

Technical Notes on using Analog Devices' DSP components and development tools

Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Porting Code From ADSP-218x To ADSP-219x

Last modified 5/17/01

This tech-note is intended for existing users of the ADSP-218x, who are familiar with the architecture and instruction set of the ADSP-218x, and plan to upgrade their design to the ADSP-219x. It is divided into two main sections - the first describes the enhancements and differences between the ADSP-218x family of DSPs and the ADSP-219x in terms of the DSP architecture and the instruction set, while the second section illustrates the new ELF assembler and linker formats with the help of an example.

One of the goals in designing the ADSP-219x has been to keep its instruction-set and assembly syntax as closely compatible with the ADSP-21xx family of DSPs as possible. The exceptions (which have been kept to a minimum and intended to be as minimally intrusive to the customer as possible) will be documented in this Application note. The changes allow for a DSP architecture that is more C-friendly, resulting in a more efficient C compiler. It also allows users additional instructions to improve efficiency of assembly code. These changes facilitate the increased core processor operating speeds.

DIFFERENCES IN ARCHITECTURE

1. Unified Memory Space

The ADSP-218x had two separate memory spaces – PM and DM. The ADSP-219x on the other hand, has a unified memory space with separate memory blocks to differentiate between 24 and 16-bit memory. For example, the first GP member of the ADSP-219x family, ADSP-2192 has dual DSP cores, with core 0 having 16Kx24 and 64Kx16 words of on-chip memory, and core 1 having 16Kx24 and 32Kx16 words of on-chip memory. A detailed description of the ADSP-2192 memory map can be obtained from the ADSP-2192 datasheet. Figure 1 shows the memory organization of the ADSP-2192.

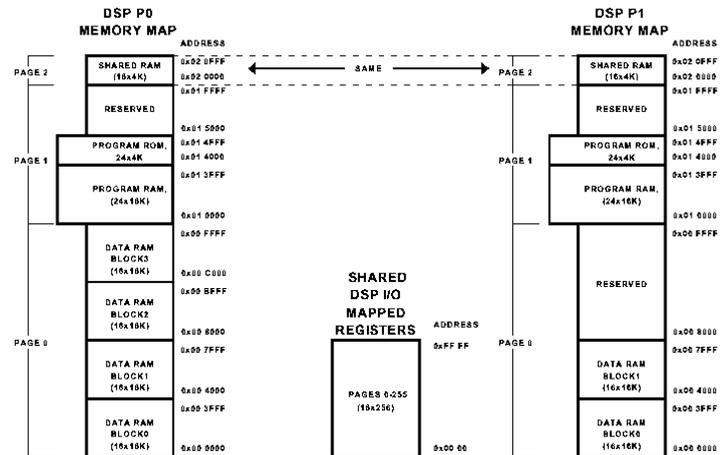


Figure 1. ADSP-2192 memory map

The program sequencer will be able to fetch two data operands from any two memory blocks in a single cycle. While having a unified memory map allows for more flexible memory addressing, it's important to understand that on the ADSP-219x, the addresses PM(0x0000) and DM(0x0000) access the same physical memory location. The first syntax fetches/writes 24-bit data, while the second syntax fetches/writes 16-bits of data.

A linker description file (LDF) provided at the end of this tech-note describes the memory map for the ADSP-2192.

2. 16-bit DAG Registers and Memory Paging

The ADSP-219x has a paged architecture that uses 16-bit DAG Registers to access 64K pages. But since the address buses in the ADSP-219x are 24 bits wide, there are also two new page registers, DMPG1 and DMPG2, which are used to store the upper 8 bits of the 24-bit address. DMPG1 works with DAG registers I0-I3 and DMPG2 works with I4-I7, respectively. Page registers can be initialized as shown in the following example

```
I2 = 0x3456;
DMPG1 = 0x12; /* Page register can be set to an
absolute page value...*/
DMPG2 = page(data_buffer); /* ..OR could also be
initialized to a buffer */
L2 = 0;
AX0 = 0xaaaa;
AR = MR1 - AX0;
DM(I2,M2) = AR;
```

Note that program execution continues linearly through memory. Local jumps, loops, and calls within page boundaries do not affect page registers.

