

# 6 SIMD Processing

## Overview

The ADSP-2106x DSP core is a Single Instruction, Single Data (SISD) processor engine. By comparison, the ADSP-21160 core supports a Single Instruction, Multiple Data (SIMD) multiprocessing paradigm. SIMD support is mode selectable via the PEYEN bit in the MODE1 system control register. When this bit is cleared (0), the ADSP-21160 will execute an instruction sequence much as the ADSP-2106x would execute the same sequence. When the PEYEN bit is set (1), the ADSP-21160 will:

1. dispatch a single instruction to two identical sets of computation units,
2. load two sets of data from memory to support the dual execution,
3. maintain a second register file to manage operands and results for the second execution,
4. store data results from the dual executions to memory.

The ADSP-21160 in SIMD mode can take advantage of the instruction level parallelism inherent in many codes and produce an up to 2X performance improvement over an equivalently clocked ADSP-2106x DSP.

## SIMD Mode and Instruction Support

The ADSP-21160 provides SIMD support features that provide the programmer with a means of realizing the performance increase potential that is provided by a second set of compute units.

Some of these additional SIMD support features include:

- Full second register file and alternate register set to support the second compute unit set,
- Useful SIMD extensions of conditional instruction execution,
- Double register (64-bit) loads and stores,
- SIMD extensions to data move operations (loads, stores, and register transfers),

Figure 6-1 displays a block diagram of the ADSP-21160 core and memory elements.

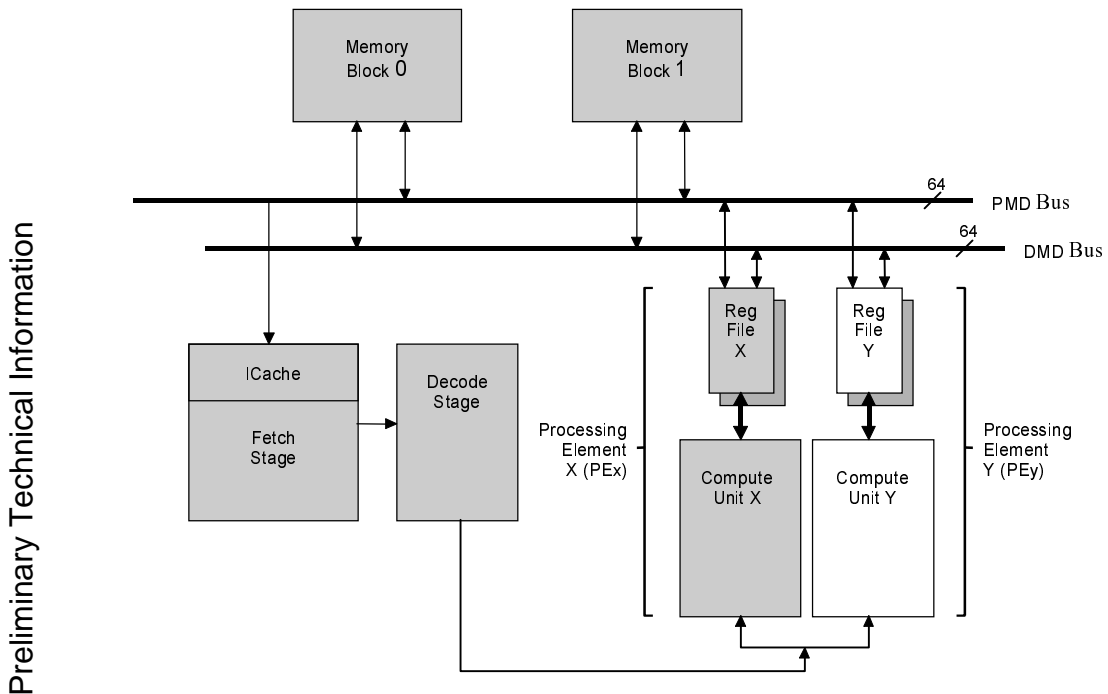


Figure 6-1. Block Diagram Showing Dual Processing Elements

## Instruction Set SIMD Extensions

When the ADSP-21160 operates in SIMD mode, instruction definitions are extended to support a SIMD processing paradigm. This extended definition applies to most types of instructions, and allows for an implied execution of an operation on the contents of a secondary register file in parallel with the execution of the same operation type on the primary regis-

ter file. Thus, a single Type 1 instruction in SIMD mode describes up to 6 computational operations and up to 4 load/store operations involving up to 12 register operands and up to 6 register results (6 operands and 3 results in the PEx register file and 6 operands and 3 results in the PEy register file).

The types of instructions whose definitions are extended as indicated above include any instruction that does either a compute operation or a memory/register transfer operation. These include types 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 14, 15, 16, and 17.

Program sequencing operations, and system register or DAG register manipulation operations are also extended to support SIMD operation. While in SIMD mode, these operations execute as if only one processing element is active. However, the support of SIMD conditional instruction execution and the extended data move operations result in several changes to the program sequencing and DAG register manipulation operations. This chapter defines the modifications to these operations, when ADSP-21160 is operating in the SIMD mode.

## Instruction Type Summary

Appendix A summarizes the more obvious SIMD mode extensions to the ADSP-21160's instruction set. Details of these, and the more subtle features are provided in the remainder of this chapter.

## SIMD Support Features

Figure 6-1 provides a block diagram of the ADSP-21160 core and memory. The execution stage of the ADSP-21160 consists of two, processing elements (PE's). While Processing Element X (PE<sub>x</sub>) and Processing Element Y (PE<sub>y</sub>) operate similarly, PE<sub>x</sub> is always enabled, supporting the ADSP-2106x SISD (Single Instruction, Single Data) compute model. When the PEYEN bit in the Mode1 system control register is set, both Processing Elements are enabled; duplicate instruction information is dispatched to both PE<sub>x</sub> and PE<sub>y</sub>. The two PE's operate independently, and the only interaction or passing of state between the two PE's occurs as the result of explicit data transfer or data swap instructions.

All data transfer operations use either the PM bus or DM bus. No additional data pathway between the two PE's is required.

The two PE's are fairly symmetrical, and each PE contains the following unique and in most cases identical components:

- ALU
- Multiplier and multiplier primary and secondary result registers
- Shifter
- Load and Store data path logic (not the DAG's, these are shared resources).
- 16 entry by 40-bit primary register file
- 16 entry by 40-bit secondary register file

- ALU, Multiplier, and Shifter status registers and condition compare logic

The DAG resources are not duplicated across the PE's. This results in some restrictions on the location of load and store data. A given DAG may be either shared by the two PE's, or one DAG may be temporarily assigned to each PE. The sharing or unique assignment of DAG's to PE's can change on an instruction by instruction basis.



The following sections describe some of the PE components in more detail.

## Dual Compute Units Sets

The computation units (ALU, Multiplier, and Shifter) in the two processing elements are identical in all respects. The register load and store data path logic differs slightly between the two PE's. These differences allow for asymmetric data moves to, from, and between the two PE's.



## Dual Register Files

The two 16 entry data register files (one in each PE) and their operand and result busing and porting are identical. The same is true for each 16 entry alternate register files. As mentioned earlier, the load and store data path logic does differ between the two PE's, and thus the interface between the register files and the PMD and DMD buses differ. The decision of transfer direction, source and destination, which bus to transfer data on, and which short-word lanes of that bus to employ depends on the following conditions:

- the state of the PEYEN and BDCSTx bits in the MODE1 system control register (note that broadcast loads require that DAG index registers I1 or I9 must be used)
- the state of the IMDW1-0 bits in the SYSCON IOP register
- the type of access specified by memory map alias range (long word, normal word, short word)
- the outcome of the instruction condition comparison
- whether the register file belongs to PEx or PEy
- UREG field bits 6-4 (PEx reg file, PEy reg file )
- LW field bit (long word override)
- Type 5a or Type 5b register to register transfer instructions
- and of course the state of the various instruction fields that specify DAG1/DM or DAG2/PM

The load/store ports into the register files are 80-bits wide. This is to accommodate double register data accesses. There are four such ports for each register file, two load ports and two store ports.

Figure 6-2 displays a block diagram of the register file load/store data path interface to the PMD and DMD buses.



