National Semiconductor

February 1992

NS32FX161-15/NS32FX161-20/NS32FX164-20/ NS32FX164-25/NS32FV16-20/NS32FV16-25 Advanced Imaging/Communication Signal Processors

NS32FX161-15/NS32FX161-20/NS32FX164-20/NS32FX164-25/NS32FV16-20/NS32FV16-25
Advanced Imaging/Communication Signal Processors

General Description

The NS32FX164, the NS32FV16 and the NS32FX161 are high-performance 32-bit members of the Series 32000®/EP™ family of National's Embedded System Processors™ specifically optimized for CCITT Group 2 and Group 3 Facsimile Applications, Data Modems, Voice Mail Systems, Laser Printers, or any combination of the above.

Unless specified otherwise any reference to the NS32FX164 in this document applies to the NS32FV16 and the NS32FX161 as well.

The NS32FX164 can perform all the computations and control functions required for a stand-alone Fax system, a PC add-in Fax/Voice/Data Modem card or a Laser/Fax system.

It also meets the performance requirements to implement 14400, 9600 and 7200 bps modems complying with CCITT V.17, V.29 and V.27 standards. The NS32FV16 supports V.29 and V.27 standards as well as voice. The NS32FX161 supports V.29 and V.27 standards.

The NS32FX164 provides a 16 Mbyte Linear external address space and a 16-bit external data bus.

The CPU core, which is the same as that of the NS32CG16, incorporates a 32-bit ALU and instruction pipeline, and an 8-byte prefetch queue.

Also integrated on-chip with the CPU are a DSP Module (DSPM) and a 4K-byte RAM Array (2K in the NS32FV16 and NS32FX161). The DSPM is a complete processing unit, capable of autonomous operation parallel to the CPU core operation. The DSPM executes programs stored in an internal on-chip Random Access Memory (RAM), and manipulates data stored either in the internal RAM or in an external off-chip memory. To maximize utilization of hardware resources, the DSPM contains a pipelined DSP-oriented datapath, and a control logic that implements a set of DSP vector commands.

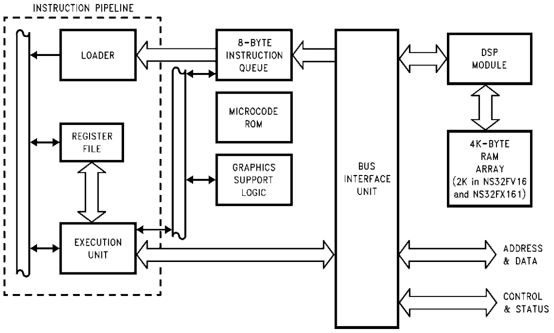
The NS32FX164 capabilities can be expanded by using an external floating point unit (FPU) which directly interfaces to the NS32FX164 using the slave protocol. The CPU-FPU cluster features high speed execution of the floating-point instructions.

The NS32FX164 highly-efficient architecture combined with the NS32CG16 graphics instructions and the high-performance vector operation capability, makes the device the ideal choice for Postscript™ and Fax applications.

Features

- Software compatible with the Series 32000/EP processors
- Designed around the CPU core of the NS32CG16
- Pin compatible with the NS32FX16
- 32-bit architecture and implementation
- On-chip DSP Module for high-speed DSP operations
- Special support for graphics applications
  - — 18 graphics instructions
  - — Binary compression/expansion capability for font storage using RLL encoding
  - — Pattern magnification
  - — Interface to an external BITBLT processing units for fast color BITBLT operations
- 4K-byte on-chip RAM array (2K in NS32FV16 and NS32FX161)
- On-chip clock generator
- Floating-point support via the NS32081 or NS32181
- Optimal interface to large memory arrays via the NS32CG821 and the DP84xx family of DRAM controllers
- Power save mode
- High-speed CMOS technology
- 68-pin PLCC package

Block Diagram



TL/EE/11267–1

FIGURE 1-1. CPU Block Diagram

Series 32000® is a registered trademark of National Semiconductor Corporation.
EP™ and Embedded System Processors™ are trademarks of National Semiconductor Corporation.
Postscript™ is a trademark of Adobe Systems, Inc.

RRD-B30M115/Printed in U. S. A.

Table of Contents

Table of Contents (Continued)

List of Figures

List of Figures (Continued)

List of Figures (Continued)

List of Tables

1.0 Product Introduction

The NS32FX164 is a high speed CMOS microprocessor in the Series 32000/EP family.

It includes two main execution units: the NS32CG16 compatible CPU core and the DSP Module. The CPU core is designed for general purpose computations and system control functions. The DSP Module is tuned to perform the DSP primitives needed in Voice Band Modems. The NS32FX164 also incorporates a 4K-byte RAM Array as a shared resource for both the CPU core and the DSP Module (2K-byte in the NS32FV16 and the NS32FX161).

The NS32FX164 is software-compatible with all other CPUs in the family.

The device incorporates all of the Series 32000 advanced architectural features, with the exception of the virtual memory capability.

Brief descriptions of the NS32FX164 features that are shared with other members of the family are provided below:

Powerful Addressing Modes. Nine addressing modes available to all instructions are included to access data structures efficiently.

Data Types. The architecture provides for numerous data types, such as byte, word, doubleword, and BCD, which may be arranged into a wide variety of data structures.

Symmetric Instruction Set. While avoiding special case instructions that compilers can't use, the Series 32000 family incorporates powerful instructions for control operations, such as array indexing and external procedure calls, which save considerable space and time for compiled code.

Memory-to-Memory Operations. The Series 32000 CPUs represent two-address machines. This means that each operand can be referenced by any one of the addressing modes provided.

This powerful memory-to-memory architecture permits memory locations to be treated as registers for all useful operations. This is important for temporary operands as well as for context switching.

Large, Uniform Addressing. The NS32FX164 has 24-bit address pointers that can address up to 16 megabytes without any segmentation; this addressing scheme provides flexible memory management without add-on expense.

Modular Software Support. Any software package for the Series 32000 architecture can be developed independent of all other packages, without regard to individual addressing. In addition, ROM code is totally relocatable and easy to access, which allows a significant reduction in hardware and software cost.

Software Processor Concept. The Series 32000 architecture allows future expansions of the instruction set that can be executed by special slave processors, acting as extensions to the CPU. This concept of slave processors is unique to the Series 32000 architecture. It allows software compatibility even for future components because the slave hardware is transparent to the software. With future advances in semiconductor technology, the slaves can be physically integrated on the CPU chip itself.

To summarize, the architectural features cited above provide three primary performance advantages and characteristics:

- High-Level Language Support
- Easy Future Growth Path
- Application Flexibility

1.1 NS32FX164 SPECIAL FEATURES

In addition to the above Series 32000 features, the NS32FX164 provides features that make the device extremely attractive for a wide range of applications where graphics support, low chip count, and low power consumption are required.

The most relevant of these features are the enhanced Digital Signal Processing performance which makes the chip very attractive for facsimile applications, and the graphics support capabilities, that can be used in applications such as printers, CRT terminals, and other varieties of display systems, where text and graphics are to be handled.

Graphics support is provided by eighteen instructions that allow operations such as BITBLT, data compression/expansion, fills, and line drawing, to be performed very efficiently. In addition, the device can be easily interfaced to an external BITBLT Processing Unit (BPU) for high BITBLT performance.

The NS32FX164 allows systems to be built with a relatively small amount of random logic. The bus is highly optimized to allow simple interfacing to a large variety of DRAMs and peripheral devices. All the relevant bus access signals and clock signals are generated on-chip. The cycle extension logic is also incorporated on-chip.

The device is fabricated in a low-power, high speed CMOS technology. It also includes a power-save feature that allows the clock to be slowed down under software control, thus minimizing the power consumption. This feature can be used in those applications where power saving during periods of low performance demand is highly desirable.

The power save feature, the DSP Module and the Bus Characteristics are described in the ‘‘Functional Description’’ section. A general overview of BITBLT operations and a description of the graphics support instructions is provided in Section 2.5. Details on all the NS32FX164 graphics instructions can be found in the NS32CG16 Printer/Display Processor Programmer's Reference Supplement.

1.0 Product Introduction (Continued)

Below is a summary of the instructions that are directly applicable to graphics along with their intended use.

Instruction	Application
BBAND BBOR BBFOR BBXOR BBSTOD BITWT EXTBLT	The BITBLT group of instructions provide a method of quickly imaging characters, creating patterns, windowing and other block oriented effects.
MOVMP	Move Multiple Pattern is a very fast instruction for clearing memory and drawing patterns and lines.
TBITS	Test Bit String will measure the length of 1's or 0's in an image, supporting many data compression methods (RLL), TBITS may also be used to test for boundaries of images.
SBITS	Set Bit String is a very fast instruction for filling objects, outline characters and drawing horizontal lines. The TBITS and SBITS instructions support Group 3 and Group 4 CCITT standards for compression and decompression algorithms.
SBITPS	Set Bit Perpendicular String is a very fast instruction for drawing vertical, horizontal and 45° lines. In printing applications SBITS and SBITPS may be used to express portrait and landscape respectively from the same compressed font data. The size of the character may be scaled as it is drawn.
SBIT CBIT TBIT IBIT	The Bit group of instructions enable single pixels anywhere in memory to be set, cleared, tested or inverted.
INDEX	The INDEX instruction combines a multiply-add sequence into a single instruction. This provides a fast translation of an X-Y address to a pixel relative address.

2.0 Architectural Description

2.1 REGISTER SET

The NS32FX164 has 32 internal registers. 17 of these registers belong to the CPU portion of the device and are addressed either implicitly by specific instructions or through the register addressing mode. The other 15 control the operation of the DSP Module, and are memory mapped. *Figure 2-1* shows the NS32FX164 internal registers.

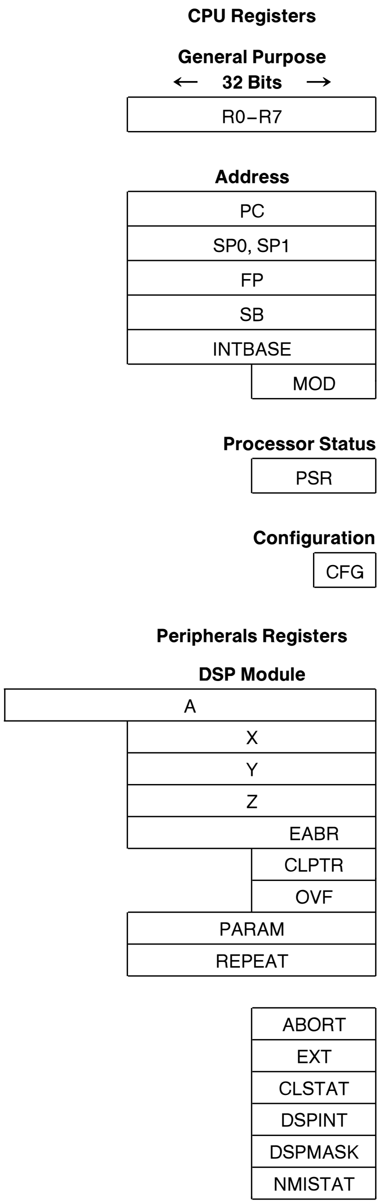


FIGURE 2-1. NS32FX164 Internal Registers

2.1.1 General Purpose Registers

There are eight registers (R0–R7) used for satisfying the high speed general storage requirements, such as holding temporary variables and addresses. The general purpose registers are free for any use by the programmer. They are 32 bits in length. If a general purpose register is specified for an operand that is 8 or 16 bits long, only the low part of the register is used; the high part is not referenced or modified.

2.0 Architectural Description (Continued)

2.1.2 Address Registers

The seven address registers are used by the processor to implement specific address functions. Except for the MOD register that is 16 bits wide, all the others are 32 bits. A description of the address registers follows.

PC—Program Counter. The PC register is a pointer to the first byte of the instruction currently being executed. The PC is used to reference memory in the program section.

SP0, SP1—Stack Pointers. The SP0 register points to the lowest address of the last item stored on the INTERRUPT STACK. This stack is normally used only by the operating system. It is used primarily for storing temporary data, and holding return information for operating system subroutines and interrupt and trap service routines. The SP1 register points to the lowest address of the last item stored on the USER STACK. This stack is used by normal user programs to hold temporary data and subroutine return information.

When a reference is made to the selected Stack Pointer (see PSR S-bit), the terms "SP Register" or "SP" are used. SP refers to either SP0 or SP1, depending on the setting of the S bit in the PSR register. If the S bit in the PSR is 0, SP refers to SP0. If the S bit in the PSR is 1 then SP refers to SP1.

Stacks in the Series 32000 architecture grow downward in memory. A Push operation pre-decrements the Stack Pointer by the operand length. A Pop operation post-increments the Stack Pointer by the operand length.

FP—Frame Pointer. The FP register is used by a procedure to access parameters and local variables on the stack. The FP register is set up on procedure entry with the ENTER instruction and restored on procedure termination with the EXIT instruction.

The frame pointer holds the address in memory occupied by the old contents of the frame pointer.

SB—Static Base. The SB register points to the global variables of a software module. This register is used to support relocatable global variables for software modules. The SB register holds the lowest address in memory occupied by the global variables of a module.

INTBASE—Interrupt Base. The INTBASE register holds the address of the dispatch table for interrupts and traps (Section 3.2.1).

MOD—Module. The MOD register holds the address of the module descriptor of the currently executing software module. The MOD register is 16 bits long, therefore the module table must be contained within the first 64 kbytes of memory.

2.1.3 Processor Status Register

The Processor Status Register (PSR) holds status information for the microprocessor.

The PSR is sixteen bits long, divided into two eight-bit halves. The low order eight bits are accessible to all programs, but the high order eight bits are accessible only to programs executing in Supervisor Mode.

15							8	7							0
B				I	P	S	U	N	Z	F	J	K	L	T	C

FIGURE 2-2. Processor Status Register (PSR)

C The C bit indicates that a carry or borrow occurred after an addition or subtraction instruction. It can be used with the ADDC and SUBC instructions to perform multiple-precision integer arithmetic calculations. It may have a setting of 0 (no carry or borrow) or 1 (carry or borrow).

T The T bit causes program tracing. If this bit is set to 1, a TRC trap is executed after every instruction (Section 3.3.1).

L The L bit is altered by comparison instructions. In a comparison instruction the L bit is set to "1" if the second operand is less than the first operand, when both operands are interpreted as unsigned integers. Otherwise, it is set to "0". In Floating-Point comparisons, this bit is always cleared.

K Reserved for use by the CPU.

J Reserved for use by the CPU.

F The F bit is a general condition flag, which is altered by many instructions (e.g., integer arithmetic instructions use it to indicate overflow).

Z The Z bit is altered by comparison instructions. In a comparison instruction the Z bit is set to "1" if the second operand is equal to the first operand; otherwise it is set to "0".

N The N bit is altered by comparison instructions. In a comparison instruction the N bit is set to "1" if the second operand is less than the first operand, when both operands are interpreted as signed integers. Otherwise, it is set to "0".

U If the U bit is "1" no privileged instructions may be executed. If the U bit is "0" then all instructions may be executed. When U=0 the processor is said to be in Supervisor Mode; when U=1 the processor is said to be in User Mode. A User Mode program is restricted from executing certain instructions and accessing certain registers which could interfere with the operating system. For example, a User Mode program is prevented from changing the setting of the flag used to indicate its own privilege mode. A Supervisor Mode program is assumed to be a trusted part of the operating system, hence it has no such restrictions.

S The S bit specifies whether the SP0 register or SP1 register is used as the Stack Pointer. The bit is automatically cleared on interrupts and traps. It may have a setting of 0 (use the SP0 register) or 1 (use the SP1 register).

P The P bit prevents a TRC trap from occurring more than once for an instruction (Section 3.3.1). It may have a setting of 0 (no trace pending) or 1 (trace pending).

I If I=1, then all interrupts will be accepted. If I=0, only the NMI interrupt is accepted. Trap enables are not affected by this bit.

B Reserved for use by the CPU. This bit is set to 1 during the execution of the EXTBLT instruction and causes the $\overline{\text{BPU}}$ signal to become active. Upon reset, B is set to zero and the $\overline{\text{BPU}}$ signal is set high.

Note 1: When an interrupt is acknowledged, the B, I, P, S and U bits are set to zero and the $\overline{\text{BPU}}$ signal is set high. A return from interrupt will restore the original values from the copy of the PSR register saved in the interrupt stack.

Note 2: If BITBLT (BB) or EXTBLT instructions are executed in an interrupt routine, the PSR bits J and K must be cleared first.

2.1.4 Configuration Register

The Configuration Register (CFG) is 32 bits wide, of which 5 bits are implemented. The implemented bits enable various operating modes for the CPU, including vectoring of interrupts, execution of floating-point instructions, processing of exceptions and selection of clock scaling factor. The CFG is programmed by the SETCFG instruction. The format of CFG is shown in *Figure 2-3*. The various control bits are described below.

31	8	7				0
Reserved	DE	Res	C	M	F	I

FIGURE 2-3. Configuration Register (CFG)

I Interrupt vectoring. This bit controls whether maskable interrupts are handled in nonvectored (I=0) or vectored (I=1) mode. Refer to Section 3.2.3 for more information.

F Floating-point instruction set. This bit indicates whether a floating-point unit (FPU) is present to execute floating-point instructions. If this bit is 0 when the CPU executes a floating-point instruction, a Trap (UND) occurs. If this bit is 1, then the CPU transfers the instruction and any necessary operands to the FPU using the slave-processor protocol described in Section 3.1.3.1.

M Clock scaling. This bit is used in conjunction with the C-bit to select the clock scaling factor.

C Clock scaling. Same as the M-bit above. Refer to Section 3.5.3 on ‘‘Power Save Mode’’ for details.

DE Direct-Exception mode enable. This bit enables the Direct-Exception mode for processing exceptions. When this mode is selected, the CPU response time to interrupts and other exceptions is significantly improved. Refer to Section 3.2 for more information.

2.1.5 DSP Module Registers

The DSP Module (DSPM) contains 15 memory-mapped registers. All the registers, except OVF, CLSTAT, ABORT, DSPINT and NMISTAT, are readable and writable. OVF, CLSTAT, DSPINT and NMISTAT are read-only. ABORT is write-only.

The DSPM registers are divided into two groups, according to their function. PARAM, OVF, X, Y, Z, A, REPEAT, CLPTR and EABR are called DSPM dedicated registers. CLSTAT, ABORT, DSPINT, DSPMASK, EXT and NMISTAT are called CPU core interface registers.

Accesses to these registers must be aligned; word and double-word accesses must occur on word and double-word address boundaries respectively. Failing to do so will cause unpredictable results. *Figure 2-4* shows the address map of the DSP Module registers.

Register Name	Register Address
PARAM	FFFF8000
OVF	FFFF8004
X	FFFF8008
Y	FFFF800C
Z	FFFF8010
A	FFFF8014
REPEAT	FFFF8018
CLPTR	FFFF8020
EABR	FFFF8024
CLSTAT	FFFF9000
ABORT	FFFF9004
DSPINT	FFFF9008
DSPMASK	FFFF900C
EXT	FFFF9010
NMISTAT	FFFF9014

FIGURE 2-4. DSP Module Registers Address Map

A—Accumulator

The format of the accumulator is shown in *Figure 2-5*.

33 0	33 0
Imaginary	Real

FIGURE 2-5. Accumulator Format

The A register is a complex accumulator. It has two 34-bit fields: a real part, and an imaginary part. Bits 15 through 30 of the real and the imaginary parts of the accumulator can be read or written by the core in one double-word access. Bits 15 through 30 of the real part are mapped to the operand's bits 0 through 15, and bits 15 through 30 of the imaginary part are mapped to the operand's bits 16 through 31. The accumulator can also be read and written by the command-list execution unit using the SA, SEA, LA and LEA instructions (See Section 3.4 for more information).

Note that when a value is stored in the accumulator by the core, the value of PARAM.RND bit is copied into bit position 14 of both real and imaginary parts of the accumulator. This technique allows rounding of the accumulator's value in the following DSPM instructions (See Section 3.4.5.3 for more information on rounding).

When the Accumulator is loaded either by the core or by the LA or LEA instructions, bits 31–33 of the real and the imaginary accumulators are loaded with the values of bit 30 of the real and the imaginary parts respectively.

When the Accumulator is loaded either by the core or by the LA instruction, bits 0–13 of the real and the imaginary accumulators are loaded with zeros.

X, Y, Z—Vector Pointers

The format of X, Y, and Z registers is shown in *Figure 2-6*.

31 16	15 8	7 4	3 0
ADDRESS	Reserved	WRAP-AROUND	INCREMENT

FIGURE 2-6. X, Y, Z Registers Format

2.0 Architectural Description (Continued)

The X, Y, and Z registers are used for addressing up to three vector operands. They are 32-bit registers, with three fields: ADDRESS, INCREMENT, and WRAP-AROUND. The value in the ADDRESS field specifies the address of a word in the on-chip memory. This field has 16 bits, and can address up to 64 Kwords of internal memory. The ADDRESS fields are initialized with the vector operands' start-addresses by commands in the command list. At the beginning of each vector operation, the contents of the ADDRESS field are copied to incrementors. Increments can be used by vector instructions to step through the corresponding vector operands while executing the appropriate calculations. There is an address wrap-around for those vector instructions that require some of their operands to be located in cyclic buffers. The allowed values for the increment field are 0 through 15. The actual increment will be $2^{\text{increment}}$ words. The allowed values for the WRAP-AROUND field are 0 through 15. The actual wrap-around will be $2^{\text{WRAP-AROUND}}$ words. The WRAP-AROUND must be greater or equal to the INCREMENT.

The X, Y, and Z registers can be read and written by the core. These registers can be read and written by the command-list execution unit, as well as by the core, when using SX, SXL, SXH, SY, SZ, LX, LY and LZ instructions.

EABR—External Address Base Register

The format of the external address base register is shown in *Figure 2-7*.

31 17	16 0
ADDRESS	0

FIGURE 2-7. EABR Register Format

The EABR register is used together with a 16-bit address field to form a 32-bit external address. External addresses are specified as the sum of the value in EABR and two times the value of the 16-bit address pointed by registers X, Y or Z. The only value allowed to be written into bits 0 through 16 of EABR is ''0''. The EABR register can be read and written by the core. It can also be written by the command-list execution unit by using the LEABR instruction.

EABR can hold any value except for FFFE0000. Accessing external memory with an FFFE0000 in the EABR will cause unpredictable results.

CLPTR—Command List Pointer

The CLPTR is a 16-bit register that holds the address of the current command in the internal RAM. Writing into the CLPTR causes the DSPM command-list execution unit to begin executing commands, starting from the address in CLPTR. The CLPTR can be read and written by the core while the command-list execution is idle.

Whenever the DSPM command-list execution unit reads a command from the DSPM RAM, the value of CLPTR is updated to contain the address of the next command to be executed. This implies, for example, that if the last command in a list is in address N, the CLPTR will hold a value of N + 1 following the end of command list execution.

OVF—Overflow Register

The format of the overflow register is shown in *Figure 2-8*.

15 2	1	0
Reserved	OVF	SAT

FIGURE 2-8. OVF Register Format

The OVF register holds the current status of the DSPM arithmetic unit. It has two fields: OVF and SAT. The OVF bit is set to ''1'' whenever an overflow is detected in the DSPM 34-bit ALU (e.g., bits 32 and 33 of the ALU are not equal). No overflow detection is provided for integers. The SAT bit is set to ''1'' whenever a value read from the accumulator cannot be represented within the limits of its data type (e.g., 16 bits for real and integer, and 31 bits for extended real). In this case the value read from the accumulator will either be the maximum allowed value or the minimal allowed value for this data type depending on the sign of the accumulator value. Note that in some cases when the OVF is set, the SAT will not be set. The reason is that if an OVF occurred, the value in the accumulator can no longer be used for proper SAT detection. Upon reset, and whenever the ABORT register is written, the non reserved bits of the OVF register is cleared to ''0''.

The OVF is a read only register. It can be read by the core. It can also be read by the command-list execution unit using the SOVF instruction. Reading the OVF by either the core or the command-list execution unit clears it to ''0''.

PARAM—Vector Parameter Register

The format of the PARAM register is shown in Figure 2-9.

31 26	25	24 19	18	17	16	15 0
Reserved	RND	OP	SUB	CLR	COJ	LENGTH

FIGURE 2-9. PARAM Register Format

The PARAM register is used to specify the number of iterations and special options for the various instructions. The options are: RND, OP, SUB, CLR, and COJ. The effect of each of the bits of the PARAM register is specified in Section 3.4.

The PARAM register can be read and written by the core. It can also be written by the command-list execution unit, by using the LPARAM instruction. The value written into PARAM.LENGTH must be greater then 0.

The value of PARAM.LENGTH is not changed during command-list execution, unless it is written into using the LPARAM instruction.

REPEAT—Command-List Repeat Register

The format of the repeat register is shown in *Figure 2-10*.

31 16	15 0
COUNT	TARGET

FIGURE 2-10. REPEAT Register Format

The REPEAT register is used, together with appropriate commands, to implement loops and branches in the command list. The count is used to specify the number of times a loop in the command list is to be repeated. The target is used to specify a jump address within the command list.

The REPEAT register can be read and written by the core. It can also be read and written by the command-list execution unit by using SREPEAT and LREPEAT instructions respectively.

The value of REPEAT.COUNT changes during the execution of the DJNZ command.

ABORT—Abort Register

The ABORT register is used to force execution of the command list to halt. Writing any value into this register stops execution, and clears the contents of OVF, EXT, DSPINT and DSPMASK. The ABORT register can only be written and only by the core.

EXT—External Memory Reference Control Register

The format of the external memory reference control register is shown in *Figure 2-11*.

15	1	0
Reserved		HOLD

FIGURE 2-11. EXT Register Format

The EXT register controls external references. The command-list execution unit checks the value of EXT.HOLD before each external memory reference. When EXT.HOLD is "0", external memory references are allowed. When EXT.HOLD is "1", and external memory references are requested, the execution of the command list will stop until EXT.HOLD is "0". Upon reset, and whenever the ABORT register is written, EXT.HOLD is cleared to "0". The EXT register can be read or written by the core.

CLSTAT—Command-List Execution Status Register

The format of the command-list execution status register is shown in *Figure 2-12*.

15	1	0
Reserved		RUN

FIGURE 2-12. CLSTAT Register Format

The CLSTAT register displays the current status of the execution of the command list. When the command-list execution is idle, CLSTAT.RUN is "0", and when it is active, CLSTAT.RUN is "1". Upon reset, the CLSTAT register is cleared to "0". It can only be read, and only by the core.

DSPINT, DSPMASK, NMISTAT—Interrupt Control Registers

The format of DSPINT and DSPMASK is shown in *Figure 2-13*.

15	1	0
Reserved		HALT

FIGURE 2-13. DSPINT and DSPMASK Register Format

The DSPINT register holds the current status of interrupt requests. Whenever execution of the command list is stopped, the DSPINT.HALT bit is set to "1". The DSPINT is a read only register. It is cleared to "0" whenever it is read, whenever the ABORT register is written, and upon reset.

The DSPMASK register is used to mask the DSPINT. HALT flag. An interrupt request is transferred to the interrupt logic of the $\overline{\text{IOUT}}$ output pin whenever the DSPINT.HALT bit is set to "1", and the DSPMASK.HALT bit is unmasked (set to "1"). See Section 4.0 for the functionality of $\overline{\text{IOUT}}$. DSPMASK can be read and written by the core. Upon reset, and whenever the ABORT register is written, all the bits in DSPMASK are cleared to "0".

The format of the NMISTAT register is shown in *Figure 2-14*.

15	3	2	1	0
Reserved		ERR	UND	EXT

FIGURE 2-14. NMISTAT Register Format

The NMISTAT holds the status of the current pending Non-Maskable Interrupt (NMI) requests.

Whenever the core attempts to access the DSPM address space while the CLSTAT.RUN bit is "1" (except for accesses to the CLSTAT, EXT, DSPINT, NMISTAT, DSPMASK, and ABORT registers) NMISTAT.ERR is set to "1".

Whenever there is an attempt to execute a DBPT instruction, or a reserved DSPM instruction (Section 3.4), the NMISTAT.UND bit is set to "1".

When a high to low transition is detected on the $\overline{\text{NMI}}$ input pin, NMISTAT.EXT bit is set to "1".

When one of the bits in NMISTAT is set to "1", an NMI request to the core is issued.

The NMISTAT register is cleared to 0 upon reset, and each time its contents are read.

When one of the bits in NMISTAT is set to 1, an NMI occurs. The NMI handler can read the NMISTAT register to determine the source of the interrupt. Note that since NMIs may be nested, it is possible that a second NMI handler (invoked while the previous handler has not yet exited) will read and handle more than one set bit in NMISTAT. Since the read operation clears the register, the interrupted handler may find that no bits are set.

2.2 MEMORY ORGANIZATION

The main memory of the NS32FX164 is a uniform linear address space. Memory locations are numbered sequentially starting at zero and ending at $2^{24} - 1$. The number specifying a memory location is called an address. The contents of each memory location is a byte consisting of eight bits. Unless otherwise noted, diagrams in this document show data stored in memory with the lowest address on the right and the highest address on the left. Also, when data is shown vertically, the lowest address is at the top of a diagram and the highest address at the bottom of the diagram. When bits are numbered in a diagram, the least significant bit is given the number zero, and is shown at the right of the diagram. Bits are numbered in increasing significance and toward the left.

7	0
A	

Byte at Address A

Two contiguous bytes are called a word. Except where noted, the least significant byte of a word is stored at the lower address, and the most significant byte of the word is stored at the next higher address. In memory, the address of a word is the address of its least significant byte, and a word may start at any address.

15	8	7	0
A+1		A	
MSB		LSB	

Word at Address A

Two contiguous words are called a double-word. Except where noted, the least significant word of a double-word is stored at the lowest address and the most significant word of the double-word is stored at the address two higher. In memory, the address of a double-word is the address of its least significant byte, and a double-word may start at any address.

31	24	23	16	15	8	7	0
A+3		A+2		A+1		A	
MSB						LSB	

Double Word at Address A

Although memory is addressed as bytes, it is actually organized as words. Therefore, words and double-words that are aligned to start at even addresses (multiples of two) are accessed more quickly than words and double-words that are not so aligned.

2.2.1 Address Mapping

The NS32FX164 supports the use of memory-mapped peripheral devices and coprocessors. Such memory-mapped devices can be located at arbitrary locations within the 16-Mbyte address range available externally.

Addresses marked as Reserved in *Figure 2-15* are not available in the present implementation of the NS32FX164, and should not be used. The top 8-Mbyte block is reserved by National Semiconductor Corporation, and only a few locations within this block are presently used to access the on-chip RAM array and DSP Module registers. *Figure 2-15* shows the NS32FX164 address mapping.

Start Address (HEX)	
00000000	Memory and I/O
00FFFE00	Interrupt Control
01000000	Reserved
FFFE0000	DSPM Internal RAM
FFFE1000	Reserved
FFFF8000	DSPM Dedicated Registers
FFFF8028	Reserved
FFFF9000	DSPM Control/Status Registers
FFFF9014	Reserved

FIGURE 2-15. NS32FX164 Address Mapping

2.3 MODULAR SOFTWARE SUPPORT

The NS32FX164 provides special support for software modules and modular programs.

Each module in a NS32FX164 software environment consists of three components:

1. Program Code Segment.

   This segment contains the module's code and constant data.

2. Static Data Segment.

   Used to store variables and data that may be accessed by all procedures within the module.

3. Link Table.

   This component contains two types of entries: Absolute Addresses and Procedure Descriptors.

   An Absolute Address is used in the external addressing mode, in conjunction with a displacement and the current MOD Register contents to compute the effective address of an external variable belonging to another module.

   The Procedure Descriptor is used in the call external procedure (CXP) instruction to compute the address of an external procedure.

Normally, the linker program specifies the locations of the three components. The Static Data and Link Table typically reside in RAM; the code component can be either in RAM or in ROM. The three components can be mapped into non-contiguous locations in memory, and each can be independently relocated. Since the Link Table contains the absolute addresses of external variables, the linker need not assign absolute memory addresses for these in the module itself; they may be assigned at load time.

To handle the transfer of control from one module to another, the NS32FX164 uses a module table in memory and two registers in the CPU.

The Module Table is located within the first 64 kbytes of memory. This table contains a Module Descriptor (also called a Module Table Entry) for each module in the address space of the program. A Module Descriptor has four 32-bit entries corresponding to each component of a module:

- The Static Base entry contains the address of the beginning of the module's static data segment.
- The Link Table Base points to the beginning of the module's Link Table.
- The Program Base is the address of the beginning of the code and constant data for the module.
- A fourth entry is currently unused but reserved.

The MOD Register in the CPU contains the address of the Module Descriptor for the currently executing module.

The Static Base Register (SB) contains a copy of the Static Base entry in the Module Descriptor of the currently executing module, i.e., it points to the beginning of the current module's static data area.

This register is implemented in the CPU for efficiency purposes. By having a copy of the static base entry or chip, the CPU can avoid reading it from memory each time a data item in the static data segment is accessed.

In an NS32FX164 software environment modules need not be linked together prior to loading. As modules are loaded, a linking loader simply updates the Module Table and fills the Link Table entries with the appropriate values. No modification of a module's code is required. Thus, modules may be stored in read-only memory and may be added to a system independently of each other, without regard to their individual addressing. *Figure 2-16* shows a typical NS32FX164 run-time environment.

2.4 INSTRUCTION SET

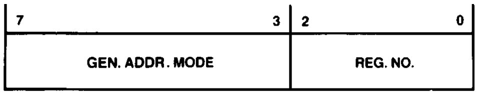
2.4.1 General Instruction Format

Figure 2-17 shows the general format of a Series 32000 instruction. The Basic Instruction is one to three bytes long and contains the Opcode and up to two 5-bit General Addressing Mode (''Gen'') fields. Following the Basic Instruction field is a set of optional extensions, which may appear depending on the instruction and the addressing modes selected.

Index Bytes appear when either or both Gen fields specify Scaled Index. In this case, the Gen field specifies only the Scale Factor (1, 2, 4 or 8), and the Index Byte specifies which General Purpose Register to use as the index, and which addressing mode calculation to perform before indexing.

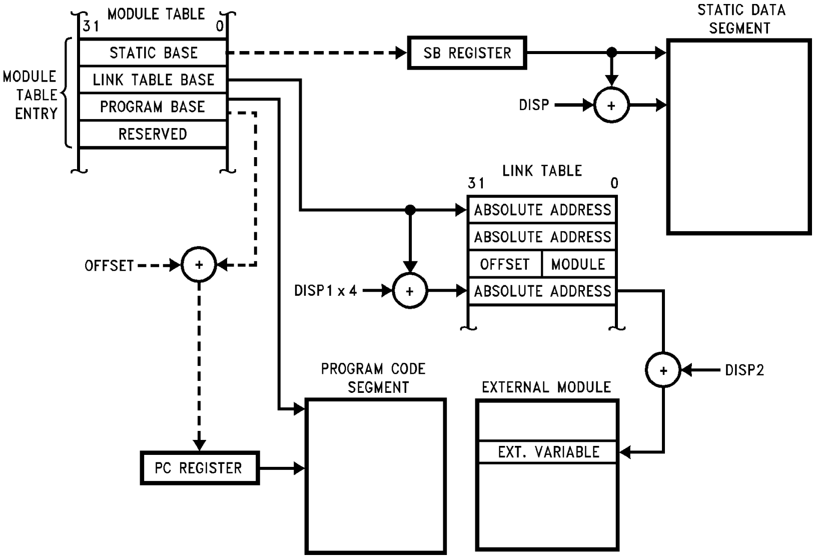
Following Index Bytes come any displacements (addressing constants) or immediate values associated with the selected addressing modes. Each Disp/Imm field may contain one of two displacements, or one immediate value. The size of a Displacement field is encoded within the top bits of that field, as shown in *Figure 2-19*, with the remaining bits interpreted as a signed (two's complement) value. The size of an immediate value is determined from the Opcode field. Both Displacement and Immediate fields are stored most-significant byte first. Note that this is different from the memory representation of data (Section 2.2).

Some instructions require additional "implied" immediates and/or displacements, apart from those associated with addressing modes. Any such extensions appear at the end of the instruction, in the order that they appear within the list of operands in the instruction definition (Section 2.4.3).



TL/EE/11267–3

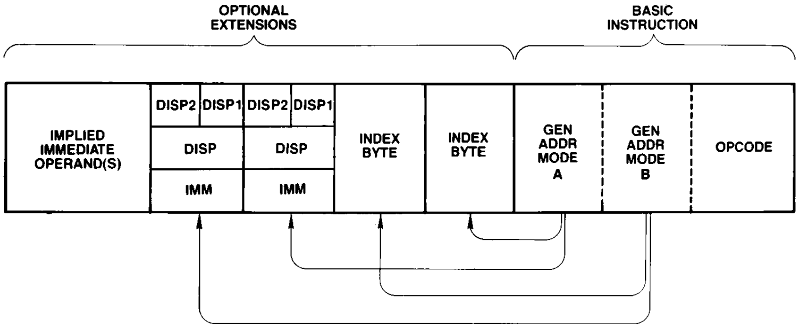
FIGURE 2-18. Index Byte Format



TL/EE/11267–2

Note: Dashed lines indicate information copied to register during transfer of control between modules.

FIGURE 2-16. NS32FX164 Run-Time Environment



TL/EE/11267–4

FIGURE 2-17. General Instruction Format

2.4.2 Addressing Modes

The NS32FX164 CPU generally accesses an operand by calculating its Effective Address based on information available when the operand is to be accessed. The method to be used in performing this calculation is specified by the programmer as an "addressing mode".

Addressing modes in the NS32FX164 are designed to optimally support high-level language accesses to variables. In nearly all cases, a variable access requires only one addressing mode, within the instruction that acts upon that variable. Extraneous data movement is therefore minimized.

NS32FX164 Addressing Modes fall into nine basic types:

Register: The operand is available in one of the eight General Purpose Registers. In certain Slave Processor instructions, an auxiliary set of eight registers may be referenced instead.

Register Relative: A General Purpose Register contains an address to which is added a displacement value from the instruction, yielding the Effective Address of the operand in memory.

Memory Space: Identical to Register Relative above, except that the register used is one of the dedicated registers PC, SP, SB or FP. These registers point to data areas generally needed by high-level languages.

Memory Relative: A pointer variable is found within the memory space pointed to by the SP, SB or FP register. A displacement is added to that pointer to generate the Effective Address of the operand.

Immediate: The operand is encoded within the instruction. This addressing mode is not allowed if the operand is to be written.

Absolute: The address of the operand is specified by a displacement field in the instruction.

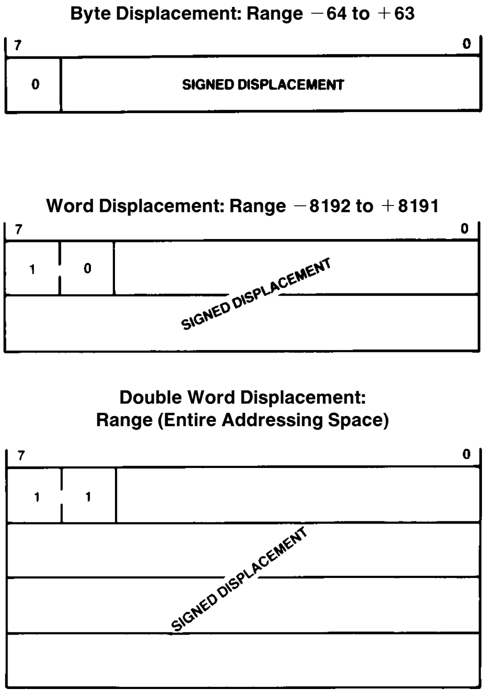
External: A pointer value is read from a specified entry of the current Link Table. To this pointer value is added a displacement, yielding the Effective Address of the operand.

Top of Stack: The currently-selected Stack Pointer (SP0 or SP1) specifies the location of the operand. The operand is pushed or popped, depending on whether it is written or read.

Scaled Index: Although encoded as an addressing mode, Scaled Indexing is an option on any addressing mode except Immediate or another Scaled Index. It has the effect of calculating an Effective Address, then multiplying any General Purpose Register by 1, 2, 4 or 8 and adding into the total, yielding the final Effective Address of the operand.

Table 2-1 is a brief summary of the addressing modes. For a complete description of their actions, see the Series 32000 Instruction Set Reference Manual.

In addition to the general modes, Register-Indirect with auto-increment/decrement and warps or pitch are available on several of the graphics instructions.



Byte Displacement: Range −64 to +63

Word Displacement: Range −8192 to +8191

Double Word Displacement:
Range (Entire Addressing Space)

TL/EE/11267-5

FIGURE 2-19. Displacement Encodings

TABLE 2-1. NS32FX164 Addressing Modes

ENCODING	MODE	ASSEMBLER SYNTAX	EFFECTIVE ADDRESS
Register			
00000	Register 0	R0 or F0	None: Operand is in the specified register.
00001	Register 1	R1 or F1	
00010	Register 2	R2 or F2	
00011	Register 3	R3 or F3	
00100	Register 4	R4 or F4	
00101	Register 5	R5 or F5	
00110	Register 6	R6 or F6	
00111	Register 7	R6 or F7	
Register Relative			
01000	Register 0 relative	disp(R0)	Disp + Register.
01001	Register 1 relative	disp(R1)	
01010	Register 2 relative	disp(R2)	
01011	Register 3 relative	disp(R3)	
01100	Register 4 relative	disp(R4)	
01101	Register 5 relative	disp(R5)	
01110	Register 6 relative	disp(R6)	
01111	Register 7 relative	disp(R7)	
Memory Relative			
10000	Frame memory relative	disp2(disp1 (FP))	Disp2 + Pointer; Pointer found at address Disp 1 + Register. "SP" is either SP0 or SP1, as selected in PSR.
10001	Stack memory relative	disp2(disp1 (SP))	
10010	Static memory relative	disp2(disp1 (SB))	
Reserved			
10011	(Reserved for Future Use)		
Immediate			
10100	Immediate	value	None: Operand is input from instruction queue.
Absolute			
10101	Absolute	@disp	Disp.
External			
10110	External	EXT (disp1) + disp2	Disp2 + Pointer; Pointer is found at Link Table Entry number Disp1.
Top Of Stack			
10111	Top of stack	TOS	Top of current stack, using either User or Interrupt Stack Pointer, as selected in PSR. Automatic Push/Pop included.
Memory Space			
11000	Frame memory	disp(FP)	Disp + Register; "SP" is either SP0 or SP1, as selected in PSR.
11001	Stack memory	disp(SP)	
11010	Static memory	disp(SB)	
11011	Program memory	* + disp	
Scaled Index			
11100	Index, bytes	mode[Rn:B]	EA (mode) + Rn.
11101	Index, words	mode[Rn:W]	EA (mode) + 2×Rn.
11110	Index, double words	mode[Rn:D]	EA (mode) + 4×Rn.
11111	Index, quad words	mode[Rn:Q]	EA (mode) + 8×Rn. "Mode" and "n" are contained within the Index Byte. EA (mode) denotes the effective address generated using mode.

