Converting Code from the TMS320C5x DSP to the TMS320C2xx DSP

APPLICATION REPORT: SPRA293

Henry D. Hendrix
Senior Technical Staff DSP Applications

Digital Signal Processing Solutions
January 1998

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software 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.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI 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 semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Tables

Converting Code from the TMS320C5x DSP to the TMS320C2xx DSP

Abstract

With the introduction of the TMS320C2xx (C2xx) family of low-cost digital signal processors (DSPs), many customers are discovering that they can utilize this DSP family for designs that previously required the processing power of a TMS320C5x (C5x) DSP. Although some code has been written for the C2xx, reuse of the large repository of C5x application code can dramatically speed up designs using the C2xx. Since the architecture and instruction set of the C2xx is similar to the C5x, porting of code from the C5x to the C2xx is fairly straightforward.

The C2xx instruction set is fully capable of implementing the functionality of the omitted and changed C5x instructions, although usually at the cost of a minor increase in program memory and cycle counts. Since most code replacement involves the use of alternate resources, efficient porting of C5x code requires careful review of the surrounding code to prevent unneeded context save and restore. Sometimes restructuring the code is the only way to prevent large inefficiencies. For these reasons, no automatic conversion utilities exist.

This paper describes the architectural differences in the C5x and C2xx CPU cores and provides example replacement code for all omitted and changed C5x instructions. The implications of each replacement are also described, including the cost in cycles and memory usage. Because of the need for system-level consideration, differences in internal memory and peripherals and their configuration are not covered.

Product Support on the World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Major Conversion Considerations

The architectural differences between the C5x and C2xx cores relevant to producing working code can be summarized as follows:

- ❑ Functional units and registers
  - Parallel logic unit (PLU) and dynamic bit manipulation register (DBMR)
  - Accumulator buffer (ACCB)
  - Barrel shifter
  - Temporary registers (TREG1, TREG2)
- ❑ Addressing modes
  - Memory-mapped register addressing
  - Circular buffering
  - Auxiliary register compare register (ARCR)
  - Index register (INDX)
  - Dynamic addressing via block move address register (BMAR)
  - I/O data moves
- ❑ Program control differences
  - Conditional execution
  - Delayed branches
  - Block repeat
  - Interrupt return and shadow registers
  - Control and status bits
  - Idle mode
  - Automatic zero of accumulator and product register
  - Immediate operands

Although these differences define the major concerns, other system-level issues should be addressed when porting a design to the C2xx:

- ❑ Memory-map differences

  The C2xx has 544 words of internal dual-access RAM (DARAM); the C5x has 1056 words. This difference has no effect on specific instruction replacements but is a major system level consideration: the use of internal vs. external memory can have a large effect on processor throughput. In addition, internal single-access memory (SARAM) differences also exist among different family members in both the C2xx and C5x. The processor mode status register (PMST), which controls the configuration of SARAM, among other things, is also configured differently in the C2xx than the C5x.

- ❑ Peripheral hardware differences

  The types and numbers of peripherals (serial ports, timers, wait-state generators, host-port interfaces, PLLs, etc.) vary greatly among the different family members in both the C2xx and C5x families. For the most part, the registers used to control these peripherals are configured and accessed differently in each DSP family.

Design Porting Methodology

When converting a design from the C5x to the C2xx, several considerations can make the process easier:

- ❑ Invalid C2xx code detection

  All unsupported C5x instructions can be detected by simply running the C5x code through the assembler with the *-v2xx* switch. The resulting listing file will have an error or warning for each invalid C2xx instruction.

  However, certain valid instructions utilize different resources on the C5x than the C2xx and are **NOT** flagged as errors or warnings. These resources are limited to the TREG1/2, ARCR, and INDX registers and the memory-mapped I/O ports. Refer to the sections on temporary registers, auxiliary registers, and I/O ports for more details.

- ❑ Memory-map changes

  Remapping of the memory should be completed before attempting the instruction changes. Some instructions, such as DMOV and MACD, only work properly using internal memory. Use of these instructions with external operands will not be detected as errors but will result in non-working code (for a method to eliminate the need for data movement in external memory, see the section, *Circular Buffering*.) Also be aware of any hard-coded addresses that may need changing. It is best to use relocatable addresses and let the linker assign their value.

- ❑ Peripheral changes

  Control of peripheral hardware should be handled separately from the other instruction changes. Although many of the peripherals are similar, the location and contents of their control registers are often different. Changing the instructions that access the peripherals will not likely be adequate.

- ❑ Test procedure

  Due to the upward compatibility of the C2xx, code converted to the C2xx can still be run on a C5x platform, isolating any problems to specific code changes. The memory-map can be left in C5x-compatible form or converted to emulate the memory available on the C2xx. All peripheral access code should be left in C5x-compatible form. Then the converted code is reassembled for the C5x and debugged on its original platform. Two compatibility bits in the C5x PMST register must be managed to properly emulate C2xx CPU. The NDX and TRM bits should be set to zero to allow C2xx instructions utilizing the TREG1/2, ARCR, and INDX to be properly emulated on the C5x.

To get the most efficiency from the ported code, you must have some knowledge of the surrounding code. The following suggestions deal with minor differences between the instruction replacements and the original code:

❑ Context changes from replacement code

Several of the replacement methods given here affect the content of registers and status bits (C, TC, OV, etc.) differently than the original code. While code can be inserted to save the context and exactly match the effect on status bits, this is often unnecessary. However, when debugging ported code, these affected registers and status bits are a good area to check.

❑ Location of temporary memory

Memory locations are typically used to emulate missing registers or to save the context of functional units. For context saves, it is most efficient to use available memory locations on the current data page, as there is a 4-cycle penalty for changing the data page, then changing it back. Register replacements can also be located on the current data page if their use is temporary. However, to emulate a register with global visibility, it is often better to use a specific memory location (on data page zero, for instance). Although the overhead is higher, this method allows data to be passed between functions in that emulated register. If an auxiliary register is available, it can be used to access replacement memory anywhere in the map without data page changes.

❑ Conditional instructions

Conditional branches or calls often directly follow instructions that set the conditions. If the instruction which set the condition must be replaced, it is often more effective to utilize a new conditional instruction, rather than inserting code to reproduce the original condition. For example, consider the following C5x code:

```
APL  #0Fh,dma    ;clear dma, except 4 LSBs,
                 ;set TC if LSBs = 0
BCND next,TC     ;branch if LSBs = 0
```

This could be converted into code that performs the “AND” and explicitly sets the TC at a cost of 4 or 5 cycles. However, since we must use the accumulator for the “AND”, we might as well test the condition there:

```
LACC  dma         ;ACC = dma
AND   #0Fh,dma    ;ACC = 0000 0000 0000 xxxx
SACL  dma         ;dma = 0000 xxxx
BCND  next,EQ     ;branch if LSBs = 0
```

- Single instruction repeat mode

  Alternate replacement methods may be more efficient when replacing code in single-instruction repeat mode. For example, a repeated MACD (multiply and accumulate with data move) ported to operate on external data could be split into two small loops, rather than one large one. The first loop implements the MAC only, retaining its single-cycle operation, and the second does the data move, using BLDD, and is also single-cycle. If both operations were in a block repeat, neither would be single-cycle and the overhead of a branch instruction would be incurred.