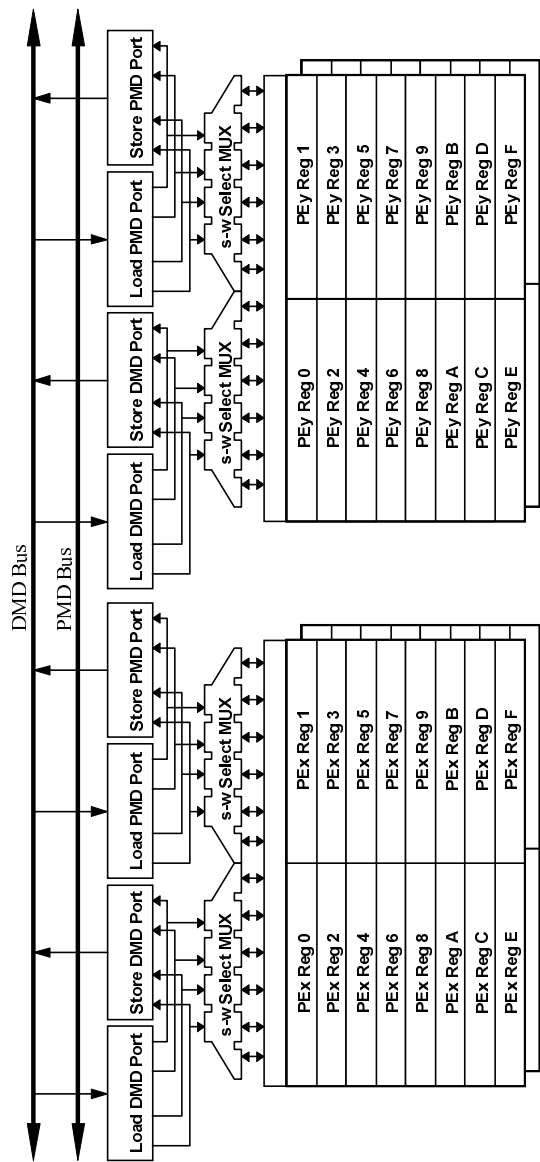


Figure 6-2. Register File Load/Store Multiplexing



## Alternate Register Set Selection

Both register files consist of a primary set of 16 by 40-bit registers, and an alternate set of 16 by 40-bit registers. Context switching between the two sets of registers occur in parallel between the two PE's. In other words, the SRRFL bit in the MODE1 system control register selects either the primary or secondary set of R0-R7 registers in both PEx and PEy. Likewise, the SRRFH bit in the MODE1 system control register selects either the primary or secondary set of R8-R15 registers in both PEx and PEy.

## Mode Level SIMD Support

This section describes MODE1 system control register state that is dedicated to the support of SIMD operation in the ADSP-21160. [Table 6-1](#) summarizes the definition of the new MODE1 control bits.

Table 6-1. MODE1 System Register SIMD Control State

Bit	Name	Definition
20	****	Reserved
21	PEYEN	1=Enable Processing Element Y, 0=Disable Processing Element Y
22	BDCST9	Enable I9 broadcast of data loads to a register in both processing elements (DAG2)
23	BDCST1	Enable I1 broadcast of data loads to a register in both processing elements (DAG1)


### Processing Element X Enable

Processing Element X (computation units and register files) is always enabled for accepting instruction dispatches.

### Processing Element Y Enable

When the PEYEN bit is set, Processing Element Y (computation units and register files) is enabled for accepting instruction dispatches. When the PEYEN bit is cleared, Processing Element Y is disabled and goes into a low power mode. There is a one cycle delay after PEYEN is set or cleared, before the change to or from SIMD mode takes effect.

When the PEYEN bit is cleared, the ADSP-21160 executes instructions in Single Instruction, Single Data (SISD) mode. When the PEYEN bit is set, the ADSP-21160 executes instructions in SIMD mode

The PEYEN bit  cleared following chip reset.

### Register Load Broadcasting

When the BDCST9 bit is set, data register loads from the PMD bus that use the I9 DAG2 index register are “broadcast” to a register or register pair in each PE. For example, if the register load to R0 is requested, then both R0 in PEx and S0 in PEy will receive the same data value.

When the BDCST1 bit is set, data register loads from the DMD bus that use the I1 DAG1 index register are “broadcast” to a register or register pair in each PE.

Enabling either DAG1 or DAG2 register load broadcasting has no affect on register stores, or loads to system registers other than the register file data registers. Broadcasting cannot be used for memory writes. Both bits are cleared following chip reset.

[Table 6-2](#) summarizes the affects of a register load operation on both PE's with register load broadcasting enabled. Note that the PEYEN bit (SISD/SIMD mode select) does not influence broadcast operations.

Table 6-2. Register Load Dual PE Broadcast Operation

Instruction	PE <sub>x</sub> operation	PE <sub>y</sub> operation
R <sub>x</sub> = DM(I1,Ma) PM(I9,Mb) DM(I1,Ma), PM(I9,Mb) ;	R <sub>x</sub> = DM(I1,Ma) PM(I9,Mb) DM(I1,Ma), PM(I9,Mb) ;	S <sub>x</sub> = DM(I1,Ma) PM(I9,Mb) DM(I1,Ma), PM(I9,Mb) ;

### Mode Mask System Register (MMASK)

This register contains one mask bit for each bit in the MODE1 register. The mask is applied to the MODE1 register under the following conditions:

- each time a Type 20, PUSH STS instruction is executed
- each time an external interrupt (IRQ), timer interrupt, or VIRPT vector interrupt cause an automatic push of status onto the status stack.

A bit that is set in the MMASK register results in the clearing of the corresponding bit in the MODE1 register immediately

following the push of status. In other words, the original MODE1 register contents are first pushed onto the 15 entry status stack, and then the MODE1 register contents are modified based on the state of the MMASK register.

The MMASK register is initialized to 0x0020 0000 (PEYEN bit masked forcing SISD operation on PEx) following chip reset.

Interrupts that must execute in SISD mode can have the state of the PEYEN bit saved and the PEYEN bit cleared (SISD operation on PEx enabled) either automatically, or with one instruction in the ISR preamble by using this feature.

## Instruction Level SIMD Support

This section describes those SIMD features that are accessible at the ADSP-21160 instruction level. These features are affected by the state of the PEYEN processing element enable.

Each processing element has a different view of the universal register space encodings. Processing element “X” decodes the universal space as displayed in [Table 6-7](#). Processing element “Y” decodes the universal space as displayed in [Table 6-8](#).

These two encodings for the universal register space create a set of complementary register pairs. When an instruction is fetched, decoded, and dispatched to either (or both) processing element “X” and/or processing element “Y”, PEx will process the instruction using the [Table 6-7](#). PEy will process the instruction using the [Table 6-8](#). The complementary register pairs appear in [Table 6.3](#) (those universal registers that don’t appear [Table 6.3](#) have the same identities in both PEx and



PEy). If one PE refers to a register in either column, the other PE will refer the complimentary register in the opposite column.

Table 6.3. SIMD Mode Complementary Register Pairs

R0	S0
R1	S1
R2	S2
R3	S3
R4	S4
R5	S5
R6	S6
R7	S7
R8	S8
R9	S9
R10	S10
R11	S11
R12	S12
R13	S13
R14	S14
R15	S15

Table 6.3. SIMD Mode Complementary Register Pairs (Cont'd)

USTAT1	USTAT2
USTAT3	USTAT4
ASTATx	ASTATy
STKYx	STKYy
PX1	PX2

Data Move Operations

The ADSP-21160 supports a number of data move operations that provide flexible data load, store, and both unidirectional and bidirectional register to register transfers, swaps, and rotates from, to, and between both DSP core processing elements.

The register files of both processing elements are accessible via the universal register space opcode field, thus allowing loads, stores, and register to register transfers, swaps, and rotates from/to these registers via Type 1, 3, 4, 5, 6, 10, 14, 15, and 17 instructions.

Table 6-7 and Table 6-8 display the UREG field (and DREG sub-field) encodings (as used by PEx and PEy respectively) that provide access to the Rx and Sx registers. The Rx encodings access the PEx data register files (both primary and alternate). The Sx encodings access the PEy data register files (both primary and alternate).



The following instruction level, dual PE, data move operations are supported.

**SIMD Mode Data Load & Store Summary.** The Memory chapter of this specifications provides details of the data addressing options provided for the support of data move operations in SIMD mode. This section provides a brief summary of this support.

Single-data, normal precision, normal word addressed accesses in SIMD mode will access two consecutive 32-bit values in memory. The explicitly addressed normal word will be associated with PEx, the alternate normal word will be associated with PEy. Accesses to normal word space access double 32-bit words on even 32-bit boundaries. The least significant address bit will select which 32-bit value goes to each execution complex. An even address will fetch the even normal word to PEx and the odd normal word to PEy. An odd address will fetch the odd normal word to PEx and the even normal word to PEy. Note that the address sequence to access consecutive data in SIMD mode would then proceed as in the following sequence (with an assumed base address in the normal word space)... 0x00, 0x02, 0x04, 0x06, ....

