

It's important to note that **using C55x native instructions does not necessarily implies that you have to set C54CM=0 (C55x native mode)**. All C55x architectural features (new registers, new instructions,…) are also available under C54x compatibility mode (C54CM=1) **with the differences noted in Table 4–1.** However, to make the code porting process more orthogonal, we advise, in this Phase of the C54x-to-C55x code porting process, to switch to C55x native mode (C54CM=0) as part of the function context switching.

Table 4–1. Differences Between C54x Compatibility Mode and C55x Native Mode

Feature	Behavior Under C54CM=0 (C55x native)	Behavior Under C54CM=1 (C54x compatibility)
Nested block repeats	supported	not supported
Circular addressing	BKO3 used	BK used
Register indexing	T0 used	AR0 used

Table 4–2. New C55x Architectural Features to Use in C55x Native Coding

New C55x Features	Reference
❑ Dual-mac capability	See section 4.3, page 4-7.
❑ New circular addressing features	See section 4.4, page 4-9.
❑ Optimal loop implementations	See section 4.5, page 4-14.
❑ A-unit ALU	See section 4.6, page 4-17.
❑ Instruction-level parallelism	The new user-defined instruction parallelism is covered in detail in the *TMS20C55x DSP Programmer's Guide* (SPRU376).
❑ Additional accumulators and T registers	See section 4.7, page 4-18.
❑ Improved dual read/writes	See section 4.8, page4-18.
❑ Less restrictive xmem/ymem addressing	See section 4.9, page 4-19.
❑ Additional TC bits	See section 4.10, page 4-19.

4.2 Safe C54x/C55x Context Swapping

When switching between the C54x native codes (ported using masm55) and the C55x native code, you must make sure there is a context swap (a save, switch, and restore sequence).

We recommend that the switching to C55x native code should happen in a function-by-function basis.The context-swap code can reside in either the calling function or the called function:

- If context swapping is done in the calling function (the way the C55x compiler does it), the original called function can be used. The drawback is potentially larger code size because the context-swap code has to be duplicated in every calling function.
- If context swapping is done in the called function, changes in the C54x calling function (like inserting C55x native instructions) are avoided. In addition, overall code size is potentially reduced because this approach prevents the duplication of context-swap code in each calling function.

The examples in sections 4.2.1 and 4.2.2 implement context swapping in the called function.

4.2.1 Calling a C55x Native Function from C54x Code

Example 4–1 shows how C54x assembly code can call a C55x-native assembly function. If you follow the suggestion to convert to C55x native code in the C54x leaf functions first (page 4-2), it is expected that a C54x assembly function calling a C55x assembly function will be the most common operation. As shown in Example 4–1, the overhead in context switching depends on the C55x function code itself. Example 4–2 shows the C55x_ENTRY and C55x_EXIT macros that are used in Example 4–1.

Example 4–1. Calling a C55x Native Function from C54x Code

```
c54x_code:
;...
   CALL c55x_func
;...
```

```
c55x_func:
       ; Code to adjust parameter passing (if required)
       ; and save to the stack the C54x registers that
       ; need to survive the call to c55x_func.
       C55X_ENTRY
       ; Your code. You can use/set any C55x register.
       ; This includes C55x registers used as MASM55
       ; temporary registers because they do not need
       ; to survive across calls.
       C55X_EXIT
       ; If applicable, code restores C54x registers
       ; from the stack.
       RET
```

Example 4–2. Macros to Use When Calling a C55x Native Function from C54x Code

```
C55x_ENTRY .macro
       BCLR C54CM
       ; See Note 3
       .endm
```

```
C55x_EXIT    .macro
       BSET   C54CM                  ; See Note 3
       BCLR   ARMS                   ; See Note 1
       BCLR   M40                    ; See Note 1
       BCLR   RDM                    ; See Note 1
       AND    #00FF00h, *(ST2_55)    ; See Note 1
       MOV    #0, mmap (@DPH)        ; See Note 2
       MOV    #0, mmap (@CDPH)       ; See Note 2
       MOV    #0, mmap (@BSA01)      ; See Note 2
       MOV    #0, mmap (@BSA23)      ; See Note 2
       MOV    #0, mmap (@BSA45)      ; See Note 2
       MOV    #0, mmap (@BSA67)      ; See Note 2
       .endm
```

Notes: 1) This instruction is needed only if ARMS/M40/RDM/ARnLC is set to 1 inside the c55x native function.

