2.9.15 Task State Register (TSR)

The task state register (TSR) contains all of the status bits that determine or indicate the current execution environment. TSR is saved in the event of an interrupt or exception to the ITSR or NTSR, respectively. All bits are readable by the **MVC** instruction. The TSR is shown in Figure 2-28 and described in Table 2-23. The SGIE bit in TSR is used by the **DINT** and **RINT** instructions to globally disable and reenable interrupts.

The GIE and SGIE bits may be written in both User mode and Supervisor mode. The remaining bits all have restrictions on how they are written. See Section 8.2.4.2 for more information.

The GIE bit in TSR is physically the same bit as the GIE bit in CSR. It is retained in CSR for compatibility reasons, but placed in TSR so that it will be copied in the event of either an exception or an interrupt.

Figure 2-28. Task State Register (TSR)

31															16
Reserved															
R-0															

15	14	13 11	10	9	8	7 6	5	4	3	2	1	0
IB	SPLX	Reserved	EXC	INT	Rsvd	CXM	Rsvd	DBGM	XEN	GEE	SGIE	GIE
R-0	R-0	R-0	R/C-0	R-0	R-0	R/W-0	R-0	R/W-0	R/W-0	R/S-0	R/W-0	R/W-0

LEGEND: R = Readable by the **MVC** instruction; W = Writeable in Supervisor mode; C = Clearable in Supervisor mode; S = Can be set in Supervisor mode; *-n* = value after reset

Table 2-23. Task State Register (TSR) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0	Reserved. Read as 0.
15	IB		Interrupts blocked. Not writable by the **MVC** instruction; set only by hardware.
		0	Interrupts not blocked in previous cycle (interruptible point).
		1	Interrupts were blocked in previous cycle.
14	SPLX		SPLOOP executing. Not writable by the **MVC** instruction; set only by hardware.
		0	Not currently executing SPLOOP
		1	Currently executing SPLOOP
13-11	Reserved	0	Reserved. Read as 0.
10	EXC		Exception processing. Clearable by the **MVC** instruction in Supervisor mode. Not clearable by the **MVC** instruction in User mode.
		0	Not currently processing an exception.
		1	Currently processing an exception.
9	INT		Interrupt processing. Not writable by the **MVC** instruction.
		0	Not currently processing an interrupt.
		1	Currently processing an interrupt.
8	Reserved	0	Reserved. Read as 0.
7-6	CXM	0-3h	Current execution mode. Not writable by the **MVC** instruction; these bits reflect the current execution mode of the execute pipeline. CXM is set to 1 when you begin executing the first instruction in User mode. See Chapter 8 for more information.
		0	Supervisor mode
		1h	User mode
		2h-3h	Reserved (an attempt to set these values is ignored)
5	Reserved	0	Reserved. Read as 0.
4	DBGM		Emulator debug mask. Writable in Supervisor and User mode. Writable by emulator.
		0	Enables emulator capabilities.
		1	Disables emulator capabilities.

Table 2-23. Task State Register (TSR) Field Descriptions (continued)

Bit	Field	Value	Description
3	XEN		Maskable exception enable. Writable only in Supervisor mode.
		0	Disables all maskable exceptions.
		1	Enables all maskable exceptions.
2	GEE		Global exception enable. Can be set to 1 only in Supervisor mode. Once set, cannot be cleared except by reset.
		0	Disables all exceptions except the reset interrupt.
		1	Enables all exceptions.
1	SGIE		Saved global interrupt enable. Contains previous state of GIE bit after execution of a **DINT** instruction. Writable in Supervisor and User mode.
		0	Global interrupts remain disabled by the **RINT** instruction.
		1	Global interrupts are enabled by the **RINT** instruction.
0	GIE		Global interrupt enable. Same physical bit as the GIE bit in the control status register (CSR). Writable in Supervisor and User mode. See Section 5.2 for details on how the GIE bit affects interruptibility.
		0	Disables all interrupts except the reset interrupt and NMI (nonmaskable interrupt).
		1	Enables all interrupts.

Instruction Set

This chapter describes the assembly language instructions of the TMS320C64x DSP and TMS320C64x+ DSP . Also described are parallel operations, conditional operations, resource constraints, and addressing modes.

The C64x and C64x+ DSP uses all of the instructions available to the TMS320C62x DSP, but it also uses other instructions that are specific to the C64x and C64x+ DSP. These specific instructions include 8-bit and 16-bit extensions, nonaligned word loads and stores, data packing/unpacking operations.

Topic **Page**

3.1 Instruction Operation and Execution Notations

Table 3-1 explains the symbols used in the instruction descriptions.

Table 3-1. Instruction Operation and Execution Notations

Symbol	Meaning
abs(x)	Absolute value of x
and	Bitwise AND
-a	Perform 2s-complement subtraction using the addressing mode defined by the AMR
+a	Perform 2s-complement addition using the addressing mode defined by the AMR
b_i	Select bit i of source/destination b
bit_count	Count the number of bits that are 1 in a specified byte
bit_reverse	Reverse the order of bits in a 32-bit register
byte0	8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
byte1	8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
byte2	8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
byte3	8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
bv2	Bit vector of two flags for s2 or u2 data type
bv4	Bit vector of four flags for s4 or u4 data type
$b_{y..z}$	Selection of bits y through z of bit string b
cond	Check for either *creg* equal to 0 or *creg* not equal to 0
creg	3-bit field specifying a conditional register, see Section 3.5
*cst*n	n-bit constant field (for example, cst5)
dint	64-bit integer value (two registers)
dst_e	lsb32 of 64-bit *dst* (placed in even-numbered register of a 64-bit register pair)
dst_h	msb8 of 40-bit *dst* (placed in odd-numbered register of 64-bit register pair)
dst_l	lsb32 of 40-bit *dst* (placed in even-numbered register of a 64-bit register pair)
dst_o	msb32 of 64-bit *dst* (placed in odd-numbered register of 64-bit register pair)
dws4	Four packed signed 16-bit integers in a 64-bit register pair
dwu4	Four packed unsigned 16-bit integers in a 64-bit register pair
gmpy	Galois Field Multiply
i2	Two packed 16-bit integers in a single 32-bit register
i4	Four packed 8-bit integers in a single 32-bit register
int	32-bit integer value
lmb0(x)	Leftmost 0 bit search of x
lmb1(x)	Leftmost 1 bit search of x
long	40-bit integer value
lsb*n* or LSB*n*	n least-significant bits (for example, lsb16)
msb*n* or MSB*n*	n most-significant bits (for example, msb16)
nop	No operation
norm(x)	Leftmost nonredundant sign bit of x
not	Bitwise logical complement
op	Opfields
or	Bitwise OR
R	Any general-purpose register
ROTL	Rotate left
sat	Saturate
sbyte0	Signed 8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
sbyte1	Signed 8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
sbyte2	Signed 8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
sbyte3	Signed 8-bit value in the most-significant byte position in 32-bit register (bits 24-31)

Table 3-1. Instruction Operation and Execution Notations (continued)