- Barrel shift and store

  In many cases, a BSAR followed by a store is used to store accumulator data with a right shift, since SACL/H only allows a left shift. For shift values of 5 and more followed by an SACL, it is more efficient to do a left shift of the accumulator followed by a shifted SACH. The amount of the left shift must be 16 - R, where R is the desired amount of the right shift. Since SACH allows a shift up to 7 bits, the number of discrete left shifts required is 9 - R. Thus for R > 9, all left shifts can be done by the SACH instruction. Although this method results in different accumulator contents, it provides a good example of the most efficient replacement method when all that was needed is a shifted store. The following example saves 5 cycles over the right-shift then store method.

  C5x code **C2xx code**

```
BSAR  7             SFL
SACL  temp          SFL
                    SACH  temp,7
```

Summary of Workarounds

Table 1 summarizes the proposed replacements for non-supported C5x code.

Table 1. Replacements for Non-Supported C5x Code

Changed from C5x	Workaround	Typical Cycle Increase	Comments
No PLU	Use accumulator	2	
No accumulator buffer	Use local data memory	1 cycle for 32-bit arithmetic op's 9 cycles for 32-bit boolean, compare, and shift op's	No extra cycles for 16-bit operations
No single-cycle barrel shifts	Use multi-cycle shifts or restructure code	1 per shifted bit	Can often use output shifter
No specialized registers (ARCR, INDX, TREG1/2)	Use other registers	0	
No memory-mapped registers	Set data page and/or use direct loads and stores (LAR, LT, etc.)	2	
No circular buffering hardware	Use DMOV or bit-reversed addressing	0	Avoid manual address checks
No dynamic addressing modes	Use immediate addressing or ACC instead of BMAR	0	Use ACC to retain dynamic addressing
No conditional execution (XC)	Use conditional branch	1	
No delayed branches, calls, or returns	Use non-delayed versions	2	
No zero-overhead block repeat	Use BANZ or unroll loop	4 per loop	Small loops can be unrolled
No interrupt shadow registers	User must save context	21 cycles per ISR	Must only save those registers modified by ISR
No auto-zero of accumulator and product register	Load zero	1	
Limited use of long immediate operands	Use data memory	1	

Detailed Instruction Replacements

The following sections detail the replacement code for each non-supported C5x instruction. Each section explains the C5x instruction(s), the replacement code, and any context changes incurred by the replacement code. The replacements are organized by the architectural differences as outlined above, with all relevant instructions included for the given difference.

The following notation is used in the replacement code tables:

[ ]	= optional
wxyz	= WXYZ register replacement in data memory (for example, DBMR, BMAR)
dma	= data memory access (either direct or indirect)
pma	= program memory access
ARx	= auxiliary register x
ACC	= accumulator
ACCB	= accumulator buffer
Preg	= product register
Treg	= temporary register

Context Save/Restore

Most of the replacement code has the effect of changing the contents of specific registers or status bits (for example, the accumulator or carry bit). Rather than including code to restore the exact context after every replacement, this paper will simply list any change in context for each replacement. If these context changes affect the surrounding code, the code should be rearranged, if possible, to minimize the effect. If this is not possible, the register or status bit should be saved beforehand and restored afterward. Appendix B details methods for context save and restore.

Missing Functional Units and Registers

Parallel Logic Unit (PLU)

The functional Parallel Logic Unit (PLU) and associated Dynamic Bit Manipulation Register (DBMR) do not exist on the C2xx. Four C5x instructions make use of these resources: APL, OPL, XPL, and CPL. This unit allows direct operations on memory without affecting the accumulator's contents.

APL, OPL, and XPL

These instructions perform Boolean operations directly on data memory locations, replacing the data with the result of the operation. If no immediate value is specified, the DBMR register is used as the first operand. The Boolean operation sets the TC bit to 1 if the result is zero, and sets the TC bit to 0 if the result is not zero.

Replacement

The preferred replacement technique is to perform the same function using the accumulator. When DBMR is used as one operand, its current value (if local) or a memory location (if global) must be used in the replacement function.

Table 2. Replacement Code for APL, OPL, and XPL Instructions

C5x Instruction	C2xx Replacement Code	Comments

`APL [#lk,]dma`	`LACC dma`
	`AND [#lk][dbmr]`
	`SACL dma`
		Modifies ACC
		Does not set TC

`OPL [#lk,]dma`	`LACC dma`
	`OR [#lk][dbmr]`
	`SACL dma`
		Modifies ACC
		Does not set TC

`XPL [#lk,]dma`	`LACC dma`
	`XOR [#lk][dbmr]`
	`SACL dma`
		Modifies ACC
		Does not set TC

If the TC bit is required to be set, the following code can be used at a cost of 4 or 5 cycles:

```
           CLRC  TC          ;set TC = 0
           BCND  next,EQ     ;if ACC = 0,
           SETC  TC          ;set TC = 1
next       next instruction
```

However, if possible, the accumulator status (EQ or NEQ) should be used by an instruction utilizing the condition instead of the TC bit.

CPL

This instruction compares the immediate value or DBMR value with the data. The data memory location is unchanged. The TC bit is set to 1 if the values are the same and set to 0 if they are different.

Replacement

Two replacement methods are available:

- ❑ Using the accumulator
- ❑ Using auxiliary registers

The accumulator method is more efficient but indicates the results via the accumulator status bits. The auxiliary register method sets the TC bit. When DBMR is used as one operand, its current value (if local) or a memory location (if global) must be used in the replacement function.

Table 3. Replacement Code for CPL Instruction

C5x Instruction	C2xx Replacement Code	Comments

`CPL [#lk],dma`	`LAR AR0,[#lk][dbmr]`
	`LAR ARx,dma`
	`MAR *,ARx`
	`CMPR 00`
		Compare using two ARs
		Modifies AR0, ARx, and ARP
		Sets TC

	`LACC [#lk][dbmr]`
	`SUB dma`
		Compare using ACC
		Sets ACC status (EQ or NEQ)
		Modifies ACC and carry bit (C)
		Does not set TC

If the second method is used and the TC bit is required to be set, the following code can be used at a cost of 4 or 5 cycles.

```
        CLRC  TC          ;set TC = 0
        BCND  next,EQ     ;if ACC = 0,
        SETC  TC          ;set TC = 1
next    next instruction
```

However, if possible, the accumulator status (EQ or NEQ) should be used by an instruction utilizing the condition instead of the TC bit.

Accumulator Buffer (ACCB)

The Accumulator Buffer (ACCB) 32-bit register is not present on the C2xx. The C5x can access it only through the accumulator – it is not a memory-mapped register. The 16 instructions that utilize the ACCB can be grouped as load/store instructions, arithmetic instructions, Boolean instructions, compare instructions, and 65-bit shift and rotate instructions.

General Replacement Strategy

Two data memory locations must be used to fully emulate the 32 bits of the ACCB. To be most efficient, these memory locations should be on the current data page. However, if global visibility is required, it may be preferable to allocate memory on a specific data page used throughout the code. Data page management must be included in this case.

Many replacements require temporary memory to maintain the contents of the ACC while the ACCB is being emulated. Since these memory locations are always temporary, they should be on the current data page, unless the data page has already been changed to access the replacement ACCB.

If only 16 bits of the ACCB are being used, the replacement code can often be streamlined. In addition, only one memory location is needed.

