630069-001

National Semiconductor

NS32CG16 Technical Design Handbook

1988

1988 NS32CG16 Technical Design Handbook National Semiconductor

A Corporate Dedication to Quality and Reliability

National Semiconductor is an industry leader in the manufacture of high quality, high reliability integrated circuits. We have been the leading proponent of driving down IC defects and extending product lifetimes. From raw material through product design, manufacturing and shipping, our quality and reliability is second to none.

We are proud of our success . . . it sets a standard for others to achieve. Yet, our quest for perfection is ongoing so that you, our customer, can continue to rely on National Semiconductor Corporation to produce high quality products for your design systems.

Charles E. Sporck
President, Chief Executive Officer
National Semiconductor Corporation

Wir fühlen uns zu Qualität und Zuverlässigkeit verpflichtet

National Semiconductor Corporation ist führend bei der Herstellung von integrierten Schaltungen hoher Qualität und hoher Zuverlässigkeit. National Semiconductor war schon immer Vorreiter, wenn es galt, die Zahl von IC Ausfällen zu verringern und die Lebensdauern von Produkten zu verbessern. Vom Rohmaterial über Entwurf und Herstellung bis zur Auslieferung, die Qualität und die Zuverlässigkeit der Produkte von National Semiconductor sind unübertroffen.

Wir sind stolz auf unseren Erfolg, der Standards setzt, die für andere erstrebenswert sind. Auch ihre Ansprüche steigen ständig. Sie als unser Kunde können sich auch weiterhin auf National Semiconductor verlassen.

La Qualité et La Fiabilité:

Une Vocation Commune Chez National Semiconductor Corporation

National Semiconductor Corporation est un des leaders industriels qui fabrique des circuits intégrés d'une très grande qualité et d'une fiabilité exceptionelle. National a été le premier à vouloir faire chuter le nombre de circuits intégrés défectueux et a augmenter la durée de vie des produits. Depuis les matières premières, en passant par la conception du produit sa fabrication et son expédition, partout la qualité et la fiabilité chez National sont sans équivalents.

Nous sommes fiers de notre succès et le standard ainsi défini devrait devenir l'objectif à atteindre par les autres sociétés. Et nous continuons à vouloir faire progresser notre recherche de la perfection; il en résulte que vous, qui êtes notre client, pouvez toujours faire confiance à National Semiconductor Corporation, en produisant des systèmes d'une très grande qualité standard.

Un Impegno Societario di Qualità e Affidabilità

National Semiconductor Corporation è un'industria al vertice nella costruzione di circuiti integrati di altà qualità ed affidabilità. National è stata il principale promotore per l'abbattimento della difettosità dei circuiti integrati e per l'allungamento della vita dei prodotti. Dal materiale grezzo attraverso tutte le fasi di progettazione, costruzione e spedizione, la qualità e affidabilità National non è seconda a nessuno.

Noi siamo orgogliosi del nostro successo che fissa per gli altri un traguardo da raggiungere. Il nostro desiderio di perfezione è d'altra parte illimitato e pertanto tu, nostro cliente, puoi continuare ad affidarti a National Semiconductor Corporation per la produzione dei tuoi sistemi con elevati livelli di qualità.

Charles E. Sporck
President, Chief Executive Officer
National Semiconductor Corporation

NS32CG16 TECHNICAL DESIGN HANDBOOK

1988

- Processors
- Peripherals
- Development Tools
- Software
- Application Notes

TRADEMARKS

Following is the most current list of National Semiconductor Corporation's trademarks and registered trademarks.

AbuseableTM
AnadigTM
ANS-R-TRANTM
APPSTM
ASPECTTM
Auto-Chem DeflasherTM
BCPTM
BI-FETTM
BI-FET IITM
BI-LINETM
BIPLANTM
BLCTM
BLXTM
Brite-LiteTM
BTLTM
CheckTrackTM
CIMTM
CIMBUSTM
CLASICTM
Clock✓ChekTM
COMBOTM
COMBO ITM
COMBO IITM
COPSTM microcontrollers
Datachecker®
DENSPAKTM
DIBTM
Digitalker®
DISCERNTM
DISTILLTM
DNR®
DPVMTM
ELSTARTM
E-Z-LINKTM
FACTTM
FAIRCADTM
FairtechTM
FAST®
5-Star ServiceTM
GAL®
GENIXTM
GNXTM
HAMRTM
HandiScanTM
HEX 3000TM
HPCTM
I3L®
ICMTM
INFOCHEXTM
Integral ISETM
IntelisplayTM
ISETM
ISE/06TM
ISE/08TM
ISE/16TM
ISE32TM
ISOPLANARTM
ISOPLANAR-ZTM
KeyScanTM
LMCMOSTM
M2CMOSTM
MacrobusTM
MacrocomponentTM
MAXI-ROM®
Meat✓ChekTM
MenuMasterTM
MicrobusTM data bus
MICRO-DACTM
μtalkerTM
MicrotalkerTM
MICROWIRETM
MICROWIRE/PLUSTM
MOLETM
MSTTM
Naked-8TM
National®
National Semiconductor®
National Semiconductor Corp.®
NAX 800TM
Nitride PlusTM
Nitride Plus OxideTM
NMLTM
NOBUSTM
NSC800TM
NSCISETM
NSX-16TM
NS-XC-16TM
NTERCOMTM
NURAMTM
OXISSTM
P2CMOSTM
PC MasterTM
Perfect WatchTM
Pharma✓ChekTM
PLANTM
PLANARTM
PolycraftTM
POSilinkTM
POSitalkerTM
Power + ControlTM
POWERplanarTM
QUAD3000TM
QUIKLOOKTM
RATTM
RTX16TM
SABRTM
Script✓ChekTM
SCXTM
SERIES/800TM
Series 900TM
Series 3000TM
Series 32000®
Shelf✓ChekTM
SofChekTM
SPIRETM
STARTM
StarlinkTM
STARPLEXTM
SuperChipTM
SuperScriptTM
SYS32TM
TapePak®
TDSTM
TeleGateTM
The National Anthem®
Time✓ChekTM
TINATM
TLCTM
TrapezoidalTM
TRI-CODETM
TRI-POLYTM
TRI-SAFETM
TRI-STATE®
TURBOTRANSCEIVERTM
VIPTM
VR32TM
WATCHDOGTM
XMOSTM
XPUTM
Z STARTM
883B/RETSTM
883S/RETSTM