Symbol	Meaning
scst*n*	n-bit signed constant field
se	Sign-extend
sint	Signed 32-bit integer value
slong	Signed 40-bit integer value
sllong	Signed 64-bit integer value
slsb16	Signed 16-bit integer value in lower half of 32-bit register
smsb16	Signed 16-bit integer value in upper half of 32-bit register
src1_e or *src2_e*	lsb32 of 64-bit *src* (placed in even-numbered register of a 64-bit register pair)
src1_h or *src2_h*	msb8 of 40-bit *src* (placed in odd-numbered register of 64-bit register pair)
src1_l or *src2_l*	lsb32 of 40-bit *src* (placed in even-numbered register of a 64-bit register pair)
src1_o or *src2_o*	msb32 of 64-bit *src* (placed in odd-numbered register of 64-bit register pair)
s2	Two packed signed 16-bit integers in a single 32-bit register
s4	Four packed signed 8-bit integers in a single 32-bit register
-s	Perform 2s-complement subtraction and saturate the result to the result size, if an overflow occurs
+s	Perform 2s-complement addition and saturate the result to the result size, if an overflow occurs
ubyte0	Unsigned 8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
ubyte1	Unsigned 8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
ubyte2	Unsigned 8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
ubyte3	Unsigned 8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
ucst*n*	n-bit unsigned constant field (for example, ucst5)
uint	Unsigned 32-bit integer value
ulong	Unsigned 40-bit integer value
ullong	Unsigned 64-bit integer value
ulsb16	Unsigned 16-bit integer value in lower half of 32-bit register
umsb16	Unsigned 16-bit integer value in upper half of 32-bit register
u2	Two packed unsigned 16-bit integers in a single 32-bit register
u4	Four packed unsigned 8-bit integers in a single 32-bit register
x clear *b,e*	Clear a field in x, specified by b (beginning bit) and e (ending bit)
x ext *l,r*	Extract and sign-extend a field in x, specified by l (shift left value) and r (shift right value)
x extu *l,r*	Extract an unsigned field in x, specified by l (shift left value) and r (shift right value)
x set *b,e*	Set field in x to all 1s, specified by b (beginning bit) and e (ending bit)
xint	32-bit integer value that can optionally use cross path
xor	Bitwise exclusive-ORs
xsint	Signed 32-bit integer value that can optionally use cross path
xslsb16	Signed 16 LSB of register that can optionally use cross path
xsmsb16	Signed 16 MSB of register that can optionally use cross path
xs2	Two packed signed 16-bit integers in a single 32-bit register that can optionally use cross path
xs4	Four packed signed 8-bit integers in a single 32-bit register that can optionally use cross path
xuint	Unsigned 32-bit integer value that can optionally use cross path
xulsb16	Unsigned 16 LSB of register that can optionally use cross path
xumsb16	Unsigned 16 MSB of register that can optionally use cross path
xu2	Two packed unsigned 16-bit integers in a single 32-bit register that can optionally use cross path
xu4	Four packed unsigned 8-bit integers in a single 32-bit register that can optionally use cross path
→	Assignment
+	Addition
++	Increment by 1
×	Multiplication

Table 3-1. Instruction Operation and Execution Notations (continued)

Symbol	Meaning
-	Subtraction
==	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<<	Shift left
>>	Shift right
>>s	Shift right with sign extension
>>z	Shift right with a zero fill
~	Logical inverse
&	Logical AND

3.2 Instruction Syntax and Opcode Notations

Table 3-2 explains the syntaxes and opcode fields used in the instruction descriptions.

Table 3-2. Instruction Syntax and Opcode Notations

Symbol	Meaning
baseR	base address register
creg	3-bit field specifying a conditional register, see Section 3.5
cst	constant
csta	constant a
cstb	constant b
cstn	n-bit constant field
dst	destination
dw	doubleword; 0 = word, 1 = doubleword
fcyc	SPLOOP fetch cycle
fstg	SPLOOP fetch stage
h	MVK or MVKH instruction
ii_n	bit n of the constant *ii*
ld/st	load or store; 0 = store, 1 = load
mode	addressing mode, see Section 3.8
na	nonaligned; 0 = aligned, 1 = nonaligned
N3	3-bit field
offsetR	register offset
op	opfield; field within opcode that specifies a unique instruction
op_n	bit n of the opfield
p	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
ptr	offset from either A4-A7 or B4-B7 depending on the value of the *s* bit. The *ptr* field is the 2 least-significant bits of the *src2* (*baseR*) field—bit 2 of register address is forced to 1.
r	LDDW/LDNDW/LDNW instruction
rsv	reserved
s	side A or B for destination; 0 = side A, 1 = side B.
sc	scaling mode; 0 = nonscaled, *offsetR/ucst5* is not shifted; 1 = scaled, *offsetR/ucst5* is shifted
scstn	n-bit signed constant field

Table 3-2. Instruction Syntax and Opcode Notations (continued)

Symbol	Meaning
$scst_n$	bit n of the signed constant field
sn	sign
src	source
src1	source 1
src2	source 2
stg_n	bit n of the constant *stg*
sz	data size select; 0 = primary size, 1 = secondary size (see Section 3.9.2.2)
t	side of source/destination (*src/dst*) register; 0 = side A, 1 = side B
ucstn	n-bit unsigned constant field
$ucst_n$	bit n of the unsigned constant field
unit	unit decode
x	cross path for *src2*; 0 = do not use cross path, 1 = use cross path
y	.D1 or .D2 unit; 0 = .D1 unit, 1 = .D2 unit
z	test for equality with zero or nonzero

3.2.1 32-Bit Opcode Maps

The C64x CPU and C64x+ CPU 32-bit opcodes are mapped in Appendix C through Appendix H.

3.2.2 16-Bit Opcode Maps

The C64x+ CPU 16-bit opcodes used for compact instructions are mapped in Appendix C through Appendix H. See Section 3.9 for more information about compact instructions.

3.3 Delay Slots

The execution of the additional instructions can be defined in terms of delay slots. The number of delay slots is equivalent to the number of additional cycles required after the source operands are read for the result to be available for reading. For a single-cycle type instruction (such as **CMPGT2**), source operands read in cycle *i* produce a result that can be read in cycle *i* + 1. For a 2-cycle instruction (such as **AVGU4**), source operands read in cycle *i* produce a result that can be read in cycle *i* + 2. For a four-cycle instruction (such as **DOTP2**), source operands read in cycle *i* produce a result that can be read in cycle *i* + 4. Table 3-3 shows the number of delay slots associated with each type of instruction.

Delay slots are equivalent to an execution or result latency. All of the instructions in the C64x and C64x+ DSP have a functional unit latency of 1. This means that a new instruction can be started on the functional unit each cycle. Single-cycle throughput is another term for single-cycle functional unit latency.

