Application Note: Machine Code Programming

IsoMax allows individual words to be written in machine code as well as "high-level" language code. Such words are indistinguishable in function from high-level words, and may be used freely in application programs and state machines.

Assembler Programming

The IsoPod uses the Motorola DSP56F805 microprocessor. The machine language of this processor is described in Motorola's *DSP56800 16-Bit Digital Signal Processor Family Manual,* available at

http://e-www.motorola.com/brdata/PDFDB/docs/DSP56800FM.pdf

IsoMax does *not* include a symbolic assembler for this processor. You must use an external assembler to convert your program to the equivalent hexadecimal machine code, and then insert these numeric opcodes and operands into your IsoMax source code.[1] For an example, let's use an assembler routine to stop Timer C2:

```
; Timer/Counter
; -------------
; Timer control register
;  000x xxxx xxxx xxxx = no count
andc    #$1FFF,X:$0D56  ; TMRC2_CTRL

; Timer status & control register
; Clear TCF flag, clear interrupt enable flag
bfclr   #$8000,X:$0D57  ; TMRC2_SCR  clear TCF
bfclr   #$4000,X:$0D57  ; TMRC2_SCR  clear TCFIE
```

Translated to machine code, this is:

```
80F4    andc    #$1FFF,X:$0D56
0D56
E000
80F4    bfclr   #$8000,X:$0D57
0D57
8000
80F4    bfclr   #$4000,X:$0D57
0D57
4000
```

To compile this manually into an IsoMax word, you must append each hexadecimal value to the dictionary with the `P,` operator. (The "P" refers to Program space,where all machine code must reside.) You can put more than one value per line:

```
80F4 P, 0D56 P, E000 P,
80F4 P, 0D57 P, 8000 P,
80F4 P, 0D57 P, 4000 P,
```

All that remains is to add this as a word to the IsoMax dictionary, and to return from the assembler code to IsoMax. There are three ways to do this: with `CODE`, `CODE-SUB`, and `CODE-INT`.

[1] If you wish to translate your programs manually to machine code, a summary chart of DSP56800 instruction encoding is given at the end of this application note.

CODE functions

The special word CODE defines a machine language word as follows:

```
CODE word-name
    (machine language for your word)
    (machine language for JMP NEXT)
END-CODE
```

Machine code words that are created with CODE must return to IsoMax by performing a jump to the special address NEXT. *In IsoMax versions 0.52 and higher, this is address $0080. Earlier versions of IsoMax do not support NEXT and you must use CODE-SUB, described below, to write machine code words.*

An absolute jump instruction is $E984. Thus a JMP NEXT translates to $E984 $0080, and our example STOP-TIMERC2 word could be written as follows:

```
HEX
CODE STOP-TIMERC2
    80F4 P, 0D56 P, E000 P,
    80F4 P, 0D57 P, 8000 P,
    80F4 P, 0D57 P, 4000 P,
    E984 P, 0080 P, ( JMP NEXT )
END-CODE
```

Remember, this example will only work on recent versions of IsoMax (0.52 or later).

CODE-SUB functions

The special word CODE-SUB is just like CODE, except that the machine code returns to IsoMax with an ordinary RTS instruction. This can be useful if you need to write a machine code routine that can be called both from IsoMax and from other machine code routines. It's also useful if the NEXT address is not available (as in IsoMax versions prior to 0.52). The syntax is similar to CODE:

```
CODE-SUB word-name
    (machine language for your word)
    (machine language for RTS)
END-CODE
```

An RTS instruction is $EDD8, so STOP-TIMERC2 could be written with CODE-SUB as follows:

```
HEX
CODE-SUB STOP-TIMERC2
    80F4 P, 0D56 P, E000 P,
    80F4 P, 0D57 P, 8000 P,
    80F4 P, 0D57 P, 4000 P,
    EDD8 P, ( RTS )
END-CODE
```

This example will work in all versions of IsoMax.

CODE-INT functions

CODE-INT is just like CODE-SUB, except that the machine code returns to IsoMax with an RTI (Return from Interrupt) instruction, $EDD9. This is useful if you need to write a machine code interrupt handler that can also be called directly from IsoMax. *CODE-INT is only available on IsoMax versions 0.52 and later.*

```
HEX
CODE-INT STOP-TIMERC2
    80F4 P, 0D56 P, E000 P,
    80F4 P, 0D57 P, 8000 P,
    80F4 P, 0D57 P, 4000 P,
    EDD9 P, ( RTI )
END-CODE
```

To obtain the address of the machine code after it is compiled, use the phrase

```
' word-name CFA 2+
```

Note: if you are using EEWORD to put this new word into Flash ROM, use EEWORD *before* trying to obtain the address of the machine code. EEWORD will change this address.

Register Usage

In the current version of IsoMax software, all DSP56800 address and data registers may be used in your CODE and CODE-SUB words. You need not preserve R0-R3, X0, Y0, Y1, A, B, or N. Do not change the "mode" registers M01 or OMR, and do not change the stack pointer SP.

Future versions of IsoMax may add more restrictions on register use. If you are concerned about compatibility with future kernels, you should save and restore all registers that your machine code will use.

CODE-INT words are expected to be called from interrupts, and so they should save any registers that they use.

Calling High-Level Words from Machine Code

You can call a high-level IsoMax word from within a machine-code subroutine. This is done by calling the special subroutine ATO4 with the address of the word you want to execute.[2] This address must be a Code Field Address (CFA) and is obtained with the phrase

```
' word-name CFA
```

This address must be passed in register R0. You can load a value into R0 with the machine instruction $87D0, $xxxx (where xxxx is the value to be loaded).

The address of the ATO4 routine can be obtained from a constant named ATO4. You can use this constant directly when building machine code. The opcode for a JSR instruction is $E9C8, $aaaa where aaaa is an absolute address. So, to write a CODE-SUB routine that calls the IsoMax word DUP, you could write:

```
HEX
CODE-SUB NEWDUP
    87D0 P, ' DUP CFA P,          ( move DUP CFA to R0 )
    E9C8 P, ATO4 P,               ( JSR ATO4 )
    EDD8 P,                       ( RTS )
END-CODE
```

Observe that the phrases ' DUP CFA and ATO4 are used *within* the CODE-SUB to generate the proper addresses where required.

[2]The name ATO4 comes from "Assembler to Forth" and refers to the Forth underpinnings of IsoMax.