2.4.3 Instruction Set Summary

Table 2-2 presents a brief description of the NS32FX164 instruction set. The Format column refers to the Instruction Format tables (Appendix A). The Instruction column gives the instruction as coded in assembly language, and the Description column provides a short description of the function provided by that instruction. Further details of the exact operations performed by each instruction may be found in the Series 32000 Instruction Set Reference Manual and the NS32CG16 Printer/Display Processor Programmer's Reference.

Notations:

i = Integer length suffix: B = Byte
W = Word
D = Double Word

f = Floating Point length suffix: F = Standard Floating
L = Long Floating

gen = General operand. Any addressing mode can be specified.

short = A 4-bit value encoded within the Basic Instruction (see Appendix A for encodings).

imm = Implied immediate operand. An 8-bit value appended after any addressing extensions.

disp = Displacement (addressing constant): 8, 16 or 32 bits. All three lengths legal.

reg = Any General Purpose Register: R0–R7.

areg = Any Processor Register: SP, SB, FP, INTBASE, MOD, PSR, US (bottom 8 PSR bits).

cond = Any condition code, encoded as a 4-bit field within the Basic Instruction (see Appendix A for encodings).

TABLE 2-2. NS32FX164 Instruction Set Summary

MOVES

Format	Operation	Operands	Description
4	MOVi	gen,gen	Move a value.
2	MOVQi	short,gen	Extend and move a signed 4-bit constant.
7	MOVMi	gen,gen,disp	Move multiple: disp bytes (1 to 16).
7	MOVZBW	gen,gen	Move with zero extension.
7	MOVZiD	gen,gen	Move with zero extension.
7	MOVXBW	gen,gen	Move with sign extension.
7	MOVXiD	gen,gen	Move with sign extension.
4	ADDR	gen,gen	Move effective address.

INTEGER ARITHMETIC

Format	Operation	Operands	Description
4	ADDi	gen,gen	Add.
2	ADDQi	short,gen	Add signed 4-bit constant.
4	ADDCi	gen,gen	Add with carry.
4	SUBi	gen,gen	Subtract.
4	SUBCi	gen,gen	Subtract with carry (borrow).
6	NEGi	gen,gen	Negate (2's complement).
6	ABSi	gen,gen	Take absolute value.
7	MULi	gen,gen	Multiply.
7	QUOi	gen,gen	Divide, rounding toward zero.
7	REMi	gen,gen	Remainder from QUO.
7	DIVi	gen,gen	Divide, rounding down.
7	MODi	gen,gen	Remainder from DIV (Modulus).
7	MEIi	gen,gen	Multiply to extended integer.
7	DEIi	gen,gen	Divide extended integer.

PACKED DECIMAL (BCD) ARITHMETIC

Format	Operation	Operands	Description
6	ADDPi	gen,gen	Add packed.
6	SUBPi	gen,gen	Subtract packed.

TABLE 2-2. NS32FX164 Instruction Set Summary (Continued)

INTEGER COMPARISON

Format	Operation	Operands	Description
4	CMPi	gen,gen	Compare.
2	CMPQi	short,gen	Compare to signed 4-bit constant.
7	CMPMi	gen,gen,disp	Compare multiple: disp bytes (1 to 16).

LOGICAL AND BOOLEAN

Format	Operation	Operands	Description
4	ANDi	gen,gen	Logical AND.
4	ORi	gen,gen	Logical OR.
4	BICi	gen,gen	Clear selected bits.
4	XORi	gen,gen	Logical exclusive OR.
6	COMi	gen,gen	Complement all bits.
6	NOTi	gen,gen	Boolean complement: LSB only.
2	Scondi	gen	Save condition code (cond) as a Boolean variable of size i.

SHIFTS

Format	Operation	Operands	Description
6	LSHi	gen,gen	Logical shift, left or right.
6	ASHi	gen,gen	Arithmetic shift, left or right.
6	ROTi	gen,gen	Rotate, left or right.

BIT FIELDS

Bit fields are values in memory that are not aligned to byte boundaries. Examples are PACKED arrays and records used in Pascal. ‘‘Extract’’ instructions read and align a bit field. ‘‘Insert’’ instructions write a bit field from an aligned source.

Format	Operation	Operands	Description
8	EXTi	reg,gen,gen,disp	Extract bit field (array oriented).
8	INSi	reg,gen,gen,disp	Insert bit field (array oriented).
7	EXTSi	gen,gen,imm,imm	Extract bit field (short form).
7	INSSi	gen,gen,imm,imm	Insert bit field (short form).
8	CVTP	reg,gen,gen	Convert to bit field pointer.

ARRAYS

Format	Operation	Operands	Description
8	CHECKi	reg,gen,gen	Index bounds check.
8	INDEXi	reg,gen,gen	Recursive indexing step for multiple-dimensional arrays.

STRINGS

String instructions assign specific functions to the General Purpose Registers:

R4 — Comparison Value

R3 — Translation Table Pointer

R2 — String 2 Pointer

R1 — String 1 Pointer

R0 — Limit Count

Options on all string instructions are:

B (Backward): Decrement string pointers after each step rather than incrementing.

U (Until match): End instruction if String 1 entry matches R4.

W (While match): End instruction if String 1 entry does not match R4.

All string instructions end when R0 decrements to zero.

2.0 Architectural Description (Continued)

TABLE 2-2. NS32FX164 Instruction Set Summary (Continued)

Format	Operation	Operands	Description
5	MOVSi	options	Move string 1 to string 2.
	MOVST	options	Move string, translating bytes.
5	CMPSi	options	Compare string 1 to string 2.
	CMPST	options	Compare, translating string 1 bytes.
5	SKPSi	options	Skip over string 1 entries.
	SKPST	options	Skip, translating bytes for until/while.

JUMPS AND LINKAGE

Format	Operation	Operands	Description
3	JUMP	gen	Jump.
0	BR	disp	Branch (PC Relative).
0	Bcond	disp	Conditional branch.
3	CASEi	gen	Multiway branch.
2	ACBi	short,gen,disp	Add 4-bit constant and branch if non-zero.
3	JSR	gen	Jump to subroutine.
1	BSR	disp	Branch to subroutine.
1	CXP	disp	Call external procedure
3	CXPD	gen	Call external procedure using descriptor.
1	SVC		Supervisor call.
1	FLAG		Flag trap.
1	BPT		Breakpoint trap.
1	ENTER	[reg list], disp	Save registers and allocate stack frame (Enter Procedure).
1	EXIT	[reg list]	Restore registers and reclaim stack frame (Exit Procedure).
1	RET	disp	Return from subroutine.
1	RXP	disp	Return from external procedure call.
1	RETT	disp	Return from trap. (Privileged)
1	RETI		Return from interrupt. (Privileged)

CPU REGISTER MANIPULATION

Format	Operation	Operands	Description
1	SAVE	[reg list]	Save general purpose registers.
1	RESTORE	[reg list]	Restore general purpose registers.
2	LPRi	areg,gen	Load dedicated register. (Privileged if PSR or INTBASE)
2	SPRi	areg,gen	Store dedicated register. (Privileged if PSR or INTBASE)
3	ADJSPi	gen	Adjust stack pointer.
3	BISPSRi	gen	Set selected bits in PSR. (Privileged if not Byte length)
3	BICPSRi	gen	Clear selected bits in PSR. (Privileged if not Byte length)
5	SETCFG	[option list]	Set configuration register. (Privileged)

2.0 Architectural Description (Continued)

TABLE 2-2. NS32FX164 Instruction Set Summary (Continued)

FLOATING POINT

Format	Operation	Operands	Description
11	MOVf	gen,gen	Move a floating point value.
9	MOVLF	gen,gen	Move and shorten a long value to standard.
9	MOVFL	gen,gen	Move and lengthen a standard value to long.
9	MOVif	gen,gen	Convert any integer to standard or long floating.
9	ROUNDfi	gen,gen	Convert to integer by rounding.
9	TRUNCfi	gen,gen	Convert to integer by truncating, toward zero.
9	FLOORfi	gen,gen	Convert to largest integer less than or equal to value.
11	ADDf	gen,gen	Add.
11	SUBf	gen,gen	Subtract.
11	MULf	gen,gen	Multiply.
11	DIVf	gen,gen	Divide.
11	CMPf	gen,gen	Compare.
11	NEGf	gen,gen	Negate.
11	ABSf	gen,gen	Take absolute value.
9	LFSR	gen	Load FSR.
9	SFSR	gen	Store FSR.
12	POLYf	gen,gen	Polynomial Step.
12	DOTf	gen,gen	Dot Product.
12	SCALBf	gen,gen	Binary Scale.
12	LOGBf	gen,gen	Binary Log.

MISCELLANEOUS

Format	Operation	Operands	Description
1	NOP		No operation.
1	WAIT		Wait for interrupt.
1	DIA		Diagnose. Single-byte ''Branch to Self'' for hardware breakpointing. Not for use in programming.

GRAPHICS

Format	Operation	Operands	Description
5	BBOR	options*	Bit-aligned block transfer 'OR'.
5	BBAND	options	Bit-aligned block transfer 'AND'.
5	BBFOR		Bit-aligned block transfer fast 'OR'.
5	BBXOR	options	Bit-aligned block transfer 'XOR'.
5	BBSTOD	options	Bit-aligned block source to destination.
5	BITWT		Bit-aligned word transfer.
5	EXTBLT	options	External bit-aligned block transfer.
5	MOVMPi		Move multiple pattern.
5	TBITS	options	Test bit string.
5	SBITS		Set bit string.
5	SBITPS		Set bit perpendicular string.

BITS

Format	Operation	Operands	Description
4	TBITi	gen,gen	Test bit.
6	SBITi	gen,gen	Test and set bit.
6	SBITIi	gen,gen	Test and set bit, interlocked.
6	CBITi	gen,gen	Test and clear bit.
6	CBITIi	gen,gen	Test and clear bit, interlocked.
6	IBITi	gen,gen	Test and invert bit.
8	FFSi	gen,gen	Find first set bit.

*__Note:__ Options are controlled by fields of the instruction, PSR status bits, or dedicated register values.

2.5 GRAPHICS SUPPORT

The following sections provide a brief description of the NS32FX164 graphics support capabilities. Basic discussions on frame buffer addressing and BITBLT operations are also provided. More detailed information on the NS32FX164 graphics support instructions can be found in the NS32CG16 Printer/Display Processor Programmer's Reference.

2.5.1 Frame Buffer Addressing

There are two basic addressing schemes for referencing pixels within the frame buffer: Linear and Cartesian (or x-y). Linear addressing associates a single number to each pixel representing the physical address of the corresponding bit in memory. Cartesian addressing associates two numbers to each pixel representing the x and y coordinates of the pixel relative to a point in the Cartesian space taken as the origin. The Cartesian space is generally defined as having the origin in the upper left. A movement to the right increases the x coordinate; a movement downward increases the y coordinate.

The correspondence between the location of a pixel in the Cartesian space and the physical (BIT) address in memory is shown in *Figure 2-20*. The origin of the Cartesian space (x=0, y=0) corresponds to the bit address 'ORG'. Incrementing the x coordinate increments the bit address by one. Incrementing the y coordinate increments the bit address by an amount representing the warp (or pitch) of the Cartesian space. Thus, the linear address of a pixel at location (x, y) in the Cartesian space can be found by the following expression.

$$ADDR = ORG + y * WARP + x$$

Warp is the distance (in bits) in the physical memory space between two vertically adjacent bits in the Cartesian space.

Example 1 below shows two NS32FX164 instruction sequences to set a single pixel given the x and y coordinates. Example 2 shows how to create a fat pixel by setting four adjacent bits in the Cartesian space.

Example 1: Set pixel at location (x, y)

Setup: R0 x coordinate
R1 y coordinate

Instruction Sequence 1:

```
MULD   WARP, R1            ; Y*WARP
ADDD   R0, R1              ; + X = BIT OFFSET
SBITD  R1, ORG             ; SET PIXEL
```

Instruction Sequence 2:

```
INDEXD R1, (WARP-1), R0   ; Y*WARP + X
SBITD  R1, ORG            ; SET PIXEL
```

Example 2: Create fat pixel by setting bits at locations (x, y), (x+1, y), (x, y+1) and (x+1, y+1).

Setup: R0 x coordinate
R1 y coordinate

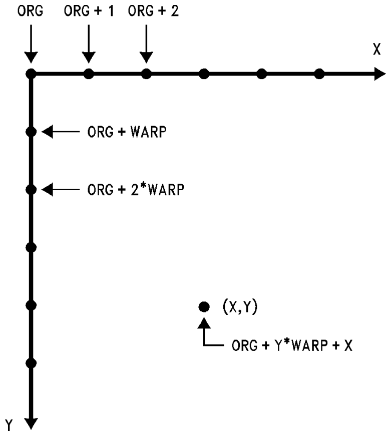
Instruction Sequence:

```
INDEXD  R1, (WARP-1), R0   ; BIT ADDRESS
SBITD   41, ORG            ; SET FIRST PIXEL

ADDQD   1, R1              ; (X+1, Y)
SBITD   R1, ORG            ; SECOND PIXEL

ADDD    (WARP-1), R1       ; (X, Y+1)
SBITD   R1, ORG            ; THIRD PIXEL

ADDQD   1, R1              ; (X+1, Y+1)
SBITD   R1, ORG            ; LAST PIXEL
```



TL/EE/11267–6

FIGURE 2-20. Correspondence between Linear and Cartesian Addressing

2.5.2 BITBLT Fundamentals

BITBLT, BIT-aligned BLock Transfer, is a general operator that provides a mechanism to move an arbitrary size rectangle of an image from one part of the frame buffer to another. During the data transfer process a bitwise logical operation can be performed between the source and the destination data. BITBLT is also called RasterOp: operations on rasters. It defines two rectangular areas, source and destination, and performs a logical operation (e.g., AND, OR, XOR) between these two areas and stores the result back to the destination. It can be expressed in simple notation as:

Source op Destination ⟶ Destination
op: AND, OR, XOR, etc.

2.5.2.1 Frame Buffer Architecture

There are two basic types of frame buffer architectures: plane-oriented or pixel-oriented. BITBLT takes advantage of the plane-oriented frame buffer architecture's attribute of multiple, adjacent pixels-per-word, facilitating the movement of large blocks of data. The source and destination starting addresses are expressed as pixel addresses. The width and height of the block to be moved are expressed in terms of pixels and scan lines. The source block may start and end at any bit position of any word, and the same applies for the destination block.

2.5.2.2 Bit Alignment

Before a logical operation can be performed between the source and the destination data, the source data must first be bit aligned to the destination data. In *Figure 2-21*, the source data needs to be shifted three bits to the right in order to align the first pixel (i.e., the pixel at the top left corner) in the source data block to the first pixel in the destination data block.

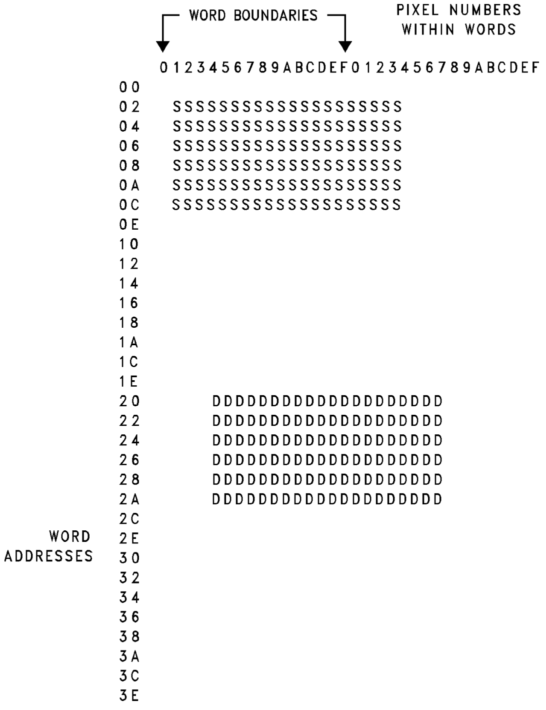
2.5.2.3 Block Boundaries and Destination Masks

Each BITBLT destination scan line may start and end at any bit position in any data word. The neighboring bits (bits sharing the same word address with any words in the destination data block, but not a part of the BITBLT rectangle) of the BITBLT destination scan line must remain unchanged after the BITBLT operation.

Due to the plane-oriented frame buffer architecture, all memory operations must be word-aligned. In order to preserve the neighboring bits surrounding the BITBLT destination block, both a left mask and a right mask are needed for all the leftmost and all the rightmost data words of the destination block. The left mask and the right mask both remain the same during a BITBLT operation.

The following example illustrates the bit alignment requirements. In this example, the memory data path is 16 bits wide. *Figure 2-21* shows a 32 pixel by 32 scan line frame buffer which is organized as a long bit stream which wraps around every two words (32 bits). The origin (top left corner) of the frame buffer starts from the lowest word in memory (word address 00 (hex)).

Each word in the memory contains 16 bits, D0–D15. The least significant bit of a memory word, D0, is defined as the first displayed pixel in a word. In this example, BITBLT addresses are expressed as pixel addresses relative to the origin of the frame buffer. The source block starting address is 021 (hex) (the second pixel in the third word). The destination block starting address is 204 (hex) (the fifth pixel in the 33rd word). The block width is 13 (hex), and the height is 06 (hex) (corresponding to 6 scan lines). The shift value is 3.



```
   0123456789ABCDEF0123456789ABCDEF
00
02  SSSSSSSSSSSSSSSSSSSS
04  SSSSSSSSSSSSSSSSSSSS
06  SSSSSSSSSSSSSSSSSSSS
08  SSSSSSSSSSSSSSSSSSSS
0A  SSSSSSSSSSSSSSSSSSSS
0C  SSSSSSSSSSSSSSSSSSSS
0E
10
12
14
16
18
1A
1C
1E
20     DDDDDDDDDDDDDDDDDDDD
22     DDDDDDDDDDDDDDDDDDDD
24     DDDDDDDDDDDDDDDDDDDD
26     DDDDDDDDDDDDDDDDDDDD
28     DDDDDDDDDDDDDDDDDDDD
2A     DDDDDDDDDDDDDDDDDDDD
2C
2E
30
32
34
36
38
3A
3C
3E
```

TL/EE/11267–7

FIGURE 2-21. 32-Pixel by 32-Scan Line Frame Buffer

2.5.2.4 BITBLT Directions

A BITBLT operation moves a rectangular block of data in a frame buffer. The operation itself can be considered as a subroutine with two nested loops. The loops are preceded by setup operations. In the outer loop the source and destination starting addresses are calculated, and the test for completion is performed. In the inner loop the actual data movement for a single scan line takes place. The length of the inner loop is the number of (aligned) words spanned by each scan line. The length of the outer loop is equal to the height (number of scan lines) of the block to be moved. A skeleton of the subroutine representing the BITBLT operation follows.

```
BITBLT:       calculate BITBLT setup parameters;
              (once per BITBLT operation).
              such as
              width, height
              bit misalignment (shift number)
              left, right masks
              horizontal, vertical directions
              etc
              •
              •
OUTERLOOP:    calculate source, dest addresses;
              (once per scanline).
INNERLOOP:    move data, (logical operation) and incre-
              ment addresses;
              (once per word).
UNTIL         done horizontally
UNTIL         done vertically
RETURN        (from BITBLT).
```

Note: In the NS32FX164 only the setup operations must be done by the programmer. The inner and outer loops are automatically executed by the BITBLT instructions.

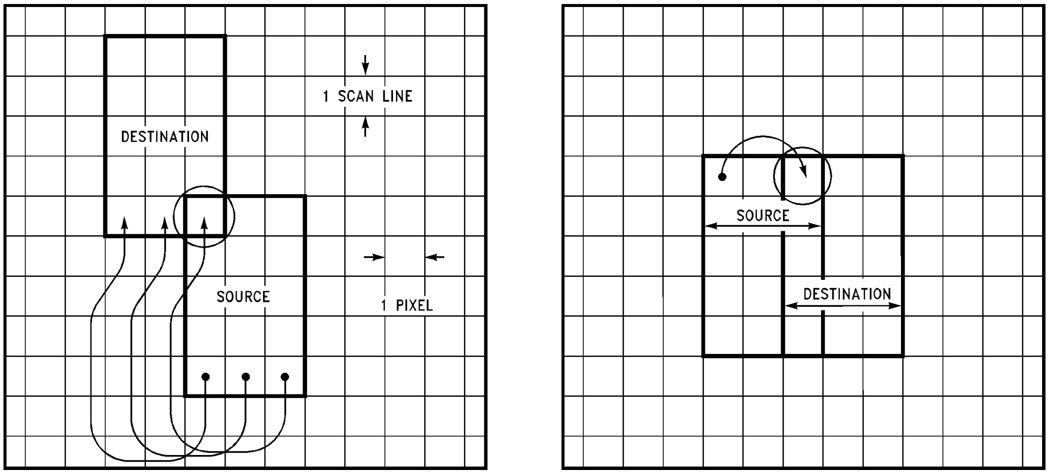
Each loop can be executed in one of two directions: the inner loop from left to right or right to left, the outer loop from top to bottom (down) or bottom to top (up).

The ability to move data starting from any corner of the BITBLT rectangle is necessary to avoid destroying the BITBLT source data as a result of destination writes when the source and destination are overlapped (i.e., when they share pixels). This situation is routinely encountered while panning or scrolling.

A determination of the correct execution directions of the BITBLT must be performed whenever the source and destination rectangles overlap. Any overlap will result in the destruction of source data (from a destination write) if the correct vertical direction is not used. Horizontal BITBLT direction is of concern only in certain cases of overlap, as will be explained below.

Figures 2-22(a) and *(b)* illustrate two cases of overlap. Here, the BITBLT rectangles are three pixels wide by five scan lines high; they overlap by a single pixel in *(a)* and a single column of pixels in *(b)*. For purposes of illustration, the BITBLT is assumed to be carried out pixel-by-pixel. This convention does not affect the conclusions.

In *Figure 2-22(a)*, if the BITBLT is performed in the UP direction (bottom-to-top) one of the transfers of the bottom scan line of the source will write to the circled pixel of the destination. Due to the overlap, this pixel is also part of the uppermost scan line of the source rectangle. Thus, data needed later is destroyed. Therefore, this BITBLT must be performed in the DOWN direction. Another example of this oc-



TL/EE/11267–8

(a)

TL/EE/11267–9

(b)

FIGURE 2-22. Overlapping BITBLT Blocks

The left mask and the right mask are 0000,1111,1111,1111 and 1111,1111,0000,0000 respectively.

Note 1: Zeros in either the left mask or the right mask indicate the destination bits which will not be modified.

Note 2: The BB(function) and EXTBLT instructions use different set up parameters, and techniques.

curs any time the screen is moved in a purely vertical direction, as in scrolling text. It should be noted that, in both of these cases, the choice of horizontal BITBLT direction may be made arbitrarily.

Figure 2-22(b) demonstrates a case in which the horizontal BITBLT direction may not be chosen arbitrarily. This is an instance of purely horizontal movement of data (panning). Because the movement from source to destination involves data within the same scan line, the incorrect direction of movement will overwrite data which will be needed later. In this example, the correct direction is from right to left.

2.5.2.5 BITBLT Variations

The "classical" definition of BITBLT, as described in "Smalltalk-80 The Language and its Implementation", by Adele Goldberg and David Robson, provides for three operands: source, destination and mask/texture. This third operand is commonly used in monochrome systems to incorporate a stipple pattern into an area. These stipple patterns provide the appearance of multiple shades of gray in single-bit-per-pixel systems, in a manner similar to the "halftone" process used in printing.

Texture op1 Source op2 Destination → Destination

While the NS32FX164 and the external BPU (if used) are essentially two-operand devices, three-operand BITBLT operations can be implemented quite flexibly and efficiently by performing the two operations serially.

2.5.3 GRAPHICS SUPPORT INSTRUCTIONS

The NS32FX164 provides eleven instructions for supporting graphics oriented applications. These instructions are divided into three groups according to the operations they perform. General descriptions for each of them and the related formats are provided in the following sections.

2.5.3.1 BITBLT (BIT-aligned BLock Transfer)

This group includes seven instructions. They are used to move characters and objects into the frame buffer which will be printed or displayed. One of the instructions works in conjunction with an external BITBLT Processing Unit (BPU) to maximize performance. The other six are executed by the NS32FX164.

BIT-aligned BLock Transfer

Syntax: BB(function) Options

Setup:	R0	base address, source data
	R1	base address, destination data
	R2	shift value
	R3	height (in lines)
	R4	first mask
	R5	second mask
	R6	source warp (adjusted)
	R7	destination warp (adjusted)
	0(SP)	width (in words)
Function:	AND, OR, XOR, FOR, STOD	
Options:	IA	Increasing Address (default option). When IA is selected, scan lines are transferred in the increasing BIT/BYTE order.
	DA	Decreasing Address.
	S	True Source (default option).
	−S	Inverted Source.

These five instructions perform standard BITBLT operations between source and destination blocks. The operations available include the following:

BBAND:	src	AND	dst
	−src	AND	dst
BBOR:	src	OR	dst
	−src	OR	dst
BBXOR:	src	XOR	dst
	−src	XOR	dst
BBFOR:	src	OR	dst
BBSTOD:	src	TO	dst
	−src	TO	dst

'src' and '−src' stand for 'True Source' and 'Inverted Source' respectively; 'dst' stands for 'Destination'.

Note 1: For speed reasons, the BB instructions require the masks to be specified with respect to the source block. In *Figure 2-21* masking was defined relative to the destination block.

Note 2: The options −S and DA are not available for the BBFOR instruction.

Note 3: BBFOR performs the same operation as BBOR with IA and S options.

Note 4: IA and DA are mutually exclusive and so are S and −S.

Note 5: The width is defined as the number of words of source data to read.

Note 6: An odd number of bytes can be specified for the source warp. However, word alignment of source scan lines will result in faster execution.

The horizontal and vertical directions of the BITBLT operations performed by the above instructions, with the exception of BBFOR, are both programmable. The horizontal direction is controlled by the IA and DA options. The vertical direction is controlled by the sign of the source and destination warps. *Figure 2-23* and Table 2-3 show the format of the BB instructions and the encodings for the 'op' and 'i' fields.

23 16	15			8	7 0
0 0 0 0 0 0 D X	$\overline{S}$	0	op	i	0 0 0 0 1 1 1 0

- D is set when the DA option is selected
- $\overline{S}$ is set when the −S option is selected
- X is set for BBAND, and it is clear for all other BB instructions

FIGURE 2-23. BB Instructions Format

TABLE 2-3. 'op' and 'i' Field Encodings

Instruction	Options	'op' Field	'i' Field
BBAND	Yes	1010	11
BBOR	Yes	0110	01
BBXOR	Yes	1110	01
BBFOR	No	1100	01
BBSTOD	Yes	0100	01

BIT-aligned Word Transfer

Syntax: BITWT

Setup:	R0	Base address, source word
	R1	Base address, destination double word
	R2	Shift value

The BITWT instruction performs a fast logical OR operation between a source word and a destination double word, stores the result into the destination double word and increments registers R0 and R1 by two. Before performing the OR operation, the source word is shifted left (i.e., in the direction of increasing bit numbers) by the value in register R2.

This instruction can be used within the inner loop of a block OR operation. Its use assumes that the source data is 'clean' and does not need masking. The BITWT format is shown in *Figure 2-24*.

23							16	15							8	7							0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	1	0

FIGURE 2-24. BITWT Instruction Format

External BITBLT

Syntax: EXTBLT

Setup:	R0	base addresses, source data
	R1	base address, destination data
	R2	width (in bytes)
	R3	height (in lines)
	R4	horizontal increment/decrement
	R5	temporary register (current width)
	R6	source warp (adjusted)
	R7	destination warp (adjusted)

Note 1: R0 and R1 are updated after execution to point to the last source and destination addresses plus related warps. R2, R3 and R5 will be modified. R4, R6, and R7 are returned unchanged.

Note 2: Source and destination pointers should point to word-aligned operands to maximize speed and minimize external interface logic.

This instruction performs an entire BITBLT operation in conjunction with an external BITBLT Processing Unit (BPU). The external BPU Control Register should be loaded by the software before the instruction is executed (refer to the DP8510 or DP8511 data sheets for more information on the BPU). The NS32FX164 generates a series of source read, destination read and destination write bus cycles until the entire data block has been transferred. The BITBLT operation can be performed in either horizontal direction. As controlled by the sign of the contents of register R4.

Depending on the relative alignment of the source and destination blocks, an extra source read may be required at the beginning of each scan line, to load the pipeline register in the external BPU. The L bit in the PSR register determines whether the extra source read is performed. If L is 1, no extra read is performed. The instructions CMPQB 2,1 or CMPQB 1,2 could be executed to provide the right setting for the L bit just before executing EXTBLT. *Figure 2-25* shows the EXTBLT format. The bus activity for a simple BITBLT operation is shown in *Figure 2-30.*

23								15							8	7							0
0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	1	1	1	0

FIGURE 2-25. EXTBLT Instruction Format

2.5.3.2 Pattern Fill

Only one instruction is in this group. It is usually used for clearing RAM and drawing patterns and lines.

Move Multiple Pattern

Syntax: MOVMPi

Setup:	R0	base address of the destination
	R1	pointer increment (in bytes)
	R2	number of pattern moves
	R3	source pattern

Note: R1 and R3 are not modified by the instruction. R2 will always be returned as zero. R0 is modified to reflect the last address into which a pattern was written.

This instruction stores the pattern in register R3 into the destination area whose address is in register R0. The pattern count is specified in register R2. After each store operation the destination address is changed by the contents of register R1. This allows the pattern to be stored in rows, in columns, and in any direction, depending on the value and sign of R1. The MOVMPi instruction format is shown in *Figure 2-26*.

23								15							8	7							0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	i		0	0	0	0	1	1	1	0

FIGURE 2-26. MOVMPi Instruction Format

2.5.3.3 Data Compression, Expansion and Magnify

The three instructions in this group can be used to compress data and restore data from compression. A compressed character set may require from 30% to 50% less memory space for its storage.

The compression ratio possible can be 50:1 or higher depending on the data and algorithm used. TBITS can also be used to find boundaries of an object. As a character is needed, the data is expanded and stored in a RAM buffer. The expand instructions (SBITS, SBITPS) can also function as line drawing instructions.

Test Bit String

Syntax: TBITS option

Setup:	R0	base address, source (byte address)
	R1	starting source bit offset
	R2	destination run length limited code
	R3	maximum value run length limit
	R4	maximum source bit offset
Option:	1	count set bits until a clear bit is found
	0	count clear bits until a set bit is found

Note: R0, R3 and R4 are not modified by the instruction execution. R1 reflects the new bit offset. R2 holds the result.

This instruction starts at the base address, adds a bit offset, and tests the bit for clear if "option" = 0 (and for set if "option" = 1). If clear (or set), the instruction increments to the next higher bit and tests for clear (or set). This testing for clear proceeds through memory until a set bit is found or until the maximum source bit offset or maximum run length value is reached. The total number of clear bits is stored in the destination as a run length value.

When TBITS finds a set bit and terminates, the bit offset is adjusted to reflect the current bit address. Offset is then ready for the next TBITS instruction with "option" = 0. After the instruction is executed, the F flag is set to the value of the bit previous to the bit currently being pointed to (i.e., the value of the bit on which the instruction completed execution). In the case of a starting bit offset exceeding the maximum bit offset (R1 ≥ R4), the F flag is set if the option was 1 and clear if the option was 0. The L flag is set when the desired bit is found, or if the run length equalled the maximum run length value and the bit was not found. It is cleared otherwise. *Figure 2-27* shows the TBITS instruction format.

23								15							8	7							0
0	0	0	0	0	0	0	0	S	0	1	0	0	1	1	1	0	0	0	0	1	1	1	0

- S is set for 'TBITS 1' and clear for 'TBITS 0'.

FIGURE 2-27. TBITS Instruction Format

2.0 Architectural Description (Continued)

Set Bit String

Syntax: SBITS

Setup: R0 base address of the destination
R1 starting bit offset (signed)
R2 number of bits to set (unsigned)
R3 address of string look-up table

Note: When the instruction terminates, the registers are returned unchanged.

SBITS sets a number of contiguous bits in memory to 1, and is typically used for data expansion operations. The instruction draws the number of ones specified by the value in R2, starting at the bit address provided by registers R0 and R1. In order to maximize speed and allow drawing of patterned lines, an external 1k byte lookup table is used. The lookup table is specified in the NS32CG16 Printer/Display Processor Programmer's Reference Supplement.

When SBITS begins executing, it compares the value in R2 with 25. If the value in R2 is less than or equal to 25, the F flag is cleared and the appropriate number of bits are set in memory. If R2 is greater than 25, the F flag is set and no other action is performed. This allows the software to use a faster algorithm to set longer strings of bits. *Figure 2-28* shows the SBITS instruction format.

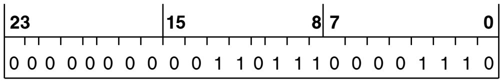


FIGURE 2-28. SBITS Instruction Format

Set BIT Perpendicular String

Syntax: SBITPS

Setup: R0 base address, destination (byte address)
R1 starting bit offset
R2 number of bits to set
R3 destination warp (signed value, in bits)

Note: When the instruction terminates, the R0 and R3 registers are returned unchanged. R1 becomes the final bit offset. R2 is zero.

The SBITPS can be used to set a string of bits in any direction. This allows a font to be expanded with a 90 or 270 degree rotation, as may be required in a printer application. SBITPS sets a string of bits starting at the bit address specified in registers R0 and R1. The number of bits in the string is specified in R2. After the first bit is set, the destination warp is added to the bit address and the next bit is set. The process is repeated until all the bits have been set. A negative raster warp offset value leads to a 90 degree rotation. A positive raster warp value leads to a 270 degree rotation. If the R3 value is = (space warp +1 or −1), then the result is a 45 degree line. If the R3 value is +1 or −1, a horizontal line results.

SBITS and SBITPS allow expansion on any 90 degree angle, giving portrait, landscape and mirror images from one font. *Figure 2-29* shows the SBITPS instruction format.

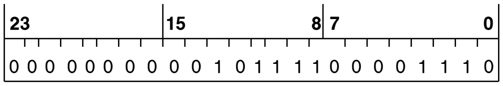
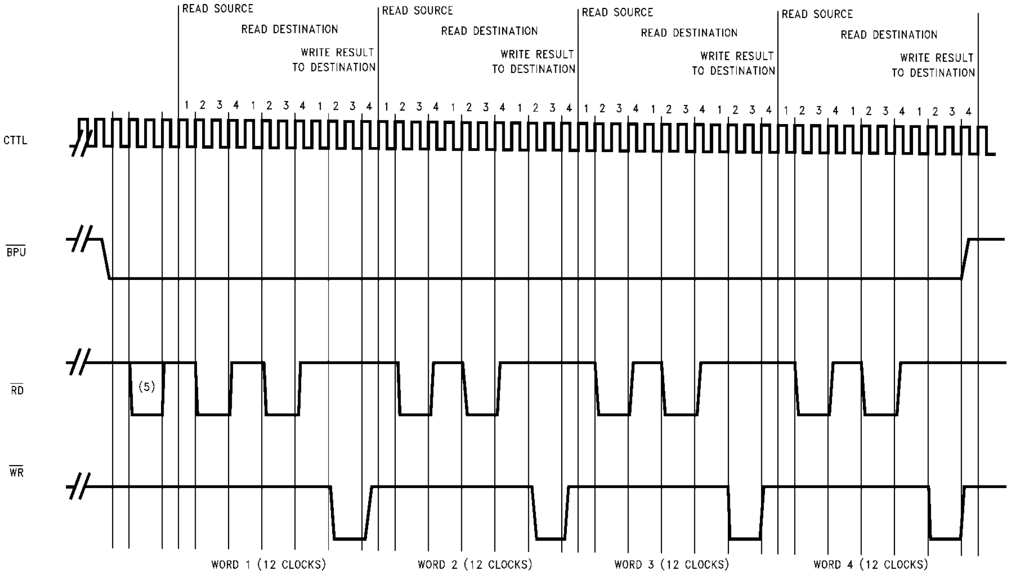


FIGURE 2-29. SBITPS Instruction Format



TL/EE/11267–10

FIGURE 2-30. Bus Activity for a Simple BITBLT Operation

Note 1: This example is for a block 4 words wide and 1 line high.

Note 2: The sequence is common with all logical operations of the DP8510/DP8511 BPU.

Note 3: Mask values, shift values and number of bit planes do not affect the performance.

Note 4: Zero wait states are assumed throughout the BITBLT operation.

Note 5: The extra read is performed when the BPU pipeline register needs to be preloaded.

2.5.3.3.1 Magnifying Compressed Data

Restoring data is just one application of the SBITS and SBITPS instructions. Multiplying the "length" operand used by the SBITS and SBITPS instructions causes the resulting pattern to be wider, or a multiple of "length".

As the pattern of data is expanded, it can be magnified by 2x, 3x, 4x, . . . , 10x and so on. This creates several sizes of the same style of character, or changes the size of a logo. A magnify in both dimensions X and Y can be accomplished by drawing a single line, then using the MOVS (Move String) or the BB instructions to duplicate the line, maintaining an equal aspect ratio.

More information on this subject is provided in the NS32CG16 Printer/Display Processor Programmer's Reference Supplement.

3.0 Functional Description

This chapter provides details on the functional characteristics of the NS32FX164 microprocessor.

The chapter is divided into five main sections:

Instruction Execution, Exception Processing, Debugging, DSP Module and System Interface.

3.1 INSTRUCTION EXECUTION

To execute an instruction, the NS32FX164 performs the following operations:

- Fetch the Instruction
- Read Source Operands, if Any (1)
- Calculate Results
- Write Result Operands, if Any
- Modify Flags, if Necessary
- Update the Program Counter

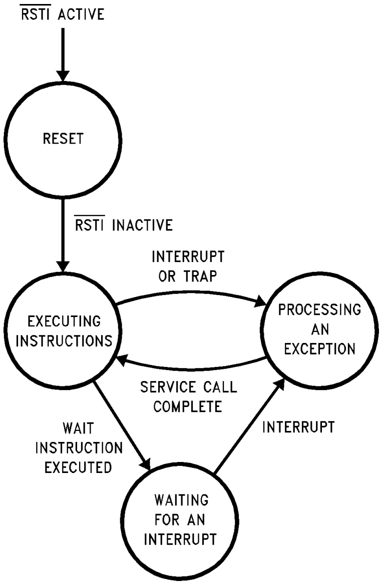
Under most circumstances, the CPU can be conceived to execute instructions by completing the operations above in strict sequence for one instruction and then beginning the sequence of operations for the next instruction. However, due to the internal instruction pipelining, as well as the occurrence of exceptions, the sequence of operations performed during the execution of an instruction may be altered. Furthermore, exceptions also break the sequentiality of the instructions executed by the CPU.

Note 1: In this and following sections, memory locations read by the CPU to calculate effective addresses for Memory-Relative and External addressing modes are considered like source operands, even if the effective address is being calculated for an operand with access class of write.

3.1.1 Operating States

The CPU has four operating states regarding the execution of instructions and the processing of exceptions: Reset, Executing Instructions, Processing An Exception and Waiting-For-An-Interrupt. The various states and transitions between them are shown in *Figure 3-1*.

Whenever the $\overline{\text{RSTI}}$ signal is asserted, the CPU enters the reset state. The CPU remains in the reset state until the $\overline{\text{RSTI}}$ signal is driven inactive, at which time it enters the Executing-Instructions state. In the Reset state the contents of certain registers are initialized. Refer to Section 3.5.4 for details.



TL/EE/11267–11

FIGURE 3-1. Operating States

In the Executing-Instructions state, the CPU executes instructions. It will exit this state when an exception is recognized or a WAIT instruction is encountered. At which time it enters the Processing-An-Exception state or the Waiting-For-An-Interrupt state respectively.

While in the Processing-An-Exception state, the CPU saves the PC, PSR and MOD register contents on the stack and reads the new PC and module linkage information to begin execution of the exception service procedure.

Following the completion of all data references required to process an exception, the CPU enters the Executing-Instructions state.

In the Waiting-For-An-Interrupt state, the CPU is idle. A special status identifying this state is presented on the system interface (Section 3.5). When an interrupt is detected, the CPU enters the Processing-An-Exception State.

3.1.2 Instruction Endings

The NS32FX164 checks for exceptions at various points while executing instructions. Certain exceptions, like interrupts, are in most cases recognized between instructions. Other exceptions, like Divide-By-Zero Trap, are recognized during execution of an instruction. When an exception is recognized during execution of an instruction, the instruction ends in one of four possible ways: completed, suspended, terminated, or partially completed. Each type of exception causes a particular ending, as specified in Section 3.2.

3.1.2.1 Completed Instructions

When an exception is recognized after an instruction is completed, the CPU has performed all of the operations for that instruction and for all other instructions executed since the last exception occurred. Result operands have been written, flags have been modified, and the PC saved on the Interrupt Stack contains the address of the next instruction to execute. The exception service procedure can, at its conclusion, execute the RETT instruction (or the RETI instruction for maskable interrupts), and the CPU will begin executing the instruction following the completed instruction.

3.1.2.2 Suspended Instructions

An instruction is suspended when one of several trap conditions is detected during execution of the instruction. A suspended instruction has not been completed, but all other instructions executed since the last exception occurred have been completed. Result operands and flags due to be affected by the instruction may have been modified, but only modifications that allow the instruction to be executed again and completed can occur. For certain exceptions (Trap (UND) the CPU clears the P-flag in the PSR before saving the copy that is pushed on the Interrupt Stack. The PC saved on the Interrupt Stack contains the address of the suspended instruction.

To complete a suspended instruction, the exception service procedure takes either of two actions:

1. The service procedure can simulate the suspended instruction's execution. After calculating and writing the instruction's results, the flags in the PSR copy saved on the Interrupt Stack should be modified, and the PC saved on the Interrupt Stack should be updated to point to the next instruction to execute. The service procedure can then execute the RETT instruction, and the CPU begins executing the instruction following the suspended instruction. This is the action taken when floating-point instructions are simulated by software in systems without a hardware floating-point unit.
2. The suspended instruction can be executed again after the service procedure has eliminated the trap condition that caused the instruction to be suspended. The service procedure should execute the RETT instruction at its conclusion; then the CPU begins executing the suspended instruction again. This is the action taken by a debugger when it encounters a BPT instruction that was temporarily placed in another instruction's location in order to set a breakpoint.