Table 3-3. Delay Slot and Functional Unit Latency

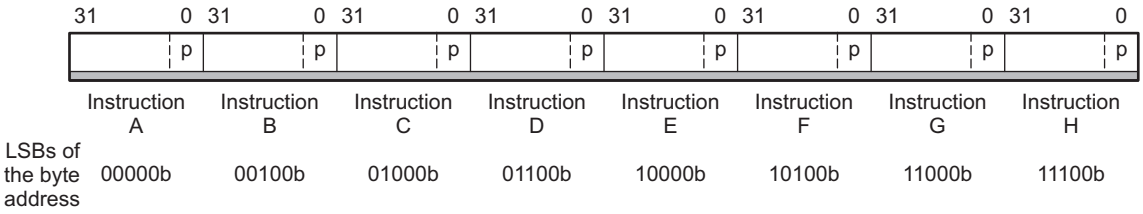
Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles (1)	Write Cycles (1)	Branch Taken (1)
NOP (no operation)	0	1			
Store	0	1	i	i	
Single cycle	0	1	i	i	
Two cycle	1	1	i	i + 1	
Multiply (16 × 16)	1	1	i	i + 1	
Four cycle	3	1	i	i + 3	
Load	4	1	i	i, i + 4 (2)	
Branch	5	1	i (3)		i + 5

(1) Cycle i is in the E1 pipeline phase.
(2) For loads, any address modification happens in cycle i. The loaded data is written into the register file in cycle i + 4.
(3) The branch to label, branch to IRP, and branch to NRP instructions do not read any general-purpose registers.

3.4 Parallel Operations

Instructions are always fetched eight words at a time. This constitutes a *fetch packet*. On the C64x CPU this will always be eight instructions. On the C64x+ CPU, this may be as many as 14 instructions due to the existence of compact instructions in a header based fetch packet. The basic format of a fetch packet is shown in Figure 3-1. Fetch packets are aligned on 256-bit (8-word) boundaries.

Figure 3-1. Basic Format of a Fetch Packet



The C64x+ CPU supports compact 16-bit instructions. Unlike the normal 32-bit instructions, the *p*-bit information for compact instructions is not contained within the instruction opcode. Instead, the *p*-bit is contained within the *p*-bits field within the fetch packet header. See Section 3.9 for more information.

The execution of the individual noncompact instructions is partially controlled by a bit in each instruction, the *p*-bit. The *p*-bit (bit 0) determines whether the instruction executes in parallel with another instruction. The *p*-bits are scanned from left to right (lower to higher address). If the *p*-bit of instruction *I* is 1, then instruction *I* + 1 is to be executed in parallel with (in the same cycle as) instruction *I*. If the *p*-bit of instruction *I* is 0, then instruction *I* + 1 is executed in the cycle after instruction *I*. All instructions executing in parallel constitute an *execute packet*. An execute packet can contain up to eight instructions. Each instruction in an execute packet must use a different functional unit.

On the CPU, the execute packet can cross fetch packet boundaries, but will be limited to no more than eight instructions in a fetch packet. The last instruction in an execute packet will be marked with its *p*-bit cleared to zero. There are three types of *p*-bit patterns for fetch packets. These three *p*-bit patterns result in the following execution sequences for the eight instructions:

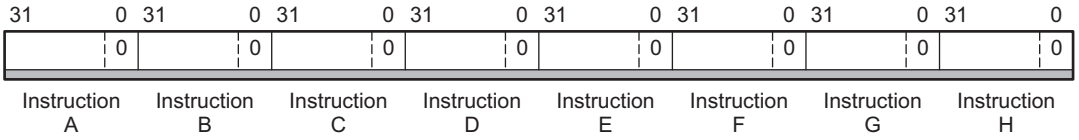
- Fully serial
- Fully parallel
- Partially serial

Example 3-1 through Example 3-3 show the conversion of a *p*-bit sequence into a cycle-by-cycle execution stream of instructions.

Example 3-1. Fully Serial p-Bit Pattern in a Fetch Packet

The eight instructions are executed sequentially.

This *p*-bit pattern:



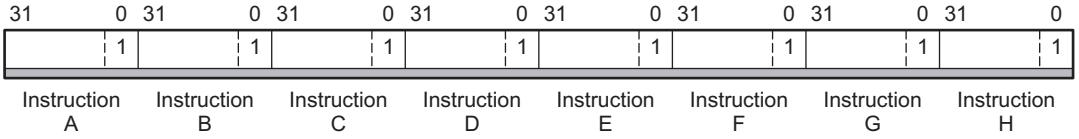
results in this execution sequence:

Cycle/Execute Packet	Instructions
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H

Example 3-2. Fully Parallel p-Bit Pattern in a Fetch Packet

All eight instructions are executed in parallel.

This *p*-bit pattern:

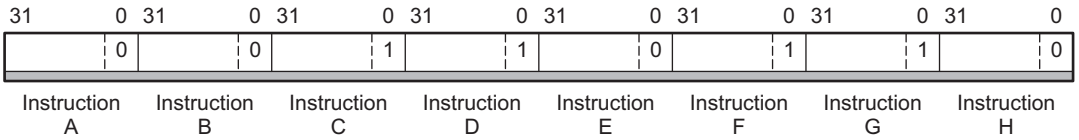


results in this execution sequence:

Cycle/Execute Packet	Instructions							
1	A	B	C	D	E	F	G	H

Example 3-3. Partially Serial p-Bit Pattern in a Fetch Packet

This *p*-bit pattern:



results in this execution sequence:

Cycle/Execute Packet	Instructions		
1	A		
2	B		
3	C	D	E
4	F	G	H

3.4.1 Example Parallel Code

The vertical bars || signify that an instruction is to execute in parallel with the previous instruction. The code for the fetch packet in Example 3-3 would be represented as this:

```
      instruction A

      instruction B

      instruction C
||    instruction D
||    instruction E

      instruction F
||    instruction G
||    instruction H
```

3.4.2 Branching Into the Middle of an Execute Packet

If a branch into the middle of an execute packet occurs, all instructions at lower addresses are ignored. In Example 3-3, if a branch to the address containing instruction D occurs, then only D and E execute. Even though instruction C is in the same execute packet, it is ignored. Instructions A and B are also ignored because they are in earlier execute packets. If your result depends on executing A, B, or C, the branch to the middle of the execute packet will produce an erroneous result.

3.5 Conditional Operations

Most instructions can be conditional. The condition is controlled by a 3-bit opcode field (*creg*) that specifies the condition register tested, and a 1-bit field (*z*) that specifies a test for zero or nonzero. The four MSBs of every opcode are *creg* and *z*. The specified condition register is tested at the beginning of the E1 pipeline stage for all instructions. For more information on the pipeline, see Chapter 4. If *z* = 1, the test is for equality with zero; if *z* = 0, the test is for nonzero. The case of *creg* = 0 and *z* = 0 is treated as always true to allow instructions to be executed unconditionally. The *creg* field is encoded in the instruction opcode as shown in Table 3-4.

Compact (16-bit) instructions on the C64x+ DSP do not contain a *creg* field and always execute unconditionally. See Section 3.9 for more information.

Table 3-4. Registers That Can Be Tested by Conditional Operations

Specified Conditional Register	*creg*			*z*
Bit:	**31**	**30**	**29**	**28**
Unconditional	0	0	0	0
Reserved	0	0	0	1
B0	0	0	1	z
B1	0	1	0	z
B2	0	1	1	z
A1	1	0	0	z
A2	1	0	1	z
A0	1	1	0	z
Reserved	1	1	1	x(1)

(1) x can be any value.

Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name. The following execute packet contains two **ADD** instructions in parallel. The first **ADD** is conditional on B0 being nonzero. The second **ADD** is conditional on B0 being zero. The character ! indicates the inverse of the condition.

```
      [B0]    ADD       .L1       A1,A2,A3
||    [!B0]   ADD       .L2       B1,B2,B3
```

The above instructions are mutually exclusive, only one will execute. If they are scheduled in parallel, mutually exclusive instructions are constrained as described in Section 3.7. If mutually exclusive instructions share any resources as described in Section 3.7, they cannot be scheduled in parallel (put in the same execute packet), even though only one will execute.

The act of making an instruction conditional is often called predication and the conditional register is often called the predication register.

3.6 SPMASKed Operations

On the C64x+ CPU, the **SPMASK** and **SPMASKR** instructions can be used to inhibit the execution of instructions from the SPLOOP buffer. The selection of which instruction to inhibit can be specified by the **SPMASK** or **SPMASKR** instruction argument or can be marked by the addition of a caret (^) next to the parallel code marker as shown below:

```
     SPMASK
||^  LDW    .D1   *A0,A1            ;This instruction is SPMASKed
||^  LDW    .D2   *B0,B1            ;This instruction is SPMASKed
||   MPY    .M1   A3,A4,A5          ;This instruction is Not SPMASKed
```

See Chapter 7 for more information.

3.7 Resource Constraints

No two instructions within the same execute packet can use the same resources. Also, no two instructions can write to the same register during the same cycle. The following sections describe how an instruction can use each of the resources.

3.7.1 Constraints on Instructions Using the Same Functional Unit

Two instructions using the same functional unit cannot be issued in the same execute packet.

The following execute packet is invalid:

```
    ADD .S1    A0, A1, A2     ;.S1 is used for
||  SHR .S1    A3, 15, A4     ;...both instructions
```

The following execute packet is valid:

```
    ADD .L1    A0, A1, A2     ;Two different functional
||  SHR .S1    A3, 15, A4     ;...units are used
```

3.7.2 Constraints on the Same Functional Unit Writing in the Same Instruction Cycle

The .M unit has two 32-bit write ports; so the results of a 4-cycle 32-bit instruction and a 2-cycle 32-bit instruction operating on the same .M unit can write their results on the same instruction cycle. Any other combination of parallel writes on the .M unit will result in a conflict. On the C64x+ DSP this will result in an exception.

On the C64x DSP and C64x+ DSP , this will result in erroneous values being written to the destination registers.

For example, the following sequence is valid and results in both A2 and A5 being written by the .M1 unit on the same cycle.

```
DOTP2  .M1   A0,A1,A2            ;This instruction has 3 delay slots
NOP
AVG2   .M1   A4,A5               ;This instruction has 1 delay slot
NOP                              ;Both A2 and A5 get written on this cycle
```

The following sequence is invalid. The attempt to write 96 bits of output through 64-bits of write port will fail.

```
SMPY2  .M1   A5,A6,A9:A8         ;This instruction has 3 delay slots; but generates a 64 bit
result
NOP
MPY    .M1   A1,A2,A3            ;This instruction has 1 delay slot
NOP
```

3.7.3 Constraints on Cross Paths (1X and 2X)

Up to two units (.S, .L, .D, or .M unit) per data path, per execute packet, can read a source operand from its opposite register file via the cross paths (1X and 2X) provided that each unit is reading the same operand.

For example, the .S1 unit can read both its operands from the A register file; or it can read an operand from the B register file using the 1X cross path and the other from the A register file. The use of a cross path is denoted by an X following the functional unit name in the instruction syntax (as in S1X).

The following execute packet is invalid because the 1X cross path is being used for two different B register operands:

```
   MV .S1X B0, A0 ; Invalid. Instructions are using the 1X cross path
|| MV .L1X B1, A1 ; with different B registers
```

The following execute packet is valid because all uses of the 1X cross path are for the same B register operand, and all uses of the 2X cross path are for the same A register operand:

```
    ADD .L1X A0,B1,A1 ; Instructions use the 1X with B1
 || SUB .S1X A2,B1,A2 ; 1X cross paths using B1
 || AND .D1  A4,A1,A3 ;
 || MPY .M1  A6,A1,A4 ;
 || ADD .L2  B0,B4,B2 ;
 || SUB .S2X B4,A4,B3 ; 2X cross paths using A4
 || AND .D2X B5,A4,B4 ; 2X cross paths using A4
 || MPY .M2  B6,B4,B5 ;
```

The following execute packet is invalid because more than two functional units use the same cross path operand:

```
   MV .L2X A0, B0 ; 1st cross path move
|| MV .S2X A0, B1 ; 2nd cross path move
|| MV .D2X A0, B2 ; 3rd cross path move
```

The operand comes from a register file opposite of the destination, if the x bit in the instruction field is set.

3.7.4 Cross Path Stalls

The DSP introduces a delay clock cycle whenever an instruction attempts to read a register via a cross path that was updated in the previous cycle. This is known as a cross path stall. This stall is inserted automatically by the hardware, no **NOP** instruction is needed. It should be noted that no stall is introduced if the register being read has data placed by a load instruction, or if an instruction reads a result one cycle after the result is generated.

Here are some examples:

```
ADD  .S1   A0, A0, A1  ; / Stall is introduced; A1 is updated
                       ;   1 cycle before it is used as a
ADD  .S2X  A1, B0, B1  ; \ cross path source


ADD  .S1   A0, A0, A1  ; / No stall is introduced; A0 not updated
                       ;   1 cycle before it is used as a cross
ADD  .S2X  A0, B0, B1  ; \ path source


LDW  .D1 *++A0[1], A1  ; / No stall is introduced; A1 is the load
                       ;   destination
NOP  4                 ;   NOP 4 represents 4 instructions to
ADD  .S2X  A1, B0, B1  ; \ be executed between the load and add.
LDW  .D1  *++A0[1], A1 ; / Stall is introduced; A0 is updated
ADD  .S2X   A0, B0, B1 ;   1 cycle before it is used as a
                       ; \ cross path source
```

It is possible to avoid the cross path stall by scheduling an instruction that reads an operand via the cross path at least one cycle after the operand is updated. With appropriate scheduling, the DSP can provide one cross path operand per data path per cycle with no stalls. In many cases, the TMS320C6000 Optimizing Compiler and Assembly Optimizer automatically perform this scheduling.

3.7.5 Constraints on Loads and Stores

The data address paths named DA1 and DA2 are each connected to the .D units in both data paths. Load and store instructions can use an address pointer from one register file while loading to or storing from the other register file. Two load and store instructions using a destination/source from the same register file cannot be issued in the same execute packet. The address register must be on the same side as the .D unit used.