Postscript™ is a trademark of Adobe Systems Inc.
Laserjet™ and PCL™ are trademarks of Hewlett Packard
UNIX® and DWB® are registered trademarks of AT & T
IBM® is a registered trademark and IBM-PC®, XT® and AT™ are trademarks of International Business Machines Corporation
VAX™, VMS™, DEC™, PDP-11™, RSX-11™ are trademarks of Digital Equipment Corporation
VRTX®, IOX®, and FMX® are registered trademarks of Hunter & Ready Corporation
Opus5™ is a trademark of Opus Systems

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 2900 Semiconductor Drive, P.O. Box 58090, Santa Clara, California 95052-8090 (408) 721-5000
TWX (910) 339-9240

Introduction

Dear Customer,

Introduction of the NS32CG16 marks a major milestone in the continuing evolution of the Series 32000® family of high performance 32-bit microprocessors. With the NS32CG16, your system can be powered with a 32-bit processor optimized for embedded control applications.

The NS32CG16 offers high integration, the performance of a fully programmable 32-bit microprocessor and graphics support—all on one chip. Our endeavor has been to design a microprocessor with the system designer's needs in mind. We hope you will benefit from this effort.

National also offers an array of VLSI solutions for peripheral functions, from DRAM controllers to single-chip SCSI controllers and Ethernet controllers. With this offering we hope to meet all of your VLSI needs.

Richard L. Sanquini
Division Vice President
Micro Systems Group

Table of Contents

Section 1

Processors

National Semiconductor

PRELIMINARY

NS32CG16-10/NS32CG16-15 High-Performance Printer/Display Processor

General Description

The NS32CG16 is a 32-bit microprocessor in the Series 32000® family that provides special features for graphics applications. It is specifically designed to support page oriented printing technologies such as Laser, LCS, LED, Ion-Deposition and InkJet.

The NS32CG16 provides a 16 Mbyte linear address space and a 16-bit external data bus. It also has a 32-bit ALU, an eight-byte prefetch queue, and a slave processor interface.

The capabilities of the NS32CG16 can be expanded by using an external floating point unit which interfaces to the NS32CG16 as a slave processor. This combination provides optimal support for outline character fonts.

The NS32CG16's highly efficient architecture, in addition to the built-in capabilities for supporting BITBLT (BIT-aligned BLock Transfer) operations and other special graphics functions, make the device the ideal choice to handle a variety of page description languages such as Postscript™ and PCL™.

Features

- Software compatible with the Series 32000 family
- 32-bit architecture and implementation
- 16 Mbyte linear address space
- Special support for graphics applications
  - — 18 graphics instructions
  - — Binary compression/expansion capability for font storage using RLL encoding
  - — Pattern magnification for Epson and HP LaserJet™ emulations
  - — 6 BITBLT instructions on chip
  - — Interface to an external BITBLT processing unit for very fast BITBLT operations (optional)
- Floating point support via the NS32081 or the NS32381 for outline fonts, scaling and rotation
- On-chip clock generator
- Optimal interface to large memory arrays via the DP84xx family of DRAM controllers
- Power save mode
- High-speed CMOS technology
- 68-pin plastic PCC package

1.0 Product Introduction

The NS32CG16 is a high speed CMOS microprocessor in the Series 32000 family. It is software compatible with all the 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 NS32CG16 features that are shared with other members of the family are provided below:

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

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

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

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

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

Large, Uniform Addressing. The NS32CG16 has 24-bit address pointers that can address up to 16 megabytes without any segmentation; this addressing scheme provides flexible memory management without added-on expense.