Note 1: It may be necessary for the exception service procedure to alter the P-flag in the PSR copy saved on the Interrupt Stack: If the exception service procedure simulates the suspended instruction and the P-flag was cleared by the CPU before saving the PSR copy, then the saved T-flag must be copied to the saved P-flag (like the floating-point instruction simulation described above). Or if the exception service procedure executes the suspended instruction again and the P-flag was not cleared by the CPU before saving the PSR copy, then the saved P-flag must be cleared (like the breakpoint trap described above). Otherwise, no alteration to the saved P-flag is necessary.

3.1.2.3 Terminated Instructions

An instruction being executed is terminated when reset occurs. Any result operands and flags due to be affected by the instruction are undefined, as is the contents of the PC.

3.1.2.4 Partially Completed Instructions

When an interrupt condition is recognized during execution of a string instruction, the instruction is said to be partially completed. A partially completed instruction has not completed, but all other instructions executed since the last exception occurred have been completed. Result operands and flags due to be affected by the instruction may have been modified, but the values stored in the string pointers and other general-purpose registers used during the instruction's execution allow the instruction to be executed again and completed.

The CPU clears the P-flag in the PSR before saving the copy that is pushed on the Interrupt Stack. The PC saved on the Interrupt Stack contains the address of the partially completed instruction. The exception service procedure can, at its conclusion, simply execute the RETT instruction (or the RETI instruction for maskable interrupts), and the CPU will resume executing the partially completed instruction.

3.1.3 Slave Processor Instructions

The NS32FX164 supports only one group of instructions, the floating-point instruction set, as being executable by a slave processor. The floating-point instruction set is validated by the F-bit in the CFG register.

If a floating-point instruction is encountered and the F-bit in the CFG register is not set, a Trap (UND) will result, without any slave processor communication attempted by the CPU. This allows software emulation in case an external floating-point unit (FPU) is not used.

3.1.3.1 Slave Processor Protocol

Slave Processor instructions have a three-byte Basic Instruction field, consisting of an ID Byte followed by an Operation Word. The ID Byte has three functions:

1. It identifies the instruction as being a Slave Processor instruction.
2. It specifies which Slave Processor will execute it.
3. It determines the format of the following Operation Word of the instruction.

Upon receiving a Slave Processor instruction, the CPU initiates the sequence outlined in *Figure 3-2*. While applying Status Code 1111 (Broadcast ID, Section 3.5.5.1), the CPU transfers the ID Byte on the least-significant half of the Data Bus (AD0–AD7). All Slave Processors input this byte and decode it. The Slave Processor selected by the ID Byte is activated, and from this point the CPU is communicating only with it. If any other slave protocol was in progress (e.g., an aborted Slave instruction), this transfer cancels it.

3.0 Functional Description (Continued)

The CPU next sends the Operation Word while applying Status Code 1101 (Transfer Slave Operand, Section 3.5.5.1). Upon receiving it, the Slave Processor decodes it, and at this point both the CPU and the Slave Processor are aware of the number of operands to be transferred and their sizes. The Operation Word is swapped on the Data Bus; that is, bits 0–7 appear on pins AD8–AD15 and bits 8–15 appear on pins AD0–AD7.

Using the Address Mode fields within the Operation Word, the CPU starts fetching operands and issuing them to the Slave Processor. To do so, it references any Addressing Mode extensions which may be appended to the Slave Processor instruction. Since the CPU is solely responsible for memory accesses, these extensions are not sent to the Slave Processor. The Status Code applied is 1101 (Transfer Slave Processor Operand, Section 3.5.5.1).

After the CPU has issued the last operand, the Slave Processor starts the actual execution of the instruction. Upon completion, it will signal the CPU by pulsing $\overline{\text{SPC}}$ low.

While the Slave Processor is executing the instruction, the CPU is free to prefetch instructions into its queue. If it fills the queue before the Slave Processor finishes, the CPU will wait, applying Status Code 0011 (Waiting for Slave).

Upon receiving the pulse on $\overline{\text{SPC}}$, the CPU uses $\overline{\text{SPC}}$ to read a Status Word from the Slave Processor, applying Status Code 1110 (Read Slave Status). This word has the format shown in *Figure 3-3*. If the Q-bit (''Quit'', Bit 0) is set, this indicates that an error was detected by the Slave Processor. The CPU will not continue the protocol, but will immediately trap through the Slave vector in the Interrupt Table. Certain Slave Processor instructions cause CPU PSR bits to be loaded from the Status Word.

Status Combinations:
Send ID (ID): Code 1111
Xfer Operand (OP): Code 1101
Read Status (ST): Code 1110

Step	Status	Action
1	ID	CPU Sends ID Byte
2	OP	CPU Sends Operation Word
3	OP	CPU Sends Required Operands
4	—	Slave Starts Execution. CPU Pre-Fetches.
5	—	Slave Pulses $\overline{\text{SPC}}$ Low
6	ST	CPU Reads Status Word. (Trap? Alter Flags?)
7	OP	CPU Reads Results (If Any).

FIGURE 3-2. Slave Processor Protocol

The last step in the protocol is for the CPU to read a result, if any, and transfer it to the destination. The Read cycles from the Slave Processor are performed by the CPU while applying Status Code 1101 (Transfer Slave Operand).

3.1.3.2 Floating-Point Instructions

Table 3-1 gives the protocols followed for each Floating-Point instruction. The instructions are referenced by their mnemonics. For the bit encodings of each instruction, see Appendix A.

TABLE 3-1. Floating-Point Instruction Protocols

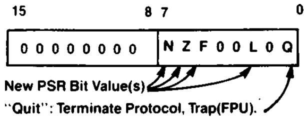
Mnemonic	Operand 1 Class	Operand 2 Class	Operand 1 Issued	Operand 2 Issued	Returned Value Type and Dest.	PSR Bits Affected
ADDf	read.f	rmw.f	f	f	f to Op.2	none
SUBf	read.f	rmw.f	f	f	f to Op.2	none
MULf	read.f	rmw.f	f	f	f to Op.2	none
DIVf	read.f	rmw.f	f	f	f to Op.2	none
MOVf	read.f	write.f	f	N/A	f to Op.2	none
ABSf	read.f	write.f	f	N/A	f to Op.2	none
NEGf	read.f	write.f	f	N/A	f to Op.2	none
CMPf	read.f	read.f	f	f	N/A	N,Z,L
FLOORfi	read.f	write.i	f	N/A	i to Op.2	none
TRUNCfi	read.f	write.i	f	N/A	i to Op.2	none
ROUNDfi	read.f	write.i	f	N/A	i to Op.2	none
MOVFL	read.F	write.L	F	N/A	L to Op.2	none
MOVLF	read.L	write.F	L	N/A	F to Op.2	none
MOVif	read.i	write.f	i	N/A	f to Op.2	none
LFSR	read.D	N/A	D	N/A	N/A	none
SFSR	N/A	write.D	N/A	N/A	D to Op. 2	none
POLYf	read.f	read.f	f	f	f to F0	none
DOTf	read.f	read.f	f	f	f to F0	none
SCALBf	read.f	rmw.f	f	f	f to Op. 2	none
LOGBf	read.f	write.f	f	N/A	f to Op. 2	none

Notes:
D = Double Word
i = Integer size (B, W, D) specified in mnemonic.
f = Floating-Point type (F, L) specified in mnemonic.
N/A = Not Applicable to this instruction.

The Operand class columns give the Access Class for each general operand, defining how the addressing modes are interpreted (see Series 32000 Instruction Set Reference Manual).

The Operand Issued columns show the sizes of the operands issued to the Floating-Point Unit by the CPU. "D" indicates a 32-bit Double Word. "i" indicates that the instruction specifies an integer size for the operand (B = Byte, W = Word, D = Double Word). "f" indicates that the instruction specifies a Floating-Point size for the operand (F = 32-bit Standard Floating, L = 64-bit Long Floating).

The Returned Value Type and Destination column gives the size of any returned value and where the CPU places it. The PSR Bits Affected column indicates which PSR bits, if any, are updated from the Slave Processor Status Word *(Figure 3-3)*.



TL/EE/11267–12

FIGURE 3-3. Slave Processor Status Word

Any operand indicated as being of type "f" will not cause a transfer if the Register addressing mode is specified. This is because the Floating-Point Registers are physically on the Floating-Point Unit and are therefore available without CPU assistance.

3.2 EXCEPTION PROCESSING

Exceptions are special events that alter the sequence of instruction execution. The CPU recognizes two basic types of exceptions: interrupts and traps.

An interrupt occurs in response to an event generated either internally, by the on-chip DSP Module, or externally, by activating $\overline{\text{NMI}}$ or $\overline{\text{INT}}$. External interrupts are typically requested by peripheral devices that require the CPU's attention.

Traps occur as a result either of exceptional conditions (e.g., attempted division by zero) or of specific instructions whose purpose is to cause a trap to occur (e.g., supervisor call instruction).

When an exception is recognized, the CPU saves the PC, PSR and optionally the MOD register contents on the interrupt stack and then it transfers control to an exception service procedure.

Details on the operations performed in the various cases by the CPU to enter and exit the exception service procedure are given in the following sections.

It is to be noted that the reset operation is not treated here as an exception. Even though, like any exception, it alters the instruction execution sequence.

The reason being that the CPU handles reset in a significantly different way than it does for exceptions.

Refer to Section 3.5.4 for details on the reset operation.

3.2.1 Exception Acknowledge Sequence

When an exception is recognized, the CPU goes through three major steps:

1. Adjustment of Registers. Depending on the source of the exception, the CPU may restore and/or adjust the contents of the Program Counter (PC), the Processor Status Register (PSR) and the currently-selected Stack Pointer (SP). A copy of the PSR is made, and the PSR is then set to reflect Supervisor Mode and selection of the Interrupt Stack. Trap (TRC) always disabled. Maskable interrupts are also disabled if the exception is caused by an interrupt.
2. Vector Acquisition. A vector is either obtained from an external interrupt control unit or is supplied internally by default.
3. Service Call. The CPU performs one of two sequences common to all exceptions to complete the acknowledge process and enter the appropriate service procedure. The selection between the two sequences depends on whether the Direct-Exception mode is disabled or enabled.

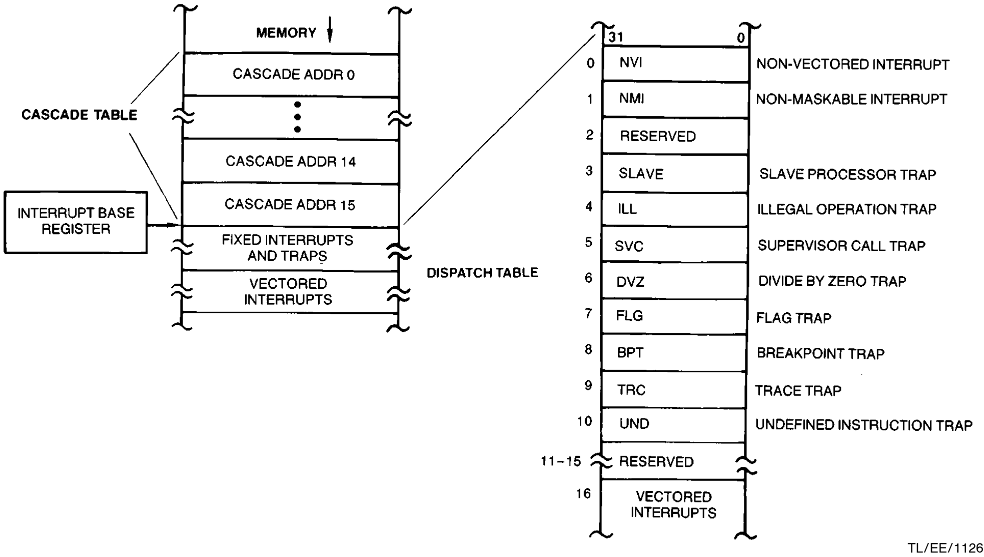
Direct-Exception Mode Disabled

The Direct-Exception mode is disabled while the DE bit in the CFG register is 0 (Section 2.1.4). In this case the CPU first pushes the saved PSR copy along with the contents of the MOD and PC registers on the interrupt stack. Then it reads the double-word entry from the Interrupt Dispatch table at address "INTBASE" + vector × 4". See *Figures 3-4* and *3-5*. The CPU uses this entry to call the exception service procedure, interpreting the entry as an external procedure descriptor.

A new module number is loaded into the MOD register from the least-significant word of the descriptor, and the static-base pointer for the new module is read from memory and loaded into the SB register. Then the program-base pointer for the new module is read from memory and added to the most-significant word of the module descriptor, which is interpreted as an unsigned value. Finally, the result is loaded into the PC register.

Direct-Exception Mode Enabled

The Direct-Exception mode is enabled when the DE bit in the CFG register is set to 1. In this case the CPU first pushes the saved PSR copy along with the contents of the PC register on the Interrupt Stack. The word stored on the Interrupt Stack between the saved PSR and PC register is reserved for future use; its contents are undefined. The CPU then reads the double-word entry from the Interrupt Dispatch Table at address "INTBASE + vector × 4". The CPU uses this entry to call the exception service procedure, interpreting the entry as an absolute address that is simply loaded into the PC register. *Figure 3-6* provides a pictorial of the acknowledge sequence. It is to be noted that while the direct-exception mode is enabled, the CPU can respond more quickly to interrupts and other exceptions because fewer memory references are required to process an exception. The MOD and SB registers, however, are not initialized before the CPU transfers control to the service procedure. Consequently, the service procedure is restricted from executing any instructions, such as CXP, that use the contents of the MOD or SB registers in effective address calculations.


MEMORY
CASCADE ADDR 0
CASCADE TABLE
CASCADE ADDR 14
CASCADE ADDR 15
INTERRUPT BASE REGISTER
FIXED INTERRUPTS AND TRAPS
VECTORED INTERRUPTS
DISPATCH TABLE
31
0
0 NVI NON-VECTORED INTERRUPT
1 NMI NON-MASKABLE INTERRUPT
2 RESERVED
3 SLAVE SLAVE PROCESSOR TRAP
4 ILL ILLEGAL OPERATION TRAP
5 SVC SUPERVISOR CALL TRAP
6 DVZ DIVIDE BY ZERO TRAP
7 FLG FLAG TRAP
8 BPT BREAKPOINT TRAP
9 TRC TRACE TRAP
10 UND UNDEFINED INSTRUCTION TRAP
11-15 RESERVED
16 VECTORED INTERRUPTS

TL/EE/11267-13

FIGURE 3-4. Interrupt Dispatch and Cascade Tables

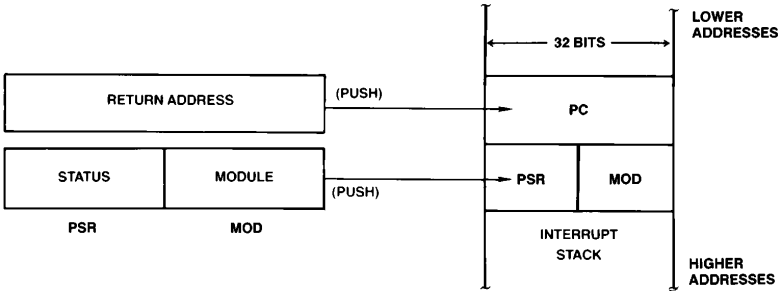
3.2.2 Returning from an Exception Service Procedure

To return control to an interrupted program, one of two instructions can be used: RETT (Return from Trap) and RETI (Return from Interrupt).

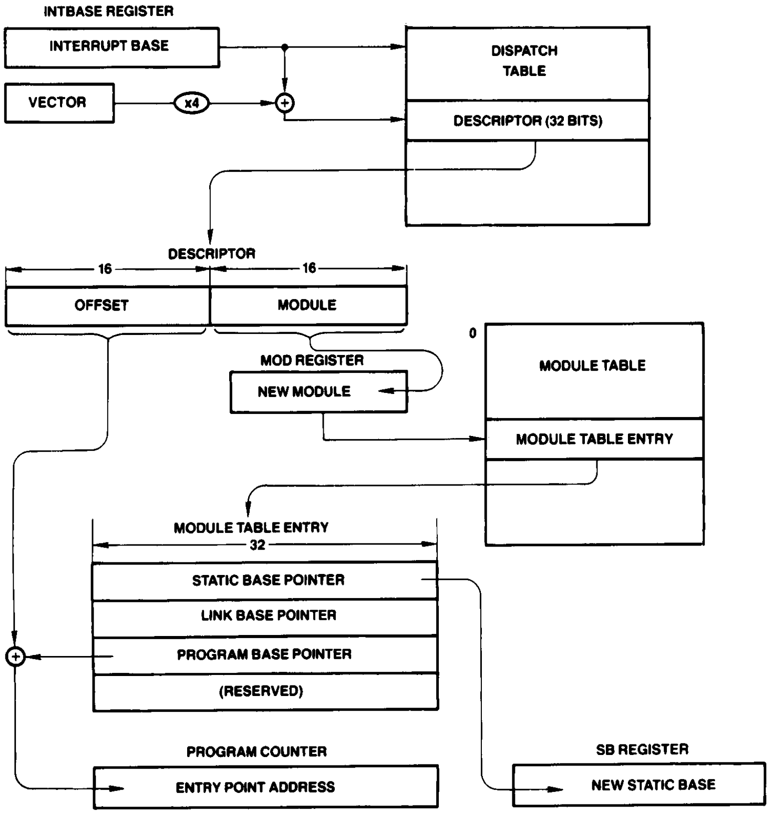
RETT is used to return from any trap or non-maskable interrupt service procedure. Since some traps are often used deliberately as a call mechanism for supervisor mode procedures, RETT can also adjust the Stack Pointer (SP) to discard a specified number of bytes from the original stack as surplus parameter space.

RETI is used to return from a maskable interrupt service procedure. A difference of RETT, RETI also informs the on-chip ICU as well as any external interrupt control logic that interrupt service has completed. Since interrupts are generally asynchronous external events, RETI does not discard parameters from the stack.

Both of the above instructions always restore the Program Counter (PC) and the Processor Status Register from the interrupt stack. If the Direct-Exception mode is disabled, they also restore the MOD and SB register contents. *Figures 3-7* and *3-8* show the RETT and RETI instruction flows when the Direct-Exception mode is disabled.



TL/EE/11267–16



TL/EE/11267–17

FIGURE 3-5. Exception Acknowledge Sequence: Direct-Exception Mode Disabled

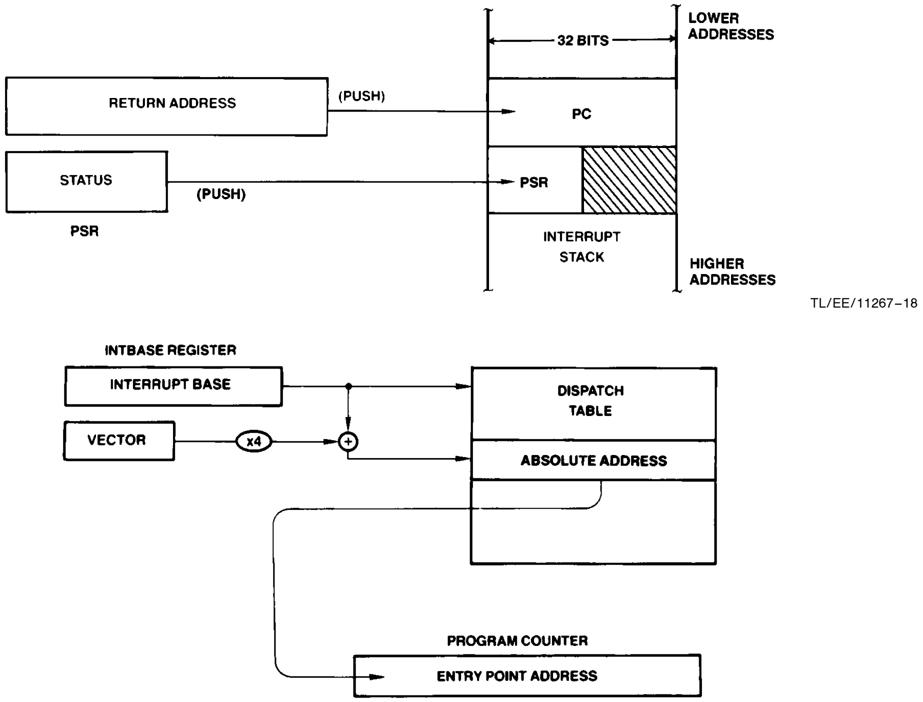
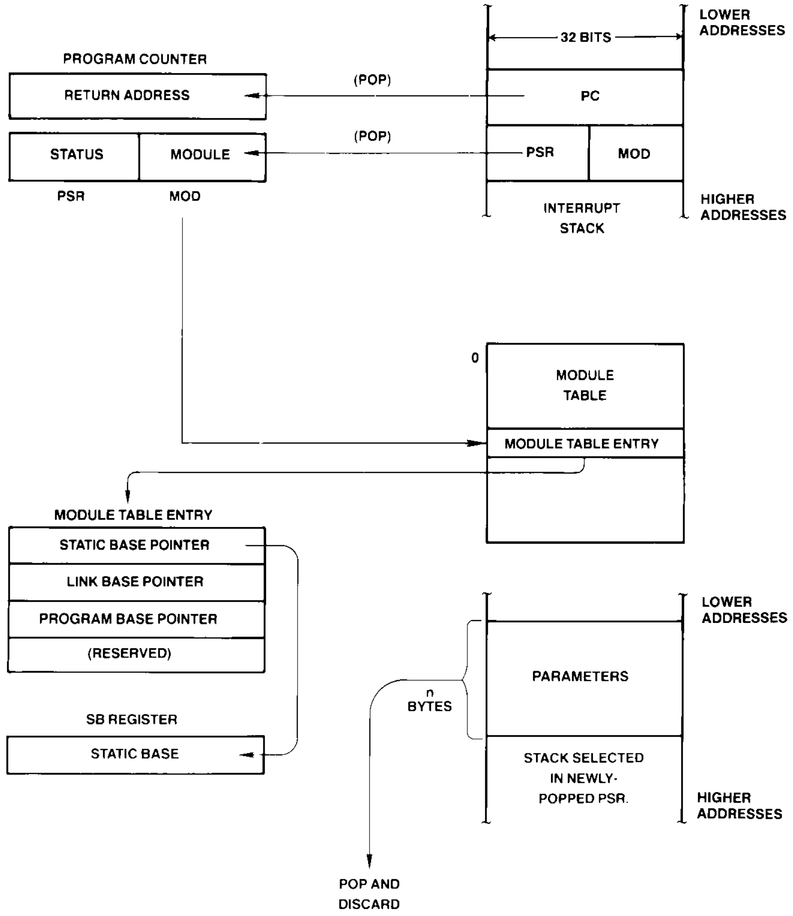
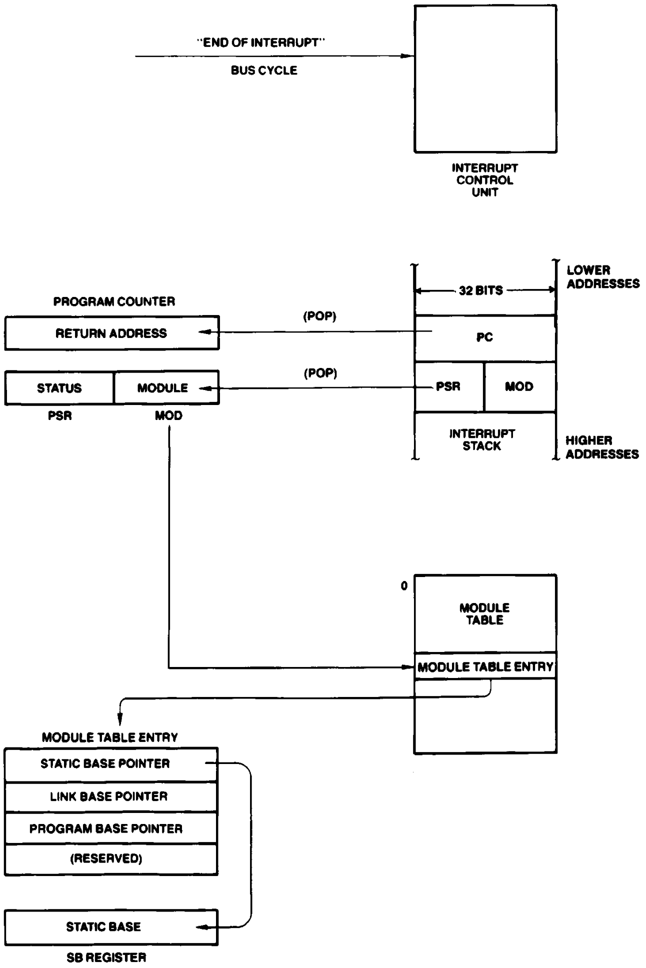


FIGURE 3-6. Exception Acknowledge Sequence: Direct-Exception Mode Enabled



TL/EE/11267–20

FIGURE 3-7. Return from Trap (RETTn) Instruction Flow: Direct-Exception Mode Disabled



TL/EE/11267–21

FIGURE 3-8. Return from Interrupt (RETI) Instruction Flow: Direct-Exception Mode Disabled

3.2.3 Maskable Interrupts

The $\overline{\text{INT}}$ pin is a level-sensitive input. A continuous low level is allowed for generating multiple interrupt requests. The input is maskable, and is therefore enabled to generate interrupt requests only while the Processor Status Register I bit is set. The I bit is automatically cleared during service of an $\overline{\text{INT}}$ or $\overline{\text{NMI}}$ request, and is restored to its original setting upon return from the interrupt service routine via the RETT or RETI instruction.

The $\overline{\text{INT}}$ pin may be configured via the SETCFG instruction as either Non-Vectored (CFG Register bit I = 0) or Vectored (bit I = 1).

3.2.3.1 Non-Vectored Mode

In the Non-Vectored mode, an interrupt request on the $\overline{\text{INT}}$ pin will cause an Interrupt Acknowledge bus cycle, but the CPU will ignore any value read from the bus and use instead a default vector of zero. This mode is useful for small systems in which hardware interrupt prioritization is unnecessary.

3.0 Functional Description (Continued)

3.2.3.2 Vectored Mode: Non-Cascaded Case

In the Vectored mode, the CPU uses an Interrupt Control Unit (ICU) to prioritize up to 16 interrupt requests. Upon receipt of an interrupt request on the $\overline{\text{INT}}$ pin, the CPU performs an "Interrupt Acknowledge, Master" bus cycle reading a vector value from the low-order byte of the Data Bus. This vector is then used as an index into the Dispatch Table in order to find the External Procedure Descriptor for the proper interrupt service procedure. The service procedure eventually returns via the Return from Interrupt (RETI) instruction, which performs an End of Interrupt bus cycle, informing the ICU that it may re-prioritize any interrupt requests still pending. The ICU provides the vector number again, which the CPU uses to determine whether it needs also to inform a Cascaded ICU.

In a system with only one ICU (16 levels of interrupt), the vectors provided must be in the range of 0 through 127; that is, they must be positive numbers in eight bits. By providing a negative vector number, an ICU flags the interrupt source as being a Cascaded ICU (see below).

Note: During a return from interrupt, the CPU looks at Bit 7 of the vector number from the master ICU. If Bit 7 is 0, bits 0 through 6 are ignored.

3.2.3.3 Vectored Mode: Cascaded Case

In order to allow up to 256 levels of interrupt, provision is made both in the CPU and in the NS32202 Interrupt Control Unit (ICU) to transparently support cascading. *Figure 3-10* shows a typical cascaded configuration. Note that the Interrupt output from a Cascaded ICU goes to an Interrupt Request input of the Master ICU, which is the only ICU which drives the CPU $\overline{\text{INT}}$ pin.

In a system which uses cascading, two tasks must be performed upon initialization:

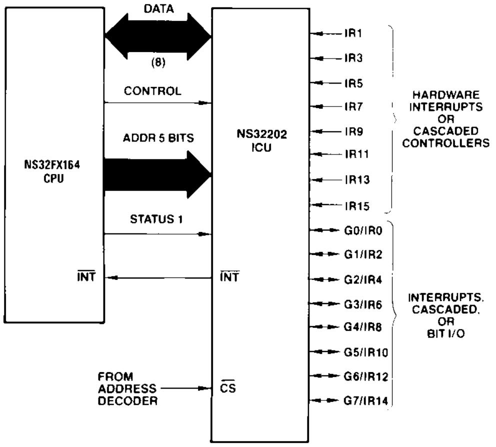
1) For each Cascaded ICU in the system, the Master ICU must be informed of the line number (0 to 15) on which it receives the cascaded requests.
2) A Cascade Table must be established in memory. The Cascade Table is located in a NEGATIVE direction from the location indicated by the CPU Interrupt Base (INTBASE) Register. Its entries are 32-bit addresses, pointing to the Vector Registers of each of up to 16 Cascaded ICUs.

Figure 3-4 illustrates the position of the Cascade Table. To find the Cascade Table entry for a Cascaded ICU, take its Master ICU line number (0 to 15) and subtract 16 from it, giving an index in the range −16 to −1. Multiply this value by 4, and add the resulting negative number to the contents of the INTBASE Register. The 32-bit entry at this address must be set to the address of the Hardware Vector Register of the Cascaded ICU. This is referred to as the "Cascade Address."

Upon receipt of an interrupt request from a Cascaded ICU, the Master ICU interrupts the CPU and provides the negative Cascade Table index instead of a (positive) vector number. The CPU, seeing the negative value, uses it as an index into the Cascade Table and reads the Cascade Address from the referenced entry. Applying this address, the CPU performs an "Interrupt Acknowledge, Cascaded" bus cycle, reading the final vector value. This vector is interpreted by the CPU as an unsigned byte, and can therefore be in the range of 0 through 255.

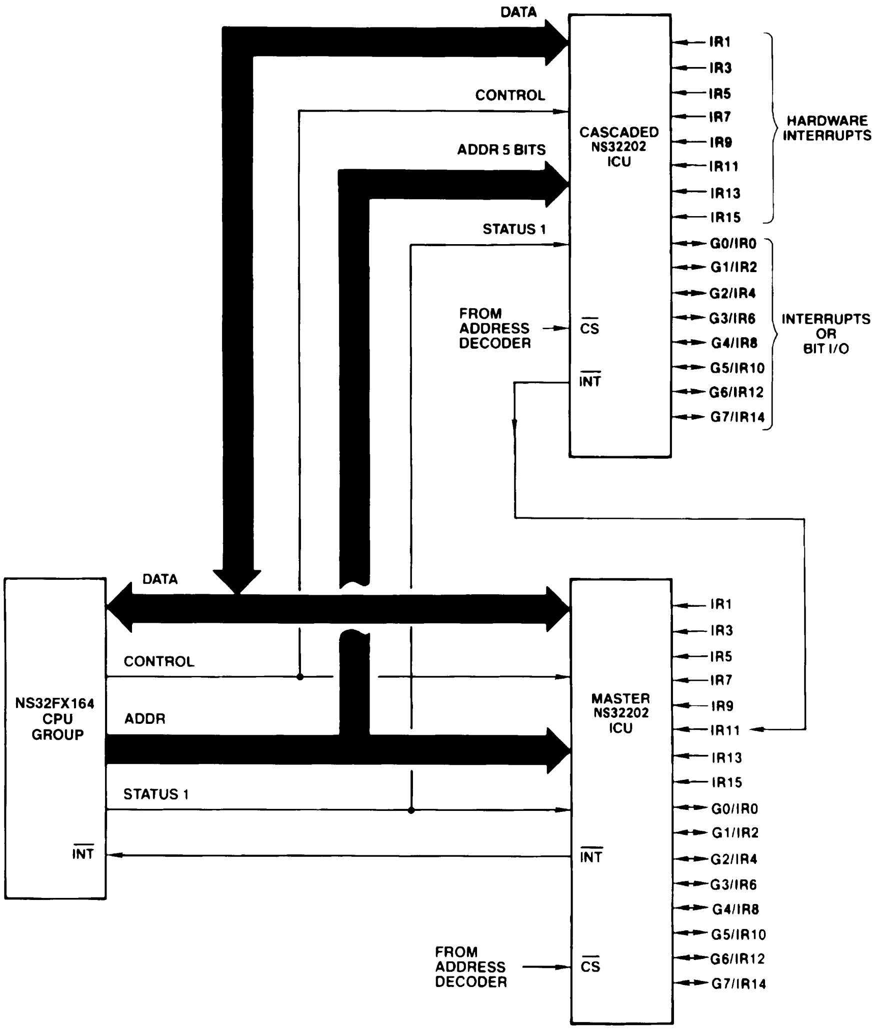
In returning from a Cascaded interrupt, the service procedure executes the Return from Interrupt (RETI) instruction, as it would for any Maskable Interrupt. The CPU performs an "End of Interrupt, Master" bus cycle, whereupon the Master ICU again provides the negative Cascaded Table index. The CPU, seeing a negative value, uses it to find the corresponding Cascade Address from the Cascade Table. Applying this address, it performs an "End of Interrupt, Cascaded" bus cycle, informing the Cascaded ICU of the completion of the service routine. The byte read from the Cascaded ICU is discarded.

Note: If an interrupt must be masked off, the CPU can do so by setting the corresponding bit in the Interrupt Mask Register of the Interrupt Controller. However, if an interrupt is set pending during the CPU instruction that masks off that interrupt, the CPU may still perform an interrupt acknowledge cycle following that instruction since it might have sampled the $\overline{\text{INT}}$ line before the ICU deasserted it. This could cause the ICU to provide an invalid vector. To avoid this problem the above operation should be performed with the CPU interrupt disabled.



TL/EE/11267–22

FIGURE 3-9. Interrupt Control Unit Connections (16 Levels)



TL/EE/11267–23

FIGURE 3-10. Cascaded Interrupt Control Unit Connections

3.2.4 Non-Maskable Interrupt

The Non-Maskable Interrupt is triggered whenever a falling edge is detected on the $\overline{\text{NMI}}$ pin. The CPU performs an "Interrupt Acknowledge" bus cycle from Address FFFF00_{16} when processing of this interrupt actually begins. The vector value used for the Non-Maskable Interrupt is taken as 1, regardless of the value read from the bus.

The service procedure returns from the Non-Maskable-Interrupt using the Return from Trap (RETT) instruction. No special bus cycles occur on return.

3.2.5 Traps

Traps are processing exceptions that are generated as direct results of the execution of an instruction.

The return address saved on the stack by any trap except Trap (TRC) is the address of the first byte of the instruction during which the trap occurred.

When a trap is recognized, maskable interrupts are not disabled.

There are 8 trap conditions recognized by the NS32FX164 as described below.

Trap (SLAVE): An exceptional condition was detected by the Floating-Point Unit during the execution of a Slave Instruction. This trap is requested via the Status Word returned as part of the Slave Processor Protocol (Section 3.1.3.1).

Trap (ILL): Illegal operation. A privileged operation was attempted while the CPU was in User Mode (PSR bit U = 1).

Trap (SVC): The Supervisor Call (SVC) instruction was executed.

Trap (DVZ): An attempt was made to divide an integer by zero. (The FPU trap is used for Floating-Point division by zero.)

Trap (FLG): The FLAG instruction detected a "1" in the PSR F-bit.

Trap (BPT): The Breakpoint (BPT) instruction was executed.

Trap (TRC): The instruction just completed is being traced. Refer to Section 3.3.1 for details.

Trap (UND): An undefined opcode was encountered by the CPU.

3.2.6 Priority among Exceptions

The CPU checks for specific exceptions at various points while executing an instruction. It is possible that several exceptions occur simultaneously. In that event, the CPU responds to the exception with highest priority.

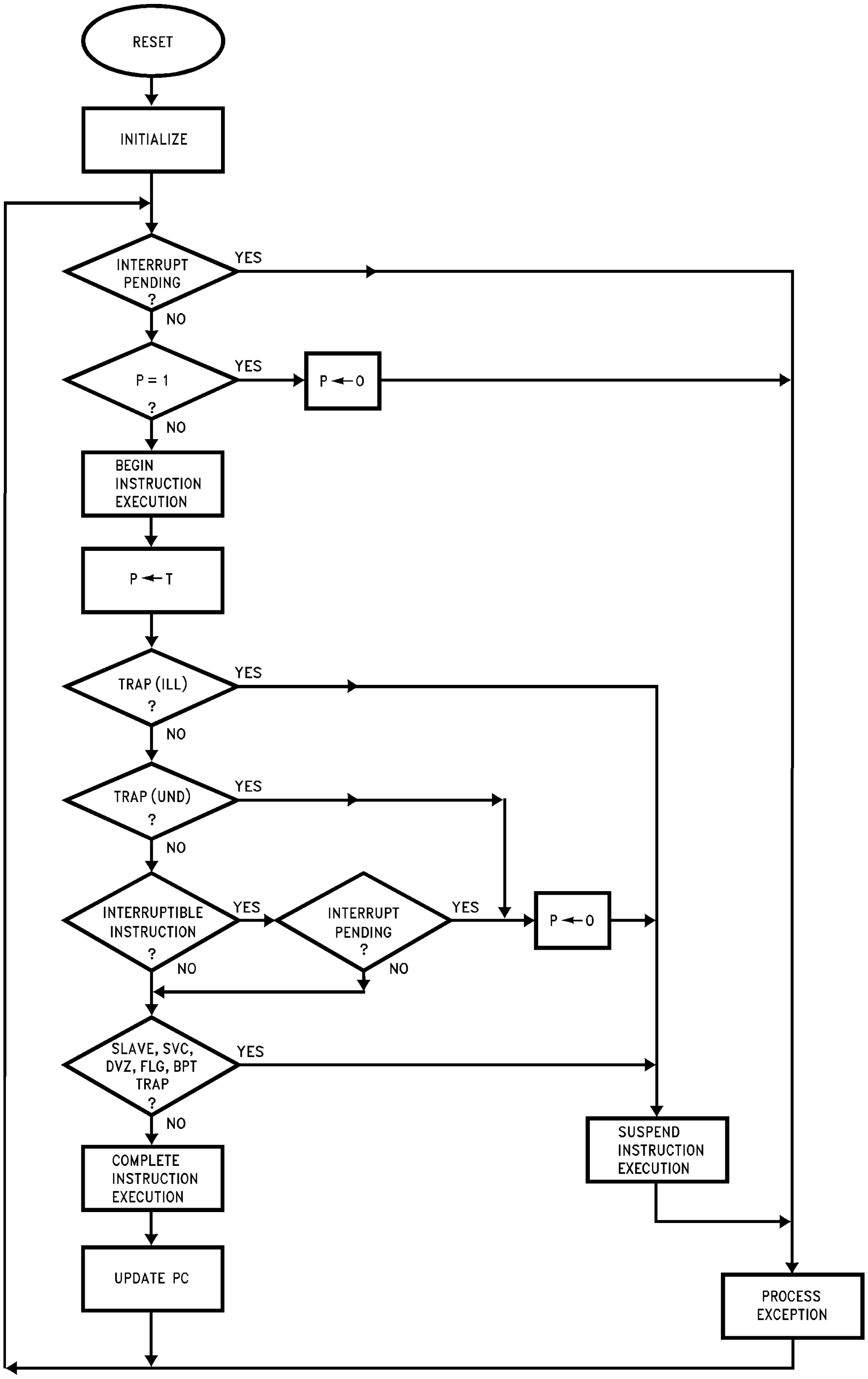
Figure 3-11 shows an exception processing flowchart.

Before executing an instruction, the CPU checks for pending interrupts, or Trap (TRC). The CPU responds to any pending interrupt requests; nonmaskable interrupts are recognized with higher priority than maskable interrupts. If no interrupts are pending, then the CPU checks the P-flag in the PSR to determine whether a Trap (TRC) is pending. If the P-flag is 1, a Trap (TRC) is processed. If no interrupt or Trap (TRC) is pending, the CPU begins executing the instruction.

While executing an instruction, the CPU may recognize up to two exceptions:

1. Interrupt, if the instruction is interruptible.
2. One of 7 mutually exclusive traps: SLAVE, ILL, SVC, DVZ, FLG, BPT, UND

If no exception is detected while the instruction is executing, then the instruction is completed and the PC is updated to point to the next instruction.



TL/EE/11267–24

FIGURE 3-11. Exception Processing Flowchart

3.2.7 Exception Acknowledge Sequences: Detailed Flow

For purposes of the following detailed discussion of exception acknowledge sequences, a single sequence called "service" is defined in *Figure 3-12*.

Upon detecting any interrupt request or trap condition, the CPU first performs a sequence dependent upon the type of exception. This sequence will include saving a copy of the Processor Status Register and establishing a vector and a return address. The CPU then performs the service sequence.

3.2.7.1 Maskable/Non-Maskable Interrupt Sequence

This sequence is performed by the CPU when the $\overline{\text{NMI}}$ pin receives a falling edge, or the $\overline{\text{INT}}$ pin becomes active with the PSR I bit set. The interrupt sequence begins either at the next instruction boundary or, in the case of the String instructions, or Graphics instructions which have interior loops (BBOR, BBXOR, BBAND, BBFOR, EXTBLT, MOVMP, SBITPS, TBITS), at the next interruptible point during its execution. The graphics instructions are interruptible.

1. If a String instruction was interrupted and not yet completed:
   a. Clear the Processor Status Register P bit.
   b. Set "Return Address" to the address of the first byte of the interrupted instruction.

   Otherwise, set "Return Address" to the address of the next instruction.
2. Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits S, U, T, P and I.
3. If the interrupt is Non-Maskable:
   a. Read a byte from address FFFF00_{16}, applying Status Code 0100 (Interrupt Acknowledge, Master: Section 3.4.1). Discard the byte read.
   b. Set "Vector" to 1.
   c. Go to Step 8.
4. If the interrupt is Non-Vectored:
   a. Read a byte from address FFFE00_{16}, applying Status Code 0100. Discard the byte read.
   b. Set "Vector" to 0.
   c. Go to Step 8.
5. Here the interrupt is Vectored. Read "Byte" from address FFFE00_{16}, applying Status Code 0100.
6. If "Byte" $\geq$ 0, then set "Vector" to "Byte" and go to Step 8.
7. If "Byte" is in the range -16 through -1, then the interrupt source is Cascaded. (More negative values are reserved for future use.) Perform the following:
   a. Read the 32-bit Cascade Address from memory. The address is calculated as INTBASE + 4* Byte.
   b. Read "Vector", applying the Cascade Address just read and Status Code 0101.
8. Perform Service (Vector, Return Address), *Figure 3-12.*

3.2.7.2 SLAVE/ILL/SVC/DVZ/FLG/BPT/UND Trap Sequence

1. Restore the currently selected Stack Pointer and the Processor Status Register to their original values at the start of the trapped instruction.
2. Set "Vector" to the value corresponding to the trap type.

SLAVE:	Vector = 3.
ILL:	Vector = 4.
SVC:	Vector = 5.
DVZ:	Vector = 6.
FLG:	Vector = 7.
BPT:	Vector = 8.
UND:	Vector = 10.

