National Semiconductor

PRELIMINARY
December 1992

NS32AM162-20/NS32AM163-20
Voice Processor with Serial CODEC Interface

General Description

The NS32AM162 and the NS32AM163 are integrated 32 bit members of the Series 32000®/EP family of National's Embedded System Processors™, tuned for the Digital (tapeless) Answering Machine (DAM) market. These processors integrate the functions of a traditional Digital Signal Processing (DSP) chip and of a system controller. The devices contain system support functions such as DRAM Controller, Interrupt Control Unit, Pulse Width Modulator, CODEC Interface, WATCHDOG™ timer, and a Clock Generator. The NS32AM162 and the NS32AM163 can execute instructions from either an on-chip ROM or from an external ROM.

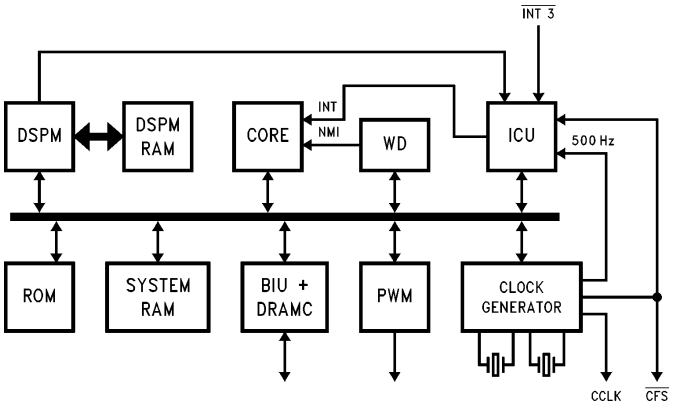
The NS32AM162 and NS32AM163 have all the features of National's NS32AM160 and NS32AM161 respectively. The main difference between the formers and the latters is in the CODEC interface. The NS32AM160 and NS32AM161 support a parallel CODEC. The NS32AM162 and NS32AM163 support one or two serial CODECs.

Throughout this data sheet, unless otherwise mentioned, every reference to the NS32AM162 is applicable to the NS32AM163 as well.

Features

- NS32AM162 is software and pin compatible with NS32AM160 (NS32AM163—with NS32AM161)
- Software compatible with the Series 32000/EP processors
- Designed around the CPU core of the NS32CG16
- 32-bit architecture and implementation
- 20.48 MHz operation
- 2.1 Kbyte on-chip RAM
- On-chip DSP Module (DSPM) for high speed DSP operations
- Three modes of operation configured via strap pins:
  - — Internal ROM mode
    - 25 Kbyte Internal ROM (32 Kbyte in the NS32AM163)
    - 8-bit external data bus
    - 16-bit programmable I/O lines
    - 8 output lines
  - — External ROM mode
    - 16-bit external data bus
    - 8-bit programmable I/O lines
    - 16-bit address bus with support for external 128 Kbyte ROM
    - Support for external I/O devices
  - — Development mode
    - 16-bit external data bus
    - 18-bit address bus with support for external 512 Kbyte ROM
    - Support for external I/O devices
    - Support for device test via status pins
- On-chip Interrupt Control Unit (ICU) provides 4 levels of interrupts
- On-chip DRAM Controller for 4 Mbit and 16 Mbit devices
- On-chip CODEC clock generation and interface
- On-chip 2 ms Real Time Counter
- On-chip 8-bit Pulse Width Modulator (PWM) module
- On-chip WATCHDOG Timer
- Power down mode

Block Diagram



TL/EE/11732–1

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

 TL/EE11732

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

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

List of Figures

List of Figures (Continued)

List of Tables

1.0 Product Introduction

The NS32AM162 processor performs the main tasks of a Digital Answering Machine: system control, voice compression/decompression, and other voice services.

System control includes user interface via keyboard and display handling. This task also controls the phone line, and monitors the activity on the line. The system control also keeps track of the time and detects power failures.

The voice compression/decompression consists of performing transformations between voice samples and compressed digital data. The on-chip DSPM allows the implementation of different voice handling algorithms, such as GSM, Sub-Band Coding (SBC), and Linear Predictive Code (LPC).

Other voice services include DTMF detection and generation, tone generation, voice synthesis, voice recognition, VOX detection, etc.

Three different system configurations are supported:

Internal ROM Mode. This mode provides the lowest chip count for a full DAM solution. In this mode the NS32AM162 provides 25 Kbytes of on-chip program ROM (32 Kbytes in the NS32AM163), and three on-chip general purpose I/O ports. *Figure 1-1* shows a DAM based on the NS32AM162 in its Internal ROM mode.

External ROM Mode. This mode allows program flexibility in the DAM application. In this mode, an external ROM can be attached to the NS32AM162 to provide an easy way of changing the DAM's program. One on-chip general purpose I/O port is provided, and two other I/O ports can be added with minimal logic. *Figure 1-2* shows a DAM based on the NS32AM162 in its External ROM mode.

Development Mode. Development mode is useful for evaluation and testing. In this mode, external ROM, RAM, and I/O devices can be connected to the NS32AM162. Some pins are used to reflect the internal status of the NS32AM162. No on-chip I/O ports are provided in this mode. *Figure 1-3* shows an Evaluation Board based on the NS32AM162 in its Development mode.

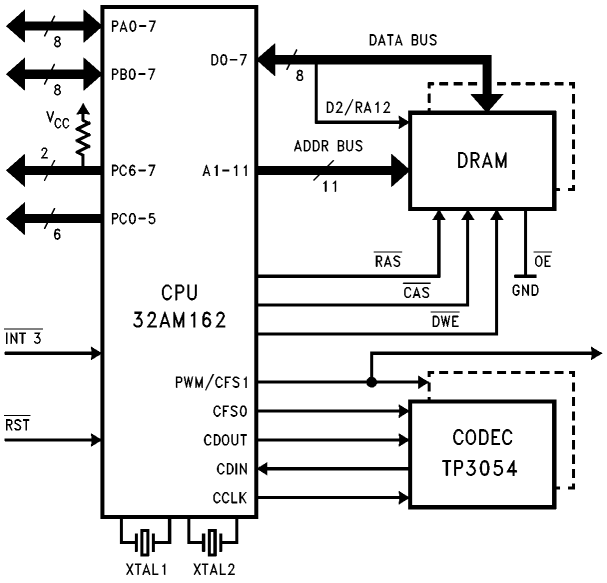
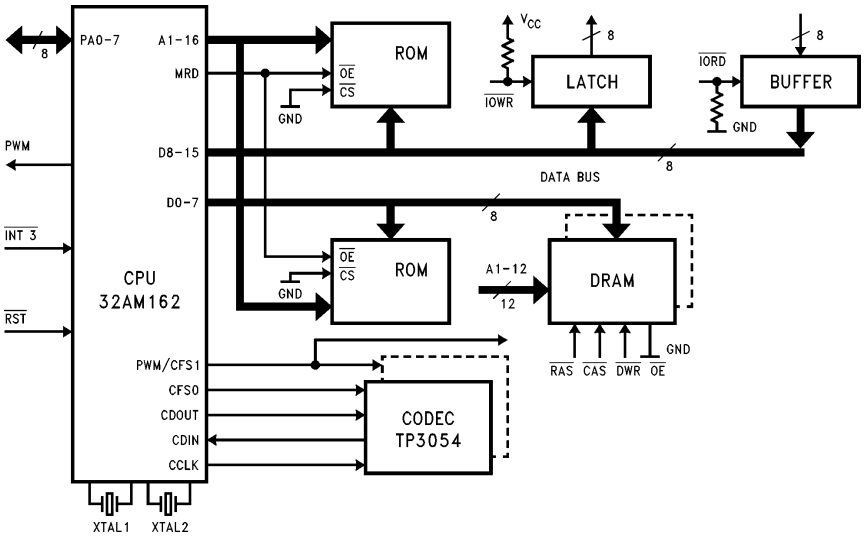


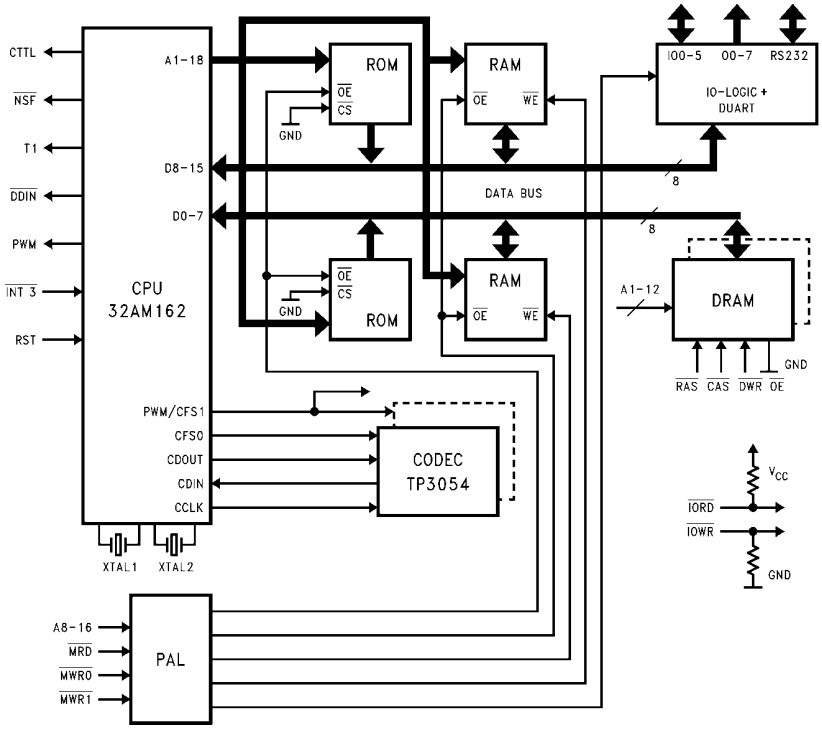
FIGURE 1-1. NS32AM162—Internal ROM Mode

1.0 Product Introduction (Continued)



TL/EE/11732–3

FIGURE 1-2. NS32AM162—External ROM Mode



TL/EE/11732–4

FIGURE 1-3. NS32AM162—Development Mode

1.0 Product Introduction (Continued)

The NS32AM162 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 NS32AM162 features that are shared with other members of the family are provided below:

Powerful Addressing Modes. Eight 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 NS32AM162 has 32-bit address pointers that can address up to 4 gigabytes 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.

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

- High-Level Language Support
- Application Flexibility

1.1 NS32AM162 SPECIAL FEATURES

In addition to the above Series 32000 features, the NS32AM162 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 voice applications.

Graphics support is provided by seventeen instructions that allow operations such as BITBLT, data compression/expansion, fills, and line drawing, to be performed very efficiently.

The NS32AM162 allows systems to be built with either no or 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 down mode to minimize the power consumption.

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 NS32AM162 graphics instructions can be found in the NS32CG16 Printer/Display Processor Programmer's Reference Supplement.

2.0 Architectural Description

2.1 REGISTER SET

The NS32AM162 has 45 internal registers and a 2.1 Kbyte RAM array. 16 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 29 belong to the DSP Module and to the on-chip peripherals. *Figure 2-1* shows the NS32AM162 internal registers.

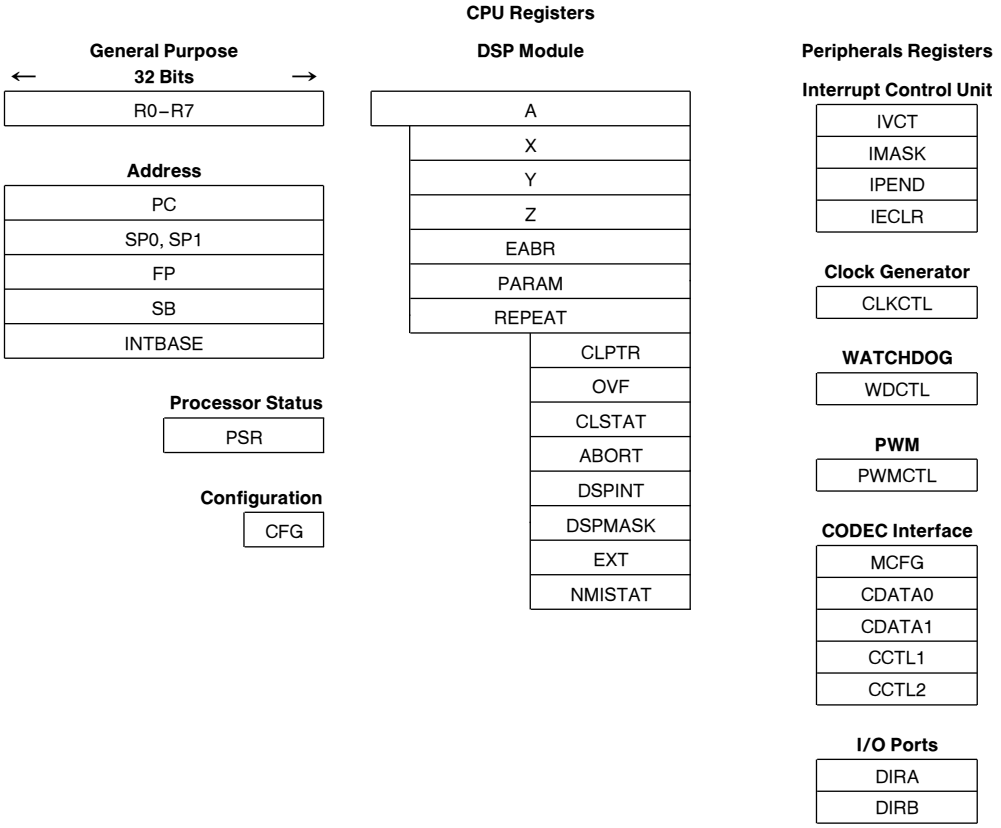


FIGURE 2-1. NS32AM162 Internal Registers

2.0 Architectural Description (Continued)

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.

2.1.2 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).

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.

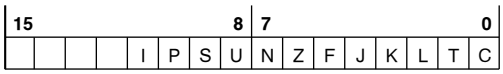


FIGURE 2-2. Processor Status Register (PSR)

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.

2.0 Architectural Description (Continued)

2.1.4 Configuration Register

The Configuration Register (CFG) is 32 bits wide, of which 5 bits are implemented.

CFG is programmed by the SETCFG instruction. Whenever the program writes into CFG, a 0 must be written into bits 1, 2, 3.

The user must set bit 8 to 1 using the SETCFG[DE] instruction during the initialization of the chip. The format of CFG is shown in *Figure 2-3*. The various control bits are described below.

31	8	7				0
Reserved	1	Res	0	0	0	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.

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 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 3.5 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 3.5.5.3 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

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 $2^{INCREMENT}$ words. The allowed values for the WRAP-AROUND field are 0 through 15. The actual WRAP-AROUND will be $2^{WRAP\text{-}AROUND}$ 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.

2.0 Architectural Description (Continued)

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 0xFFFE0000. Accessing external memory with an 0xFFFE0000 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 3.5.

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 (see Section 3.5.5.7). 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 $\overline{IOUT}$ output pin whenever the DSPINT.HALT bit is set to ''1'', and the DSPMASK.HALT bit is unmasked (set to ''1''). See Section 6.1 for the functionality of $\overline{IOUT}$. 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 4	3	2	1	0
Reserved	WD	ERR	UND	Res

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, a reserved DSPM instruction (Section 3.5), the NMISTAT.UND bit is set to ''1''.

When the WATCHDOG is not cleared in time (see Section 3.4.6), the NMISTAT.WD bit is set to ''1''.

When one of the bits in NMISTAT is set to ''1'', a 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.

2.1.6 Interrupt Control Unit (ICU) Register

IVCT—Interrupt Vector Register

Byte wide. Read only. IVCT holds the encoded number of the highest priority unmasked pending interrupt request. Interrupt vector numbers are always positive, in the range 0x11 to 0x14.

7	6	5	4	3	2 0
0	0	0	1	0	VECTOR

FIGURE 2-15. IVCT Register Format

IMASK—Mask Register

Byte wide. A value of ''0'' in bit position i (i in [1..4]) disables the corresponding interrupt source. IMASK bits 0 and 5 through 7 are reserved. The non-reserved bits of IMASK register are cleared to ''0'' upon reset, and when CLKCTL.PDM is ''1''.