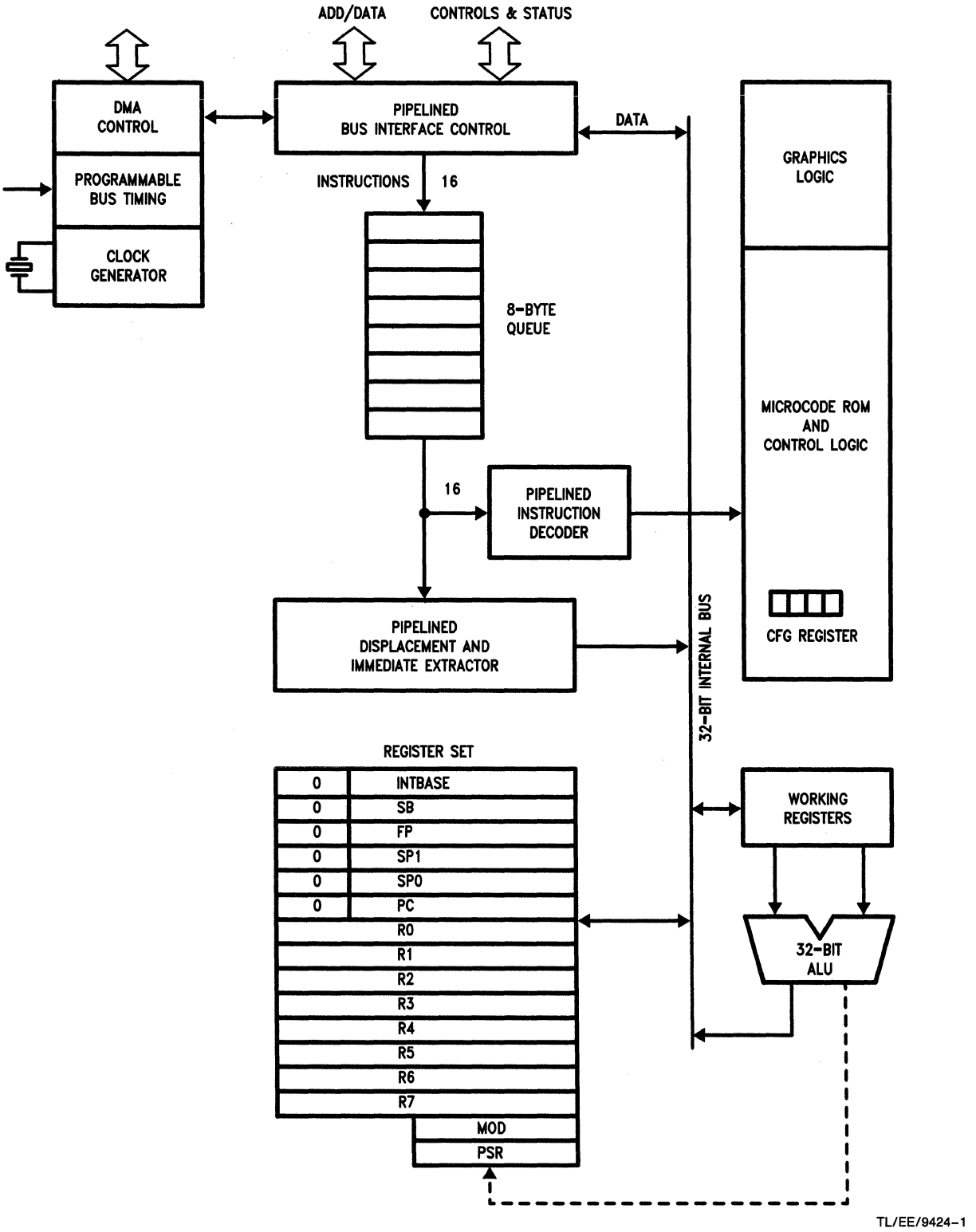
Modular Software Support. Any software package for the Series 32000 family can be developed independent of all other packages, without regard to individual addressing. In addition, ROM code is totally relocatable and easy to access, which allows a significant reduction in hardware and software cost.

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

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

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

Block Diagram



TL/EE/9424-1

Table of Contents

List of Illustrations

List of Illustrations (Continued)

List of Tables

1.0 Product Information (Continued)

1.1 NS32CG16 SPECIAL FEATURES

In addition to the above Series 32000 features, the NS32CG16 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 graphics support capabilities, that can be used in applications such as printers, CRT terminals, and other varieties of display systems, where text and graphics are to be handled.

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

The NS32CG16 allows systems to be built with a relatively small amount of random logic. The bus is highly optimized to allow simple interfacing to a large variety of DRAMs and peripheral devices. All the relevant bus access signals and clock signals are generated on-chip. The cycle extension logic is also incorporated on-chip.

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

The bus characteristics and the power save feature 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 2.4. Details on all the NS32CG16 instructions can be found in the NS32CG16 Printer/Display Processor Programmer's Reference Supplement and the related NS32CG16 supplement.