ACCB Load and Store: LACB, SACB, and EXAR

The LACB, SACB, and EXAR instructions load the ACCB from the ACC, store data from the ACCB to the ACC, and exchange data between the ACCB and ACC.

Replacement

The load and store instructions are replaced with simple load/store instructions from memory. The EXAR instruction requires additional memory as a temporary holding location for the data to be exchanged.

Table 4. Replacement Code for ACCB Load and Store Instructions

C5x Instruction	C2xx Replacement Code	Comments

`LACB`	`LACC accbh,16`
	`OR accbl`
		accbh/l in memory

`SACB`	`SACH accbh`
	`SACL accbl`
		accbh/l in memory

`EXAR`	`SACH temp2`
	`SACL temp1`
	`LACC accbh,16`
	`OR accbl`
	`BLDD #temp2,accbh`
	`BLDD #temp1,accbl`
		accbh/l and temp1/2 in memory on same DP

ACCB Arithmetic Instructions: ADCB, ADDB, SBB, and SBBB

The ACCB Arithmetic instructions perform arithmetic operations between the ACCB and ACC. The results are placed in the ACC, leaving the ACCB unchanged. Like other arithmetic operations, they affect the carry (C) and overflow (OV) bits and are affected by the overflow mode (OVM).

Replacement

Since 32-bit arithmetic is required, the add/subtract instructions with sign-suppression are used for the lower 16 bits. The order of operation is important if the status of the carry bit is to be maintained.

Table 5. Replacement Code for ACCB Arithmetic Instructions

C5x Instruction	C2xx Replacement Code	Comments

`ADCB`	`ADDC accbl`
	`ADD accbh,16`
		accbh/l in memory

`ADDB`	`ADDS accbl`
	`ADD accbh,16`
		accbh/l in memory

`SBB`	`SUBS accbl`
	`SUB accbh,16`
		accbh/l in memory

`SBBB`	`SUBB accbl`
	`SUB accbh,16`
		accbh/l in memory

ACCB Boolean Instructions: ANDB, ORB, and XORB

The ACCB Boolean instructions perform Boolean operations between the ACCB and ACC. The results are placed in the ACC, leaving the ACCB unchanged. No status bits are set.

Replacement

Since C2xx Boolean operations always place the results in the lower ACC, extra memory locations are required to properly place the 32-bit result in the accumulator.

Table 6. Replacement Code for ACCB Boolean Instructions

C5x Instruction	C2xx Replacement Code	Comments

`ANDB`	`SACH  temp1`
	`AND   accbl`
	`SACL  temp2`
	`LACC  temp1`
	`AND   accbh`
	`SACL  temp1`
	`LACC  temp1,16`
	`OR    temp2`
		accbh/l and temp1/2 in memory on same DP

`ORB`	`SACH  temp1`
	`OR    accbl`
	`SACL  temp2`
	`LACC  temp1`
	`OR    accbh`
	`SACL  temp1`
	`LACC  temp1,16`
	`OR    temp2`
		accbh/l and temp1/2 in memory on same DP

`XORB`	`SACH  temp1`
	`XOR   accbl`
	`SACL  temp2`
	`LACC  temp1`
	`XOR   accbh`
	`SACL  temp1`
	`LACC  temp1,16`
	`OR    temp2`
		accbh/l and temp1/2 in memory on same DP

ACCB Compare Instructions: CRGT and CRLT

The ACCB Compare instructions compare the value in the ACC with the value in the ACCB. The larger value (for CRGT) or smaller value (for CRLT) is placed in both registers. The carry bit (C) is set to one if the condition is true.

Replacement

A 32-bit comparison is done through the ACC and the appropriate value is then manually loaded into the ACC and ACCB. The carry bit must also be manually set, if required.

Table 7. Replacement Code for ACCB Compare Instructions

C5x Instruction	C2xx Replacement Code	Comments

CRGT	`      SUBS  accbl    ;test for larger value`
	`      SUB   accbh,16`
	`      BCND  abig,GEQ ;branch if ACC bigger`
	`      LACC  accbh,16 ;else set ACC = ACCB`
	`      OR    accbl`
	`      CLRC  C        ;clear carry bit`
	`      B     next`
	`abig  ADDS  accbl    ;restore ACC`
	`      ADD   accbh,16`
	`      SACH  accbh    ;set ACCB = ACC`
	`      SACL  accbl`
	`      SETC  C        ;set carry bit`
	`next`
		accbh/l in memory

CRLT	`      SUBS  accbl    ;test for smaller value`
	`      SUB   accbh,16`
	`      BCND  asml,LT  ;branch if ACC smaller`
	`      LACC  accbh,16 ;else set ACC = ACCB`
	`      OR    accbl`
	`      CLRC  C        ;clear carry bit`
	`      B     next`
	`asml  ADDS  accbl    ;restore ACC`
	`      ADD   accbh,16`
	`      SACH  accbh    ;set ACCB = ACC`
	`      SACL  accbl`
	`      SETC  C        ;set carry bit`
	`next`
		accbh/l in memory

65-Bit Shift and Rotate Instructions: ROLB, RORB, SFRB, and SFLB

The C5x allows shifts and rotates of the accumulator through the ACCB and carry bit, enabling a 65-bit shift path. Except SFRB, the 65 bit Shift and Rotate instructions are not affected by SXM.

Replacement

These instructions are replaced by code that extracts the MSB or LSB from the ACC (ACCB), then rotates/shifts it into the ACCB (ACC). Two temporary memory locations are required to maintain the contents of both the ACC and ACCB.

Table 8. Replacement Code for 65-Bit Shift and Rotate Instructions

C5x Instruction	C2xx Replacement Code	Comments
ROLB	SACH temp1 ;save ACC SACL temp2 LACC accbh,16 ;load ACCB OR accbl ROL ;shift C <- ACCB <- C SACH accbh ;save shifted ACCB SACL accbl LACC temp1,16 ;reload ACC OR temp2 ROL ;shift C <- ACC <- C	accbh/l and temp1/2 in memory on same DP
RORB	ROR ;shift C -> ACC -> C SACH temp1 ;save ACC SACL temp2 LACC accbh,16 ;load ACCB OR accbl ROR ;shift C -> ACCB -> C SACH accbh ;save shifted ACCB SACL accbl LACC temp1,16 ;reload ACC OR temp2	accbh/l and temp1/2 in memory on same DP
SFLB	SACH temp1 ;save ACC SACL temp2 LACC accbh,16 ;load ACCB OR accbl SFL ;shift C <- ACCB <- 0 SACH accbh ;save shifted ACCB SACL accbl LACC temp1,16 ;reload ACC OR temp2 ROL ;shift C <- ACC <- C	accbh/l and temp1/2 in memory on same DP
SFRB	SFR ;shift 0* -> ACC -> C SACH temp1 ;save shifted ACC SACL temp2 LACC accbh,16 ;load ACCB OR accbl ROR ;shift C -> ACCB -> C SACH accbh ;save shifted ACCB SACL accbl LACC temp1,16 ;reload shifted ACC OR temp2	* shift value depends on SXM accbh/l and temp1/2 in memory on same DP

In some cases, it may be more efficient to restructure the algorithm than to emulate it directly. This is especially true if multiple bit shifts or rotates are done, for example, a repeated ROLB or SFRB.

Barrel Shifter

The C2xx does not have the 1 to 16-bit prescaling shifter of the C5x, thus cannot do multiple bit shifts on the accumulator or accumulator loads with variable shifts.