2) This instruction is needed only if DPH/CDPH/BSAxx is set to a nonzero value inside the C55x native function.

3) This instruction is needed only if nested loops or C55x circular addresing is used inside the C55x native functions.

4.2.2 Calling a C54x Routine from C55x Native Code

Example 4–3 shows how C55x assembly native code can also call C54x ported code. As shown below the overhead in context switching depends on the code of the C55x function itself. Example 4–4 shows the C54x_ENTRY and C54x_EXIT macros used in Example 4–3.

Example 4–3. Calling a Ported C54x Function from Native C55x Code

```
c55x_code:
;...
   CALL c54x_func
;...
```

```
c54x_func:
       ; Code to adjust parameter passing (if required)
       ; and save to the stack the C55x registers that
       ; need to survive the call to c54x_func. C54x to
       ; C55x Register mapping is covered in the
       ; the C55x Assembly Language Tools User's Guide.
       C54X_ENTRY
       ; Your code. You can use/set any C54x register.
       C54X_EXIT
       ; If applicable, code restores C54x registers from
       ; the stack.
       RET
```

Example 4–4. Macros to Use When Calling a C54x Function from C55x Code

```
C54X_ENTRY   .macro
   PSH    mmap(@ST1_55)    ; See Note 1
   PSH    mmap(@ST2_55)    ; See Note 1
   PSH    mmap(@BSA01)     ; See Note 1
   PSH    mmap(@BSA23)     ; See Note 1
   PSH    mmap(@BSA45)     ; See Note 1
   PSH    mmap(@BSA67)     ; See Note 1
   PSH    mmap(@DPH)       ; See Note 1
   PSH    mmap(@CDPH)       ; See Note 1

   BSET   C54CM             ; See Note 2
   BCLR   M40               ; See Note 1
   BCLR   ARMS              ; See Note 1
   BCLR   RDM               ; See Note 1
   AND    #0FF00h,*(ST2_55) ; See Note 1
   MOV    #0, mmap(@DPH)    ; See Note 1
   MOV    #0, mmap(@CDPH)   ; See Note 1
   MOV    #0, mmap(@BSA01)  ; See Note 1
   MOV    #0, mmap(@BSA23)  ; See Note 1
   MOV    #0, mmap(@BSA45)  ; See Note 1
   MOV    #0, mmap(@BSA67)  ; See Note 1
          .endm
```

```
C54X_EXIT .macro
   BCLR   C54CM        ; See Note 2
   POP    mmap(@CDPH)  ; See Note 1
   POP    mmap(@DPH)   ; See Note 1
   POP    mmap(@BSA67) ; See Note 1
   POP    mmap(@BSA45) ; See Note 1
   POP    mmap(@BSA23) ; See Note 1
   POP    mmap(@BSA01) ; See Note 1
   POP    mmap(@ST2_55); See Note 1
   POP    mmap(@ST1_55); See Note 1
          .endm
```

Notes: 1) This instruction is required only if the C55x code assumes a value for this register or bit other than the value required for “C54x masm55 compatibility” (refer to the masm55 C54x context requirements in Table 2–2 on page 2-3).

2) This instruction is needed only if nested loops or C55x circular addresing is used inside the C55x native functions.

4.3 Dual MAC Optimizations

Background

- A C54x DSP can implement one multiply-and-accumulate (MAC) operation per cycle with two independent data operands coming from the C and D buses of the CPU, as shown below:

  ```
  MAC*AR2+, AR3+, A     ; A = A + (  *AR2   x   *AR3)
  ```

  Where AR2 uses the C bus and AR3 uses the D bus

- A C55x DSP can implement two MAC operations per cycle with three independent operands coming from the C, D, and B buses. The data provided by the B bus is used by both MAC units of the CPU; therefore, the third and fourth operands of the dual-MAC instruction must be the same, as shown in the C55x dual-MAC instruction below:

  ```
  MAC*AR2+, *CDP+, AC0  ::  MAC  *AR3+, *CDP+, AC1
  ```

  Where AR2, AR3 and CDP use the C, D and B bus respectively. CDP points to the common operand, typically a filter coefficient when dual MAC is used to implement a typical block FIR. Also the common operand (pointed by CDP) must be located in internal memory, as the B bus is not connected to external memory.

Recommendation

When possible, replace C54x functions that perform single-MAC operations with C55x native functions that perform dual-MAC operations. It could give you a 2:1 cycle improvement ratio.