3. Increased variety in DAG Addressing Modes

The ADSP-219x architecture has been enhanced to provide added flexibility in DAG addressing modes. There are four new enhanced addressing modes such as

- Pre-modify-without update addressing (in addition to the existing post-modify with update mode that existed on the ADSP-218x)

e.g., DM(M1,I0) = AR;

- Pre-modify and post-modify with an 8-bit 2's-complement immediate modify value instead of an M register

e.g., AX0 = PM(I5,-4);

- DAG modify with an 8-bit 2's complement immediate-modify value

e.g., MODIFY(I7,24);

Note: In terms of syntax, the ADSP-219x assembler supports a new and more intuitive "C-style" format in addition to the existing format as shown in the table below. It is hoped that this new representation will make code more readable and easier to understand.

	Legacy	New
Post-modify	AX0 = DM(I5,M6);	AX0 = DM(I5 += M6); OR AX0 = DM(I5,M6);
Pre-modify	N/A	PM(I1 + -4) = MR2; OR PM(-4,I1) = MR2;

4. Increased Variety and Types of Jumps and Function Calls (Relative and Absolute/Long)

The ADSP-218x with its 16K words of accessible space only required a single form of conditional/unconditional jump/call instruction of the form

[IF COND] CALL <address>;

The address could either be an absolute 14-bit value provided in the instruction, or could be an indirect address pointed to by one of the DAG2 Index registers (I4, I5, I6, or I7).

However, the ADSP-219x has an available addressable space of 64K words. Hence, it provides a wider variety of conditional and unconditional jumps and calls, which may be either delayed or non-delayed. The available options are:

- 13-bit non-delayed or delayed relative conditional jump
- Conditional indirect jump or call with address pointed to by a DAG register. **In this case, the upper 8 bits of the address are stored in a "Jump Page Register" called IJPG, which is new to the ADSP-219x. Note that any one of the 8 DAG registers can be used for jump address.**
- 16-bit non-delayed or delayed relative unconditional jump or call
- 24-bit conditional non-delayed long jump or call

Jumps, calls and returns can take up to 4 DSP clock cycles if the branch is taken and no penalty if the branch is not taken (as explained in section 5)

Note that the use of delayed branches and jumps will effectively reduce the above latency by 2. This is because the two instructions following the jump are also executed.

5. The ADSP-219x Instruction Pipeline

The ADSP-219x has a six-stage Instruction pipeline, comprising the **L**(ook-ahead), **P**(re-fetch), **F**(etch), **A**(ddress), **D**(ecode), and **E**(xecute) stages. The pipeline is completely transparent from a user standpoint. Incorporated within the instruction pipeline is a two-stage memory pipeline. The additional depth in the pipeline means that memory is no longer double pumped leading to less power consumption. The added depth of the pipeline is also required to accommodate the increased operating speed of the processor. For a detailed description of the ADSP-219x pipeline, please refer to EE-Note EE-123 titled “An Overview of the ADSP-219x Pipeline”.

The added depth to the pipeline enables a programmer to use delayed branches and function calls. Figure 2 shows the pipeline structure for a delayed jump that is taken (Figure 2). Note that the two instructions immediately following the jump are executed, while the instructions further down, P and Q (corresponding to the **F** and **P** stages of the pipeline) are flushed. If the jump had not been taken, normal program execution would have continued without any lost cycles.