The DA1 and DA2 resources and their associated data paths are specified as T1 and T2, respectively. T1 consists of the DA1 address path and the LD1 and ST1 data paths. LD1 is comprised of LD1a and LD1b to support 64-bit loads; ST1 is comprised of ST1a and ST1b to support 64-bit stores. Similarly, T2 consists of the DA2 address path and the LD2 and ST2 data paths. LD2 is comprised of LD2a and LD2b to support 64-bit loads; ST2 is comprised of ST2a and ST2b to support 64-bit stores. The T1 and T2 designations appear in the functional unit fields for load and store instructions.

The DSP can access words and doublewords at any byte boundary using nonaligned loads and stores. As a result, word and doubleword data does not need alignment to 32-bit or 64-bit boundaries. No other memory access may be used in parallel with a nonaligned memory access. The other .D unit can be used in parallel, as long as it is not performing a memory access.

The following execute packet is invalid:

```
   LDNW  .D2T2 *B2[B12],B13 ; \ Two memory operations,
|| LDB   .D1T1 *A2,A14      ; / one non-aligned
```

The following execute packet is valid:

```
   LDNW .D2T2 *B2[B12], A13 ; \ One non-aligned memory
                            ;   operation,
|| ADD  .D1x  A12, B13, A14 ;   one non-memory .D unit
                            ; / operation
```

3.7.6 Constraints on Long (40-Bit) Data

Both the C62x and C67x device families had constraints on the number of simultaneous reads and writes of 40-bit data due to shared data paths.

The C64x and C64x+ CPU maintain separate datapaths to each functional unit, so these constraints are removed.

The following, for example, is valid:

```
    DDOTPL2   .M1    A1:A0,A2,A5:A4
||  DDOTPL2   .M2    B1:B0,B2,B5:B4
||  STDW      .D1    A9:A8,*A6
||  STDW      .D2    B9:B8,*B6
||  SUB       .L1    A25:A24,A20,A31:A30
||  SUB       .L2    B25:B24,B20,B31:B30
||  SHL       .S1    A11:A10,5,A13:A12
||  SHL       .S2    B11:B10,8,B13:B12
```

3.7.7 Constraints on Register Reads

More than four reads of the same register cannot occur on the same cycle. Conditional registers are not included in this count.

The following execute packets are invalid:

```
   MPY  .M1   A1, A1, A4  ; five reads of register A1
|| ADD  .L1   A1, A1, A5
|| SUB  .D1   A1, A2, A3


   MPY  .M1   A1, A1, A4  ; five reads of register A1
|| ADD  .L1   A1, A1, A5
|| SUB  .D2x  A1, B2, B3
```

The following execute packet is valid:

```
          MPY  .M1  A1, A1, A4  ; only four reads of A1
|| [A1]   ADD  .L1  A0, A1, A5
||        SUB  .D1  A1, A2, A3
```

3.7.8 Constraints on Register Writes

Two instructions cannot write to the same register on the same cycle. Two instructions with the same destination can be scheduled in parallel as long as they do not write to the destination register on the same cycle. For example, an **MPY** issued on cycle *i* followed by an **ADD** on cycle *i* + 1 cannot write to the same register because both instructions write a result on cycle *i* + 1. Therefore, the following code sequence is invalid unless a branch occurs after the **MPY**, causing the **ADD** not to be issued.

```
MPY .M1  A0, A1, A2
ADD .L1  A4, A5, A2
```

However, this code sequence is valid:

```
       MPY  .M1  A0, A1, A2
||     ADD  .L1  A4, A5, A2
```

Figure 3-2 shows different multiple-write conflicts. For example, **ADD** and **SUB** in execute packet L1 write to the same register. This conflict is easily detectable.

MPY in packet L2 and **ADD** in packet L3 might both write to B2 simultaneously; however, if a branch instruction causes the execute packet after L2 to be something other than L3, a conflict would not occur. Thus, the potential conflict in L2 and L3 might not be detected by the assembler. The instructions in L4 do not constitute a write conflict because they are mutually exclusive. In contrast, because the instructions in L5 may or may not be mutually exclusive, the assembler cannot determine a conflict. If the pipeline does receive commands to perform multiple writes to the same register, the result is undefined.

Figure 3-2. Examples of the Detectability of Write Conflicts by the Assembler

```
L1:                ADD  .L2  B5,B6,B7     ; \ detectable, conflict
       ||          SUB  .S2  B8,B9,B7     ; /
L2:                MPY  .M2  B0,B1,B2     ; \ not detectable
L3:                ADD  .L2  B3,B4,B2     ; /
L4:         [!B0]  ADD  .L2  B5,B6,B7     ; \ detectable, no conflict
       ||   [B0]   SUB  .S2  B8,B9,B7     ; /
L5:         [!B1]  ADD  .L2  B5,B6,B7     ; \ not detectable
       ||   [B0]   SUB  .S2  B8,B9,B7     ; /
```

3.7.9 Constraints on AMR Writes

A write to the addressing mode register (AMR) using the **MVC** instruction that is immediately followed by a **LD, ST, ADDA,** or **SUBA** instruction causes a 1 cycle stall, if the **LD, ST, ADDA,** or **SUBA** instruction uses the A4-A7 or B4-B7 registers for addressing.

3.7.10 Constraints on Multicycle NOPs

Two instructions that generate multicycle NOPs cannot share the same execute packet. Instructions that generate a multicycle **NOP** are:

- NOP *n* (where *n* > 1)
- IDLE
- BNOP target, *n* (for all values of *n*, regardless of predication)
- ADDKPC label, reg, *n* (for all values of *n*, regardless of predication)

3.7.11 Constraints on Unitless Instructions

3.7.11.1 SPLOOP Restrictions

The **NOP**, **NOP *n***, and **BNOP** instructions are the only unitless instructions allowed to be used in an SPLOOP(D/W) body. The assembler disallows the use of any other unitless instruction in the loop body.

See Chapter 7 for more information.

3.7.11.2 BNOP <disp>,n

A **BNOP** instruction cannot be placed in parallel with the following instructions if the **BNOP** has a non-zero NOP count:

- ADDKPC
- CALLP
- NOP *n*

3.7.11.3 DINT

A **DINT** instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- MVC reg, CSR
- B IRP
- B NRP
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

A **DINT** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.4 IDLE

An **IDLE** instruction cannot be placed in parallel with the following instructions:

- DINT
- NOP *n* (if *n* > 1)
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

An **IDLE** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.5 NOP n

A **NOP *n*** (with *n* > 1) instruction cannot be placed in parallel with other multicycle **NOP** counts (**ADDKPC**, **BNOP**, **CALLP**) with the exception of another **NOP *n*** where the NOP count is the same. A **NOP *n*** (with *n* > 1) instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

3.7.11.6 RINT

A **RINT** instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- MVC reg, CSR
- B IRP
- B NRP
- DINT
- IDLE
- NOP *n* (if *n* > 1)
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

A **RINT** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.7 SPKERNEL(R)

An **SPKERNEL(R)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)

- SPMASK(R)
- SWE
- SWENR

An **SPKERNEL(R)** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.8 SPLOOP(D/W)

An **SPLOOP(D/W)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPKERNEL(R)
- SPMASK(R)
- SWE
- SWENR

An **SPLOOP(D/W)** instruction can be placed in parallel with the **NOP** instruction:

3.7.11.9 SPMASK(R)

An **SPMASK(R)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)
- SPKERNEL(R)
- SWE
- SWENR

An **SPMASK(R)** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.10 SWE

An **SWE** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)
- SPKERNEL(R)
- SWENR

An **SWE** instruction can be placed in parallel with the **NOP** instruction.