Appendix: DSP56F805 Instruction Encoding

DSP56800 OPCODE ENCODING

```
(1)         00Wk  kHHH  Fjjj  xmRR  (14)       P1DALU  jjj,F          X:<ea_m>,HHH

(2)         010y  y0yy  y*pp  pppp  (11-*)  ADD/SUB/CMP/INC/DEC X:<aa>[,fff]
(3)         010y  y0yy  y+aa  aaaa  (11-*)  ADD/SUB/CMP/INC/DEC X:(SP-xx)[,fff]
(4)         010y  y1yy  y00B  BBBB  (10)       ADD/SUB/CMP #<0-31>,fff
(5a)        010y  y1yy  y10-  ----  (5-2)      ADD/SUB/CMP #xxxx,fff
(5b)        010y  y1yy  yw11  -1--  (6-2)      ADD/SUB/CMP/INC/DEC X:xxxx[,fff]

(7)         011u  u0v1  Fvjj  xm-v  (10)       P2DALU jj,F   X:<ea_m>,reg X:<ea_v>,X0
(8a)        011L  L1L-  FQQQ  10FF  (9)        DALU3OP  QQQ,FFF
(8b)        011I  I1II  FQQQ  11FF  (10)       DALU3OP2 QQQ,FFF
(8c)        011K  K1K-  F000  0h00  (4)        DALU2OPF  ~F,F   (KKK = KK0)  (h=1: Tcc)
(8d)        011K  K1K-  F000  0h00  (4)        DALU2OPY   Y,F   (KKK = KK1)  (h=1 used)
(8e)        011K  K1K-  F000  0hF1  (5)        DALU2OPB1  B1,FF              (h=1: Tcc)
(8f)        011K  K1K-  F010  0hF1  (5)        DALU2OPA1  A1,FF              (h=1: Tcc)
(8g)        011K  K1K-  F0qq  0h00  (6)        DALU1OPF   F            (qq != 00)
(h=1 used)
(8h)        011K  K1K-  F0q1  0hF1  (6)        DALU1OPFF  FF
(h=1: LSL,LSR)
(8i)        011K  K1K-  F1JJ  0hFF  (8)        DALU2OPJJ  JJ,FFF        (h=1: DIV,Tcc)
(8j)        0110  11CC  FJJJ  01CZ  (8)        Tcc            JJJ,F   [R0->R1]   (h=1:
Tcc)

(9)         10W1  HHHH  0Ppp  pppp  (12)       MOVE X:<Ppp>,REG
(10a)       10W1  HHHH  1*AA  AAAA  (11)       MOVE X:(R2+xx),REG
(10b)       10W1  HHHH  1+aa  aaaa  (11)       MOVE X:(SP-xx),REG
(11)        11W1  DDDD  D0-M  RMRR  (12)       MOVE X:<ea_MM>,DDDDD
(12)        11W1  DDDD  D1-0  R1RR  (10)       MOVE X:(Rn+N),DDDDD
(13)        11W1  DDDD  D1-0  R0RR  (10-2)  MOVE X:(Rn+xxxx),DDDDD
(14)        11W1  DDDD  D1-1  -1--  (7-2)      MOVE X:<abs_adr>,DDDDD

(15)        1000  DDDD  D00d  dddd  (10)       MOVE ddddd,DDDDD
(16)        1000  1110  *011  00RR  (2)        TSTW (Rn)-
(17)        1000  UUU+  110d  dddd  (8-2)      BITFIELD DDDDD;     MOVE #xxxx,DDDDD
(18)        1000  UUU0  111+  -+--  (3-3)      BITFIELD X:xxxx;    MOVE #xxxx,X:xxxx
(19a)       1010  UUU0  1+aa  aaaa  (9-2)      BITFIELD X:(SP-xx); MOVE #xxxx,X:(SP-xx)
(19b)       1010  UUU0  1*AA  AAAA  (9-2)      BITFIELD X:(R2+xx); MOVE #xxxx,X:(R2+xx)
(20)        1010  UUU1  1Ppp  pppp  (10-2)  BITFIELD X:<Ppp>;   MOVE #xxxx,X:<Ppp>
(21)        1010  CCCC  0Aaa  aaaa  (11)       Bcc <aa>,  BRA

(22)        1100  HHHH  *BBB  BBBB  (11)       MOVE #xx,HHHH
(23)        1100  11E0  1*BB  BBBB  (7-*)      DO/REP #xx
(24)        1100  11E0  11-d  dddd  (6-*)      DO/REP ddddd
(25a)       1110  CCCC  10A-  -1AA  (7-2)      Jcc, JMP  xxxxx
(25b)       1110  1001  11A0  10AA  (*-2)      JSR  xxxxx
(26)        1110  1101  11-1  10-0  (0)        RTS
(27)        1110  1101  11-1  10-1  (0)        RTI
(29)        1110  HHHH  *0W*  *mRR  (8)        MOVE P:<ea_m>,HHHH

(30)        1110  ----  -1--  0000  (0)        NOP
(31)        1110  ----  -1--  0001  (0)        DEBUG
(--)        1110  ----  -1--  0010  (0)        ($E042 -reserved for "ADD <reg>,<mem>")
(32)        1110  ----  -1--  01tt  (2)        STOP, WAIT, SWI,  ILLEGAL

(--)        1100  ----  111-  ----  (9)        <Available Hole>
(--)        1110  ----  111-  ----  (9)        <Available Hole>
(--)        1110  ----  01--  ----  (10)       <Available Hole>
```

Understanding entries in the above encoding:

A typical entry in the encoding files looks like this:

```
(8b)        011I  I1II  FQQQ  11FF  (10)       DALU3OP2 QQQ,FFF

 ^          \                          /   ^    ^
 |           ---------v----------   |           |
 |                     |                     |           |
```

```
|                       |                      |           +---- (see #1 below)
|                       |                      +-------------- (see #2 below)
|                       +---------------------------- (see #3 below)
+------------------------------------------------ (see #4 below)
```

#1: This field gives the name of the instruction or of a class of instructions which are encoded with the bit pattern specified in #3.

An example of where this field contains an instruction is for the "TSTW (Rn)-" instruction. In this case, only the operands of the instruction are encoded with the bits in #3 below.

An example of where this field contains a class of instructions is given in the example above "DALU3OP2 QQQ,FFF". In this case, the entry DALU3OP2 represents a class of instructions, and the instruction selected within this class is selected by the IIII field within the encoding specified in #2.

Instruction classes such as "DALU3OP2" can be seen by searching in this file for the following field - "DALU3OP2:", where the field is located in the very first character of the line.

#2: The number here indicates how many bits are required to encode this instruction. For the example shown above, 10 bits are required to hold the following bits - IIIIFQQQFF. The information in this particular field is useful to the design group.

If the number in this field is followed by a "-2" or "-3", the "-2" is used to indicate a two word instruction, and the "-3" is used to indicate a three word instruction.

For the case of the "ADD/SUB/CMP/INC/DEC X:<aa>[,fff]" instruction which uses "(11-*)", this indicates that this class of instructions can vary in number of instruction words. For this particular example, this can be seen more clearly in the section entitled "Unusual Instruction Encodings" located within this document.

#3: This portion represents the 16 opcode bits of the instruction. For single word instructions, it contains the entire one word 16-bit opcode. For multiword instructions, it contains the first word for the instruction.

The example above contains the following fields within the instruction:

IIII, FFF, QQQ

Note that although there are four I bits to form the "IIII" field, these bits are not necessarily all next to each other. This is also the case for the three bits comprising the "FFF" field.

#4: The number here gives a unique number to this particular instruction or class of instructions. This is used simply for identification purposes.