The *TMS320C55x DSP Programmer's Guide* (SPRU376) provides extensive examples on how to take advantage of the C55x dual-MAC hardware when writing code for such things as FIR filters, multichannel applications, complex vector multiplication, symmetrical/anti-symmetrical filters, and matrix multiplication.

Example 4–5 and Example 4–6 show equivalent C55x native code (single- and dual-MAC versions) for the original C54x block FIR code shown in Example 3–2 (page 3-4). The dual-MAC implementation shown in Example 4–6 provides a great reduction in execution cycles but some increase in code size. It is up to the programmer to make the trade-off between code size and speed. Both versions of the code use the new C55x calling convention as well as the C55x circular addressing scheme (which is discussed in section 4.4). Table 4–3 shows benchmarks on the C54x block FIR code from the initial C54x version through to the native C55x dual-MAC implementation.

Table 4–3. Block FIR Example Cycle Benchmarks

	Native C54x (Example 3–2)	MASM55 (Phase 1)	Modified MASM55 (Phase 2)	Native C55x Using Single MAC (Phase 3)	Native C55x Using Dual MAC (Phase 3)
Code Size (Bytes)					
fir.asm	94	102 (8.5%)	98 (4.2%)	109	135 (43.6%)
(NX = 256, NH = 64)					
fir.asm	9410 (0%)	9672 (2.78%)	9542 (1.4%)	8767 (–6.8%)	4555 (–51.6%)

Note: A positive percentage indicates C55x cycle growth.

Example 4–5. Native C55x FIR Assembly Function (Single MAC)

```
;...
_fir:
;...
   ||RPTBLOCAL  END_LOOP

       MOV    *x_ptr+, *db_ptr

       MPYM   *db_ptr+, *h_ptr+, AC0
       ||RPT  #inner_cnt
          MACM   *db_ptr+, *h_ptr+, AC0, AC0
       MACMR  *(db_ptr–T1), *h_ptr+, AC0, AC0
END_LOOP:
       MOV    HI(AC0), *r_ptr+
;...
```

Example 4–6. Native C55x FIR Assembly Function (Dual MAC)

```
;...
_fir:
;...
   ||RPTBLOCAL   END_LOOP

       MOV    *x_ptr+, *db_ptr1
       MOV    *x_ptr+, *db_ptr2

       MPY    *db_ptr1+, *h_ptr+, AC0
       ::MPY  *db_ptr2+, *h_ptr+, AC1
       ||RPT  #inner_cnt
          MAC    *db_ptr1+, *h_ptr+, AC0
          ::MAC  *db_ptr2+, *h_ptr+, AC1
       MACR   *(db_ptr1–T1), *h_ptr+, AC0
       ::MACR *(db_ptr2–T1), *h_ptr+, AC1
END_LOOP:
       MOV    pair(HI(AC0)), dbl(*r_ptr+)
;...
```

4.4 Circular Addressing Optimization

Background

- C55x native circular addressing is different from C54x circular addressing. First, C55x devices require no circular buffer memory alignment and can support up to five different buffers and three sizes of circular buffers. This allows you to simplify memory mapping and increase flexibility. Second, in C54x ARn contained the absolute data address. In C55x native circular addressing, ARn is now an index (offset) into the data buffer pointed by a BSAxx register. This implies software changes.

- There are two ways of circularly modifying ARn on a C55x DSP: You can set the corresponding linear/circular mode bit in ST2_55, or you can use the circ() instruction qualifier. However, circ() makes all AR addressing in the instruction circular.

- In the C54x-compatible mode, the C55x native circular addressing scheme is not accessible and the alignment requirements of the C54x circular buffering scheme need to be respected. **You must set C54CM=0 when wanting to use the C55x native circular mode. However, the C54x circular mode will run in both C54CM=0 and C54CM=1.**

Recommendation

If your original C54x code uses circular addressing, you should use – –purecirc (see section 3.2 on page 3-6) for more efficient code porting. However, in a few cases, MASM55 porting of C54x circular addressing code still might not be cycle and code size efficient. You can inspect how MASM55 ports your specific circular addressing code by looking at the listing file and deciding whether manual code replacement with C55x native circular mode is required. An example of manual replacement is shown below.

Example: Manual Code Replacement for C55x Circular Addressing

Example 4–7 and Example 4–8 show equivalent circular buffer implementations in C54x code and in C55x code, respectively. Sections 4.4.1 through 4.4.3 describe the steps used to produce the C55x circular addressing code of Example 4–8.