3. If Trap (UND)
   a. Clear the Processor Status Register P Bit.
4. Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits T, U, S, and P.
5. Set "Return Address" to the address of the first byte of the trapped instruction.
6. Perform Service (Vector, Return Address), *Figure 3-12*.

3.2.7.3 Trace Trap Sequence

1. In the Processor Status Register (PSR), clear the P bit.
2. Copy the PSR into a temporary register, then clear PSR bits S, U and T.
3. Set "Vector" to 9.
4. Set "Return Address" to the address of the next instruction.
5. Perform Service (Vector, Return Address), *Figure 3-12*.

Service (Vector, Return Address):

1. **Push the PSR copy onto the Interrupt Stack as a 16-bit value.**
2. **Read 32-bit Interrupt Dispatch Table (IDT) entry at address "INTBASE + vector × 4".**
3. **If Direct-Exception mode is selected, then go to Step 10.**
4. **Move the LS word of the IDT entry (Module Field) into the temporary MOD register.**
5. **Read the Program Base pointer from memory address "MOD + 8", and add to it the M.S. word of the IDT entry (Offset Field), placing the result in the Program Counter.**
6. **Read the new Static Base pointer from the memory address contained in MOD, placing it into the SB Register.**
7. **Push MOD Register into the Interrupt Stack as a 16-bit value.**
8. **Copy temporary MOD Register into MOD Register.**
9. **Go to Step 11.**
10. **Place IDT entry in the Program Counter.**
11. **Push the Return Address onto the Interrupt Stack as a 32-bit quantity.**
12. **Flush queue: Non-sequentially fetch first instruction of Exception Service Routine.**

FIGURE 3-12. Service Sequence
Invoked during All Interrupt/Trap Sequences

TABLE 3-2. Summary of Exception Processing

Exception	Instruction Ending	Cleared before Saving PSR	Cleared after Saving PSR
Interrupt	Before Instruction	None /P*	TUSPI
UND	Suspended	P	TUS
SLAVE, SVC, DVZ, FLG, BPT, ILL	Suspended	None	TUSP
TRC	Before Instruction	P	TUS

3.3 DEBUGGING SUPPORT

The NS32FX164 provides features to assist in program debugging.

Besides the Breakpoint (BPT) instruction that can be used to generate soft breaks, the CPU also provides the instruction tracing capability.

3.3.1 Instruction Tracing

Instruction tracing is a very useful feature that can be used during debugging to single-step through selected portions of a program. Tracing is enabled by setting the T-bit in the PSR Register. When enabled, the CPU generates a Trace Trap (TRC) after the execution of each instruction.

At the beginning of each instruction, the T-bit is copied into the PSR P (Trace "Pending") bit. If the P-bit is set at the end of an instruction, then the Trace Trap is activated. If any other trap or interrupt request is made during a traced instruction, its entire service procedure is allowed to complete before the Trace Trap occurs. Each interrupt and trap sequence handles the P-bit for proper tracing, guaranteeing only one Trace Trap per instruction, and guaranteeing that the Return Address pushed during a Trace Trap is always the address of the next instruction to be traced.

The beginning of the execution of a TRAP(UND) is not considered to be a beginning of an instruction, and hence the T-bit is not copied into the P-bit.

Due to the fact that some instructions can clear the T- and P-bits in the PSR, in some cases a Trace Trap may not occur at the end of the instruction. This happens when one of the privileged instructions BICPSRW or LPRW PSR is executed.

In other cases, it is still possible to guarantee that a Trace Trap occurs at the end of the instruction, provided that special care is taken before returning from the Trace Trap Service Procedure. In case a BICPSRB instruction has been executed, the service procedure should make sure that the T-bit in the PSR copy saved on the Interrupt Stack is set before executing the RETT instruction to return to the program being traced. If the RETT or RETI instructions have to be traced, the Trace Trap Service Procedure should set the P- and T-bits in the PSR copy on the Interrupt Stack that is going to be restored in the execution of such instructions.

While debugging the NS32FX164 instructions which have interior loops (BBOR, BBXOR, BBAND, BBFOR, EXTBLT, MOVMP, SBITPS, TBITS), special care must be taken with the single-step trap. If an interrupt occurs during a single-step of one of the graphics instructions, the interrupt will be serviced. Upon return from the interrupt service routine, the new NS32FX164 instruction will not be re-entered, due to a single-step trap. Both the NMI and INT interrupts will cause this behavior. Another single-step operation (S command in DBG16/MONCG) will resume from where the instruction was interrupted. There are no side effects from this early termination, and the instruction will complete normally.

For all other Series 32000 instructions, a single-step operation will complete the entire instruction before traping back to the debugger. On the instructions mentioned above, several single-step commands may be required to complete the instruction, ONLY when interrupts are occurring.

There are some methods to give the appearance of single-stepping for these NS32FX164 instructions.

1. MON16/MONCG monitors the return from single-step trap vector, PC value. If the PC has not changed since the last single-step command was issued, the single-step operation is repeated. It is also advisable to ensure that one of the NS32FX164 instructions is being single-stepped, by inspecting the first byte of the address pointed to by the PC register. If it is 0x0E, then the instruction is an NS32FX164-specific instruction.
2. A breakpoint following the instruction would also trap after the instruction had completed.

Note: If instruction tracing is enabled while the WAIT instructioin is executed, the Trap (TRC) occurs after the next interrupt, when the interrupt service procedure has returned.

3.4 DSP MODULE

The following sections give full specifications for the 32FX164 on-chip DSP Module.

3.4.1 Programming Model

The DSPM programming model consists of the following elements:

- Internal RAM
- Dedicated registers
- Command-list execution unit
- Interface with CPU core
- Vector instruction set

The Internal RAM is used by the DSPM for fetching commands to be executed, and for reading or writing data that is needed in the course of program execution. DSPM Programs are encoded as command lists and are interpreted by the command-list execution unit.

Computations are performed by commands selected from the set of available ones. These commands employ the DSP-oriented datapath in a pipelined manner, thus maximizing the utilization of on-chip hardware resources. A set of dedicated registers is used to specify operands and options for subsequent vector commands. These dedicated registers can be loaded and stored by appropriate commands in between initiations of vector commands. Additional commands are available for controlling the flow of execution of the command list, as needed for programming loops and branches (see Section 3.4.5.7).

The CPU core interface specifies the mapping of the DSPM internal RAM as a contiguous block within the CPU core's address space, thus making it possible for normal CPU instructions to access and manipulate data and commands in the DSPM internal RAM (see Section 3.4.4.2). In addition, the CPU core interface contains control and status registers that are needed to synchronize the execution of CPU core instructions concurrently with execution of the DSPM command lists (see Section 3.4.4.1).

3.4.2 RAM Organization and Data Types

The DSPM internal RAM is organized as a word or double-word addressable, uniform, linear address space. Memory locations are numbered sequentially, starting at 0 for the first location, and incremented by 1 for each successive location. The content of each memory location is a 16-bit word. Double-words must be aligned to an even address. Valid RAM addresses for access by the command-list execution unit are 0 through 0x7FF. Access to memory locations out of the DSMP RAM boundary are not allowed.

The organization of the DSPM internal RAM is shown below:

15	**0**
Location 0	
Location 1	
. . .	
Location *n*	
. . .	

The RAM array is not restricted to use by the DSPM, it can also be accessed by the core with any type of memory access (e.g., byte, word, or double-word accesses aligned to any byte address).

The internal RAM stores command lists to be executed, and data to be manipulated during program execution. Command lists consist of 16-bit commands, so that each individual command occupies one memory location.

Each data item is represented as having either a 16-bit or a 32-bit value, as follows:

- Integer values (16-bit)
- Aligned-integer values (32-bit)
- Real values (16-bit)
- Aligned-real values (32-bit)
- Extended-precision real values (32-bit)
- Complex values (32-bit)

3.4.2.1 Integer Values

Integer values are represented as signed 16-bit binary numbers in 2's complement format. The range of integer values is from -2^{15} (-32768) through $2^{15} - 1$ (32767). Bit 0 is the Least Significant Bit (LSB), and bit 15 is the Most Significant Bit (MSB).

15	**0**
Integer Value	

Integer values are typically used for addressing vector operands and for lookup-table index manipulations.

3.4.2.2 Aligned-Integer Values

Aligned-integer values are represented as pairs of integer values, and must be aligned on a double-word boundary. The less significant half represents one integer vector element, and must be contained in an even-numbered memory location. The more significant half represents the next vector element, and must be contained in the next (odd-numbered) memory location.

15	**0**	
Integer Value (Low)		(Location 2*n*)
Integer Value (High)		(Location 2*n* + 1)

Aligned-integer values are used for higher throughput in operations where two sequential integer vector elements can be used in a single iteration. Both elements of an aligned-integer value have the same range and accuracy as specified for integer values above.

3.4.2.3 Real Values

Real values are represented as 16-bit signed fixed-point fractional numbers, in 2's complement format. Bit 15 (MSB) is the sign bit. Bits 0 (LSB) through 14 represent the fractional part. The binary digit is assumed to lie between bits 14 and 15.

15	**0**
Real Value	

Real values are used to represent samples of analog signals, coefficients of filters, energy levels, and similar continuous quantities that can be represented using 16-bit accuracy. The range of real values is from -1.0 (represented as **0x8000**) through $1.0 - 2^{-15}$ (represented as **0x7FFF**).

3.4.2.4 Aligned-Real Values

Aligned-real values are represented as pairs of real values, and they must be aligned on a double-word boundary. The less significant half represents one real vector element, and must be contained in an even-numbered memory location. The more significant half represents the next vector element, and must be contained in the next (odd-numbered) memory location.

15	**0**	
Real Value (Low)		(Location 2*n*)
Real Value (High)		(Location 2*n* + 1)

Aligned-real values are used for higher throughput in operations where two sequential real vector elements can be used in a single iteration. Both elements of an aligned-real value have the same range and accuracy as specified for real values above.

3.4.2.5 Extended-Precision Real Values

Extended-precision real values are represented as 32-bit signed fixed-point fractional numbers, in 2's complement format. Extended-precision real values must be aligned on a double-word boundary, so that the less significant half is contained in an even-numbered memory location, and the more significant half is contained in the next (odd-numbered) memory location. Bit 15 (MSB) of the more significant part is the sign bit. Bits from 0 (LSB) of the less significant part, through 14 of the more significant part, are used to represent the fractional part. The binary digit is assumed to lie between bits 14 and 15 of the more significant part. When extended-precision values are loaded or stored in the accumulator, bits 1 through 31 of the extended-precision argument are loaded or stored in bits 0 through 30 of the accumulator. Bit 0 of the extended-precision argument is not used during calculations. This bit is always set to "0" when stored back in the internal memory.

15 0	
Less Significant Part	(Location 2*n*)
More Significant Part	(Location 2*n* + 1)

Extended-precision real values are used to represent various continuous quantities that require high accuracy. The range of extended-precision real values is from −1.0 (represented as **0x80000000**) through 1.0 -2^{-30} (represented as **0x7FFFFFFE**).

3.4.2.6 Complex Values

Complex values are represented as pairs of real values, and must be aligned on a double-word boundary. The less significant half represents the real part, and must be contained in an even-numbered memory location. The more significant half represents the imaginary part, and must be contained in the next (odd-numbered) memory location.

15 0	
Real Part	(Location 2*n*)
Imaginary Part	(Location 2*n* + 1)

Complex values are used to represent samples of complex baseband signals, constellation points in the complex plane, coefficients of complex filters, and rotation angles as points on the unit circle, etc. Both the real and imaginary parts have the same range and accuracy as specified for real values above.

3.4.3 Command List Format

All commands have the same fixed format, consisting of a 5-bit *opcode* field and a 11-bit *arg* field, as shown below:

15 11	10 0
opcode	*arg*

The *opcode* field specifies an operation to be performed. The *arg* field interpretation is determined by the class to which the command belongs. There are several classes of commands, as follows:

- Load Register Instructions
- Store Register Instructions
- Adjust Register Instructions
- Flow Control Instructions
- Internal Memory Move Instructions
- External Memory Move Instructions
- Arithmetic/Logical Instructions
- Multiply-and-Accumulate Instructions
- Multiply-and-Add Instructions
- Clipping and Min/Max Instructions
- Special Instructions

See Section 3.4.5 for detailed information on the DSPM instruction set.

3.4.4 CPU Core Interface

The interface between the DSPM and the CPU core consists of the following elements:

- Parallel Operation and Synchronization
- CPU Core Address Space Map
- External Memory References

3.4.4.1 Synchronization of Parallel Operation

Since the DSPM is capable of autonomous operation parallel to the CPU core operation, a mechanism is needed to synchronize the two threads of execution. The parallel synchronization mechanism consists of several control and status registers, which are used to synchronize the following activities:

- Initiation of the command list execution
- Termination of the command list execution
- Check the DSPM status
- Access to DSPM internal RAM and registers by CPU core instructions
- Access to external memory by DSPM commands

The following CPU core interface control and status registers are available:

Register	Function
CLPTR	Command-List Pointer
CLSTAT	Command-List Status Register
ABORT	Abort Register
EXT	Disable External Memory References
DSPINT	Interrupt Register
DSPMASK	Mask Register
NMISTAT	NMI Status Register

Execution of the command list begins when the CPU core writes a value into the CLPTR control register. This causes the DSPM command-list execution unit to begin executing commands, starting at the address written to the CLPTR register. If the written value is outside the range of valid RAM addresses, the result is unpredictable.

Once started, execution of the command list continues until one of the following occurs: a HALT or a DBPT command is executed, the CPU core writes any value into the ABORT control register, an attempt to execute a reserved command, an attempt to access the DSPM address space while the CLSTAT.RUN bit is "1" (except for accesses to the CLSTAT, EXT, DSPINT, DSPMASK, NMISTAT, and ABORT registers), or reset occurs. In the last case, the contents of the DSPM internal RAM, REPEAT, and CLPTR registers are unpredictable when execution terminates.

The CLSTAT status register can be read by CPU core instructions to check whether execution of the DSPM command list is active or idle. A "0" value read from the CLSTAT.RUN bit indicates that execution is idle, and a "1" value indicates that it is active.

Whenever the execution of the command list terminates, CLSTAT.RUN changes its value from "1" to "0", and DSPINT.HALT is set to "1". The value of the DSPINT.HALT status bit can be used to generate interrupts. If DSPMASK.HALT is set, a "1" value on the DSPINT.HALT will cause the $\overline{\text{IOUT}}$ output signal to become active (low). $\overline{\text{IOUT}}$ can be connected to an external Interrupt Controller Unit (ICU), or directly to the $\overline{\text{INT}}$ input of the NS32FX164.

The DSPM internal RAM and the dedicated registers, as well as the interface control and status registers, are mapped into certain areas of the CPU core address space (see Section 2.2.1). Whenever execution of the DSPM command list is idle, CPU core instructions may access these memory areas for any purpose, exactly as they would access external off-chip memory locations. However, when the DSPM command list execution unit is active, any attempt to read or write a location within the above memory areas, except for accessing the CLSTAT, EXT, DSPMASK, DSPINT, NMISTAT, or ABORT control registers (see below), will be treated as follows: All read data will have unpredictable values, and any attempt to write data will not change the DSPM memory and registers. Whenever such an access occurs, NMISTAT.ERR bit is set to "1", an NMI request to the core is issued, and the command list execution terminates. In this case, as the command-list execution terminates asyncronously, the currently executed command may be aborted. The DSPM RAM and the A, X, Y, Z, and REPEAT registers may hold temporary values created in this aborted instruction.

Some of the vector instructions executable by the DSPM can access external off-chip memory to transfer data in or out of the internal RAM, or to reference large lookup tables. Normally, external memory references initiated by the DSPM and CPU core are interleaved by the CPU core bus-arbitration logic. As a result, it is the user's responsibility, to make sure that whenever a write operation is involved, the DSPM and CPU core should not reference the same external memory locations, since the order of these transactions is unpredictable.

Each time the DSPM needs to access the external bus, it issues an internal HOLD request to the CPU core, and waits for an internal HOLD acknowledge. External HOLD requests (when the $\overline{\text{HOLD}}$ signal is asserted) have higher priority than DSPM HOLD requests.

In order to ensure fast response for time-critical interrupt requests, the DSPM external referencing mechanism will relinquish the core bus for one clock cycle after each memory transaction. This allows the core to use the bus for one memory transaction. To further enhance the core speed on critical interrupt routines, the EXT.HOLD control flag is provided.

Whenever the core sets EXT.HOLD to "1", the DSPM stops its external memory references. When the DSPM needs to perform an external memory reference but is disabled, it enters a HOLD state until a value of "0" is written to the EXT.HOLD control register.

3.4.4.2 DSPM RAM Organization

The mapping of these locations to CPU core address space is shown below, where *base* corresponds to the start of the mapped area (address **0xFFFE0000**):

15 8	7 0	
base + 1	*base* + 0	(RAM Location 0)
base + 3	*base* + 2	(RAM Location 1)
. . .	. . .	
base + 2*n* + 1	*base* + 2*n*	(RAM Location *n*)
. . .	. . .	

The RAM array is not restricted to use by the DSPM, but can also be used by the core as a fast, zero wait-state, on-chip memory for instructions and data storage. The core can access each byte, word, or double-word of the RAM, with no restrictions on alignment.

3.4.5 DSPM Instruction Set

3.4.5.1 Conventions

The formal description below of DSPM command-list instructions is based on the "C" programming language, using the following conventions:

low Bits 0 through 15 of a 32 bits entity.

high Bits 16 through 31 of a 32 bits entity.

LENG Value of PARAM.LENGTH.

A Accumulator.

aligned__addr An even number in the range [0, 2^{16}], used for specifying a double word-aligned address in internal memory.

mem[*k*] A value in internal memory whose first word address is *k*, where $0 \leq k < 2^{16}$.

ext__mem[*k*] A value in external memory whose first byte address is *k*, where $0 \leq k < 2^{32}$.

X Vector in internal memory whose first address is pointed to by X.ADDR.

Y Vector in internal memory whose first address is pointed to by Y.ADDR.

Z Vector in internal memory whose first address is pointed to by Z.ADDR.

X[*n*] A value in internal memory whose address is formed by adding an offset to a cyclic buffer base address. The base address is formed by clearing the (X.WRAP − 1) less-significant bits of X.ADDR. The offset within the buffer is calculated by: $(X.ADDR + n \times 2^{X.INCR})$ modulo $2^{X.WRAP}$.

Y[*n*] A value in internal memory whose address is formed by adding an offset to a cyclic buffer base address. The base address is formed by clearing the (Y.WRAP − 1) less-significant bits of Y.ADDR. The offset within the buffer is calculated by: $(Y.ADDR + n \times 2^{Y.INCR})$ modulo $2^{Y.WRAP}$.

Z[*n*] A value in internal memory whose address is formed by adding an offset to a cyclic buffer base address. The base address is formed by clearing the (Z.WRAP − 1) less-significant bits of Z.ADDR. The offset within the buffer is calculated by: $(Z.ADDR + n \times 2^{Z.INCR})$ modulo $2^{Z.WRAP}$.

&X[*n*] The word address of X[*n*].

&Y[*n*] The word address of Y[*n*].

&Z[*n*] The word address of Z[*n*].

3.4.5.2 Type Casting

The following data type definitions are used in DSPM instruction description:

integer An integer value, as described in Section 3.4.2.1.

aligned__integer An aligned integer value, as described in Section 3.4.2.2.

real A real value, as described in Section 3.4.2.3.

aligned__real	An aligned real value, as described in Section 3.4.2.4.
extended	An extended-precision real value, as described in Section 3.4.2.5.
complex	A complex value, as described in Section 3.4.2.6.
vector__ptr	A valid value for X, Y, and Z registers.
repeat__reg	A valid value for REPEAT register.
param__reg	A valid value for PARAM register.
eabr__reg	A valid value for EABR register.
real__acc	A 34-bit value inside either the real part or the imaginary part of the accumulator.
complex__acc	A 68-bit value inside the complex accumulator.

3.4.5.3 General Notes

The values of the EABR, PARAM, X, Y, and Z registers are not changed by the execution of the command list.

Some instructions use the accumulator as a temporary register and therefore destroy its contents. In general, the user should assume that the contents of the accumulator are unpredictable after an instruction terminates, unless stated otherwise in the notes section following that instruction's formal specification.

Non-complex instructions that use the accumulator, can use either the real or the imaginary parts, or both. In general, when an integer or real data type is to be read, it is taken from the real part. An extended-precision real data type is taken from the imaginary part. When a non-complex data type is loaded into the accumulator (by the LEA instruction or within other instructions prior to saving it into memory), it is written to both real and imaginary parts.

Rounding is implemented by copying PARAM.RND into bit position 14 of both the real and the imaginary part of the accumulator, performing the requested operation, and truncating the contents of the accumulator upon storing results to memory. In Multiply-and-Add instructions and some of the special instructions, this is done transparently on each vector element iteration. In Multiply-and-Accumulate instructions, when PARAM.CLR is "0", the previous content of the accumulator is used, so that rounding control is actually performed when the accumulator is first loaded and not when the multiply operations is executed. On the other hand, if PARAM.CLR is "1", the PARAM.RND value is copied into bit 14 of the cleared accumulator, so that rounding control is done at the same time that the multiply operation is executed.

Rounding is performed only for real, aligned-real and complex data types. In operations on complex operands, the order of accumulation is as follows: the result of the multiplication with the real part of the X operand is added first to the accumulator, and only then the result of the multiplication with the imaginary part of the X operand is added.

In general, the X, Y, and Z vectors can overlap. However, because of the pipelined structure of the DSPM datapath, the user must verify that a value written into the DSPM internal memory will not be used in the same vector instruction as a source operand for the next 8 iterations, in all instructions except VCPOLY. In VCPOLY, Y[0] cannot be over-ridden at all.

The description below specifies the encoding of each DSPM instruction. All other values are reserved for future use. Any attempt to execute any reserved instructions will terminate execution of the command list, issue an NMI request, and set NMISTAT.UND to "1". In this case the contents of the EXT and DSPMASK remain unchanged, but the contents of the Accumulator and OVF may change.

3.4.5.4 Load Register Instructions

LX—Load X Vector Pointer

The LX instruction loads the double-word at *aligned__addr* into the X register.

Syntax:

LX *aligned__addr*

15	**11**	**10**	**0**
00010		*aligned__addr*	

Operation:

```
{
  X = (vector_ptr) mem[aligned_addr];
}
```

Notes: The value at mem[*aligned__addr*] should conform to vector pointer specification format.

Accumulator is not affected.

LY—Load Y Vector Pointer

The LY instruction loads the double-word at *aligned__addr* into the Y register.

Syntax:

LY *aligned__addr*

15	**11**	**10**	**0**
00011		*aligned__addr*	

Operation:

```
{
  Y = (vector_ptr) mem[aligned_addr];
}
```

Notes: The value at mem[*aligned__addr*] should conform to vector pointer specification format.

Accumulator is not affected.

LZ—Load Z Vector Pointer

The LZ instruction loads the double-word at *aligned__addr* into the Z register.

Syntax:

LZ *aligned__addr*

15	**11**	**10**	**0**
00100		*aligned__addr*	

Operation:

```
{
  Z = (vector_ptr) mem[aligned_addr];
}
```

Notes: The value at mem[*aligned__addr*] should conform to vector pointer specification format.

Accumulator is not affected.

LA—Load Accumulator

The LA instruction loads the complex value at *aligned__addr* into the A accumulator as a complex value.

Syntax:

LA *aligned__addr*

15 11	10 0
00101	*aligned__addr*

Operation:

```
{
  (complex) A = (complex) mem[aligned_addr];
}
```

Notes: The real and imaginary parts are placed in bits 15 through 30 of the real and imaginary parts of the accumulator.

When PARAM.RND is set to "1", bit 14 of the real and imaginary parts is set to "1", in order to implement rounding upon subsequent additions into the accumulator. Otherwise, it is cleared to "0".

LEA—Load Extended Accumulator

The LEA instruction loads the accumulator with the extended value specified by X[0].

Both the real and the imaginary parts of the accumulator are loaded.

Syntax:

EXEC LEA

15 11	10 0
10000	101 0011 0011

Operation:

```
{
  extended X;
  A = (extended) X[0];
}
```

Note: Bits 1 through 31 of the memory location are read into bit positions 0 through 30 of the accumulator.

LPARAM—Load Parameters Register

The LPARAM instruction loads the double-word at *aligned__addr* into the PARAM register.

Syntax:

LPARAM *aligned__addr*

15 11	10 0
00000	*aligned__addr*

Operation:

```
{
  PARAM = (param_reg) mem[aligned_addr];
}
```

Notes: The value at mem[*aligned__addr*] should conform to this register format. The value written into PARAM.LENGTH must be greater then 0.

Accumulator is not affected.

LREPEAT—Load Repeat Register

The LREPEAT instruction loads the double-word at *aligned__addr* into the REPEAT register.

Syntax:

LREPEAT *aligned__addr*

15 11	10 0
00110	*aligned__addr*

Operation:

```
{
  REPEAT = (repeat_reg) mem[aligned_addr];
}
```

Notes: The value at mem[*aligned__addr*] should conform to the REPEAT register format.

Accumulator is not affected.

LEABR—Load External Address Base Register

The LEABR instruction loads the double-word at mem[*aligned__addr*] into the EABR register.

Syntax:

LEABR *aligned__addr*

15 11	10 0
00111	*aligned__addr*

Operation:

```
{
  EABR = (eabr_reg) mem[aligned_addr];
}
```

Notes: The value at mem[*aligned__addr*] should conform to vector pointer specification format, that is, bit positions 0 through 16 must be specified as "0'.

Accumulator is not affected.

3.4.5.5 Store Register Instructions

SX—Store X Vector Pointer

The SX instruction stores the contents of the X register into the double-word at *aligned__addr*.

Syntax:

SX *aligned__addr*

15 11	10 0
01010	*aligned__addr*

Operation:

```
{
  (vector_ptr) mem[aligned_addr] = X;
}
```

Note: Accumulator is not affected.

SXL—Store X Vector Pointer Lower Half

The SXL instruction stores the contents of the lower-half of the X register into the word at mem[*addr*].

Syntax:

SXL *addr*

15 11	10 0
11100	*addr*

Operation:

```
{
  mem[aligned_addr] = X.low;
}
```

Note: Accumulator is not affected.

SXH—Store X Vector Pointer Higher Half

The SXH instruction stores the contents of the higher-half of the X register into the word at mem[*addr*].

Syntax:

SXH *addr*

15 11	10 0
11101	*addr*

Operation:

```
{
  mem[aligned_addr] = X.high;
}
```

Note: Accumulator is not affected.

SY—Store Y Vector Pointer

The SY instruction stores the contents of the Y register into the double-word at *aligned__addr.*

Syntax:

SY *aligned__addr*

15 11	10 0
01011	*aligned__addr*

Operation:

```
{
  (vector_ptr) mem[aligned_addr] = Y;
}
```

Note: Accumulator is not affected.

SZ—Store Z Vector Pointer

The SZ instruction stores the contents of the Z register into the double-word at *aligned__addr*.

Syntax:

SZ *aligned__addr*

15 11	10 0
01100	*aligned__addr*

Operation:

```
{
  (vector_pointer mem[aligned_addr] = Z;
}
```

Note: Accumulator is not affected.

SA—Store Accumulator

The SA instruction stores the contents of the A accumulator as a complex value into mem[*aligned__addr*].

Syntax:

SA *aligned__addr*

15 11	10 0
01101	*aligned__addr*

Operation:

```
{
  (complex mem[aligned_addr] = (complex) A;
}
```

Notes: Bits 15 through 30 of the real and imaginary parts of the accumulator are placed in the real and imaginary parts of the complex value at mem[*aligned__addr*].

Accumulator is not affected.

SEA—Store Extended Accumulator

The SEA stores the contents of bits 0–30 of the imaginary accumulator as an extended value into a DSPM memory location specified by Z[0].

Bit 0 of this memory location is loaded with "0".

Syntax:

EXEC SEA

15 11	10 0
10000	101 0011 0110

Operation:

```
{
  extended Z;
Z[0] = (extended) A;
}
```

Note: Accumulator is not affected.

SREPEAT—Store Repeat Register

The SREPEAT instruction stores the contents of the REPEAT register in the double-word at mem[*aligned__addr*].

Syntax:

SREPEAT *aligned__addr*

15 11	10 0
01110	*aligned__addr*

Operation:

```
{
  (repeat_reg) mem[aligned_addr] = REPEAT;
}
```

Note: Accumulator is not affected.

SOVF—Store and Clear OVF Register

The SOVF instruction stores the contents of the OVF register in the word at mem[*addr*]. The OVF register is then cleared to "0".

Syntax:

SOVF *addr*

15 11	10 0
01001	*addr*

Operation:

```
{
  (ovf_reg) mem[aligned_addr] = OVF;
}
```

Note: Accumulator is not affected.

3.4.5.6 Adjust Register Instructions

INCX—Increment X Vector Pointer

The INCX instruction increments the X vector pointer by one element, according to the *increment* and the *wrap*.

Syntax:

EXEC INCX

15 11	10 0
10000	100 0101 1001

Operation:

```
{
  X.ADDR = &X[1];
}
```

Note: Accumulator is not affected.

INCY—Increment Y Vector Pointer

The INCY instruction increments the Y vector pointer by one element, according to the *increment* and the *wrap*.

Syntax:

EXEC INCY

15 11	10 0
10000	100 0101 1011

Operation:

```
{
  Y.ADDR = &Y[1];
}
```

Note: Accumulator is not affected.

INCZ—Increment Z Vector Pointer

The INCZ instruction increments the Z vector pointer by one element, according to the *increment* and the *wrap*.

Syntax:

EXEC INCZ

15 11	10 0
10000	100 0101 1101

Operation:

```
{
Z.ADDR = &Z[1];
}
```

Note: Accumulator is not affected.

DECX—Decrement X Vector Pointer

The DECX instruction decrements the X vector pointer by one element, according to the *increment* and the *wrap*.

Syntax:

EXEC DECX

15 11	10 0
10000	101 0010 1101

Operation:

```
{
  X.ADDR = &X[−1]
}
```

Note: Accumulator is not affected.

DECY—Decrement Y Vector Pointer

The DECY instruction decrements the Y vector pointer by one element, according to the *increment* and the *wrap*.

Syntax:

EXEC DECY

15 11	10 0
10000	101 0010 1111

Operation:

```
DECY
{
  Y.ADDR = &Y[-1];
}
```

Note: Accumlator is not affected.

DECZ—Decrement Z Vector Pointer

The DECZ instruction decrements the Z vector by one element, according to the *increment* and the *wrap*.

Syntax:

EXEC DECZ

15 11	10 0
10000	101 0011 0001

Operation:

```
{
  Z.ADDR = &Z[-1];
}
```

Note: Accumulator is not affected.

3.4.5.7 Flow Control Instructions

NOPR—No Operation

The NOPR command passes control to the next command in the command list. No operation is performed.

Syntax:

NOPR

15 11	10 0
11010	00000000

Note: Accumulator is not affected.

HALT—Terminate Command-List Execution

The HALT command terminates execution of the command list. No further commands are executed. This event is made visible to the CPU core, as specified in Section 3.6.

Syntax:

HALT

15 11	10 0
11001	00000000000

Note: Accumulator is not affected.

DJNZ—Decrement and Jump If Not Zero

The DJNZ command is used to implement loops and branches in the command list. The value of the REPEAT.COUNT field is decremented by 1 and compared to 0. If it is not equal to 0, then execution of the command list continues with the command located in the RAM address specified by the REPEAT.TARGET field. When the REPEAT.COUNT field is equal to 0, then execution continues with the next command in the command list.

The DSPM has only one REPEAT register. To nest loops, user must save the contents of the REPEAT register before starting an inner loop, and restore it at the end of the inner loop.

Syntax:

EXEC DJNZ

15 11	10 0
10000	101 0110 1100

Note: Accumulator is not affected.

DBPT—Debug Breakpoint

The DBPT instruction is used for implementing software debug breakpoint in the DSPM command-list. Whenever there is an attempt to execute a DBPT instruction, the NMISTAT.UND bit is set to “1”.

Syntax:

EXEC DBPT

15 11	10 0
10000	111 1111 1110

Note: Accumulator is not affected.

3.4.5.8 Internal Memory Move Instructions

VRMOV—Vector Real Move

The VRMOV instruction copies the real X vector to the real Z vector.

Syntax:

EXEC VRMOV

15 11	10 0
10000	101 0010 1011

Operation:

```
{
  real X, Z;
  for (n = 0; n < LENG; n++)
  {
    Z[n] = X[n];
  }
}
```

VARMOV—Vector Aligned Real Move

The VARMOV instruction copies the aligned real X vector to the aligned real Z vector.

Syntax:

EXEC VARMOV

15 11	10 0
10000	100 0011 1000

Operation:

```
{

  aligned_real X, Z;
  for (n = 0; n < LENG; n++)
  {
    Z[n].low = X[n].low;
    Z[n].high = X[n].high;
  }
}
```

VRGATH—Vector Real Gather

The VRGATH instruction gathers non-contiguous elements of the X real vector, as specified by the Y integer vector, and places them in contiguous locations in the Z real vector.

Syntax:

EXEC VRGATH

15 11	10 0
10000	100 0011 1010

Operation:

```
{
  real X, Z;
  integer X.ADDR, Y;
  for (n = 0; n < LENG; n++)
  {
    Z[n] = mem[(X.ADDR+Y[n]) & 0xFFFF];
  }
}
```

VRSCAT—Vector Real Scatter

The VRSCAT instruction scatters contiguous elements of the X real vector, and places them in non-contiguous locations in the Z real vector, as specified by the Y integer vector.

Syntax:

EXEC VRSCAT

15 11	10 0
10000	100 0100 0000

Operation:

```
{
  real X, Z;
  integer Z.ADDR, Y;
  for (n=0; n < LENG; n++)
  {
    mem[Z.ADDR+Y[n]) & 0xFFFF] = X[n];
  }
}
```

3.4.5.9 External Memory Move Instructions

VXLOAD—Vector External Load

The VXLOAD instruction loads a vector from external memory into the Z vector. The external memory address is specified in the EABR and X registers.

Syntax:

EXEC VXLOAD

15 11	10 0
10000	100 0100 1111

Operation:

```
VXLOAD
{
  real X, Z;
  ext_address EABR;
  for (n=0; n<LENG; n++)
  {
    Z[n] = ext_mem[EABR + (ext_address)
2*&X[n]]
  }
}
```

VXSTORE—Vector External Store

The VXSTORE instruction stores the Z vector into an external memory vector. The external memory address is specified in the EABR and X registers.

Syntax:

EXEC VXSTORE

15 11	10 0
10000	100 0101 0101

Operation:

```
{
  real X, Z;
  ext_address EABR;
  for (n=0; n < LENG; n++)
  {
    ext_mem[EABR + (ext_address) 2*&Z[n]] =
X[n];
  }
}
```

VXGATH—Vector External Gather

The VXGATH instruction gathers non-contiguous elements of the external memory vector, as specified by the Y integer vector, and places them in contiguous locations in the Z real vector. The external memory address is specified in the EABR and X registers.

Syntax:

EXEC VXGATH

15 11	10 0
10000	100 0100 0110

Operation:

```
{
  real X, Z;
  ext_address EABR;
  integer Y, X.ADDR;
  for (n=0; n < LENG; n++)
  {
    Z[n]=ext_mem
[EABR+(ext_address)2*((X.ADDR+(integer)Y[n])
& 0xFFFF)];
  }
}
```

3.4.5.10 Arithmetic/Logical Instructions

VROP—Vector Real Op

The VROP instruction performs one of 7 operations between corresponding elements of the X and Y real vectors, and writes the result in the corresponding place in the Z output vector. The operation to be performed is specified in PARAM.OP field.

Syntax:

EXEC VROP

15 11	10 0
10000	101 0110 1000

Operation:

```
{
  real X,Y,Z;
  for (n=0; n < LENG; n++)
  {
    Z[n] = (real) (X[n] <op> Y[n]);
  }
}
```

The allowed values in PARAM.OP are:

<op>	Operation	
011010	ADD	Z = X + Y
100111	SUB	Z = X − Y
001000	BIC	Z = X & $\overline{Y}$
100000	AND	Z = X & Y
111000	OR	Z = X | Y
011000	XOR	Z = X ⊕ Y
001100	INV	Z = $\overline{Y}$

VAROP—Vector Aligned Real Op

The VAROP instruction performs one of 7 operations between corresponding elements of the X and Y aligned vectors, and writes the result in the coresponding place in the Z output vector. The operation to be performed is specified in PARAM.OP field.

Syntax:

EXEC VAROP

15 11	10 0
10000	100 0001 1010

Operation:

```
{
  aligned_real X,Y,Z;
  for (n=0; n < LENG; n++)
  {
  Z[n].low = (real) (X[n].low <op>
Y[n].low);
  Z[n].high = (real) (X[n].high <op>
Y[n].high);
  }
}
```

Note: The allowed values in PARAM.OP are the same as those in VROP.

3.4.5.11 Multiply-and-Accumulate Instructions

VRMAC—Vector Real Multiply and Accumulate

The VRMAC instruction performs a convolution sum of the X and Y real vectors. The previous value of the accumulator is used and the result stored in `Z[0]`.

Syntax:

EXEC VRMAC

15 11	10 0
10000	100 0000 0111

Operation:

```
{
  real X,Y,Z;
  real_acc A;
  for (n=0; n < LENG; n++)
  {
    A = A + X[n] * Y[n];
  }
  Z[0] = (real) A;
}
```

Note: When PARAM.CLR is set to "1", A is cleared to "0" prior to the first addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

VARMAC—Vector Aligned Real Multiply and Accumulate

The VARMAC instruction performs a convolution sum of the X and Y real vectors. The previous value of the accumulator is used and the result is stored in Z[0].

Syntax:

EXEC VARMAC

15 11	10 0
10000	100 0000 0000

Operation:

```
{
  aligned_real X,Y;
  real Z;
  real_acc A;
  for (n=0; n < LENG; n++)
  {
   A = A + X[n].low * Y[n].low +
       X[n].high * Y[n].high ;
  }
  Z[0] = (real) A;
}
```

Note: When PARAM.CLR is set to "1", A is cleared to "0" prior to the first addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

VCMAC—Vector Complex Multiply and Accumulate

The VCMAC instruction performs a convolution sum of the X and Y complex vectors. The previous value of the accumulator is used, and the result is stored in Z[0].

Syntax:

EXEC VCMAC

15 11	10 0
10000	100 0111 0101

Operation:

```
{
  complex X,Y,Z;
  complex_acc A;
  for (n=0; n < LENG; n++)
  {
    A = A + X[n] * Y[n];
  }
  Z[0] = (complex) A;
}
```

Note: When PARAM.COJ is set to "1", X[n] is multiplexed by the conjugate of Y[n]. When PARAM.CLR is set to "1", A is cleared to "0" prior to the first addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

VRLATP—Vector Real Lattice Propagate

The VRLATP instruction is used for implementing lattice and inverse lattice filter operations. This instruction is used to update the propagating values of vector Z.

Syntax:

EXEC VRLATP

15 11	10 0
10000	100 0010 1100

Operation:

```
{
  real X,Y,Z;
  real_acc A;
  A = (real_acc) Z[0];
  for (n=1; n < LENG; n++)
  {
    A = A + X[n - 1] * Y[n - 1];
    Z[n] = (real) A;
    A = (real_acc) Z[n];
  }
}
```

Note: When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign. The LENG parameter for this operation must be greater than 1.

VCLATP—Vector Complex Lattice Propagate

The VCLATP instruction is used for implementing lattice and inverse lattice filter operations. This instruction is used to update the propagating values of vector Z.

Syntax:

EXEC VCLATP

15 11	10 0
10000	100 1110 1000

Operation:

```
{
  complex X,Y,Z;
  complex_acc A;
  A = (complex_acc) Z[0];
  for (n=1; n < LENG; n++)
  {
    A = A + X[n-1] * Y[n-1];
    Z[n] = (complex) A;
  }
}
```

Note: When PARAM.COJ is set to "1", X[n] is multiplied by the conjugate of Y[n]. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign. The LENG parameter for this operation must be greater than 1.

3.4.5.12 Multiply-and-Add Instructions

VAIMAD—Vector Aligned Integer Multiply and Add

The VAIMAD instruction multiplies corresponding elements of the X and Y integer vectors, and adds or subtracts the result, as an integer value, to the integer vector Z. This result is placed in the Z output vector.

Syntax:

EXEC VAIMAD

15 11	10 0
10000	100 0001 0100

Operation:

```
{
  aligned_integer X,Y;
  integer Z;
  for (n=0; n < LENG; n++)
  {
    Z[2n] = (integer) (Z[2n] + X[n].low *
Y[n].low);
    Z[2n+1] = (integer) (Z[2n+1] + X[n].high
* Y[n].high);
  }
}
```

Note: When PARAM.CLR is set to "1", only multiplication is done without addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

VAIMADS—Vector Aligned Integer Multiply and Add Saturated

The VAIMADS instruction multiplies corresponding elements of the X and Y integer vectors, and adds or subtracts the result, as an integer value, to the integer vector Z. This result is placed in the Z output vector. The saturation logic provides clamping of the accumulator results before writing the result back to the Z vector whenever the result cannot be represented correctly within the limits of the integer data type.

Syntax:

EXEC VAIMADS

15 11	10 0
10000	101 0101 1100

Operation:

```
{
  aligned_integer X,Y;
  integer Z;
  for (n=0; n < LENG; n++)
  {
  Z[2n] = (integer) (Z[2n] + X[n].low *
Y[n].low);
  Z[2n+1] = (integer) (Z[2n+1] + X[n].high *
Y[n].high);
  }
}
```

VRMAD—Vector Real Multiply and Add

The VRMAD instruction multiplies corresponding elements of the X and Y real vectors and adds or subtracts the result to the real vector Z. This result is placed in the Z output vector.

Syntax:

EXEC VRMAD

15 11	10 0
10000	100 0011 0011

Operation:

```
{
  real X,Y,Z;
  for (n=0; n < LENG; n++)
  {
    Z[n] = (real) (Z[n] + X[n] * Y[n]);
  }
}
```

Note: When PARAM.CLR is set to "1", only multiplication is performed, without addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

VARMAD—Vector Aligned Real Multiply and Add

The VARMAD instruction multiplies corresponding elements of the X and Y real vectors and adds or subtracts the result to the real vector Z. This result is placed in the Z output vector.