BSAR

The BSAR instruction shifts the accumulator right by 1 to 16 bits.

Replacement

The simplest method of replacement is to perform repeated single bit shifts. Since the repeat instruction performs n+1 repetitions, the repeat should be done one less time than the value specified in BSAR. The repeat loop can be unrolled to save a cycle.

Table 9. Replacement Code for BSAR Instruction

C5x Instruction	C2xx Replacement Code	Comments

`BSAR n`	`RPT #n-1`
	`SFR`
		#n single bit shifts
		n is a constant

	`SFR`
	`SFR`
	`SFR`
	`...`
		#n discrete shifts
		(unrolled loop)

SATL, SATH

The SATL and SATH instructions shift the accumulator right by the amount specified in TREG1. SATL shifts by the amount specified by the 4 LSBs; SATH shifts by 16 bits, if bit 4 is a one. These instructions cannot be emulated through TREG, although other instructions using TREG1 and TREG2 can be (see the section, *Temporary Registers (TREG1, TREG2)).*

Replacement

Since the TREG1 register does not exist on the C2xx (see the section, *Temporary Registers (TREG1, TREG2)),* it must be replaced by a memory location or constant, if possible. SATL is replaced by repeated single bit shifts. SATH requires a bit test followed by a save of the high ACC and a store to the low ACC. SATH also requires a temporary memory location to implement the shift.

Table 10. Replacement Code for SATL and SATH Instructions

C5x Instruction	C2xx Replacement Code	Comments

`SATL`	`RPT   treg1`
	`SFR`
	`ROL`
		treg1 in memory
		ROL needed since repeat operates n+1 times

`SATH`	`BIT   treg1,11`
	`BCND  same,NTC`
	`SACH  temp`
	`LACC  temp`
	`same next instr...`
		treg1 and temp in memory on same DP

Temporary Registers (TREG1, TREG2)

The TREG1 and TREG2 registers do not exist on the C2xx, but the TREG register, which is called TREG0 on the C5x, emulates their usage.

BITT, LACT, ADDT, and SUBT:

The BITT, LACT, ADDT, and SUBT instructions operate exactly the same as on the C5x, except that they use TREG instead of TREG1 or TREG2. They are valid C2xx instructions and thus will not cause errors or warnings when assembled for the C2xx. However, in working C5x code, they must be preceded by a load of TREG1 or TREG2, which *will* cause an error when assembled for the C2xx.

Replacement

These instructions do not need to be replaced, but their usage should be checked to determine if they can use TREG or if an alternate method is needed. If more than one TREG is currently used in the C5x code, a context save and restore of TREG (see Appendix B) may be necessary to utilize it for this instruction. In this case, it is often more efficient to restructure this piece of code than to use one register for multiple functions.

Test Method on the C5x

Setting the C5x PMST bit TRM = 0 causes any load of TREG to also load the same value into TREG0, TREG1, and TREG2. Thus, the C2xx code will properly execute the affected instructions on a C5x. However, care must be taken to ensure that all TREG1 and TREG2 loads have been converted, or their values could be corrupted.

Addressing Mode Differences

Load/Store of Memory-Mapped Registers

The C5x makes most of its on-chip registers available at specific addresses on data page 0, allowing direct reading and writing of these registers without setting or modifying the data page pointer. The C2xx does not have this capability.

SAMM and LAMM Load/store the low accumulator from/to the specified address on data page 0, without explicitly changing the data page pointer. LAMM also sets the high accumulator to zero.

SMMR and LMMR Copy data from/to a data memory location (addressed by 16-bit constant “#addr”) to/from a memory-mapped register (addressed by lower 7 bits of dma). Note that these instructions do not load/store a constant, but rather to or from a constant address (usually a label).

Replacement

Of the C5x memory mapped registers, only the auxiliary registers (AR0-AR7), temporary register (TREG0), and DARAM block B2 exist in the same form on the C2xx. Access to any other register requires either emulation of that register with a memory location or system-level changes. Replacement code for many of these registers are described throughout this document.

An immediate value is often loaded to a memory-mapped register by first loading it into the accumulator. In this case, more efficient results can be obtained by directly loading the value into the C2xx register.

Each of these instructions can use indirect addressing to point to the memory-mapped register. In this case, the user must determine which register is being addressed, then use the proper replacement based on this determination.

Load/Store of Auxiliary Registers

The auxiliary registers can only be accessed through the LAR and SAR instructions. Simple replacement will require use of local memory or a change in the data page pointer.

Table 11. Replacement Code for Load/Store of Auxiliary Registers

C5x Instruction	C2xx Replacement Code	Comments

`SAMM  ARx`	`SACL  temp`
	`LAR   ARx,temp`
		temp in memory

`LAMM  ARx`	`SAR   ARx,temp`
	`LACL  temp`
		temp in memory

`SMMR  ARx,#addr`	`LDP   #addr`
	`SAR   ARx,addr`
		DP changed

`LMMR  ARx,#addr`	`LDP   #addr`
	`LAR   ARx,addr`
		DP changed

Load/Store of TREG

TREG can be loaded via the LT instruction but only read by moving its data through the accumulator.

Table 12. Replacement Code for Load/Store of TREG

C5x Instruction	C2xx Replacement Code	Comments

`SAMM  TREG0`	`SACL  temp`
	`LT    temp`
		temp in memory

`LAMM  TREG0`	`MPY   #1`
	`PAC`
	`AND   #0FFFFh`
		Preg changed
		Make sure product shift mode is set to zero (PM=00).
		AND instruction only required to ensure ACCH = 0

`SMMR TREG0,#addr`	`MPY   #1`
	`PAC`
	`LDP   #addr`
	`SACL  addr`
		Preg and DP changed
		Make sure product shift mode is set to zero (PM=00).

`LMMR TREG0,#addr`	`LDP   #addr`
	`LT    addr`
		DP changed

Load/Store of DARAM Block B2

Memory-mapped accesses to block B2 must be replaced by code that explicitly manages the data page pointer.

Table 13. Replacement Code for Load/Store of DRAM Block B2

C5x Instruction	C2xx Replacement Code	Comments

`SAMM  dma`	`LDP   #0`
	`SACL  dma`
		DP changed

`LAMM  dma`	`LDP   #0`
	`LACL  dma`
		DP changed

`SMMR  dma,#addr`	`LDP   #0`
	`BLDD  dma,#addr`
		DP changed

`LMMR  dma,#addr`	`LDP   #0`
	`BLDD  #addr,dma`
		DP changed

Circular Buffering

The C2xx family does not contain the C5x circular buffering hardware, including the five registers (CBSR1, CBER1, CBSR2, CBER2, and CBCR) used to configure this feature. Loading one or more of these registers indicates use of this capability.

Replacement with a Linear Buffer

Sometimes a linear buffer can be used instead of a circular one. For example, in a filter application, the C2xx automatic data movement (DMOV) capabilities can be used to implement the delay line rather than a circular buffer. Although this method is limited to use in on-chip memory, in some cases it is the most efficient.

Replacement with Bit-Reversed Addressing

The bit-reversed addressing capabilities of the C2xx can be used to implement a circular buffer, with the following limitations:

- ❑ The size of the buffer must be a power of two (2^n), although it can be used with any filter length.
- ❑ The buffer must be aligned so that the starting address of the buffer has *n* LSBs equal to zero.
- ❑ All pointer updates must use the bit-reversed update mode (*BRO+/-).