Example 4–7. C54x Implementation of a Circular Buffer

```
cbuffer .sect ”.cbuffer”          ; circular buffer (memory aligned)
   .word 1
   .word 2
;...
   .word 256

lbuffer .usect ”.lbuffer”, 256    ; linear buffer
   .text
;...

   STM    #256, BK                ; BK = block size
   STM    #1,AR0                  ; AR0 = index through circular buffer
   STM    #cbuffer, AR1           ; AR1 = circular buffer pointer
   STM    #lbuffer, AR3           ; AR3 = linear buffer pointer
   RPT    #255                    ; copy 256 data
   MVDD   *AR1+0%,*AR3+
;...
```

Example 4–8. C55x Native Implementation of a Circular Buffer

```
cbuffer .sect ”.cbuffer”             ; circular buffer (no memory aligned)
   .word 1
   .word 2
;...
   .word 256

lbuffer .usect ”.lbuffer”, 256       ; linear buffer
   .text
;...

   MOV    #256, BK03                 ; set block size
   OR     #2, mmap(ST2_55)           ; configure AR1 as circular, AR3 as linear
   AMOV   #cbuffer, XAR1             ; set main data page in AR1H (bits 23–16)
   MOV    #(cbuffer & 0ffffh), BSA01 ; set buffer start address (bits 15–0)
   MOV    #0, AR1                    ; AR1 = index within buffer (initially 0)
   AMOV   #lbuffer, XAR3             ; AR3 = linear buffer pointer
   RPT    #255                       ; copy 256 data
   MOV    *AR1+,*AR3+                ; no need of AR0 as a index
;...
```

4.4.1 Step 1: Load the Buffer Size Register

Store the buffer size in the buffer size register that corresponds to the chosen circular pointer:

For This Pointer ...	Use This Buffer Size Register ...
AR0, AR1, AR2, or AR3	BK03
AR4, AR5, AR6, or AR7	BK47
CDP	BKC

In Example 4–8 (page 4-10), BK03 is used because AR1 is the circular pointer:

C54x Code	C55x Code
`STM #256, BK`	`MOV #256, BK03`

4.4.2 Step 2: Tell the CPU to Modify the Pointer Circularly

Write to the ST2_55 circular/linear bits for the ARn used, or use the circ( ) modifier. Default (reset) behavior for the pointers is linear addressing. The OR instruction shown below (from Example 4–8) writes to ST2_55 bits: AR1LC=1 (AR1 circular) and AR3LC=0 (AR3 linear). The circ() modifier cannot be used with the MOV instruction here because AR3 must be modified as a linear pointer.

C54x Code	C55x Code
`STM #0, AR0`	
`MVDD *AR1+0%, *AR3+`	`OR #2, mmap(ST2_55)`
	`MOV *AR1+, *AR3+`

Note that C55x DSPs offer a more flexible Xmem/Ymem addressing and do not require the use of AR0 as an index.

4.4.3 Step 3: Load the Buffer Start Address Register and the Pointer

For This Pointer ...	Load This Start Address Register ...	Load this Auxiliary Register Extender ...
AR0	BSA01	XAR0
AR1	BSA01	XAR1
AR2	BSA23	XAR2
AR3	BSA23	XAR3
AR4	BSA45	XAR4
AR5	BSA45	XAR5
AR6	BSA67	XAR6
AR7	BSA67	XAR7
CDP	BSAC	XCDP

In C54x DSPs, there are no buffer start address (BSA) registers, and a circular buffer requires memory alignment in all cases. ARn is initialized to the absolute address inside the buffer, typically to the beginning of the circular buffer.

In C55x DSPs, the 23-bit starting address of the buffer consists of the 7 most-significant bits(ARxH) of the corresponding auxiliary register extender concatenated with the corresponding 16-bit BSAxx register. As shown in the following tables, for AR1, the corresponding auxiliary register extender and BSAxx register are XAR1 and BSA01 respectively. Therefore, you must initialize the BSA register, the pointer, and the corresponding auxiliary register extender, as shown in Example 4–8 (page 4-10) and reproduced below. The buffer start address (cbuffer) is written to the BSA register (BSA01) and to the auxiliary register extender (XAR1). AR1 is then initialized to 0 (it points to the top of the buffer).