Syntax:

EXEC VARMAD

15 11	10 0
10000	100 0000 1110

Operation:

```
{
  aligned_real X,Y,Z;
  for (n=0; n < LENG; n++)
  {
  Z[n].low = (real) (Z[n].low + X[n].low *
Y[n].low);
  Z[n].high = (real) (Z[n].high + X[n].high
* Y[n].high);
  }
}
```

Note: When PARAM.CLR is set to "1", only multiplication is performed, without addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

VEMAD—Vector Extended Multiply and Add

The VEMAD instruction multiplies corresponding elements of the X and Y real vectors and adds or subtracts the result, as an extended-precision value, to the extended-precision vector Z. This result is placed in the Z output vector.

Syntax:

EXEC VEMAD

15 11	10 0
10000	101 0001 0010

Operation:

```
{
  aligned_real X,Y;
  extended Z;
  for (n=0; n < LENG; n++)
  {
  Z[2n] = (extended) (Z[2n] + X[n].low *
Y[n].low) ;
  Z[2n+1] = (extended) (Z[2n+1] + X[n].high
* Y[n].high) ;
  }
}
```

Note: When PARAM.CLR is set to "1", only multiplication is performed, without addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

VCMAD—Vector Complex Multiply and Add

The VCMAD instruction multiplies the corresponding elements of the X and Y complex vectors and adds or subtracts the result to the complex vector Z. This result is placed in the Z output vector.

Syntax:

EXEC VCMAD

15 11	10 0
10000	100 1110 0000

Operation:

```
{
  complex X,Y,Z;
  for (n=0; n < LENG; n++)
  {
    Z[n] = (complex) (Z[n] + X[n] * Y[n]);
  }
}
```

Note: When PARAM.COJ is set to "1", X[n] is multiplied by the conjugate of Y[n]. When PARAM.CLR is set to "1", only multiplication is performed, without addition. When PARAM.SUB is set to "1", the "+" sign is replaced by a "−" sign.

3.4.5.13 Clipping and Min/Max Instructions

VARABS—Vector Aligned Real Absolute Value

The VARABS instruction computes the absolute value of each element in the real vector X and places the result in the corresponding place in the Y output vector.

Syntax:

EXEC VARABS

15 11	10 0
10000	100 0001 1111

Operation:

```
{
  aligned_real X,Z;
  for (n=0; n < LENG; n++)
  {
    Z[n].low = abs (X[n].low);
    Z[n].high = abs (X[n].high);
  }
}
```

Note: There is no representation for the absolute value of 0x8000. Whenever an absolute value of 0x8000 is needed, OVF.SAT is set to "1", and the maximum positive number 0x7FFF is returned.

VARMIN—Vector Aligned Real Minimum

The VARMIN instruction compares corresponding elements of the X and Y real vectors, and writes the smaller of the two in the corresponding place in the Z integer vector.

Syntax:

EXEC VARMIN

15 11	10 0
10000	100 0101 1111

Operation:

```
{
  aligned_real X,Y,Z;
  for (n=0; n < LENG; n++)
  {
    Z[n].low = min (X[n].low ,Y[n].low);
    Z[n].high = min (X[n].high ,Y[n].high);
  }
}
```

VARMAX—Vector Aligned Real Maximum

The VARMAX instruction compares corresponding elements of the X and Y real vectors, and writes the larger of the two in the corresponding place in the Z integer vector.

Syntax:

EXEC VARMAX

15 11	10 0
10000	100 0110 0110

Operation:

```
{
  aligned_real X,Y,Z;
  for (n=0; n < LENG; n++)
  {
    Z[n].low = max (X[n].low , Y[n].low);
    Z[n].high = max (X[n].high , Y[n].high);
  }
}
```

VRFMIN—Vector Real Find Minimum

The VRFMIN instruction scans the X real vector and returns the address of the element with the smallest value. The resulting address is placed in Z[0].

Syntax:

EXEC VRFMIN

15 11	10 0
10000	100 0110 1101

Operation:

```
{
  real X ;
  integer Z ;
  internal_register real    tempX;
  internal_register integer tempA;
  tempX = X[0];
  tempA = &X[0];
  for (n=1; n < LENG; n++)
  {
    if (X[n] < tempX)
    {
      tempX = X[n];
      tempA = &X[n];
    }
  }
  Z{0} = tempA;
}
```

Note: The LENG parameter for this operation must be greater than 1.

VRFMAX—Vector Real Find Maximum

The VRFMAX instruction scans the X real vector and returns the address of the element with maximum value. The resulting address is placed in Z[0].

Syntax:

EXEC VRFMAX

15 11	10 0
10000	100 0010 0100

Operation:

```
{
  real X;
  integer Z;
  internal_register real tempX;
  internal_register integer tempA;
  tempX = X[0];
  tempA = &X[0];
  for (n=1; n < LENG; n++)
  {
    if (X[n] > tempX)
    {
      tempX = X[n];
      tempA = &X[n];
    }
  }
  Z[0] = tempA;
}
```

Note: The LENG parameter for this operation must be greater than 1.

EFMAX—Extended Find Maximum

This instruction is not supported by the NS32FX161.

The EFMAX instruction implements a single iteration of maximum search loop. The extended value in the accumulator is compared with the first element of the extended Z vector. The large value is stored back into the Z vector. In case the larger value was the accumulator, then ss is stored in the second location of the Z-vector (as an integer).

Syntax:

EXEC EFMAX

15 11	10 0
10000	101 0100 1011

Operation:

```
{
  integer Y, Z[1];
  extended temp, Z[0];
  real X;
  real_acc A;
  A = (real_acc) ((extended)A);
  temp = Z[0];
  if (A > temp)
  {
    temp = (extended) A;
    Z[1] = &X[0];
  }
  Z[0] = temp;
}
```

Note: The Y vector must hold the following values: Y[0] must be 0x7fff, Y[1] must be 0x0001, and Y[2] must be 0x4000.

3.4.5.14 Special Instructions

ESHL—Extended Shift Left

This instruction is not supported by the NS32FX161.

The ESHL instruction performs a shift-left operation on extended-precision data in the accumulator, and stores the more significant half of the result as a real value into the first element of the real Z vector.

Syntax:

EXEC ESHL

15 11	10 0
10000	101 0110 0100

Operation:

```
{
  real_acc A;
  A = (real_acc) ((extended)A);
  if (LENG > 1) for (n=1; n<LENG; n++)
  {
    A = A + A;
  }
  Z[0] = (real) A;
}
```

Note: The LENG parameter for this operation must be greater than 0. When LENG equals 1, only the real part of the accumulator is updated. When LENG is greater than 1, both the real and the imaginary parts of the accumulator are updated to the same value.

VCPOLY—Vector Complex Polynomial

The VCPOLY instruction performs one iteration of evaluating a polynomial with real coefficients, for a vector of complex-valued arguments, including down-scaling of the coefficients to avoid overflow. In addition, the instruction accumulates the scaled-down energy, with a decay factor, of the polynomial's real coefficients.

Syntax:

EXEC VCPOLY

15 11	10 0
10000	101 0001 1000

Operation:

```
{
  complex X,Z;
  real Y;
  complex temp;
  temp.re = (real) Y[0] * X[0].re;
  temp.im = 0;
  for (n=0; n < LENG; n++)
  {
    Z[n] = (complex) Z[n] * X[n+1] + temp;
  }
  Z[LENG].re = (real) (Z[LENG].re *
X[LENG+1].re + Y[0] * temp.re);
  Y.ADDR = &Y[1];
}
```

Note: The LENG parameter for this operation must be greater than 1.

VDECIDE—Vector Nearest Neighbor Decision Logic

The VDECIDE instruction is used to implement nearest neighbor decision in Quadrature Amplitude Modulation (QAM) modem applications. The input is the X complex vector. The output is placed in the Z integer vector, which can be used as an index vector to extract information from look-up tables. The indicated constant values are taken from the Y vector.

Syntax:

EXEC VDECIDE

15 11	10 0
10000	100 1111 0000

Operation:

```
{
  complex X;
  aligned_real Y;
  real Z;
  internal_register complex temp;
  for (n=0; n < LENG; n++)
  {
  temp.re = min (X[n].re, Y[0].low);
  temp.im = min (X[n].im, Y[0].high);
  temp.re = max (temp.re, Y[1].low);
  temp.im = max (temp.im, Y[1].high);
  X[n] = temp;
  Z[n] = (real) ((temp.re * Y[2].low) &
(extended) Y[3].low) |
  ((temp.im * Y[2].high) & (extended)
Y[3].high) ;
  }
}
```

Note: Y.INCR must be specified as 1, and Y.WRAP must be specified as 3.

VDIST—Vector Euclidean Distance

The VDIST instruction calculates the square of the Euclidean distance between corresponding elements of the X and Y complex vectors, and places the result in the Z real vector.

Syntax:

EXEC VDIST

15 11	10 0
10000	100 1111 1110

Operation:

```
{
  complex X,Y;
  real Z;
  for (n=0; n < LENG; n++)
  {
    Z[n] = (real) (X[n].re - Y[n].re)** 2+
                  (X[n].im - Y[n].im) **2   ;
  }
}
```

VFFT—Vector Fast Fourier Transform

The VFFT instruction implements one pass of in-place FFT vector update, according to the radix-2 FFT method.

Syntax:

EXEC VFFT

15 11	10 0
10000	101 0000 0110

Operation:

```
{
  complex X,Y,Z;
  complex temp;
  for (n=0; n < LENG; n++)
  {
    temp = (complex) (Z[n] + X[n] * Y[n]);
    Y[n] = (complex) (Z[n] - X[n] * Y[n]);
    Z[n] = temp
  }
}
```

VESIIR—Vector Extended Single-Pole IIR

This instruction is not supported by the NS32FX161.

The VESIIR instruction performs a special form of an Infinite-Impulse Response (IIR) filter. The samples and coefficient are given as real values, as well as the output result. However, the accumulation is performed using extended-precision arithmetic.

Syntax:

EXEC VESIIR

15 11	10 0
10000	101 0011 0111

Operation:

```
{
  real X,Y,Z;
  real_acc A;
  for (n=0; n < LENG; n++)
  {
    A = (real_acc) ((extended)A);
    A = (real_acc) (A * X[n])) + Y[n+2];
  Z[n] = (real) A;
  }
}
```

Note: The term (A * X [n]) is a 32-bit by 16-bit multiplication. During the conversion of this product to a real__accumulator data type, rounding is done if PARAM.RND is "1". During the conversion of A to a real data type, the result is rounded if Y[0] = 0x0080, or truncated if Y[0] = 0x0. The result with other values of Y[0] are unpredictable. Y[1] must be specified as 0x7fff.

3.5 SYSTEM INTERFACE

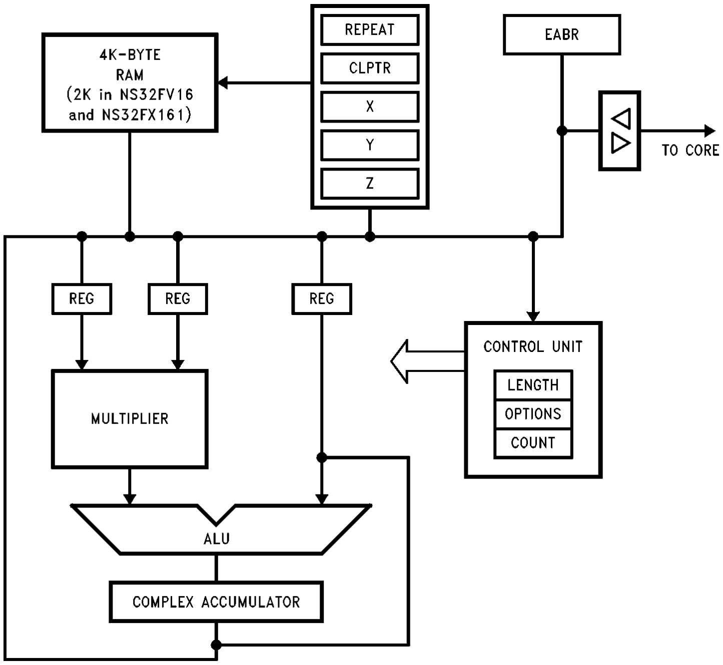
This section provides general information on the NS32FX164 interface to the external world. Descriptions of the CPU requirements as well as the various bus characteristics are provided here. Details on other device characteristics including timing are given in Sections 4.2–4.4.2.

3.5.1 Power and Grounding

The NS32FX164 requires a single 5V power supply, applied on the V_{CC} pins. These pins should be connected together by a power (V_{CC}) plane on the printed circuit board.

The grounding connections are made on the GND pins. These pins should be connected together by a ground (GND) plane on the printed circuit board.

Both power and ground connections are shown in *Figure 3-14.*



TL/EE/11267–25

FIGURE 3-13. DSP Module Block Diagram

For optimal noise immunity, the power and ground pins should be connected to V_{CC} and ground planes respectively. If V_{CC} and ground planes are not used, single conductors should be run directly from each V_{CC} pin to a power point, and from each GND pin to a ground point. Daisy-chained connections should be avoided.

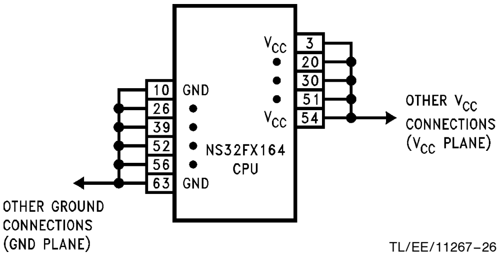
Decoupling capacitors should also be used to keep the noise level to a minimum. Standard 0.1 μF ceramic capacitors can be used for this purpose. They should attach to V_{CC}, GND pins as close as possible to the NS32FX164.

During prototype using wire-wrap or similar methods, the capacitors should be soldered directly to the power pins of the NS32FX164 socket, or as close as possible, with very short leads.

Design Notes

When constructing a board using high frequency clocks with multiple lines switching, special care should be taken to avoid resonances on signal lines. A separate power and ground layer is recommended. This is true when designing boards for the NS32FX164. Switching times of under 5 ns on some lines are possible. Resonant frequencies should be maintained well above the 200 MHz frequency range on signal paths by keeping traces short and inductance low. Loading capacitance at the end of a transmission line contributes to the resonant frequency and should be minimized if possible. Capacitors should be located as close as possible across each power and ground pair near the NS32FX164.

Power and ground connections are shown in *Figure 3-14*.



VCC
3
20
30
51
54
VCC
10
26
39
52
56
63
GND
GND
NS32FX164
CPU
OTHER VCC CONNECTIONS (VCC PLANE)
OTHER GROUND CONNECTIONS (GND PLANE)

TL/EE/11267–26

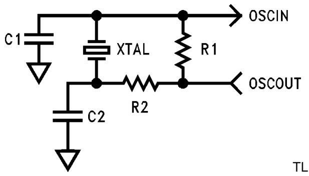
FIGURE 3-14. Power and Ground Connections

3.5.2 Clocking

The NS32FX164 provides an internal oscillator that interacts with an external clock source through two signals; OSCIN and OSCOUT.

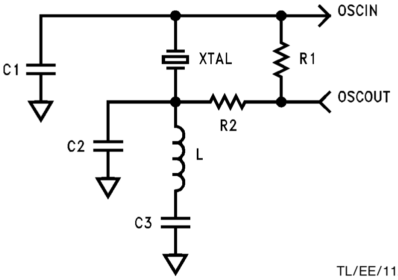
Either an external single-phase clock signal or a crystal can be used as the clock source. If a single-phase clock source is used, only the connection on OSCIN is required; OSCOUT should be left unconnected or loaded with no more than 5 pF of stray capacitance. The voltage level requirements specified in Section 4.3 must also be met for proper operation.

When operation with a crystal is desired, special care should be taken to minimize stray capacitances and inductances. The crystal, as well as the external components, should be placed in close proximity to the OSCIN and OSCOUT pins to keep the printed circuit trace lengths to an absolute minimum. *Figures 3-15* and *3-16* show the external crystal interconnections. Table 3-3 provides the crystal characteristics and the values of the R, C, and L components, including stray capacitance, required for various frequencies.



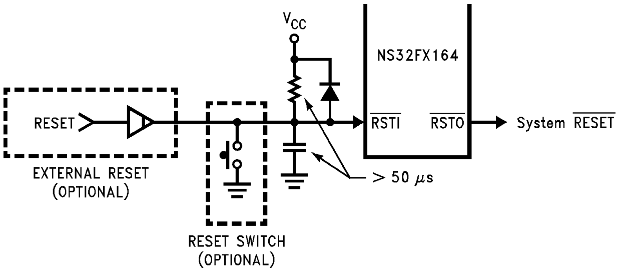
TL/EE/11267–27

FIGURE 3-15. Crystal Interconnections—30 MHz



TL/EE/11267–28

FIGURE 3-16. Crystal Interconnections— 40 MHz, 50 MHz



TL/EE/11267–29

FIGURE 3-17. Recommended Reset Connections

TABLE 3-3. External Oscillator Specifications Crystal Characteristics

Type	AT-Cut
Tolerance	0.005% at +25°C
Stability	0.01% from 0°C to +70°C
Resonance	
30 MHz:	Fundamental (Parallel)
40 MHz or 50 MHz:	Third Overtone (Parallel)
Maximum Series Resistance	50Ω
Maximum Shunt Capacitance	7 pF

R, C and L Values

Frequency (MHz)	R1 (kΩ)	R2 (Ω)	C1 (pF)	C2 (pF)	C3 (pF)	L (μH)
30	180	51	20	20		
30	180	51	20	20	800–1300	3.3
40	150	51	20	20	800–1300	1.8
50	150	51	20	20	800–1300	1.1

3.5.3 Power Save Mode

The NS32FX164 provides a power save feature that can be used to significantly reduce the power consumption at times when the computational demand decreases. The device uses the clock signal at the OSCIN pin to derive the internal clock as well as the external signals CTTL and FCLK. The frequency of these clock signals is affected by the clock scaling factor. Scaling factors of 1, 2, 4, or 8 can be selected by properly setting the C- and M-bits in the CFG register. The power save mode should not be used to reduce the clock frequency below the minimum frequency required by the CPU.

Upon reset, both C and M are set to zero, thus maximum clock rate is selected.

Due to the fact that the C- and M-bits are programmed by the SETCFG instruction, the power save feature can only be controlled by programs running in supervisor mode.

The following table shows the C- and M-bit settings for the various scaling factors, and the resulting supply current for a crystal frequency of 50 MHz.

Clock Scaling Factor vs Supply Current

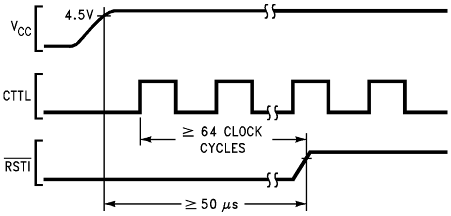
C	M	Scaling Factor	CPU Clock Frequency	Typical I_{CC} at +5V
0	0	1	25 MHz	200 mA
0	1	2	12.5 MHz	120 mA
1	0	4	6.25 MHz	80 mA
1	1	8	3.13 MHz	55 mA

3.5.4 Resetting

The $\overline{\text{RSTI}}$ input pin is used to reset the NS32FX164. The CPU samples $\overline{\text{RSTI}}$ on the falling edge of CTTL.

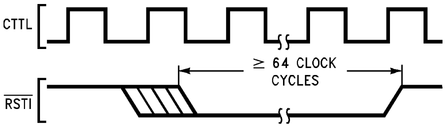
Whenever a low level is detected, the CPU responds immediately. Any instruction being executed is terminated; any results that have not yet been written to memory are discarded; and any pending interrupts and traps are eliminated. The internal latch for the edge-sensitive $\overline{\text{NMI}}$ signal is cleared. The DSP module ST register is set to 0.

On application of power, $\overline{\text{RSTI}}$ must be held low for at least 50 μs after V_{CC} is stable. This is to ensure that all on-chip voltages are completely stable before operation. Whenever a Reset is applied, it must also remain active for not less than 64 CTTL cycles. See *Figures 3-18* and *3-19*.



TL/EE/11267–30

FIGURE 3-18. Power-On Reset Requirements



TL/EE/11267–31

FIGURE 3-19. General Reset Timing

While in the Reset state, the CPU drives the signals $\overline{\text{ADS}}$, $\overline{\text{IAS}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{DBE}}$, $\overline{\text{TSO}}$, $\overline{\text{BPU}}$, $\overline{\text{IOUT}}$ and $\overline{\text{DDIN}}$ inactive. AD0–AD15, A16–A23 and $\overline{\text{SPC}}$ are floated, ALE is HIGH and the state of all other output signals is undefined.

The internal CPU clock and CTTL run at half the frequency of the signal on the OSCIN pin.

The $\overline{\text{HOLD}}$ signal must be kept inactive. After the $\overline{\text{RSTI}}$ signal is driven high, the CPU will stay in the reset condition for approximately 8 clock cycles and then it will begin execution at address 0.

The PSR is reset to 0. The CFG C- and M-bits are reset to 0. FCLK runs at the same frequency as OSCIN. $\overline{\text{NMI}}$ is enabled to allow Non-Maskable Interrupts. The following conditions are present after reset due to the PSR being reset to 0:

Tracing is disabled.

Supervisor mode is enabled.

Supervisor stack space is used when the TOS addressing mode is indicated.

No trace traps are pending.

Only $\overline{\text{NMI}}$ is enabled. Maskable interrupts are disabled.

$\overline{\text{BPU}}$ is inactive high.

The Clock Scaling Factor is set to 1, refer to Section 3.5.3.

Note that vector/non-vectored interrupts have not been selected. While interrupts are disabled, a SETCFG [I] instruction must be executed to enable vectored interrupts. If non-vectored interrupts are required, a SETCFG without the [I] must be executed.

The presence/absence of the NS32081, NS32181, or NS32381 has also not been declared. If there is a Floating-Point Unit, a SETCFG [F] instruction must be executed. If there is no floating-point unit, a SETCFG without the [F] must be executed.

In general, a SETCFG instruction must be executed in the reset routine, in order to properly configure the CPU. The options should be combined, and executed in a single instruction. For example, to declare vectored interrupts, a Floating-Point unit installed, and full CPU clock rate, execute a SETCFG [F, I] instruction. To declare non-vectored interrupts, no FPU, and full CPU clock rate, execute a SETCFG [ ] instruction.

3.5.5 Bus Cycles

The NS32FX164 will perform bus cycles for one of the following reasons:

1. To fetch instructions from memory.
2. To write or read data to or from memory or external peripheral devices.
3. To acknowledge an interrupt, or to acknowledge completion of an interrupt service routine.
4. To notify external logic of any accesses to the on-chip peripheral device registers or internal RAM.
5. To transfer information to or from a Slave Processor.

3.5.5.1 Bus Status

The NS32FX164 CPU presents four bits of Bus Status information on pins ST0–ST3. The various combinations on these pins indicate why the CPU is performing a bus cycle, or, if it is idle on the bus, they why it is idle.

The Bus Status pins are interpreted as a 4-bit value, with ST0 the least significant bit. Their values decode as follows:

0000 — The bus is idle because the CPU does not need to perform a bus access.

0001 — The bus is idle because the CPU is executing the WAIT instruction.

0010 — DSP Module Data Transfer.

0011 — The bus is idle because the CPU is waiting for a Slave Processor to complete an instruction.

0100 — Interrupt Acknowledge, Master

The CPU is performing a Read cycle to acknowledge an interrupt request. See Section 3.2.3.

0101 — Interrupt Acknowledge, Cascaded.

The CPU is reading an interrupt vector to acknowledge a maskable interrupt request from a Cascaded Interrupt Control Unit.

0110 — End of Interrupt, Master.

The CPU is performing a Read cycle to indicate that it is executing a Return from Interrupt (RETI) instruction at the completion of an interrupt's service procedure.

0111 — End of Interrupt, Cascaded.

The CPU is performing a read cycle from a Cascaded Interrupt Control Unit to indicate that it is executing a Return from Interrupt (RETI) instruction at the completion of an interrupt's service procedure.

1000 — Sequential Instruction Fetch.

The CPU is reading the next sequential word from the instruction stream into the Instruction Queue. It will do so whenever the bus would otherwise be idle and the queue is not already full.

1001 — Non-Sequential Instruction Fetch

The CPU is performing the first fetch of instruction code after the Instruction Queue is purged. This will occur as a result of any jump or branch, any interrupt or trap, or execution of certain instructions.

1010 — Data Transfer.

The CPU is reading or writing an operand of an instruction.

1011 — Read RMW Operand.

The CPU is reading an operand which will subsequently be modified and rewritten. The write cycle of RMW will have a "write" status.

1100 — Read for Effective Address Calculation.

The CPU is reading information from memory in order to determine the Effective Address of an operand. This will occur whenever an instruction uses the Memory Relative or External addressing mode.

1101 — Transfer Slave Processor Operand.

The CPU is either transferring an instruction operand to or from a Slave Processor, or it is issuing the Operation Word of a Slave Processor instruction.

1110 — Read Slave Processor Status.

The CPU is reading a Status Word from a Slave Processor after the Slave Processor has signalled completion of an instruction.

1111 — Broadcast Slave ID.

The CPU is initiating the execution of a Slave Processor instruction by transferring the first byte of the instruction, which represents the slave processor indentification.

3.5.5.2 Basic Read and Write Cycles

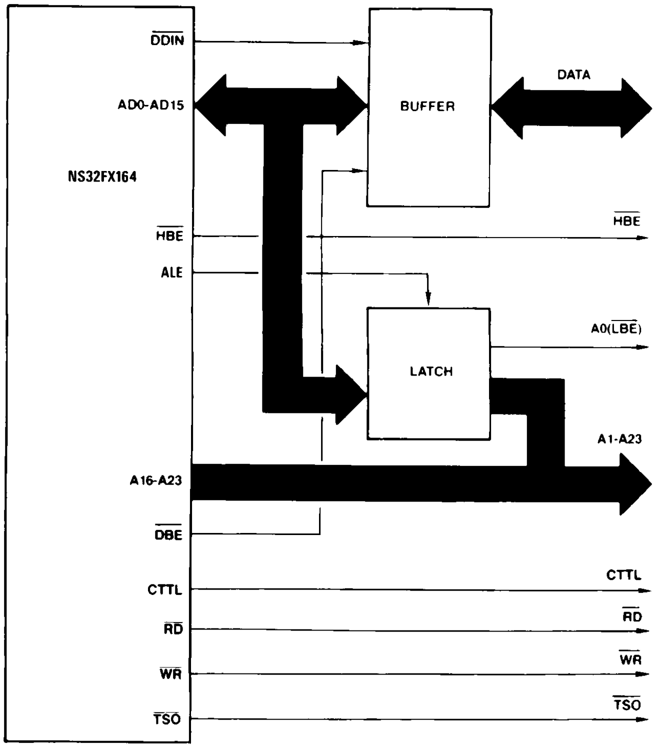
The sequence of events occurring during a CPU access to either memory or peripheral device is shown in *Figure 3-21* for a read cycle, and *Figure 3-22* for a write cycle.

The cases shown assume that the selected memory or peripheral device is capable of communicating with the CPU at full speed. If not, then cycle extension may be requested through $\overline{\text{CWAIT}}$.

A full-speed bus cycle is performed in four cycles of the CTTL clock signal, labeled T1 through T4. Clock cycles not associated with a bus cycle are designated Ti (for "idle").

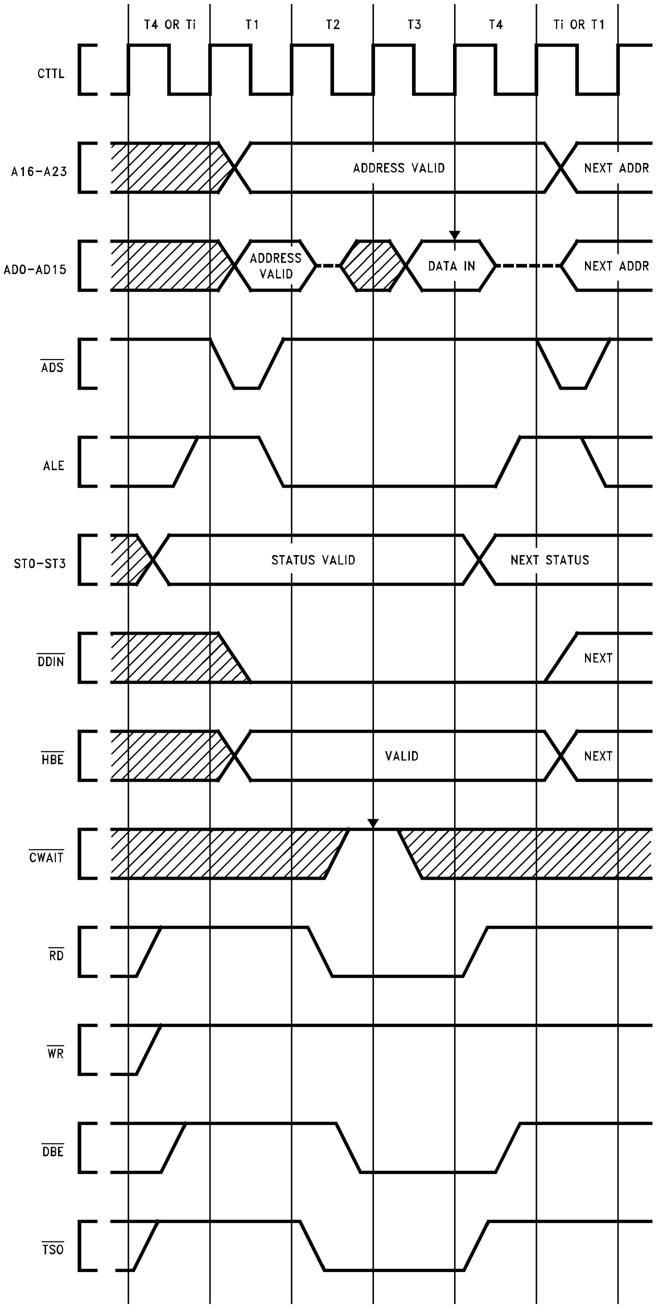
During T1, the CPU applies an address on pins AD0–AD15 and A16–A23 and provides a low-going pulse on the $\overline{\text{ADS}}$ pin, which serves the dual purpose of informing external circuitry that a bus cycle is starting and of providing control to an external latch for demultiplexing Address bits 0–15 from the AD0–AD15 pins. It also deasserts the ALE signal, which eliminates the need to invert $\overline{\text{ADS}}$ to generate the strobe for the address latches. See *Figure 3-20*. During this time also the status signals $\overline{\text{DDIN}}$, indicating the direction of the transfer, and $\overline{\text{HBE}}$, indicating whether the high byte (AD8–AD15) is to be referenced, become valid.

During T2 the CPU switches the Data Bus, AD0–AD15, to either accept or present data. Note that the signals A16–A23 remain valid, and need not be latched.



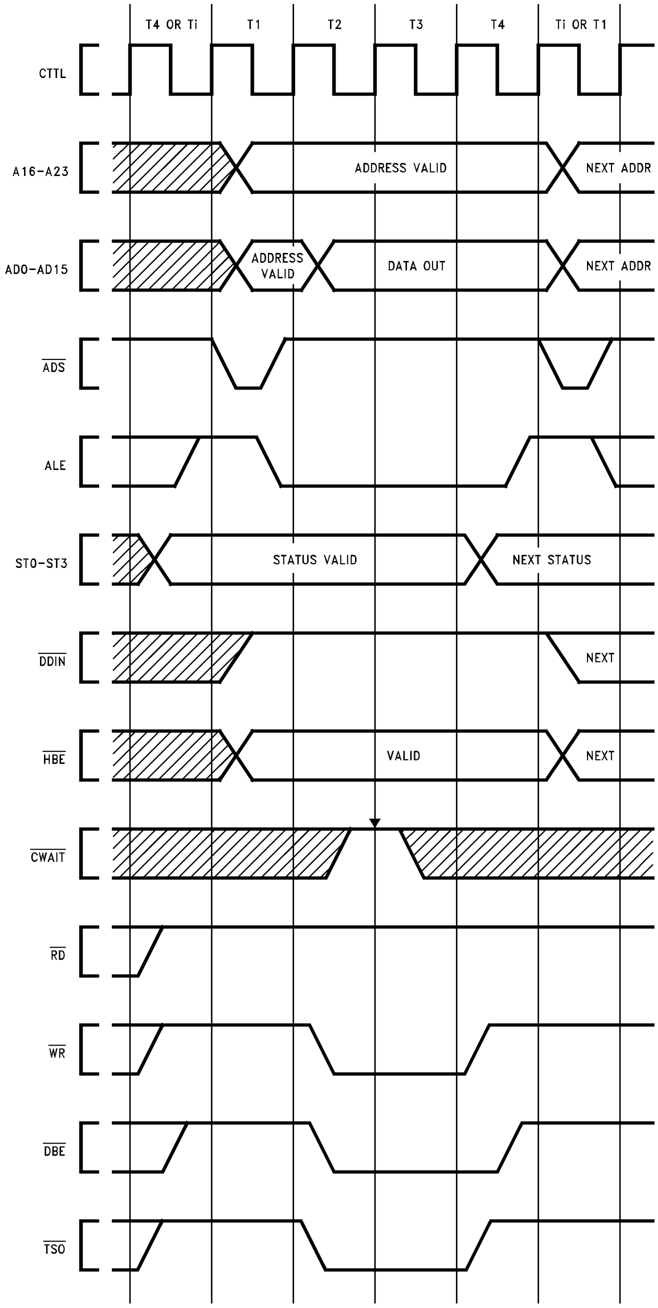
TL/EE/11267-32

FIGURE 3-20. Bus Connections



TL/EE/11267–33

FIGURE 3-21. Read Cycle Timing



TL/EE/11267–34

FIGURE 3-22. Write Cycle Timing

At this time the signals $\overline{\text{TSO}}$ (Timing State Output), $\overline{\text{DBE}}$ (Data Buffer Enable) and either $\overline{\text{RD}}$ (Read Strobe) or $\overline{\text{WR}}$ (Write Strobe) will also be activated.

The T3 state provides for access time requirements, and it occurs at least once in a bus cycle. At the end of T2, on the rising edge of CTTL, the $\overline{\text{CWAIT}}$ signal is sampled to determine whether the bus cycle will be extended. See Section 3.5.5.3.

If the CPU is performing a read cycle, the data bus (AD0–AD15) is sampled at the beginning of T4 on the rising edge of CTTL. Data must, however, be held a little longer to meet the data hold time requirements. The $\overline{\text{RD}}$ signal is guaranteed not to go inactive before this time, so its rising edge can be safely used to disable the device providing the input data.

The T4 state finishes the bus cycle. At the beginning of T4, the $\overline{\text{RD}}$ or $\overline{\text{WR}}$, and $\overline{\text{TSO}}$ signals go inactive, and on the falling edge of CTTL, $\overline{\text{DBE}}$ goes inactive, having provided for necessary data hold times. Data during Write cycles remains valid from the CPU throughout T4. Note that the Bus Status lines (ST0–ST3) change at the beginning of T4, anticipating the following bus cycle (if any).

3.5.5.3 Cycle Extension

To allow sufficient access time for any speed of memory or peripheral device, the NS32FX164 provides for extension of a bus cycle. Any type of bus cycle except a Slave Processor cycle and a special bus cycle can be extended.

In *Figures 3-21* and *3-22*, note that during T3 all bus control signals from the CPU are flat. Therefore, a bus cycle can be cleanly extended by causing the T3 state to be repeated. This is the purpose of the $\overline{\text{CWAIT}}$ input signal.

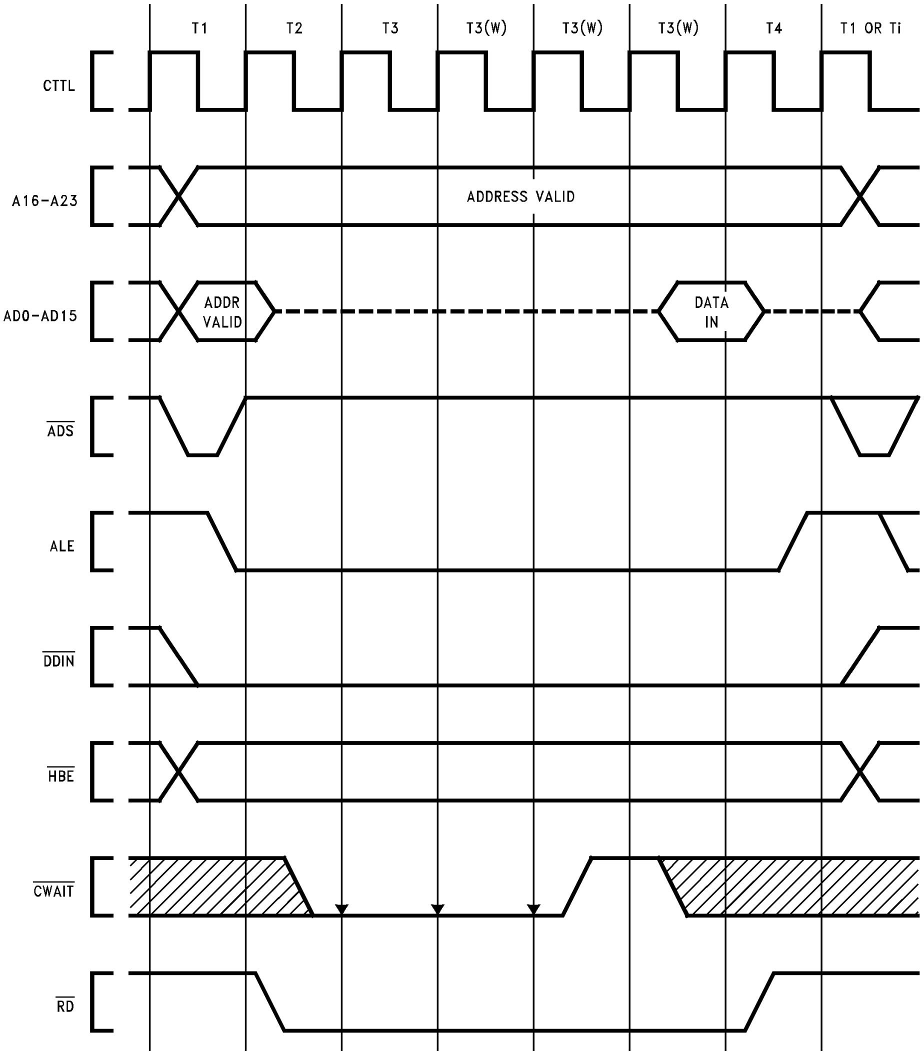
At the end of state T2, on the rising edge of CTTL, $\overline{\text{CWAIT}}$ is sampled.

$\overline{\text{CWAIT}}$ causes wait states to be inserted continuously as long as it is sampled active. It is normally used when the number of wait states to be inserted in the CPU bus cycle is not known in advance.

The following sequence shows the CPU response to the $\overline{\text{WAIT}}$1–2 and $\overline{\text{CWAIT}}$ inputs.

1. Start bus cycle.
2. Sample $\overline{\text{CWAIT}}$ at the end of state T2.
3. If $\overline{\text{CWAIT}}$ is not active, then go to step 6.
4. Insert one wait state.
5. Sample $\overline{\text{CWAIT}}$ again, then go to step 3.
6. Complete bus cycle.

Figure 3-23 shows a bus cycle extended by three wait states due to $\overline{\text{CWAIT}}$.



TL/EE/11267–35

FIGURE 3-23. Cycle Extension of a Read Cycle

3.5.5.4 Instruction Fetch Cycles

Instructions for the NS32FX164 CPU are "prefetched"; that is, they are input before being needed into the next available entry of the eight-byte instruction Queue. The CPU performs two types of instruction Fetch cycles: Sequential and Non-Sequential. These can be distinguished from each other by their differing status combinations on pins ST0–ST3 (Section 3.5.5.1).

A Sequential Fetch will be performed by the CPU whenever the Data Bus would otherwise be idle and the Instruction Queue is not currently full. Sequential Fetches are always Even Word Read cycles (Table 3-5).

A Non-Sequential Fetch occurs as a result of any break in the normally sequential flow of a program. Any jump or branch instruction, a trap or an interrupt will cause the next Instruction Fetch cycle to be Non-Sequential. In addition, certain instructions flush the instruction queue, causing the next instruction fetch to display Non-Sequential status. Only the first bus cycle after a break displays Non-Sequential status, and that cycle is either an Even Word Read or an Odd Byte Read, depending on whether the distination address is even or odd.

3.0 Functional Description (Continued)

3.5.5.5 Interrupt Control Cycles

Activating the $\overline{\text{INT}}$ or $\overline{\text{NMI}}$ pin on the CPU will initiate one or more bus cycles whose purpose in interrupt control rather than the tranfer of instructions or data. Execution of the Return from Interrupt Instruction (RETI) will also cause Interrupt Control bus cycles. These differ from instruction or data transfers only in the status presented on pins ST0–ST3. All Interrupt Control cycles are single-byte Read cycles.

Table 3-4 shows the Interrupt Control sequences associated with each interrupt and with the return from its service routine. For full details of the NS32FX164 interrupt structure, see Section 3.2.

TABLE 3-4. Interrupt Sequences

Cycle	Status	Address	$\overline{\text{DDIN}}$	$\overline{\text{HBE}}$	A0	High Bus	Low Bus
A. Non-Maskable Interrupt Control Sequence							
Interrupt Acknowledge							
1	0100	FFFF00$_{16}$	0	1	0	Don't Care	Don't Care
Interrupt Return							
None: Performed through Return from Trap (RETT) instruction.							
B. Non-Vectored Interrupt Control Sequence							
Interrupt Acknowledge							
1	0100	FFFE00$_{16}$	0	1	0	Don't Care	Don't Care
Interrupt Return							
None: Performed through Return from Trap (RETT) instruction.							
C. Vectored Interrupt Sequence: Non-Cascaded							
Interrupt Acknowledge							
1	0100	FFFE00$_{16}$	0	1	0	Don't Care	Vector: Range: 0–127
Interrupt Return							
1	0110	FFFE00$_{16}$	0	1	0	Don't Care	Vector: Same as in Previous Int. Ack. Cycle
D. Vectored Interrupt Sequence: Cascaded							
Interrupt Acknowledge							
1	0100	FFFE00$_{16}$	0	1	0	Don't Care	Cascade Index: range −16 to −1
(The CPU here uses the Cascade Index to find the Cascade Address.)							
2	0101	Cascade Address	0	1 or 0*	0 or 1*	Vector, range 0–255; on appropriate half or Data Bus for even/odd address	
Interrupt Return							
1	0110	FFFE00$_{16}$	0	1	0	Don't Care	Cascade Index: same as in previous Int. Ack. Cycle
(The CPU here uses the Cascade Index to find the Cascade Address.)							
2	0111	Cascade Address	0	1 or 0*	0 or 1*	Don't Care	Don't Care

* If the Cascaded ICU Address is Even (A0 is low), then the CPU applies $\overline{\text{HBE}}$ high and reads the vector number from bits 0–7 of the Data Bus.