Notes for Above Encoding:
---

1. Where a "*" is present in a bit in the encoding, this means the PLAs often use this bit to line up in a field, but that the assembler should always see this as a "0". Where a "+" is present, it is similar, but assembles as a "1". A "-" is ignored by the PLAs and assembled as a "0".

2. It is important to note that several instructions are not found on the first page of the encoding, which summarizes the entire instruction set. These instructions are instead found in the section entitled "Unusual Instruction Encodings" located within this document. Instructions in this section include:
   - ADD fff,X:<aa>:
   - ADD fff,X:(SP-xx):
   - ADD fff,X:xxxx:
   - LEA
   - TSTW
   - POP
   - CLR (although CLR is also encoded in the Data ALU section)
   - ENDDO

See this section to see how these instructions are encoded.

3. The use of the bit pattern labelled
"($E042 -reserved for "ADD <reg>,<mem>")"
is explained in more detail in the "Unusual Instruction Encodings" section. It is not an instruction in itself, but rather enables an encoding trick discussed for the ADD instruction in that section.

Understanding the 2 and 1 Operand Data ALU Encodings

The Data ALU operations were encoded in a manner which is not straightforward. The three operand instructions were relatively straightforward, but the encoding of the two and one operand instructions was more difficult.

More information is presented at the field definitions for "KKK" and "JJJ". This is the best place to clearly understand the Data ALU encodings.

(Also see the encoding information located at the "KKK" field.)

Data ALU Source and Destination Register Field Definitions:

F:

F	Destination Accumulator
0	A
1	B

~F:

"~F" is a unique notation used in some cases to signify the source register in a DALU operation. It's exact definition is as follows:
If "F" is the "A" accumulator, Then "~F" is the "B" accumulator.
If "F" is the "B" accumulator, Then "~F" is the "A" accumulator.

FF:

FF	Destination Register
00	X0 (NOTE: not all DALU instrs can have this as a destination)
10	(reserved)
01	Y0 (NOTE: not all DALU instrs can have this as a destination)
11	Y1 (NOTE: not all DALU instrs can have this as a destination)

FFF:

FFF	Destination Register
000	A
100	B
001	X0 (NOTE: not all DALU instrs can have this as a destination)
101	(reserved)
011	Y0 (NOTE: not all DALU instrs can have this as a destination)
111	Y1 (NOTE: not all DALU instrs can have this as a destination)

NOTE: The MPY, MAC, MPYR, and MACR instructions allow x0, y0, or y1 as a destination. FFF=FF1 IS allowed for the case of a negated product: -y0,x0,FFF for example is allowed. Also, MPYsu, MACsu, IMPY16, LSRR, ASRR, and ASLL allow FFF as a destination, but the ASRAC & LSRAC instructions only allow F, and LSLL only allows DD as destinations.

Although the LSLL only allows 16-bit destinations, there is the ASLL instruction which performs exactly the same operation and allows an accumulator as well as a destination.

fff:

fff	Destination Register
000	A (ADD/SUB/CMP only)
001	B (ADD/SUB/CMP only)
100	X0 (ADD/SUB/CMP only)
101	(reserved for X1)
110	Y0 (ADD/SUB/CMP only)
111	Y1 (ADD/SUB/CMP only)

QQQ: (6-4)

This field specifies two input registers for instructions in the DALU3OP, DALU3OP2, and P1DALU instruction classes. There are some instructions where the ordering of the two source operands is important and some where the ordering is unimportant.

Three different cases are presented below for instructions using the QQQ field. Some examples are also included for clarification. Note that the bottom 4 entries are designed to overlay the "QQ" field.

1. "QQQ" definition for: ASRR, ASLL, LSRR, LSLL, ASRAC, & LSRAC instrs

QQQ	Shifter inputs (must be in this order)
000	(reserved for X1,Y1)
001	B1,Y1
010	Y0,Y0
011	A1,Y0
100	Y0,X0
101	Y1,X0
110	(reserved for X1,Y0)
111	Y1,Y0

For Multi-bit shift instructions:
- 1st reg specified is value to be shifted
- 2nd reg specified is shift count (uses 4 LSBs)

Examples of valid Multi-bit shift instructions:

```
asll b1,y1,a          ; b1 is value to be shifted, y1 is shift amount
asrr y1,x0,b          ; y1 is value to be shifted, x0 is shift amount
```

Examples of INVALID Multi-bit shift instructions:

```
asll y1,b1,a          ; Not allowed - b1 must be first for QQQ=001
asrr x0,y1,b          ; Not allowed - y1 must be first for QQQ=101
```

2. "QQQ" definition for: MPYsu and MACsu instrs

QQQ	Multiplier inputs (must be in this order)
000	(reserved for Y1,X1)
001	Y1,B1
010	Y0,Y0
011	Y0,A1
100	X0,Y0
101	X0,Y1
110	(reserved for Y0,X1)
111	Y0,Y1

For MPYsu or MACsu instructions:
- 1st reg specified in QQQ above is "signed" value
- 2nd reg specified in QQQ above is "unsigned" value

Examples of valid MPYsu and MACsu instructions:

```
mpysu y1,b1,a   ; y1 is signed, b1 unsigned, QQQ = 001
macsu x0,y1,b   ; x0 is signed, y1 unsigned, QQQ = 101
```

Examples of INVALID MPYsu and MACsu instructions:

```
mpysu b1,y1,a   ; Not allowed - y1 must be signed for QQQ=001
macsu y1,x0,b   ; Not allowed - x0 must be signed for QQQ=101
```

The Multi-bit shift instructions include:
ASRR, ASLL, LSRR, LSLL, ASRAC, and LSRAC

3. "QQQ" definition for: All other instructions using "QQQ"

QQQ	Multiplier inputs	Also Accepted by Assembler

```
                    ---         -----------------       --------------------------
                    000         (reserved for Y1,X1)    (reserved for X1,Y1)
                    001         Y1,B1                             B1,Y1
                    010         Y0,Y0                             Y0,Y0
                    011         Y0,A1                             A1,Y0
                    100         X0,Y0                             Y0,X0
                    101         X0,Y1                             Y1,X0
                    110         (reserved for Y0,X1)    (reserved for X1,Y0)
                    111         Y0,Y1                             Y1,Y0

                For all other of these instructions:
                      - operands can be specified in either order

                Examples of valid MPY and MAC instructions:
                      mpy   y1,b1,a   ; Operands are: y1 and b1   (ordering unimpt)
                      mpy   b1,y1,a   ; Operands are: y1 and b1   (ordering unimpt)
                      mac   x0,y1,b   ; Operands are: y1 and x0   (ordering unimpt)
                      mac   y1,x0,b   ; Operands are: y1 and x0   (ordering unimpt)

     NOTE: If the source operand ordering is incorrect, then the assembler
               must flag this as an error.

Data-Alu Opcode Field Definitions:
==================================

q:   used to specify "non-multiply" one operand DALU/P1DALU instructions.
     See the "KKK" field definition below.

qq:  used to specify "non-multiply" one operand DALU/P1DALU instructions.
     See the "KKK" field definition below.

DALU3OP:
--------
LLL:        LLL         Multiplication Operation
            ---         -----------------------
            000          MPY  +   (neither operand inverted)
            001          MPY  -   (one     operand inverted)
            010          MAC  +   (neither operand inverted)
            011          MAC  -   (one     operand inverted)
            100          MPYR +   (neither operand inverted)
            101          MPYR -   (one     operand inverted)
            110          MACR +   (neither operand inverted)
            111          MACR -   (one     operand inverted)

h: (2)
            The "h" bit, when set to a "1" is used to encode the following
            non-multiply DALU instructions:
               - ADC, SBC
               - NORM R0
               - LSL, LSR
               - DIV

            For exact details on this, see the "KKK" field definition below.

DALU2OPF:
DALU2OPY:
DALU2OPB1:
DALU2OPA1:
DALU1OPF:
DALU1OPFF:
DALU2OPJJ:

KKK: ()

            The KKK fields cannot be uniquely decoded without looking at the
            values in some other bits of the opcode.  In the below charts, the
            KKK field holds many different encodings depending on the values
            in bits 6-4, what was previously called the JJJ field, and bit 2,
            which was previously labelled as "h".  The JJJ and h fields have
            now been removed and this chart now contains the information
            previously held by these bits.
```

