Freescale Semiconductor, Inc.

Freescale Semiconductor

APR22/D

Application Conversion from the DSP56100 Family to the DSP56300/600 Families

by
Tom Zudock

© Freescale Semiconductor, Inc., 2004. All rights reserved.

**For More Information On This
Go to: www.freescale**

freescale™
semiconductor

Freescale Semiconductor, Inc.

Freescale Semiconductor, Inc. (vertical, left margin)

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

For More Information On This Product,
Go to: www.freescale.com

TABLE OF CONTENTS

DSP56100 to DSP56300/600 Conversion   **iii**
For More Information On This Product,
Go to: www.freescale.com

LIST OF TABLES

SECTION 1

INTRODUCTION

This document summarizes information needed by a user to estimate the effort required to convert an application from the DSP56100 to the DSP56300/600 and the details involved in translating the software.

1.1   SCOPE

The Motorola DSP56100 family and DSP56300/600 families are similar in many ways, but the parts are not fully compatible. Hence, using a DSP56300/600 in place of a DSP56100 requires modification to the software and the hardware. This document summarizes information needed by a user to estimate the effort required to convert an application and the details involved in translating the software

There are many compatibility issues that are beyond the scope of this document. This document focuses on differences that relate to software translation. Attempting to include all the differences would distract from this intent. For details on any aspect not included in this document, please refer to the DSP56100, DSP56300, and DSP56600 family manuals. For details on peripherals or specifics for a given part in the family, refer to the manual or data sheet for the DSP product of interest (e.g., DSP56302, DSP56303, DSP56603, etc.).

1.2   TERMS AND DEFINITIONS

Definitions for abbreviations or acronyms used throughout the document are provided in the following list.

AGU	Address Generation Unit
CCR	Condition Code Register
Data ALU	Data Arithmetic Logic Unit
DSP	Digital Signal Processor
LSP	Least Significant Portion
I/O	Input/Output
MIPS	Million Instructions Per Second
MSP	Most Significant Portion
RAM	Random Access Memory
ROM	Read Only Memory
SR	Status Register

SECTION 2

DSP56100 AND DSP56300/DSP56600 DIFFERENCES

The primary difference in architecture between the DSP56100 and the DSP56300/600 families lies in the organization of data memory.

Architecture

Although there are more similarities than differences between the DSP56100 and the DSP56300/600 families, some of the differences are fundamental and require careful attention. In this section, the differences are organized into three categories: the architecture, the addressing modes, and the instruction set. A brief examination of these categories will help an engineer estimate the amount of effort that will be required to translate an application.

2.1 ARCHITECTURE

2.1.1 Memory Organization

The primary difference in architecture between the DSP56100 and the DSP56300/600 families lies in the organization of data memory. DSP56100 family parts have a single data memory, X memory, that supports dual accessed reads. In the DSP56300/600 family, data memory is divided into X and Y memory, which can be simultaneously read or written. Since the DSP56300/600 does not allow dual read access to X or Y memory *individually*, dual accesses to X memory in the DSP56100 software must be split into accesses to X memory in parallel with accesses to Y memory.

The instruction width on the DSP56300/600 is 24 bits as opposed to 16 bits on the DSP56100. If an application will require external RAM, the larger width of the instructions must be accounted for in the external RAM. If the final application will reside internal on ROM implementations, this is of little consequence.

2.1.2 Data Arithmetic Logic Unit (Data ALU)

The DSP56100 is a 16-bit machine while the DSP56300 is a 24-bit machine. To generate bit-exact output using a DSP56300 family part, 16-bit Arithmetic mode must be used. All of the code examples in this document assume that if the DSP56300 is the target, it will be executing the code in 16-bit Arithmetic mode. The DSP56600 is a 16-bit machine, so this is not an issue.

Although Data ALUs provide the same register set, they differ in some of the mathematical operations they provide. The DSP56100 Data ALU allows the Most Significant Portion (MSP) of the accumulator to be used as an argument for an integer multiply accumulate, whereas the DSP56300/600 does not. Shifting capabilities also differ. The DSP56100 provides single bit shifting and shifting in larger fixed increments (ASR4, ASR16, ASL4). The DSP56300/600 Data ALU includes

For More Information On This Product,
Go to: www.freescale.com

a barrel shifter, which allows shifting in any increment over the range of the accumulator size. This also provides single cycle normalization, which is useful for maintaining dynamic range efficiently. Finally, the DSP56100 supports bit-field operations with immediate data. On the DSP56300/600, this must be performed using logical ANDs, ORs, or individual bit-position operations (i.e., BCLR).