To set up this method, AR0 (used for indexed addressing) must be set to half the buffer size (2^{n-1}), which is actually a bit-reversed one. Another AR is used to access the buffer and can be initialized anywhere within the buffer. When the pointer is updated using bit-reversed mode, the arithmetic carries propagate to the right instead of the left, resulting in modulo arithmetic that keeps the pointer within the desired range. For more details, see reference [3].

Replacement with Manual Address Checks

The most direct replacement is to manually check the address after every modification of the AR associated with the circular buffer. This requires both a different setup and added code every time the circular buffer pointer is modified.

The following example illustrates one method to replace this code. This method requires that AR0 be available for use in the code to be changed. First, all of the setup code is replaced. Then, when the circular buffer is used, code is added to check for required pointer wrap-around. If ARP modification is included in the code line utilizing circular buffering, it must be moved to the end of the added code.

Table 14. Replacement Code Example Using Manual Address Checks

C5x Code	C2xx Replacement Code
LACC #bstart	LACC #bstart ;start of buffer
SAMM CBSR1 ;start of buffer	ADD #blength
SAMM AR2 ;AR2 -> bstart	SACL temp ;(end of buffer)+1
ADD #blength-1	LAR AR0,temp ;used for compare
SAMM CBER1 ;end of buffer	
SPLK #000A,CBCR ;enable buffer #1 and	
;associate AR2 with it	
. . .	. . .
MAR *,AR2	MAR *,AR2
ADD *+ ;use circular buffer (AR2)	ADD *+ ;use circular buf (AR2)
	CMPR 00 ;if past end, TC=1
	BCND n1,NTC ;if not past, continue
	LAR AR2,#bstart ;if past, reset AR2
MPY *+,0,AR6 ;use circular buffer (AR2)	n1 MPY *+ ;use circular buf (AR2)
	CMPR 00 ;if past end, TC=1
	BCND n2,NTC ;if not past, continue
	LAR AR2,#bstart ;if past, reset AR2
	n2 MAR *,AR6 ;do ARP modification
ADD *+ ;not a circular buffer (AR6)	ADD *+ ;not a circular buf (AR6)
etc.	etc.

Auxiliary Register Compare Register (ARCR)

The Auxiliary Register Compare Register (ARCR) does not exist on the C2xx but is emulated with the AR0 register. ARCR is used solely by the CMPR instruction, which performs a comparison of the specified AR with ARCR. Its usage in the CMPR instruction will not cause assembler errors or warnings, although an error will occur when attempting to load it.

Replacement

Just load AR0 instead of ARCR and its usage should remain the same. However, since AR0 can be used for other purposes, you should make sure that its value is not corrupted before its use.

Test method on C5x

Setting the C5x PMST bit NDX = 0 causes any load of AR0 to also load the same value into ARCR and INDX. Thus, the C2xx code will properly execute the CMPR instruction on a C5x. However, care must be taken to ensure that all ARCR and INDX loads have been converted or their values can be corrupted.

Index Register (INDX)

The Index Register (INDX) does not exist on the C2xx but is emulated with the AR0 register. INDX is used for indirect indexed addressing modes (*0, *0+, *0-). Its usage in these address modes will not cause assembler errors or warnings, although an error will occur when attempting to load it.

Replacement

Just load AR0 instead of INDX and its usage should remain the same. However, since AR0 can be used for other purposes, you should make sure that its value is not corrupted before its use.

Test method on C5x

Setting the C5x PMST bit NDX = 0 causes any load of AR0 to also load the same value into ARCR and INDX. Thus, the C2xx code properly executes indexed addressing modes on a C5x. However, care must be taken to ensure that all ARCR and INDX loads have been converted, or their values can be corrupted.

Dynamic Addressing Modes via the Block Move Address Register (BMAR)

For instructions that use two operands, the C2xx can only specify one of these at run time. The second operand must be hard-coded; that is, its address must be specified as an immediate value. The C5x uses the Block Move Address Register (BMAR) to enable dynamic, or run-time, specification of the second operand for the following instructions:

MADS, MADD

The MADS and MADD instructions multiply a data memory value (*dma) by a program memory value (*pma in BMAR) and accumulate the previous product. MADD also performs a data move of *dma to *(dma+1). When these instructions are repeated, they become single cycle and allow the program memory address to be automatically incremented by the prefetch counter (part of the program counter).

Replacement

The most efficient method is to use the MAC or MACD instruction with the pma hard-coded. This method retains the single-cycle execution with auto-pma increment in repeat mode. However, hard-coding the address of the pma makes the routine usable for only one set of coefficients. Thus, if MADD/MADS is used in a routine called with differing coefficient tables (different values in BMAR) then separate routines must be created for each instance.

The dynamic addressing capability can be maintained by using the LTA[LTD] and MPY instructions. This method performs the same functions, but requires an additional cycles.

Table 15 shows both replacement methods discussed above.

Table 15. Replacement Code Example for MADS and MADD Instructions

C5x Code	C2xx Replacement Code	Comments
LACL #coeff SAMM BMAR LAR AR2,#d_end MAR *,AR2 RPT #n-1 **MADD** *- APAC SACL output	LAR AR2,#d_end MAR *,AR2 RPT #n-1 MACD coeff,*- APAC SACL output	Still single-cycle in repeat mode Must create separate instance for each coefficient set Use MAC to replace MADS, MACD to replace MADD
	LAR AR7,#n-1 LAR AR2,#d_end LAR AR3,#coeff LAR *,AR2 lp LTA *-,AR3 MPY *+,AR7 BANZ lp,AR2 APAC SACL output	Retains dynamic addressing 6-cycle loop (vs. 1-cycle) AR3 and AR7 changed Use LTA in MADS replacement, LTD in MADD replacement

BLDD, BLDP, and BLPD

The BLDD, BLDP, and BLPD instructions copy data between memory locations. The BMAR can be used to specify the second memory location, allowing dynamic addressing. If BMAR is not used, an immediate operand must specify the second memory location. Single-cycle execution with automatic address increment is obtained when in repeat mode. The BLDD/BLPD versions with long immediate operands **are valid** on the C2xx, but all other versions will be flagged as errors.

Replacement

For BLDD and BLPD, the simplest replacement is to use the long-immediate version of the instruction. However, this method requires hard coding of one of the addresses, preventing its use for multiple routines. To maintain the dynamic addressing capability, TBLR (copy *pma to *dma) can be used to replace BLPD, and TBLW (copy *dma to *pma) can be used to replace BLDP. These instructions achieve single-cycle execution in repeat mode but require the accumulator for the address of the second memory location. The following table shows examples of all three instructions and their replacements.