Dual-data, normal precision, normal word addressed accesses in SIMD mode will access two sets of two consecutive normal word values, each from a different block of memory. Data is accessed on even normal word boundaries and even and odd words are associated with PE's as in the single-data access case, but in this case, each PE loads or stores two normal word values.

Extended precision, normal word addressed accesses in SIMD mode are always dual-data accesses. Two extended precision, normal word values are accessed, each from a different block of memory. While in SIMD mode, all DM bus accesses will load or store to PEx, and all PM bus accesses will load or store to PEy. While in SISR mode, two extended precision, normal word values are accessed, each from a different block of memory as in SIMD mode, however, both values load or store to registers in the PEx register file.

Single data, short word addressed accesses in SIMD mode will access two consecutive 32-bit values from memory. The least significant address bit directs either the even or odd 16-bit value of each 32-bit value to the appropriate execution complex (selected by bit <1> of the address). Interleaving of data to take advantage of SIMD in this case gets a little more complicated. Data should still be interleaved on 32-bit boundaries. In other words, if the first two 16-bit locations contain data destined for PEx, then the second two 16-bit locations contain data destined for PEy, etc. The address sequence to access consecutive data in SIMD mode would then proceed in the following sequence (with an assumed base address in the short word space)... 0x00, 0x01, 0x04, 0x05, 0x08, 0x09, 0x0C, 0x0D, ....

Dual-data, short word addressed accesses in SIMD mode will access two sets of two consecutive 32-bit values from memory. Data is accessed on even normal word boundaries and the least significant two address bits of each address select the short word component and the PE association as in the single data case, but in this case, each PE loads or stores two short word values.



Long word addressed accesses in SIMD mode will always be dual-data accesses. Two sets of two consecutive 32-bit words are accessed on even 32-bit boundaries. Each set of double words is accessed from a different memory block. Each full 64-bit value will be loaded to (or stored from) two consecutive register file registers in each PE. While in SIMD mode, all DM bus accesses will load or store to PEx, and all PM bus accesses will load or store to PEy. The data access behavior is similar for SISD mode dual-data, long word addressed accesses except that both full 64-bit values will be loaded to or stored from two register file data register pairs in PEx.

External memory accesses and MP memory space accesses support a broad set of packing and unpacking options that allow efficient use of the 64-bit data paths.

**Register to Register Transfers.** Unidirectional and Bidirectional register to register transfers are accomplished via the Type 5a instruction. All four combinations of inter-PE and intra-PE transfers ( $\text{PE}_x \leftrightarrow \text{PE}_x$ ,  $\text{PE}_x \leftrightarrow \text{PE}_y$ ,  $\text{PE}_y \leftrightarrow \text{PE}_x$ , and  $\text{PE}_y \leftrightarrow \text{PE}_y$ ) are possible in both SISD (unidirectional) and SIMD (bidirectional) modes.

While the ADSP-21160 is in SISD execution mode (the PEYEN bit is cleared in the MODE1 system control register), Type 5a register to register transfers are unidirectional. In other words, the operation is performed on one PE and not duplicated by the other PE. The transfer consists of a single source register, and a single destination register with either register existing in either PE data register file. [Table 6-4](#) illustrates

the affect of Type 5a unidirectional transfers on the two PE's while SIMD mode is disabled.

Table 6-4. SISD Mode Type 5a Unidirectional Register Transfer

Instruction	Explicit (Pex) data transfer	Implied (Pey) data transfer
IF condition compute, Rx = Ry;	Rx <= Ry	na
IF condition compute, Rx = Sy;	Rx <= Sy	na
IF condition compute, Sx = Ry;	Sx <= Ry	na
IF condition compute, Sx = Sy;	Sx <= Sy	na

While the ADSP-21160 is in SIMD execution mode (the PEYEN bit in the MODE1 system control register is set), Type 5a register to register transfers are bidirectional. In other words, the operation is performed by both PE's in parallel. The instruction transfers two source registers, one from each of the PE data register files, and two destination registers, one in each of the PE data register files. Table 6-5 illustrates the affect of Type 5a bidirectional transfers (SIMD mode enabled) on the two PE's. Note that the first and last instructions are synonymous and achieve the same results, and the middle two instructions are synonymous and achieve the same results (this

only applies to the unconditional or always-true conditional form of this instruction).

Table 6-5. SIMD Mode Type 5a Bidirectional Register Transfers

Instruction	Explicit data transfer	Implied data transfer
IF condition compute, Rx = Ry;	Rx <= Ry	Sx <= Sy
IF condition compute, Rx = Sy;	Rx <= Sy	Sx <= Ry
IF condition compute, Sx = Ry;	Sx <= Ry	Rx <= Sy
IF condition compute, Sx = Sy;	Sx <= Sy	Rx <= Ry

The conditional form of the Type 5a instruction will cause a bidirectional (SIMD mode) register to register transfer to conditionally become a unidirectional transfer, or a NOP (if the condition fails on both PE's).

Instruction Type 5a, SIMD mode (bidirectional) register to register transfers between a register file data register and one of the DAG, control, or status registers is allowed. When the DAG, control, or status register is a source of the transfer, the destination can be a register file data register. This results in the contents of the single source register being duplicated in a data register in both PE's.

Use care in the case where the DAG, control, or status register is a destination of a transfer from a register file data register source. If the instruction executes on both PE's, the target register will be updated with the source explicitly referenced in the instruction; the implied transfer will not occur. If this is not

the desired behavior, make sure that a conditional operation is used to select either one PE, or neither PE as the source.

In the case where a DAG, control, or status register is both source and destination, the data move operation will execute the same as it would if SIMD mode were disabled.

**Register Swaps and Rotates.** The Type 5b instruction provides for bidirectional register to register swaps and rotates. Only the 4-bit DREG data register sub-field is used to define register sources and destinations for this instruction. This instruction is dependent of the state of the PEYEN bit in the MODE1 system register. The swap or rotate always occurs between one register in each PE data register file.

A register to register rotate is characterized by a register of the same relative location in each processing element data register file exchanging values. To cause an inter-PE, register to register rotate, the source register and destination register are specified by the same DREG value in both instruction Type 5b source and destination fields.

A register to register swap is characterized by a register of different relative location in each processing element data register file exchanging values. To cause an inter-PE, register to register swap, the source register and destination register are specified by different DREG values in the instruction Type 5b source and destination fields.

Only single, 40-bit register to register swaps and rotates are supported (no double register operations). Since both DMD and PMD are fully utilized to support a single register swap,



double register swaps can't be supported (limited bus and register file port resources).

Table 6-6 illustrates the affect of Type 5b register swaps and rotates on the two PE's. Note the difference between this register swap/rotate operation and the bidirectional register to register transfer described in the previous section. The unconditional register swap operation is asymmetric across the PE's and always involves just two registers , both on different PE's. The unconditional swap is, therefore, always destructive where the transfer is not.

Table 6-6. Register to Register Swaps and Rotates

Type	Instruction	Explicit PEx xfer	Implied Pey xfer
Rotate	IF condition compute, Rx <-> Sx;	Rx <= Sx	Sx <= Rx
Swap	IF condition compute, Rx <-> Sy;	Rx <= Sy	Sy <= Rx

The conditional form of the Type 5b instruction conditionally causes the rotate and swap to become a unidirectional register to register transfer, or a NOP (if the condition fails on both PEs). This case will only occur if SIMD mode is enabled.

If SIMD mode is disabled, both PE data register files will either participate in the rotate/swap or not based on the outcome of the condition test. If SIMD mode is enabled, both PE's register files participate if both condition tests are true. The swap/rotate degrades to a unidirectional transfer (as defined in Table 6-6) if one of the condition tests is true.