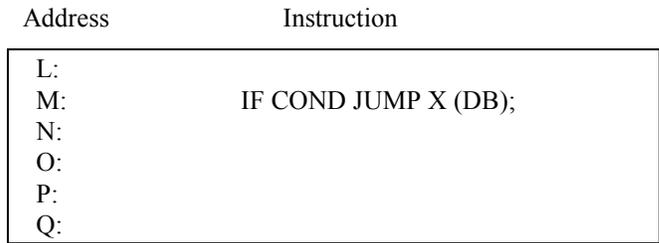


Figure 2

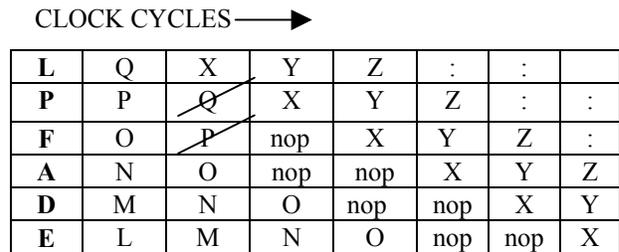
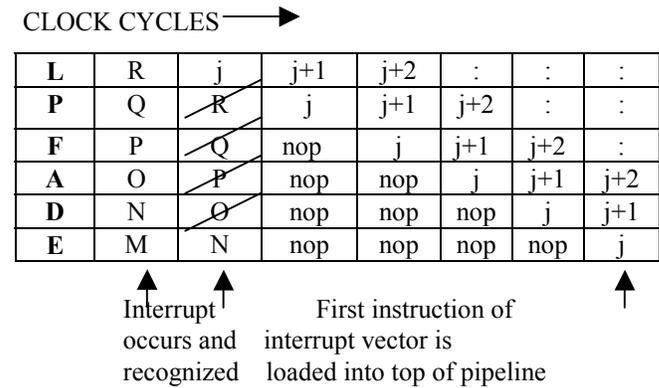


Figure 3

There is a 4-cycle delay in servicing an interrupt (Fig 4). On receiving an interrupt request, the DSP completes execution of the present instruction, flushes the pipeline, and fetches the first instruction of the interrupt vector.



where j = first instruction of ISR

Figure 4

6. Base Registers For Circular Buffers (Removes Restriction on Starting Addresses That Used To Exist On 218x)

The ADSP-219x uses base registers for addressing circular buffers. This removes existing hardware restrictions on the ADSP-218x with regards to the location of the starting/base address of circular buffers, thereby enabling any number of circular buffers to be declared. Further, the base registers are mapped to “register” space on the core. Switching between primary and secondary DAG register banks automatically also causes primary and secondary base registers to be enabled.

A slice of ADSP-218x initialization code that looks like:

```
...
.VAR/CIRC some_buffer[N];
...
I1 = ^some_buffer;
L1 = %some_buffer;
...
```

would now be written as:

```
...
.VAR some_buffer[N];
...
I1 = some_buffer;
L1 = length (some_buffer);
AR = I1;
REG(B1) = AR;
...
```

It is important to note that the ‘^’ character is no longer needed for initializing the Index register Ix, and also that the ‘%’ character used to initialize the Length register Lx, is now replaced by the ‘length’ qualifier.

7. System Control Register Space

ADSP-219x system registers (such as the DAG base registers discussed in the previous section) are now mapped to and reside in a separate register space on the ADSP-219x. The syntax “REG(*register*)” should now be used for accessing these registers.

8. Addition of SR as a Dual Accumulator for The MAC in place of the MAC Feedback Register (MF)

The SR register which was 32-bits wide on the ADSP-218x (comprising two 16-bit registers SR0 and SR1) has

now been expanded to 40 bits with the introduction of a new 16-bit register called SR2. The lower 8 bits of SR2 are used in shift operations, and SR2 can also be used as a full 16-bit scratch register. SR2 has replaced the MAC feedback register MF. Thus, the ADSP-219x now has two true 40-bit accumulators: MR and SR.