Table 16. Replacement Code Examples for BLDD, BLDP, and BLPD Instructions

C5x Code	C2xx Replacement Code	Comments
LAR AR0,#dest MAR *,AR0 LACL #src SAMM BMAR RPT #n-1 **BLDD** BMAR,*+	LAR AR0,#dest MAR *,AR0 RPT #n-1 BLDD #src,*+	Still single-cycle in repeat mode Must create separate instance for each buffer copy
LAR AR0,#src MAR *,AR0 LACL #dest SAMM BMAR RPT #n-1 **BLDP** *+	LAR AR0,#src MAR *,AR0 LACL #dest RPT #n-1 TBLW *+	Still single-cycle in repeat mode Maintains dynamic addressing ACC changed
LAR AR0,#dest MAR *,AR0 LACL #src SAMM BMAR RPT #n-1 **BLPD** BMAR,*+	LAR AR0,#dest MAR *,AR0 RPT #n-1 BLPD #src,*+	Still single-cycle in repeat mode Must create separate instance for each buffer copy
	LAR AR0,#dest MAR *,AR0 LACL #src RPT #n-1 TBLR *+	Still single-cycle in repeat mode Maintains dynamic addressing ACC changed

I/O Data Moves

The C5x allows a subset of its I/O memory locations to be accessed as memory-mapped registers for the load/store and IN/OUT instructions. These are addressed as PA0-PA15 in the C5x, which map to addresses 0x50-0x5F. Assembly of "PAx" in the C2xx IN and OUT instructions produce the absolute addresses 0-15 rather than the memory-mapped locations. Additionally, (in assembler version 6.60) **the use of “PAx” will not produce an error or warning!** IN/OUT instructions using a normal 16-bit address do not require modification.

Replacement

Use absolute addressing for all IN and OUT instructions. Direct access via the memory-mapped load/store instructions (LAMM, SAMM, LMMR, SMMR) requires substitution of the appropriate IN or OUT instruction at the correct I/O address.

Table 17. Replacement Code for I/O Data Moves

C5x Code	C2xx Replacement Code	Comments
`IN  data,PAx`	`IN  data,50h + x`	PAx = address 50h + x 0 ≤ x ≤ 15
`OUT data,PAx`	`OUT data,50h + x`	PAx = address 50h + x 0 ≤ x ≤ 15

Program Control Differences

Conditional Execution (XC)

The C5x allows efficient implementation of “if-then-else” constructs via the XC instruction. This instruction eliminates branches by either executing the next 1 or 2 instructions or inserting NOPs. The C2xx does not have this capability.

Replacement

The most efficient replacement utilizes conditional branching based on the opposite condition than used in the XC instruction. BIO is the only condition without an opposite; thus, it requires an extra branch.

Table 18. Replacement Code for Conditional Execution

C5x Code	C2xx Replacement Code	Conditions

`XC  2,cond`
` instruction_1`
` instruction_2`
`next_instruction`
	`   BCND A1,!cond`
	`    instruction_1`
	`    instruction_2`
	`A1: next_instruction`
		EQ ⇔ NEQ
		LT ⇔ GEQ
		GT ⇔ LEQ
		C ⇔ NC
		OV ⇔ NOV
		TC ⇔ NTC

`XC  2,BIO`
` instruction_1`
` instruction_2`
`next_instruction`
	`   BCND A1,BIO`
	`   B    A2`
	`A1: instruction_1`
	`    instruction_2`
	`A2: next_instruction`	

Delayed Branches, Calls, and Returns (BD, BACCD, BANZD, BCNDD, CALAD, CALLD, CCD, RETD, and RETCD)

Since branches flush the pipeline, the C5x improves code performance by allowing two instructions to be performed before the branch is taken. The C2xx does not have this feature.

Replacement

Use the standard non-delayed versions of these instructions.

Block Repeat (RPTB)

The C5x contains dedicated hardware, including a loop counter called the Block Repeat Counter Register (BRCR), which allows it to perform zero-overhead block repeats. The C2xx does not have the hardware or the BRCR, and thus requires overhead for looping.

Replacement

The most efficient method is to unroll the loop, eliminating any overhead. However, when this is not possible, code can be inserted to perform the loop control. The loop counter (BRCR) must be replaced with an AR and a branch must be placed at the end of the loop. The code within the loop must be modified to properly manage the ARP. Table 19 shows how this is done for a typical loop.

Table 19. Replacement Code for Block Repeats

C5x Code	C2xx Replacement Code	Comments
`MAR  *,AR1`	`MAR  *,AR1`	One AR changed
`LACC #n-1`	`LAR  AR7,#n-1`	4 extra cycles in loop
`SAMM BRCR`		
`RPTB end-1`		
`ADD  *+`	`st:  ADD  *+`	Must manage ARP to use additional AR for loop counter
`SACL temp`	`SACL temp`	
`AND  *-`	`AND  *-,AR7`	
	`BANZ st,AR1`	
`end: next_instr`	`end: next_instr`	

Interrupts

Shadow Registers

The C5x has shadow registers to automatically store the CPU context on interrupt traps. The following registers are shadowed: ACC, ACCB, PREG, ST0, ST1, PMST, ARCR, INDX, TREG0, TREG1, and TREG2. The C2xx does not contain these shadow registers.

Replacement

The most direct replacement is for the interrupt service routine (ISR) to manually save these registers. However, *only* those registers that are modified in the ISR need to be saved. This is typically only a few of the total number. See Appendix B for more details on context saves and restores. Because ISRs are typically important for proper real-time system operation, restructuring of interrupts to minimize the overhead should be given thorough study.

RETE and RETI

The RETE and RETI instructions implement specialized returns from an interrupt. As opposed to normal returns, they pop the shadow registers as well as copy the top of the stack to the program counter. In addition, RETE automatically enables interrupts (sets INTM=0).

Replacement

Because the shadow registers are not present on the C2xx, use a simple RET and manually restore any changed registers as outlined in the section, *Interrupts*. Manual interrupt enable is also required to replace RETE.

Table 20. Replacement Code for RETE and RETI Specialized Return Instructions

C5x Code	C2xx Replacement Code	Comments
`RETI`	`RET`	Manually restore changed context

`RETE`	`SETC  INTM`
	`RET`
		Manually restore changed context

Control and Status Bits

The C5x has four control and status registers: PMST, CBCR, ST0, and ST1. Of these, only ST0 and ST1 are available on the C2xx. The C2xx's ST0 and ST1 are identical except for the hold-mode bit, HM (ST1, bit 6). Attempts to set or clear non-existent bits or registers will result in an error.

Replacement

These differences require system-level study to determine replacement strategy.

Idle Mode (IDLE2)

The C5x and C2xx both offer low-power modes of operation in which the CPU activities are halted but the peripherals remain active. The IDLE instruction places the devices in this state. The C5x offers an additional power mode, IDLE2, which shuts down the entire device. This mode is not available on the C2xx.

Replacement

Use IDLE, which is lowest power mode available.

Automatic Zero of Accumulator and P Register (ZAP, ZPR, RPTZ)

These instructions automatically zero the accumulator and product register (Preg) on the C5x. This capability does not exist on the C2xx.

Replacement

The accumulator and/or product register must be manually cleared.

Table 21. Replacement Code for ZAP, ZPR, and RPTZ Instructions

C5x Code	C2xx Replacement Code	Comments

`ZAP`	`MPY   #0`
	`PAC`
		Zero Preg first, then move to ACC.

`ZPR`	`MPY   #0`	Zero Preg only.

`RPTZ   #lk`	`MPY   #0`
	`PAC`
	`RPT   #lk`
		Zero Preg and ACC, then repeat.
		See *Immediate Operands* for RPT concerns.

Immediate Operands

Except for the long immediate operand format, the following two instructions are supported by the C2xx. The C5x allows 16-bit immediate operands for these instructions:

MPY #lk Immediate operand restricted to 13 bits – values from -1000h (-4096) to 0FFFh (4095)

RPT #lk Immediate operand restricted to 8 bits (largest value is 255 or FFh)

These instructions truncate any 16-bit operands, resulting in warnings but not errors when assembled for the C2xx. When modified for the C2xx, they assemble correctly for the C5x.