If the address is Odd (A0 is high), then the CPU applies $\overline{\text{HBE}}$ low and reads the vector number from bits 8–15 of the Data Bus. The vector number may be in the range 0–225.

3.5.5.6 Special Bus Cycles

Special bus cycles are performed during CPU accesses to the DSP Module (DSPM) registers or internal RAM. These cycles may be used by external logic to track CPU activities involving on-chip bus transactions.

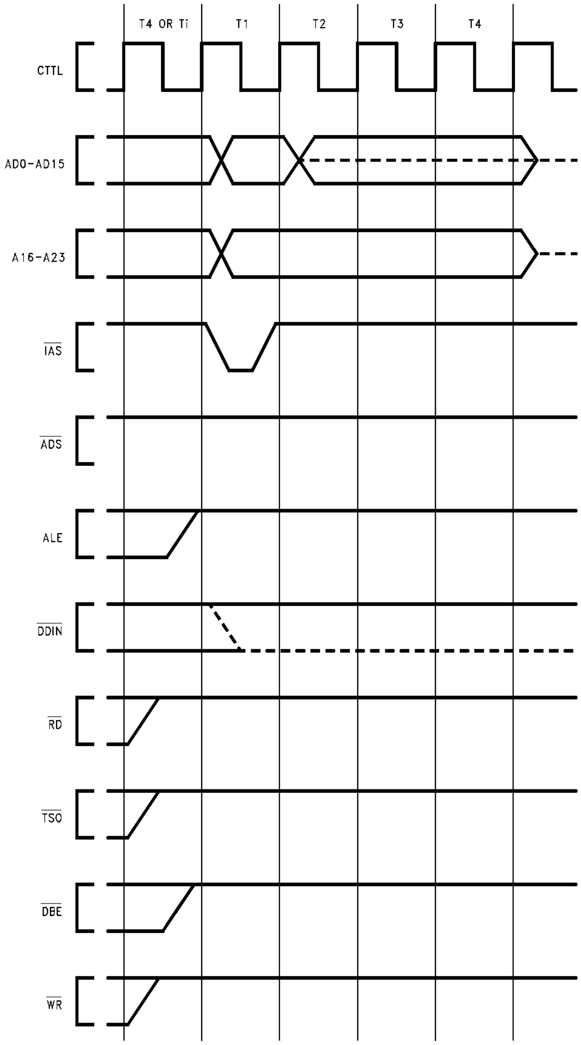
A special bus cycle starts with the assertion of the special output signal $\overline{IAS}$. The ALE signal stays high during the entire cycle, and the signals $\overline{ADS}$, $\overline{TSO}$, $\overline{DBE}$, $\overline{RD}$ and $\overline{WR}$ are not activated. $\overline{CWAIT}$ is ignored.

A CPU access to a DSP Module register or internal RAM occurring while a vector operation is being executed, is delayed until the end of the vector operation. This delay cannot be observed externally.

The CPU drives the data bus with the same data that is being written into the on-chip register or RAM during a special write cycle, and ignores the data placed on the data bus during a special read cycle. The 24 least significant address bits of the DSPM register being accessed are output on the AD0–AD15 and A16–A23 signals. *Figure 3-24*. shows the timing for special read and write cycles.

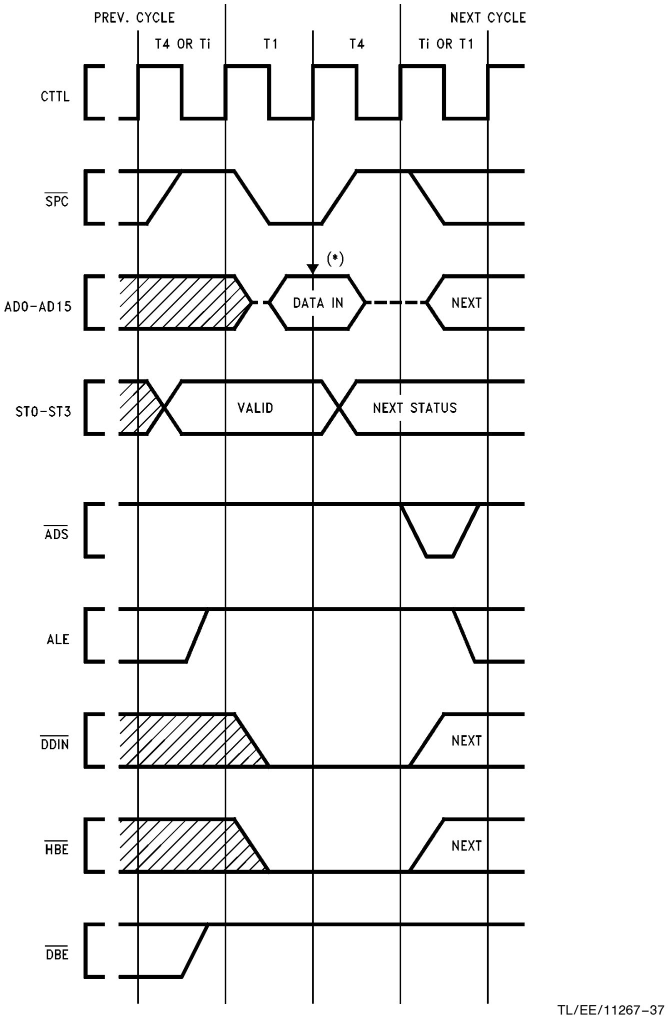
3.5.5.7 Slave Processor Bus Cycles

A Slave Processor bus cycle always takes exactly two clock cycles, labeled T1 and T4 (see *Figures 3-25* and *3-26*). During a Read cycle $\overline{SPC}$ is active from the beginning of T1 to the beginning of T4, and the data is sampled at the end of T1. The Cycle Status pins lead the cycle by one clock period, and are sampled on the leading edge of $\overline{SPC}$. During a Write cycle, the CPU applies data and activates $\overline{SPC}$ at T1, removing $\overline{SPC}$ at T4. The Slave Processor latches the status on the leading edge of $\overline{SPC}$ and latches data on the trailing edge.



TL/EE/11267–36

FIGURE 3-24. Special Bus Cycle Timing



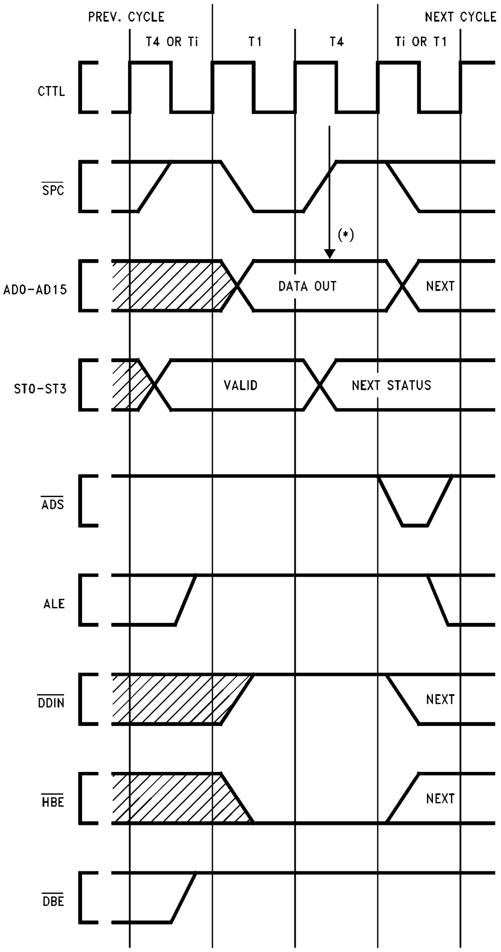
Note: CPU samples Data Bus here.

FIGURE 3-25. Slave Processor Read Cycle

The CPU does not pulse the Address Strobe ($\overline{\text{ADS}}$), and no bus signals are generated. The direction of a transfer is determined by the sequence (''protocol'') established by the instruction under execution; but the CPU indicates the direction on the $\overline{\text{DDIN}}$ pin for hardware debugging purposes.

A Slave Processor operand is transferred in one or more Slave bus cycles. A Byte operand is transferred on the least-significant byte of the Data Bus (AD0–AD7), and a Word operand is transferred on the entire bus. A Double Word is transferred in a consecutive pair of bus cycles, least-significant word first. A Quad Word is transferred in two pairs of Slave cycles, with other bus cycles possibly occurring between them. The word order is from least-significant word to most-significant.

Figure 3-27 shows the NS32FX164 and FPU connection diagram.



TL/EE/11267-38

*Note: Slave Processor samples Data Bus here.

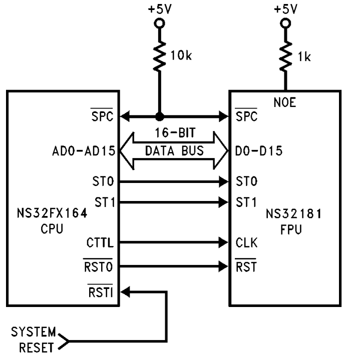
FIGURE 3-26. Slave Processor Write Cycle

3.5.5.8 Data Access Sequences

The 24-bit address provided by the NS32FX164 is a byte address; that is, it uniquely identifies one of up to 16,777,216 8-bit memory locations. An important feature of the NS32FX164 is that the presence of a 16-bit data bus imposes no restrictions on data alignment; any data item, regardless of size, may be placed starting at any memory address. The NS32FX164 provides a special control signal, High Byte Enable ($\overline{\text{HBE}}$), which facilitates individual byte addressing on a 16-bit bus.

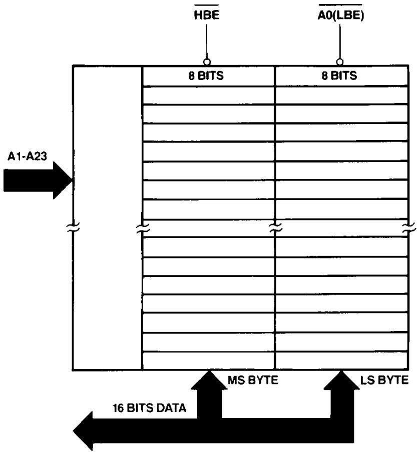
Memory is organized as two 8-bit banks, each bank receiving the word address (A1–A23) in parallel. One bank, connected to Data Bus pins AD0–AD7, is enabled to respond to even byte addresses; i.e., when the least significant address bit (A0) is low. The other bank, connected to Data Bus pins AD8–AD15, is enabled when $\overline{\text{HBE}}$ is low. See *Figure 3-28*.

Any bus cycle falls into one of three categories: Even Byte Access, Odd Byte Access, and Even Word Access. All accesses to any data type are made up of sequences of these cycles. Table 3-5 gives the state of A0 and $\overline{\text{HBE}}$ for each category.



TL/EE/11267-39

FIGURE 3-27. NS32FX164 and FPU Interconnections



TL/EE/11267-40

FIGURE 3-28. Memory Interface

TABLE 3-5. Bus Cycle Categories

Category	$\overline{\text{HBE}}$	A0
Even Byte	1	0
Odd Byte	0	1
Even Word	0	0

Accesses of operands requiring more than one bus cycle are performed sequentially, with no idle T-states separating them. The number of bus cycles required to transfer an operand depends on its size and its alignment (i.e., whether it starts on an even byte address or an odd byte address). Table 3-6 lists the bus cycles performed for each situation. For the timing of A0 and $\overline{\text{HBE}}$, see Section 3.5.5.2.

TABLE 3-6. Data Access Sequences

A. Odd Word Access Sequence

Cycle	Type	Address	$\overline{\text{HBE}}$	A0	High Bus	Low Bus	
					Byte 1	Byte 0	← A
1	Odd Byte	A	0	1	Byte 0	Don't Care	
2	Even Byte	A + 1	1	0	Don't Care	Byte 1	

B. Even Double-Word Access Sequence

			Byte 3	Byte 2	Byte 1	Byte 0	← A
1	Even Word	A	0	0	Byte 1	Byte 0	
1	Even Word	A + 2	0	0	Byte 3	Byte 2	

C. Odd Double-Word Access Sequence

			Byte 3	Byte 2	Byte 1	Byte 0	← A
1	Odd Byte	A	0	1	Byte 0	Don't Care	
2	Even Word	A + 1	0	0	Byte 2	Byte 1	
3	Even Byte	A + 3	1	0	Don't Care	Byte 3	

D. Even Quad-Word Access Sequence

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	← A

1	Even Word	A	0	0	Byte 1	Byte 0
2	Even Word	A + 2	0	0	Byte 3	Byte 2

Other Bus Cycles (Instruction Prefetch or Slave) can occur here.

3	Even Word	A + 4	0	0	Byte 5	Byte 4
4	Even Word	A + 6	0	0	Byte 7	Byte 6

E. Odd Quad-Word Access Sequence

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	← A

1	Odd Byte	A	0	1	Byte 0	Don't Care
2	Even Word	A + 1	0	0	Byte 2	Byte 1
3	Even Byte	A + 3	1	0	Don't Care	Byte 3

Other Bus Cycles (Instruction Prefetch or Slave) can occur here.

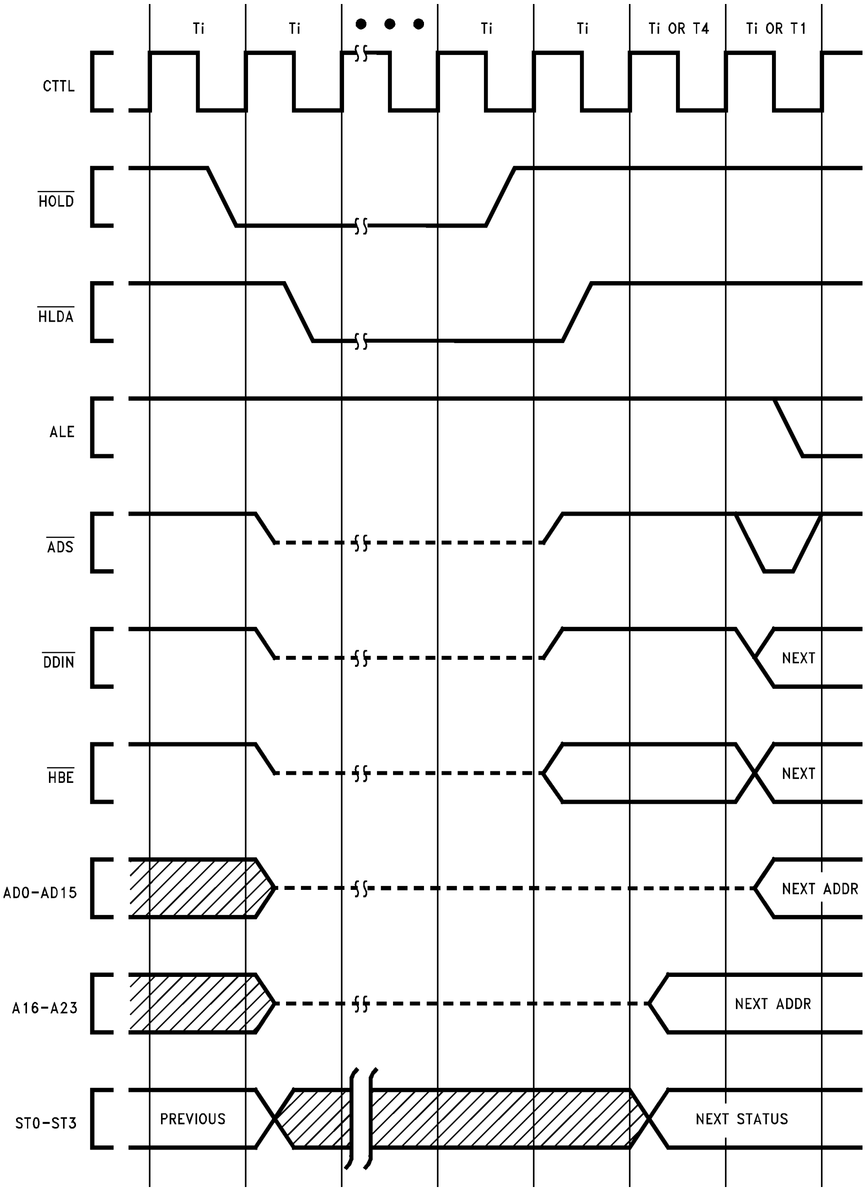
4	Odd Byte	A + 4	0	1	Byte 4	Don't Care
5	Even Word	A + 5	0	0	Byte 6	Byte 5
6	Even Byte	A + 7	1	0	Don't Care	Byte 7

3.5.5.9 Bus Access Control

The NS32FX164 CPU has the capability of relinquishing its control of the bus upon request from a DMA controller or another CPU. This capability is implemented by means of the $\overline{\text{HOLD}}$ (Hold Request) and $\overline{\text{HLDA}}$ (Hold Acknowledge) pins. By asserting $\overline{\text{HOLD}}$ low, an external device requests access to the bus. On receipt of $\overline{\text{HLDA}}$ from the CPU, the device may perform bus cycles, as the CPU at this point has set AD0–AD15, A16–A23 and $\overline{\text{HBE}}$ to the TRI-STATE® condition and has switched $\overline{\text{ADS}}$ and $\overline{\text{DDIN}}$ to the input mode. ALE is asserted in T4, and stays high during the time the bus is granted. The CPU now monitors $\overline{\text{ADS}}$ and $\overline{\text{DDIN}}$ from the external device to generate the relevant strobe signals (i.e., $\overline{\text{TSO}}$, $\overline{\text{DBE}}$, $\overline{\text{RD}}$ or $\overline{\text{WR}}$). To return control of the bus to the CPU, the device sets $\overline{\text{HOLD}}$ inactive, and the CPU acknowledges it by setting $\overline{\text{HLDA}}$ inactive.

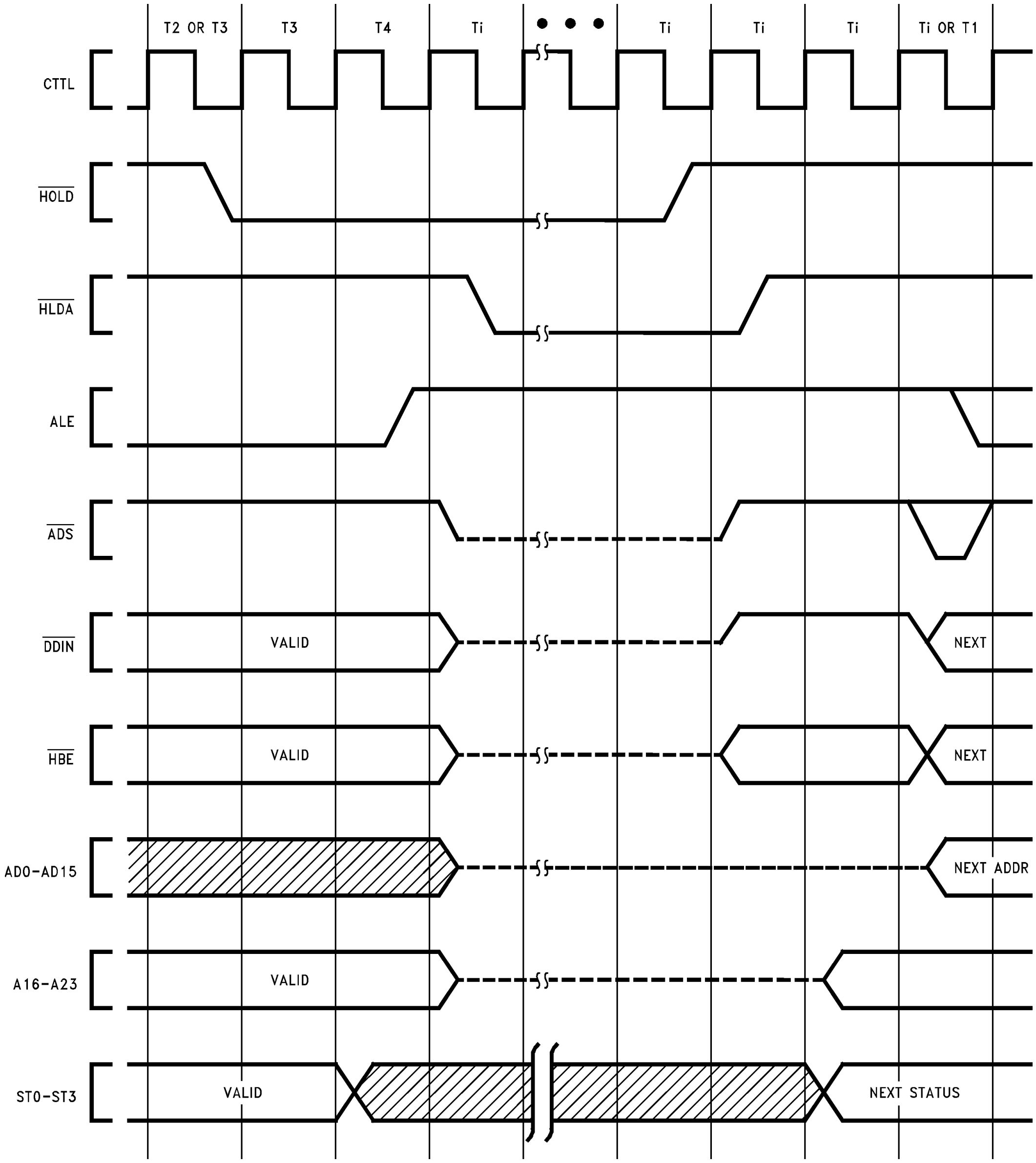
How quickly the CPU releases the bus depends on whether it is idle on the bus at the time the $\overline{\text{HOLD}}$ request is made, as the CPU must always complete the current bus cycle. *Figure 3-29* shows the timing sequence when the CPU is idle. In this case, the CPU grants the bus during the immediately following clock cycle. *Figure 3-30* shows the sequence when the CPU is using the bus at the time the $\overline{\text{HOLD}}$ request is made. If the request is made during or before the clock cycle shown (two clock cycles before T4), the CPU will release the bus during the clock cycle following T4. If the request occurs closer to T4, the CPU may already have decided to initiate another bus cycle. In that case it will not grant the bus until after the next T4 state. Note that this situation will also occur if the CPU is idle on the bus but has initiated a bus cycle internally.

Note 1: The logic value of the status pins, ST0–3, is undefined during DMA activity.



TL/EE/11267–41

FIGURE 3-29. $\overline{\text{HOLD}}$ Timing (Bus Initially Idle)



TL/EE/11267–42

FIGURE 3-30. HOLD Timing (Bus Initially Not Idle)

3.0 Functional Description (Continued)

3.5.5.10 Instruction Status

In addition to the four bits of Bus Cycle status (ST0–3), the NS32FX164 CPU also presents Instruction Status information on three separate pins. These pins differ from ST0–3 in that they are synchronous to the CPU's internal instruction execution section rather than to its bus interface section.

$\overline{\text{PFS}}$ (Program Flow Status) is pulsed low as each instruction begins execution. It is intended for debugging purposes.

U/$\overline{\text{S}}$ originates from the U-bit of the Processor Status Register, and indicates whether the CPU is currently running in User or Supervisor mode. Although it is not synchronous to bus cycles, there are guarantees on its validity during any given bus cycle. See the Timing Specifications in Section 4.4.2.

$\overline{\text{ILO}}$ (Interlocked Operation) is activated during an SBITI (Set Bit, Interlocked) or CBITI (Clear Bit, Interlocked) instruction. It is made available to external bus arbitration circuitry in order to allow these instructions to implement the semaphore primitive operations for multi-processor communication and resource sharing. $\overline{\text{ILO}}$ is guaranteed to be active during the operand accesses performed by the interlocked instructions.

Note: The acknowledge of $\overline{\text{HOLD}}$ is on a cycle by cycle basis. Therefore, it is possible to have $\overline{\text{HLDA}}$ active when an interlock operation is in progress. In this case, $\overline{\text{ILO}}$ remains low and the interlocked instruction continues only after $\overline{\text{HOLD}}$ is de-asserted.

4.0 Device Specifications

4.1 NS32FX164 PIN DESCRIPTIONS

The following is a brief description of all NS32FX164 pins. The descriptions reference portions of the Functional Description, Section 3.0.

Note: An asterisk next to the signal name indicates a TRI-STATE condition for that signal during $\overline{\text{HOLD}}$ acknowledge.

4.1.1 Supplies

V_{CC} **Power.**

+5V positive supply.

GND **Ground.**

Ground reference for both on-chip logic and output drivers.

4.1.2 Input Signals

$\overline{\text{RSTI}}$ **Reset Input.**

Schmitt triggered, asynchronous signal used to generate a CPU reset. See Section 3.5.4.

Note: The reset signal is a true asynchronous input. Therefore, no external synchronizing circuit is needed.

$\overline{\text{HOLD}}$ **Hold Request.**

When active, causes the CPU to release the bus for DMA or multiprocessing purposes. See Section 3.5.5.9.

Note: If the $\overline{\text{HOLD}}$ signal is generated asynchronously, its set up and hold times may be violated. In this case, it is recommended to synchronize it with CTTL to minimize the possibility of metastable states.

The CPU provides only one synchronization stage to minimize the $\overline{\text{HLDA}}$ latency. This is to avoid speed degradations in cases of heavy $\overline{\text{HOLD}}$ activity (i.e., DMA controller cycles interleaved with CPU cycles).

$\overline{\text{INT}}$ **Interrupt.**

A low level on this pin requests a maskable interrupt. $\overline{\text{INT}}$ must be kept asserted until the interrupt is acknowledged.

$\overline{\text{NMI}}$ **Non-Maskable Interrupt.**

A High-to-Low transition on this signal requests a non-maskable interrupt.

Note: $\overline{\text{INT}}$ and $\overline{\text{NMI}}$ are true asynchronous inputs. Therefore, no external synchronizing circuit is needed.

$\overline{\text{CWAIT}}$ **Continuous Wait.**

Causes the CPU to insert continuous wait states if sampled low at the end of T2 and each following T-State. See Section 3.5.5.3.

OSCIN **Crystal/External Clock Input.**

Input from a crystal or an external clock source. See Section 3.5.2.

4.1.3 Output Signals

A16–A23 *__High-Order Address Bits.__

These are the most significant 8 bits of the memory address bus.

$\overline{\text{HBE}}$ *__High Byte Enable.__

Status signal used to enable data transfers on the most significant byte of the data bus.

ST0–3 **Status.**

Bus cycle status code; ST0 is the least significant. Encodings are:

0000— Idle: CPU Inactive on Bus.
0001— Idle: WAIT Instruction.
0010— DSP Module Data Transfer.
0011— Idle: Waiting for Slave.
0100— Interrupt Acknowledge, Master.
0101— Interrupt Acknowledge, Cascaded.
0110— End of Interrupt, Master.
0111— End of Interrupt, Cascaded.
1000— Sequential Instruction Fetch.
1001— Non-Sequential Instruction Fetch.
1010— Data Transfer.
1011— Read Read-Modify-Write Operand.
1100— Read for Effective Address.
1101— Transfer Slave Operand.
1110— Read Slave Status Word.
1111— Broadcast Slave ID.

U/$\overline{\text{S}}$ **User/Supervisor.**

User or Supervisor Mode status. High indicates User Mode; low indicates Supervisor Mode.

$\overline{\text{ILO}}$ **Interlocked Operation.**

When active, indicates that an interlocked operation is being executed.

$\overline{\text{HLDA}}$ **Hold Acknowledge.**

Activated by the CPU in response to the $\overline{\text{HOLD}}$ input to indicate that the CPU has released the bus.

$\overline{\text{PFS}}$ **Program Flow Status.**

A pulse on this signal indicates the beginning of execution of an instruction.

$\overline{\text{BPU}}$ **BPU Cycle.**

This signal is activated during a bus cycle to enable an external BITBLT processing unit. The EXTBLT instruction activates this signal.

Note: $\overline{\text{BPU}}$ is low (Active) only during bus cycles involving prefetching instructions and execution of EXTBLT operands. It is recommended that $\overline{\text{BPU}}$, $\overline{\text{ADS}}$ and status lines (ST0–ST3) be used to qualify BPU bus cycles. If a DMA circuit exists in the system, the $\overline{\text{HLDA}}$ signal should be used to further qualify BPU cycles. $\overline{\text{BPU}}$ may become active during T4 of a non-BPU bus cycle, and may become inactive during T4 of a BPU bus cycle. $\overline{\text{BPU}}$ must be qualified by $\overline{\text{ADS}}$ and status lines (ST0–ST3) to be used as an external gating signal.

$\overline{\text{RSTO}}$ **Reset Output.**

This signal becomes active when $\overline{\text{RSTI}}$ is low, initiating a system reset.

$\overline{\text{RD}}$ **Read Strobe.**

Activated during CPU or DMA read cycles to enable reading of data from memory or peripherals. See Section 3.5.5.2.

$\overline{\text{WR}}$ **Write Strobe.**

Activated during CPU or DMA write cycles to enable writing of data to memory or peripherals.

$\overline{\text{TSO}}$ **Timing State Output.**

The falling edge of $\overline{\text{TSO}}$ identifies the beginning of state T2 of a bus cycle. The rising edge identifies the beginning of state T4.

$\overline{\text{DBE}}$ **Data Buffers Enable.**

Used to control external data buffers. It is active when the data buffers are to be enabled.

OSCOUT **Crystal Output.**

This line is used as the return path for the crystal (if used). When an external clock source is used, OSCOUT should be left unconnected or loaded with no more than 5 pF of stray capacitance.

$\overline{\text{IAS}}$ **Special Cycle Address Strobe.**

Signals the beginning of a special bus cycle.

CTTL1–2 **System Clock.**

Output clock for bus timing. CTTL1 and CTTL2 must be externally connected together.

FCLK **Fast Clock.**

This clock is derived from the clock waveform on OSCIN. Its frequency is either the same as OSCIN or is lower, depending upon the scale factor programmed into the CFG register.

ALE **Address Latch Enable.**

Active high signal that can be used to control external address latches.

$\overline{\text{IOUT}}$ **Interrupt Output**

Activated when the execution of a command list stops and the associated interrupt is enabled.

4.1.4 Input-Output Signals

AD0–15 *__Address/Data Bus.__*

Multiplexed Address/Data Information. Bit 0 is the least significant bit of each.

$\overline{\text{SPC}}$ **Slave Processor Control.**

Used by the CPU as the data strobe output for slave processor transfers; used by a slave processor to acknowledge completion of a slave instruction. See Section 3.5.5.7.

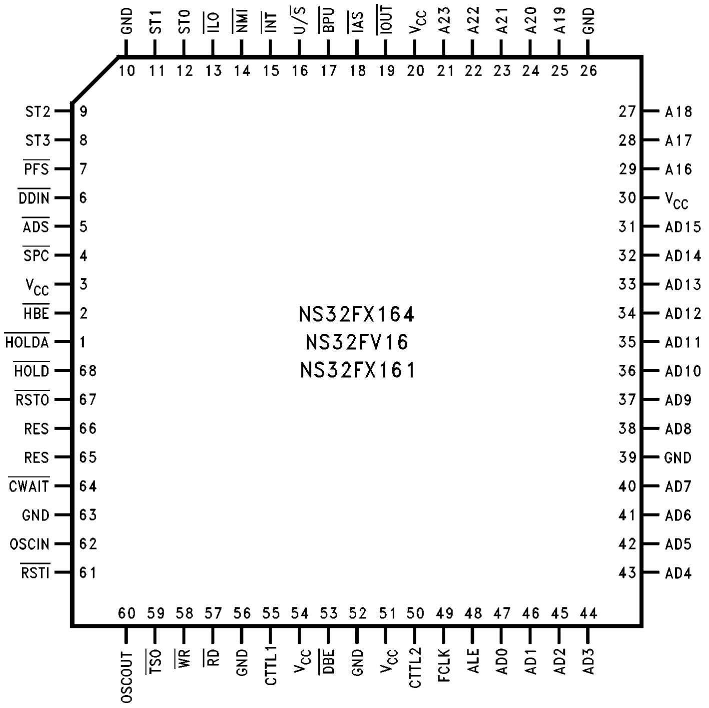
$\overline{\text{DDIN}}$ ***Data Direction.**

Status signal indicating the directon of the data transfer during a bus cycle. During $\overline{\text{HOLD}}$ acknowledge this signal becomes an input and determines the activation of $\overline{\text{RD}}$ or $\overline{\text{WR}}$.

$\overline{\text{ADS}}$ ***Address Strobe**

Controls address latches; signals the beginning of a bus cycle. During $\overline{\text{HOLD}}$ acknowledge this signal becomes an input and the CPU monitors it to detect the beginning of a DMA cycle and generate the relevant strobe signals. When a DMA is used, $\overline{\text{ADS}}$ should be pulled up to V_{CC} through a 10 kΩ resistor.

68-Pin PCC Package



TL/EE/11267–43

Bottom View

Order Number NS32FX164V-20, NS32FX164V-25, NS32FV16-20, NS32FV16-25, NS32FX161V-15 or NS32FX161-20
NS Package Number V68A

FIGURE 4-1. Connection Diagram

Note: Pins 65 and 66 must be connected to GND or V_{CC}

4.0 Device Specifications (Continued)

4.2 ABSOLUTE MAXIMUM RATINGS

If Military/Aerospace specified devices are required, please contact the National Semiconductor Sales Office/Distributors for availability and specifications.

Temperature under Bias	0°C to +70°C
Storage Temperature	−65°C to +150°C
All Input or Output Voltages with Respect to GND	−0.5V to +6.5V

Note: *Absolute maximum ratings indicate limits beyond which permanent damage may occur. Continuous operation at these limits is not intended; operation should be limited to those conditions specified under Electrical Characteristics.*

4.3 ELECTRICAL CHARACTERISTICS T_A = 0°C to +70°C, V_{CC} = 5V ±10%, GND = 0V

Symbol	Parameter	Conditions	Min	Typ	Max	Units
V_{IH}	High Level Input Voltage		2.0		V_{CC} + 0.5	V
V_{IL}	Low Level Input Voltage		−0.5		0.8	V
V_{XL}	OSCIN Input Low Voltage				0.5	V
V_{XH}	OSCIN Input High Voltage		3.8			V
V_{RIH}	$\overline{\text{RSTI}}$ High Level Input Voltage		Max (3.5, V_{CC} − 1.5)		V_{CC} + 0.5	V
V_{RIL}	$\overline{\text{RSTI}}$ Low Level Input Voltage		−0.5		0.7	V
V_{RHYS}	$\overline{\text{RSTI}}$ Hysteresis Loop Width (Note 3)		0.5			V
V_{HYS}	$\overline{\text{INT}}$, $\overline{\text{NMI}}$ Hysteresis Loop Width (Note 3)		0.2			V
V_{OH}	High Level Output Voltage	I_{OH} = −400 μA	2.4			V
V_{OL}	Low Level Output Voltage	I_{OL} = 4 mA			0.45	V
I_{ILS}	$\overline{\text{SPC}}$ Input Current (Low)	V_{IN} = 0.4V, $\overline{\text{SPC}}$ in Input Mode			1.0	mA
I_I	Input Load Current	$0 \leq V_{IN} \leq V_{CC}$, All Inputs except $\overline{\text{SPC}}$	−20		20	μA
I_L	Leakage Current Output and I/O Pins in TRI-STATE or Input Mode	$0.4 \leq V_{OUT} \leq V_{CC}$	−20		20	μA
I_{CC}	Active Supply Current	I_{OUT} = 0, T_A = 25°C (Note 2)		200		mA

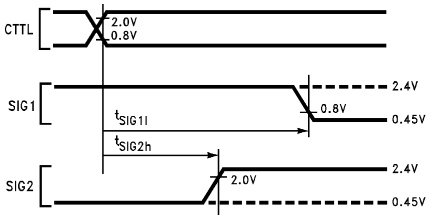
Note 1: Care should be taken by designers to provide a minimum inductance path between the GND pins and system ground in order to minimize noise.

Note 2: I_{CC} is affected by the clock scaling factor selected by the C- and M-bits in the CFG register, see Section 3.5.3.

4.4 SWITCHING CHARACTERISTICS

4.4.1 Definitions

All the timing specifications given in this section refer to 0.8V or 2.0V on the rising or falling edges of all the signals as illustrated in *Figures 4-2* and *4-3* unless specifically stated otherwise. The capacitive load is assumed to be 100 pF on CTTL and 50 pF on all the other output signals.

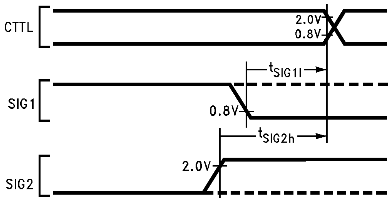


TL/EE/11267–44

FIGURE 4-2. Output Signals Specification Standard

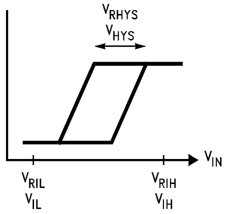
Abbreviations:

L.E.— Leading Edge	R.E.— Rising Edge
T.E.— Traling Edge	F.E.— Falling Edge



TL/EE/11267–45

FIGURE 4-3a. Input Signals Specification Standard



TL/EE/11267–71

FIGURE 4-3b. $\overline{\text{RSTI}}$, $\overline{\text{INT}}$, $\overline{\text{NMI}}$ Hysteresis

4.0 Device Specifications (Continued)

4.4.2 Timing Tables

4.4.2.1 Output Signals: Internal Propagation Delays, NS32FX161-15, NS32FX164-20, NS32FX164-25

- The output to input timings (e.g., address to data-in) are at least 2 ns better than the worst case values calculated from the output valid and input setup times relative to CTTL.

Symbol	Figure	Description	Reference/ Conditions	NS32FX161-15		NS32FX164-20		NS32FX164-25		Units
				Min	Max	Min	Max	Min	Max	
t_{CTp}	4-15	CTTL Clock Period	R.E., CTTL to Next R.E., CTTL	66	1000	50	1000	40	1000	ns
t_{CTh}	4-15	CTTL High Time	At 2.0V (Both Edges)	0.5 t_{CTp} − 6 ns		0.5 t_{CTp} − 5 ns		0.5 t_{CTp} − 5 ns		
t_{CTl}	4-15	CTTL Low Time	At 0.8V (Both Edges)	0.5 t_{CTp} − 6 ns		0.5 t_{CTp} − 5 ns		0.5 t_{CTp} − 4 ns		
t_{CTr}	4-15	CTTL Rise Time	0.8V to 2.0V on R.E., CTTL		6		5		4	ns
t_{CTf}	4-15	CTTL Fall Time	2.0V to 0.8V on F.E., CTTL		6		5		4	ns
t_{XCTd}	4-15	OSCIN to CTTL Delay	4.2V on R.E., OSCIN to R.E., CTTL		29		29		25	ns
t_{XFr}	4-15	OSCIN to FCLK R.E. Delay	4.2V on R.E., OSCIN to R.E., FCLK		25		20		15	ns
t_{FCr}	4-15	FCLK to CTTL R.E. Delay	R.E., FCLK to R.E., CTTL		10		10		10	ns
t_{FCf}	4-15	FCLK to CTTL F.E. Delay	R.E., FCLK to F.E., CTTL		10		10		10	ns
t_{ALv}	4-4	AD0–AD15 Valid (Note 5)	After R.E., CTTL T1		14		13		12	ns
t_{ALh}	4-4	AD0–AD15 Hold	After R.E., CTTL T2	0		0		0		ns
t_{AHv}	4-4	A16–A23 Valid (Note 5)	After R.E., CTTL T1		14		13		12	ns
t_{AHh}	4-4	A16–A23 Hold	After R.E., CTTL Next T1 or Ti	0		0		0		ns
t_{ALfr}	4-4	AD0–AD15 Floating (during Read)	After R.E., CTTL T2		14		13		12	ns
t_{ALf}	4-7	AD0–AD15 Floating	After R.E., CTTL Ti		14		13		12	ns
t_{AHf}	4-7	A16–A23 Floating	After R.E., CTTL Ti		14		13		12	ns
t_{Dv}	4-5	Data Valid (Write Cycle)	After R.E., CTTL T2 or T1		14		13		12	ns
t_{Dh}	4-5	Data Hold	After R.E., CTTL Next T1 or Ti	0		0		0		ns
t_{ADSa}	4-4	$\overline{\text{ADS}}$ Signal Active	After R.E., CTTL T1		14		13		12	ns
t_{ADSia}	4-4	$\overline{\text{ADS}}$ Signal Inactive (Note 4)	After R.E., CTTL T1	0.5 t_{CTp} −6 ns	0.5 t_{CTp} +16 ns	0.5 t_{CTp} −6 ns	0.5 t_{CTp} +15 ns	0.5 t_{CTp} −6 ns	0.5 t_{CTp} +14 ns	
t_{ADSw}	4-5	$\overline{\text{ADS}}$ Pulse Width	At 0.8V (Both Edges)	20		15		10		ns
t_{ADSf}	4-7	$\overline{\text{ADS}}$ Floating	After R.E., CTTL Ti		14		13		12	ns
t_{ALADSs}	4-4	AD0–AD15 Setup	Before $\overline{\text{ADS}}$ T.E.	10		10		10		ns
t_{HBEv}	4-4	$\overline{\text{HBE}}$ Signal Valid	After R.E., CTTL T1		14		13		12	ns
t_{HBEh}	4-4	$\overline{\text{HBE}}$ Signal Hold	After R.E., CTTL Next T1 or Ti	0		0		0		ns
t_{HBEf}	4-7	$\overline{\text{HBE}}$ Signal Floating	After R.E., CTTL Ti		14		13		12	ns

4.0 Device Specifications (Continued)

4.4.2 Timing Tables (Continued)

4.4.2.1 Output Signals: Internal Propagation Delays, NS32FX161-15, NS32FX164-20, NS32FX164-25

Symbol	Figure	Description	Reference/ Conditions	NS32FX161-15		NS32FX164-20		NS32FX164-25		Units
				Min	Max	Min	Max	Min	Max	
t_{DDINv}	4-4	$\overline{\text{DDIN}}$ Signal Valid	After R.E., CTTL T1		14		13		12	ns
t_{DDINh}	4-4	$\overline{\text{DDIN}}$ Signal Hold	After R.E., CTTL Next T1 or Ti	0		0		0		ns
t_{DDINf}	4-7	$\overline{\text{DDIN}}$ Floating	After R.E., CTTL Ti		14		13		12	ns
t_{SPCa}	4-10	$\overline{\text{SPC}}$ Output Active	After R.E., CTTL T1		14		13		12	ns
t_{SPCia}	4-10	$\overline{\text{SPC}}$ Output Inactive	After R.E., CTTL T4		14		13		12	ns
t_{HLDAa}	4-7	$\overline{\text{HLDA}}$ Signal Active	After R.E., CTTL Ti		14		13		12	ns
t_{HLDAia}	4-8	$\overline{\text{HLDA}}$ Signal Inactive	After R.E., CTTL Ti		14		13		12	ns
t_{STv}	4-4	Status ST0–ST3 Valid	After R.E., CTTL T4 (Before T1, see Note 1)		14		13		12	ns
t_{STh}	4-4	Status ST0–ST3 Hold	After R.E., CTTL T4	0		0		0		ns
t_{BPUv}	4-4	$\overline{\text{BPU}}$ Signal Valid	After R.E., CTTL T4 or Ti		14		13		12	ns
t_{BPUh}	4-4	$\overline{\text{BPU}}$ Signal Hold	After R.E., CTTL T4 or Ti	0		0		0		ns
t_{TSOa}	4-4	$\overline{\text{TSO}}$ Signal Active	After R.E., CTTL T2		14		13		12	ns
t_{TSOia}	4-4	$\overline{\text{TSO}}$ Signal Inactive	After R.E., CTTL T4		14		13		12	ns
t_{RDa}	4-4	$\overline{\text{RD}}$ Signal Active	After R.E., CTTL T2		14		13		12	ns
t_{RDia}	4-4	$\overline{\text{RD}}$ Signal Inactive	After R.E., CTTL T4		14		13		12	ns
t_{WRa}	4-5	$\overline{\text{WR}}$ Signal Active	After R.E., CTTL T2		14		13		12	ns
t_{WRia}	4-5	$\overline{\text{WR}}$ Signal Inactive	After R.E., CTTL T4		14		13		12	ns
$t_{DBEa(R)}$	4-4	$\overline{\text{DBE}}$ Active (Read Cycle) (Note 4)	After R.E., CTTL T2	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +16 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +15 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +14 ns	
$t_{DBEa(W)}$	4-5	$\overline{\text{DBE}}$ Active (Write Cycle)	After R.E., CTTL T2		14		13		12	ns
t_{DBEia}	4-5, 4-6	$\overline{\text{DBE}}$ Inactive (Note 4)	After R.E., CTTL T4	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +16 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +15 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +14 ns	
t_{USv}	4-4	U/$\overline{\text{S}}$ Signal Valid	After R.E., CTTL T4		14		13		12	ns
t_{USh}	4-4	U/$\overline{\text{S}}$ Signal Hold	After R.E., CTTL T4	0		0		0		ns
t_{PFSa}	4-13	$\overline{\text{PFS}}$ Signal Active (Note 4)	After R.E., CTTL	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +16 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +15 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +14 ns	
t_{PFSia}	4-13	$\overline{\text{PFS}}$ Signal Inactive (Note 4)	After R.E., CTTL	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +16 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +15 ns	$0.5\ t_{CTp}$ −3 ns	$0.5\ t_{CTp}$ +14 ns	
t_{ALEa}	4-5	ALE Signal Active (Note 4)	After R.E., CTTL T4	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +16 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +15 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +14 ns	
t_{ALEia}	4-5	ALE Signal Inactive (Note 4)	After R.E., CTTL T1	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +16 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +15 ns	$0.5\ t_{CTp}$ −6 ns	$0.5\ t_{CTp}$ +14 ns	
T_{ALALEs}	4-5	AD0–AD15 Setup	Before ALE T.E.	10		10		10		ns

4.4.2 Timing Tables (Continued)

4.4.2.1 Output Signals: Internal Propagation Delays, NS32FX161-15, NS32FX164-20, NS32FX164-25

Symbol	Figure	Description	Reference/ Conditions	NS32FX161-15		NS32FX164-20		NS32FX164-25		Units
				Min	Max	Min	Max	Min	Max	
t_{IASa}	4-6	$\overline{\text{IAS}}$ Signal Active	After R.E., CTTL T1		14		13		12	ns
t_{IASia}	4-6	$\overline{\text{IAS}}$ Signal Inactive (Note 4)	After R.E., CTTL T1	0.5 t_{CTp} −6 ns	0.5 t_{CTp} +16 ns	0.5 t_{CTp} −6 ns	0.5 t_{CTp} +15 ns	0.5 t_{CTp} −6 ns	0.5 t_{CTp} +14 ns	
t_{IASw}	4-6	$\overline{\text{IAS}}$ Pulse Width	At 0.8V (Both Edges)	20		15		10		ns
t_{AIASs}	4-6	AD0−AD15 Setup	Before $\overline{\text{IAS}}$ T.E.	10		10		10		ns
t_{ILOa}	4-14	$\overline{\text{ILO}}$ Signal Active	After R.E., CTTL		14		13		12	ns
t_{ILOia}	4-14	$\overline{\text{ILO}}$ Signal Inactive	After R.E., CTTL		14		13		12	ns
t_{RSTOa}	4-19	$\overline{\text{RSTO}}$ Signal Active	After R.E., CTTL		14		13		12	ns
t_{RSTOia}	4-19	$\overline{\text{RSTO}}$ Signal Inactive	After R.E., CTTL		14		13		12	ns
t_{RTOI}	4-19	Reset to Idle (Note 3)	After F.E. of $\overline{\text{RSTO}}$		10		10		10	t_{CTp}
t_{IOUTv}	4-20	$\overline{\text{IOUT}}$ Signal Valid	After R.E. CTTL		14		13		12	ns
t_{IOUTh}	4-20	$\overline{\text{IOUT}}$ Signal Hold	After R.E. CTTL	0		0		0		ns

Note 1: Every memory cycle starts with T4, during which Cycle Status is applied. If the CPU was idling, the sequence will be ". . . Ti, T4, T1 . . .". If the CPU was not idling, the sequence will be ". . . T4, T1 . . .".