7 5	4	3	2	1	0
Reserved	M4	M3	M2	M1	Reserved

FIGURE 2-16. IMASK Register Format

IPEND—Interrupt Pending Register

Byte wide. Read only. Reading a value of ''1'' in bit position i (i in [1..4]) indicates that the respective interrupt source is active. IPEND bits 0 and 5 through 7 are reserved. The non-reserved bits of IPEND are cleared to ''0'' upon reset and when CLKCTL.PDM is ''1''.

7 5	4	3	2	1	0
Reserved	P4	P3	P2	P1	Reserved

FIGURE 2-17. IPEND Register Format

IECLR—Edge Interrupt Clear Register

Byte wide. Write only. A pending edge triggered interrupt is cleared by writing ''1'' to the respective bit position in IECLR. Writing ''0'' has no effect. Note that INT2 does not have a corresponding clear bit in IECLR. INT2 is a level sensitive interrupt, and it is cleared by writing directly to the DSPINT register. IECLR bits 0 and 5 through 7 are reserved.

7 5	4	3	2	1	0
Reserved	CLR4	CLR3	0	CLR1	Reserved

FIGURE 2-18. IECLR Register Format

2.1.7 CODEC Interface Registers

The CODEC Interface contains five 8-bit registers that are used to select the CODEC interface and to communicate with the CODEC. The CODEC Interface registers should not be accessed during Power Down.

MCFG—Modules Configuration Register

This register controls the CODEC Interface configuration. Bits 0–2 of MCFG are desinated CMC (CODEC Mode Control). Bits 3–7 are reserved.

2.0 Architectural Description (Continued)

The different configurations are as follows:

CMC	CODEC Configuration	CODEC Protocol	PWM/CFS1 Output
000	Undefined (Reset)		PWM
001	One Serial CODEC	Short Frame Format	PWM
101	One Serial CODEC	Long Frame Format	PWM
011	Two Serial CODECs	Short Frame Format	CFS1
111	Two Serial CODECs	Long Frame Format	CFS1

All other CMC values are reserved.

Upon reset, CMC is set to 000. It must be set to the appropriate value prior to any reference to the CODEC, or to the PWMCNT register, or to the other CODEC Interface registers. The value of MCFG is retained during power down.

CDATA0—CODEC0 Data

Data to be transferred to CODEC0 is written by software to this register. It is shifted serially to that CODEC following the next frame sync signal. Data that was shifted in from CODEC0 can be read by software from this register. Bit 7 is shifted first. The value of CDATA0 is unpredictable during the shift itself.

CDATA1—CODEC1 Data

Data to be transferred to CODEC1 is written by software to this register. It is shifted serially to that CODEC following the next frame sync signal. Data that was shifted in from CODEC1 can be read by software from this register. Bit 7 is shifted first. The value of CDATA1 is unpredictable during the shift itself.

CCTL1, CCTL2—CODEC Clock Control

Control the CODEC frame sync clock and master clock (CCLK). The NSVOICE software package supports two sampling rates—8000 Hz and 7273 Hz. The respective CCTL1 and CCTL2 values are the following:

Sampling Rate	CCLK Frequency	CCTL1	CCTL2
8000 Hz	2.048 MHz	0	33 (hex)
7273 Hz	1.862 MHz	0	23 (hex)

Upon reset and upon exit from Power Down mode, CCTL2 is set to 33 (hex), and CCTL1 to 0, selecting a sampling rate of 8000 Hz.

2.1.8 Pulse Width Modulator (PWM) Registers

The Pulse Width Modulator (PWM) contains one 8-bit register (PWMCTL) that controls the duty cycle of the 80 KHz PWM output.

See Section 3.4.4 for a detailed description of the operation of the PWM.

2.1.9 Clock Generator Registers

The Clock Generator contains one 8-bit register that controls the high frequency oscillator and the power down mode.

CLKCTL—Clock Generator Control Register

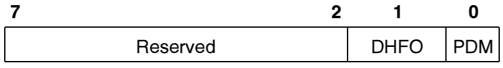


FIGURE 2-19. CLKCTL Register Format

PDM Power Down Mode Control
PDM = 0 : normal operation mode
PDM = 1 : power down mode

DHFO Disable High Frequency Oscillator
DHFO = 0 : High Frequency Oscillator enabled
DHFO = 1 : High Frequency Oscillator disabled

2.1.10 WATCHDOG (WD) Registers

The WATCHDOG (WD) contains one 8-bit register (WDCTL) that controls the operation of the WD.

See Section 3.4.6 for a detailed description of the operation of the WD.

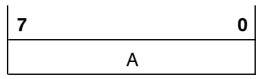
2.1.11 I/O Ports Registers

The I/O Ports block contains two 8-bit write-only registers, DIRA and DIRB, that control the direction of the bits of Port A and Port B respectively.

A ''1'' in one of the bits configures the associated I/O pin as an output. A ''0'' configures it as an input.

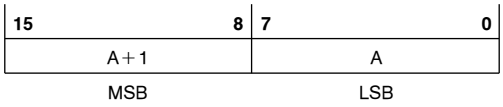
2.2 MEMORY ORGANIZATION

The main memory of the NS32AM162 is a uniform linear address space. Memory locations are numbered sequentially starting at zero and ending at $2^{32} - 1$. 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.



Byte at Address A

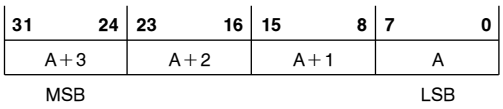
Two contiguous bytes are called a word. Except where noted, the least significant byte of a word is stored at the lower address, and the most significant byte of the word is stored at the next higher address. In memory, the address of a word is the address of its least significant byte, and a word may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



Word at Address A

2.0 Architectural Description (Continued)

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.



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 NS32AM162 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.

Figure 2-20 shows the NS32AM162 address mapping.

First Address (Hex)	Last Address (Hex)	
00000000	000063FF(1)	Internal ROM Mode Internal ROM (25 Kbytes)
00000000	0001FFFF	External ROM Mode External Memory
00000000	0007FFFF	Development Mode External Memory
02000000	027FFFFF	External DRAM
FFFDFC10	FFFDFFFF	System On-Chip RAM (1008 Bytes)
FFFE0000	FFFE045F	DSPM Internal RAM (1120 Bytes)
FFFF8000	FFFF8027	DSPM Dedicated Registers
FFFF9000	FFFF9013	DSPM Control/Status Registers
FFFFA000	FFFFA047	On-Chip Modules Registers
FFFFFE00	FFFFFFFF	ICU and NMI Control

All other address ranges are reserved.

FIGURE 2-20a. NS32AM162 Address Mapping

Note 1: 00007FFF in the NS32AM163 (32 Kbytes)

Module	Register	Address
ICU	IVCT	FFFFFE00
	IMASK	FFFFFE04
	IPEND	FFFFFE08
	IECLR	FFFFFE0C
I/O	DIRA	FFFFA101
	DIRB	FFFFA201
	PORTA	FFFFA401
	PORTB	FFFFA501
	PORTC	FFFFA601
Clock Generator	CLKCTL	FFFFA010
WATCHDOG	WDCTL	FFFFA000
PWM	PWMCTL	FFFFA020
CODEC Interface	MCFG	FFFFA024
	CDATA0	FFFFA028
	CDATA1	FFFFA02A
	CCTL1	FFFFA02C
	CCTL2	FFFFA02E

FIGURE 2-20b. NS32AM162 Modules Address Mapping

2.3 INSTRUCTION SET

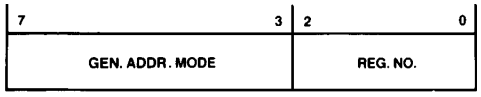
2.3.1 General Instruction Format

Figure 2-22 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 2-23*, 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 2.3.3).



TL/EE/11732–5

FIGURE 2-21. Index Byte Format

2.3.2 Addressing Modes

The NS32AM162 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 NS32AM162 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.

NS32AM162 Addressing Modes fall into eight 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.

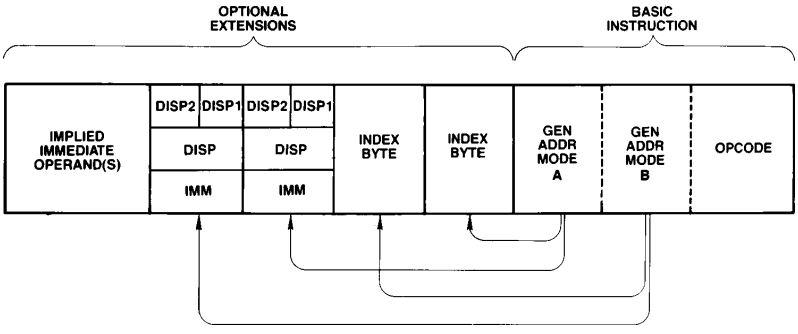
Absolute: The address of the operand is specified by a displacement field in the instruction.

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.

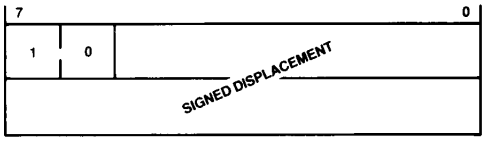


TL/EE/11732–6

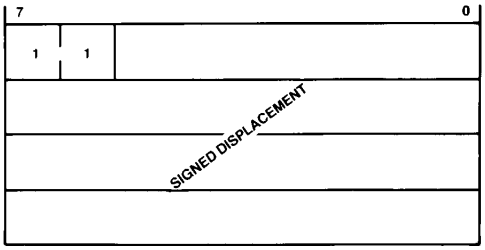
FIGURE 2-22. General Instruction Format

Byte Displacement: Range −64 to +63

Word Displacement: Range −8192 to +8191



Double Word Displacement:
Range (Entire Addressing Space)



TL/EE/11732–7

FIGURE 2-23. Displacement Encodings

TABLE 2-1. NS32AM162 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.
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.

2.0 Architectural Description (Continued)

2.3.3 Instruction Set Summary

Table 2-2 presents a brief description of the NS32AM162 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, 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. NS32AM162 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. NS32AM162 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.

TABLE 2-2. NS32AM162 Instruction Set Summary (Continued)

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.

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	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	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)

TABLE 2-2. NS32AM162 Instruction Set Summary (Continued)

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	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	CBITi	gen,gen	Test and clear bit.
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.

2.4 GRAPHICS SUPPORT

The following sections provide a brief description of the NS32AM162 graphics support capabilities. Basic discussions on frame buffer addressing and BITBLT operations are also provided. More detailed information on the NS32AM162 graphics support instructions can be found in the NS32CG16 Printer/Display Processor Programmer's Reference.

2.4.1 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 2-24*. 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.

2.0 Architectural Description (Continued)

Example 1 below shows two NS32AM162 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, 0RG           ; SET PIXEL
```

Instruction Sequence 2:

```
INDEXD R1, (WARP-1), R0  ; Y*WARP + X
SBITD  R1, 0RG           ; 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

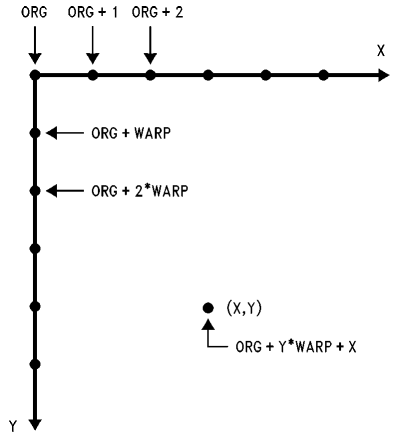
Instruction Sequence:

```
INDEXD  R1, (WARP-1), R0  ; BIT ADDRESS
SBITD   41, 0RG           ; SET FIRST PIXEL

ADDQD   1, R1             ; (X+1, Y)
SBITD   R1, 0RG           ; SECOND PIXEL

ADDD    (WARP-1), R1      ; (X, Y+1)
SBITD   R1, 0RG           ; THIRD PIXEL

ADDQD   1, R1             ; (X+1, Y+1)
SBITD   R1, 0RG           ; LAST PIXEL
```



TL/EE/11732–8

FIGURE 2-24. Correspondence between Linear and Cartesian Addressing

2.4.2 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.

2.4.2.1 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.

2.4.2.2 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 2-25*, 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.

2.4.2.3 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 2-25* 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.

2.0 Architectural Description (Continued)

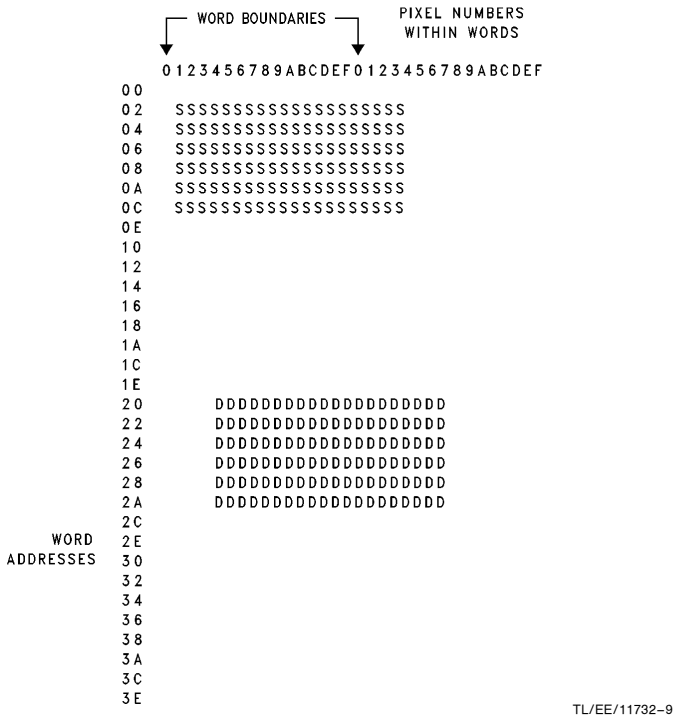


FIGURE 2-25. 32-Pixel by 32-Scan Line Frame Buffer

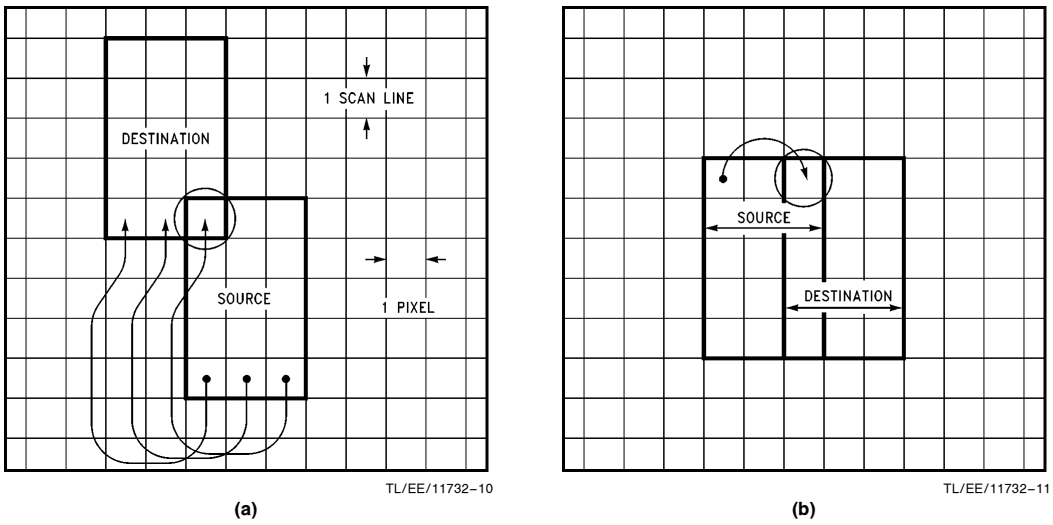


FIGURE 2-26. Overlapping BITBLT Blocks

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) instruction uses different set up parameters, and techniques.