Instructions of the form

```
[IF cond] MF = MR + MX0*MY1 (UU);
```

will now become

```
[IF cond] SR = MR +MX0*MY1 (UU);
```

while instructions that used the MF register as a source can now use the 16-bit SR1 register. E.g.,

```
MR = MR + MX1*MF (SS);
```

will now become

```
1. MR = MR + MX1*SR1 (SS);
```

9. Differences In MSTAT Register And The Availability Of An Alternate Set Of DAGs

The MSTAT register, which was 7 bits wide on the ADSP-218x, is now an 8-bit register.

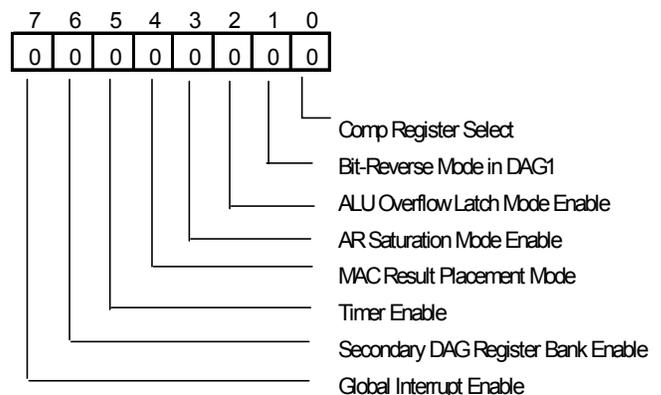


Figure 5

While the operation of bits 0-5 is unchanged, clearing or setting bit 6 now enables the user to switch between primary and secondary DAG register banks, respectively.

e.g.,
ENA SEC_DAG;

Setting or clearing bit 7 will let the user enable or disable global interrupts. The "Go Mode" as was defined for the ADSP-218x no longer exists for the ADSP-219x (the DSP is configured to always work in "Go Mode"). Multiple modes may be set/cleared in a single cycle, but mode enables and disables cannot be mixed in the same instruction. For example

```
ENA M_MODE, TIMER;
```

is valid, but

```
ENA M_MODE, DIS TIMER;
```

is not.

Changes to the MSTAT using these (type 18) mode control instructions have zero latency, meaning the results take effect the very next cycle. Note however, that there are still non-zero latencies when writing to the MSTAT register directly.

10. SE Is No Longer A DREG

With the introduction of a new DREG in the form of SR2, it became necessary to move one of the existing DREGs to a general core register space. This was the SE register, which is arguably the least used DREG. What this means is that multifunction instructions that used the SE as a general DREG or as a scratch register will now have to use one of the other DREGs as a scratch register. Multifunction instructions that used SE will have to be done the following way. For example,

```
SR = LSHIFT MR1 (HI), SE = DM (I6, M5);
```

will now be

```
SR = LSHIFT MR1 (HI);  
SE = DM (I6, M5);
```

It is important to note however that the functionality of the SE register has remained unchanged.

11. The **NORM** instruction differs slightly between the ADSP-219x and previous 16-bit, fixed-point DSP's in

the ADSP-21xx family. The difference can only be seen when performing "overflow normalization".

+ On the ADSP-219x, the NORM instruction checks only that (SE == +1) for performing the shift in of the AC flag (overflow normalization).

+ On previous ADSP-2100 family DSP's, the NORM instruction checks both that (SE == +1) and (AV == 1) before shifting in the AC flag.

The EXP(HIX) instruction always sets (SE = +1) when the AV flag is set, so only when NORM is used without a preceding EXP instruction is the implementation difference manifest.

12. IRPTL Register Instead Of IFC

The IRPTL Register has taken the place of what used to be the IFC Register on the ADSP-21xx.

The operation of the two registers is however identical, so instructions of the form

```
IFC = AX0;
```

or

```
IFC = 0x08;
```

will now become

```
IRPTL = AX0;
```

or

```
IRPTL = 0x08;
```

respectively.

The IRPTL register (which is similar in functionality to the IRPTL on the SHARC) also provides a way of latching interrupts, something the 21xx did not have.