Note 2: The parameters related to the "floating/not floating" conditions are guaranteed by characterization. Due to tester conditions, these parameters are not 100% tested.

Note 3: Not tested, guaranteed by design.

Note 4: Minimum values not tested, guaranteed by design.

Note 5: When the load on AD0−15 is increased to 90 pF the value of t_{ALv} is increased by no more than 5 ns. When the load on A16−23 is increased to 90 pF the value of t_{AHv} is increased by no more than 5 ns.

4.4.2.2 Input Signal Requirements: NS32FX164-15, NS32FX164-20 and NS32FX164-25

Symbol	Figure	Description	Reference/ Conditions	NS32FX164-15		NS32FX164-20		NS32FX164-25		Units
				Min	Max	Min	Max	Min	Max	
t_{Xp}	4-15	OSCIN Clock Period	R.E., OSCIN to Next R.E, OSCIN	33	500	25	500	20	500	ns
t_{Xh}	4-15	OSCIN High Time (External Clock)	At 3.5V (Both Edges)	0.5 t_{Xp} − 5 ns		0.5 t_{Xp} − 4 ns		0.5 t_{Xp} − 3 ns		
t_{Xl}	4-15	OSCIN Low Time	At 1.0V (Both Edges)	0.5 t_{Xp} − 5 ns		0.5 t_{Xp} − 4 ns		0.5 t_{Xp} − 3 ns		
t_{DIs}	4-4, 4-11	Data In Setup	Before R.E., CTTL T4	15		14		10		ns
t_{DIh}	4-4, 4-11	Data In Hold (Note 1)	After R.E., CTTL T4	0		0		0		ns
t_{CWs}	4-4, 4-5	$\overline{\text{CWAIT}}$ Signal Setup	Before R.E., CTTL T3 or T3(w)	18		13		10		ns
t_{CWh}	4-4, 4-5	$\overline{\text{CWAIT}}$ Signal Hold	After R.E., CTTL T3 or T3(w)	0		0		0		ns
t_{HLDs}	4-7, 4-8	$\overline{\text{HOLD}}$ Setup Time	Before R.E., CTTL T2 or Ti	16		15		14		ns
t_{HLDh}	4-7, 4-8	$\overline{\text{HOLD}}$ Hold Time	After R.E., CTTL Ti	0		0		0		ns

4.0 Device Specifications (Continued)

4.4.2.2 Input Signal Requirements: NS32FX161-15, NS32FX164-20 and NS32FX164-25 (Continued)

Symbol	Figure	Description	Reference/ Conditions	NS32FX161-15		NS32FX164-20		NS32FX164-25		Units
				Min	Max	Min	Max	Min	Max	
t_{PWR}	4-18	Power Stable to $\overline{\text{RSTI}}$ R.E. (Note 2)	After V_{CC} Reaches 4.5V	50		40		30		μs
t_{RSTw}	4-19	$\overline{\text{RSTI}}$ Pulse Width	At 0.8V (Both Edges)	64		64		64		t_{CTp}
t_{INTh}	4-16	$\overline{\text{INT}}$ Signal Hold	After R.E., CTTL T2 of Interrupt Acknowledge Cycle	0		0		0		ns
t_{NMIs}	4-17	$\overline{\text{NMI}}$ Setup Time	Before F.E., CTTL	15		14		12		ns
t_{NMIh}	4-17	$\overline{\text{NMI}}$ Hold Time	After F.E., CTTL	0		0		0		ns
t_{SPCd}	4-12	SPC Pulse Delay from Slave (Note 2)	After F.E., CTTL T4	2		2		2		t_{CTp}
t_{SPCs}	4-12	$\overline{\text{SPC}}$ Input Setup	Before R.E., CTTL	22		21		20		ns
t_{SPCh}	4-12	SPC Hold Time	After R.E., CTTL	0		0		0		ns
t_{ADSs}	4-9	$\overline{\text{ADS}}$ Input Setup	Before F.E., CTTL	15	$t_{CTp}-3$	14	$t_{CTp}-3$	12	$t_{CTp}-3$	ns
t_{ADSh}	4-9	$\overline{\text{ADS}}$ Input Hold (Note 3)	After F.E., CTTL T1	0		0		0		ns
t_{DDINs}	4-9	$\overline{\text{DDIN}}$ Input Setup	Before F.E., CTTL	15		14		12		ns
t_{DDINih}	4-9	$\overline{\text{DDIN}}$ Input Hold	After R.E., CTTL T4	0		0		0		ns

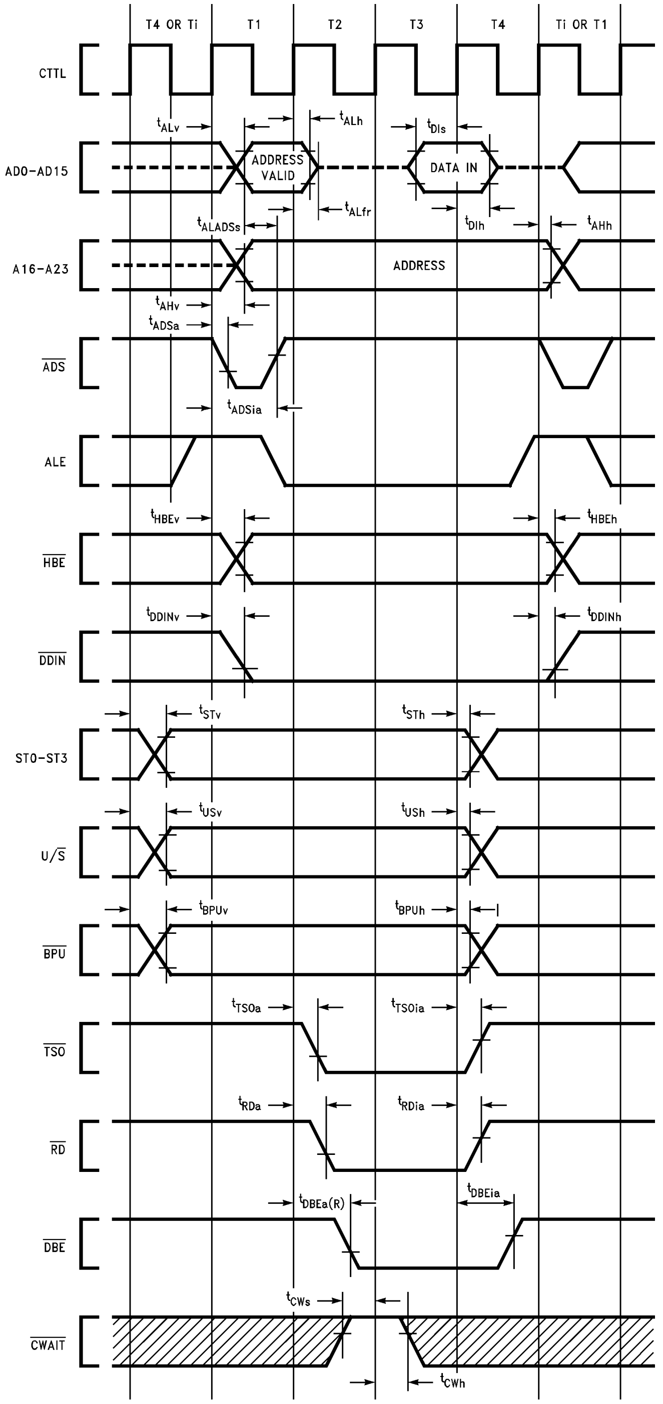
Note 1: t_{Dih} is always less than or equal to t_{RDia}.

Note 2: Not tested, guaranteed by design.

Note 3: $\overline{\text{ADS}}$ must be deasserted before state T4 of the DMA controller cycle.

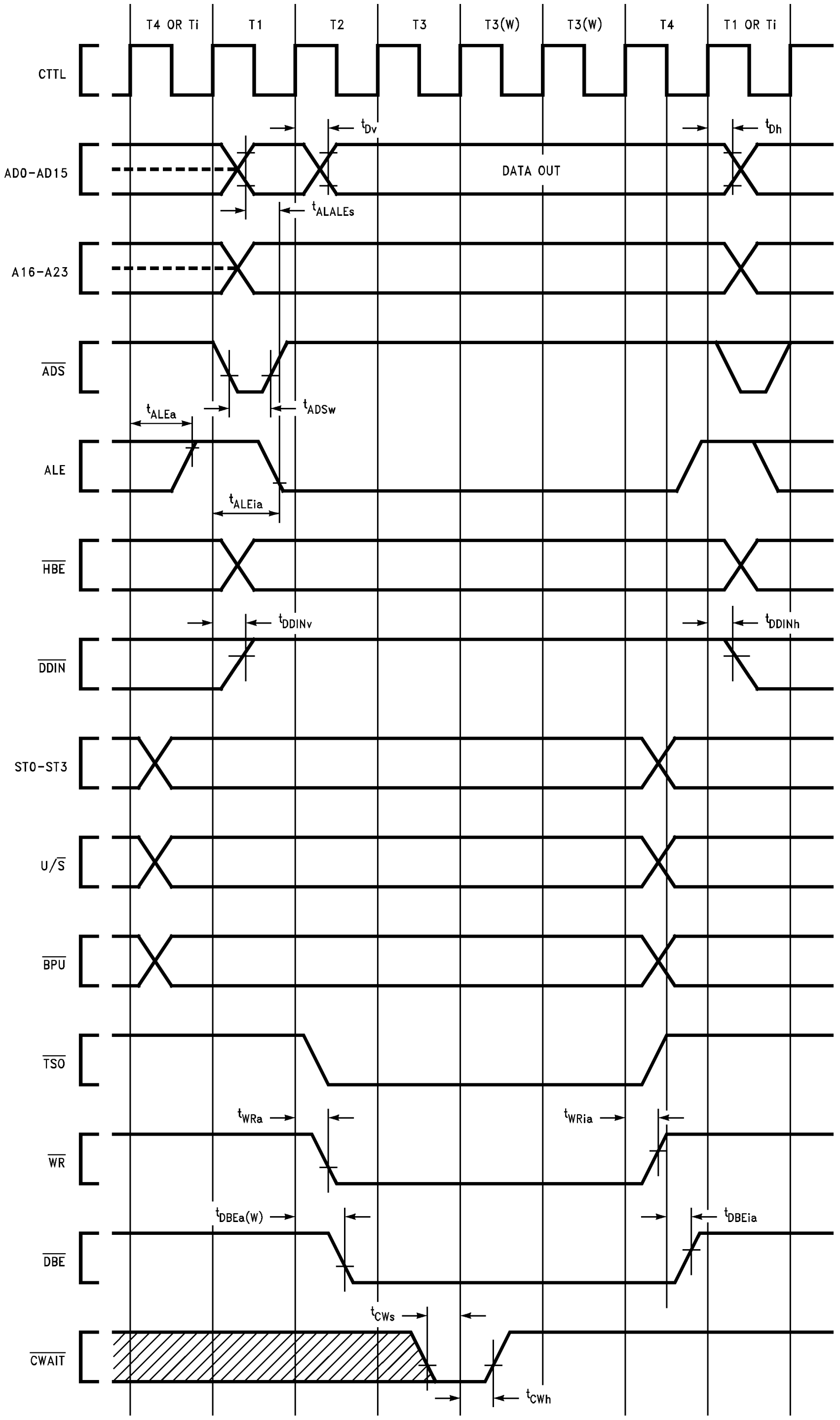
4.0 Device Specifications (Continued)

4.4.3 Timing Diagrams



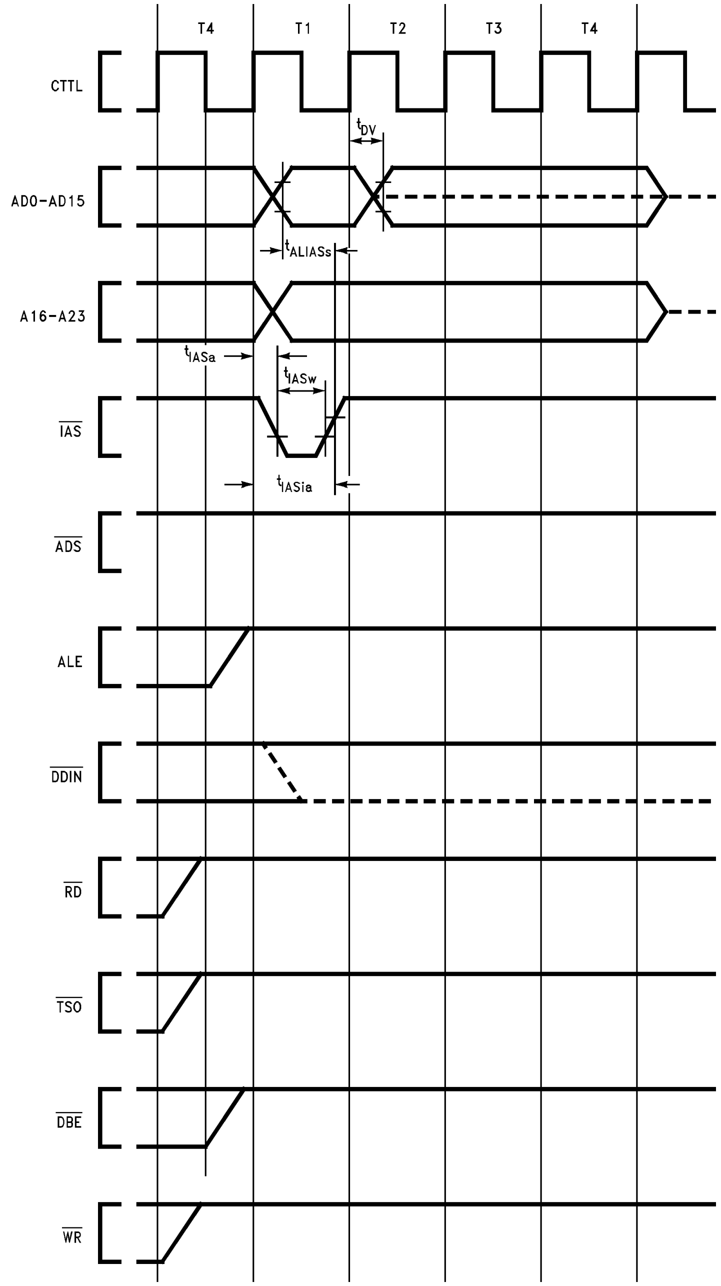
TL/EE/11267–46

FIGURE 4-4. Read Cycle



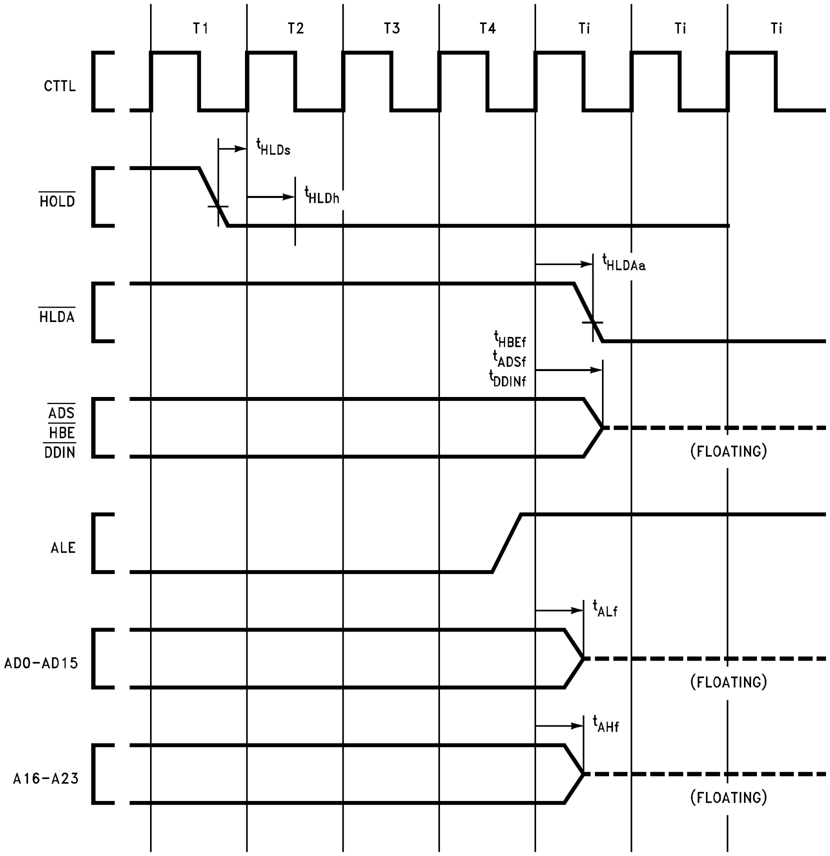
TL/EE/11267–47

FIGURE 4-5. Write Cycle



TL/EE/11267–48

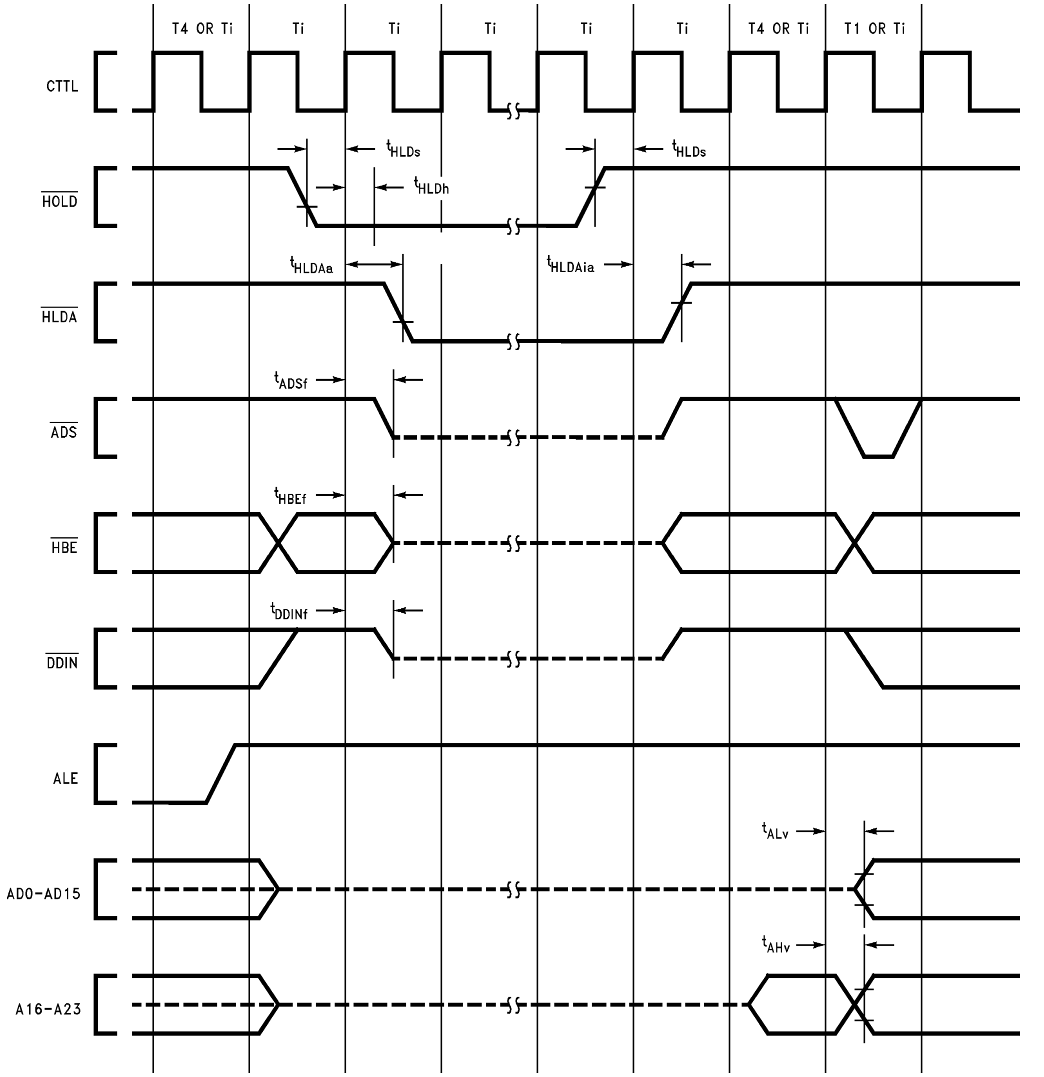
FIGURE 4-6. Special Bus Cycle



TL/EE/11267–49

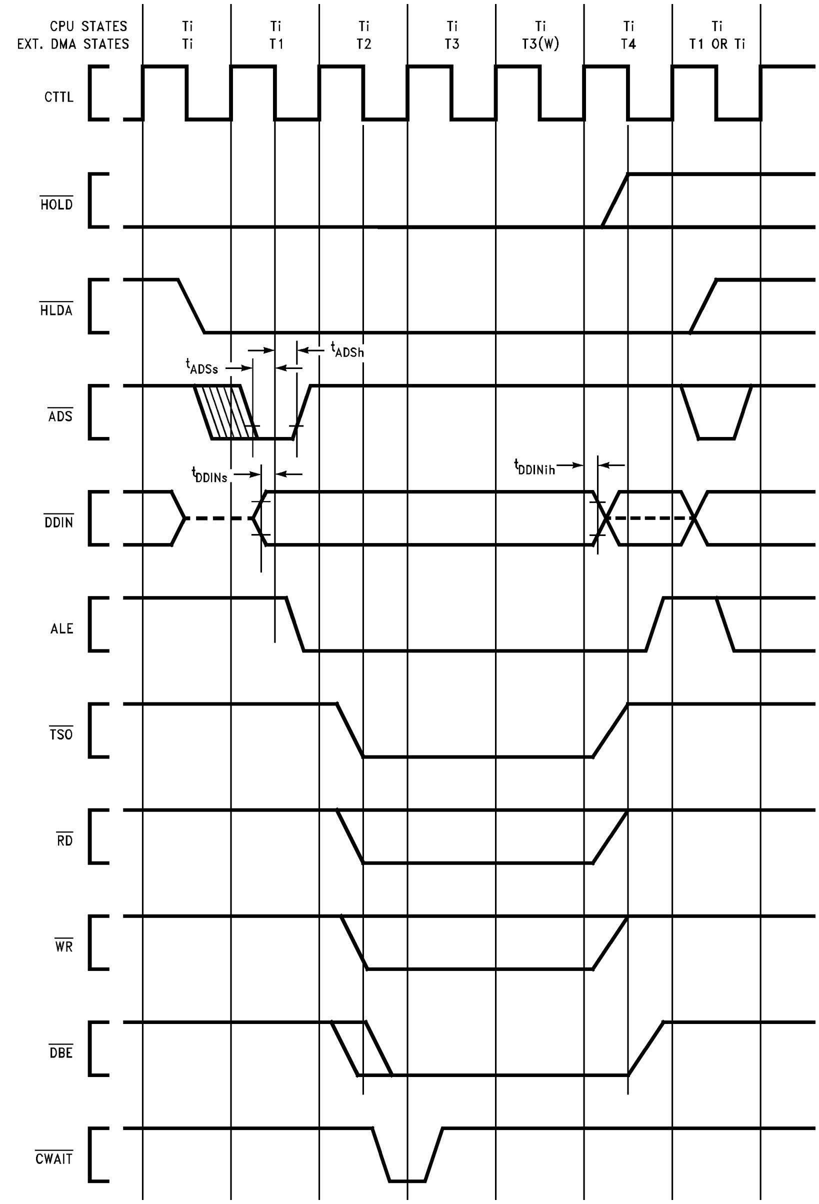
Note: When the bus is not idle, $\overline{\text{HOLD}}$ must be asserted before the rising edge of CTTL of the timing state that precedes state T4 in order for the request to be acknowledged.

FIGURE 4-7. $\overline{\text{HOLD}}$ Acknowledge Timing (Bus Initially Not Idle)



TL/EE/11267–50

FIGURE 4-8. $\overline{\text{HOLD}}$ Timing (Bus Initially Idle)



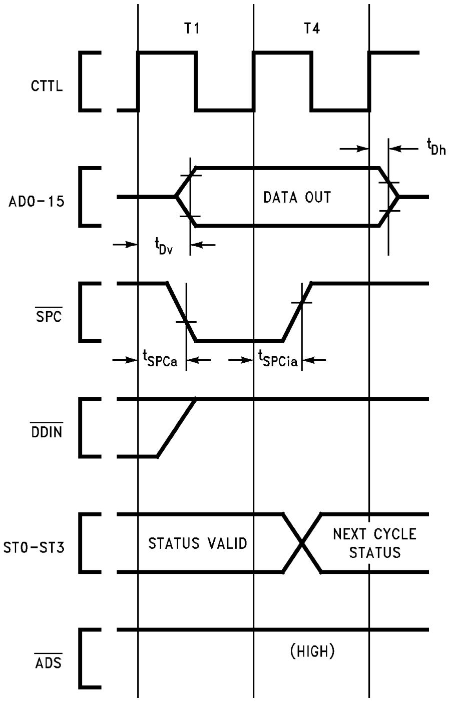
TL/EE/11267–51

Note 1: $\overline{\text{ADS}}$ must be deactivated before state T4 of the external DMA controller cycle.

Note 2: During an external DMA cycle $\overline{\text{WAIT}}$1–2 must be kept inactive unless they are monitored by the DMA Controller. An external DMA cycle is similar to a CPU cycle. The NS32FX164 generates $\overline{\text{TSO}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, ALE and $\overline{\text{DBE}}$. The external DMA controller drives the address/data lines $\overline{\text{HBE}}$, $\overline{\text{ADS}}$ and $\overline{\text{DDIN}}$.

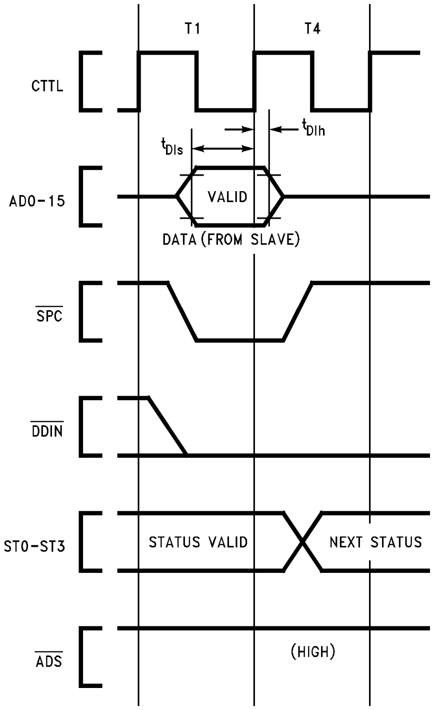
Note 3: During an external DMA cycle, if the $\overline{\text{ADS}}$ signal is pulsed in order to initiate a bus cycle, the $\overline{\text{HOLD}}$ signal must remain asserted until state T4 of the DMA cycle.

FIGURE 4-9. External DMA Controller Bus Cycle



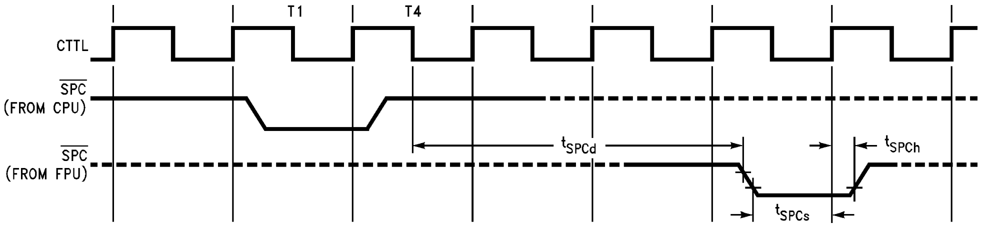
TL/EE/11267–52

FIGURE 4-10. Slave Processor Write Timing



TL/EE/11267–53

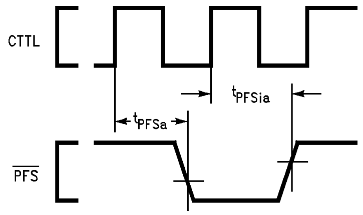
FIGURE 4-11. Slave Processor Read Timing



TL/EE/11267–54

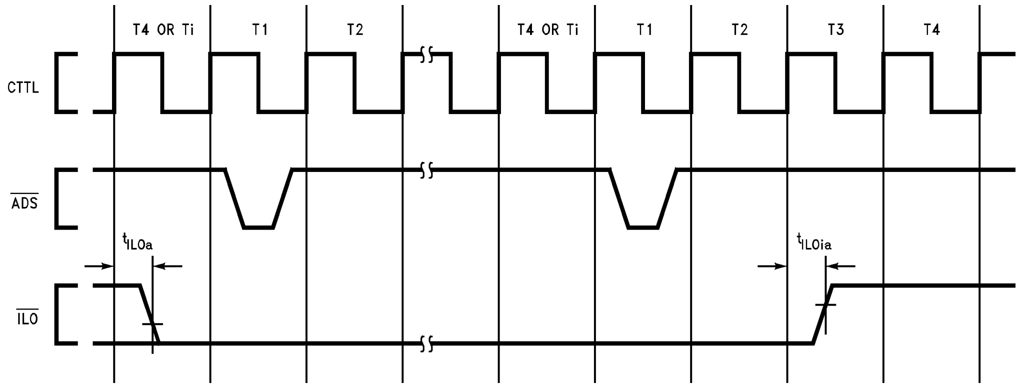
After transferring the last operand to the FPU, the CPU turns OFF the output driver and holds $\overline{SPC}$ high with an internal 5 kΩ pullup.

FIGURE 4-12. $\overline{SPC}$ Timing



TL/EE/11267–55

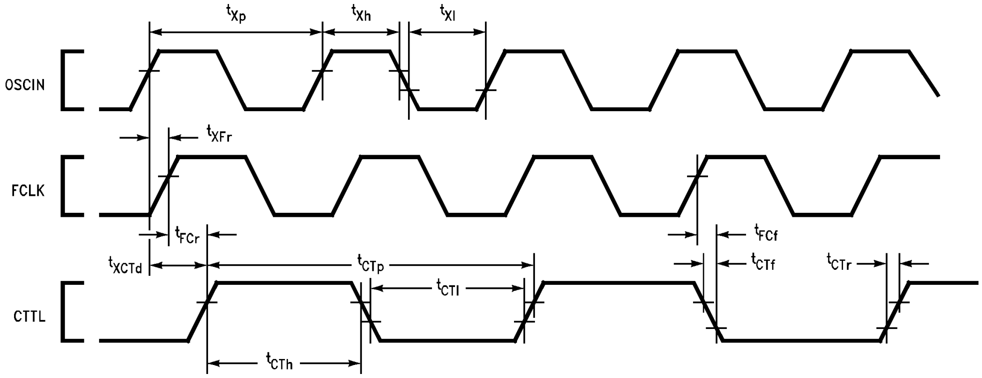
FIGURE 4-13. $\overline{PFS}$ Signal Timing



TL/EE/11267–56

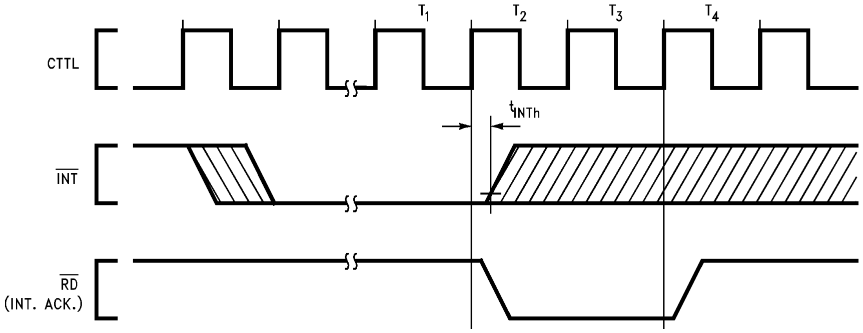
Note: $\overline{ILO}$ may be asserted more than one clock cycle before the beginning of an interlocked access.

FIGURE 4-14. $\overline{ILO}$ Signal Timing



TL/EE/11267–57

FIGURE 4-15. Clock Waveforms

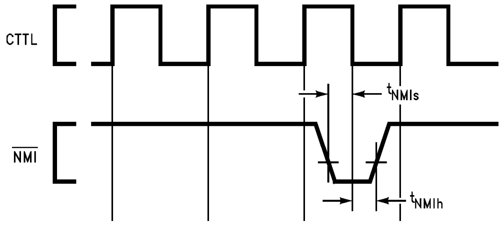


TL/EE/11267–58

FIGURE 4-16. $\overline{\text{INT}}$ Signal Timing

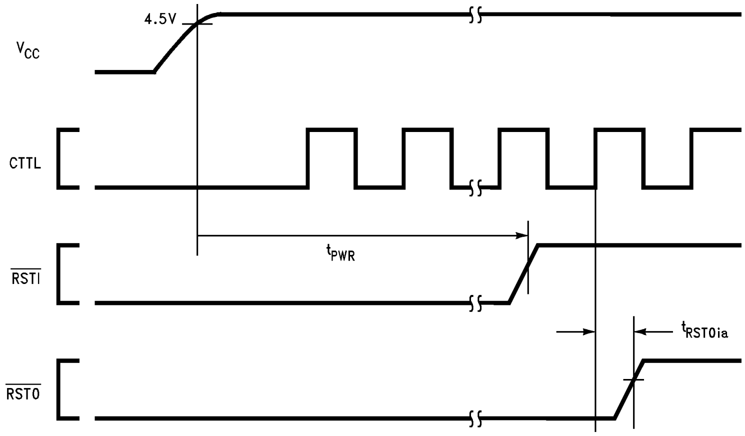
Note 1: Once $\overline{\text{INT}}$ is asserted, it must remain asserted until it is acknowledged.

Note 2: $\overline{\text{INTA}}$ is the Interrupt Acknowledge bus cycle (not a CPU signal). Refer to Section 3.2.1.



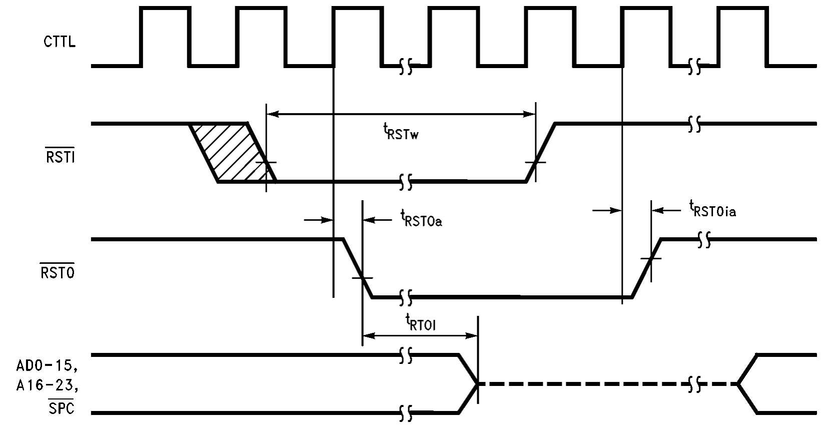
TL/EE/11267–59

FIGURE 4-17. $\overline{\text{NMI}}$ Signal Timing



TL/EE/11267–60

FIGURE 4-18. Power-On Reset

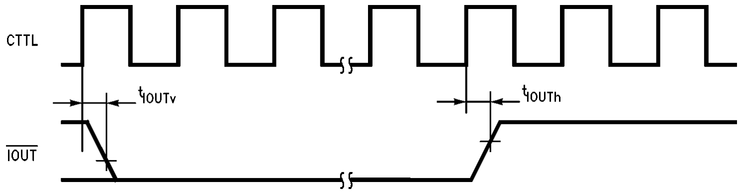


TL/EE/11267-61

Note 1: During Reset the $\overline{\text{HOLD}}$ signal must be kept high.

Note 2: After $\overline{\text{RSTI}}$ is deasserted the first bus cycle will be an instruction fetch at address zero.