Replacement

If a 16-bit immediate operand is truly required, the value should be stored in a memory location, which can then be used as the operand. A temporary variable on the current data page should be used, if available. Otherwise, the data page pointer must be saved, set, and restored afterward.

Table 22. Replacement Code to Support 16-Bit Immediate Operand

C5x Code	C2xx Replacement Code	Comments

`MPY  #7FFFh`	`SPLK #7FFFh,temp`
	`MPY  temp`
		temp in memory (watch DP)

`RPT  #3456h`	`SPLK #3456h,temp`
	`RPT  temp`
		temp in memory (watch DP)

Appendix A. Summary of Instruction Replacements

The following tables list replacements for all unsupported instructions and detail their cost in cycles and memory. The cycle costs assume that local data memory is used (same DP) and that context save and restore is not required. Use of memory can incur additional cycle costs due to data page changes or extra AR setup. Context saves and restores will also add cycle and memory costs.

Table 23 lists C5x instructions that are individually replaceable. Table 24 lists those C5x instructions and operations that require some surrounding context to be considered in the replacement. Any instruction or operation not listed in these tables requires system-level considerations.

Table 23. Directly Replaceable C5x Instructions

C5x Instruction	C2xx Replacement Code	Changed Context	Extra Cycles	Extra Memory	Comments
APL [#lk,]dma	LACC dma AND [#lk][dbmr] SACL dma	ACC	2	None or 1 (dbmr)	Does not set TC
OPL [#lk,]dma	LACC dma OR [#lk][dbmr] SACL dma	ACC	2	None or 1 (dbmr)	Does not set TC
XPL [#lk,]dma	LACC dma XOR [#lk][dbmr] SACL dma	ACC	2	None or 1 (dbmr)	Does not set TC
CPL [#lk,]dma	LAR AR0,[#lk][dbmr] LAR ARx,dma MAR *,ARx CMPR 00	AR0 ARx ARP	5	None or 1 (dbmr)	Sets TC
	LACC [#lk][dbmr] SUB dma	ACC C	1	None or 1 (dbmr)	Sets ACC status (EQ or NEQ), not TC
LACB	LACC accbh,16 OR accbl		1	2 (accbh, accbl)	
SACB	SACH accbh SACL accbl		1	2 (accbh, accbl)	

C5x Instruction	C2xx Replacement Code	Changed Context	Extra Cycles	Extra Memory	Comments
EXAR	SACH temp2 SACL temp1 LACC accbh,16 OR accbl BLDD #temp2,accbh BLDD #temp1,accbl		9	4 (accbh, accbl, temp1, temp2)	
ADCB	ADDC accbl ADD accbh,16		1	2 (accbh, accbl)	
ADDB	ADDS accbl ADD accbh,16		1	2 (accbh, accbl)	
SBB	SUBS accbl SUB accbh,16		1	2 (accbh, accbl)	
SBBB	SUBB accbl SUB accbh,16		1	2 (accbh, accbl)	
ANDB	SACH temp1 AND accbl SACL temp2 LACC temp1 AND accbh SACL temp1 LACC temp1,16 OR temp2		7	4 (accbh, accbl, temp1, temp2)	
ORB	SACH temp1 OR accbl SACL temp2 LACC temp1 OR accbh SACL temp1 LACC temp1,16 OR temp2		7	4 (accbh, accbl, temp1, temp2)	
XORB	SACH temp1 XOR accbl SACL temp2 LACC temp1 XOR accbh SACL temp1 LACC temp1,16 OR temp2		7	4 (accbh, accbl, temp1, temp2)	

C5x Instruction	C2xx Replacement Code	Changed Context	Extra Cycles	Extra Memory	Comments
CRGT	SUBS accbl SUB accbh,16 BCND abig,GEQ LACC accbh,16 OR accbl CLRC C B next abig ADDS accbl ADD accbh,16 SACH accbh SACL accbl SETC C next		10	2 (accbh, accbl)	
CRLT	SUBS accbl SUB accbh,16 BCND asml,LT LACC accbh,16 OR accbl CLRC C B next asml ADDS accbl ADD accbh,16 SACH accbh SACL accbl SETC C next		10	2 (accbh, accbl)	
ROLB	SACH temp1 SACL temp2 LACC accbh,16 OR accbl ROL SACH accbh SACL accbl LACC temp1,16 OR temp2 ROL		9	4 (accbh, accbl, temp1, temp2)	

C5x Instruction	C2xx Replacement Code	Changed Context	Extra Cycles	Extra Memory	Comments

`RORB`	`ROR`
	`SACH temp1`
	`SACL temp2`
	`LACC accbh,16`
	`OR accbl`
	`ROR`
	`SACH accbh`
	`SACL accbl`
	`LACC temp1,16`
	`OR temp2`
			9	4 (accbh, accbl, temp1, temp2)	

`SFLB`	`SACH temp1`
	`SACL temp2`
	`LACC accbh,16`
	`OR accbl`
	`SFL`
	`SACH accbh`
	`SACL accbl`
	`LACC temp1,16`
	`OR temp2`
	`ROL`
			9	4 (accbh, accbl, temp1, temp2)	

`SFRB`	`SFR`
	`SACH temp1`
	`SACL temp2`
	`LACC accbh,16`
	`OR accbl`
	`ROR`
	`SACH accbh`
	`SACL accbl`
	`LACC temp1,16`
	`OR temp2`
			9	4 (accbh, accbl, temp1, temp2)	

`BSAR n`	`RPT #n-1`
	` SFR`
			n	None	n-1 single-bit shifts

	`SFR` `SFR` `SFR` …		n-1	None	Unrolled loop: do n-1 individual shifts

`SATL`	`RPT treg1`
	` SFR`
	`ROL`
			n+1	1 (treg1)	n is shift count in treg1

C5x Instruction	C2xx Replacement Code	Changed Context	Extra Cycles	Extra Memory	Comments

`SATH`	`BIT treg1,11`
	`BCND same,NTC`
	`SACH temp`
	`LACC temp`
	`same next instr…`
			4	2 (treg1, temp)	n is shift count in treg1

`SAMM ARx`	`SACL temp`
	`LAR ARx,temp`
			2	1 (temp)	

`LAMM ARx`	`SAR ARx,temp`
	`LACL temp`
			1	1 (temp)	

`SMMR ARx,#addr`	`LDP #addr`
	`SAR ARx,addr`
		DP	1	None	

`LMMR ARx,#addr`	`LDP #addr`
	`LAR ARx,addr`
		DP	2	None	

`SAMM TREG0`	`SACL temp`
	`LT temp`
			1	1 (temp)	

`LAMM TREG0`	`MPY #1`
	`PAC`
	`AND #0FFFFh`
		Preg	3 (1)	None	Make sure product shift mode is set to zero (PM=00).
					AND instruction only required to ensure ACCH = 0

`SMMR TREG0,#addr`	`MPY #1`
	`PAC`
	`LDP #addr`
	`SACL addr`
		Preg
		DP
			3	None	Make sure product shift mode is set to zero (PM=00)