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

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

2.0 Architectural Description

2.1 REGISTER SET

The NS32CG16 CPU has 17 internal registers grouped according to functions as follows: 8 general purpose, 7 address, 1 processor status and 1 configuration. *Figure 2-1* shows the NS32CG16 internal registers.

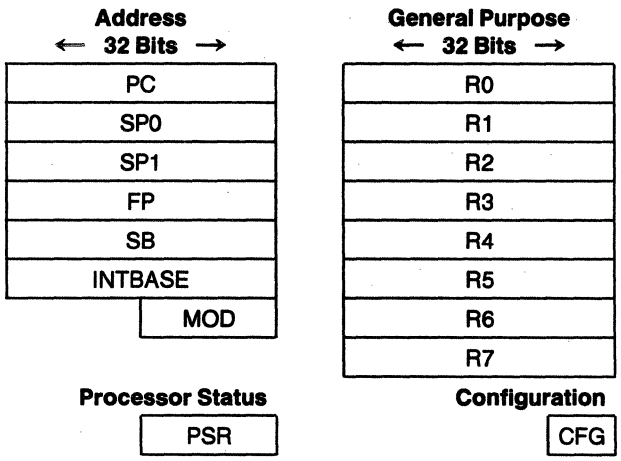


FIGURE 2-1. NS32CG16 Internal Registers

2.1.1 General Purpose Registers

There are eight registers (R0–R7) used for satisfying the high speed general storage requirements, such as holding temporary variables and addresses. The general purpose registers are free for any use by the programmer. They are 32 bits in length. If a general purpose register is specified for

2.0 Architectural Description (Continued)

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. In the NS32CG16 only the lower 24 bits are implemented in the six 32-bit address registers. The top 8 bits are always zero. 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 family grow downward in memory. A Push operation pre-decrements the Stack Pointer by the operand length. A Pop operation post-increments the Stack Pointer by the operand length.

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

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

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

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

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

2.1.3 Processor Status Register

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

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

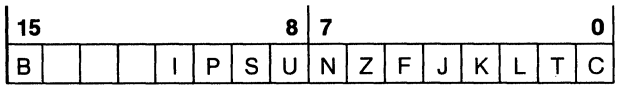


FIGURE 2-2. Processor Status Register (PSR)

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)

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

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

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

2.1.4 Configuration Register

The Configuration Register (CFG) is 8 bits wide, of which four bits are implemented. The implemented bits are used to declare the presence of certain external devices and to select the clock scaling factor. CFG is programmed by the SETCFG instruction. The format of CFG is shown in *Figure 2-3*. The various control bits are described below.

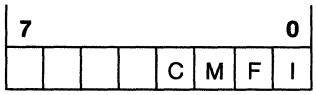


FIGURE 2-3. Configuration Register (CFG)

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

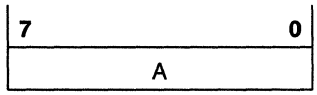
F Floating-point instruction set. This bit indicates whether a floating-point unit (FPU) is present to execute floating-point instructions. If this bit is 0 when the CPU executes a floating-point instruction, a Trap (UND) occurs. If this bit is 1, then the CPU transfers the instruction and any necessary operands to the FPU using the slave-processor protocol described in Section 3.1.4.1.

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

C Clock scaling. Same as the M bit above. Refer to Section 3.2.1 on "Power Save Mode" for details.

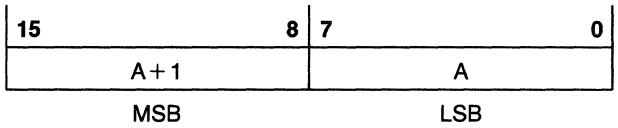
2.2 MEMORY ORGANIZATION

The main memory of the NS32CG16 is a uniform linear address space. Memory locations are numbered sequentially starting at zero and ending at $2^{24}-1$. The number specifying a memory location is called an address. The contents of each memory location is a byte consisting of eight bits. Unless otherwise noted, diagrams in this document show data stored in memory with the lowest address on the right and the highest address on the left. Also, when data is shown vertically, the lowest address is at the top of a diagram and the highest address at the bottom of the diagram. When bits are numbered in a diagram, the least significant bit is given the number zero, and is shown at the right of the diagram. Bits are numbered in increasing significance and toward the left.



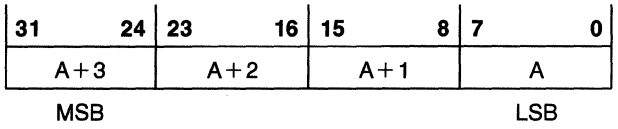
Byte at Address A

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

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.