Four different charts are presented below, where the four charts correspond to different values "00, 01, 10, and 11" in bits 2 and 0 of the opcode.

Note that the KKK entries are numbered in an ascending order from 0 to 7. This also differs from the numbering in the original encoding file (encode8) so the entries in the chart will now appear to be in a different order.

Notation for the below charts:

- <<NA>> - Indicates field is not available for any instruction
- <<Tc>> - Indicates space is not available because it is occupied by the Tcc instruction.
- ~F - Indicates source is the accumulator not used as the dest
- --- - Indicates field is unused

Chart 1 - Basic Data ALU, Destination is "F"

This chart is used to encode MOST non-multiply Data ALU instructions where the result of the operation is stored in one of the accumulators, A or B, i.e. is of the form "NONMPY_DALUOP <src>,F".

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "F" bit.

bbb b b iii i i ttt t t 654 2 0		KKK --- | | |							
KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KK0 000 0 0	~F	ADD	<<NA>>	TFR	<<NA>>	SUB	<<NA>>	CMP	<<NA>>
KK1 000 0 0	Y	<<NA>>	ADD	<<NA>>	--	<<NA>>	SUB	<<NA>>	--
KKK 001 0 0	F	DECW	--	NEG	NOT	RND	--	TST	--
KKK 010 0 0	F	--	--	ABS	--	--	--	--	--
KKK 011 0 0	F	INCW	--	CLR	--	ASL	ROL	ASR	ROR
KKK 100 0 0	X0	ADD	OR	TFR	--	SUB	AND	CMP	EOR
KKK 101 0 0	Y0	ADD	OR	TFR	--	SUB	AND	CMP	EOR
KKK 110 0 0	--	--	--	--	--	--	--	--	--
KKK 111 0 0	Y1	ADD	OR	TFR	--	SUB	AND	CMP	EOR

Note that there are nine rows above. This is because the entry for "JJJ" = 000 is broken into two different rows - one where the LSB of "KKK" is "0" (source is "~F") and one row where the LSB is "1" (source is "Y") .

Chart 2 - Basic Data ALU, Destination is "DD"

This chart is used to encode MOST non-multiply Data ALU instructions where the result of the operation is stored in one of the data regs, X0, Y0 or Y1, i.e. is of the form "NONMPY_DALUOP <src>,DD".

This chart encodes both the arithmetic operation and source register

for the operation. The destination is encoded with the "FF" bits.

bbb b b		KKK
iii i i		---
ttt t t		|
... . .		|
654 2 0		|

KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KKK 000 0 1	B1	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 001 0 1	F	DECW	--	--	NOT	--	--	--	--
KKK 010 0 1	A1	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 011 0 1	F	INCW	--	--	--	*	ROL	ASR	ROR
KKK 100 0 1	X0	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 101 0 1	Y0	ADD	OR	--	--	SUB	AND	CMP	EOR
KKK 110 0 1	--	--	--	--	--	--	--	--	--
KKK 111 0 1	Y1	ADD	OR	--	--	SUB	AND	CMP	EOR

* For 16-bit destinations, "asl" is identical to "lsl". Thus, if a user has "asl x0" in his program, it should instead assemble into "lsl x0". Always disassembles as "lsl x0".

Chart 3 - Supplemental Data ALU, Destination is "F"

This chart is used to encode A FEW non-multiply Data ALU instructions where the result of the operation is stored in one of the accumulators, A or B, i.e. is of the form "NONMPY_DALUOP <src>,F".

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "F" bit.

bbb b b		KKK
iii i i		---
ttt t t		|
... . .		|
654 2 0		|

KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KK0 000 1 0	~F	--	<<NA>>	<<Tc>>	<<NA>>	--	<<NA>>	--	<<NA>>
KK1 000 1 0	Y	<<NA>>	ADC	<<NA>>	<<Tc>>	<<NA>>	SBC	<<NA>>	--
KKK 001 1 0	F	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 010 1 0	F	--	--	<<Tc>>	<<Tc>>	--	--	--	--
KKK 011 1 0	F	--	--	<<Tc>>	<<Tc>>	--	LSL	NORM	LSR
KKK 100 1 0	X0	DIV	--	<<Tc>>	<<Tc>>	--	--	--	--

KKK 101 1 0	Y0	DIV	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 110 1 0	--	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 111 1 0	Y1	DIV	--	<<Tc>>	<<Tc>>	--	--	--	--	--

Note that there are nine rows above. This is because the entry for "JJJ" = 000 is broken into two different rows - one where the LSB of "KKK" is "0" (source is "~F") and one row where the LSB is "1" (source is "Y") .

Tcc instructions that occupy space on this chart are Tcc instructions where the "Z" bit is a "0". This corresponds to Tcc instructions of the form "tcc <reg>,F", i.e., without an AGU register transfer.

Chart 4 - Supplemental Data ALU, Destination is "DD"

This chart is used to encode A FEW non-multiply Data ALU instructions where the result of the operation is stored in one of the data regs, X0, Y0 or Y1, i.e. is of the form "NONMPY_DALUOP <src>,DD".

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "FF" bits.

bbb b b		KKK
iii i i		---
ttt t t		|
... . .		|
654 2 0		|

KKK JJJ h F	SRC	000	001	010	011	100	101	110	111
KK0 000 1 1	B1	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 001 1 1	DD	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 010 1 1	A1	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 011 1 1	DD	--	--	<<Tc>>	<<Tc>>	--	LSL	--	LSR	
KKK 100 1 1	X0	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 101 1 1	Y0	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 110 1 1	--	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--
KKK 111 1 1	Y1	--	--	<<Tc>>	<<Tc>>	--	--	--	--	--

Tcc instructions that occupy space on this chart are Tcc instructions where the "Z" bit is a "1". This corresponds to Tcc instructions of the form "tcc <reg>,F r0,r1", i.e., with an AGU register transfer.

yyyyy:

The "yyyyy" field is used to determine the operand encoding and destination operand definitions for data ALU instructions where one source operand is not a Data ALU register. It is described as "010" type instructions because all instructions in this class begin with "010" in bits 15-13.