`LMMR TREG0,#addr`	`LDP #addr`
	`LT addr`
		DP	2	None	

`SAMM B2dma`	`LDP #0`
	`SACL B2dma`
		DP	2	None	

`LAMM B2dma`	`LDP #0`
	`LACL B2dma`
		DP	2	None	

`SMMR B2dma,#addr`	`LDP #0`
	`BLDD B2dma,#addr`
		DP	3	None	

`LMMR B2dma,#addr`	`LDP #0`
	`BLDD #addr,B2dma`
		DP	3	None	

C5x Instruction	C2xx Replacement Code	Changed Context	Extra Cycles	Extra Memory	Comments

`ZAP`	`MPY #0`
	`PAC`
			1	None	

`ZPR`	`MPY #0`		0	None	

`RPTZ #lk`	`MPY #0`
	`PAC`
	`RPT #lk`
			2	None	Make sure RPT is converted properly

`MPY #lk`
`(> 13 bits)`
	`SPLK #lk,temp`
	`MPY temp`
			1	1 (temp)	Warning only

`RPT #lk`
`(> 8 bits)`
	`SPLK #lk,temp`
	`RPT temp`
			1	1 (temp)	Warning only

Table 24. Non-Directly Replaceable C5x Operations

C5x Operation	C2xx Replacement	Cost	Comments
Circular buffering	Linear buffer using DMOV	None	Can't be used in external memory
	Bit-reversed addressing	AR0 usage	Buffer must be size 2^n and properly aligned
	Manual address checks	5 or 6 cycles per access	High overhead
ARCR usage	Use AR0	AR0 usage	Make sure AR0 isn't being used for other purposes
INDX usage	Use AR0	AR0 usage	Make sure AR0 isn't being used for other purposes
TREG1 or TREG2 usage (BITT, LACT, ADDT, and SUBT instructions)	Use Treg	Treg usage	Make sure Treg isn't being used for other purposes
MADS & MADD instructions	Use MAC[D] with long-immediate addressing	None	Single-cycle in repeat mode, but loses dynamic addressing
	Use LTA[D] and MPY	2 AR's 4 extra cycles per loop	Retains dynamic addressing, but has high overhead
BLDD instruction using BMAR	Use BLDD with long-immediate addressing	None	Single-cycle in repeat mode, but loses dynamic addressing
BLDP instruction using BMAR	Use TBLW	Alters ACC	Retains dynamic-addressing; single-cycle in repeat mode
BLPD instruction using BMAR	Use TBLR	Alters ACC	Retains dynamic-addressing; single-cycle in repeat mode
	Use BLPD with long-immediate addressing	None	Single-cycle in repeat mode, but loses dynamic addressing

C5x Operation	C2xx Replacement	Cost	Comments
IN or OUT instruction using PAx notation	Use physical address	None	Not detected by assembler (ver 6.60)
XC instruction	Use BCND with opposite condition	1 extra cycle (3 extra for BIO)	
Delayed branches, calls, and returns	Use normal branches, calls, and returns	2 extra cycles	
Block repeats	Unroll the loop	None	More code space required, but actually faster
	Add BANZ to end of loop	4 extra cycles 1 AR altered	
Automatic context save on interrupts	Manually save and restore context	Depends on ISR	Typically minimal impact
RETE and RETI instructions	Use RET and manually enable INTM	1 cycle for INTM enable	Must save and restore context
Control and Status register manipulation	System-dependent	Depends	
IDLE2 mode	Use IDLE	None	Can’t shut down peripherals

Appendix B. Context Save and Restore

In some cases the current context must be saved and restored to ensure that the modifications do not alter values used by other code sections. This section does not deal with replacement code – only methods to ensure that the other replacements do not alter the current context.

Accumulator

Two temporary data locations are required to save the current accumulator value. The most efficient method is to use locations on the current data page. If a separate data page is required, the data page must also be set and restored.

```
save:      SACH  temp1         ;save high ACC
           SACL  temp2         ;save low ACC

restore:   LACC  temp1,16      ;restore high ACC
           OR    temp2         ;restore low ACC
```

Data Page Pointer

The data page pointer is stored as the lower 9 bits of Status Register 0 (ST0). Thus to save and restore the DP value, it is easiest to save and restore ST0:

```
save:      SST   #0,temp      ;save @temp on DP #0

restore:   LDP   #0           ;manually set to DP #0
           LDP   temp         ;restore ST0
```

Note that the save is done automatically to DP #0, but the restore must set the DP. Either SST or LST can also be used with indirect addressing to use any data page. Since ST0 also contains the ARP, OV, and OVM bits, these can be restored as well by the "LST #0,temp" instruction. The INTM bit, although in ST0, is not set by the LST instruction.

Auxiliary Registers

To save the contents of an auxiliary register, a temporary data location is required. The most efficient method is to use a location on the current data page. If a separate data page is required, the data page must also be set and restored.

```
save:     SAR   ARx,temp    ;store ARx

restore:  LAR   ARx,temp    ;restore ARx
```

T Register and P Register

Since the only method to save Treg is through the Preg, these registers will typically be saved and restored together. The Preg should be saved first, followed by the Treg. As with other context saves, the most efficient method uses memory on the current data page. If a separate data page is required, the data page must also be set and restored.

```
save:       SPM   0       ; set for no shift on Preg xfer
            SPH   temp1   ; push PregH
            SPL   temp2   ; push PregL
            MPY   #1      ; Treg -> Preg
            SPL   temp3   ; push Treg

restore:    LT    temp2   ; pop PregL
            MPY   #1      ; restore PregL
            LPH   temp1   ; pop PregH
            LT    temp3   ; pop Treg
```

Interrupt Context Save and Restore

Of the registers automatically saved by the C5x when servicing an interrupt, the only ones present on the C2xx are the ACC, PREG, ST0, ST1, and TREG. Thus, only these must be saved and restored in addition to whatever other context is saved by the C5x routine (such as any modified AR's). The following outlines one method for saving and restoring these:

```
* Assume AR7 has been initialized as the stack pointer
save:       MAR   *,AR7   ; 7->ARP, ARP->ARB
            MAR   *-      ; point to top of stack
            SST   #0,*-   ; push ST0
            SST   #1,*-   ; push ST1
            SACH  *-      ; push ACCH
            SACL  *-      ; push ACCL
            SPM   0       ; no shift on Preg xfer
            SPH   *-      ; push PregH
            SPL   *-      ; push PregL
            MPY   #1      ; Treg -> Preg
            SPL   *-      ; push Treg
            ...
```

```
* Assume AR7 has been initialized as the stack pointer
restore:    MAR   *,AR7    ; 7->ARP, ARP->ARB
            MAR   *+       ; point to bottom of stack
            MAR   *+       ; skip Treg
            LT    *-       ; pop PregL
            MPY   #1       ; restore PregL
            LT    *+       ; pop Treg
            MAR   *+       ; skip PregL
            LPH   *+       ; pop PregH
            LACC  *+       ; pop ACCL
            ADD   *+,16    ; pop ACCH
            LST   #1,*+    ; pop ST1
            LST   #0,*+    ; pop ST0
            EINT           ; enable interrupts
            RET
```

References

[1] *TMS320C2xx User's Guide,* Digital Signal Processing Products, Texas Instruments, 1997.

[2] *TMS320C5x User's Guide,* Digital Signal Processing Products, Texas Instruments, 1997.

[3] Hendrix, Henry, "Implementing circular buffers with bit-reversed addressing", application report, Texas Instruments, Inc., 1997