13. Introduction Of the SV Condition (Programmed Using the CCODE Register and SWCOND)

The ADSP-219x provides a new arithmetic status condition, Shifter Overflow. A bit (bit 9) in a newly introduced register called the CCODE register tests for shifter overflow. This is done as follows:

```
CCODE = 0x09;
IF NOT SWCOND SR = LSHIFT MX0 (LO);
```

will shift the contents of MX0 into SR0 until SR overflows.

14. Using the CCODE Register and SWCOND To Test For The XIN POS/NEG Condition

The CCODE register also detects for the ALU condition IF POS/IF NEG (Bit 8 in this case). ADSP-218x code that used this condition, for example:

```
IF POS AR = ABS AX0;
```

would become

```
CCODE = 0x08;                                IF
NOT SWCOND AR = ABS AX0;
```

DIFFERENCES IN EXECUTABLE FILE FORMATS, INSTRUCTION SYNTAX

If writing new code for the ADSP-219x, users are encouraged to use the new assembler and linker format and syntax. For a detailed description of the ELF assembler and linker formats, please refer the ADSP-219x tools documentation.

15. Architecture files replaced by more powerful Linker Description (LDF) files

The new common syntax that the software tools are adopting have replaced the architecture files with the more powerful linker-description files. A detailed description of LDF files is beyond the scope of this application note, but details can be found in the VisualDSP User's guide and Reference.

16. Support of Legacy ADSP-218x Code

The ADSP-219x ELF assembler and linker however allow existing ADSP-218x users to port existing applications to the ADSP-219x with minimal modifications. The assembler has a couple of switches (*-legacy -c*), which allows the assembler to recognize ADSP-218x specific syntax. The use of these switches is illustrated with the help of an example. Consider an example ADSP-218x program shown in Table 1. By making the changes highlighted in the boxes, one can assemble and link the source code program to work on the ADSP-219x.

Table 2 shows a sample LDF file specific to the ADSP-2192. Note that the linker place default code sections into the section titled "program", while default data buffers are placed in the section titled "data".

Table 3 shows the same program written entirely using the "new" ELF assembler and linker syntax. This programming style is recommended for any new applications.

If a segment name is not specified for a DM data buffer, the linker places that buffer in the default LDF section called "data"

If a segment name is not specified for a module, the linker places the module in the default LDF section called "program"

Remember that on the 219x, you don't have to (and shouldn't) use /CIRC directive. Use base registers instead.

It is recommended that variable declarations placed at an absolute address be rewritten to instead reside within a segment that starts at that address. i.e., change it to something like..

Default PM data buffers are stored in the "program" section.

Add a couple of lines here to initialize base registers:
AX0 = I7;
REG(B7) = AX0;

Can be optimized by using new ADSP-219x pre-modify DAG addressing

Replace instances of MF with dual accumulator SR

```

MODULE/RAM main2181;
.VAR/RAM/DM/CIRC buffer1[0x100];
.INIT buffer1: <text1.txt>;
.VAR/RAM word1;
.VAR/RAM/DM/SEG=int dm2 buffer2 = 0xabcd;
.VAR/RAM/DM/ABS=0x9000 buffer3 = 0xdead;
/* .VAR/RAM/DM/SEG=seg_buffer3 buffer3 = 0xdead; */
.VAR/RAM/PM pm_buffer1;
.INIT24 pm_buffer1: 0x123456;

JUMP start; RTI;RTI;RTI; /* interrupt vector table - Processor specific */
RTI;RTI;RTI;RTI; RTI;RTI;RTI;RTI;
RTI;RTI;RTI;RTI;

start: /* start of main code */

#ifdef AR_SET_TO_2
#define N 100
AR = 0x0001;
#endif

#ifdef AR_SET_TO_2
#define N 10
AR = 0x0002;
#endif

I7 = ^buffer1;
L7 = %buffer1;
M7 = 1;

M5 = 6;

I2 = ^buffer2;
L2 = 0;

I1 = ^buffer3;
L1 = 0;

AX0 = DM(I7,M7);
MODIFY(I7,M5);
MY1 = DM(I7,M7);
MR = 0;
MF = AR*MY1 (RND), MX1=PM(I7,M7); {MF = x2}
MR=MR+MX1*MF (SS);

CNTR = N;
DO this_loop UNTIL CE;
this_loop: AR = AR + AY0, AY0 = PM(I7,M7);

.ENDMOD;