For instructions of this type, the destination is always specified with the "fff" field.

yyyyy Operation

```
            -----       ------------
            00fff       ADD <src>,fff
            10fff       SUB <src>,fff
            11fff       CMP <src>,fff
            01100       DEC <dst>     NOTE: src and dst is a memory location, not a reg
            01101       INC <dst>     NOTE: src and dst is a memory location, not a reg
            0111x       <Available>

            DALU3OP2 - Shifting and Multiplication Encoding Information
            ----------------------------------------------------

DALU3OP2:
---------
IIII: ()
            Specifies Integer Multiplication, Signed*Uns, and Shifting Instructions

                        IIII        Operation
                        ----       ---------
                        1000        MPYsu
                        1100        MACsu
                        0010        IMPY16
                        1001        LSRR                  (multibit logical right
shift)
                        1101        LSRAC     (used for shifting 32-bit values)
                        0001        ASRR                  (multibit arithm  right
shift)
                        0101        ASRAC     (multibit arithm  right shift w/ acc)
                        0011        ASLL or LSLL          (multibit arithm  left
shift)
                        ----
                        ^^^^
                        ||||
                        |||+--- Indicates no shifting or shifting
                        ||+---- Shift shift dirn and whether LSP goes to DXB1
                        |+----- Selects mpy vs mac operation
                        +------ Selects signed*signed vs signed*unsigned

     Note: no inversion of multiplier result or rounding is allowed.

     NOTE: All of the above allow FFF as a destination EXCEPT
               LSRAC and ASRAC which only allow F as a destination,
               and LSLL which only allows X0, Y0, and Y1 as destinations.

               Although the LSLL only allows 16-bit destinations, there is
               the ASLL instruction which performs exactly the same operation
               and allows an accumulator as well as a destination.

Single Parallel Move Encodings:
===============================

P1DALU:
-------
x:
kk:
jjj:
      P1DALU operation and source register encodings (xkkjjj)
            x kk jjj
            - -- ---
            0 KK JJJ   - KK  specifies the arithm operation for non-multiply instrs
                           - JJJ specifies one source operand         for non-multiply
instrs
                                    (kk becomes KK when x=0)
                                    (jjj becomes JJJ when x=0)
            1 LL QQQ   - LL  specifies the arithm operation for       multiply instrs
                         - QQQ specifies one source operand         for
            multiply instrs
                                    (kk becomes LL when x=1)
                                    (jjj becomes QQQ when x=1)

JJJ:
     Specifies the source registers for the "non-multiply" P1DALU class
     of instructions as well as the Tcc instruction.
```

```
JJJ          Source register
---          ---------------
000          ~F
001          F               (not used by the Tcc instruction)
01x          F               (not used by the Tcc instruction)
01x          F               (not used by the Tcc instruction)
100          X0
101          Y0
110          (reserved for X1)
111          Y1
```

KK: ()

Chart 5 - Single Parallel Move Data ALU, Destination is "F"

This chart is used to encode all of the non-multiply arithmetic operations with a SINGLE PARALLEL MOVE, where the result of the operation is stored in one of the accumulators, A or B. In this case, the instruction is of the following form

"NONMPY_DALUOP <src>,F <single_pll_mov>"

This chart encodes both the arithmetic operation and source register for the operation. The destination is encoded with the "F" bit.

bbb iii ttt ... 654	KK

KK JJJ	SRC	00	01	10	11
KK 000	~F	ADD	TFR	SUB	CMP
KK 001	F	DECW	NEG	RND	TST
KK 010	F	--	ABS	--	--
KK 011	F	INCW	CLR	ASL	ASR
KK 100	X0	ADD	TFR	SUB	CMP
KK 101	Y0	ADD	TFR	SUB	CMP
KK 110	--	--	--	--	--
KK 111	Y1	ADD	TFR	SUB	CMP

Note that this chart is simply extraced from the above chart where bit_2 == 0 and bit_0 == 0. In this case, only the even values within the "KKK" field are retained.

Dual Parallel Read Encodings:

P2DALU:

x: ()
uu: ()
jj: ()

P2DALU operation and source register encodings (xuujj)

```
x uu jj
- -- --
0 UU GG   - UU specifies the arithm operation for non-multiply instrs
              - GG specifies one source operand   for non-multiply instrs
                   (uu becomes UU when x=0)
                   (jj becomes GG when x=0)
1 LL QQ   - LL specifies the arithm operation for     multiply instrs
```

- QQ specifies one source operand for multiply instrs
(uu becomes LL when x=1)
(jj becomes QQ when x=1)

GG: ()
UU: ()
Specifies "non-multiply" P2DALU instructions and operands.

x	UU	GG	Non-Multiply Operation	DALU Source Register
0	00	JJ	ADD	JJ
0	10	JJ	SUB	JJ
0	01	--	MOVE	<none>
0	11	--	(reserved)	<none>

JJ: ()
Specifies the source registers for the "non-multiply" P2DALU instructions.

JJ	source register
00	X0
01	Y0
10	(reserved for X1)
11	Y1

LL: ()

LL	Multiplication Operation
00	MPY + (neither operand inverted)
01	MAC + (neither operand inverted)
10	MPYR + (neither operand inverted)
11	MACR + (neither operand inverted)

QQ: ()
Input registers for the "multiply" P2DALU instructions.

QQ	Multiplier inputs
00	Y0,X0
01	Y1,X0
10	(reserved for X1,Y0)
11	Y1,Y0

vvv: (9,6,0)
Specifies the destination registers for the dual X memory parallel read instruction WITH arithmetic operation.

vvv	1st read	2nd access	
000	X:(R0),Y0	X:(R3)+,X0	-
010	X:(R0),Y0	X:(R3)-,X0	-
100	X:(R0),Y1	X:(R3)+,X0	-
110	X:(R0),Y1	X:(R3)-,X0	-
001	X:(R1),Y0	X:(R3)+,X0	-
011	X:(R1),Y0	X:(R3)-,X0	-
101	X:(R1),Y1	X:(R3)+,X0	-
111	X:(R1),Y1	X:(R3)-,X0	-

```
---
^^^
|||
||+---  (effectively an "r" bit for 1st read - R0 vs R1)
|+----  (effectively an "m" bit for 2nd read - (R3)+ vs (R3)-)
+-----  (effectively a  "V" bit for 1st read - Y0 vs Y1)
```

NOTE: Above table does not show any addressing mode information for the 1st read. See the "m" field for this information. The above table does contain addressing mode info for the second access as seen above.

Move Register Field Definitions:
==========

HHH: destination registers for the "P1DALU X:<ea_m>,HHH" instruction.

HHH	register
000	X0
001	Y0
010	(reserved for X1)
011	Y1
100	A
101	B
110	A1
111	B1

RRR: ()

RRR	register
000	R0
001	R1
010	R2
011	R3
111	SP

HHHH: destination registers for the "#xx,HHHH" instruction.