2.1.3   Hardware Stack

Both processor families provide a hardware stack to support nested DO loops, subroutine calls, and interrupt processing. On the DSP56100, the stack is fifteen levels deep while the DSP56300/600 provides sixteen levels. Additionally, the DSP56300/600 supports stack extension. Because the DSP56300/600 provides substantially more (Million Instructions Per Second) MIPS, it is possible that applications executed on a single DSP56300/600 will be the consolidation of multiple single DSP applications. This may result in a need for a single DSP56300/600 to support more interrupts and increasingly complex software. Often, this software requires more stack space. Stack extension fills that void.

2.2   ADDRESSING MODES

Although there are several subtle differences between the AGUs, the DSP56100 addressing mode that will need the most attention for conversion is upper word accumulator addressing (e.g., X:(A1) or X:(B1)). As due compensation for lack of this addressing mode, the DSP56300/600 has twice as many address register sets, eight total. In the majority of cases (those using simple integer addition), an extra address register provides a simple solution.

One distinct advantage of the DSP56300/600 is that it allows more address register combinations for parallel memory accesses. When performing a parallel read on the DSP56100, one of the address registers is required to be R3. On the DSP56300/600, any of the R0–R3 registers may be used in parallel memory accesses with any of the R4–R7 registers. The DSP56300/600 additionally supports multiple wrap around modulo addressing. In this mode, if an index offset is used that is larger than the buffer modulus (as dictated by the value in the Mn register), the address is properly calculated as having wrapped around multiple times.

The table below summarizes differences in the addressing modes. It is a combination of the tables taken from the DSP56100 and DSP56300/600 family manuals. Check the notes below the table for a full description. The table clearly indicates that there are far more new options with DSP56300/600 addressing modes (marked with an N)

Addressing Modes

than those that are unsupported (marked with a U). The modes common to both are marked with B. Hence, there are sufficient options on the DSP56300/600 to work around the unsupported DSP56100 addressing modes.

Table 2-1 Comparison of DSP56100 and DSP56300/600 Addressing Modes

Addressing Mode	Operand Reference										Syntax
	S	C	D	A	P	X	Y	L	XY	XX	—
Register Direct											
Data or Control Register	U	B	B	—	—	—	—	—	—	—	—
Address Register Rn	—	—	—	B	—	—	—	—	—	—	—
Address Modifier Register Mn	—	—	U	N	—	—	—	—	—	—	—
Address Offset Register Nn	—	—	—	B	—	—	—	—	—	—	—
Address Register Indirect											
No Update	—	—	—	—	B	B	N	N	N	—	(Rn)
Postincrement by 1	—	—	—	—	B	B	N	N	N	U	(Rn)+
Postdecrement by 1	—	—	—	—	B	B	N	N	N	—	(Rn)–
Postincrement by Offset Nn	—	—	—	—	B	B	N	N	N	U	(Rn) + Nn
Postdecrement by Offset Nn	—	—	—	—	N	N	N	N	—	—	(Rn) – Nn
Indexed by Offset Nn	—	—	—	—	N	B	N	N	—	—	(Rn + Nn)
Predecrement by 1	—	—	—	—	N	B	N	N	—	—	–(Rn)
Short Displacement	—	—	—	—	—	B	N	N	—	—	(Rn + displ)
Long Displacement	—	—	—	—	—	N	N	N	—	—	(Rn + displ)
PC Relative	—	—	—	—	—	—	—	—	—	—	
Short Displacement PC Relative	—	U	—	—	B	—	—	—	—	—	(PC + displ)
Long Displacement PC Relative	—	U	—	—	N	—	—	—	—	—	(PC + displ)
Address Register	—	U	—	U	N	—	—	—	—	—	(PC + Rn)
Special											

Table 2-1 Comparison of DSP56100 and DSP56300/600 Addressing Modes (Continued)

Addressing Mode	Operand Reference										Syntax
	S	**C**	**D**	**A**	**P**	**X**	**Y**	**L**	**XY**	**XX**	—
Upper Word Accumulator	—	—	—	—	—	U	—	—	—	—	—
Immediate Short Data	—	—	—	—	B	—	—	—	—	—	—
Immediate Long Data	—	—	—	—	B	—	—	—	—	—	—
Absolute Address	—	—	—	—	B	B	N	N	—	—	—
Absolute Short Address	—	—	—	—	U	B	N	N	—	—	—
Short Jump Address	—	—	—	—	B	—	—	—	—	—	—
I/O short Address	—	—	—	—	—	B	N	—	—	—	—
Implicit	B	B	—	—	B	—	—	—	—	—	—

Note:

S = System Stack Reference
C = Program Control Unit Register Reference
D = Data ALU Register Reference
A = Address ALU Reference
P = Program Memory Reference
X = X Memory Reference
Y = Y Memory Reference
L = L Memory Reference
XY = XY Memory Reference
XX = XX Memory Reference
U = Unsupported by DSP56300/600
B = Supported by both the DSP56100 and the DSP56300/600
N = New on DSP56300/600 or unsupported by the DSP56100

2.3 INSTRUCTION SET

Translating DSP56100 software requires replacing unsupported instructions with their DSP56300/600 equivalents. Additionally, some instructions supported on both processors should be replaced with more efficient DSP56300/600 versions. For

example, repeated shift operations can be replaced by an instruction that uses the barrel shifter.

The table that follows summarizes the differences in the instruction sets of all three processors. The left column contains instructions that will need to be converted to DSP56300/600 equivalents, while the right column shows additional instructions provided by the DSP56300/600. Footnotes indicate additional information regarding specific instructions.

Table 2-2 DSP56100, DSP56300, and DSP56600 Instruction Set Differences

DSP56100 Instructions Unsupported or Modified by the DSP56300/600	Instructions Added by Using the DSP56300/600
ADC[1]	
	ADDL
	ADDR
ASL4	
ASR4	
ASR16	
	BCHG
	BCLR
BFCHG	
BFCLR	
BFSET	
BFTSTH	
BFTSTL	
	BRCLR[2]
	BRSET[2]

For More Information On This Product,
Go to: www.freescale.com

Table 2-2 DSP56100, DSP56300, and DSP56600 Instruction Set Differences

DSP56100 Instructions Unsupported or Modified by the DSP56300/600	Instructions Added by Using the DSP56300/600
	BSCLR[2]
	BSET
	BSSET[2]
	BTST
	CLB
CHKAAU	
CLR24	
DEC[1]	
DEC24	
DIV[1]	
	DOR[2]
	DOR FOREVER[2]
EXT	
	EXTRACT
	EXTRACTU
	IFcc
	IFcc.U
IMAC	
IMPY	
INC24	
	INSERT
	JCLR

For More Information On This Product,
Go to: www.freescale.com

Table 2-2 DSP56100, DSP56300, and DSP56600 Instruction Set Differences

DSP56100 Instructions Unsupported or Modified by the DSP56300/600	Instructions Added by Using the DSP56300/600
	JSCLR
	JSET
	JSSET
	LRA
MAC^3	
MACR^3	
	MACI
	MACRI
	MAX
	MAXM
	MERGE
MOVE X: X:	
	MOVE Y:
	MOVE Y: R:
	MOVE L:
	MOVE X: Y:
MPY^3	
MPYR^3	
	MPYI
	MPYIR
NEG^4	
NEGC	

Table 2-2 DSP56100, DSP56300, and DSP56600 Instruction Set Differences

DSP56100 Instructions Unsupported or Modified by the DSP56300/600	Instructions Added by Using the DSP56300/600
NORM^2	
	NORMF
	PFLUSH^2
	PFLUSHUN^2
	PFREE^2
	PLOCKR^2
	PUNLOCK^2
	PUNLOCKR^2
REPcc	
	SUBR
SWAP	
SWI	
TFR(2)	
TFR(3)	
	TRAP
	TRAPcc
TST(2)	
ZERO	

Note: 1. The DSP56300/600 does not allow a parallel move with this instruction.
2. The DSP56600 does not support this instruction.
3. Some registers are not supported as arguments for this instruction on the DSP56300/600.
4. CCR behavior is not identical on both the DSP56100 and the DSP56300/600.

Freescale Semiconductor, Inc.

SECTION 3

SOFTWARE CONVERSION

This section examines unsupported DSP56100 instructions and provides a functionally equivalent DSP56300/600 solution.

3.1   CONVERTING UNSUPPORTED INSTRUCTIONS

Converting unsupported instructions is the primary focus of the code conversion effort. The approach taken is to maintain the functionality of the DSP56100 software rather than fully emulate the unsupported DSP56100 instructions. *This will maintain efficiency, but requires the software engineer to examine the code being translated.* Globally replacing unsupported instructions with complete emulations would yield inefficient software and emulate aspects of an unsupported instruction not required by the DSP56100 code being converted.

As stated earlier in this document, all DSP56300 code portions must be executed with the DSP56300 in 16-bit Arithmetic mode.

3.2   INSTRUCTION TRANSLATIONS

This section examines unsupported DSP56100 instructions and provides a functionally equivalent DSP56300/600 solution. *The intent is provide solutions for the typical use of the DSP56100 instruction rather than support complete emulation (e.g., emulating all the condition codes).* In short, the exact behavior of all the registers and condition codes is not exactly duplicated in many cases. All of the emulation code is written in macro form.