2.4.2.4 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 increment addresses;
             (once per word).
UNTIL        done horizontally
UNTIL        done vertically
RETURN       (from BITBLT).
```

Note: In the NS32AM162 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 2-26(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 2-26(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 occurs 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 2-26(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.

2.4.3 GRAPHICS SUPPORT INSTRUCTIONS

The NS32AM162 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.

2.4.3.1 BITBLT (BIT-aligned BLock Transfer)

This group includes six instructions. They are used to move characters and objects into the frame buffer which will be printed or displayed.

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 2-25* masking was defined relative to the destination block.

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

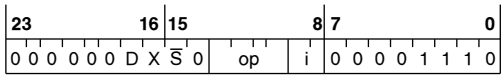
Note 3: BBFOR performs the same operation as BBOR with IA and S options.

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 2-27* and Table 2-3 show the format of the BB instructions and the encodings for the 'op' and 'i' fields.



- 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 2-27. 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 2-28*.

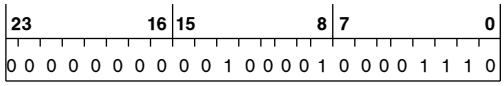


FIGURE 2-28. BITWT Instruction Format

2.4.3.2 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 2-29*.

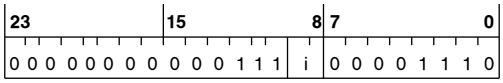


FIGURE 2-29. MOVMPi Instruction Format

2.4.3.3 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

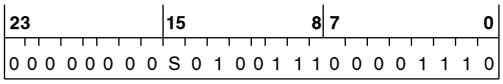
Option:
1 count set bits until a clear bit is found
0 count clear bits until a set bit is found

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 2-30* shows the TBITS instruction format.



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

FIGURE 2-30. TBITS Instruction Format

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 2-31* shows the SBITS instruction format.

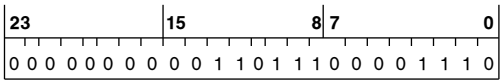


FIGURE 2-31. SBITS Instruction Format

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 2-32* shows the SBITPS instruction format.

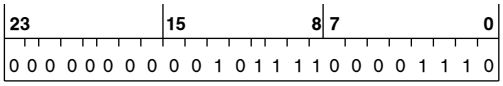


FIGURE 2-32. SBITPS Instruction Format

2.4.3.3.1 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 NS32AM162 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 NS32AM162 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 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 $\overline{RST}$ signal is asserted, the CPU enters the reset state. The CPU remains in the reset state until the $\overline{RST}$ 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.

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.

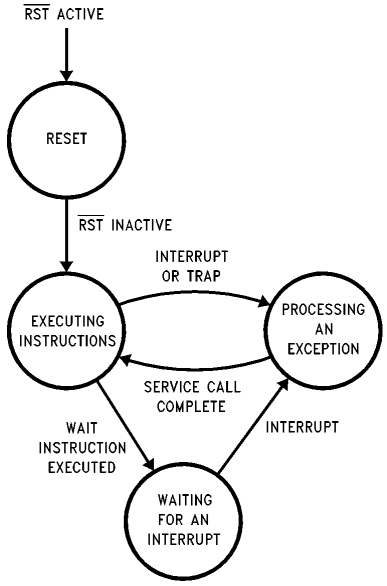
While in the Processing-An-Exception state, the CPU saves the PC and PSR register contents on the stack.

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 NS32AM162 checks for exceptions at various points while executing instructions. Certain exceptions, like interrupts, are in most cases recognized between instructions.



TL/EE/11732–12

FIGURE 3-1. Operating States

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.
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.

3.2 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 signaled by activating the $\overline{\text{NMI}}$ or $\overline{\text{INT3}}$ input signals. 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, and the PSR 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 3.6.4 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.

2) Vector Acquisition.

   A Vector is either obtained from the Data Bus or is supplied by default.

3) Service Call.

   The Vector is used as an index into the Interrupt Dispatch Table, whose base address is taken from the CPU Interrupt Base (INTBASE) Register. See *Figure 3-2*. A 32-bit address of the exception service procedure is read from the table entry, and is loaded into the PC register.

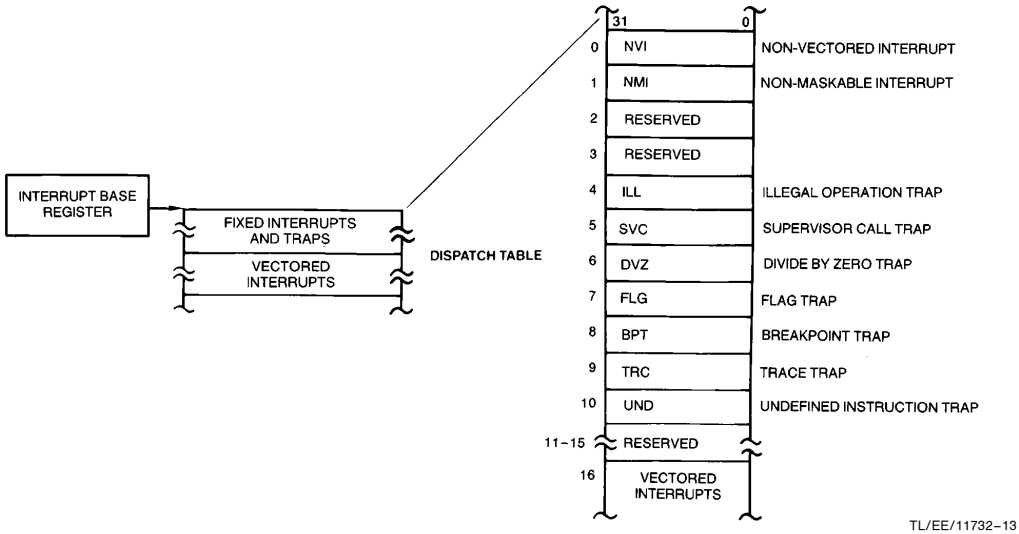


FIGURE 3-2. Interrupt Dispatch and Cascade Tables

This process is illustrated in *Figure 3-3,* from the viewpoint of the programmer.

Details on the sequences of events in processing interrupts and traps are given in the following sections.

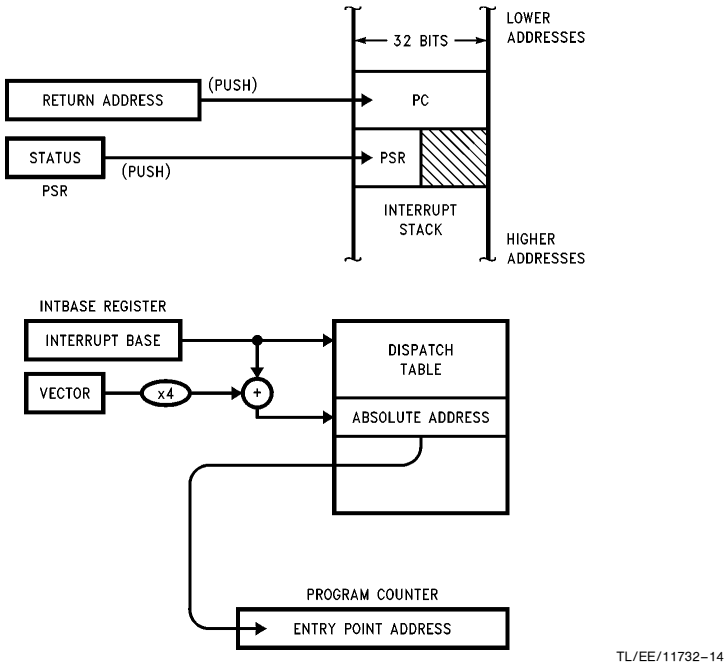


FIGURE 3-3. Exception Acknowledge Sequence

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 a 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 any external interrupt control units 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 PSR and the PC registers to their previous contents.

3.2.3 Maskable Interrupts

Maskable interrupt requests are generated either externally through the $\overline{INT3}$ pin or internally. These requests are enabled to generate an interrupt only while the I-bit in the PSR register is set to 1. The I-bit is automatically cleared during service of a maskable interrupt or NMI, and is restored to its original setting upon return from the interrupt service routine via the RETT or RETI instruction.

Maskable interrupts can be configured through the I-bit in the CFG register to be either non-vectored (CFG bit I = 0) or vectored (CFG bit I = 1).

If the non-vectored mode is selected, a default vector value of zero is always used. For the vectored mode instead, the on-chip Interrupt Control Unit will provide the CPU with a vector value. This vector value is then used as an index into the Dispatch Table in order to find the entry 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.

3.2.3.1 Non-Vectored Mode

In the Non-Vectored mode, an interrupt request 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.

3.2.4 Non-Maskable Interrupt

The Non-Maskable Interrupt is triggered whenever one of the bits in the NMISTAT register is set to ''1''. The CPU performs an ''Interrupt Acknowledge'' bus cycle from Address $FFFFFF00_{16}$ 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 7 trap conditions recognized by the NS32AM162 as described below.

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.

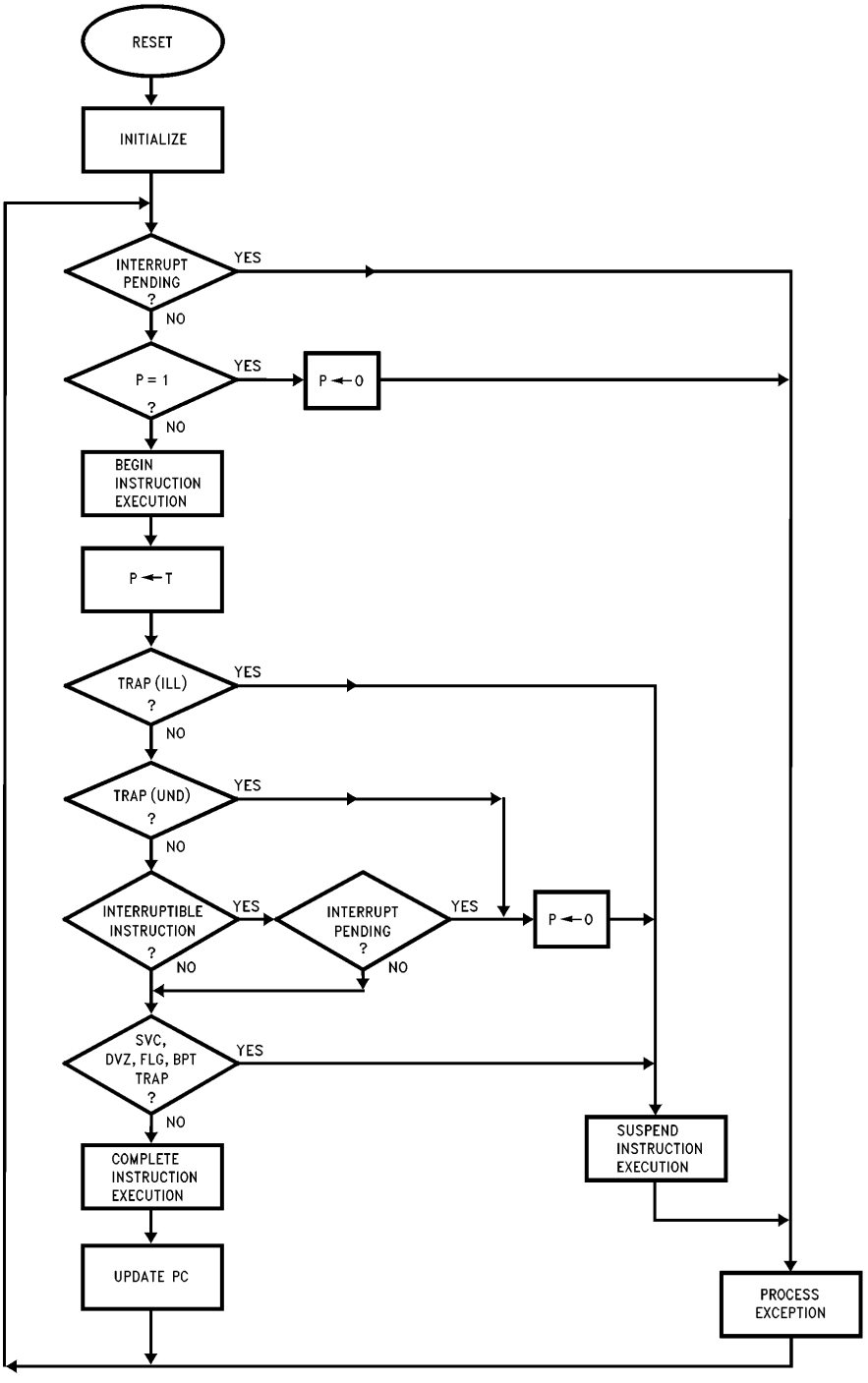
Figure 3-4 shows an exception processing flowchart.

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 6 mutually exclusive traps: 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/11732–15

FIGURE 3-4. Exception Processing Flowchart

3.0 Functional Description (Continued)

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 3-5*.

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 an NMI request is active, or an interrupt request is 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, 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 $FFFFFF00_{16}$, applying Status Code 0100 (Interrupt Acknowledge). Discard the byte read.
   b. Set ''Vector'' to 1.
   c. Go to Step 6.
4. If the interrupt is Non-Vectored:
   a. Read a byte from address $FFFFFE00_{16}$, applying Status Code 0100 (Interrupt Acknowledge). Discard the byte read.
   b. Set ''Vector'' to 0.
   c. Go to Step 6.
5. Here the interrupt is Vectored.
   a. Read ''Byte'' from address $FFFFFE00_{16}$, applying Status Code 0100 (Interrupt Acknowledge).
   b. Read vector byte from the IVECT register of the on-chip Interrupt Control Unit.
6. Perform Service (Vector, Return Address), *Figure 3-5.*

3.2.7.2 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.
   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 3-5*.

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 3-5*.

Service (Vector, Return Address):

1. **Push the PSR copy onto the Interrupt Stack as a 16-bit value.**
2. **Read the 32-bit Interrupt Dispatch Table (IDT) entry: address is Vector*4+INTBASE Register contents.**
3. **Place the IDT entry in the Program Counter.**
4. **Push the Return Address onto the Interrupt Stack as a 32-bit quantity.**
5. **Flush Queue: Non-sequentially fetch first instruction of Interrupt Routine.**

FIGURE 3-5. Service Sequence
Invoked during All Interrupt/Trap Sequences

3.3 DEBUGGING SUPPORT

The NS32AM162 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

TABLE 3-1. 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
SVC, DVZ, FLG, BPT, ILL	Suspended	None	TUSP
TRC	Before Instruction	P	TUS

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 NS32AM162 instructions which have interior loops (BBOR, BBXOR, BBAND, BBFOR, 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 NS32AM162 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 NS32AM162 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 NS32AM162 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 NS32AM162-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 ON-CHIP PERIPHERALS

3.4.1 Interrupt Controller Unit

The Interrupt Control Unit (ICU) monitors the internal and external interrupt sources and generates a vectored interrupt to the NS32AM162 when required. Priority is resolved on a fixed scheme. Each interrupt source can be masked by a mask register. Pending interrupts can be polled using the interrupt pending register.

The ICU handles four sources of interrupts: three of them are internal, and one external. The external interrupt is triggered by a falling edge on the $\overline{INT3}$ input pin. The $\overline{INT3}$ has a Schmitt Trigger input buffer in order to produce jitter-free interrupt requests out of slowly changing input signals. An on-chip circuit synchronizes $\overline{INT3}$ to the NS32AM162 clock. For proper interrupt detection, $\overline{INT3}$ must be pulled low for at least 3 clock cycles.

Another interrupt, INT2, is level sensitive. It is triggered by the DSPM upon completion of a command list execution and when both DSPINT.HALT and DSPMASK.HALT are ''1''. INT2 is used to synchronize between command list execution, and a core program. This can reduce the total CPU utilization of applications which require asynchronous operation of the DSPM.

The other two interrupts are called INT4 and INT1 and are edge sensitive. They are triggered by the falling edge of the CODEC and 500 Hz clocks respectively. These clocks are generated in the Clock Generation Unit.

INT4 is used for timing the accesses to the CODEC. The same clock that triggers the interrupt is also connected to the $\overline{CFS}$ input of the CODEC.

All the interrupts are latched by the interrupt pending register (IPEND). An edge sensitive pending interrupt is cleared by writing to the edge interrupt clear register (IECLR). The INT4 pending bit is also reset when the CODEC is accessed.

There is no hardware limitation on nesting of interrupts. Interrupt nesting is controlled by software writing into the mask register (IMASK). When an interrupt is acknowledged by the core, the PSR.I bit is cleared to ''0'', thus disabling interrupts. While an interrupt is in service, the user may allow other interrupts to occur by setting PSR.I bit to ''1''. The IMASK register can be used to control which of the other interrupts is allowed. Clearing bits in the IMASK register should be done while the PSR.I bit is ''0''. Setting bits in the IMASK register may be done regardless of the PSR.I bit state.

Clearing an interrupt request before it is serviced may cause a false interrupt, where the NS32AM162 may detect an interrupt not reflected by IVCT. The user is advised to clear interrupt requests only when interrupts are disabled.

During power down mode (CLKCTL.PDM = ''1''), the ICU is disabled. The user must clear the PSR.I bit to ''0'' before entering power down mode, and should not attempt to read or write the ICU registers while in this mode.

3.0 Functional Description (Continued)

3.4.1.1 Interrupt Sources

Name	Type	Source	Vector	Priority
INT1	2 ms	Clock Generator	0x11	Lowest Priority
INT2	DSPM	DSPM	0x12	
INT3		External	0x13	
INT4	CODEC	Clock Genertor	0x14	Highest Priority

3.4.2 BIU and DRAM Controller

The BIU controls all the internal and external accesses. It provides control signals for the internal cycles to the other on-chip modules. It also provides control signals to four types of external devices: DRAM, ROM/RAM, CODEC, and I/O ports. Different type of accesses are done to each of the different devices.

The BIU provides four types of accesses to the DRAM: read, write, refresh cycles during normal operation, and special refresh cycles during power down mode (CLKCTL.PDM = ''1''). No reads and writes to the DRAM are allowed in power down mode.

The BIU provides two type of accesses to the ROM/RAM devices: read and write cycles. These cycles can also be performed in power down mode.

The BIU provides two type of accesses to the CODEC: read and write cycles. These cycles are not allowed in power down mode.

The BIU provides two type of accesses to I/O devices in External ROM mode and in Development mode: read and write cycles. These cycles can also be performed in power mode.

All control signals of external devices are inactive while reset.

3.4.2.1 DRAM Accesses

The DRAM Controller (DRAMC) supports transactions between the NS32AM162 and external DRAM and performs refresh cycles. The DRAMC supports 1M x 4, 1M x 1, 4M x 1 or 4M x 4 DRAM devices. The supported DRAM devices require minimum 500 ns cycle time and minimum 350 ns $\overline{RAS}$ access time, and a short refresh period.

The external data bus used for all DRAM accesses is 8-bit wide. There is no hardware support for nibble or byte gathering. The user can handle the nibble gathering with software. CPU accesses are only to an aligned word in the DRAM (byte or double word accesses are not allowed).

During read cycles the DRAMC provides the $\overline{RAS}$ and $\overline{CAS}$ signals. The DRAMC does not use fast page mode accesses. The user must connect the $\overline{OE}$ pin of the DRAM to GND. On write cycles the DRAMC provides the $\overline{RAS}$, $\overline{CAS}$, and $\overline{WE}$ signals to perform early writes according to the DRAM specifications.

When the NS32AM162 enters the power down mode, the DRAMC continues to refresh the DRAM array. The low frequency clock generates $\overline{RAS}$ and $\overline{CAS}$ signals. In this mode no reads and writes to the DRAM are allowed. Note also that the user must make sure that the instruction that sets CLKCTL.PDM bit does not directly follow an access to the DRAM.

The DRAM address range is 0x02000000 to 0x027FFFFF, and its size is 8 Mbytes. In a typical system, where only a single 1M x 4-DRAM device is used, only 2 Mbytes are accessible, and only one nibble out of four can actually store data.

During reads and writes to the DRAM in Internal ROM mode, the DRAMC provides the row and column address on pins A1–A11 and RA12. The row address is bits A11–A22 of the data item's address. It is provided on pins A1–A11 and RA12. The column address is bits A1–A10 of the data item's address. It is provided on pins A1–A10.