FIGURE 4-19. Non-Power-On Reset



TL/EE/11267-72

FIGURE 4-20. Interrupt Out

Appendix A: Instruction Formats

NOTATIONS

i = Integer Type Field
- B = 00 (Byte)
- W = 01 (Word)
- D = 11 (Double Word)

f = Floating-Point Type Field
- F = 1 (Std. Floating: 32 bits)
- L = 0 (Long Floating: 64 bits)

op = Operation Code
- Valid encodings shown with each format.

gen, gen 1, gen 2 = General Addressing Mode Field
- See Section 2.4.2 for encodings.

reg = General Purpose Register Number

cond = Condition Code Field
- 0000 = EQual: Z = 1
- 0001 = Not Equal: Z = 0
- 0010 = Carry Set: C = 1
- 0011 = Carry Clear: C = 0
- 0100 = Higher: L = 1
- 0101 = Lower or Same: L = 0
- 0110 = Greater Than: N = 1
- 0111 = Less or Equal: N = 0
- 1000 = Flag Set: F = 1
- 1001 = Flag Clear: F = 0
- 1010 = LOwer: L = 0 and Z = 0
- 1011 = Higher or Same: L = 1 or Z = 1
- 1100 = Less Than: N = 0 and Z = 0
- 1101 = Greater or Equal: N = 1 or Z = 1
- 1110 = (Unconditionally True)
- 1111 = (Unconditionally False)

short = Short Immediate Value. May contain
- quick: Signed 4-bit value, in MOVQ, ADDQ, CMPQ, ACB
- cond: Condition Code (above), in Scond.
- areg: CPU Dedicated Register, in LPR, SPR
  - 0000 = UPSR
  - 0001–0111 = (Reserved)
  - 1000 = FP
  - 1001 = SP
  - 1010 = SB
  - 1011 = (Reserved)
  - 1100 = (Reserved)
  - 1101 = PSR
  - 1110 = INTBASE
  - 1111 = MOD

Options: in String Instructions

U/W	B	T

T = Translated

B = Backward

U/W = 00: None
- 01: While Match
- 11: Until Match

Configuration bits in SETCFG instruction:

C	M	F	I

7 0

cond	1 0 1 0

Format 0

Bcond (BR)

7 0

op	0 0 1 0

Format 1

BSR	—0000	ENTER	—1000
RET	—0001	EXIT	—1001
CXP	—0010	NOP	—1010
RXP	—0011	WAIT	—1011
RETT	—0100	DIA	—1100
RETI	—0101	FLAG	—1101
SAVE	—0110	SVC	—1110
RESTORE	—0111	BPT	—1111

15 8|7 0

gen	short	op	1 1	i

Format 2

ADDQ	—000	ACB	—100
CMPQ	—001	MOVQ	—101
SPR	—010	LPR	—110
Scond	—011		

15 8|7 0

gen	op	1 1 1 1 1	i

Format 3

CXPD	—0000	ADJSP	—1010
BICPSR	—0010	JSR	—1100
JUMP	—0100	CASE	—1110
BISPSR	—0110		

Trap (UND) on XXX1, 1000

15 8|7 0

gen 1	gen 2	op	i

Format 4

ADD	—0000	SUB	—1000
CMP	—0001	ADDR	—1001
BIC	—0010	AND	—1010
ADDC	—0100	SUBC	—1100
MOV	—0101	TBIT	—1101
OR	—0110	XOR	—1110

23 16|15 8|7 0

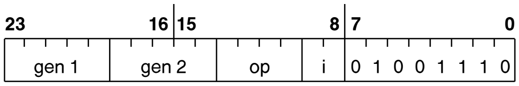
0 0 0 0 0	short	0	op	i	0 0 0 0 1 1 1 0

Appendix A: Instruction Formats (Continued)

Format 5

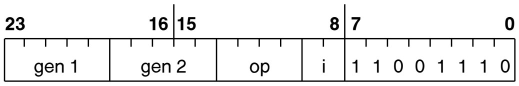
MOVS	−0000	BITWT	−1000
CMPS	−0001	TBITS	−1001
SETCFG	−0010	BBAND	−1010
SKPS	−0011	SBITPS	−1011
BBSTOD	−0100	BBFOR	−1100
EXTBLT	−0101	SBITS	−1101
BBOR	−0110	BBXOR	−1110
MOVMP	−0111		

No Operation on 1111



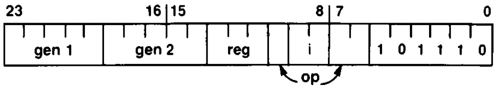
Format 6

ROT	−0000	NEG	−1000
ASH	−0001	NOT	−1001
CBIT	−0010	Trap (UND)	−1010
CBITI	−0011	SUBP	−1011
Trap (UND)	−0100	ABS	−1100
LSH	−0101	COM	−1101
SBIT	−0110	IBIT	−1110
SBITI	−0111	ADDP	−1111



Format 7

MOVM	−0000	MUL	−1000
CMPM	−0001	MEI	−1001
INSS	−0010	Trap (UND)	−1010
EXTS	−0011	DEI	−1011
MOVXBW	−0100	QUO	−1100
MOVZBW	−0101	REM	−1101
MOVZiD	−0110	MOD	−1110
MOVXiD	−0111	DIV	−1111

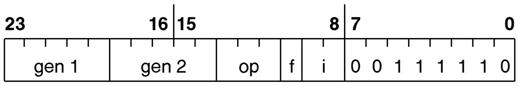


TL/EE/11267–62

Format 8

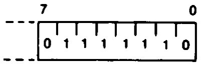
EXT	−0 00	INDEX	−1 00
CVTP	−0 01	FFS	−1 01
INS	−0 10		
CHECK	−0 11		

Trap (UND) on −1 10 and −1 11



Format 9

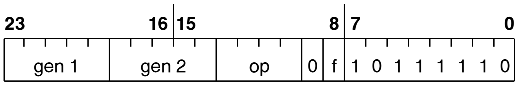
MOVif	−000	ROUND	−100
LFSR	−001	TRUNC	−101
MOVLF	−010	SFSR	−110
MOVFL	−011	FLOOR	−111



TL/EE/11267–63

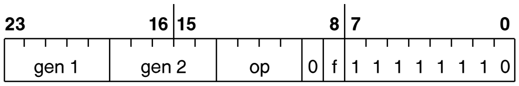
Format 10

Trap (UND) Always



Format 11

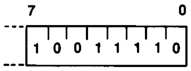
ADDf	−0000	DIVf	−1000
MOVf	−0001	(Note 1)	−1001
CMPf	−0010	Trap (UND)	−1010
(Note 3)	−0011	Trap (UND)	−1011
SUBf	−0100	MULf	−1100
NEGf	−0101	ABSf	−1101
Trap (UND)	−0110	Trap (UND)	−1110
Trap (UND)	−0111	Trap (UND)	−1111



Format 12

(Note 2)	−0000	(Note 2)	−1000
(Note 1)	−0001	(Note 1)	−1001
POLYf	−0010	Trap (UND)	−1010
DOTf	−0011	Trap (UND)	−1011
SCALBf	−0100	(Note 2)	−1100
LOGBf	−0101	(Note 1)	−1101
Trap (UND)	−0110	Trap (UND)	−1110
Trap (UND)	−0111	Trap (UND)	−1111

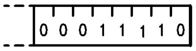
*Instructions with Format 12 are available only when the NS32381 is used.



TL/EE/11267–64

Format 13

Trap (UND) Always

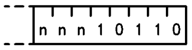


TL/EE/11267–65

Appendix A: Instruction Formats (Continued)

Format 14

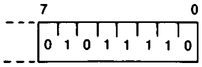
Trap (UND) Always



TL/EE/11267–66

Format 15

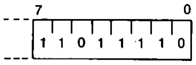
Trap (UND) Always



TL/EE/11267–67

Format 16

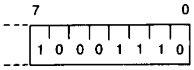
Trap (UND) Always



TL/EE/11267–68

Format 17

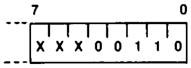
Trap (UND) Always



TL/EE/11267–69

Format 18

Trap (UND) Always

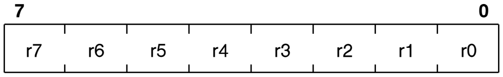


TL/EE/11267–70

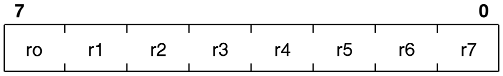
Format 19

Trap (UND) Always

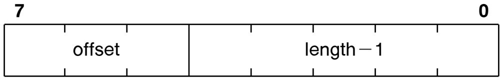
Implied Immediate Encodings:



Register Mask, appended to SAVE, ENTER



Register Mask, appended to RESTORE, EXIT



Offset/Length Modifier appended to INSS, EXTS

Note 1: Opcode not defined; CPU treats like MOVf. First operand has access class of read; second operand has access class of write; f-field selects 32-bit or 64-bit data.

Note 2: Opcode not defined; CPU treats like ADDf. First operand has access class of read; second operand has access class of read-modify-write. f-field selects 32-bit or 64-bit data.

Note 3: Reserved opcode; execution of this opcode will generate an undefined result.

Appendix B: Instruction Execution Times

This section provides the necessary information to calculate the instruction execution times for the NS32FX164.

The following assumptions are made:

- The entire instruction, with all displacements and immediate operands, is assumed to be present in the instruction queue when needed.
- Interference from instruction prefetches, which is very dependent upon the preceding instruction(s), is ignored. This assumption will tend to affect the timing estimate in an optimistic direction.
- It is assumed that all memory operand transfers are completed before the next instruction begins execution. In the case of an operand of access class rmw in memory, this is pessimistic, as the Write transfer occurs in parallel with the execution of the next instruction.
- It is assumed that there is no overlap between the fetch of an operand and the following sequences of microcode. This is pessimistic, as the fetch of Operand 1 will generally occur in parallel with the effective address calculation of Operand 2, and the fetch of Operand 2 will occur in parallel with the execution phase of the instruction.
- Where possible, the values of operands are taken into consideration when they affect instruction timing, and a range of times is given. Where this is not done, the worst case is assumed.

B.1 BASIC AND FLOATING-POINT INSTRUCTIONS

Execution times for basic and floating-point instructions are given in Tables B-1 and B-2. The parameters needed for the various calculations are defined below.

TEA— The time required to calculate an operand's Effective Address. For a Register or Immediate operand, this includes the fetch of that operand.

TEA1— TEA value for the GEN or GEN1 operand.

TEA2— TEA value for the GEN2 operand.

TOPB— The time needed to read or write a memory byte.

TOPW— The time needed to read or write a memory word.

TOPD— The time needed to read or write a memory double-word.

TOPi— The time needed to read or write a memory operand, where the operand size is given by the operation length of the instruction. It is always equivalent to either TOPB, TOPW or TOPD.

TCY— Internal processing overhead, in clock cycles.

L— Internal processing whose duration depends on the operation length. The number of clock cycles is derived by multiplying this value by the number of bytes in the operation length.

NCYC— Number of bus cycles performed by the CPU to fetch or store an operand. NCYC depends on the operand size and alignment.

TPR— CPU processing (in clock cycles) performed in parallel with the FPU.

TFPU— Processing time required by the FPU to execute the instruction. This is the time from the last data sent to the FPU, until done is issued. TFPU can be found in the FPU data sheets.

f— This parameter is related to the floating-point operand size.

Tf— The time required to transfer 32 bits of floating point value to or from the FPU.

Ti— The time required to transfer an integer value to or from the FPU.

B.1.1 Equations

The following equations assume that:

- Memory accesses occur at full speed.
- Any wait states should be reflected in the calculations of TOPB, TOPW and TOPD.

Note: When multiple writes are performed during the execution of an instruction, wait states occurring during intermediate write transactions may be partially hidden by the internal execution. Therefore, a certain number of wait states can be inserted with no effect on the execution time. For example, in the case of the MOVSi instructions each wait state on write operations subtracts 1 clock cycle per write bus access, from the TCY of the instruction, since updating the pointers occurs in parallel with the write operation. This means that wait states can be added to write cycles without changing the execution time of the instruction, up to a maximum of 13 wait states on writes for MOVSB and MOVSW, and 4 wait states on writes for MOVSD.

TEA— TEA values for the various addressing modes are provided in the following table.

TEA TABLE

Addressing Mode	TEA Value	Notes
IMMEDIATE, ABSOLUTE	4	
EXTERNAL	11 + 2 * TOPD	
MEMORY RELATIVE	7 + TOPD	
REGISTER	2	
REGISTER RELATIVE, MEMORY SPACE	5	
TOP OF STACK	4 2 3	Access Class Write Access Class Read Access Class RMW
SCALED INDEXED	TI1 + TI2	

TI1 = TEA of the basemode except:

if basemode is REGISTER then TI1 = 5

if basemode is TOP OF STACK then TI1 = 4

TI2 depends on the scale factor:

if byte indexing TI1 = 5

if word indexing TI2 = 7

if double-word indexing TI2 = 8

if quad-word indexing TI2 = 10

TOPB— If operand is in a register or is immediate then TOPB = 0

else TOPB = 3

TOPW— If operand is in a register or is immediate then TOPW = 0

else TOPW = 4 • NCYC − 1

TOPD— If operand is in a register or is immediate then TOPD = 0

else TOPD = 4 • NCYC − 1

TOPi— If operand is in a register or is immediate then TOPi = 0
else if i = byte then TOPi = TOPB
else if i = word then TOPi = TOPW
else (i = double-word) then TOPi = TOPD

L— If i (operation length) = byte then L = 1
else if i = word then L = 2
else (i = double-word) L = 4

f— If standard floating (32 bits): f = 1
If long floating (64 bits): f = 2

Tf— Tf = 4

Ti— If integer = byte or word, then Ti = 2
If integer = double-word, then Ti = 4

B.1.2 Notes on Table Use

Values in the #TEA1 and #TEA2 columns indicate whether effective addresses need to be calculated.

A value of 1 indicates that address calculation time is required for the corresponding operand. A 0 indicates that the operand is either missing, or it is in a register and the instruction has an optimized form which eliminates the TEA calculation for it.

In the L column, multiply the entry by the operation length in bytes (1, 2 or 4).

In the TCY column, special notations sometimes appear:

n1 → n2 means n1 minimum, n2 maximum

n1%n2 means that the instruction flushes the instruction queue after n1 clock cycles and nonsequentially fetches the next instruction. The value n2 indicates the number of clock cycles for the internal execution of the instruction (including n1).

The effective number of cycles (TCY) must take into account the time (T_{fetch}) required to fetch the portion of the next instruction including the basic encoding and the index bytes. This time depends on the size and the alignment of this portion.

If only one memory cycle is required, then:

$$TCY = n1 + 6 + T_{fetch}$$

If more than one memory cycle is required, then:

$$TCY = n1 + 5 + T_{fetch}$$

In the notes column, notations held within angle brackets < > indicate alternatives in the operand addressing modes which affect the execution time. A table entry which is affected by the operand addressing may have multiple values, corresponding to the alternatives. These addressing notations are:

<I>	Immediate
<R>	CPU Register
<M>	Memory
<F>	FPU Register, either 32 or 64 Bits
<x>	Any Addressing Mode
<ab>	a and b represent the addressing modes of operand 1 and 2 respectively. Both a and b can be any addressing mode (e.g., <MR> means memory to CPU register).

Note: Unless otherwise specified the TCY value for immediate addressing is the same as for CPU register addressing.

B.1.3. Calculation of the Execution Time TEX for Basic Instructions

The execution time for a basic instruction is obtained by performing the following steps:

1. Find the desired instruction in Table B-1.
2. Calculate the values of TEA, TOPB, etc. using the numbers in the table and the equations given in the previous sections.
3. The result derived by adding together these values is the execution time TEX in clock cycles.

EXAMPLE

Calculate TEX for the instruction CMPW R0, TOS.

Operand 1 is in a register; Operand 2 is in memory. This means that we must use the table values corresponding to the <xM> case as given in the Notes column.

Only the #TEA1, #TEA2, #TOPi and TCY columns have values assigned for the CMPi instruction. Therefore, they are they only ones that need to be calculated to find TEX. The blank columns are irrelevant to this instruction.

Both #TEA1 and #TEA2 columns contain 1 for the <xM> case. This means that effective address times have to be calculated for both operands. (For the <MR> case, the Register operand would have required no TEA time, therefore only the Memory operand TEA would have been necessary.) From the equations:

TEA1 (Register mode) = 2.

TEA2 (Top of Stack mode, access class read) = 2.

The #TOPi column represents potential operand transfers to or from memory. For a Compare instruction, each operand is read once, for a total of two operand transfers.

TOPi (Word, Register) = 0,
TOPi (Word, TOS) = 3 (assuming the operand aligned)
Total TOPi = 3

TCY is the time required for internal operation within the CPU. The TCY value for this case is 3.

TEX = TEA1 + TEA2 + TOPi + TCY = 2 + 2 + 3 + 3 = 10 machine cycles.

If the CPU is running at 20 MHz then a machine cycle (clock cycle) is 50 ns. Therefore, this instruction would take 10 × 50 ns, or 0.5 μs, to execute.

B.1.4 Calculation of the Execution Time TEX for Floating-Point Instructions

The execution time for a floating-point instruction is obtained by performing the following steps:

1. Find the desired instruction in Table B-2.
2. Calculate the values of TEA1, TEA2, TOPB, etc., using the numbers in the table, and the equations given in the previous sections.
3. Get the floating-point instruction execution time TFPU from the appropriate FPU data sheet.
4. Choose the higher value between TPR and TFPU + 3.
5. The result derived by adding together these values is the execution time TEX in clock cycles.

EXAMPLE 1

Calculate TEX for the instruction MOVLF F0,@h′3000.

Assumptions:

- The FPU being used is the NS32181.
- Write cycles are performed with no wait states.

Appendix B: Instruction Execution Times (Continued)

TEX Calculation:

Operand 1 is in a register, operand 2 is in memory. This means that we have to use the table values for the <FM> case.

The following parameter values are obtained from Table B-2 and the equations in the previous sections.

TEA2 (Absolute Mode) = 4

TOPD (Memory Write) = 7 (Operand aligned, no waits)

Tf = 4

TCY = 32

TPR = TEA2 + 6 = 4 + 6 = 10

From the FPU Execution Timing table in the NS32181 data sheet we get a TFPU for MOVLF of 19 clock cycles.

The higher value between TPR and TFPU + 3 is 22. The total execution time in clock cycles is:

TEX = TEA2 + TOPD + TF + TCY + 22 = 65

EXAMPLE 2

Calculate TEX for the instruction MULF 20(R0), 4(10(FP))

Assumptions:

- The FPU being used is the NS32181.
- 20(R0) is an aligned read with one wait state.
- 10(FP) is an aligned read with no wait states.
- 4(10 (FP)) is an unaligned rmw with two wait states.

TEX Calculation:

Operand 1 and operand 2 are both in memory. Therefore, the table values for the <MM> case must be used.

The parameter values obtained from Table B-2 and the equations in the previous sections are as follows:

TEA1 (Register Relative Mode) = 5

TEA2 (Memory Relative Mode) = 8 + TOPD = 15 (TOPD = 7 (Operand Aligned, No Wait))

TOPD$_1$ (Read from GEN1) = 7 + 2 = 9 (Operand Aligned, One Wait)

TOPD$_2$ (RMW from GEN2) = 11 + 6 = 17 (Operand Unaligned, Two Waits)

T$_f$ = 4

TCY = 22 → 28

TPR = 0

From the FPU Execution Timing Table in the NS32181 data sheet we get a TFPU for MULF of 33 clock cycles.

The higher value between TPR and TFPU + 3 is 36. The total execution time in clock cycles is:

TEX = TEA1+TEA2+TOPD$_1$+TOPD$_2$+3•T$_f$+TCY+36=5+15+9+17+(22 → 28)+36=133 → 140

TABLE B-1. Basic Instructions

Mnemonic	#TEA1	#TEA2	#TOPB	#TOPW	#TOPD	#TOPi	#L	TCY	Notes
ABSi	1	1	—	—	—	2	—	9	SCR < 0
	1	1	—	—	—	2	—	8	SCR > 0
ACBi	1	—	—	—	—	2	—	16	<M> no branch
	1	—	—	—	—	2	—	15%20	<M> branch
	—	—	—	—	—	—	—	18	<R> no branch
	—	—	—	—	—	—	—	17%22	<R> branch
ADDi	1	1	—	—	—	3	—	3	<xM>
	1	—	—	—	—	1	—	4	<MR>
	—	—	—	—	—	—	—	4	<RR>
ADDCi	1	1	—	—	—	3	—	3	<xM>
	1	—	—	—	—	1	—	4	<MR>
	—	—	—	—	—	—	—	4	<RR>
ADDPi	1	1	—	—	—	3	—	16	No Carry
	1	1	—	—	—	3	—	18	Carry
ADDQi	—	1	—	—	—	2	—	6	<M>
	—	—	—	—	—	—	—	4	<R>
ADDR	1	1	—	—	1	—	—	2	<xM>
	1	—	—	—	—	—	—	3	<xR>
ADJSPi	1	—	—	—	—	1	—	6	
ANDi	1	1	—	—	—	3	—	3	<xM>
	1	—	—	—	—	1	—	4	<MR>
	—	—	—	—	—	—	—	4	<RR>
ASHi	1	1	1	—	—	2	—	14 → 45	
Bcond	—	—	—	—	—	—	—	7	no branch
	—	—	—	—	—	—	—	6%10	branch
BICi	1	1	—	—	—	3	—	3	<xM>
	1	—	—	—	—	1	—	4	<MR>
	—	—	—	—	—	—	—	4	<RR>

Appendix B: Instruction Execution Times (Continued)

TABLE B-1. Basic Instructions (Continued)

Mnemonic	#TEA1	#TEA2	#TOPB	#TOPW	#TOPD	#TOPi	#L	TCY	Notes
BICPSRB	1	—	1	—	—	—	—	18%22	
BICPSRW	1	—	—	1	—	—	—	30%34	
BISPSRB	1	—	1	—	—	—	—	18%22	
BISPSRW	1	—	—	1	—	—	—	30%34	
BPT	—	—	—	2	4	—	—	40	
BR	—	—	—	—	—	—	—	4%10	
BSR	—	—	—	—	1	—	—	6%16	
CASEi	1	—	—	—	—	1	—	4%9	
CBITi	1 1	1 —	2 —	— —	— —	1 1	— —	15 7	<xM> <xR>
CBITIi	1 1	1 —	2 —	— —	— —	1 1	— —	15 7	<xM> <xR>
CHECKi	1 1 1	1 1 1	— — —	— — —	— — —	3 3 3	— — —	7 10 11	high low ok
CMPi	1 1 —	1 — —	— — —	— — —	— — —	2 1 —	— — —	3 3 3	<xM> <MR> <RR>
CMPMi	1	1	—	—	—	2 * n	—	9 * n + 24	n = # of elements in block
CMPQi	1 —	— —	— —	— —	— —	1 —	— —	3 3	<M> <R>
CMPSi	—	—	—	—	—	2 * n	—	35 * n + 53	n = # of elements, not Translated
CMPST	—	—	n	—	—	2 * n	—	38 * n + 53	Translated
COMi	1	1	—	—	—	2	—	7	
CVTP	1	1	—	—	1	—	—	7	
CXP	—	—	—	3	4	—	—	16%21	
CXPD	1	—	—	3	3	—	—	13%18	
DEIi	1 1	1 —	— —	— —	— —	5 1	16 16	38 31	<xM> <xR>
DIA	—	—	—	—	—	—	—	3%7	
DIVi	1	1	—	—	—	3	16	58 → 68	
ENTER	—	—	—	—	n + 1	—	—	4 * n + 18	n = # of general registers saved
EXIT	—	—	—	—	n + 1	—	—	5 * n + 17	n = # of general registers restored
EXTi	1 1	1 1	— —	— —	1 —	1 1	— —	19 → 29 17 → 51	field in memory field in register
EXTSi	1	1	—	—	1	1	—	26 → 36	
FFSi	1	1	2	—	—	1	24	24 → 28	
FLAG	— —	— —	— —	— 4	— 3	— —	— —	6 44	no trap trap
IBITi	1 1	1 —	2 —	— —	— —	1 —	— —	17 9	<xM> <xR>

Appendix B: Instruction Execution Times (Continued)

TABLE B-1. Basic Instructions (Continued)

Mnemonic	#TEA1	#TEA2	#TOPB	#TOPW	#TOPD	#TOPi	#L	TCY	Notes
INDEXi	1	1	—	—	—	2	16	25	
INSi	1 1	1 —	— —	— —	2 —	1 1	— —	29 → 39 28 → 96	field in memory field in register
INSSi	1	1	—	—	2	1	—	39 → 49	
JSR	1	—	—	—	1	1	—	5%15	
JUMP	1	—	—	—	—	—	—	2%6	
LPRi	1	—	—	—	—	1	—	19 → 33	
LSHi	1	1	1	—	—	2	—	14 → 45	
MEIi	1	1	—	—	—	4	16	23	
MODi	1	1	—	—	—	3	16	54 → 73	
MOVi	1 1 —	1 — —	— — —	— — —	— — —	2 1 —	— — —	1 3 3	<xM> <MR> <RR>
MOVMi	1	1	—	—	—	2 * n	—	3 * n + 20	n = # of elements in block
MOVQi	1 —	— —	— —	— —	— —	1 —	— —	2 3	<M> <R>
MOVSB, W	— —	— —	— —	— —	— —	2 * n 2 * n	— —	14 * n + 59 24 * n + 54	n = # elements no options B, W and/or U option in effect
MOVSD	— —	— —	— —	— —	— —	2 * n 2 * n	— —	10 * n + 59 24 * n + 54	n = # of elements no options B, W and/or U option in effect
MOVST	—	—	n	—	—	2 * n	—	27 * n + 54	Translated
MOVXBD	1	1	1	—	1	—	—	6	
MOVXBW	1	1	1	1	—	—	—	6	
MOVXWD	1	1	—	1	1	—	—	6	
MOVZBD	1	1	1	—	1	—	—	5	
MOVZBW	1	1	1	1	—	—	—	5	
MOVZWD	1	1	—	1	1	—	—	5	
MULi	1	1	—	—	—	3	16	15	
NEGi	1	1	—	—	—	2	—	5	
NOP	—	—	—	—	—	—	—	3	
NOTi	1	1	—	—	—	2	—	5	
ORi	1 1 —	1 — —	— — —	— — —	— — —	3 1 —	— — —	3 4 4	<xM> <MR> <RR>
QUOi	1	1	—	—	—	3	16	49 → 55	

TABLE B-1. Basic Instructions (Continued)

Mnemonic	#TEA1	#TEA2	#TOPB	#TOPW	#TOPD	#TOPi	#L	TCY	Notes
REMi	1	1	—	—	—	3	16	57 → 62	
RESTORE	—	—	—	—	n	—	—	5 * n + 12	n = # of general registers restored
RET	—	—	—	—	1	—	—	2%8	
RETI	— —	— —	1 2	2 2	2 3	— —	— —	60 60	Non-Cascaded Cascaded
RETT	—	—	—	2	2	—	—	45	
ROTi	1	1	1	—	—	2	—	14 → 45	
RXP	—	—	—	1	2	—	—	2%6	
Scondi	1 1	— —	— —	— —	— —	1 1	— —	9 10	False True
SAVE	—	—	—	—	n	—	—	4 * n + 13	n = # of general registers saved
SBITi	1 1	1 —	2 —	— —	— —	1 1	— —	15 7	<xM> <xR>
SBITIi	1 1	1 —	2 —	— —	— —	1 1	— —	15 7	<xM> <xR>
SETCFG	—	—	—	—	—	—	—	15	
SKPSi	—	—	—	—	—	n	—	27 * n + 51	n = # of elements, not Translated
SKPST	—	—	n	—	—	n	—	30 * n + 51	Translated
SPRi	1	—	—	—	—	1	—	21 → 27	
SUBi	1 1 —	1 — —	— — —	— — —	— — —	3 1 —	— — —	3 4 4	<xM> <MR> <RR>
SUBCi	1 1 —	1 — —	— — —	— — —	— — —	3 1 —	— — —	3 4 4	<xM> <MR> <RR>
SUBPi	1 1	1 1	— —	— —	— —	3 3	— —	16 18	no carry carry
SVC	—	—	—	2	4	—	—	40	
TBIti	1 1	1 —	1 —	— —	— —	1 1	— —	14 4	<xM> <xR>
WAIT	—	—	—	—	—	—	—	6 → ?	? = until an interrupt/reset
XORi	1 1 —	1 — —	— — —	— — —	— — —	3 1 —	— — —	3 4 4	<xM> <MR> <RR>

Appendix B: Instruction Execution Times (Continued)

TABLE B-2. Floating-Point Instructions: CPU Portion

Mnemonic	#TEA1	#TEA2	#TOPD	#TOPi	#Ti	#Tf	TCY	TPR	Notes
ADDf,	—	—	—	—	—	—	17	8	<FF>
SUBf,	1	—	f	—	—	f	(14 → 17) +3f	0	<MF>
MULf,	—	—	—	—	—	f	24 + f	0	<IF>
DIVf	—	1	2f	—	—	2f	(25 → 29) +6f	0	<FM>
	—	1	2f	—	—	3f	(27 → 30) +3f	0	<IM>
	1	1	3f	—	—	3f	(13 → 19) +9f	0	<MM>
MOVf,	—	—	—	—	—	—	17	6	<FF>
ABSf,	1	—	f	—	—	f	(14 → 17) + 3f	0	<MF>
NEGf	—	—	—	—	—	f	24 + f	0	<IF>
	—	—	f	—	—	f	23 + 3f	6 + TEA2	<FM>
	—	—	f	—	—	2f	33 + f	TEA2 − 2 − f	<IM>
	1	—	2f	—	—	2f	(20 → 23) +6f	TEA2−3	<MM>
MOVFL	—	—	—	—	—	—	17	8	<FF>
	1	—	1	—	—	1	17 → 20	0	<MF>
	—	—	—	—	—	1	25	0	<IF>
	—	—	2	—	—	2	35	6 + TEA2	<FM>
	—	—	2	—	—	3	43	TEA2 − 3	<IM>
	1	—	3	—	—	3	35 → 38	TEA2 − 3	<MM>
MOVLF	—	—	—	—	—	—	16	8	<FF>
	1	—	2	—	—	2	20 → 23	0	<MF>
	—	—	—	—	—	2	26	0	<IF>
	—	—	1	—	—	1	32	TEA2 + 6	<FM>
	—	—	1	—	—	3	42	TEA2 − 4	<IM>
	1	—	3	—	—	3	35 → 38	TEA2 − 3	<MM>
TRUNCfi,	—	—	—	—	1	—	20	9	<FR>
FLOORfi,	1	—	f	—	1	f	(17 → 20) + 3f	0	<MR>
ROUNDfi	—	—	—	—	1	f	25 + f	0	<IR>
	—	—	—	1	1	—	20	TEA2 + 6	<FM>
	—	—	—	1	1	f	26 + f	TEA2 − 2	<IM>
	1	—	f	1	1	f	(16 → 19) +4f	TEA2 − 2 − f	<MM>
MOVif	—	—	—	—	1	—	25 − f	0	<RF>
	1	—	—	1	1	—	18	0	<MF>
	—	—	—	—	1	—	26	0	<IF>
	—	1	f	—	1	f	20 + 4f	0	<RM>
	—	1	f	—	1	f	22 + 5f	0	<IM>
	1	1	f	1	1	f	(10 → 13) + 5f	0	<MM>
CMPf	—	—	—	—	—	—	23	13	<FF>
	1	—	f	—	—	f	(20 → 23) + 3f	7	<MF>
	—	—	—	—	—	f	31 + f	7	<IF>
	—	1	f	—	—	f	(27 → 30) + 3f	0	<FM>
	—	1	f	—	—	2f	29	0	<IM>
	1	1	2f	—	—	2f	(15 → 21) + 6f	0	<MM>
	—	—	—	—	—	f	37 + f	0	<FI>
	1	—	f	—	—	2f	(21 → 29) + 8f	0	<MI>
	—	—	—	—	—	2f	35 + 2f	0	<II>
SFSR	—	—	—	—	—	1	19	7	<R>
	1	—	1	—	—	1	20	TEA1 + 4	<M>
LFSR	—	—	—	—	—	1	23	0	<R>
	1	—	1	—	—	1	18 → 21	0	<M>

Appendix B: Instruction Execution Times (Continued)

B.2 SPECIAL GRAPHICS INSTRUCTIONS

This section provides the execution times for the special graphics instructions. Table B-3 lists the average instruction execution times for different shift values and for a no-wait-state system design. The ‘‘No Option’’ of each instruction is used. The effect of wait states on the execution time is rather difficult to evaluate due to the pipelined nature of the read and write operations.

Instructions that have *shift* amounts, such as BBOR, BBXOR, BBAND, BBFOR and BITWT, make use of the parallel nature of the Series 32000®/EP processors by doing the actual *shift* during the reading of the double-word destination data. This means that if there are wait states on read operations, these instructions are able to *shift* further, without impacting the overall execution time. For example, the total execution time for a BBFOR operation, *shifting* 8 bits, with 2 wait states on read operations, is the same as for a BBFOR operation *shifting* by 12 bits. This is because a destination read takes 4 clock cycles longer than a no-wait-state double-word read does. Note that this effect is not valid for more than 4 wait states because at 4 wait states, all possible *shift* values (0–15) are ‘‘hidden’’ during the destination read.

Table B-4 shows the average execution times with wait states, assuming a shift value of eight unless stated otherwise. The parameters used in the execution time equations are defined below.

Twaitrd	The number of wait states applied for a Read operation.
Twaitr	The number of wait states applied for a Write operation.
Twaitrds	The number of wait states applied for a Read operation on source data. This also refers to the number of wait states applied for a table memory access (in the SBITS instruction, for example).
Twaitrdd	The number of wait states applied for a Read operation on destination data.
Twaitwrd	The number of wait states applied for a Write operation on destination data.
Twaitbt	Twaitrds + Twaitrdd * 2 + Twaitwrd * 2, the value used for BITBLT timing.
width	The width of a BITBLT operation, in words.
height	The height of a BITBLT operation, in scan lines.
shift	The number of bits of shift applied.

B.2.1 Execution Time Calculation for Special Graphics Instructions

The execution time for a special graphics instruction is obtained by inserting the appropriate parameters to the equation for that instruction and evaluating it.

For example, to calculate the execution time of the BBOR instruction applied to a 10-word wide and 5-line high data block, assuming a shift count of 15 and a no-wait-state system, the following equation from Table B-3 is used.

$$42 + (107 + 44 * (\text{width} - 2)) * \text{height} + ((\text{shift} - 8) * \text{width} * \text{height})$$

Substituting the appropriate values to the shift, width and height parameters yields:

$$45 + (107 + 44 * (10 - 2)) * 50 + ((15 - 8) * 10 * 50)$$

or

$$42 + (107 + 352) * 50 + (7 * 500) = 26{,}492 \text{ clocks or } 1.77 \text{ ms @ } 15 \text{ MHz}$$

This represents the ‘‘worst case’’ time for this instruction, since a *shift* of greater than 15 bits can be handled by moving the source and destination pointers by 2 bytes and adjusting the *shift* amount.

The ‘‘best case’’ and ‘‘average case’’ times for most instructions are the same, due to reading the destination data during the *shifting* of the source data.

TABLE B-3. Average Instruction Execution Times with No Wait-States

Instruction	Number of Clock Cycles	Notes
BBOR	42 + (107 + 44 * (*width* − 2)) * *height* 42 + (107 + 44 * (*width* − 2)) * *height* + ((shift − 8) * *width* * *height*)	Shift = 0 → 8 Shift > 8
BBXOR	44 + (107 + 44 * (*width* − 2)) * *height* 44 + (107 + 44 * (*width* − 2)) * *height* + ((shift − 8) * *width* * *height*)	Shift = 0 → 8 Shift > 8
BBAND	45 + (111 + 44 * (*width* − 2)) * *height* 45 + (111 + 44 * (*width* − 2)) * *height* + ((shift − 8) * *width* * *height*)	Shift = 0 → 8 Shift > 8

BBFOR	48 + (61 + 25 * (*width* − 2)) * *height*
	48 + (74 + 32 * (*width* − 2)) * *height*
	48 + (74 + 32 * (*width* − 2))* *height* + ((shift − 8) * *width* * *height*)
		Shift = 0
		Shift = 1 → 8
		Shift > 8

BBSTOD	66 + (170 + 60 * (*width* − 2)) * *height* 66 + (170 + 60 * (*width* − 2)) * *height* + ((shift − 8) * *width* * *height*)	Shift = 0 → 8 Shift > 8

Appendix B: Instruction Execution Times (Continued)

TABLE B-3. Average Instruction Execution Times with No Wait-States (Continued)

Instruction	Number of Clock Cycles	Notes

BITWT	16
	28
	28 + (_shift_ − 8)
		Shift = 0
		Shift = 1 → 8
		Shift > 8

EXTBLT	35 + (19 + 12 * _width_) * _height_
	35 + (13 + 12 * _width_) * _height_
	35 + (17 + 13 * _width_) * _height_
	35 + (11 + 13 * _width_) * _height_
		Shift = 0 → 8, Pre-Read
		Shift = 0 → 8, No Pre-Read
		Shift > 8, Pre-Read
		Shift > 8, No Pre-Read

MOVMPB,W	16 + 7 * R2	
MOVMPD,W	16 + 8 * R2	
SBITS	39 42	R2 ≤ 25 R2 > 25
SBITP	8 + (34 * R2)	

TABLE B-4. Average Instruction Execution Times with Wait-States

Instruction	Number of Clock Cycles	Notes
BBOR	42 + ((107 + 2 * Twaitblt) + (44 + Twaitblt) * (_width_ − 2)) * _height_	
BBXOR	44 + ((107 + 2 * Twaitblt) + (44 + Twaitblt) * (_width_ − 2)) * _height_	
BBAND	45 + ((111 + 2 * Twaitblt) + (44 + Twaitblt) * (_width_ − 2)) * _height_	
BBFOR	48 + ((74 + 2 * Twaitblt) + (32 + Twaitblt) * (_width_ − 2)) * _height_	
BBSTOD	66 + ((170 + 2 * Twaitblt) + (60 + Twaitblt) * (_width_ − 2)) * _height_	

BITWIT	16 + Twaitrds + Twaitrdd + Twaitwrd
	28 + Twaitblt
		Shift = 0
		Shift = 1 → 8

EXTBLT	35 + (19 + (12 + (Twaitrds + Twaitrdd + Twaitwrd))* _width_) * _height_ 35 + (13 + (12 + (Twaitrds + Twaitrdd + Twaitwrd)) * _width_) * _height_	Pre-Read No Pre-Read
MOVMPB,W	16 + 7 * R2 + (Twaitwr − 1) * R2 16 + 7 * R2	Twaitwr > 1 Twaitwr ≤ 1
MOVMPD	16 + 8 * R2 + Twaitwr * R2	
SBITS	39 + (2 * Twaitrdd + 2 * Twaitwrd + 2 * Twaitrds) 42 + (2 * Twaitrdd + 2 * Twaitrds)	R2 ≤ 25 R2 > 25
SBITP	8 + (34 * R2) + ((Twaitrdd + Twaitwrd) * R2)	

B.3 DSPM INSTRUCTIONS

The performance of the command list operations is given in the following tables:

Load Register Instructions

Instruction	Cycles
LX	3
LY	3
LZ	3
LA	3
LEA	5
LPARAM	3
LREPEAT	3
LEABR	3

Store Register Instructions

Instruction	Cycles
SX	3
SXL	3
SXH	4
SY	3
SZ	3
SA	3
SEA	3
SREPEAT	3
SOVF	3

Appendix B: Instruction Execution Times (Continued)

Adjust Register Instructions

Instruction	Cycles
INCX	4
INCY	4
INCZ	4
DECX	4
DECY	4
DECZ	4

Flow Control Instructions

Instruction	Cycles
NOPR	2
HALT	1
DJNZ	5
DBPT	3

Internal Memory Move Instructions

Instruction	Cycles
VRMOV	2 × *leng* + 2
VARMOV	2 × *leng* + 2
VRGATH	4 × *leng* + 4
VRSCAT	4 × *leng* + 4

External Memory Move Instructions

Assuming EXT.HOLD = 0:

Instruction	Cycles
VXLOAD	(5 + w) * *leng* + k + 2
VXSTORE	(5 + w) * *leng* + k + 2
VXGATH	(5 + w) * *leng* + k + 2

w = Number of wait states in external memory access.

k = Number of cycles until HLDA is received, in external memory instructions.

Arithmetic/Logical Instructions

Instruction	Cycles
VROP	3 × *leng* + 3
VAROP	3 × *leng* + 4

Multiply-and-Accumulate Instructions

Instruction	Cycles
VRMAC	2 × *leng* + 7
VARMAC	2 × *leng* + 7
VCMAC	4 × *leng* + 7
VRLATP	4 × *leng* + 5
VCLATP	4 × *leng* + 2

Multiply-and-Add Instructions

Instruction	Cycles
VAIMAD	6 * *leng* + 2
VAIMADS	6 * *leng* + 4
VRMAD	4 * *leng* + 3
VARMAD	4 * *leng* + 4
VEMAD	6 * *leng* + 2
VCMAD	4 * *leng* + 6

Clipping and Min/Max Instructions

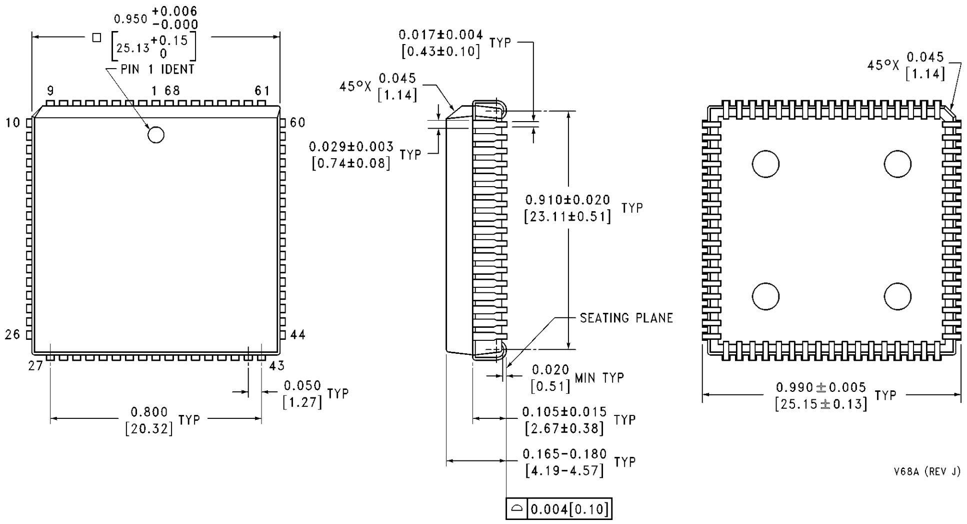
Instruction	Cycles
VARABS	2 × *leng* + 5
VARMIN	7 × *leng* + 2
VARMAX	7 × *leng* + 2
VRFMIN	4 × *leng* + 6
VRFMAX	4 × *leng* + 6
EFMAX	17

Special Instructions

Instruction	Cycles
ESHL	1 × *leng* + 4
VCPOLY	4 × *leng* + 16
VDECIDE	12 × *leng* + 4
VDIST	5 × *leng* + 5
VFFT	8 × *leng* + 6
VESIIR	16 × *leng* + 6

If *leng* = 1 in ESHL instruction, then the timing is 4 cycles.

Physical Dimensions inches (millimeters)



68-Pin Plastic Leaded Chip Carrier (V)
Order Number NS32FX164V-15, NS32FX164V-20 or NS32FX164V-25
NS Package Number V68A

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
1111 West Bardin Road
Arlington, TX 76017
Tel: 1(800) 272-9959
Fax: 1(800) 737-7018

National Semiconductor Europe
Fax: (+49) 0-180-530 85 86
Email: cnjwge@tevm2.nsc.com
Deutsch Tel: (+49) 0-180-530 85 85
English Tel: (+49) 0-180-532 78 32
Français Tel: (+49) 0-180-532 93 58
Italiano Tel: (+49) 0-180-534 16 80

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductor Japan Ltd.
Tel: 81-043-299-2309
Fax: 81-043-299-2408

National does not assume any responsibility for use of any circuitry described, no circuit patent licenses are implied and National reserves the right at any time without notice to change said circuitry and specifications.