Table 6-7. Universal Register Codes (as seen by PEx)

bits 3 2 1 0	bits 6 5 4							
	000	001	010	011	100	101	110	111
0000	R0	I0	M0	L0	B0	S0	FADDR	USTAT1
0001	R1	I1	M1	L1	B1	S1	DADDR	USTAT2
0010	R2	I2	M2	L2	B2	S2		MODE1
0011	R3	I3	M3	L3	B3	S3	PC	MMASK
0100	R4	I4	M4	L4	B4	S4	PCSTK	MODE2
0101	R5	I5	M5	L5	B5	S5	PCSTKP	FLAGS
0110	R6	I6	M6	L6	B6	S6	LADDR	ASTATX
0111	R7	I7	M7	L7	B7	S7	CURLCNTR	ASTATY
1000	R8	I8	M8	L8	B8	S8	LCNTR	STKYX
1001	R9	I9	M9	L9	B9	S9		STKYY
1010	R10	I10	M10	L10	B10	S10		IRPTL
1011	R11	I11	M11	L11	B11	S11	PX	IMASK
1100	R12	I12	M12	L12	B12	S12	PX1	IMASKP
1101	R13	I13	M13	L13	B13	S13	PX2	LIRPTL
1110	R14	I14	M14	L14	B14	S14	TPERIOD	USTAT3
1111	R15	I15	M15	L15	B15	S15	TCOUNT	USTAT4

Table 6-8. Universal Register Codes (as seen by PEy)

bits 3 2 1 0	bits 6 5 4							
	000	001	010	011	100	101	110	111
0000	S0	I0	M0	L0	B0	R0	FADDR	USTAT2
0001	S1	I1	M1	L1	B1	R1	DADDR	USTAT1
0010	S2	I2	M2	L2	B2	R2		MODE1
0011	S3	I3	M3	L3	B3	R3	PC	MMASK
0100	S4	I4	M4	L4	B4	R4	PCSTK	MODE2
0101	S5	I5	M5	L5	B5	R5	PCSTKP	FLAGS
0110	S6	I6	M6	L6	B6	R6	LADDR	ASTATY
0111	S7	I7	M7	L7	B7	R7	CURLCNTR	ASTATX
1000	S8	I8	M8	L8	B8	R8	LCNTR	STKYY
1001	S9	I9	M9	L9	B9	R9		STKYX
1010	S10	I10	M10	L10	B10	R10		IRPTL
1011	S11	I11	M11	L11	B11	R11	PX	IMASK
1100	S12	I12	M12	L12	B12	R12	PX2	IMASKP
1101	S13	I13	M13	L13	B13	R13	PX1	LIRPTL
1110	S14	I14	M14	L14	B14	R14	TPERIOD	USTAT4
1111	S15	I15	M15	L15	B15	R15	TCOUNT	USTAT3

**Forcing Long Word accesses to/from Normal Word Address Space.** Instruction types 3, 14, and 15 provide a long word override bit field that forces 32-bit normal word addressed accesses and short word addressed accesses to load or store 64-bit values instead.



When a Type 3, 14, or 15 instruction specifying a long word override is executed, load and store accesses across the DM bus are forced to be long word (dual register) accesses. Data accesses occur as if long word address space were being targeted for the access. All system registers accessible from the universal register space are accessible by this mode.

The LW bit has effect only when normal word address space is accessed, and the IMDWx bit for the block accessed is cleared.

### Conditional Instructions

The ADSP-21160 provides some features that enhance the execution of conditional instructions while in SIMD mode. There are six basic types of conditional functions that need special SIMD support:

- conditional compute operations
- conditional branches (Jumps, Calls, Returns)
- conditional loops (DO - UNTIL)
- conditional data moves
- conditional DAG operations
- floating point exceptions





Only the compute resources (PE<sub>x</sub> and PE<sub>y</sub>) of the ADSP-2106x are duplicated to support SIMD. The other programming model resources (e.g., program sequencer, DAGs, etc.) are not duplicated. Many instructions use one (or more) shared resource(s); however, these instruction's conditional execution can be a function of the state of both independent PEs. The following table summarizes how each conditional function type is resolved when SIMD mode is enabled. The following sections describe how each conditional function is handled in detail.

Table 6-9. Conditional Execution Summary

Function	Resolution
Conditional Compute Ops	The local "scope" of each PE determines conditional execution on that PE.
Conditional Branches	Logical "AND" of the state of the two PEs.
Conditional Loops	Logical "AND" of the state of the two PEs.
Conditional Data Moves	The transfer for each PE is determined independently by the state of the respective PE.
Conditional DAG Ops	Logical "OR" of the state of the two PEs.
Floating Point Exceptions	Logical "OR" of the state of the two PEs.

**Status Flags.** To provide for behavior of conditional operations that is appropriate for either PE independently or both PE's together, an independent set of computation status is maintained for each PE. PEx maintains an Arithmetic Status system register (ASTAT<sub>x</sub>) and a Sticky Status system register

(STKYx). PEy maintains its own independent set of similar state in the complementary registers ASTATy and STKYy.

Status-pushing interrupts, status-popping interrupt returns, PUSH STS instructions, and POP STS instructions all affect both ASTATx and ASTATy registers in parallel (ie. both registers are pushed onto the hardware status stack and popped from the stack at the same time).

All four status registers (ASTATx, ASTATy, STKYx, STKYy) are accessible via system register accesses using Type 3, 5, 14, 15, 17, and 18 instructions. [Table 6-7](#) and [Table 6-8](#) display the universal register encodings (as decoded by PEx and PEy respectively) that allow for accessing these registers. Note that the registers are grouped so that similar registers may be efficiently accessed via double register accesses (using long word addressed accesses, accesses to any of the memory mapped spaces using a Type 3, 14, or 15 instruction with the LW bit field (long word override) bit set).



**Conditional Compute Operations.** While in SIMD mode, a conditional compute operation may execute on both PE's, either one PE, or neither PE dependent on the outcome of the status flag test. Flag testing is independently performed on each PE.

On SIMD machines, this independent instruction execution on a per PE basis can be used to introduce asymmetric control flow patterns in parallel threads of execution (without actually performing a branch operation).

ASTATx/y status flags are dynamic and will update as the result of subsequent computations. This dynamic behavior

makes it difficult to execute a lengthy sequence of conditional computations based on the outcome of a single initial computation (subsequent computations result in the loss of the initial test status). Note that if a conditional computation results in a NULL-operation on a PE due to a false condition test outcome, the status state associated with that PE (contents of the appropriate ASTAT<sub>x/y</sub> and STKY<sub>x/y</sub> status registers) will not change.

The ASTAT<sub>x/y</sub> Bit Test Flag (BTF) bit can be conditionally set based on the state of the other ASTAT<sub>x/y</sub> and/or STKY<sub>x/y</sub> status flags using the BIT TST or BIT XOR forms or the Type 18, System Register Bit Manipulation instruction. Note that a BIT TST ASTAT<sub>x</sub> instruction will result in the testing of the appropriate bit(s) in the ASTAT<sub>x</sub> register by PEx, and the testing of the same relative bit(s) in the complementary ASTAT<sub>y</sub> register by PEy. This process can be used as a means of capturing status that is to be used to qualify a sequence of conditional computations. The IF TF compute instruction can then be used to predicate the conditional sequence.

**Conditional Branches.** Data dependent branching is usually inconsistent with SIMD processing. The ADSP-21160 will take a branch as the result of the logical AND'ing of the condition tests of both PE's. In other words, a conditional branch will be TAKEN only if the specified condition tests true for both PEx and PEy. Many DSP SIMD algorithms benefit most from this control construct, often coupled with the conditional execution of the compute instructions (e.g., terminating a loop upon reaching a certain error threshold.)

Note that if the condition test during a conditional branch is true for only one of the PE's, then the computation sequence

following the branch probably needs to be a series of conditional compute operations. This allows NULL-op'ing of the PE that possesses the true conditional (only the false condition thread will execute).

For many algorithms, OR'ing of the SIMD conditional tests is required for the conditional branch. OR'ing the condition tests permits easy construction of “early out” SIMD algorithms (e.g., searches). The ADSP-2106x architecture provides several mechanisms to achieve equivalent OR'ing of the condition tests. In many cases, testing for the negative of the test condition is provided, allowing the AND to OR mapping to be achieved by transforming the test using DeMorgan's laws. Also, since the test flags are accessible (and writable) through the System Register map, in complimentary register pairs, if state changes on one PE, it is relatively straightforward to reflect that change in the other PE.



As an example, the following code synthesizes the logical OR of the two PE states control structure when testing for equality (AN flag set). The loop will terminate upon the 1st detection of equality in either PEX or PEY.

```

    bit set model PEYEN;    // enable SIMD mode
    ...
loop:  ...
    compu (r10, r12);
    // sets ANx status flag if r10=r12, OR if s10=s12
    if EQ astaty = astatx;
    // will set AN flag in both ASTATs if either is set,
    // due to conditional swap
    ???;
    // any instruction not dependent on ASTAT can be
    // scheduled in this slot
    If NE jump loop;
    // branch on logical "OR" has been synthesized...

```

...

Note that the conditional swap instruction (if EQ astaty = astatx) swaps only from the ASTAT register(s) in which the AN flag is set. If the comparison fails for both PEs, the swap degrades to a NOP.

In this example, the performance penalty for coding for the logical OR control flow is only one clock cycle, provided a useful instruction can be scheduled after the ASTAT write. Worst case would be two clock cycles if a NOP had to be inserted after the ASTAT write, due to the “effect” latency when writing ASTAT. Any instruction which is not dependent on ASTAT is a candidate for that slot.