During reads and writes to the DRAM in External ROM or Development modes, the DRAMC provides the row and column address on pins A1–A12. The row address is bits A11–A22 of the data item's address. It is provided on pins A1–A12. The column address is bits A1–A10 of the data item's address. It is provided on pins A1–A10.

DRAM accesses can be divided into two parts: During the first part (11 cycles), the external data bus is used by the DRAMC. During the following 2 cycles, the external data bus can be used to access every device except for the DRAM (to ensure enough DRAM precharge time).

In normal operation (CLKCTL.PDM = ''0''), DRAM refresh is done at a rate of 160000 cycles/second. The refresh clock is generated by the clock generator block. Any bus transaction except for DRAM accesses can be performed in parallel with a refresh cycle.

In power down mode (CLKCTL.PDM = ''1''), DRAM refresh is done at a ¼ of the low speed crystal oscillator frequency (If Crystal-2 is 455 kHz, the refresh rate is 113750 cycles/second). The $\overline{RAS}$ and $\overline{CAS}$ signals are activated for half a DRAM refresh cycle.

In both modes, the DRAM controller provides control signals to execute automatic ($\overline{CAS}$ before $\overline{RAS}$) refresh cycles.

3.4.2.2 CODEC Interface

The NS32AM162 provides an on-chip interface to one or two serial CODECs. The interface supports two CODEC modes of operation—long frame format and short frame format.

Selecting the CODEC interface is done through the MCFG register.

CODEC accesses are done as regular memory accesses to the addresses of the CODEC Interface registers.

The CODEC interface uses five signals—CDIN, CDOUT, CCLK, CFS0 and CFS1. When one CODEC is used, the interface uses CDIN, CDOUT, CCLK and CFS0. When two CODECs are used, they share CDIN, CDOUT and CCLK. One CODEC receives CFS0 and the other CODEC receives CFS1.

The master clock CCLK and the sampling rate are controlled by the CCTL1 and CCTL2 registers. Two values can be used, depending on the required sampling rate, as shown below:

Sampling Rate	CCLK Frequency	CCTL	CTTL1	CCTL2
8000 Hz	2.048 MHz	20.48 MHz	0	33 (hex)
7273 Hz	1.862 MHz	20.48 MHz	0	23 (hex)

Data is transferred to the CODEC through the CDOUT pin. Data is read from the CODEC through the CDIN pin. The CPU core accesses the CODECs through the CDATA0 and CDATA1 registers.

3.0 Functional Description (Continued)

When a short frame format is selected via the MCFG register (CMC = 001 or 011), data transfer between the NS32AM162 and the serial CODEC starts by asserting (high) the CFS0 frame sync signal. After one CCLK cycle, CFS0 is de-asserted, data from the NS32AM162 is sent to the CODEC through CDOUT, and simultaneously data from the CODEC is sent to the NS32AM162 through CDIN. After eight bits are shifted out (these are the bits of the CDATA0 register), CFS1 is asserted for one CCLK cycle, and then the eight bits of CDATA1 are shifted out through CDOUT, while eight bits from the CODEC are shifted in through CDIN. See *Figure 3-6*.

When a long frame format is selected via the MCFG register (CMC = 101 or 111), data transfer between the NS32AM162 and the serial CODEC starts by asserting (high) the CFS0 frame sync signal. When CFS0 is asserted, data from the NS32AM162 is sent to the CODEC through CDOUT, and simultaneously data from the CODEC is sent to the NS32AM162 through CDIN. After eight bits are shifted out (these are the bits of the CDATA0 register), CFS0 is de-asserted. One CCLK cycle later CFS1 is asserted, and the eight bits of CDATA1 are shifted out through CDOUT, while eight bits from the CODEC are shifted in through CDIN. See *Figure 3-7*.

Note that the bits of CDATA1 are shifted out as part of the protocol, regardless of whether one or two CODECs are used in the system.

The CODEC interrupt is issued after data to both CODECs is transferred. This is regardless of the actual number of CODECs in the system. The CODEC interrupt pending bit is cleared either by writing "1" to the CLR4 bit of the IECLR register, or by accessing CDATA0 or CDATA1. In order to ensure proper operation, after a CODEC interrupt, the software must first read CDATA0 (and CDATA1 if there are two CODECs), and then write new data into CDATA0 (and CDATA1 if there are two CODECs), before the next frame sync clock. Failure to update a register before the next frame sync clock will cause a value of FF (hex) to be sent from that register.

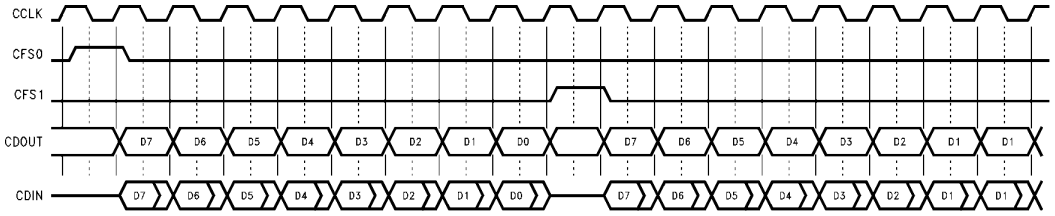
Note: In cases where two serial CODECs are used, but the PWM output is needed, the user can program the MCFG register to indicate one serial CODEC, and restore CFS1 using an external circuit. This circuit can use a 9-bit shift register, whose data input is connected to CFS0, and whose clock input is connected to CCLK.

3.4.2.3 Accesses to Off-Chip Memory Devices

In the External ROM mode, the NS32AM162 performs read accesses from external memory for all the addresses between 0x00000000 and 0x0001FFFF. In the Development mode, the NS32AM162 performs read or write accesses to external memory for all the addresses between 0x00000000 and 0x0007FFFF.

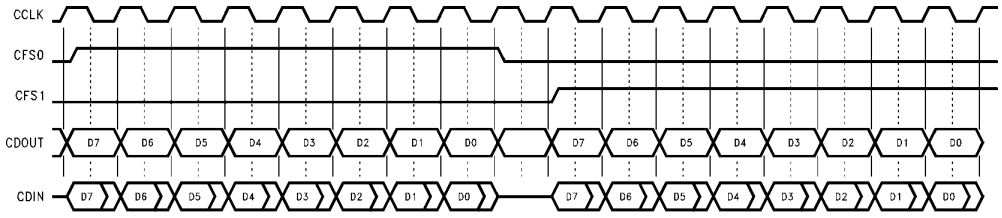
On the first cycle (T1) of a read access, the NS32AM162 asserts A1–A16, in the External ROM mode, or A1–A18 in the Development mode. The address remains active for four clock cycles (T1 through T4). In the following cycle (T2), the NS32AM162 activates the $\overline{\text{MRD}}$ signal. $\overline{\text{MRD}}$ remains active until the fourth cycle (T4). Data is sampled at the end of the third cycle (T3). See Section 4.4.3 for detailed timing diagrams.

On the first cycle (T1) of a write access, the NS32AM162 in the Development mode asserts A1–A18. The address remains active for four clock cycles (T1 through T4). In the following cycle (T2), D0–D15 are activated, and $\overline{\text{MWR0}}$ and $\overline{\text{MWR1}}$ are asserted (depending on the byte needed to be written into). D0–D15 remains active until the next T1. $\overline{\text{MWR0}}$ and $\overline{\text{MWR1}}$ remain active until the fourth cycle (T4). See Section 4.4.3 for detailed timing diagrams.



TL/EE/11732–16

FIGURE 3-6. CODEC Protocol—Short Frame



TL/EE/11732–17

FIGURE 3-7. CODEC Protocol—Long Frame

3.0 Functional Description (Continued)

3.4.3 I/O Ports

Three 8-bit I/O ports are provided in the Internal ROM mode: PA, PB and PC. Each of the bits in Ports A and B can be individually programmed as either an input or as an output. Programming the direction of the bits in ports PA and PB is done by writing to registers DIRA and DIRB respectively. Writing ''1'' to one of the bits in a DIR register configures the corresponding bit in the port as an output port. Writing ''0'' to one of the bits in a DIR register configures the corresponding bit in the port as an input. Port PC serves as an output only, and does not have a direction control register. On reset, DIRA and DIRB are cleared to ''0'', and ports PA and PB are initiated as input ports.

The bits in ports PA and PB that are programmed as outputs can also be read by the CPU by accessing the port. The values of the output in ports PA, PB, and PC can be set by writing to the port.

In the External ROM and Development modes the pins of ports PB and PC are used for different functions. In order to use these ports, external logic can be added. An external latch can be connected to the D8–D15, and $\overline{IOWR}$ signals to provide the functionality of PC. An external buffer can be connected to the D8–D15 and $\overline{IORD}$ signals to provide part of the functionality of PB. Note that in this mode PB can serve as an input only.

In the Development mode, PA pins are also used, and hence there are no ports available in this mode.

Accesses to the external latch and external buffer are similar to the accesses to off-chip memory devices, except for the pins that control the actual reads and writes. On reads, $\overline{IORD}$ is asserted, and on writes, $\overline{IOWR}$ is asserted. The timings of these signals are exactly the same as the timings of $\overline{MRD}$ and $\overline{MWR1}$.

3.4.4 Pulse Width Modulator

The Pulse Width Modulator provides one output signal, with a fixed frequency and a variable duty cycle. The frequency of the PWM output is 80 KHz. The duty cycle can be programmed by writing a value from 0 to 0xFF to the PWMCTL register. The PWM output is active (high) for the number of 20.48 MHz cycles specified in the PWMCTL register. It is inactive (low) for the rest of the 20.48 MHz cycles in the 80 KHz PWM cycle. During power down mode, and upon reset, PWMCTL register is cleared to ''0'', and the PWM output signal is not active (low). The PWM output pin is shared with the CFS1 pin of the CODEC interface. Consequently, when the MCM field in the MCFG register is set to 011 or 111, to select a direct interface to two CODECs, the PWM output signal is not available.

3.4.5 Clock Generator

The clock generator provides all the clocks needed for the various parts of the device. Two crystal oscillators provide the basic frequencies needed. The high-speed crystal oscillator is designed to operate with a 40.96 MHz crystal. The low-speed oscillator is designed to operate with a ceramic resonator at a frequency of 455 KHz. The user can operate the NS32AM162 in either normal operation or power down modes. In power down mode, most of the on-chip modules are running from a very low frequency clock or are totally disabled. In power down mode, the user can turn off the high speed crystal oscillator to further reduce the power.

The clock generator provides two clocks to the CODEC: a 1.28 MHz clock, and an 8 KHz clock. The 8 KHz clock also generates INT4.

The clock generator provides a 2 ms (0.5 KHz) time base for the system software. This time base signal generates INT1.

The clock generator provides a refresh request signal at a rate of 160 KHz during normal operation mode, and a ¼ of Crystal-2 frequency in power down mode.

The operation of the clock generator is affected by moving to power down mode. See Section 3.6.3 for a description of this mode.

3.4.6 WATCHDOG Counter

The WATCHDOG (WD) counter is used to activate a Non-Maskable Interrupt (NMI) whenever the software is out of control. The WD module is a 10 Hz timer with a reset mechanism. During normal operation mode, the user must clear the WD at a rate higher than 10 Hz by writing 0x0E into the WDCTL register. These write accesses ensure that the WATCHDOG will not issue an NMI for a full 0.1 second. Failing to clear the WD before 0.1 of a second has passed, will cause an NMI. If the user does not clear the WATCHDOG, an NMI occurs exactly ten times a second. This NMI can be used to track the time. Upon reset, the WD is disabled until the first write access to the WDCTL register.

3.4.7 Internal ROM

The size of the internal ROM is 25 Kbytes (32 Kbyte in the NS32AM163). The ROM is organized as a 16-bit wide memory array with a zero wait-state access time. The ROM's starting address is 0x00000000. When the NS32AM162 is in either External ROM or Development modes, the lower 128 Kbytes or 512 Kbytes respectively are mapped to external accesses instead of accesses to the on-chip ROM.

3.4.8 Internal RAM Arrays

The NS32AM162 provides two zero wait-state on-chip RAM arrays: a 1008 byte system RAM array and a 1120 byte DSPM RAM array. The data bus between the CPU and the system RAM array is 16 bits wide. The data bus between the DSPM and its RAM is 32 bits wide, to allow high throughput during DSP operations. While the DSPM is active, the CPU is not allowed to access the DSPM RAM.

3.5 DSP MODULE

The following sections give full specifications for the NS32AM162 on-chip DSP Module.

3.5.1 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 maximiz-

3.0 Functional Description (Continued)

ing 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 4.7.3).

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 3.6.2). 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 3.6.1).

3.5.2 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 0x22F. 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 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)

3.5.2.1 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.

3.5.2.2 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 $2n$)
Integer Value (High)		(Location $2n + 1$)

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.

3.5.2.3 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**).

3.5.2.4 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 $2n$)
Real Value (High)		(Location $2n + 1$)

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.

3.5.2.5 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**).

3.5.2.6 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.

3.5.3 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.

3.5.4 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

3.5.4.1 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.

3.0 Functional Description (Continued)

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 activate interrupt level 2 in the on-chip ICU.

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.

3.5.4.2 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.

3.5.5 DSPM Instruction Set

3.5.5.1 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 $0 \le k < 2^{16}$.
ext__mem[*k*]	A value in external memory whose first byte address is *k*, where $0 \le k < 2^{32}$.
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*].

3.5.5.2 Type Casting

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

integer	An integer value, as described in Section 3.5.2.1.
aligned__integer	An aligned integer value, as described in Section 3.5.2.2.
real	A real value, as described in Section 3.5.2.3.
aligned__real	An aligned real value, as described in Section 3.5.2.4.
extended	An extended-precision real value, as described in Section 3.5.2.5.
complex	A complex value, as described in Section 3.5.2.6.
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.

3.5.5.3 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.

3.5.5.4 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	7	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.

3.5.5.5 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.

3.5.5.6 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.

3.5.5.7 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'', (See Section 3.4.4).

Syntax:

EXEC DBPT

15 11	10 0
10000	111 1111 1110

Note: Accumulator is not affected.

3.5.5.8 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];
  }
}
```