HHHH	register
0HHH	X0, Y0, (reserved for X1), Y1, A, B, A1, B1
10RR	R0, R1, R2, R3
11NN	ND (dst only), N, NOREG (src and dst), (reserved)

DDDDD: - specifies destination registers for "ddddd,DDDDD"
- specifies source/destination registers for other DDDDD moves
- NOTE that ordering is different than "ddddd"

DDDD	D	register
0HHH	0	X0, Y0, (reserved for X1), Y1, A, B, A1, B1
10RR	0	R0, R1, R2, R3
11xx	0	ND (dst only), N, NOREG, (reserved)
00xx	1	A0, B0, A2, B2
01xx	1	M01, (res), (res), SP
1xxx	1	OMR, PINC/PAMAS, (res), HWS, (res, used as LC), SR, LC, LA

ddddd: - specifies source registers for the move ddddd,DDDDD instruction.
- specifies source registers for the DO/REP ddddd instruction.
- specifies source/destination registers for bitfield instructions
- NOTE that ordering is different than "DDDDD"

ddddd	register
00HHH	X0, Y0, (reserved for X1), Y1, A, B, A1, B1
100RR	R0, R1, R2, R3
101xx	(res-ND), N, (res-NOREG), (res)
010xx	A0, B0, A2, B2
011xx	M01, (res), (res), SP
11xxx	OMR, PINC/PAMAS, (res), HWS, (res, used as LC), SR, LC, LA

Special registers which need to be detected:

1110 0 NOREG - Prevents external bus cycle, or perhaps any memory cycle from occurring. Required because the chip may not own the bus. Forces access internal, or perhaps even disables prxrd/prxwr.

Occurs on read from reg only. Note there is no register actually present. It applies to reads from the register because this is true during an LEA where no memory cycle is desired, but this is not true for a TSTW instruction, which must actually perform a memory cycle and move the data onto the cgdb.

1100 0 ND - Accesses "N" register but also asserts pmnop. Occurs on write to reg only.

```
               1100 0   ND      - Prevents interrupts, force adr onto eab,
                                    regardless of whether it's on-chip or not.
                                    Note there is no actual register.  Asserts
                                    a new ctrl signal, pmdram.  Occurs on reads
                                    from reg only.  Used to be the DRAM register.
                                    Must disable xmem writes, similar to reads
                                    from NOREG.  Force the access internal.
               1011 1   HWS     - Any reads of this register must "pop" the
                                    HWS and HWSP.  Any writes to this register
                                    must "push" the HWS and HWSP.

RR:         RR          register
            --          --------
            00          R0
            01          R1
            10          R2
            11          R3

AGU (Address Generation Unit) Instruction Field Definitions:
============================================================

MM: specifies addressing modes for the "X:<ea_MM>,DDDDD" instruction.
            MM          addressing mode
            --          ---------------
            00          (Rn)+       or    (SP)+
            01          (Rn)+N  or    (SP)+N
            10          (Rn)-       or    (SP)-
            11          (Rn)        or    (SP)      (LEA cannot use this combination)

m:  specifies addressing modes of "P1DALU" and "P2DALU"
            m           addressing mode
            -           ---------------
            0           (Rn)+
            1           (Rn)+N

W:
            W           move direction for memory moves
            -           ------------------------------
            0           register -> memory
            1           memory   -> register

w:          w           DALU result
            -           -----------
            0           written back to memory   (not allowed for CMP or SUB instrs)
            1           remains in register

Immediates and Absolute Address Instruction Field Definitions:
==============================================================

AAA:
            Upper 3 address bits for JMP, Jcc, and JSR instructions.

BBBBBBB:
            7-bit signed integer. For #xx,HHHH and DALU #xx,F instructions.

BBBBBB:
            6-bit unsigned integer. For DO/REP #xx instruction.

AAAAAA:
              6-bit positive offset for X:(R2+xx) addressing mode.
              Allows positive offsets:  0 to  63

aaaaaa:
              6-bit negative offset for X:(SP-xx) addressing mode.
              Allows negative offsets: -1 to -64

Aaaaaaa:
              7-bit offset for MOVE, DALU & Bitfield using X:(SP-#xx), X:(R2+#xx)
              and Bcc <aa> instructions:
                 A = 0   =>   X:(R2+#xx)   allows positive offsets:  0 to  63
                 A = 1   =>   X:(SP-#xx)   allows negative offsets: -1 to -64
```

```
              For Bcc, "A" specifies the sign-extension.
              RESTRICTION: Aaaaaaa must never be all zeros for the Bcc instruction.

Ppppppp:
              7-bit absolute address for MOVE, DALU, & Bitfield on X:<pp> instr
              It is sign-extended to allow access to both the peripherals and
              the 1st 64 locations in X-memory.

Other Instruction Field Definitions:
====================================

Z: specifies the parallel moves of the address pointers in a Tcc instruction.
            Z           move
            -           ----
            0           R0->R0   (i.e., no transfer occurs in the AGU unit)
            1           R0->R1   (AGU transfers R0 register to R1 if condition true)

            For the case where Z=0, the assembler will not look for a field
            such as "teq x0,a  r0,r0".  Instead, the AGU register transfer
            will be suppressed, such as in ""teq x0,a".

E:          E           instruction
            -           -----------
            0           DO
            1           REP

tt:         tt          instruction
            -           -----------
            00          STOP
            01          WAIT
            10          SWI
            11          ILLEGAL

BITFIELD:
UUU: specifies bitfield/branch-on-bit instructions
            UUU         operations
            ---         ----------
            000         BFCLR
            001         BFSET
            010         BFCHG
            011         MOVE   (used by "move #iiii,<ea>")

            100         BFTSTL
            110         BFTSTH
            101         BRCLR       (modifies carry bit)
            111         BRSET       (modifies carry bit)

            0xx         last word = iiiiiiiiiiiiiiii
            1x0         last word = iiiiiiiiiiiiiiii
            1x1         last word = iiiiiiiiUAaaaaaa

            (note: this is the 3rd word, not 2nd, for BF/BR #xxxx,X:xxxx)

                        iiiiiiiiiiiiiiii = 16-bit immed mask
                        iiiiiiii     = 8-bit  immed mask for upper or lower byte
                        U = 1          selects upper byte
                        U = 0          selects lower byte
                        Aaaaaaa      = 7-bit relative branch field

                        Note: UAaaaaaa is not available to the BFTSTH, BFTSTL instrs

            The ANDC, ORC, EORC, and NOTC are instructions which fall directly
            onto the bitfield instructions.  They are mapped as follows:

                  ANDC is identical to a BFCLR with the mask inverted
                  ORC  is identical to a BFSET (mask not inverted)
                  EORC is identical to a BFCHG (mask not inverted)
                  NOTC is identical to a BFCHG with the mask set to $FFFF

CC-C: ()
      Specifies conditions for the Tcc instructions:
      (in this case, "CC" falls onto C10 of CCCC, "C" falls onto C2, C3 is "0")
```

```
            CC-C        condition
            ----        ---------
            00 0        cc
            01 0        cs
            10 0        ne
            11 0        eq

            00 1        ge
            01 1        lt
            10 1        gt
            11 1        le
```