3.7.11.11 SWENR

An **SWENR** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPLOOP(D/W)

- SPKERNEL(R)
- SWE

An **SWENR** instruction can be placed in parallel with the **NOP** instruction.

3.8 Addressing Modes

The addressing modes on the DSP are linear, circular using BK0, and circular using BK1. The addressing mode is specified by the addressing mode register (AMR), described in Section 2.8.3.

All registers can perform linear addressing. Only eight registers can perform circular addressing: A4-A7 are used by the .D1 unit, and B4-B7 are used by the .D2 unit. No other units can perform circular addressing. **LDB(U)/LDH(U)/LDW, STB/STH/STW, LDNDW, LDNW, STNDW, STNW, LDDW, STDW, ADDAB/ADDAH/ADDAW/ADDAD,** and **SUBAB/SUBAH/SUBAW** instructions all use AMR to determine what type of address calculations are performed for these registers. There is no **SUBAD** instruction.

3.8.1 Linear Addressing Mode

3.8.1.1 LD and ST Instructions

For load and store instructions, linear mode simply shifts the *offsetR/cst* operand to the left by 3, 2, 1, or 0 for doubleword, word, halfword, or byte access, respectively; and then performs an add or a subtract to *baseR* (depending on the operation specified). The **LDNDW** and **STNDW** instructions also support nonscaled offsets. In nonscaled mode, the *offsetR/cst* is not shifted before adding or subtracting from the *baseR*.

For the preincrement, predecrement, positive offset, and negative offset address generation options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

3.8.1.2 ADDA and SUBA Instructions

For integer addition and subtraction instructions, linear mode simply shifts the *src1/cst* operand to the left by 3, 2, 1, or 0 for doubleword, word, halfword, or byte data sizes, respectively, and then performs the add or subtract specified.

3.8.2 Circular Addressing Mode

The BK0 and BK1 fields in AMR specify the block sizes for circular addressing, see Section 2.8.3.

3.8.2.1 LD and ST Instructions

As with linear address arithmetic, *offsetR/cst* is shifted left by 3, 2, 1, or 0 according to the data size, and is then added to or subtracted from *baseR* to produce the final address. Circular addressing modifies this slightly by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N + 1 unchanged after address arithmetic. The resulting address is bounded to $2^{(N + 1)}$ range, regardless of the size of the *offsetR/cst*.

The circular buffer size in AMR is not scaled; for example, a block-size of 8 is 8 bytes, not 8 times the data size (byte, halfword, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Example 3-4 shows an **LDW** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value in AMR for this example is 0004 0001h.

Example 3-4. LDW Instruction in Circular Mode

```
LDW         .D1         *++A4[9],A1
```

	Before LDW		1 cycle after LDW (1)		5 cycles after LDW
A4	0000 0100h	A4	0000 0104h	A4	0000 0104h
A1	xxxx xxxxh	A1	xxxx xxxxh	A1	1234 5678h
mem 104h	1234 5678h	mem 104h	1234 5678h	mem 104h	1234 5678h

(1) **Note:** 9h words is 24h bytes. 24h bytes is 4 bytes beyond the 32-byte (20h) boundary 100h-11Fh; thus, it is wrapped around to (124h - 20h = 104h).

3.8.2.2 ADDA and SUBA Instructions

As with linear address arithmetic, *offsetR/cst* is shifted left by 3, 2, 1, or 0 according to the data size, and is then added to or subtracted from *baseR* to produce the final address. Circular addressing modifies this slightly by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N + 1 unchanged after address arithmetic. The resulting address is bounded to $2^{(N + 1)}$ range, regardless of the size of the *offsetR/cst*.

The circular buffer size in AMR is not scaled; for example, a block size of 8 is 8 bytes, not 8 times the data size (byte, halfword, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. Example 3-5 shows an **ADDAH** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value in AMR for this example is 0004 0001h.

Example 3-5. ADDAH Instruction in Circular Mode

```
ADDAH       .D1         A4,A1,A4
```

	Before ADDAH		1 cycle after ADDAH (1)
A4	0000 0100h	A4	0000 0106h
A1	0000 0013h	A1	0000 0013h

(1) **Note:** 13h halfwords is 26h bytes. 26h bytes is 6 bytes beyond the 32-byte (20h) boundary 100h-11Fh; thus, it is wrapped around to (126h - 20h = 106h).

3.8.2.3 Circular Addressing Considerations with Nonaligned Memory

Circular addressing may be used with nonaligned accesses. When circular addressing is enabled, address updates and memory accesses occur in the same manner as for the equivalent sequence of byte accesses.

On the C64x CPU, the only restriction is that the circular buffer size be at least as large as the data size being accessed. Nonaligned access to circular buffers that are smaller than the data being read will cause undefined results.

On the C64x+ CPU, the circular buffer size must be at least 32 bytes. Nonaligned access to circular buffers that are smaller than 32 bytes will cause undefined results.

Nonaligned accesses to a circular buffer apply the circular addressing calculation to *logically adjacent* memory addresses. The result is that nonaligned accesses near the boundary of a circular buffer will correctly read data from both ends of the circular buffer, thus seamlessly causing the circular buffer to "wrap around" at the edges.

Consider, for example, a circular buffer size of 16 bytes. A circular buffer of this size at location 20h, would look like this in physical memory:

1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3
7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8
x	x	x	x	x	x	x	x	x	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	x	x	x	x	x	x	x	x	x

The effect of circular buffering is to make it so that memory accesses and address updates in the 20h-2Fh range stay completely inside this range. Effectively, the memory map behaves in this manner:

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8
h	i	j	k	l	m	n	o	p	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	a	b	c	d	e	f	g	h	i

Example 3-6 shows an **LDNW** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value in AMR for this example is 0004 0001h. The buffer starts at address 0020h and ends at 0040h. The register A4 is initialized to the address 003Ah.

Example 3-6. LDNW in Circular Mode

```
LDNW    .D1        *++A4[2],A1
```

	Before LDNW		1 cycle after LDNW [1]		5 cycles after LDNW
A4	0000 003Ah	A4	0000 0022h	A4	0000 0022h
A1	xxxx xxxxh	A1	xxxx xxxxh	A1	5678 9ABCh
mem 0022h	5678 9ABCh	mem 0022h	5678 9ABCh	mem 0022h	5678 9ABCh

(1) **Note:** 2h words is 8h bytes. 8h bytes is 2 bytes beyond the 32-byte (20h) boundary starting at address 003Ah; thus, it is wrapped around to 0022h (003Ah + 8h = 0022h).

3.8.3 Syntax for Load/Store Address Generation

The DSP has a load/store architecture, which means that the only way to access data in memory is with a load or store instruction. Table 3-5 shows the syntax of an indirect address to a memory location. Sometimes a large offset is required for a load/store. In this case, you can use the B14 or B15 register as the base register, and use a 15-bit constant (*ucst15*) as the offset.