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

3.5.5.10 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	$Z = X \mid Y$
011000	XOR	$Z = X \oplus Y$
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.

3.5.5.11 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.

3.5.5.12 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.

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.

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.

3.5.5.13 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);
  }
}
```

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

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 larger value is stored back into the Z vector. In case the larger value was the accumulator, then the value of X.ADDR 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 = (extended_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.

3.5.5.14 Special Instructions

ESHL—Extended Shift Left

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 polynomials with real coefficients, for a vector of complex-valued arguments.

3.0 Functional Description (Continued)

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.

VESIIR—Vector Extended Single-Pole IIR

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

Syntax:

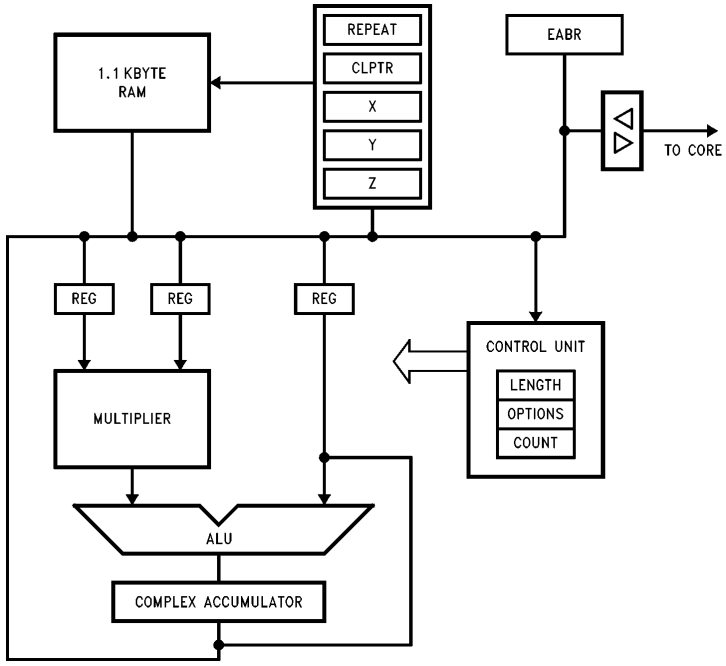
EXEC VESIIR

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.



TL/EE/11732–18

FIGURE 3-8. DSP Module Block Diagram

3.0 Functional Description (Continued)

3.6 SYSTEM INTERFACE

This section provides general information on the NS32AM162 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 Chapter 4.

3.6.1 Power and Grounding

The NS32AM162 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.

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 NS32AM162.

During prototype using wire-wrap or similar methods, the capacitors should be soldered directly to the power pins of the NS32AM162 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 NS32AM162. 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 NS32AM162.

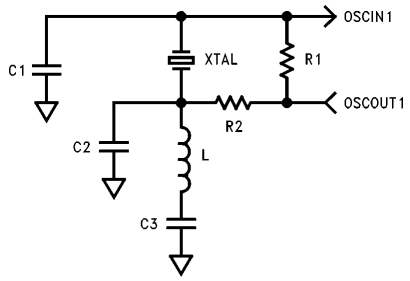
3.6.2 Clocking

3.6.2.1 High Speed Clock Oscillator

The NS32AM162 provides an internal oscillator that interacts with an external High-Speed clock source through two signals; OSCIN1 and OSCOUT1.

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 to OSCIN1 is required; OSCOUT1 should be left unconnected or loaded with no more then 5 pF of stray capacitance.

When operation with a crystal is desired, special care should be taken to minimize stray capacitance and inductance. The crystal, as well as the external components, should be placed in close proximity to OSCIN1 and OSCOUT1 pins to keep the printed circuit trace lengths to an absolute minimum. *Figure 3-9* shows the external crystal interconnections. Table 3-2 provides the crystal characteristics and the values of R, C, and L components, including stray capacitance.



TL/EE/11732–19

FIGURE 3-9. High Frequency Crystal Connections

3.6.2.2 Low Frequency Clock Oscillator

The NS32AM162 provides an internal oscillator that interacts with an external clock Low-Frequency source through two signals; OSCIN2 and OSCOUT2.

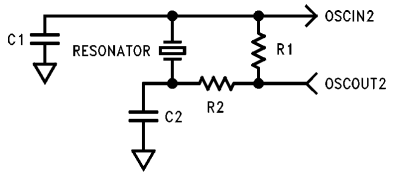
Either an external single-phase clock signal or a resonator can be used as the clock source. If a single-phase clock source is used, only the connection to OSCIN2 is required; OSCOUT2 should be left unconnected or loaded with no more then 5 pF of stray capacitance.

When operation with a crystal is desired, special care should be taken to minimize stray capacitances and inductance. The resonator, as well as the external components, should be placed in close proximity to OSCIN2 and OSCOUT2 pins to keep the printed circuit trace lengths to an absolute minimum. *Figure 3-10* shows the external crystal interconnections. Table 3-3 provides the crystal characteristics and the values of R, and C components, including stray capacitance.

3.0 Functional Description (Continued)

TABLE 3-2. High-Frequency Oscillator Circuit

Component	Value	Tolerance	Units
XTAL	Resonance 40.96 Third Overtone (parallel) Type AT-Cut Maximum Series Resistance 50 Maximum Shunt Capacitance 7		MHz Ω pF
R1	150k	10%	Ω
R2	51	5%	Ω
C1	20	10%	pF
C2	20	10%	pF
C3	1000	20%	pF
L	1.8	10%	μH

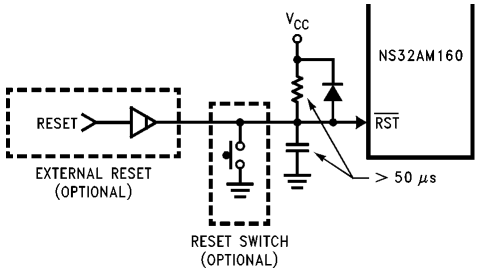


TL/EE/11732–20

FIGURE 3-10. Low Frequency Resonator Connections

TABLE 3-3. Low Frequency Oscillator Circuit

Component	Value	Tolerance	Units
RESONATOR	Ceramic Resonator 455		kHz
R1	1M	10%	Ω
R2	4.7k	10%	Ω
C1	100	20%	pF
C2	100	20%	pF



TL/EE/11732–21

FIGURE 3-11. Recommended Reset Connections

3.6.3 Power Down Mode

The Clock Generator Control register (CLKCTL) has two control bits: PDM and DHFO. The DHFO controls the high-frequency oscillator. When ''0'', the high-frequency oscillator is operating. When CLKCTL.DHFO is ''1'', the high-frequency oscillator is disabled. The PDM bit changes the mode of operation. When CLKCTL.PDM is ''0'', the processor is in normal operation mode, where all the modules operate from the high-frequency oscillator. When CLKCTL.PDM is ''1'', the NS32AM162 is in down power mode, where some of the modules are not operating, and others operate from the low-frequency oscillator. In the power down mode, DRAM refresh cycles are done at a rate of ¼ of Crystal-2 frequency, and the core operates from a clock whose frequency is ⅛ of Crystal-2. Accesses to the following modules are not allowed during low power mode:

- ICU
- CODEC
- PWM generator
- DRAM read and write cycles.

When changing from normal operation mode to power down mode, the user must set CLKCNTL.PDM to ''1'', and only then set CLKCNTL.DHFO to ''1''. When changing from power down mode to normal operation mode, the user must clear CLKCNTL.DHFO to ''0'', and only then clear CLKCNTL.PDM.

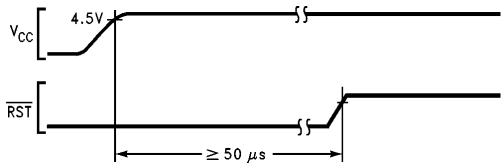
The transition between normal operation mode and power down mode occurs after a new value is written into CLKCTL.PDM. The NS32AM162 may delay this transition, if a DRAM refresh cycle is in process. The CLKCTL.PDM bit will change its value only when the transition is done. Note however that it is usually not needed to wait until the transition is done, since it is guaranteed that the processor will change its mode when the DRAM refresh cycle is over.

3.6.4 Resetting

The $\overline{RST}$ input pin is used to reset the NS32AM162. The CPU samples $\overline{RST}$ on the falling edge of CTTL.

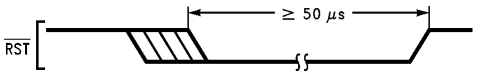
Whenever a low level is detected, the CPU responds immediately. Any instruction being executed is terminated; any results that have not yet been written to memory are discarded; and any pending interrupts and traps are eliminated. The internal latch for the edge-sensitive $\overline{NMI}$ signal is cleared.

On application of power, $\overline{RST}$ 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 50 μs. See *Figures 3-12* and *3-13*.



TL/EE/11732–22

FIGURE 3-12. Power-On Reset Requirements



TL/EE/11732–23

FIGURE 3-13. General Reset Timing

While in the Reset state, the CPU drives the signals CRD, CWR and CFS inactive.

The internal CPU clock and CTTL run at half the frequency of the signal on the OSCIN1 pin. CCLK is active (high).

The PSR is reset to 0. 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 $\overline{NMI}$ is enabled. Maskable interrupts are disabled.

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.

4.0 Device Specifications

4.1 NS32AM162 PIN DESCRIPTIONS