Type 9, 10, and 11 instructions are comprised of both a conditional branch operation and a conditional compute operation. The branch ops associated with these instructions will be taken as the logic OR’ing of the condition tests of both PE’s; the compute ops will conditionally execute as the result of the independent outcome of the condition tests of each PE’s status (as indicated in the preceding section).

**Conditional Loops.** Similar to conditional branches, a conditional DO - UNTIL loop will terminate as the result of the logical AND’ing of the condition tests of both PE’s.

As with conditional branch sequences, the contents of a conditional loop will probably consist of conditional computes. This allows NOP’ing of the PE that wishes to terminate the loop first.

**Conditional Data Moves (Register Loads, Stores, and Transfers).** The data move portions of conditional instruc-

tions are also affected by the outcome of the condition test. When the ADSP-21160 is operating in SIMD mode, the completion of that part of a data move that is associated with a PE is dependent on the condition test of the status associated with that PE. Only that portion of a data move that is associated with a PE that has a true condition test will complete.

For register store operations (memory writes), the ADSP-21160 will drive data associated with both PE's on the PMD and/or DMD buses and addresses and write control on the PMA and/or DMA buses. When the condition tests for each PE complete (later in the access) the write data associated with a condition test that fails will be aborted prior to being strobed into SRAM.

For register load operations (memory reads), the ADSP-21160 will drive read address and control on the PMA and/or DMA buses and read data associated with both PE's will be driven from memory onto the PMD and/or DMD buses. When the condition tests for each PE complete (later in the access) the data strobe (register write gate) associated with a condition test that fails will be withheld.

Unidirectional register to register transfer operations (Type 5 instructions using Rx and/or Sx source and destinations) will not complete if the condition test of either, or both PE's is false. For a bidirectional register to register transfer, the data strobe (register write gate) associated with a condition test that fails will be withheld.

**Conditional DAG Operations.** While the ADSP-21160 is in SIMD mode, DAG modify operations, either explicitly via the MODIFY(Ia, Mb) operation or implicitly as part of a post-

modify indirect load or store operation, will occur based on the logical OR'ing of the outcome of the condition tests on both PE's. Since both threads of a SIMD sequence may be dependent on a single DAG index value, either thread needs to be able to cause a modify of the index.

There is one exception to the above rule. Pre-modification of an index will always occur, independent of the outcome of the condition tests on each PE.



**Floating Point Exceptions.** The ADSP-21160 operating in SIMD mode will take an exception based on the logical OR'ing of the outcome of the exception status test on each processing element. When the interrupt associated with one of the four fixed-point or floating-point exceptions is enabled, an exception condition that tests true on either, or both PE's will result in an exception interrupt. It is up to the ISR to explicitly sample the status of both PE's to detect which of the PE(s) encountered the exception.

The return from a floating point interrupt will result in clearing of the STKY state in both PEx and PEy STCKY registers.

## SIMD Coding for the ADSP-21160

### Overview and Guidelines

#### Types of Instruction Parallelism

**Macro Scale.** Macro scale parallelism exists where lengthy or deeply nested instruction blocks can be executed in parallel

due to the absence of true data dependencies between the blocks. In the case of the IIR algorithm presented in section “[Case I: Macro Parallelism on Routine scale \(IIR\)](#)” on page 6-38, the block consists of an assembly routine containing a single large nested loop. In the case of the FFT algorithm fragment presented in section “[Case II: Macro Parallelism on loop scale \(FFT\)](#)” on page 6-42, the block consists of an inner loop or set of nested loops. The larger the cycle count of the parallel block with respect to the over-all application cycle count, the higher the level of performance increase. For example, the large scale block of the IIR routine results in a near 100% performance increase.

Porting code from SISD implementation to macro scale SIMD is usually fairly easy, and requires very little real code change as is evidenced by the examples in sections “[Case I: Macro Parallelism on Routine scale \(IIR\)](#)” on page 6-38 and “[Case II: Macro Parallelism on loop scale \(FFT\)](#)” on page 6-42. Due to the limited data access options provided with this implementation, input and output data organization is critical to realizing performance benefits. Independent input data streams should be interleaved (word granularity) in memory before entering SIMD mode, and output results will be interleaved. Where possible, the IOP should be used to perform this interleaving. Special purpose routines for initializing interleaved data, and to post-process interleaved buffers should be avoided since these may add significantly to SIMD overhead code.

**Micro Scale.** Micro scale parallelism is much harder to take advantage of. It exists where a sequence of instruction-symmetrical (similar operation with different operands) operations within an instruction run have no true data dependencies and can be scheduled in parallel. Typically, the performance



increase from this level of parallelism is lower since there are usually fewer parallel operations to amortize SIMD overhead than in the Macro scale cases. Porting code from SISD operation to micro scale SIMD is usually much more labor intensive as well.

## C Coding Examples

The most straight forward use of the SIMD support provided by the ADSP-21160 by the compiler is via C #pragma directives.

This section provides a couple of C code fragments that provide examples of how pragma directives might be used. Both examples provide scalar code for the fragment, and parallel code for the same fragment using the C pragma directive approach.

### Case I: Simple Dot Product

#### Scalar Code.

```
sum = 0;
for (j=0; j<n; j++)
    sum += a[j] * b[j];
```

#### Parallelized Code.

```
#pragma enter_SIMD
tsum[0] = tsum[1] = 0;
for (j=0; j<n; j+=2)
    tsum[0] = a[j] * b[j];
    #pragma leave_SIMD
sum = tsum[0] + tsum[1];
```

## Case II: IDCT Function

### Scalar Code.

```
/* Copyright (c) 1995 The Regents of the University of
California. All rights reserved. Permission to use, copy,
modify, and distribute this software and its documentation
for any purpose, without fee, and without written
agreement is hereby granted, provided that the above
copyright notice and the following two paragraphs appear
in all copies of this software. IN NO EVENT SHALL THE
UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. THE
UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS"
BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION
TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
MODIFICATIONS. */
```

```
/* Perform IEEE 1180 reference (64-bit floating point,
separable 8x1 direct matrix multiply) Inverse Discrete
Cosine Transform */
```

```
/* idctref.c, Inverse Discrete Fourier Transform, double
precision */
```

```
#include <math.h>
```

```
/* global declarations */
void float_idct (short *block);
```

```
/* cosine transform matrix for 8x1 IDCT */
static double c[8][8];
```

```
/* perform IDCT matrix multiply for 8x8 coefficient block
*/
```

```

void float_idct(block)
short *block;
{
    int i, j, k, v;
    double partial_product;
    double tmp[64];

    for (i=0; i<8; i++)
        for (j=0; j<8; j++)
        {
            partial_product = 0.0;

            for (k=0; k<8; k++)
                partial_product+= c[k][j]*block[8*i+k];

            tmp[8*i+j] = partial_product;
        }

    /* Transpose operation is integrated into address
    mapping by switching loop order of i and j */

    for (j=0; j<8; j++)
        for (i=0; i<8; i++)
        {
            partial_product = 0.0;

            for (k=0; k<8; k++)
                partial_product+= c[k][i]*tmp[8*k+j];

            v = simd_floor(partial_product+0.5);
            block[8*i+j] = (v<-256) ? -256 : ((v>255) ? 255 : v);
        }
}

```

### Parallelized Code.

```

/* Copyright (c) 1995 The Regents of the University of
California. All rights reserved. Permission to use, copy,
modify, and distribute this software and its documentation
for any purpose, without fee, and without written
agreement is hereby granted, provided that the above

```

copyright notice and the following two paragraphs appear in all copies of this software. IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR

\* DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS. \*/

```
/* Perform IEEE 1180 reference (64-bit floating point,
separable 8x1 direct matrix multiply) Inverse Discrete
Cosine Transform (Oh well, so we can't do double precision
floats) */
```

```
/* idctref.c, Inverse Discrete Fourier Transform, double
precision */
```

```
#include <simd_math.h>
```

```
/* global declarations */
void float_idct (short *block);
```

```
/* cosine transform matrix for 8x1 IDCT */
static double c[8][8];
```

```
/* perform IDCT matrix multiply for 8x8 coefficient block
*/
```

```
void float_idct(block)
short *block;
{
    int i, j, k, v[2];
    double partial_product[2];
    double tmp[64];

    for (i=0; i<8; i++)
```

```

        for (j=0; j<8; j++)
        {
#pragma enter_SIMD

            partial_product[0] = 0.0;

            for (k=0; k<8; k+=2) /* double the loop increment */
                partial_product[0] += c[k][j]*block[8*i+k];

#pragma leave_SIMD

            tmp[8*i+j] = partial_product[0] + partial_product[1];
        }

        /* Transpose operation is integrated into address
        mapping by switching
        loop order of i and j */

#pragma enter_SIMD

        for (j=0; j<8; j+=2) /* double the loop increment */
            for (i=0; i<8; i++)
            {
                partial_product[0] = 0.0;

                for (k=0; k<8; k++)
                    partial_product[0] += c[k][i]*tmp[8*k+j]; /* c is
                broadcast to both PEs */

                v[0] = simd_floor(partial_product[0]+0.5); /* NOTE:
                simd math funct! */
                block[8*i+j] = (v[0]<-256) ? -256 : ((v[0]>255) ?
                255 : v[0]); /* predicated execution cntrl flow and MIN(),
                MAX() ops */
            }

#pragma leave_SIMD

    }

```

## Assembly Coding Examples

The following two assembly code examples display the types of modifications that are necessary to take advantage of Macro scale parallelism using this SIMD implementation. Performance results for these cases follow the coding examples. It can be observed that little code manipulation is required to get a significant performance improvement.