Since macros accept arguments, they are useful for instances in which different arguments are used for the same instruction that needs to emulated. *None of the macros preserve registers that are destroyed performing the emulation*. This approach was chosen because it may introduce unnecessary inefficiency. For instance, the registers used may be available. The software engineer may easily remedy this, if desired, by using push and pop macros with a user-defined stack.

The software conversion engineer has options on how these emulation macros may be used. First, they may be used directly in place of the unsupported instruction. This will be the most inefficient approach, but it will require the minimum effort by the software engineer. Second, they may serve as guides on the functionality that needs to be supported to translate an instruction. The software engineer may then use a variation of the emulation macro that is most efficient for that particular section of code.

3.2.1   ASL4, ASR4, ASR16

These DSP56100 instructions perform arithmetic shifts on an accumulator by fixed amounts. The DSP56300/600 macro uses an immediate data value argument to perform the desired shift.

```
;***********************************************************
;       Registers Destroyed: None
ASL4    MACRO   D
        ASL     #4,D,D          ; Shift accum left 4 bits
        ENDM
;***********************************************************


;***********************************************************
;       Registers Destroyed: None
ASR4    MACRO   D
        ASR     #4,D,D          ; Shift accum right 4 bits
        ENDM
;***********************************************************


;***********************************************************
;       Registers Destroyed: None
ASR16   MACRO   D
        ASR     #16,D,D         ; Shift accum right 16 bits
        ENDM
;***********************************************************
```

3.2.2   BFCHG

This DSP56100 instruction performs a test and change on bits specified by an 8-bit mask. The DSP56300/600 macro will correctly perform the bit changes. *Care has not been taken to maintain the condition code behavior.*

```
;***********************************************************
;       Registers Destroyed: A1
BFCHG   MACRO   MASK,D
        MOVE    D,A1            ; Move target to accum
        EOR     MASK,A          ; Change bits from mask
        MOVE    A1,D            ; Move output to target
        ENDM
;***********************************************************
```

If the mask itself only specifies changing one bit, this macro may be largely improved by using a BCHG instruction. *Note that the BCHG instruction takes a bit position as an argument rather than the bit mask.* The macro that follows is challenging to read since it

contains many assembler functions and directives to convert the bit mask to a bit position. An example of the resulting instruction is provided for clarification. *The C bit is correctly calculated by this macro.*

```
;************************************************************
;       Registers Destroyed: None

BFCHG_1 MACRO   MASK,D
        DEFINE  MVAL    ["MASK",1,@LEN("MASK")]
        BCHG    #@CVI(@L10(MVAL)/@L10(2)),D
        UNDEF   MVAL
        ENDM
;
;************************************************************

Macro Usage:BFCHG_1  #$0008,X0
Actual Code:BCHG     #$3,X0
```

3.2.3   BFCLR

This DSP56100 instruction clears the bits in a destination operand specified by an 8-bit mask. The DSP56300/600 macro will correctly perform the bit clearing. *Care has not been taken to maintain the condition code behavior.*

```
;************************************************************
;       Registers Destroyed: R7, A2, A1, A0, X0
BFCLR   MACRO   MASK,D
        MOVE    MASK,R7         ; Make sure, right justified
        MOVE    R7,A            ; Move mask to accum
        NOT     A               ; Invert the mask
        TFR     D,A     A1,X0   ; X0=mask, A=target
        AND     X0,A            ; Clear bits from mask
        MOVE    A1,D            ; Move output to target
        ENDM
;************************************************************
```

This macro may be shortened if additional care is taken when using the macro. For instance, if the MASK argument contains the forcing operator, then the move to R7 is not needed to ensure integer alignment of the data. The first move would be of the MASK to A.

If the mask itself only specifies clearing one bit, this macro may be further improved by using a BCLR instruction. *Note that the BCLR instruction takes a bit position as an argument rather than the bit mask.* The macro that follows is challenging to read since it contains many Assembler functions and directives to convert the bit mask to a bit

position. An example of the resulting instruction is provided for clarification. *The C bit behaves in the opposite fashion of that on the DSP56100*. If the software following the macro depends upon the state of the C bit, the conditional behavior needs to be inverted (i.e., convert a BCS to BCC).

```
;*********************************************************
;       Registers Destroyed: None
BFCLR_1 MACRO    MASK,D
        DEFINE   MVAL    ["MASK",1,@LEN("MASK")] ; Strip off # character
        BCLR     #@CVI(@L10(MVAL)/@L10(2)),D    ; Convert to bit pos and use
        UNDEF    MVAL
        ENDM
;*********************************************************
```

```
Macro Used:BFCLR_1    #$0008,X0
Actual Code:BCLR      #$3,X0
```