The following is a brief description of all NS32AM162 pins.

4.1.2 Input Signals

$\overline{RST}$ — **Reset Input.** Schmitt triggered, asynchronous signal used to generate a CPU reset.

$\overline{INT3}$ — **External Interrupt.** Schmitt triggered. A High-to-Low transition requests a maskable interrupt.

OSCIN1 — **Crystal1/External Clock Input** (40.96 MHz). Input from a crystal or an external clock source.

OSCIN2 — **Crystal2/External Clock Input** (455 KHz). Input from a crystal or an external clock source.

CDIN — **Data In from CODEC.** Data is input from CODEC via this pin.

Note: After reset this pin is configured as an output, until the MCFG register is set to the appropriate value.

4.1.3 Output Signals

A1–A11 — **Address Bus.** These are the 11 least significant bits of the memory address bus. During DRAM accesses these are the row and column address bits.

$\overline{RAS}$ — **Row Address Strobe** for DRAM Control and Refresh. During DRAM accesses controls DRAM's row address latches; signals the beginning of a DRAM bus cycle. Activated also during DRAM refresh cycles.

$\overline{CAS}$ — **Column Address Strobe** for DRAM Control and Refresh. During DRAM accesses controls DRAM's column address latches. Activated also during DRAM refresh cycles.

$\overline{DWE}$ — **DRAM Write/Read Control.** Activated during DRAM write bus cycles. Enables writing data to the DRAM.

CFS0 — **CODEC0 Frame Sync.** Starts a new encode and decode cycle.

CDOUT — **Data Out to CODEC.** Data is output to the CODEC via this pin.

CCLK — **CODEC Master Clock—** CODEC's Clock input for the switched-capacitor filters and CODEC.

PWM/CFS1 — **PWM Generator Output/CODEC1 Frame Sync.** When one CODEC is used—Pulse Width Modulator output signal. This signal has a fixed frequency and a variable duty cycle.

When two CODECs are used—CODEC1's Frame Sync input. Starts a new encode and decode cycle.

OSCOUT1 — **Crystal1 Clock Output** (40.96 MHz). This line is used as the return path for the high frequency crystal. When an external clock source is used, OSCOUT1 should be left unconnected or loaded with no more than 5 pF of stray capacitance.

OSCOUT2 — **Crystal2 Clock Output** (455 KHz). This line is used as the return path for the low frequency crystal. When an external clock source is used, OSCOUT2 should be left unconnected or loaded with no more than 5 pF of stray capacitance.

PC0/A12 — **Output Port/External ROM Address Line A12.** Output Port, bit 0 in Internal ROM mode, A12 in External ROM and Development modes.

PC1/A13 — **Output Port/External ROM Address Line A13.** Output Port, bit 1 in Internal ROM mode, A13 in External ROM and Development modes.

PC2/A14 — **Output Port/External ROM Address Line A14.** Output Port, bit 2 in Internal ROM mode, A14 in External ROM and Development modes.

PC3/A15 — **Output Port/External ROM Address Line A15.** Output Port, bit 3 in Internal ROM mode, A15 in External ROM and Development modes.

PC4/A16 — **Output Port/External ROM Address Line A16.** Output Port, bit 4 in Internal ROM mode, A16 in External ROM and Development modes.

PC5/$\overline{MRD}$ — **Output Port/External ROM $\overline{OE}$ Signal.** Output Port, bit 5 in Internal ROM mode, external memory Output Enable control in External ROM and Development modes.

4.1.4 Input/Output Signals

D0–D1 — **Data Bus Bits 0 and 1.** Data bit 0 is the l.s.b.

D2/RA12 — **Data Bus Bit 2/DRAM Row Address Line A12.** Data bit 2. Row Address Line 12 in Internal ROM mode. Address line 12 is asserted valid during DRAM accesses when $\overline{RAS}$ is activated.

D3–D7 — **Data Bus Bits 3 to 7.**

PA0/$\overline{MWR0}$ — **Port A, Bit Programmable/External RAM $\overline{WE}$/Signal.** Port A, bit 0 in Internal and External ROM modes, $\overline{WE}$ signal in Development mode, activated during external memory write cycles in order to enable writing of data to the memory's even bytes.

PA1/$\overline{MWR1}$ — **Port A, Bit Programmable/External RAM $\overline{WE}$ Signal.** Port A, bit 1 in Internal and External ROM modes, $\overline{WE}$ signal in Development mode, activated during external memory write cycles in order to enable writing of data to the memory's odd bytes.

PA2/CTTL — **Port A, Bit Programmable/CPU Clock.** Port A, bit 2 in Internal and External ROM modes, CTTL clock in Development mode, this clock is similar to internal PHI1. The skew between CCTL rising edge and PHI1 rising edge is kept to a minimum.

4.0 Device Specifications (Continued)

PA3/$\overline{NSF}$ — **Port A, Bit Programmable/Non-Sequential Fetch Status.** Port A, bit 3 in Internal and External ROM modes, $\overline{NSF}$ in Development mode. $\overline{NSF}$ is a status signal activated during Non-Sequential Instruction Fetches (meaningful if T1 is also activated).

PA4/T1 — **Port A, Bit Programmable/T1** (First bus transaction's cycle). Port A, bit 4 in Internal and External ROM modes, T1 in Development mode. T1 is activated at the beginning of any core or DSPM bus transaction.

PA5/$\overline{DDIN}$ — **Port A, Bit Programmable/Data Direction.** Port A, bit 5 in Internal and External ROM modes, $\overline{DDIN}$ in Development mode. $\overline{DDIN}$ is a status signal indicating the direction of the data transfer during a bus cycle.

PA6/A17 — **Port A, Bit Programmable/External Address Line A17.** Port A, bit 6 in Internal and External ROM modes, address bit 17 in Development mode.

PA7/A18 — **Port A, Bit Programmable/External Address Line A18.** Port A, bit 7 in Internal and External ROM modes, address bit 18 in Development mode.

PB0–PB7/ D8–D15 — **Port B, Bit Programmable/Extended Data Bus Bit 8 through 15.** Port B bits 0 to 7 in Internal ROM mode, Data odd byte in External ROM and Development modes.

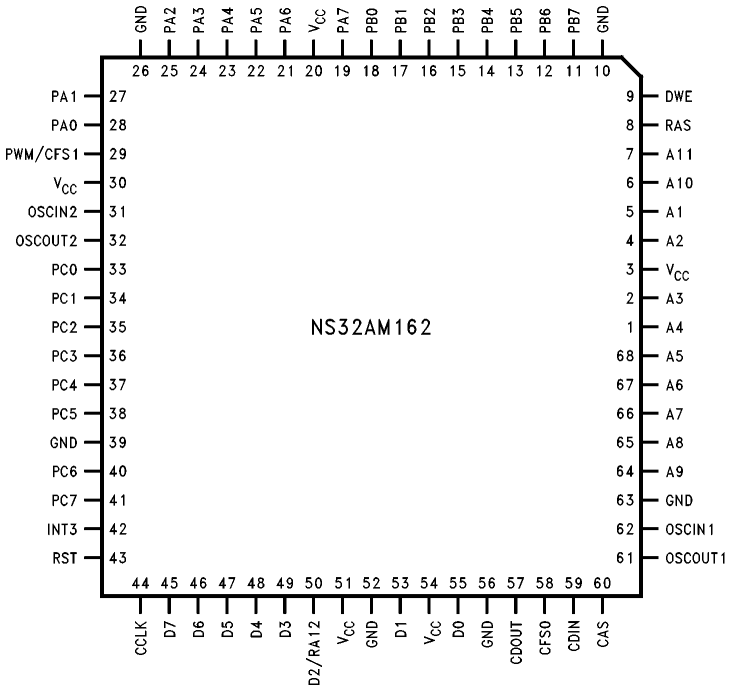
PC6/$\overline{IOWR}$/ MODE0 — **Output Port/External IO Write Control/Mode Control.** Output Port, bit 6 in internal ROM mode, external IO write control in External ROM and Development modes.

Synchronize bit 0, sampled upon reset to determine the mode of operation.

PC7/$\overline{IORD}$/ MODE1 — **Output Port/External IO Read Control/Mode Control.** Output Port, bit 7 in Internal ROM mode, external IO read control in External ROM and Development modes.

Synchronize bit 1, sampled upon reset to determine the mode of operation.

68-Pin PCC Package



TL/EE/11732–24

Top View
Order Number NS32AM162V-20 or NS32AM163V-20
NS Package Number V68A

FIGURE 4-1. Connection Diagram

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.

Storage Temperature	−65°C to +150°C
Temperature under Bias	0°C to +70°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}	Logical 1 Input Voltage		2.0		V_{CC} + 0.5	V
V_{IL}	Logical 0 Input Voltage		−0.5		0.8	V
V_{OH}	Logical 1 Output Voltage	I_{OH} = −400 μA	2.4			V
VPWMH	PWM Logical 1 Voltage (Note 1)	I_{OH} = −400 μA	V_{CC} − 0.5		V_{CC} + 0.5	V
V_{OL}	Logical 0 Output Voltage	I_{OL} = 4 mA			0.45	V
VPWML	PWM Logical 0 Voltage (Note 1)	I_{OL} = 400 μA	−0.5		+0.5	V
VX1H VX2H	OSCIN1/OSCIN2 Input High Voltage (Note 2)		4.2			V
VX1L VX2L	OSCIN1/OSCIN2 Input Low Voltage (Note 2)				1.0	V
I_L	Input Load Current	0V ≤ V_{IN} ≤ V_{CC}	−20		20	μA
I_O (Off)	Output Leakage Current (I/O Pins in Input Mode)	0V ≤ V_{OUT} ≤ V_{CC}	−20		20	μA
I_{CCH}	Active Supply Current (High Power Mode) OSCIN1 = 40.96 MHz	I_{OUT} = 0, T_A = 25°C V_{CC} = 5V			200	mA
I_{CCL}	Active Supply Current (Low Power Mode) OSCIN2 = 455 KHz	I_{OUT} = 0, T_A = 25°C V_{CC} = 5V			2.5	mA