CCCC: ()
Specifies conditions for the Jcc, JScc, and Bcc instructions

```
            CCCC        condition - for encode7
            ----        ---------
            0000        cc (same as "hs", unsigned higher or same)
            0001        cs (same as "lo", unsigned lower)
            0010        ne
            0011        eq
            0100        ge
            0101        lt
            0110        gt
            0111        le

            10**        ALWAYS TRUE condition              (PLAs decode this)

            1001        ALWAYS - JMP, BRA, JSR             (value used by assembler)
            1011        (reserved -could be used for delayed)
            1010        (reserved)
            1000        (reserved)
            1100        hi (unsigned higher)
            1101        ls (unsigned lower or same)
            1110        nn
            1111        nr
```

Unusual Instruction Encodings:
==============================
Encoding of "ADD fff,X:<aa>" and "ADD fff,X:(sp-xx)":
There is an unusual trick used to encode these two instructions.
What is so unusual is that the first word of the two word
"ADD/SUB/CMP fff,X:<aa>" instruction is identical to the one
word encoding of the "ADD/SUB/CMP X:<aa>,fff" instruction.
It is also true the first word of the two word
"ADD/SUB/CMP fff,X:(sp-xx)" instruction is identical to the one
word encoding of the "ADD/SUB/CMP X:(sp-xx),fff" instruction.

What makes these instructions differ is the encoding of the instruction
immediately following the first word. The rules are listed below.

Encoding Rules:

ADD X:<aa>,fff:
- 1st word - Simply uses the one word encoding for ADD X:<aa>,fff
- 2nd word - Any valid DSP56800 instruction, which by definition will not be the following reserved hex value: $E042.
  Note that this value is reserved in the DSP56800 bit encoding map.

ADD X:(SP-xx),fff:
- 1st word - Simply uses the one word encoding for ADD X:(SP-xx),fff
- 2nd word - Any valid DSP56800 instruction, which by definition will not be the following reserved hex value: $E042.
  Note that this value is reserved in the DSP56800 bit encoding map.

ADD X:xxxx,fff:
- 1st word - 1st word of encoding uses ADD X:xxxx,fff

with the "w" bit set to "1"
- 2nd word - second word of encoding contains the 16-bit absolute address

ADD fff,X:<aa>:
- 1st word - 1st word of this instruction uses the one word encoding for the ADD X:<aa>,fff instruction.
- 2nd word - 2nd word of this instruction is simply set to $E042.

ADD fff,X:(SP-xx):
- 1st word - 1st word of this instruction uses the one word encoding for the ADD X:(SP-xx),fff instruction.
- 2nd word - 2nd word of this instruction is simply set to $E042.

ADD fff,X:xxxx:
- 1st word - 1st word of encoding uses ADD X:xxxx,fff with the "w" bit set to "0"
- 2nd word - second word of the instruction contains the 16-bit absolute address

Thus, the presence of the hex value $E042 in the instruction immediately after a "ADD X:<aa>,fff" or "ADD X:(sp-xx),fff" indicates that the instruction is really an "ADD fff,X:<aa>" or "ADD fff,X:(sp-xx)" instruction. These later two instructions encode as two word instructions using the technique described above.

Note that this encoding (where the destination is a memory location) is NOT allowed for the SUB or CMP instructions. It is only allowed for the ADD instruction.

Encoding of LEA:

There is a trick used for encoding the LEA instruction. The trick is used in several different places within the opcode map and is simply this - anytime a MOVE instruction uses "NOREG" (located in the HHHH or DDDDD field) as a source register, the instruction is no longer interpreted as a MOVE instruction. Instead it operates as an LEA instruction. Thus, the syntax for the instruction available to the user is "LEA", but the actual bit encoding uses the MOVE instruction where the source register is "NOREG":

DSP56800 Instruction		Encoded As:
LEA (Rn)+	=>	MOVE NOREG,X:(Rn)+
LEA (Rn)-	=>	MOVE NOREG,X:(Rn)-
LEA (Rn)+N	=>	MOVE NOREG,X:(Rn)+N
LEA (R2+xx)	=>	MOVE NOREG,X:(R2+xx)
LEA (Rn+xxxx)	=>	MOVE NOREG,X:(Rn+xxxx)
LEA (SP)+	=>	MOVE NOREG,X:(SP)+
LEA (SP)-	=>	MOVE NOREG,X:(SP)-
LEA (SP)+N	=>	MOVE NOREG,X:(SP)+N
LEA (SP-xx)	=>	MOVE NOREG,X:(SP-xx)
LEA (SP+xxxx)	=>	MOVE NOREG,X:(SP+xxxx)

CAREFUL: LEA must NOT write to a memory location!
NOTE: LEA not allowed for (Rn) or (SP).

Encoding of TSTW:

There is a trick used for encoding the TSTW instruction. The trick is used in several different places within the opcode map and is simply this - anytime a MOVE instruction uses "NOREG" (located in the HHHH or DDDDD field) as a dest register, the instruction is no longer interpreted as a MOVE instruction. Instead it operates as a TSTW instruction. Thus, the syntax for the instruction available to the user is "TSTW", but the actual bit encoding uses the MOVE instruction where the destination register is "NOREG":