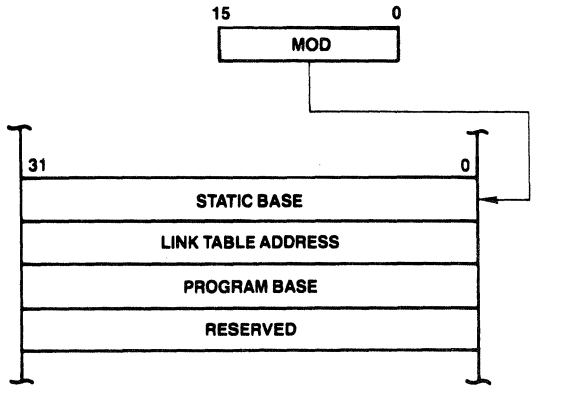
2.2.1 Dedicated Tables

Two of the NS32CG16 dedicated registers (MOD and INTBASE) serve as pointers to dedicated tables in memory.

The INTBASE register points to the Interrupt Dispatch and Cascade tables. These are described in Section 3.8.

The MOD register contains a pointer into the Module Table, whose entries are called Module Descriptors. A Module Descriptor contains four pointers, three of which are used by the NS32CG16. The MOD register contains the address of the Module Descriptor for the currently running module. It is automatically updated by the Call External Procedure instructions (CXP and CXPD).

The format of a Module Descriptor is shown in *Figure 2-4*. The Static Base entry contains the address of static data assigned to the running module. It is loaded into the CPU Static Base register by the CXP and CXPD instructions. The Program Base entry contains the address of the first byte of instruction code in the module. Since a module may have multiple entry points, the Program Base pointer serves only as a reference to find them.



TL/EE/9424–2

FIGURE 2-4. Module Descriptor Format

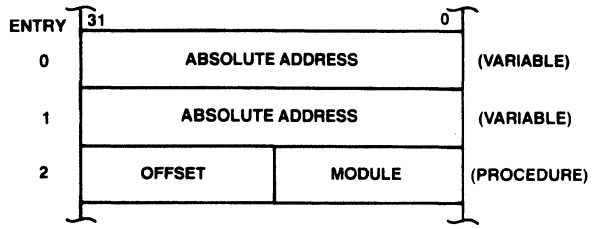
2.0 Architectural Description (Continued)

The Link Table Address points to the Link Table for the currently running module. The Link Table provides the information needed for:

1) Sharing variables between modules. Such variables are accessed through the Link Table via the External addressing mode.
2) Transferring control from one module to another. This is done via the Call External Procedure (CXP) instruction.

The format of a Link Table is given in *Figure 2-5*. A Link Table Entry for an external variable contains the 32-bit address of that variable. An entry for an external procedure contains two 16-bit fields: Module and Offset. The Module field contains the new MOD register contents for the module being entered. The Offset field is an unsigned number giving the position of the entry point relative to the new module's Program Base pointer.

For further details of the functions of these tables, see the Series 32000 Instruction Set Reference Manual.



TL/EE/9424-3

FIGURE 2-5. A Sample Link Table

2.3 INSTRUCTION SET

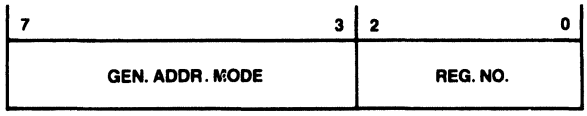
2.3.1 General Instruction Format

Figure 2-6 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 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. See *Figure 2-7*.

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 2-8*, 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/9424-5

FIGURE 2-7. Index Byte Format

2.3.2 Addressing Modes

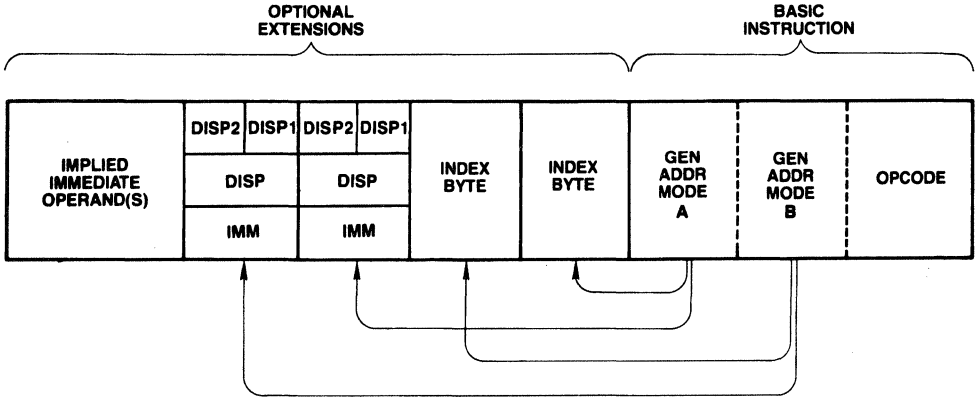
The NS32CG16 CPU generally accesses an operand by calculating its Effective Address based on information available when the operand is to be accessed. The method to be used in performing this calculation is specified by the programmer as an "addressing mode."

Addressing modes in the NS32CG16 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.