3.2.4   BFSET

This DSP56100 instruction sets the bits in a destination operand specified by an 8-bit mask. The DSP56300/600 macro will correctly perform the bit setting. *Care has not been taken to maintain the condition code behavior*.

```
;*********************************************************
;       Registers Destroyed: A1
BFSET   MACRO    MASK,D
        MOVE     D,A1            ; Move target to accum
        OR       MASK,A          ; Set bits from mask
        MOVE     A1,D            ; Move output to target
        ENDM
;*********************************************************
```

If the mask itself only specifies clearing one bit, this macro may be improved by using a BSET instruction. *Note that the BSET instruction takes a bit position as an argument rather than the bit mask.* The macro that follows is challenging to read since it contains many Assembler functions and directives to convert the bit mask to a bit position. An example of the resulting instruction is provided for clarification. *The C bit is correctly calculated by this macro.*

```
;*********************************************************
;       Registers Destroyed: None
BFSET_1 MACRO    MASK,D
        DEFINE   MVAL    ["MASK",1,@LEN("MASK")] ; Strip off # character
        BSET     #@CVI(@L10(MVAL)/@L10(2)),D    ; Convert to bit pos and use
        UNDEF    MVAL
```

```
        ENDM
;*********************************************************


Macro Used:BFSET_1    #$0008,X0
Actual Code:BSET      #$3,X0
```

3.2.5   BFTSTH

This DSP56100 instruction tests the bits in a destination operand specified by an 8-bit mask and sets the C bit of the Status Register (SR) if all of the bits in the destination operand were set. Otherwise the C bit is cleared. *This DSP56300/600 macro only maintains the functionality of the C bit since that is the most likely usage of this instruction.*

```
;*********************************************************
;       Registers Destroyed: A1
BFTSTH  MACRO   MASK,D
        TFR     D,A             ; Move target to accum
        AND     MASK,A          ; Clear bits not set in mask
        EOR     MASK,A          ; Clear bits set in mask, z=1 if all cleared
        ANDI    #$FE,CCR        ; Optional - Clear the C flag
        BNE     _EXIT           ; Optional - If Z flag not set, exit
        ORI     #$01,CCR        ; Optional - If Z flag set, set C flag
_EXIT
        ENDM
;*********************************************************
```

This macro may be improved if the user may utilize the Z bit of the Condition Code Register (CCR). Since the last three instructions of the macro use the state of the Z bit to set the C bit, they may be potentially dropped if the state of the Z bit is used (i.e., for a conditional change of flow following the instruction).

If the mask itself only specifies testing one bit, this macro may be further improved by using a BTST instruction. *Note that the BTST instruction takes a bit position as an argument rather than the bit mask.* The macro that follows is challenging to read since it contains many Assembler functions and directives to convert the bit mask to a bit position. An example of the resulting instruction is provided for clarification. *The C bit is correctly calculated by this macro.*

```
;*********************************************************
;       Registers Destroyed: None
BFTSTH_1        MACRO   MASK,D
        DEFINE  MVAL    ["MASK",1,@LEN("MASK")] ; Strip off # character
        BTST    #@CVI(@L10(MVAL)/@L10(2)),D     ; Convert to bit pos and use
        UNDEF   MVAL
        ENDM
```

```
;********************************************************

Macro Used:BFTSTH_1   #$0008,X0
Actual Code:BTST      #$3,X0
```

3.2.6   BFTSTL

This DSP56100 instruction tests the bits in a destination operand specified by an 8-bit mask and sets the C bit of the SR if all of the bits in the destination operand were clear. Otherwise the C bit is cleared. *This DSP56300/600 macro only maintains the functionality of the C bit since that is the most likely usage of this instruction.*

```
;********************************************************
;       Registers Destroyed: A1
BFTSTL  MACRO   MASK,D
        TFR     D,A             ; Move target to accum
        NOT     A               ; Invert the target
        AND     MASK,A          ; Clear bits not set in mask
        EOR     MASK,A          ; Clear bit set in mask, z=1 if all cleared
        ANDI    #$FE,CCR        ; Optional - Clear the C flag
        BNE     _EXIT           ; Optional - If Z flag not set, exit
        ORI     #$01,CCR        ; Optional - If Z flag set, set C flag
_EXIT
        ENDM
;********************************************************
```

This macro may be improved if the user may utilize the Z bit of the CCR. Since the last three instructions of the macro use the state of the Z bit to set the C bit, they may be potentially dropped if the state of the Z bit is used (i.e., for a conditional change of flow following the instruction).