VHYS	Hysteresis Loop Width (Note 1)		0.5			V
VHh	High Level Input Voltage		Max (3.5, V_{CC} − 1.5)			V
VHl	Low Level Input Voltage				0.7	V
VMODh	MODE0 and MODE1 High Level Input Voltage		Max (3.5, V_{CC} − 1.5)			V
VMOD1	MODE0 and MODE1 Low Level Input Voltage				0.7	V

Note 1: Guaranteed by design.

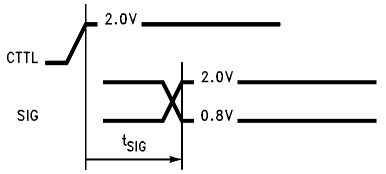
Note 2: When an external single-phase clock signal is used, the Min value of VX1H, VX2H is 4.5V, and the Max value of VX1L, VX2L is 0.5V.

4.0 Device Specifications (Continued)

4.4 SWITCHING CHARACTERISTICS

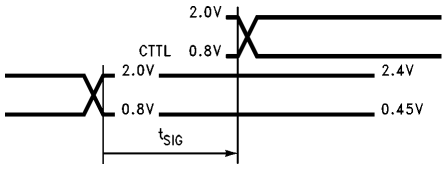
4.4.1 Definitions

All the timing specification 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, 4-3* and *4-4* unless specifically stated otherwise. CTTL and all other output signals capacitive load is assumed to be 50 pF. OSCIN1 crystal frequency is 40.96 MHz. OSCIN2 ceramic resonator frequency is 455 KHz.



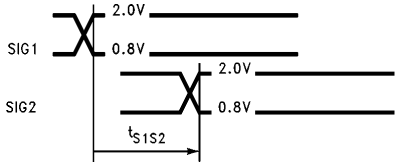
TL/EE/11732–25

FIGURE 4-2. Synchronous Output Signals Specification



TL/EE/11732–26

FIGURE 4-3. Synchronous Input Signals Specification



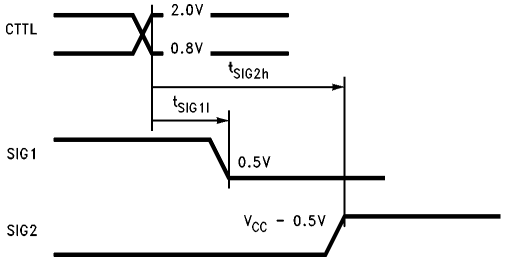
TL/EE/11732–27

FIGURE 4-4. Asynchronous Signals Specification

Abbreviations:

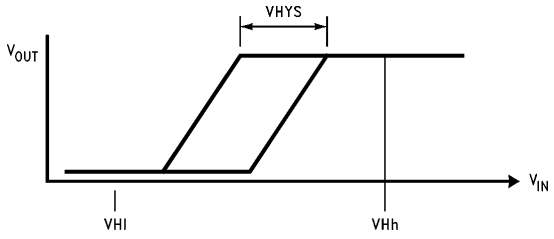
L.E. - Leading Edge T.E. - Trailing Edge

R.E. - Rising Edge F.E. - Falling Edge



TL/EE/11732–28

FIGURE 4-4a. PWM Output Signal Specification



TL/EE/11732–29

FIGURE 4-4b. Hysteresis Inputs Definition

4.0 Device Specifications (Continued)

4.4.2 Synchronous Timing Tables

4.4.2.1 Output Signals: Internal Propagation Delays, NS32AM162-20

Symbol	Figure	Description	Reference Conditions	NS32AM162		Units
				Min	Max	
tCTp	*4-15*	CTTL Clock Period (Note 1)	R.E. CTTL to Next R.E. CTTL	48.8	17582.0	ns
tCTh	*4-15*	CTTL High Time	At 2.0V (Both Edges)	tCTp/2 − 5		ns
tCT1	*4-15*	CTTL Low Time	At 0.8V (Both Edges)	tCTp/2 − 5		ns
tCCLKa	*4-10*	CCLK Active	After R.E. CTTL		13.0	ns
tCCLKia	*4-10*	CCLK Inactive	After R.E. CTTL		13.0	ns
tCFSa	*4-8*	CFS0, CFS1/Active	After R.E. CTTL		25.0	ns
tCFSia	*4-8*	CFS0, CFS1/Inactive	After R.E. CTTL		25.0	ns
tAv	*4-5a*	Address Valid (Note 5)	After R.E. CTTL T1 or T3		12.0	ns
tDv	*4-5c*	D(0:15) Valid	After R.E. CTTL T2		13.0	ns
tDf	*4-5c*	D(0:15) Float (Note 4)	After R.E. CTTL T1		13.0	ns
tCDOv	*4-8* *4-9*	CDOUT Valid	After R.E. CTTL		13.0	ns
tCDOh	*4-10*	CDOUT Hold	After R.E. CTTL	0.0		ns
tDDINv	*4-11a*	$\overline{\text{DDIN}}$ Valid	After R.E. CTTL T1		13.0	ns
tT1a	*4-11a*	T1 Active	After R.E. CTTL T1		13.0	ns
tT1ia	*4-11a*	T1 Inactive	After R.E. CTTL T2		13.0	ns
tNSFa	*4-11a*	$\overline{\text{NSF}}$ Active	After R.E. CTTL T4		13.0	ns
tNSFia	*4-11a*	$\overline{\text{NSF}}$ Inactive	After R.E. CTTL T4		13.0	ns
tRASa	*4-5a*	RAS Active (Note 2)	After R.E. CTTL T1 or T3RF	tCTp/2 − 6	tCTp/2 + 16	ns
tRASia	*4-5a*	Ras Inactive (Note 4)	After R.E. CTTL T4 or T4RF	tCTp/2 − 6	tCTp/2 + 16	ns
tCASa	*4-5a*	CAS Active (Note 2)	After R.E. CTTL T3 or T1RF	tCTp/2 − 6	tCTp/2 + 16	ns
tCASia	*4-5a*	CAS Inactive (Note 4)	After R.E. CTTL T4 or T4RF	tCTp/2 − 6	tCTp/2 + 16	ns
tDWEa	*4-5c*	DRAM Write Enable Active	After R.E. CTTL T2		13.0	ns
tDWEia	*4-5c*	DRAM Write Enable Inactive	After R.E. CTTL T4		13.0	ns
tMRDa	*4-11a*	$\overline{\text{MRD}}$ Active	After R.E. CTTL T2		13.0	ns
tMRDia	*4-11a*	$\overline{\text{MRD}}$ Inactive	After R.E. CTTL T4		13.0	ns
tIORDa	*4-11b*	$\overline{\text{IORD}}$ Active	After R.E. CTTL T2		13.0	ns
tIORDia	*4-11b*	$\overline{\text{IORD}}$ Inactive	After R.E. CTTL T4		13.0	ns
tMWRa	*4-12a*	$\overline{\text{MWR}}$ Active	After R.E. CTTL T2		13.0	ns
tMWRia	*4-12a*	$\overline{\text{MWR}}$ Inactive	After R.E. CTTL T4		13.0	ns
tIOWRa	*4-12b*	$\overline{\text{IOWR}}$ Active	After R.E. CTTL T2		13.0	ns
tIOWRia	*4-12b*	$\overline{\text{IOWR}}$ Inactive	After R.E. CTTL T4		13.0	ns
tPABCv	*4-13a*	PA, PB and PC Valid	After R.E. CTTL T2		13.0	ns

4.0 Device Specifications (Continued)

4.4.2 Synchronous Timing Tables (Continued)

4.4.2.1 Output Signals: Internal Propagation Delays, NS32AM162-20 (Continued)

Symbol	Figure	Description	Reference Conditions	NS32AM162		Units
				Min	Max	
tPWMv	*4-13b*	PWM Valid	After R.E. CTTL		25.0	ns
tRASLa	*4-7*	DRAM L.P. $\overline{RAS}$ Active	After R.E. OSCIN2		tX2p − 0.2	μs
tRASLia	*4-7*	DRAM L.P. $\overline{RAS}$ Inactive	After R.E. OSCIN2		tX2p − 0.2	μs
tCASLa	*4-7*	DRAM L.P. $\overline{CAS}$ Active	After R.E. OSCIN2		tX2p − 0.2	μs
tCASLia	*4-7*	DRAM L.P. $\overline{CAS}$ Inactive	After R.E. OSCIN2		tX2p − 0.2	μs

Note 1: tCTp can be only 48.8 ns (normal operation) or 17582 ns (power down mode).

4.4.2.2 Input Signals

Symbol	Figure	Description	Reference Conditions	NS32AM162		Units
				Min	Max	
tX1p	*4-15*	OSCIN1 Clock Period	R.E. OSCIN1	24.4		ns
tX1h	*4-15*	OSCIN1 High (External Clock)	At 4.2V (Both Edges)	tX1p/2 − 5		ns
tX1l	*4-15*	OSCIN1 Low (External Clock)	At 1.0V (Both Edges)	tX1p/2 − 5		ns
tX2p	*4-15*	OSCIN2 Clock Period	R.E. OSCIN2		2.2	μs
tX2h	*4-15*	OSCIN2 High (External Clock)	At 4.2V (Both Edges)	0.8		μs
tX2l	*4-15*	OSCIN2 Low (External Clock)	At 1.0V (Both Edges)	0.8		μs
tDIs	*4-5a*	Data In Setup	Before R.E. CTTL T4	11.0		ns
tDIh	*4-5a*	Data In Hold (Note 3)	After R.E. CTTL T4	2.0		ns
tCDIs	*4-8* *4-9*	CDIN Setup	Before R.E. CTTL	11.0		ns
tCDIh	*4-8* *4-9*	CDIN Hold	After R.E. CTTL	2.0		ns
tPABs	*4-14*	PA and PB Data in Setup	Before R.E. CTTL T4	11.0		ns
tPABh	*4-14*	PA and PB in Hold	After R.E. CTTL T4	2.0		ns
tRSTw	*4-16*	$\overline{RST}$ Pulse Width	At 0.8V (Both Edges)	50		μs
tPWR	*4-17*	Power Stable to R.E. of $\overline{RST}$ (Note 4)	After V_{CC} Reaches 4.5V	50		μs

Note 2: Address setup before $\overline{RAS}$, Address setup before $\overline{CAS}$ and Data Setup before $\overline{CA}$ are at least 9 ns, guaranteed by design.

Note 3: tDIh is always less than or equal to tMRDia, tIORDIa and tCRDia, guaranteed by design.

Note 4: Not tested, guaranteed by design.

Note 5: Refers to A(1:18) in Development mode, to A(1:16) in External ROM mode, to A(1:11) in Internal ROM mode.

4.0 Device Specifications (Continued)

4.4.3 TIMING DIAGRAMS

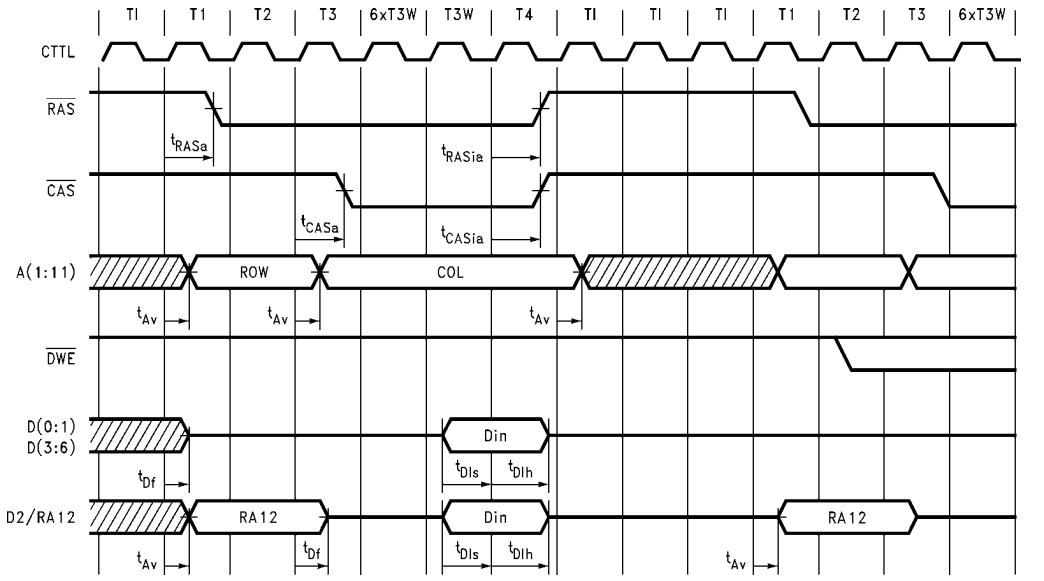


FIGURE 4-5a. DRAM Read Cycle Timing (Internal ROM Mode Only)

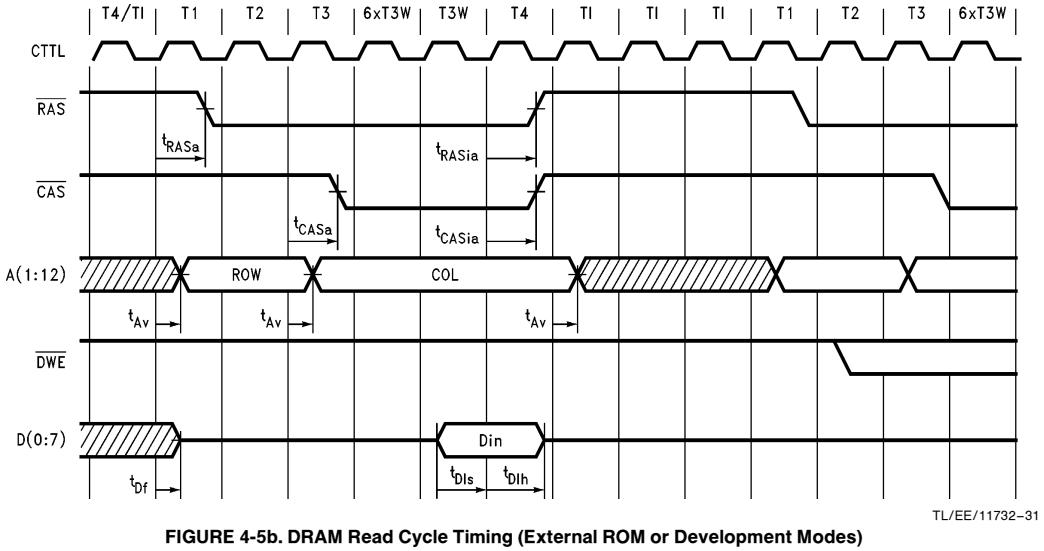
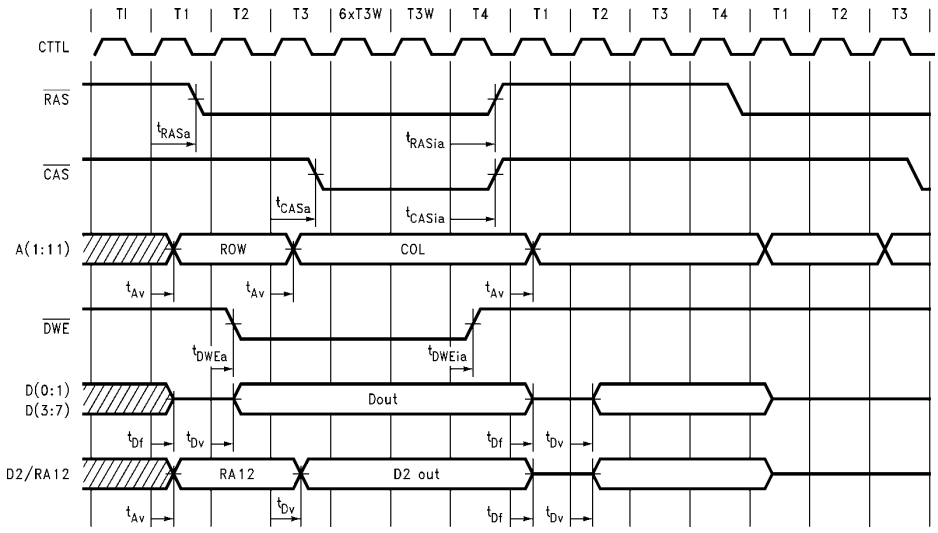
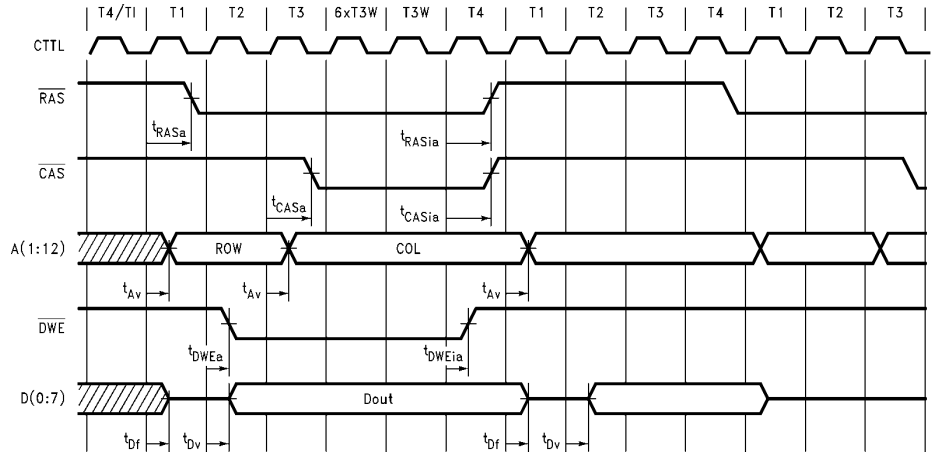


FIGURE 4-5b. DRAM Read Cycle Timing (External ROM or Development Modes)



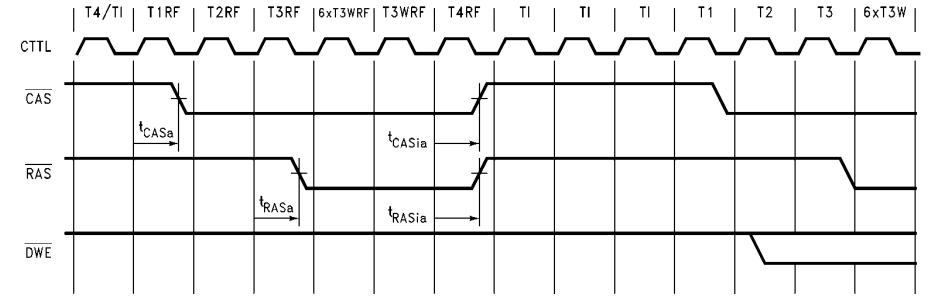
TL/EE/11732–32

FIGURE 4-5c. DRAM Write Cycle Timing (Internal ROM Mode Only)



TL/EE/11732–33

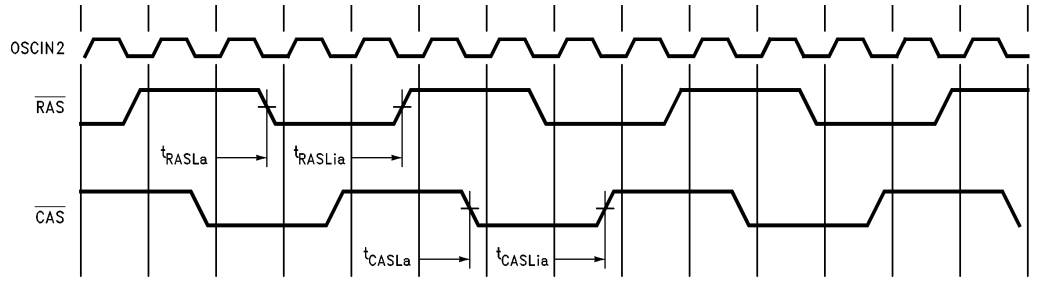
FIGURE 4-5d. DRAM Write Cycle Timing (External ROM or Development Modes)



TL/EE/11732–34

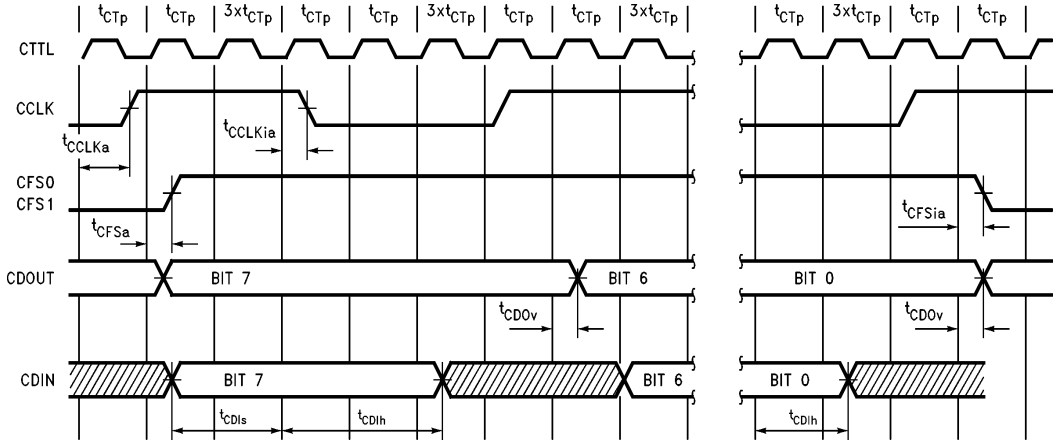
FIGURE 4-6. DRAM Refresh Cycle Timing (In Normal Operation Mode)

4.0 Device Specifications (Continued)



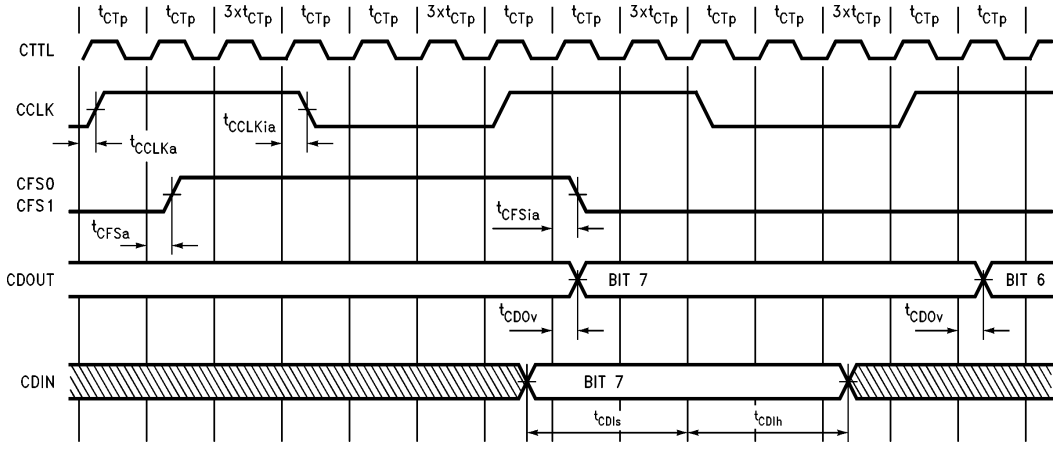
TL/EE/11732–35

FIGURE 4-7. DRAM Power Down Refresh



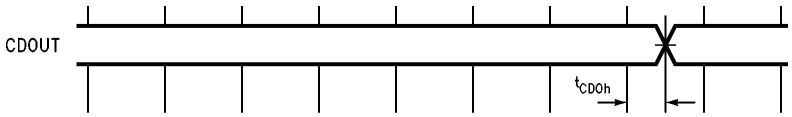
TL/EE/11732–36

FIGURE 4-8. CODEC Long Frame Timing, 8 KHz Sampling Rate



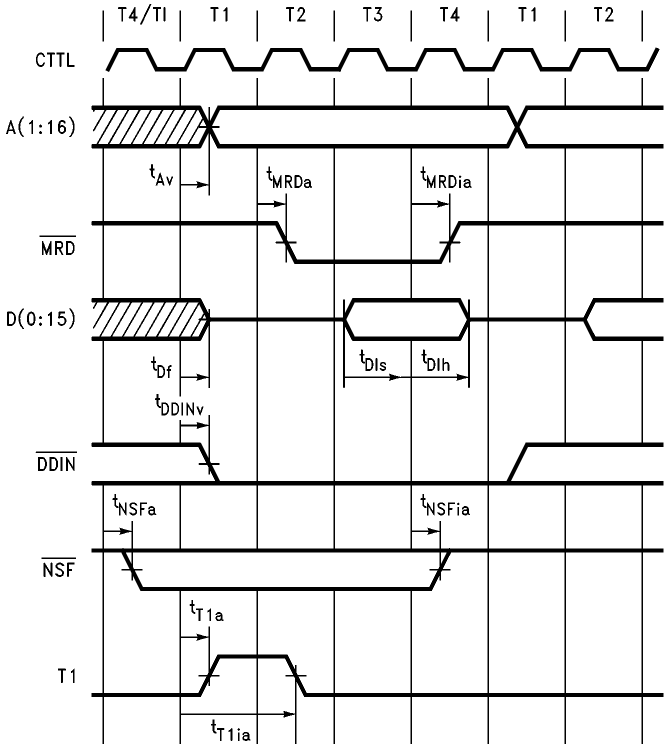
TL/EE/11732–37

FIGURE 4-9. CODEC Short Frame Timing, 8 KHz Sampling Rate



TL/EE/11732–38

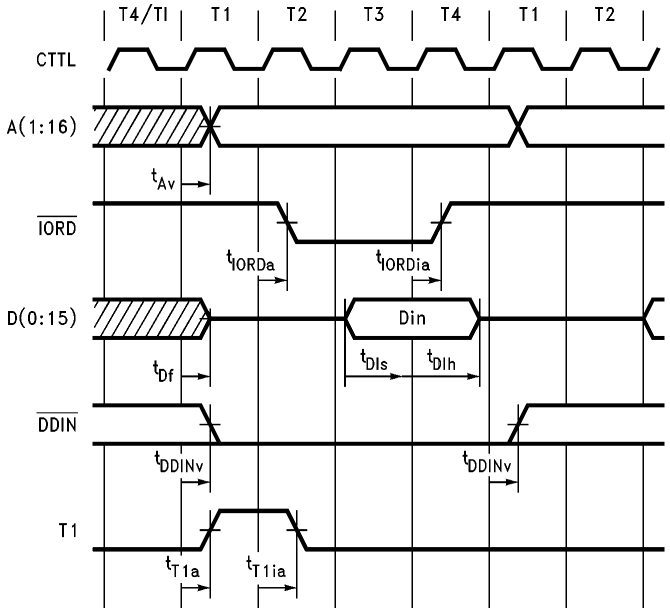
FIGURE 4-10. CDOUT Hold Timing



TL/EE/11732–39

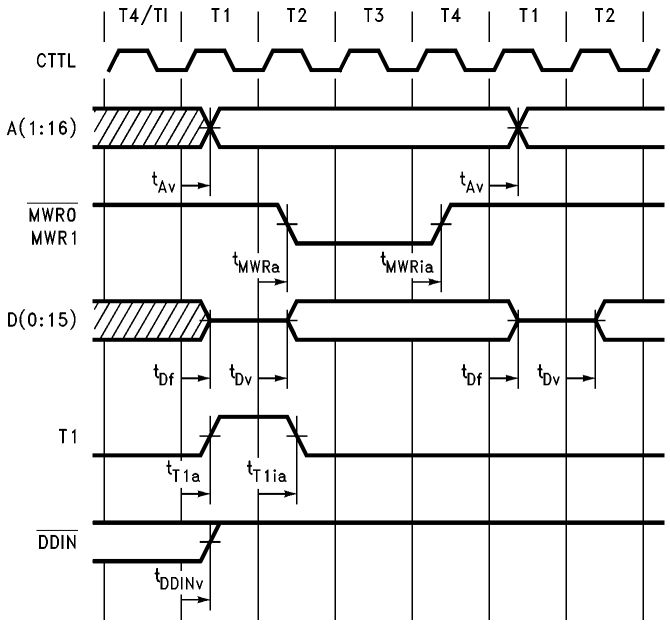
FIGURE 4-11a. External Memory Read Timing

4.0 Device Specifications (Continued)



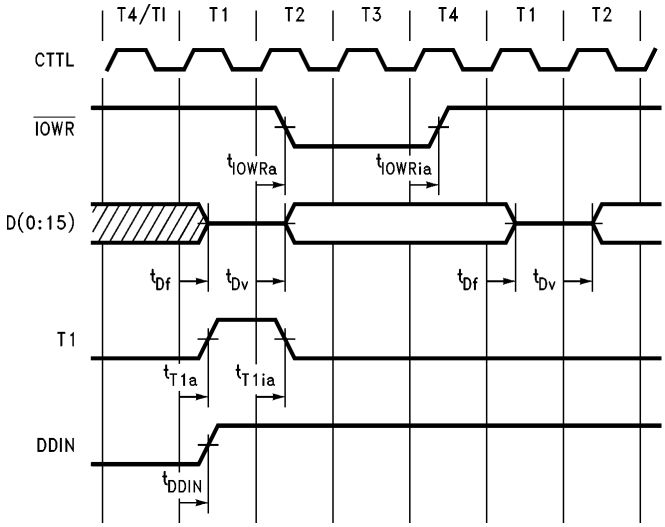
TL/EE/11732–40

FIGURE 4-11b. I/O Read Cycle



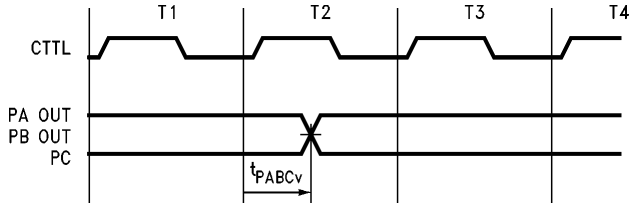