C54x Code	C55x Code
	`AMOV #cbuffer, XAR1`
`STM #cbuffer,AR1`	`MOV #(cbuffer & 0fffh), BSA01`
	`MOV #0, AR1`

Special case: C54x ARn Does Not Initially Point to the Base of the Buffer

If in the original C54x code, ARn is not initially pointing to the buffer start address, but a different value inside the circular buffer, additional rework is necessary.

The start address of the buffer must be stored in BSA01, and the buffer offset to store in AR1 is the original value minus the start address.

If after using ARn for circular addressing, you need the original value back in ARn, you can add the start address to ARn. For example:

```
ADD BSA01, AR1
```

4.5 Optimal Loop Implementations

C54x and C55x DSPs offer similar repeat/repeat block mechanisms with the following differences and improvements in the C55x DSPs:

- 2-level repeat block loop nesting supported (under C54CM = 0)
- Different end-of-loop label positioning (see section 4.5.1)
- New RPTBLOCAL instruction
- New repeat single with CSR instructions

The *TMS320C55x DSP Programmer's Guide* (SPRU376) gives details and examples on the general usage of the new RPTBLOCAL and repeat single with CSR instructions. When replacing C54x code with C55x native code, you can take advantage of these new features if you:

- Replace the C54x RPTB instruction with the C55x RPTBLOCAL instruction when possible (less than 56 bytes and no backwards jumping inside the code). This will ensure that the code inside the loop gets executed inside the instruction buffer queue (IBQ), avoiding instruction stalls that could occur while accessing memory. By reducing the number of memory accesses, you also save power in the DSP.

- Use RPT/RPTADD/RPTSUB (with CSR) looping when a C54x repeat single count is computed at run time inside an outer loop (see section 4.5.3).

- Use RPTB or RPTBLOCAL instead of BANZ to implement an outer loop (see section 4.5.2).

4.5.1 Differences in End-of-Loop Label Positioning

In C54x code, the end-of-loop label (as in RPTB label) must point to the last word of the last instruction inside the loop. A common C54x practice to meet this requirement, even in the case of the last instruction being a multi-word instruction, is to use "label–1" instead of "label" as shown in the code below.

In C55x code, the end-of-loop label position is different. The label must point to the first byte of the last instruction inside the loop. Even though MASM55 understands and adjusts for the use of "label–1", the correct, and more intuitive, way to use end-of-loop label in C55x is shown below. In this code, the loop repeats instruction inst 1 through inst n, and inst_out is the first instruction outside the loop.

C54x Code

```
       RPTB label-1
       inst 1
       inst 2
       ;...
       inst n
label  inst_out
```

C55x Code

```
       RPTB label
       inst 1
       inst 2
       ;...
label  inst n
       inst_out
```

4.5.2 Use RPTB or RPTBLOCAL Instead of BANZ to Implement an Outer Loop

You can replace an outer loop implemented via BANZ with a RPTB or RPTBLOCAL loop using BRC0, as shown in the following example. Notice that even though MASM55 maps BRC into BRC0, the usage of BRC0 and BRC1 was reversed to allow for this optimization. The C55x devices use BRC1 for inner loops and BRC0 for outer loops. As mentioned in section 3.3.1 (page 3-7), automatic handling of nested RPTB/RPTBLOCAL operations with dedicated registers is only supported in the C55x native mode (C54CM = 0).

C54x Code

```
   LD     *AR2, ASM
   STM    #6, AR5

loop0:
   ;...
   STM    #4, BRC
   RPTB   loop1-1
   ;...
loop1
   ;...
   BANZ   loop0, *AR5-
```

C55x Code

```
   MOV    *AR2,T2
   MOV    #5, BRC0
   MOV    #4, BRC1

   RPTB   newloop0-1
   ;...
   RPTB   loop1-1
   ;...
loop1
   ;...

newloop0
```

4.5.3 Use of RPTSUB and RPTADD Instructions

Compared with C54x, C55x offers new repeat single instructions with CSR (RPTADD and RPTSUB) that can be used efficiently when an instruction has to be repeated for variable number of times in a loop. The instruction repeats the number of times specified by the CSR register.