If the mask itself only specifies testing one bit, this macro may be further improved by using a BTST instruction and an additional BCHG instruction. *Note that the BTST instruction takes a bit position as an argument rather than the bit mask.* The macro that follows is challenging to read since it contains many Assembler functions and directives to convert the bit mask to a bit position. An example of the resulting instruction is provided for clarification. *The C bit behaves in the opposite fashion of that on the DSP56100.* If the software following the macro depends upon the state of the C bit, the conditional behavior needs to be inverted (i.e., convert a BCS to BCC).

```
;********************************************************
;       Registers Destroyed: None
BFTSTL_1        MACRO   MASK,D
        DEFINE  MVAL    ["MASK",1,@LEN("MASK")] ; Strip off # character
```

```
        BTST      #@CVI(@L10(MVAL)/@L10(2)),D    ; Convert to bit pos and use
        UNDEF     MVAL
        ENDM
;********************************************************
```

```
Macro Used:BFTSTL_1    #$0008,X0
Actual Code:BTST       #$3,X0
```

3.2.7   CHKAAU

This DSP56100 instruction affects the N, Z, and V bits of the SR based upon the result of the address register calculation. On the DSP56300/600, a simple testing of the resulting value will *handle the N and Z bits only.*

```
;********************************************************
;         Registers Destroyed: A
CHKAAU  MACRO     S
        MOVE      S,A
        TST       A                 ; Set SR flags based on value
        ENDM
;********************************************************
```

3.2.8   DEC24

This DSP56100 instruction decrements by one the MSP of the accumulator specified. The DSP56300/600 macro is self explanatory.

```
;********************************************************
;         Registers Destroyed: None
DEC24   MACRO     D
        SUB       #1,D              ; Decrement MSP by one
        ENDM
;********************************************************
```

3.2.9   EXT

This DSP56100 instruction sign extends the MSP of the specified accumulator. The DSP56300/600 macro achieves the same result through arithmetic shifts.

```
;********************************************************
;         Registers Destroyed: None
```

```
EXT     MACRO   D
        ASL     #8,D,D          ; Shift out current extension
        ASR     #8,D,D          ; Sign extend LSP
        ENDM
;************************************************************
```

3.2.10   IMAC, IMPY

These DSP56100 instruction perform integer multiply accumulate and integer multiply. The DSP56300/600 macros perform the calculation including the sign extension of the result.

```
;************************************************************
;       Registers Destroyed: LSP specified by argument D
IMAC    MACRO   S1,S2,D
        ASR     #15,D,D         ; LSP = fract val
        MAC     S1,S2,D         ; A = S1*S2+A in fract form
        ASR     D               ; A = S1*S2+A int form
        MOVE    D\0,D           ; Truncate and sign extend result
        ENDM
;************************************************************


;************************************************************
;       Registers Destroyed: LSP specified by argument D
IMPY    MACRO   S1,S2,D
        MPY     S1,S2,D         ; A = S1*S2+A in fract form
        ASR     D               ; A = S1*S2+A int form
        MOVE    D\0,D           ; Truncate and sign extend result
        ENDM
;************************************************************
```

The IMAC and IMPY DSP56100 instructions support the MSP of the accumulator as a source argument. Hence, the above macros may be modified to move the accumulator source to a temporary register to be used in the DSP56300/600 MAC or MPY instructions. The macro requires that the user specify the temporary Data ALU register to be used so that it does not conflict with the second Data ALU included in the original IMAC or IMPY instruction.

```
;************************************************************
;       Registers Destroyed: Specified by TMP argument
IMAC_AS MACRO   S1,S2,D,TMP
        MOVE    S1,TMP
        ASR     #15,D,D         ; LSP = fract val
        MAC     TMP,S2,D        ; A = S1*Y0+A in fract form
        ASR     D               ; A = S1*Y0+A int form
        MOVE    D\0,D           ; Truncate and sign extend result
```

```
        ENDM
;*********************************************************


;*********************************************************
;       Registers Destroyed: Specified by TMP argument
IMPY_AS MACRO   S1,S2,D,TMP
        MOVE    S1,TMP
        MPY     TMP,S2,D        ; A = S1*S2+A in fract form
        ASR     D               ; A = S1*S2+A int form
        MOVE    D\0,D           ; Truncate and sign extend result
        ENDM
;*********************************************************
```

3.2.11   INC24

This DSP56100 instruction increments the MSP of the specified accumulator. The
function of the macro is self-evident.

```
;*********************************************************
;       Registers Destroyed: None
INC24   MACRO   D
        ADD     #1,D            ; Increment MSP by 1
        ENDM
;*********************************************************
```