NS32CG16 Addressing Modes fall into nine basic types:

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

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



TL/EE/9424-4

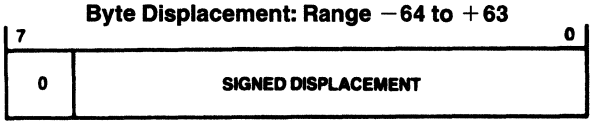
FIGURE 2-6. General Instruction Format

2.0 Architectural Description (Continued)

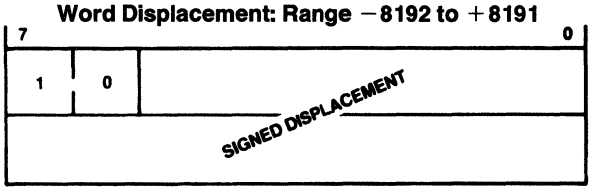
Memory Space: Identical to Register Relative above, except that the register used is one of the dedicated registers PC, SP, SB or FP. These registers point to data areas generally needed by high-level languages.

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.

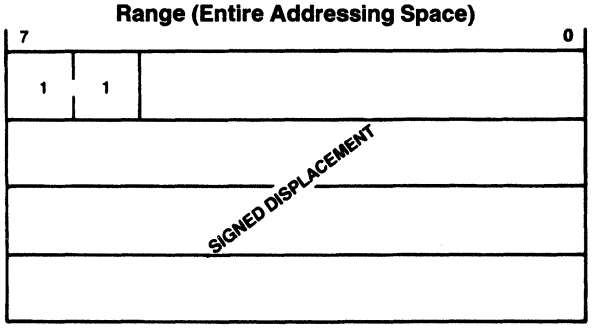
Byte Displacement: Range −64 to +63



Word Displacement: Range −8192 to +8191



Double Word Displacement: Range (Entire Addressing Space)



TL/EE/9424–6

FIGURE 2-8. Displacement Encodings

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

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

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

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

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

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

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

2.0 Architectural Description (Continued)

TABLE 2-1. NS32CG16 Addressing Modes

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

2.0 Architectural Description (Continued)

2.3.3 Instruction Set Summary

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

Notations:

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

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

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

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

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

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

reg = Any General Purpose Register: R0–R7.

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

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

TABLE 2-2. NS32CG16 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.

2.0 Architectural Description (Continued)

TABLE 2-2. NS32CG16 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.

2.0 Architectural Description (Continued)

TABLE 2-2. NS32CG16 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 strong 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	CXP	disp	Call external procedure
3	CXPD	gen	Call external procedure using descriptor.
1	SVC		Supervisor call.
1	FLAG		Flag trap.
1	BPT		Breakpoint trap.
1	ENTER	[reg list], disp	Save registers and allocate stack frame (Enter Procedure).
1	EXIT	[reg list]	Restore registers and reclaim stack frame (Exit Procedure).
1	RET	disp	Return from subroutine.
1	RXP	disp	Return from external procedure call.
1	RETT	disp	Return from trap. (Privileged)
1	RETI		Return from interrupt. (Privileged)

CPU REGISTER MANIPULATION

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

2.0 Architectural Description (Continued)

TABLE 2-2. NS32CG16 Instruction Set Summary (Continued)

FLOATING POINT

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

MISCELLANEOUS

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

GRAPHICS

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

BITS

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

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

2.0 Architectural Description (Continued)

2.4 GRAPHICS SUPPORT

The following sections provide a brief description of the NS32CG16 graphics support capabilities. Basic discussions on frame buffer addressing and BITBLT operations are also provided. More detailed information on the NS32CG16 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 2-9*. The origin of the Cartesian space (x=0, y=0) corresponds to the bit address 'ORG'. Incrementing the x coordinate increments the bit address by one. Incrementing the y coordinate increments the bit address by an amount representing the warp (or pitch) of the Cartesian space. Thus, the linear address of a pixel at location (x, y) in the Cartesian space can be found by the following expression.

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

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

Example 1 below shows two NS32CG16 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:

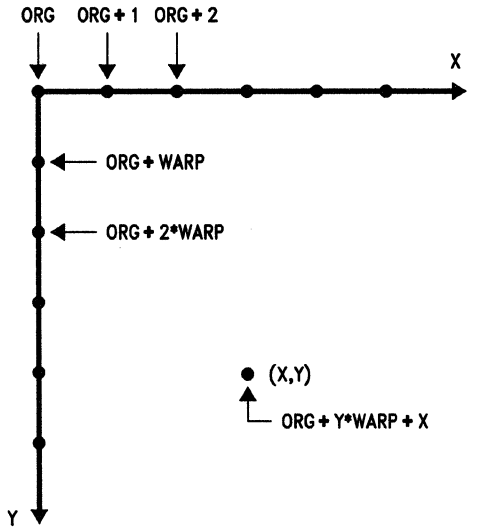
```
INDEXD R1, (WARP-1), R0  ; Y*WARP + X
SBITD  R1, 0RG           ; SET PIXEL
```

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, ORG            ; SET FIRST PIXEL

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

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

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



TL/EE/9424-61

FIGURE 2-9. 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.

2.0 Architectural Description (Continued)

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 2-10*, 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.