TL/EE/11732–41

FIGURE 4-12a. External Memory Write—Cycle Timing



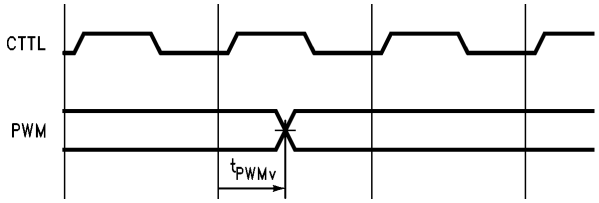
TL/EE/11732–42

FIGURE 4-12b. I/O Write Cycle Timing



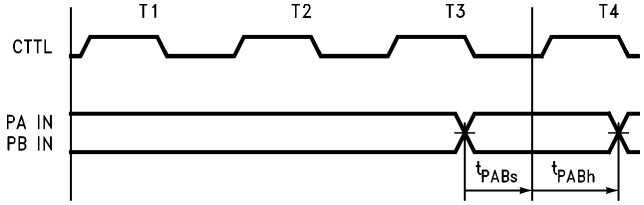
TL/EE/11732–43

FIGURE 4-13a. Port A, Port B and Port C Timing



TL/EE/11732–44

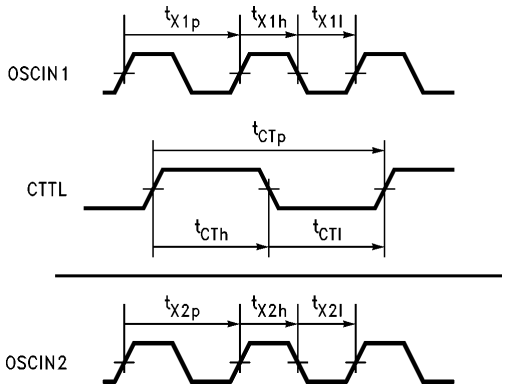
FIGURE 4-13b. PWM Output Timing



TL/EE/11732–45

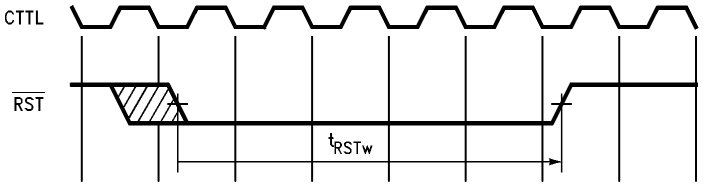
FIGURE 4-14. Port A and Port B Input Timing

4.0 Device Specifications (Continued)



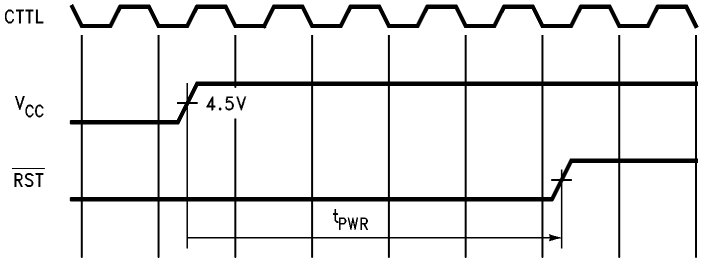
TL/EE/11732–46

FIGURE 4-15. CTTL, OSCIN1 and OSCIN2 Timing



TL/EE/11732–47

FIGURE 4-16. Non Power On Reset



TL/EE/11732–48

FIGURE 4-17. Power On Reset

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.

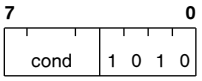
reg = General Purpose Register Number

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

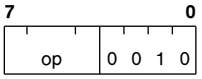
U/W	B	T

T = Translated
B = Backward
U/W = 00: None
01: While Match
11: Until Match



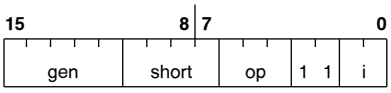
Format 0

Bcond (BR)



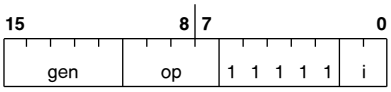
Format 1

BSR	—0000	ENTER	—1000
RET	—0001	EXIT	—1001
RETT	—0100	NOP	—1010
RETI	—0101	WAIT	—1011
SAVE	—0110	DIA	—1100
RESTORE	—0111	FLAG	—1101
		SVC	—1110
		BPT	—1111



Format 2

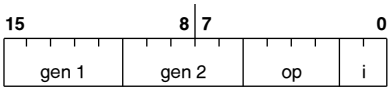
ADDQ	—000	ACB	—100
CMPQ	—001	MOVQ	—101
SPR	—010	LPR	—110
Scond	—011		



Format 3

BICPSR	—0010	ADJSP	—1010
JUMP	—0100	JSR	—1100
BISPSR	—0110	CASE	—1110

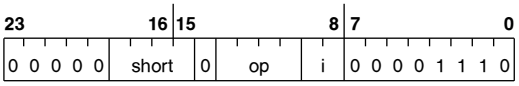
Trap (UND) on XXX1, 1000



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

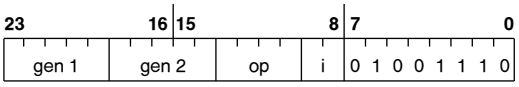
Appendix A: Instruction Formats (Continued)



Format 5

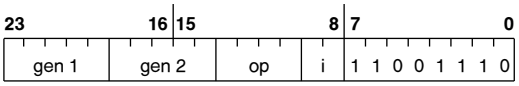
MOVS	−0000	BITWT	−1000
CMPS	−0001	TBITS	−1001
SETCFG	−0010	BBAND	−1010
SKPS	−0011	SBITPS	−1011
BBSTOD	−0100	BBFOR	−1100
BBOR	−0110	SBITS	−1101
MOVMP	−0111	BBXOR	−1110

No Operation on 1111



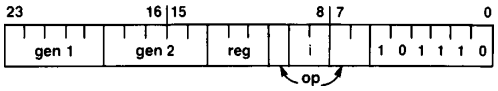
Format 6

ROT	−0000	NOT	−1001
ASH	−0001	Trap (UND)	−1011
CBIT	−0010	SUBP	−1011
Trap (UND)	−0100	ABS	−1100
LSH	−0101	COM	−1101
SBIT	−0110	IBIT	−1110
NEG	−1000	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/11732–49

Format 8

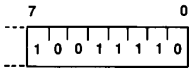
EXT	−0 00	INDEX	−1 00
CVTP	−0 01	FFS	−1 01
INS	−0 10		
CHECK	−0 11		

Trap (UND) on −1 10 and −1 11

TL/EE/11732–50

Format 10

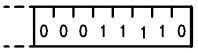
Trap (UND) Always



TL/EE/11732–51

Format 13

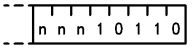
Trap (UND) Always



TL/EE/11732–52

Format 14

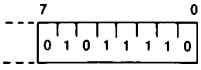
Trap (UND) Always



TL/EE/11732–53

Format 15

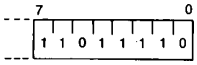
Trap (UND) Always



TL/EE/11732–54

Format 16

Trap (UND) Always

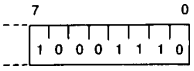


TL/EE/11732–55

Appendix A: Instruction Formats (Continued)

Format 17

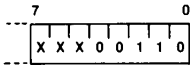
Trap (UND) Always



TL/EE/11732–56

Format 18

Trap (UND) Always

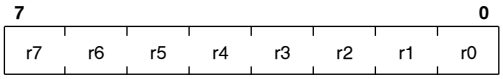


TL/EE/11732–57

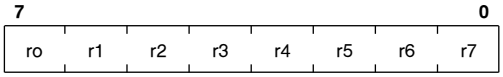
Format 19

Trap (UND) Always

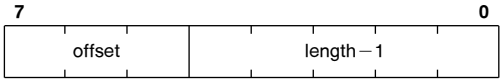
Implied Immediate Encodings:



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 NS32AM162.

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 INSTRUCTIONS

Execution times for basic and floating-point instructions are given in Table B-1. 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.

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	
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

<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 the 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.

Appendix B: Instruction Execution Times (Continued)

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	2	—	—	1	—	15	<xM>
	1	—	—	—	—	1	—	7	<xR>
CHECKi	1	1	—	—	—	3	—	7	high
	1	1	—	—	—	3	—	10	low
	1	1	—	—	—	3	—	11	ok
CMPi	1	1	—	—	—	2	—	3	<xM>
	1	—	—	—	—	1	—	3	<MR>
	—	—	—	—	—	—	—	3	<RR>
CMPMi	1	1	—	—	—	2 * n	—	9 * n + 24	n = # of elements in block
CMPQi	1	—	—	—	—	1	—	3	<M>
	—	—	—	—	—	—	—	3	<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	
DEIi	1	1	—	—	—	5	16	38	<xM>
	1	—	—	—	—	1	16	31	<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	—	19 → 29	field in memory
	1	1	—	—	—	1	—	17 → 51	field in register
EXTSi	1	1	—	—	1	1	—	26 → 36	
FFSi	1	1	2	—	—	1	24	24 → 28	
FLAG	—	—	—	—	—	—	—	6	no trap
	—	—	—	4	3	—	—	44	trap
IBITi	1	1	2	—	—	1	—	17	<xM>
	1	—	—	—	—	—	—	9	<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	—	—	2	1	—	29 → 39	field in memory
	1	—	—	—	—	1	—	28 → 96	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	—	—	—	2	—	1	<xM>
	1	—	—	—	—	1	—	3	<MR>
	—	—	—	—	—	—	—	3	<RR>
MOVMi	1	1	—	—	—	2 * n	—	3 * n + 20	n = # of elements in block
MOVQi	1	—	—	—	—	1	—	2	<M>
	—	—	—	—	—	—	—	3	<R>
MOVSB, W									n = # elements
	—	—	—	—	—	2 * n	—	14 * n + 59	no options
	—	—	—	—	—	2 * n	—	24 * n + 54	B, W and/or U option in effect
MOVSD									n = # of elements
	—	—	—	—	—	2 * n	—	10 * n + 59	no options
	—	—	—	—	—	2 * n	—	24 * n + 54	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	—	—	—	3	—	3	<xM>
	1	—	—	—	—	1	—	4	<MR>
	—	—	—	—	—	—	—	4	<RR>
QUOi	1	1	—	—	—	3	16	49 → 55	

Appendix B: Instruction Execution Times (Continued)

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	
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>
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)

B.2 SPECIAL GRAPHICS INSTRUCTIONS

This section provides the execution times for the special graphics instructions. Table B-2 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-3 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-2 is used.

42 + (107 + 44 * (width − 2)) * height + ((shift − 8) * width * 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 clocks

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-2. 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-2. 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
MOVMPB,W	16 + 7 * R2	
MOVMPD,W	16 + 8 * R2	
SBITS	39 42	R2 ≤ 25 R2 > 25
SBITP	8 + (34 * R2)	

TABLE B-3. 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
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)	

Appendix B: Instruction Execution Times (Continued)

B3. COMMAND LIST OPERATIONS

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

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 + k) * *leng* + 2
VXSTORE	(5 + W + k) * *leng* + 2
VXGATH	(5 + W + k) * *leng* + 2

Where:

w = Number of wait states in external memory access.

k = Number of cycles until HLDA is received, in external memory instructions.

Arithmetic/Logic Instructions

Instruction	Cycles
VROP	3 * *leng* + 2
VAROP	3 * *leng* + 4

Multiply-and-Accumulate Instructions

Instruction	Cycles
VRMAC	2 * *leng* + 7
VARMAC	2 * *leng* + 7
VCMAC	4 * *leng* + 6
VRLATP	4 * *leng* + 5

Multiply-and-Add Instructions

Instruction	Cycles
VAIMAD	6 * *leng* + 2
VRMAD	4 * *leng* + 3
VARMAD	4 * *leng* + 4
VCMAD	4 * *leng* + 6

Clipping and Min/Max Instructions

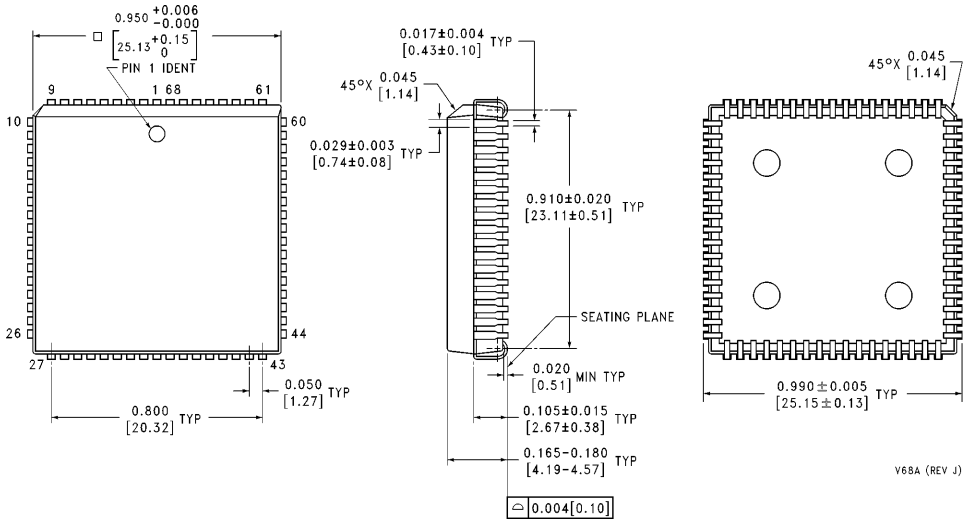
Instruction	Cycles
VARABS	2 * *leng* + 5
VARMIN	7 * *leng* + 2
VARMAX	7 * *leng* + 2
VRFMAX	4 * *leng* + 6
EFMAX	17

Special Instructions

Instruction	Cycles
ESHL	1 * *leng* + 5
VCPOLY	4 * *leng* + 15
VESIIR	16 * *leng* + 6

If *leng* = 0 in ESHL instruction, then the timing is 4 cycles.

Physical Dimensions inches (millimeters)



68-Pin Plastic Leaded Chip Carrier (V)
Order Number NS32AM162V-20 or NS32AM163V-20
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