3.2.12   MAC, MACR, MPY, and MPYR with MSP as Source

The DSP56100 allowed the MSP of an accumulator to be used as a source for the
MAC, MACR, MPY, and MPYR instructions. Supporting them simply requires
moving the source accumulator to a temporary Data ALU register just prior to the
original DSP56100 instruction. The user must specify the temporary Data ALU
register used so that it does not conflict with second source in the original DSP56100
instruction.

```
;*********************************************************
;       Registers Destroyed: Specified by TMP argument
MAC_AS  MACRO   S1,S2,D,TMP
        MOVE    S1,TMP          ; Move source to data reg
        MAC     TMP,S2,D        ; Perform mac
        ENDM
;*********************************************************


;*********************************************************
```

```
;       Registers Destroyed: Specified by TMP argument
MACR_AS MACRO   S1,S2,D,TMP
        MOVE    S1,TMP          ; Move source to data reg
        MACR    TMP,S2,D        ; Perform macr
        ENDM
;***********************************************************


;***********************************************************
;       Registers Destroyed: Specified by TMP argument
MPY_AS  MACRO   S1,S2,D,TMP
        MOVE    S1,TMP          ; Move source to data reg
        MPY     TMP,S2,D        ; Perform mpy
        ENDM
;***********************************************************


;***********************************************************
;       Registers Destroyed: Specified by TMP argument
MPYR_AS MACRO   S1,S2,D,TMP
        MOVE    S1,TMP          ; Move source to data reg
        MPYR    TMP,S2,D        ; Perfor mpyr
        ENDM
;***********************************************************
```

3.2.13   NEGC

This DSP56100 instruction performs the mathematical function of $0 - C - D$ (zero minus the Carry bit minus the accumulator value). The DSP56300/600 macro achieves this by testing the C bit and incrementing it before the negation.

```
;***********************************************************
;       Registers Destroyed: None
NEGC    MACRO   D
        BCC     _SKIP           ; If carry clear, just negate
        INC     D               ; If carry set, increment first
_SKIP   NEG     D               ; Perform negation
        ENDM
;***********************************************************
```

3.2.14   REPcc

This DSP56100 instruction repeated an instruction until the specified condition code (cc) was true. *Translating this instruction should be carefully considered due to the*

repetitive nature of the instruction. Perhaps the most common form is REPNR. Such instances should use CLB and NORMF for fast normalization on the DSP56300/600. Other uses should be examined for efficient implementation using the DSP56300/600 instruction set. For completeness, a general purpose DSP56300/600 macro follows, but is considerably more inefficient. Using the macro requires the condition be specified, as well as the instruction to be repeated. Since most instructions have more than one field, quotes will be required for the instruction argument.

```
;**********************************************************
;       Registers Destroyed: None
REPcc   MACRO   CC,INSTR
        DO      FOREVER,_LOOP   ; Do forever
        BRK\CC                  ; End the do if the condition is true
        INSTR                   ; Execute the instruction to repeat
        NOP                     ; NOP required for BRKcc near end of DO
        NOP                     ; NOP required for BRKcc near end of DO
_LOOP
        ENDM
;**********************************************************
```

An improvement that is not simply converted into a macro, but may be utilized, involves using a conditional branch on the *opposite* condition code. For instance, if the original form was REPNE, the repeated instruction could be followed by a BREQ that branches back to the instruction to be repeated. When the condition is satisfied, the branch will not occur.

3.2.15   SWAP

This DSP56100 instruction simply swaps the MSP and LSP of the specified accumulator. The DSP56300/600 macro uses a Data ALU register as an intermediate location while the words are being swapped. The user may choose to use another less often used register for temporary storage.

```
;**********************************************************
;       Registers Destroyed: X0
SWAP    MACRO   D
        MOVE    D\1,X0          ; Move MSP to X0
        MOVE    D\0,D\1         ; Move LSP to MSP
        MOVE    X0,D\0          ; Move X0 reg to LSP
        ENDM
;**********************************************************
```

3.2.16 SWI

This DSP56100 instruction initiates SWI exception processing. The equivalent DSP56300/600 processing is the TRAP instruction as shown in the macro.

```
;**********************************************************
;       Registers Destroyed: None
SWI     MACRO
        TRAP
        ENDM
;**********************************************************
```