```

Table 1. Original ADSP-218x example program.

```

ARCHITECTURE(ADSP-2192-12)

SEARCH_DIR( $ADI_DSP\219x\lib )

// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
$OBJECTS = $COMMAND_LINE_OBJECTS ;

// This LDF file reflects the memory map of Core 0 of the ADSP-2192 DSP. It will need to be modified
// to reflect other DSPs

MEMORY
{
    seg_itab { TYPE(PM RAM) START(0x010000) END(0x01003f) WIDTH(24) }
    seg_code { TYPE(PM RAM) START(0x010040) END(0x01007f) WIDTH(24) }
    seg_data1 { TYPE(DM RAM) START(0x000000) END(0x00003f) WIDTH(16) }
    seg_data2 { TYPE(DM RAM) START(0x004000) END(0x00403f) WIDTH(16) }
    seg_my_own { TYPE(DM RAM) START(0x009000) END(0x00903f) WIDTH(16) }
    seg_data3 { TYPE(DM RAM) START(0x00c000) END(0x00c03f) WIDTH(16) }
}

PROCESSOR p0
{
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST )
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
        input_sec1
        {
            INPUT_SECTIONS( $OBJECTS(seg_rth) )
        } > seg_itab
        input_sec2
        {
            INPUT_SECTIONS( $OBJECTS(program) )
        } > seg_code
        input_sec3
        {
            INPUT_SECTIONS( $OBJECTS(seg_1) )
        } > seg_code
        input_sec4
        {
            INPUT_SECTIONS( $OBJECTS(data1) )
        } > seg_data1
        input_sec5
        {
            INPUT_SECTIONS( $OBJECTS(int_dm2) )
        } > seg_data2
        input_sec6
        {
            INPUT_SECTIONS( $OBJECTS(seg_buffer3) )
        } > seg_my_own
        input_sec7
        {
            INPUT_SECTIONS( $OBJECTS(seg_data3) )
        } > seg_data3
    }
}

```

Default "PM" section

Default "DM" section

My own section for "buffer3".
I force it to go into the memory segment titled "seg_my_own"

Table 2. An ADSP-219x LDF file that reflects the ADSP-2192 memory map

```

.SECTION/DATA int_dm1;
.VAR  buffer1[0x100] = "text1.txt";

.SECTION/DATA dummy;
.VAR  buffer2[0x100];

.SECTION/DATA seg_buffer3;
.VAR  buffer3 = 0xdead;

.SECTION/PM seg_1;
.VAR/INIT24  pm_buffer1 = 0x123456;

.SECTION/CODE seg_rth;
JUMP start; RTI;RTI;RTI;      /* begin execution */
RTI;RTI;RTI;RTI;

.SECTION/CODE program;
start:

#ifdef AR_SET_TO_2
#define N 100
AR = 0x0001;
#endif

#ifdef AR_SET_TO_2
#define N 10
AR = 0x0002;
#endif

I7 = buffer1;
L7 = length(buffer1);
M7 = 1;

I2 = length(buffer2);
L2 = 0;

I1 = length(buffer3);
L1 = 0;

AX0 = DM(I7,M7);

MY1 = DM(6,I7);
MR = 0;
SR =AR*MY1 (RND), MX1=PM(I7,M7);
SR=MR+MX1*SR1 (SS);

CNTR = N;
DO this_loop UNTIL CE;
this_loop: AR = AR + AY0, AY0 = PM(I7,M7);

```

Table 3. Program from Table 1 written entirely using new ADSP-219x syntax (recommended approach for any new code)