### Case I: Macro Parallelism on Routine scale (IIR)

The following code implements a real, biquad IIR filter. Sample data is assumed to have been DMA'd into internal memory, though, for simplicity, it's actually implemented here as a data buffer (inbuf) that has been pre-loaded with either one or two impulse functions. Code for the actual cascaded biquad filter routine is not shown since the code is identical for both the SISD and SIMD implementations. Note that the two sample sets that are operated on in parallel in the SIMD implementation must be independent data streams (i.e. two data channels).

#### SISD code.

```
{ main program for calling real biquad IIR filter
  input, output data stored in memory ... no interrupts
  follows example system described in ADSP-21020 Users
Manual

    "iirmem.asm"          (requires file "iircoefs.dat")
    Analog Devices, Inc.  DSP Applications  P.O.Box 9106
    Norwood, MA 02062
    Christoph D. Cavigioli ... 25-Apr-1991
    Jan-1997 : TEL : Ported code to run on ADSP-21060
}
.EXTERN cascaded_biquad, cascaded_biquad_init;
.GLOBAL coefs, dline;
```

```

.PRECISION=40;
.ROUND_NEAREST;

#include "def21060.h"
#define SAMPLES 300
#define SECTIONS 3

.SEGMENT /DM      dm_data;
.VAR      inbuf[SAMPLES] = 1.0, 0.0;          { input = unit
impulse }
.VAR      outbuf[SAMPLES];                    { ends up holding
impulse response }
.VAR      dline[SECTIONS*2];                  { w'', w', NEXT
w'', NEXT w', ... }
.ENDSEG;

.SEGMENT /PM      pm_data;
.VAR      coefs[SECTIONS*4]="iircoef.dat";    {
a12,a11,b12,b11,a22,a21,... }
.ENDSEG;

.SEGMENT /PM      rst_svc;
                                nop;

                                jump begin;
.ENDSEG;

.SEGMENT /PM      pm_code;
begin:  r0 = 0x21AD6B5A;

                                dm( WAIT ) = r0;

                                {
pmwait=0x0021;                                zero wait states for
all of PM }
                                { dmwait=0x8421;                                zero wait
states for all of DM }
                                b3=inbuf;  l3=0;
                                b4=outbuf;  l4=0;
                                l0=0; l1=0; l8=0;
                                m1=1; m8=1;
                                call cascaded_biquad_init (db);    { zero the
delay line }
                                r0=SECTIONS;

```

```

        b0=dline;
        lcntr=SAMPLES, do filtering until lce;
                                f8=dm(i3,1);
        call cascaded_biquad (db);      { input=F8,
output=F8 }
        b0=dline;
        b8=coefs;
filtering:  dm(i4,1)=f8;
done:      idle;
.ENDSEG;

```

## SIMD code.

```

{ main program for calling real biquad IIR filter
  input, output data stored in memory ... no interrupts
  follows example system described in ADSP-21020 Users
  Manual

```

```

        "iirmem.asm"          (requires file "iircoefs.dat")
        Analog Devices, Inc.  DSP Applications  P.O.Box 9106
        Norwood, MA 02062
        Christoph D. Cavigioli ... 25-Apr-1991
        04-Mar-1997: TEL : Modified original to execute two
        independent sample sets in parallel
    }
.EXTERN cascaded_biquad, cascaded_biquad_init;
.GLOBAL coefs, dline;
.PRECISION=40;
.ROUND_NEAREST;

#include "def21060.h"
#define SAMPLES 300
#define SECTIONS 3

.SEGMENT /DM    dm_data;
.VAR    inbuf[SAMPLES*2] = 1.0, 1.0, 0.0, 0.0;      { This
is the buffer containing 2}
                                {interleaved sample streams }
.VAR    outbuf[SAMPLES*2];                          { ends up holding
2, interleaved, impulse responses }

```



```

.VAR    dline[SECTIONS*2*2];                { double and
interleave the delay line }
.ENDSEG;

.SEGMENT /PM    pm_data;
.VAR    coefs[SECTIONS*4*2]="iircoefs.dat";    { two,
interleaved sets of coefficients }
.ENDSEG;

.SEGMENT /PM    rst_svc;
                                nop;
                                jump begin;
.ENDSEG;

.SEGMENT /PM    pm_code;
begin:  r0 = 0x21AD6B5A;
                                dm( WAIT ) = r0;
                                {
pmwait=0x0021;                                zero wait states for
all of PM }
                                { dmwait=0x8421;                                zero wait
states for all of DM }
                                b3=inbuf;    l3=0;
                                b4=outbuf;    l4=0;
                                l0=0; l1=0; l8=0;
                                m1=2; m8=2;    {we're accessing two
interleaved data streams}
                                {increments must be explicitly
doubled}

                                bit set model PEYEN;{enable
SIMD mode}
                                call cascaded_biquad_init (db);    { zero the
delay line }
                                r0=SECTIONS;
                                b0=dline;
                                lcntr=SAMPLES, do filtering until lce;
                                f8=dm(i3,2);
                                call cascaded_biquad (db);    { input=F8,
output=F8 }
                                b0=dline;

```

```

                b8=coefs;
filtering:      dm(i4,2)=f8;
                                bit clr model PEYEN;{disable
SIMD mode}
done:          idle;
.ENDSEG;

```

**Performance results.** The performance results for this fragment are actually quite good. The only additional overhead for the SIMD implementation is the bit set and bit clear to enable and disable SIMD mode. With DMA overhead added (implied in the example), SISD throughput is around 35 instructions per sample. A SIMD implementation results in a throughput of around 17.5 instructions per sample for a 99.9% performance improvement.

## Case II: Macro Parallelism on loop scale (FFT)

### SISD code.

```

{ _Middle stages with radix-4 main butterfly_

{ m0=1 and m8=1 is still preset }
m1=-2;          { reverse step for twiddles }
m9=m1;
m2=3;           { forward step for twiddles }
m10=m2;
m5=4;           { first there are 4 groups }
r2=N/16;        { with N/16 butterflies in each group }
r3=N/16*3;      { step to next group }

lcntr=STAGES-2, do mstage until lce;
                    { do STAGES-2 stages }

i7=cosine;      { first real twiddle }
i15=sine;       { first imag twiddle }

r8=redata;
r9=imdata;

```

```

i0=r8;          { upper real      path }
r10=r8+r2;      i8=r9;          { upper imaginary path }
i1=r10;         { second real input path }
r10=r10+r2,     i4=r10;         { second real output path }
i2=r10;         { third real input path }
r10=r10+r2,     i5=r10;         { third real output path }
i3=r10;         { fourth real input path }
r10=r9+r2,     i6=r10;         { fourth real output path }
i9=r10;         { second imag input path }
r10=r10+r2,     i12=r10;        { second imag output path }
i10=r10;        { third imag input path }
r10=r10+r2,     i13=r10;        { third imag output path }
i11=r10;        { fourth imag input path }
i14=r10;        { fourth imag output path }
m4=r3;
m12=r3;
r4=r3+1,        m6=r2;
m3=r4;
r2=r2-1,        m11=r4;
m7=r2;

lcctr=m5,       do mgroup until lce;    { do m5 groups }
f0=dm(i7,m0),   f5=pm(i9,m8);
f8=f0*f5, f4=dm(i1,m0),   f1=pm(i15,m8);
f9=f0*f4;
f12=f1*f5, f0=dm(i7,m0), f5=pm(i11,m8);
f13=f1*f4, f12=f9+f12, f4=dm(i3,m0),   f1=pm(i15,m8);
f8=f0*f4, f2=f8-f13;
f13=f1*f5;
f9=f0*f5, f8=f8+f13, f0=dm(i7,m1), f5=pm(i10,m8);
f13=f1*f4, f12=f8+f12, f14=f8-f12, f4=dm(i2,m0),
f1=pm(i15,m9);
f11=f0*f4;
f13=f1*f5, f6=f9-f13;
f9=f0*f5, f13=f11+f13, f11=dm(i0,0);
f13=f1*f4, f8=f11+f13, f10=f11-f13;

{_____Do m7 radix-4 butterflies_____}
lcctr=m7, do mr4bfly until lce;
f13=f9-f13, f4=dm(i1,m0), f5=pm(i9,m8);
f2=f2+f6, f15=f2-f6, f0=dm(i7,m0), f1=pm(i15,m8);