3.2.17 TFR of an entire DALU register

This DSP56100 allows a transfer of the entire X or Y Data ALU register to an accumulator. The DSP56300/600 macro supports this by following a transfer of the MSP with sign extension with a move of the LSP to the accumulator.

```
;**********************************************************
;       Registers Destroyed: None
TFR_XY  MACRO   S,D             ; Necessary only if S is X or Y.
        TFR     S\1,D           ; Move MSP and sign extend
        MOVE    S\0,D\0         ; Move in LSP
        ENDM
;**********************************************************
```

3.2.18 TFR2

This DSP56100 instruction allowed transfer of an entire accumulator to a full 32-bit Data ALU register. The transfer also performed limiting on the resulting 32-bit value if necessary.

```
;**********************************************************
;       Registers Destroyed: None
TFR2    MACRO   S,D
        MOVE    S\0,D\0         ; Move LSP of accum to data reg
        MOVE    S,D\1           ; Move MSP to data REG, limit
        BLC     _EXIT           ; If no limit, no limit on data range LSP
        TST     S               ; Test the accum for GT or LT
        MOVE    #$FFFF,D\0      ; Extend limit value to LSP, assume positive
        BGT     _EXIT           ; Exit if it was positive
        MOVE    #$0,D\0         ; Else extend LSP for a negative limiting
_EXIT
```

```
        ENDM
;*********************************************************
```

3.2.19   TFR3

This DSP56100 instruction allows transfer of an accumulator to a Data ALU register along with a memory read/write to/from another Data ALU register. This may be supported on the DSP56300/600 by ensuring there are no conflicts on the XDB or YDB. For instance:

```
        MOVE    A,X1    Y:(R0)+,Y0
```

Hence, translating this instruction must be handled on a case-by-case basis by the software engineer since it is dependent upon the location of the data. If none of the data has been moved to an alternate memory, then this is very similar to splitting a parallel move into two move instructions.

3.2.20   TST2

This DSP56100 instruction provides testing of a Data ALU register for the purpose of setting the CCR based on its contents. The DSP56300/600 macro simply moves the Data ALU register to an accumulator where the TST instruction may be used.

```
;*********************************************************
;       Registers Destroyed: A2, A1, A0
TST2    MACRO   S
        MOVE    S,A             ; Move data reg to accum
        TST     A               ; Test accumulator
        ENDM
;*********************************************************
```

3.2.21   ZERO

This DSP56100 instruction ZERO extends the specified accumulator. The DSP56300/600 macro accomplishes this by moving immediate data into the extension portion of the specified accumulator.

```
;************************************************************
;       Registers Destroyed: None
ZERO    MACRO   D
        MOVE    #$0,D\2          ; Move zero to extension
        TST     D               ; Set condition codes
        ENDM
;************************************************************
```

3.2.22   Immediate Short Data

On the DSP56100, immediate short data is right justified when moved into a Data ALU register. On the DSP56300/600, this data will be left justified. This may be corrected by using the forcing operator recognized by the Assembler. The expense of this difference is that the instruction becomes two words (the data is now full size).

```
DSP56100MOVE  #3,X0
DSP56300/600MOVE#>3,X0
```

3.3   EMULATIONS ISSUES SPECIFIC TO THE DSP56600

Other than the instruction set differences specified earlier, there are few differences between the DSP56300 and DSP56600 that need to be considered.

3.3.1   NORM Rn,D

The DSP56600 does not support the single iteration NORM instruction. Typically, the NORM instruction is repeated to fully normalize an accumulator. In these instances, using the CLB instruction followed by NORMF will provide the equivalent normalized result.

```
DSP566100:REPNR
        NORM    R0,A
DSP56300/600CLBA,B
        NORMF   B1.A,A
```

There are a few issues to be aware of in this emulation. First, an additional accumulator is required. Second, the address register is not updated as in the DSP56100 code. If the address register is not used elsewhere in the code, this is of no consequence. Otherwise, the result of the CLB instruction must be moved into the address register. If the address register was nonzero before the DSP56100 NORM

instruction was executed, the value in the address register must be added to the result of the CLB instruction. This correction must be performed after the NORMF so that the accumulator is normalized properly.

Another situation to address is a single iteration of the NORM instruction. Although the occurrence of this is *highly* unlikely, it should be mentioned. An equivalent emulation of this situation follows.

```
DSP56600:BNR   <_SKIPNR
       BES     <_SHIFTRT
       ASL     A       (R0)-
       BRA     <_SKIPNR
_SHIFTRT
       ASR     A       (R0)+
_SKIPNR
```

In these situations, it is clearly most efficient to consider the specific situation. For example, code may be reduced greatly if the address register is not of interest and the direction of normalization, if needed, is known. Other reductions are simplifications by case from the example above.

```
DSP56600:ASL  A      IFNN
```