DSP56800 Instruction		Encoded As:
TSTW X:<aa>	=>	MOVE X:<aa>,NOREG
TSTW X:<pp>	=>	MOVE X:<pp>,NOREG
TSTW X:xxxx	=>	MOVE X:xxxx,NOREG

```
    TSTW X:(Rn)          =>       MOVE  X:(Rn),NOREG
    TSTW X:(Rn)+         =>       MOVE  X:(Rn)+,NOREG
    TSTW X:(Rn)-         =>       MOVE  X:(Rn)-,NOREG
    TSTW X:(Rn)+N        =>       MOVE  X:(Rn)+N,NOREG
    TSTW X:(Rn+N)        =>       MOVE  X:(Rn+N),NOREG
    TSTW X:(Rn+xxxx)     =>       MOVE  X:(Rn+xxxx),NOREG
    TSTW X:(R2+xx)       =>       MOVE  X:(R2+xx),NOREG
    TSTW X:(SP)          =>       MOVE  X:(SP),NOREG
    TSTW X:(SP)+         =>       MOVE  X:(SP)+,NOREG
    TSTW X:(SP)-         =>       MOVE  X:(SP)-,NOREG
    TSTW X:(SP)+N        =>       MOVE  X:(SP)+N,NOREG
    TSTW X:(SP+N)        =>       MOVE  X:(SP+N),NOREG
    TSTW X:(SP+xxxx)     =>       MOVE  X:(SP+xxxx),NOREG
    TSTW X:(SP-xx)       =>       MOVE  X:(SP-xx),NOREG
    TSTW <register>      =>       MOVE  ddddd,NOREG

NOTE: TSTW (Rn)- is not encoded in this manner, but instead
      has its own encoding allocated to it.

NOTE: TSTW HWS is NOT allowed.  All other on-chip registers
      are allowed.

IMPORTANT NOTE: TSTW can be done on any other instruction which
   allows a move to NOREG.  Note this doesn't make sense for LEA.

NOTE: TSTW F (operates on saturated 16 bits) differs
      from TST F (operates on full 36/32 bit accumulator)

NOTE: TSTW P:() is NOT allowed.

Encoding of POP:
The encoding of the POP follows the simple rules below.

   DSP56800 Instruction        Encoded As:
   --------------------        -----------
    POP  <reg>           =>       MOVE X:(SP)-,<reg>
    POP                  =>       LEA  (SP)-

In the first case, a register is explicitely mentioned, whereas in
the second case, no register is specified, i.e., just removing a value
from the stack.

NOTE: There is no PUSH instruction, but it is easy to write
      a simple two word macro for PUSH.

Encoding of CLR:
The encoding for a CLR on anything other than A or B
should encode into the following: "move #0,<reg>".
Allows the following instructions to be recognized by the assembler:
   CLR DD       (DD = x0,y0,y1)
   CLR F1       (F1 = a1,b1)
   CLR RR       (DD = r0,r1,r2,r3)
   CLR N
Note that no parallel move is allowed with these.
Note also that CLR F sets the condition codes,
whereas CLR on DD, F1, RR, or N does NOT set the condition codes.

Encoding of ENDDO:
The ENDDO instruction will be encoded as "MOV HWS,NOREG".

Encoding of the Tcc Instruction:
--------------------------------
The Tcc instruction is somewhat difficult to understand because it's encoding
overlays the encodings of some Data ALU instructions when Bit 2 of the opcode
is a "1".  It is overlayed obviously so that for a particular bit pattern,
there is only one unique instruction present.  Reference to this can be seen
with the "<<Tc>>" entry found within Charts 3 and 4 below.  Use the definition

      "0110 11CC FJJJ 01CZ         Tcc  JJJ,F  [R0->R1]"

to encode this instruction.
```

```
================================================================================
================================================================================

Restrictions:
-------------
   - The HWS register cannot be specified as the loop count for a DO or
     REP instruction.  Likewise, no bitfield operations (BFTSTH, BFTSTL,
     BFSET, BFCLR, BFCHG, BRSET, BRCLR) can operate on the HWS register.
     Note, however, that all other instructions which access ddddd, including
     "move #xxxx,HWS" and TSTW, can operate on the HWS register.
   - The following registers cannot be specified as the loop count for a DO or
     REP instruction - HWS, M01, SR, OMR.
   - The "lea" instruction does NOT allow the (Rn) addressing mode, i.e.,
     it only allows (Rn)+, (Rn)-, (Rn)+N, (Rn+xxxx), (R2+xx), and (SP-xx)
   - Cannot do a bitfield set/clr/change on "ND" register, i.e., the bitfield
     instruction cannot be immediately followed by an instruction which uses
     the "N" register in an addressing mode.
                       bfclr      #$1234,n
                       move       x:(r0+n),x0            ; illegal - needs one NOP
     Special care is necessary in hardware loops, where the instruction at
     LA is followed by the instruction at the top of the loop as well as the
     instruction at LA+1.
   - Cannot move a long immediate value to the "ND" register.  This is because
     the long immediate move is implemented similar to the bitfield instrs.
             move       #$1234,n               ; long immediate
             move       x:(r0+n),x0            ; ILLEGAL - needs one NOP

             move       #$4,n                              ; short immediate, uses ND
register
             move       x:(r0+n),x0            ; ALLOWED since uses short immediate
   - The value "0000000" is not allowed for Bcc.
     In addition, this same value is not allowed as the relative offset
     for a BRSET or BRCLR instruction.
   - The value "0" is not allowed for the DO #xx instruction.
     If this case is encountered by the assembler, it should not be accepted.
   - Jumps to LA and LA-1 of a hardware loop are not allowed.  This also
     applies to the BRSET and BRCLR instructions.
   - A NORM instruction cannot be immediately followed by an instruction
     which uses the Address ALU register modified by the NORM instruction
     in an addressing mode.
                       norm       r0,a
                       move       x:(r0)+,x0             ; illegal - needs one NOP
     Special care is necessary in hardware loops, where the instruction at
     LA is followed by the instruction at the top of the loop as well as the
     instruction at LA+1.
   - Only positive values less than 8192 can be moved to the LC register.
   - Cannot REP on any multiword instruction or any instruction which
     performs a P:() memory move.
   - Cannot REP on any instruction not allowed on the DSP56100.
   - IF a MOVE or bitfield instruction changes the value in R0-R3 or SP,
     then the contents of the register are not available for use until the
     second following instruction, i.e., the immediately following instruction
     should not use the modified register to access X memory or update an
     address.  This restriction does NOT apply to the N register or the
     (Rn+xxxx) addressing mode as discussed below.
   - For the case of nested looping, it is required that there are at least
     two instruction cycles after the pop of the LC and LA registers before
     the instruction at LA for the outer loop.
   - A hardware DO loop can never cross a 64K program memory boundary, i.e.,
     the DO instruction as well as the instruction at LA must both reside
     in the same 64K program memory page.
   - Jcc, JMP, Bcc, BRA, JSR, BRSET or BRCLR instructions are not allowed in
     the last two locations of a hardware do loop, i.e., at LA, and LA-1.
     This also means that a two word Jcc, JMP, or JSR instruction may not have
     its first word at LA-2, since its second word would then be at LA-1, which
     is not allowed.

Restrictions Removed:
---------------------
   - The following instruction sequence is NOW ALLOWED:
                       move       <>,lc                  ; move anything to LC reg
                       do         lc,label               ; immediately followed by DO
```

This was not allowed on the 56100 family due to its internal pipeline.

- An AALU pipeline NOP is not required in the following case:

```
                    move        <>,Rn                   ; same Rn as in following
instr
                    move        X:(Rn+xxxx),<>          ; OK, no NOP required!

                    move        <>,Rn                   ; same Rn as in following
instr
                    move        <>,X:(Rn+xxxx)          ; OK, no NOP required!
```

In this case, there will NOT be an extra instruction cycle inserted because any move with the X:(Rn+xxxx) or X:(SP+xxxx) addressing mode is already a 3 Icyc instruction.

- An AALU pipeline NOP is not required in the following case:

```
                    move        <>,Rn                               ; same Rn as in
following instr
                    lea         (Rn+xxxx)               ; OK, no NOP required!
```

In this case, there will NOT be an extra instruction cycle inserted because any lea with the (Rn+xxxx) or (SP+xxxx) addressing mode is already a 2 Icyc instruction.

- An AALU pipeline NOP is not required in the following case:

```
                    move        <>,N
                    move        X:(Rn+N),<>             ; OK, no NOP required!

                    move        <>,N
                    move        <>,X:(Rn+N)             ; OK, no NOP required!

                    move        <>,N
                    move        <>,X:(Rn)+N             ; OK, no NOP required!

                    move        <>,N
                    move        X:(Rn)+N,<>             ; OK, no NOP required!
```

In this case, there WILL be an extra instruction cycle inserted and the assembler will use the ND register, not the N register.