```

```

f8=f0*f4, f3=f8+f12, f7=f8-f12, f9=pm(i8,0);
f12=f1*f5, f9=f9+f13, f11=f9-f13, f13=f2;
f8=f0*f5, f12=f8+f12, f0=dm(i7,m0), f5=pm(i11,m8);
f13=f1*f4, f9=f9+f13, f6=f9-f13, f4=dm(i3,m0),
f1=pm(i15,m8);
f8=f0*f4, f2=f8-f13, dm(i0,m0)=f3, pm(i8,m8)=f9;
f13=f1*f5, f11=f11+f14, f7=f11-f14, dm(i4,m0)=f7,
pm(i12,m8)=f6;
f9=f0*f5, f8=f8+f13, f0=dm(i7,m1), f5=pm(i10,m8);
f13=f1*f4, f12=f8+f12, f14=f8-f12, f4=dm(i2,m0),
f1=pm(i15,m9);
f11=f0*f4, f3=f10+f15, f8=f10-f15, pm(i13,m8)=f11;
f13=f1*f5, f6=f9-f13, dm(i6,m0)=f8, pm(i14,m8)=f7;
f9=f0*f5, f13=f11+f13, f11=dm(i0,0);
mr4bfly:
f13=f1*f4, f8=f11+f13, f10=f11-f13, dm(i5,m0)=f3;
{ _____End radix-4 butterfly_____ }
{          dummy for address update          *          * }
f13=f9-f13, f0=dm(i7,m2), f1=pm(i15,m10);
f2=f2+f6, f15=f2-f6, f0=dm(i1,m4), f1=pm(i9,m12);
f3=f8+f12, f7=f8-f12, f9=pm(i8,0);
f9=f9+f13, f11=f9-f13, f0=dm(i2,m4);
f9=f9+f2, f6=f9-f2, f0=dm(i3,m4), f1=pm(i10,m12);
dm(i0,m3)=f3, pm(i8,m11)=f9;
f11=f11+f14, f7=f11-f14, dm(i4,m3)=f7, pm(i12,m11)=f6;
f3=f10+f15, f8=f10-f15, pm(i13,m11)=f11;
dm(i6,m3)=f8, pm(i14,m11)=f7;
mgroup: dm(i5,m3)=f3, f1=pm(i11,m12);

r3=m4;
r1=m5;
r2=m6;
r3=ashift r3 by -2;          { groupstep/4 }
r1=ashift r1 by 2;           { groups*4 }
m5=r1;
mstage: r2=ashift r2 by -2;   { butterflies/4 }

```

**SIMD code.**

```

{ _Middle stages with radix-4 main butterfly_ }

{ m0=2 and m8=2 is still preset }

```

```

{2x the data stride, we're feeding 2 execute streams}
m1=-4; { reverse step for twiddles, 2x again, }
        {2 interleaved twiddle sets }

m9=m1;
m2=6;   { forward step for twiddles, 2x again, }
        {2 interleaved twiddle sets }

m10=m2;
m5=4;   { first there are 4 groups }
r2=N/16; { with N/16 butterflies in each group }
r3=N/16*3; { step to next group }

lcnt=STAGES-2, do mstage until lce;
{ do STAGES-2 stages }

i7=cosine; { first real twiddle }
i15=sine;  { first imag twiddle }

r8=redata;
r9=imdata;

i0=r8; { upper realpath }
r10=r8+r2; i8=r9; { upper imaginary path }
i1=r10; { second real input path }
r10=r10+r2, i4=r10; { second real output path }
i2=r10; { third real input path }
r10=r10+r2, i5=r10; { third real output path }
i3=r10; { fourth real input path }
r10=r9+r2, i6=r10; { fourth real output path }
i9=r10; { second imag input path }
r10=r10+r2, i12=r10; { second imag output path }
i10=r10; { third imag input path }
r10=r10+r2, i13=r10; { third imag output path }
i11=r10; { fourth imag input path }
i14=r10; { fourth imag output path }
m4=r3;
r4=r3+1, m12=r3;
r4=r4+1, m6=r2; {adjust data group stride due}
                  {to SIMD data access pattern}

m3=r4;
r2=ashift r2 by -1; {divide number of inner loop passes}
                    {by 2 since we're }
                    {processing twice as}

```

```

                                { much data each time}
                                {thru the loop}

r2=r2-1, m11=r4;
m7=r2;
bit set model |PEYEN; {enable SIMD mode}
nop;    { one cycle for mode to take effect }

lcctr=m5,          do mgroup until lce;    { do m5 groups }
f0=dm(i7,m0),    f5=pm(i9,m8);
f8=f0*f5, f4=dm(i1,m0), f1=pm(i15,m8);
f9=f0*f4;
f12=f1*f5, f0=dm(i7,m0), f5=pm(i11,m8);
f13=f1*f4, f12=f9+f12, f4=dm(i3,m0), f1=pm(i15,m8);
f8=f0*f4, f2=f8-f13;
f13=f1*f5;
f9=f0*f5, f8=f8+f13, f0=dm(i7,m1), f5=pm(i10,m8);
f13=f1*f4, f12=f8+f12, f14=f8-f12, f4=dm(i2,m0),
f1=pm(i15,m9);
f11=f0*f4;
f13=f1*f5, f6=f9-f13;
f9=f0*f5, f13=f11+f13, f11=dm(i0,0);
f13=f1*f4, f8=f11+f13, f10=f11-f13;
{_____Do m7 radix-4 butterflies_____}
lcctr=m7, do mr4bfly until lce;
f13=f9-f13, f4=dm(i1,m0), f5=pm(i9,m8);
f2=f2+f6, f15=f2-f6, f0=dm(i7,m0), f1=pm(i15,m8);
f8=f0*f4, f3=f8+f12, f7=f8-f12, f9=pm(i8,0);
f12=f1*f5, f9=f9+f13, f11=f9-f13, f13=f2;
f8=f0*f5, f12=f8+f12, f0=dm(i7,m0), f5=pm(i11,m8);
f13=f1*f4, f9=f9+f13, f6=f9-f13, f4=dm(i3,m0),
f1=pm(i15,m8);
f8=f0*f4, f2=f8-f13, dm(i0,m0)=f3, pm(i8,m8)=f9;
f13=f1*f5, f11=f11+f14, f7=f11-f14, dm(i4,m0)=f7,
pm(i12,m8)=f6;
f9=f0*f5, f8=f8+f13, f0=dm(i7,m1), f5=pm(i10,m8);
f13=f1*f4, f12=f8+f12, f14=f8-f12, f4=dm(i2,m0),
f1=pm(i15,m9);
f11=f0*f4, f3=f10+f15, f8=f10-f15,
pm(i13,m8)=f11;
f13=f1*f5, f6=f9-f13, dm(i6,m0)=f8, pm(i14,m8)=f7;
f9=f0*f5, f13=f11+f13, f11=dm(i0,0);
mr4bfly;

```

```

f13=f1*f4, f8=f11+f13, f10=f11-f13, dm(i5,m0)=f3;
{_____End radix-4 butterfly_____}
{          dummy for address update          * *      }
f13=f9-f13, f0=dm(i1,m4), f1=pm(i9,m12);
f2=f2+f6, f15=f2-f6, f0=dm(i7,m2), f1=pm(i15,m10);
f3=f8+f12, f7=f8-f12, f9=pm(i8,0);
f9=f9+f13, f11=f9-f13, f0=dm(i2,m4), f1=pm(i11,m12);
f9=f9+f2, f6=f9-f2, f0=dm(i3,m4), f1=pm(i10,m12);
dm(i0,m3)=f3,    pm(i8,m11)=f9;
f11=f11+f14, f7=f11-f14, dm(i4,m3)=f7, pm(i12,m11)=f6;
f3=f10+f15, f8=f10-f15;
dm(i5,m3)=f3, pm(i13,m11)=f11;
mgroup: dm(i6,m3)=f8,    pm(i14,m11)=f7;

bit clr model PEYEN; {disable SIMD mode}
r3=m4;
r1=m5;
r2=m6;
r3=ashift r3 by -2;          { groupstep/4 }
r1=ashift r1 by 2;           { groups*4 }
m5=r1;
mstage: r2=ashift r2 by -2;          { butterflies/4 }

```

**Performance results.** The SISD implementation performs the entire 1024 point complex fft that uses this inner loop in 18,220 cycles. The SIMD implementation takes 10,175 cycles for a performance improvement of 79%. This particular loop takes 12473 cycles for SISD and 7102 cycles for SIMD for a performance improvement of 76%.