Consider a block, which is repeated 10 times. In this block, the MAC operation has to be performed 10 times in the first iteration, 9 times in the second iteration and so on…. and it has to be performed once for last iteration. This can be written as follows:

```
MOV          #9, BRC0 ; load BRC0 with 10 – 1 = 9 count
MOV          #9, CSR  ; load CSR with 10–1 = 9 count for the
                      ; first MAC operation
…
RPTBLOCAL loop
     …
     …
     RPTSUB   CSR, #1
     MACM      *AR2+, *AR3+, AC0
     …
     …
loop:
```

In the above example MACM will be performed 10 times in first iteration since CSR = 9 and then CSR value will be modified as CSR = CSR – 1, i.e., CSR = 8. Thus for next iteration MACM will be repeated nine (CSR + 1) times. Similarly, we can use RPTADD where the CSR needs to be added with a constant. In this way, these instructions provide us with a way of modifying the count value without consuming any extra cycles.

4.6 Use of the A-Unit ALU

Compared with C54x, C55x offers a new A-unit 16-bit ALU that can be used for:

- **More efficient ARn computation**. In C54x DSPs, ARn arithmetic was implemented by using accumulators. In C55x DSPs, such a practice is not required, as the device offers a separate 16-bit A-unit ALU. This translates into cycle and code size savings. For example:

C54x Code	C55x Code
	`MOV #imm,AR2`

`LD #imm, A`
`ADD AR3, A`
`STLM A, AR2`
	`ADD AR3,AR2`

- **Efficient A-unit register initialization through the AMOV instruction.** TMS320C55x provides us with two types of move instructions, MOV (that executes in the execute phase of the pipeline) and AMOV (that executes in the address phase of the pipeline). With the new AMOV instruction, C55x can initialize A-unit registers earlier in the pipeline and in this way minimizes potential pipeline stalls. This is shown in the following case :

Code with 4 Pipeline stalls		Code with no Pipeline stalls	
`MOV`	`#y, AR1`	`AMOV`	`#y, AR1`
`MOV`	`*AR1, AC0`	`MOV`	`*AR1, AC0`

For a detailed explanation, refer to the *TMS320C55x Programmer's Guide* (SPRU376).

4.7 Use of the Additional Accumulators and T Registers

C54x has two accumulators (A and B) and one Temporary register T. C55x added two accumulators (now, AC0, AC1, AC2, and AC3) and one temporary register (now T0, and T1). The extra accumulators and registers can be used effectively for temporary storage, to avoid pipeline stalls and in some cases to enable parallelism.

For temporary storage, for example, the accumulators and T registers could be used instead of data memory to store intermediate results. In this way unnecessary load/store operations to memory are avoided. Also, the additional C55x T registers can be used as ARx pointer indexes facilitating pointer manipulation. The *TMS320C55x Programmer's Guide* (SPR376) provides with examples on efficient accumulator and T register usage.

4.8 Use of Improved Dual Reads and Writes for Faster Data Movement

C54x offered two 16-bit read buses (C and D) but only one 16-bit write bus (E). C55x adds one read bus (B) and one write bus (F). This allows the C55x to achieve a 32-bit write in one cycle compared with the C54x that takes two cycles. The *TMS320C55x Programmer's Guide* (SPR376) provides with examples on efficient data movement.

4.9 Using the Less Restrictive xmem/ymem Addressing

In both C54x and C55x, the special dual AR indirect addressing mode (xmem/ymem) enables you to make two 16-bit data memory accesses. However, the C55x offers cycle savings opportunities with a more orthogonal xmem/ymem addressing:

- C54x xmem/ymem addressing was limited to only four possible combinations (*ARx, *ARx+, *ARx–, *ARx+0%). The only register indexing possible was through AR0 and has to be used under circular addressing mode (*ARx+0%), forcing the extra initialization of the BK register to zero in order to achieve linear mode indexing. C55x expands xmem/ymem register indexing to T0 and T1 and allows register indexing with or without modification of the auxiliary register involved.
- C54x xmem/ymem addressing was restricted to only a subset of the eight auxiliary registers available. Only AR2, AR3, AR4 and AR5 were allowed. Instead, C55x xmem/ymem addressing can use any of the eight auxiliary registers (AR0 to AR7) with no restrictions.

4.10 Use of the Additional TC Bits

C54x offered one single TC bit that stores the results of the ALU operations for later usage in conditional program flow instructions . C55x now offers two TC bits (TC1 and TC2) that can be used for example with the XCCPART instruction to execute code based on a logical operation between two conditions evaluated in TC1 and TC2, avoiding extra cycles. An example is shown below.

```
BTST    @#adpcm_cod_c_31, AC2, TC1
...
BTST    #1, *AR3+, TC2
...
XCCPART secomp_WB6MAG, TC1^TC2
```

The *TMS320C55x Programmer's Guide* (SPR376) provides with other examples on efficient usage of the TC bits.