Table 3-6 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

Table 3-5. Indirect Address Generation for Load/Store

Addressing Type	No Modification of Address Register	Preincrement or Predecrement of Address Register	Postincrement or Postdecrement of Address Register
Register indirect	*R	*++R	*R++
		*- -R	*R- -
Register relative	*+R[*ucst5*]	*++R[*ucst5*]	*R++[*ucst5*]
	*-R[*ucst5*]	*- -R[*ucst5*]	*R- -[*ucst5*]
Register relative with 15-bit constant offset	*+B14/B15[*ucst15*]	not supported	not supported
Base + index	*+R[*offsetR*]	*++R[*offsetR*]	*R++[*offsetR*]
	*-R[*offsetR*]	*- -R[*offsetR*]	*R- -[*offsetR*]

Table 3-6. Address Generator Options for Load/Store

Mode Field				Syntax	Modification Performed
0	0	0	0	*-R[*ucst5*]	Negative offset
0	0	0	1	*+R[*ucst5*]	Positive offset
0	1	0	0	*-R[*offsetR*]	Negative offset
0	1	0	1	*+R[*offsetR*]	Positive offset
1	0	0	0	*- -R[*ucst5*]	Predecrement
1	0	0	1	*++R[*ucst5*]	Preincrement
1	0	1	0	*R- -[*ucst5*]	Postdecrement
1	0	1	1	*R++[*ucst5*]	Postincrement
1	1	0	0	*--R[*offsetR*]	Predecrement
1	1	0	1	*++R[*offsetR*]	Preincrement
1	1	1	0	*R- -[*offsetR*]	Postdecrement
1	1	1	1	*R++[*offsetR*]	Postincrement

3.9 Compact Instructions on the C64x+ CPU

The C64x+ CPU supports a header based set of 16-bit-wide compact instructions in addition to the normal 32-bit wide instructions. The C64x CPU does not support compact instructions.

3.9.1 Compact Instruction Overview

The availability of compact instructions is enabled by the replacement of the eighth word of a fetch packet with a 32-bit header word. The header word describes which of the other seven words of the fetch packet contain compact instructions, which of the compact instructions in the fetch packet operate in parallel, and also contains some decoding information which supplements the information contained in the 16-bit compact opcode. Table 3-7 compares the standard fetch packet with a header-based fetch packet containing compact instructions.

Table 3-7. C64x+ CPU Fetch Packet Types

Standard C6000 Fetch Packet	
Word	
0	32-bit opcode
1	32-bit opcode
2	32-bit opcode
3	32-bit opcode
4	32-bit opcode
5	32-bit opcode
6	32-bit opcode
7	32-bit opcode

Header-Based Fetch Packet		
Word		
0	16-bit opcode	16-bit opcode
1	32-bit opcode	
2	16-bit opcode	16-bit opcode
3	32-bit opcode	
4	16-bit opcode	16-bit opcode
5	32-bit opcode	
6	16-bit opcode	16-bit opcode
7	Header	

Within the other seven words of the fetch packet, each word may be composed of a single 32-bit opcode or two 16-bit opcodes. The header word specifies which words contain compact opcodes and which contain 32-bit opcodes.

The compiler will automatically code instructions as 16-bit compact instructions when possible.

There are a number of restrictions to the use of compact instructions:

- No dedicated predication field
- 3-bit register address field
- Very limited 3 operand instructions
- Subset of 32-bit instructions

3.9.2 Header Word Format

Figure 3-3 describes the format of the compact instruction header word.

Figure 3-3. Compact Instruction Header Format

31	30	29	28	27 – 21	20 – 14	13 – 0
1	1	1	0	Layout	Expansion	p-bits
				7	7	14

Bits 27-21 (Layout field) indicate which words in the fetch packet contain 32-bit opcodes and which words contain two 16-bit opcodes.

Bits 20-14 (Expansion field) contain information that contributes to the decoding of all compact instructions in the fetch packet.

Bits 13-0 (p-bits field) specify which compact instructions are run in parallel.

3.9.2.1 Layout Field in Compact Header Word

Bits 27-21 of the compact instruction header contains the layout field. This field specifies which of the other seven words in the current fetch packet contain 32-bit full-sized instructions and which words contain two 16-bit compact instructions.

Figure 3-4 shows the layout field in the compact header word and Table 3-8 describes the bits.

Figure 3-4. Layout Field in Compact Header Word

27	26	25	24	23	22	21
L7	L6	L5	L4	L3	L2	L1

Table 3-8. Layout Field Description in Compact Instruction Packet Header

Bit	Field	Value	Description
27	L7	0	Seventh word of fetch packet contains a single 32-bit opcode.
		1	Seventh word of fetch packet contains two 16-bit compact instructions.
26	L6	0	Sixth word of fetch packet contains a single 32-bit opcode.
		1	Sixth word of fetch packet contains two 16-bit compact instructions.
25	L5	0	Fifth word of fetch packet contains a single 32-bit opcode.
		1	Fifth word of fetch packet contains two 16-bit compact instructions.
24	L4	0	Fourth word of fetch packet contains a single 32-bit opcode.
		1	Fourth word of fetch packet contains two 16-bit compact instructions.
23	L3	0	Third word of fetch packet contains a single 32-bit opcode.
		1	Third word of fetch packet contains two 16-bit compact instructions.
22	L2	0	Second word of fetch packet contains a single 32-bit opcode.
		1	Second word of fetch packet contains two 16-bit compact instructions.
21	L1	0	First word of fetch packet contains a single 32-bit opcode.
		1	First word of fetch packet contains two 16-bit compact instructions.

3.9.2.2 Expansion Field in Compact Header Word

Bits 20-14 of the compact instruction header contains the opcode expansion field. This field specifies properties that apply to all compact instructions contained in the current fetch packet.

Figure 3-5 shows the expansion field in the compact header word and Table 3-9 describes the bits.

Figure 3-5. Expansion Field in Compact Header Word

20	19	18 16	15	14
PROT	RS	DSZ	BR	SAT

Table 3-9. Expansion Field Description in Compact Instruction Packet Header

Bit	Field	Value	Description
20	PROT	0	Loads are nonprotected (NOPs must be explicit).
		1	Loads are protected (4 NOP cycles added after every LD instruction).
19	RS	0	Instructions use low register set for data source and destination.
		1	Instructions use high register set for data source and destination.
18-16	DSZ	0-7h	Defines primary and secondary data size (see Table 3-10)
15	BR	0	Compact instructions in the S unit are not decoded as branches
		1	Compact Instructions in the S unit are decoded as branches.
14	SAT	0	Compact instructions do not saturate.
		1	Compact instructions saturate.

Bit 20 (PROT) selects between protected and nonprotected mode for all **LD** instructions within the fetch packet. When PROT is 1, four cycles of NOP are added after each **LD** instruction within the fetch packet whether the **LD** is in 16-bit compact format or 32-bit format.

Bit 19 (RS) specifies which register set is used by compact instructions within the fetch packet. The register set defines which subset of 8 registers on each side are data registers. The 3-bit register field in the compact opcode indicates which one of eight registers is used. When RS is 1, the high register set (A16-A23 and B16-B23) is used; when RS is 0, the low register set (A0-A7 and B0-B7) is used.