### Case III: Macro Scale Predicated Execution

#### SISD code.

```

{ Subroutine to compute the Sine or Cosine values of
  a floating point input.

  Y=SIN(X) or
  Y=COS(X)

```

```
Calling Registers
F0 = Input Value X=[6E-20, 6E20]
l_reg=0

Result Registers
F0 = Sine (or Cosine) of input Y=[-1,1]

Altered Registers
F0, F2, F4, F7, F8, F12
i_reg

Computation Time
38 Cycles

Version 0.03      7/4/90      Gordon A. Sterling
}

#include "asm_glob.h"

.SEGMENT/PM  Assembly_Library_Code_Space;

.PRECISION=MACHINE_PRECISION;

#define half_PI      1.57079632679489661923

.GLOBAL  cosine, sine;

cosine:  i_reg=sine_data;
         F8=ABS F0;           {Use absolute value of input}
         F12=0.5;             {Used later after modulo}
         F2=1.57079632679489661923;    { and add PI/2}
         JUMP compute_modulo (DB); {Follow sin code from here!}
         F4=F8+F2, F2=mem(i_reg,1);
         F7=1.0;              {Sign flag is set to 1}

sine:
         i_reg=sine_data;     {Load pointer to data}
         F7=1.0;              {Assume a positive sign}
         F12=0.0;             {Used later after modulo}
         F8=ABS F0, F2=mem(i_reg,1);
         F0=PASS F0, F4=F8;
         IF LT F7=-F7;{If input was negative, invert sign}
```



```

compute_modulo:F4=F4*F2;           {Compute fp modulo value}
R2=FIX F4;           {Round nearest fractional portion}
BTST R2 BY 0;           {Test for odd number}
IF NOT SZ F7=-F7;      {Invert sign if odd modulo}
F4=FLOAT R2;           {Return to fp}
F4=F4-F12, F2=mem(i_reg,1);
      {Add cos adjust if necessary, F4=XN}

compute_f:
F12=F2*F4, F2=mem(i_reg,1); {Compute XN*C1}
F2=F2*F4, F12=F8-F12;      {Compute |X|-XN*C1, and XN*C2}
F8=F12-F2, F4=mem(i_reg,1);
      {Compute f=(|X|-XN*C1)-XN*C2}
F12=ABS F8;               {Need magnitude for test}
F4=F12-F4, F12=F8;        {Check for sin(x)=x}
IF LT JUMP compute_sign; {Return with result in F1}

compute_R:
F12=F12*F12, F4=mem(i_reg,1);
LCNTR=6, DO compute_poly UNTIL LCE;
F4=F12*F4, F2=mem(i_reg,1); {Compute sum*g}
compute_poly:
F4=F2+F4;                 {Compute sum=sum+next r}
F4=F12*F4;                 {Final multiply by g}
RTS (DB), F4=F4*F8;        {Compute f*R}
F12=F4+F8;                 {Compute Result=f+f*R}
compute_sign:
F0=F12*F7;                 {Restore sign of result}
RTS; {This return only for sin(eps)=eps path}

.ENDSEG;

.SEGMENT/SPACE Assembly_Library_Data_Space;

.PRECISION=MEMORY_PRECISION;

.VAR sine_data[11] =
0.31830988618379067154,{1/PI}
3.14160156250000000000,{C1, almost PI}
-8.908910206761537356617E-6,{C2, PI=C1+C2}
9.536743164E-7, {eps, sin(eps)=eps}

```

```

-0.737066277507114174E-12,{R7}
0.160478446323816900E-9,{R6}
-0.250518708834705760E-7,{R5}
0.275573164212926457E-5,{R4}
-0.198412698232225068E-3,{R3}
0.83333333327592139E-2,{R2}
-0.166666666666659653,{R1}

```

```
.ENDSEG;
```

### SIMD code.

```

{*****}
{
    Calling Registers
{   F0 = Input Value X[n]=[6E-20, 6E20]   PEx rf }
{   FS0 = Input Value X[n+1]=[6E-20, 6E20] PEy rf }
{   l_reg=0
{
{
    Result Registers
{   F0 = Sine of input Y[n]=[-1,1]        PEx rf }
{   SF0 = Sine of input Y[n+1]=[-1,1]    PEy rf }
{
{
    Avg Computation Time Per Result
{   20.5 Cycles    (could be 20)
{
{
    Version 0.03      7/4/90Gordon A. Sterling }
{               0.10    24-Jun-1997 : TEL
{*****}
}

```

```

#include "asm_glob.h"
#include "def21060.h";

```

```
.SEGMENT/PMAssembly_Library_Code_Space;
```

```
.PRECISION=MACHINE_PRECISION;
```

```
#define half_PI    1.57079632679489661923
```

```
.GLOBALsine_simd;
```

```

sine_simd:
    bit set model |PEYEN;    {enable SIMD mode}

```

```

    i_reg=sine_data;           {Load pointer to data}
    F7=1.0;                    {Assume a positive sign}
    F12=0.0;                    {Used later after modulo}
    F8=ABS F0, F2=mem(i_reg,2);
    F0=PASS F0, F4=F8;
    IF LT F7=-F7;{If input was negative, invert sign}

compute_modulo:
    F4=F4*F2;                    {Compute fp modulo value}
    R2=FIX F4;                  {Round nearest fractional portion}
    BTST R2 BY 0;                {Test for odd number}
    IF NOT sz F7=-F7;           {Invert sign if odd modulo}
    F4=FLOAT R2;                {Return to fp}
    F4=F4-F12, F2=mem(i_reg,2);
    {Add cos adjust if necessary, F4=XN}

compute_f:
    F12=F2*F4, F2=mem(i_reg,2); {Compute XN*C1}
    F2=F2*F4, F12=F8-F12;       {Compute |X|-XN*C1, and XN*C2}
    F8=F12-F2, F4=mem(i_reg,2);
    {Compute f=(|X|-XN*C1)-XN*C2}
    F12=ABS F8;                  {Need magnitude for test}
    F4=F12-F4, F12=F8;           {Check for sin(x)=x}
    { IF LT JUMP compute_sign;   Return with result in F1}

    bit tst ASTATX 0x00000004;   {test for LT in each PE}

compute_R:IF NOT TF F12=F12*F12, F4=mem(i_reg,2);
    LCNTR=6, DO compute_poly UNTIL LCE;
    IF NOT TF F4=F12*F4, F2=mem(i_reg,2);
    {Compute sum*g}
compute_poly:
    IF NOT TF F4=F2+F4;           {Compute sum=sum+next r}
    IF NOT TF F4=F12*F4;         {Final multiply by g}
    IF NOT TF F4=F4*F8;          {Compute f*R}
    IF NOT TF F12=F4+F8;         {Compute Result=f+f*R}

compute_sign:
    F0=F12*F7;                    {Restore sign of result}

    RTS (db);
    bit clr MODE1 PEYEN;         {disable SIMD mode}

```

```

NOP;

.ENDSEG;
.SEGMENT/SPACE Assembly_Library_Data_Space;

.PRECISION=MEMORY_PRECISION;

.VAR sine_data[22] =
    0.31830988618379067154,{1/PI}
    0.31830988618379067154,{1/PI}
    3.14160156250000000000,{C1, almost PI}
    3.14160156250000000000,{C1, almost PI}
    -8.908910206761537356617E-6,{C2, PI=C1+C2}
    -8.908910206761537356617E-6,{C2, PI=C1+C2}
    9.536743164E-7,{eps, sin(eps)=eps}
    9.536743164E-7,{eps, sin(eps)=eps}
    -0.737066277507114174E-12,{R7}
    -0.737066277507114174E-12,{S7}
    0.160478446323816900E-9,{R6}
    0.160478446323816900E-9,{S6}
    -0.250518708834705760E-7,{R5}
    -0.250518708834705760E-7,{S5}
    0.275573164212926457E-5,{R4}
    0.275573164212926457E-5,{S4}
    -0.198412698232225068E-3,{R3}
    -0.198412698232225068E-3,{S3}
    0.83333333327592139E-2,{R2}
    0.83333333327592139E-2,{S2}
    -0.1666666666666659653,{R1}
    -0.1666666666666659653,{S1}

.ENDSEG;

```

**Performance results.** The SISD implementation performs the sine function on a single data point in 38 cycles. The SIMD implementation averages 20 cycles per data point for a performance improvement of 90%.