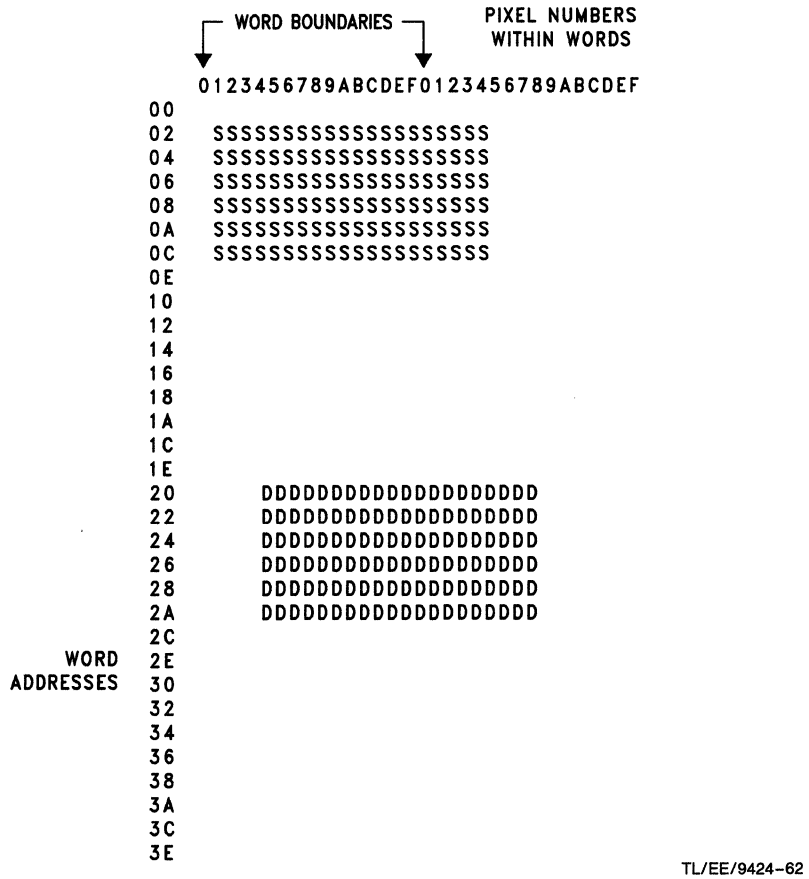
2.4.2.3 Block Boundaries and Destination Masks

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 2-10* 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.



TL/EE/9424-62

FIGURE 2-10. 32-Pixel by 32-Scan Line Frame Buffer

2.0 Architectural Description (Continued)

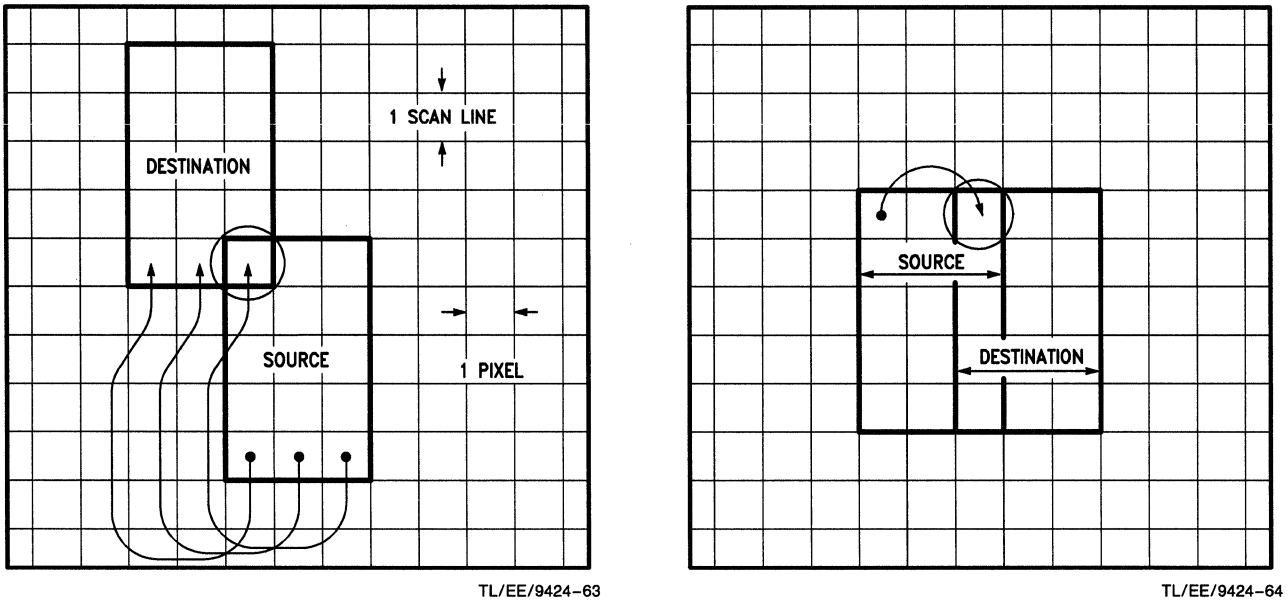


FIGURE 2-11. Overlapping BITBLT Blocks

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

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

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

2.4.2.2 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 preceeded 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 NS32CG16 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 2-11(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 2-11(a)*, if the BITBLT is performed in the UP direction (bottom-to-top) one of the transfers of the bottom scan line of the source will write to the circled pixel of the destination. Due to the overlap, this pixel is also part of the uppermost scan line of the source rectangle. Thus, data needed later is destroyed. Therefore, this BITBLT must be performed in the DOWN direction. Another example of this oc-

2.0 Architectural Description (Continued)

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

Figure 2-11(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.