4.11 Other Potential Optimizations

Table 4–2 covered some of the most important C55x architectural features to be exploid during C55x native coding. The following are some other new C55x instructions worth mentioning:

- **Use of XCC and XCCPART Instructions:** The Conditional Execute Instructions are powerful instructions that can be used to avoid branches while implementing if-else sort of operations. Compared with C54x, branches are more expensive than in C55x (between 4–6 cycles) as the C54x delayed branches do not exist in the C55x. C55x supports two types of conditional instructions, XCC and XCCPART, both taking one cycle. Depending on the code characteristics, it may be more cycle efficient to use couple of XCC (or XCCPART) instructions in sequence rather than using branches. Refer to the C55x Programmers Reference Guide for a detailed explanation of the pipeline differences between XCC and XCCPART and their use on pipeline stall avoidance.

- **Use of parallel MANT :: NEXP instruction:** The Calculation of Exponent Value (EXP) and normalization of the Accumulator (NORM) with respect to exponent value takes two cycles in the C54x. Now in the C55x this can be performed in a single cycle in C55x by the use of the MANT ACx, ACy:: NEXP ACx, Tx instruction. This instruction computes the exponent and mantissa of the source accumulator ACx and store the exponent (negated) and mantissa in Tx and ACy respectively.

Appendix A

Reserved Symbols of the TMS320C55x Code Generation Tools

This appendix lists symbols that are reserved in the code generation tools for the TMS320C55x™ (C55x™) DSPs. You cannot assign your own values to these symbols.

Topic **Page**

A.1 Operand Modifiers

ABS16
AR0B
BLOCK
COEF
DBL
DR0B
HI
LO
M40
MMR
PAIR
RND
SHORT
T0B
UNS

A.2 Register Names and Other Special Operands

AC0
AC1
AC2
AC3
AD_UNIT
AR0
AR1
AR2
AR3
AR4
AR5
AR6
AR7

BK03
BK47
BKC
BOF01
BOF23
BOF45
BOF67
BOFC
BORROW
BRC0
BRC1
BSA01
BSA23
BSA45
BSA67
BSAC

CARRY
CDP
CSR

D_UNIT
DP
DPH
DR0
DR1
DR2
DR3

HIGH_BYTE

LCRPC
LOW_BYTE

MDP
MDP05
MDP67

OVERFLOW

PC
PDP
PORT

RETA
RPTC

SP
SSP
ST0
ST1
ST2
ST3

T0
T1
T2
T3
TC1
TC2
TRN0
TRN1

XAR0
XAR1
XAR2
XAR3
XAR4
XAR5
XAR6
XAR7
XCDP
XDP
XSP
XSSP

A.3 Instruction Keywords

ABDST
ABORTI
ADSC
ADS2C

BIT
BLOCKREPEAT
BLOCKREPEATLABEL

CALL
CBIT
CIRCULAR
COMPARE
COPR
COUNT

DELAY

ESTOP_0
EALLOW
EDIS
ESTOP_1
EXECUTE
EXP

FAR
FIELD_EXPAND
FIELD_EXTRACT
FIRS
FIRSN

GOTO

IDLE
IF
INTR

LINEAR
LMS
LOCAL
LOCALREPEAT
LOCALREPEATLABEL

MANT
MAR
MAX
MAX_DIFF
MAX_DIFF_DBL
MIN
MIN_DIFF
MIN_DIFF_DBL
MMAP

NOP
NOP_16

POP
POPBOTH
PUSH
PSHBOTH

READPORT
REPEAT
RESET
RETURN
RETURN_INT

SATURATE
SFTC
SIM_TRIG
SQDST
SUBC
SWAP

TRAP

WHILE
WRITEPORT

A.4 Status Register Bit Names

ACOV0
ACOV1
ACOV2
ACOV3
ARMS
AR0LC
AR1LC
AR2LC
AR3LC
AR4LC
AR5LC
AR6LC
AR7LC
AVIS

BRAF

C16
C54CM
CACLR
CAEN
CAFRZ
CARRY
CBERR
CDPLC
CLKOFF
CPL

DBGM

EALLOW

FRCT

HINT
HM

INTM

M40
MPNMC

RDM

SATA
SATD
SMUL
SST
SXMD

TC1
TC2

XF