Bits 18-16 (DSZ) determine the two data sizes available to the compact versions of the **LD** and **ST** instructions in a fetch packet. Bit 18 determines the primary data size that is either word (W) or doubleword (DW). In the case of DW, an opcode bit selects between aligned (DW) and nonaligned (NDW) accesses. Bits 17 and 16 determine the secondary data size: byte unsigned (BU), byte (B), halfword unsigned (HU), halfword (H), word (W), or nonaligned word (NW). Table 3-10 describes how the bits map to data size.

Bit 15 (BR). When BR is 1, instructions in the S unit are decoded as branches.

Bit 14 (SAT). When SAT is 1, the **ADD, SUB, SHL, MPY, MPYH, MPYLH,** and **MPYHL** instructions are decoded as **SADD, SUBS, SSHL, SMPY, SMPYH, SMPYLH,** and **SMPYHL**, respectively.

Table 3-10. LD/ST Data Size Selection

DSZ Bits			Primary Data Size [1]	Secondary Data Size [2]
18	17	16		
0	0	0	W	BU
0	0	1	W	B
0	1	0	W	HU
0	1	1	W	H
1	0	0	DW/NDW	W
1	0	1	DW/NDW	B
1	1	0	DW/NDW	NW
1	1	1	DW/NDW	H

(1) Primary data size is word W) or doubleword (DW). In the case of DW, aligned (DW) or nonaligned (NDW).
(2) Secondary data size is byte unsigned (BU), byte (B), halfword unsigned (HU), halfword (H), word (W), or nonaligned word (NW).

3.9.2.3 P-bit Field in Compact Header Word

Unlike normal 32-bit instructions in which the *p*-bit filed in each opcode determines whether the instruction executes in parallel with other instructions; the parallel/nonparallel execution information for compact instructions is contained in the compact instruction header word.

Bits 13-0 of the compact instruction header contain the *p*-bit field. This field specifies which of the compact instructions within the current fetch packet are executed in parallel. If the corresponding bit in the layout field is 0 (indicating that the word is a noncompact instruction), then the bit in the *p*-bit field must be zero; that is, 32-bit instructions within compact fetch packets use their own *p*-bit field internal to the 32-bit opcode; therefore, the associated *p*-bit field in the header should always be zero.

Figure 3-6 shows the *p*-bits field in the compact header word and Table 3-11 describes the bits.

Figure 3-6. P-bits Field in Compact Header Word

13	12	11	10	9	8	7	6	5	4	3	2	1	0
P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Table 3-11. P-bits Field Description in Compact Instruction Packet Header

Bit	Field	Value	Description
13	P13	0	Word 6 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 6 (16 most-significant bits) of fetch packet has parallel bit set.
12	P12	0	Word 6 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 6 (16 least-significant bits) of fetch packet has parallel bit set.
11	P11	0	Word 5 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 5 (16 most-significant bits) of fetch packet has parallel bit set.
10	P10	0	Word 5 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 5 (16 least-significant bits) of fetch packet has parallel bit set.
9	P9	0	Word 4 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 4 (16 most-significant bits) of fetch packet has parallel bit set.
8	P8	0	Word 4 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 4 (16 least-significant bits) of fetch packet has parallel bit set.
7	P7	0	Word 3 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 3 (16 most-significant bits) of fetch packet has parallel bit set.
6	P6	0	Word 3 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 3 (16 least-significant bits) of fetch packet has parallel bit set.
5	P5	0	Word 2 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 2 (16 most-significant bits) of fetch packet has parallel bit set.
4	P4	0	Word 2 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 2 (16 least-significant bits) of fetch packet has parallel bit set.
3	P3	0	Word 1 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 1 (16 most-significant bits) of fetch packet has parallel bit set.
2	P2	0	Word 1 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 1 (16 least-significant bits) of fetch packet has parallel bit set.
1	P1	0	Word 0 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 0 (16 most-significant bits) of fetch packet has parallel bit set.
0	P0	0	Word 0 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 0 (16 least-significant bits) of fetch packet has parallel bit set.

3.9.3 Processing of Fetch Packets

The header information is used to fully define the 32-bit version of the 16-bit instructions. In the case where an execute packet crosses fetch packet boundaries, there are two headers in use simultaneously. Each instruction uses the header information from its fetch packet header.

3.9.4 Execute Packet Restrictions

Execute packets that span fetch packet boundaries may not be the target of branches in the case where one of the two fetch packets involved are header-based. The only exception to this is where an interrupt is taken in the cycle before a spanning execute packet reaches E1. The target of the return may be a normally disallowed target.

If the execute packet contains eight instructions, then neither of the two fetch packets may be header-based.

3.9.5 Available Compact Instructions

Table 3-12 lists the available compact instructions and their functional unit.

Table 3-12. Available Compact Instructions

Instruction	L Unit	M Unit	S Unit	D Unit
ADD	✓		✓	✓
ADDAW				✓
ADDK			✓	
AND	✓			
BNOP displacement			✓	
CALLP			✓	
CLR			✓	
CMPEQ	✓			
CMPGT	✓			
CMPGTU	✓			
CMPLT	✓			
CMPLTU	✓			
EXT			✓	
EXTU			✓	
LDB				✓
LDBU				✓
LDDW				✓
LDH				✓
LDHU				✓
LDNDW				✓
LDNW				✓
LDW				✓
LDW (15-bit offset)				✓
MPY		✓		
MPYH		✓		
MPYHL		✓		
MPYLH		✓		
MV	✓		✓	✓
MVC			✓	
MVK	✓		✓	✓
NEG	✓			

Table 3-12. Available Compact Instructions (continued)

Instruction	L Unit	M Unit	S Unit	D Unit
NOP		No unit		
OR	✓			
SADD	✓		✓	
SET			✓	
SHL			✓	
SHR			✓	
SHRU			✓	
SMPY		✓		
SMPYH		✓		
SMPYHL		✓		
SMPYLH		✓		
SPKERNEL		No unit		
SPLOOP		No unit		
SPLOOPD		No unit		
SPMASK		No unit		
SPMASKR		No unit		
SSHL			✓	
SSUB	✓			
STB				✓
STDW				✓
STH				✓
STNDW				✓
STNW				✓
STW				✓
STW (15-bit offset)				✓
SUB	✓		✓	✓
SUBAW				✓
XOR	✓			

3.10 Instruction Compatibility

The C62x, C64x, and C64x+ DSPs share an instruction set. All of the instructions valid for the C62x DSP are also valid for the C64x and C64x+ DSPs. The C64x/C64x+ DSP adds functionality to the C62x DSP with some unique instructions. See Appendix A for a list of the instructions that are common to the C62x, C64x, and C64x+ DSPs.

3.11 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Assembler syntax
- Functional units
- Compatibility
- Operands
- Opcode
- Description
- Execution
- Pipeline
- Instruction type
- Delay slots
- Functional Unit Latency
- Examples

The **ADD** instruction is used as an example to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information.