2.4.2.5 BITBLT Variations

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

Texture op1 Source op2 Destination → Destination

While the NS32CG16 and the external BPU (if used) are essentially two-operand devices, three-operand BITBLT operations can be implemented quite flexibly and efficiently by performing the two operations serially.

2.4.3 GRAPHICS SUPPORT INSTRUCTIONS

The NS32CG16 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 seven instructions. They are used to move characters and objects into the frame buffer which will be printed or displayed. One of the instructions works in conjunction with an external BITBLT Processing Unit (BPU) to maximize performance. The other six are executed by the NS32CG16.

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 2-10* masking was defined relative to the destination block.

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

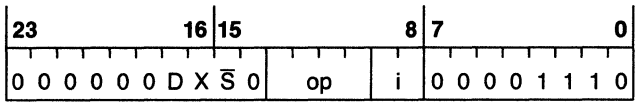
Note 3: BBFOR performs the same operation as BBOR with IA and S options.

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 2-12* 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
- $\bar{S}$ is set when the −S option is selected
- X is set for BBAND, and it is clear for all other BB instructions

FIGURE 2-12. 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.

2.0 Architectural Description (Continued)

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 2-13*.

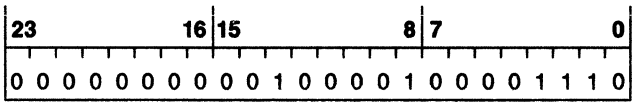


FIGURE 2-13. BITWT Instruction Format

External BITBLT

Syntax: EXTBLT

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

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

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

This instruction performs an entire BITBLT operation in conjunction with an external BITBLT Processing Unit (BPU). The external BPU Control Register should be loaded by the software before the instruction is executed (refer to the DP8510 or DP8511 data sheets for more information on the BPU). The NS32CG16 generates a series of source read, destination read and destination write bus cycles until the entire data block has been transferred. The BITBLT operation can be performed in either horizontal direction. As controlled by the sign of the contents of register R4.

Depending on the relative alignment of the source and destination blocks, an extra source read may be required at the beginning of each scan line, to load the pipeline register in the external BPU. The L bit in the PSR register determines whether the extra source read is performed. If L is 1, no extra read is performed. The instructions CMPQB 2,1 or CMPQB 1,2 could be executed to provide the right setting for the L bit just before executing EXTBLT. *Figure 2-14* shows the EXTBLT format. The bus activity for a simple BITBLT operation is shown in *Figure 2-19*.

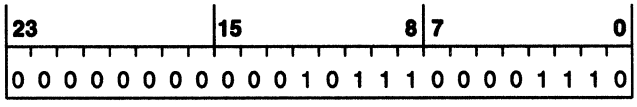


FIGURE 2-14. EXTBLT Instruction Format

B.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 2-15*.

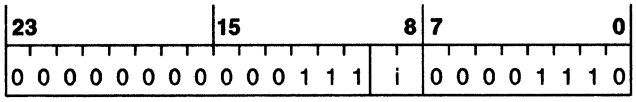


FIGURE 2-15. MOVMPi Instruction Format

B.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

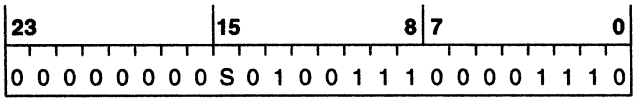
Syntax: TBITS option

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 2-16* shows the TBITS instruction format.



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

FIGURE 2-16. TBITS Instruction Format

2.0 Architectural Description (Continued)

Set Bit String

Syntax: SBITS

Setup:

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

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

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

When SBITS begins executing, it compares the value in R2 with 25. If the value in R2 is less than or equal to 25, the F flag is cleared and the appropriate number of bits are set in memory. If R2 is greater than 25, the F flag is set and no other action is performed. This allows the software to use a faster algorithm to set longer strings of bits. *Figure 2-17* shows the SBITS instruction format.

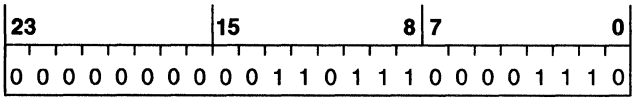


FIGURE 2-17. SBITS Instruction Format

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 2-18* shows the SBITPS instruction format.

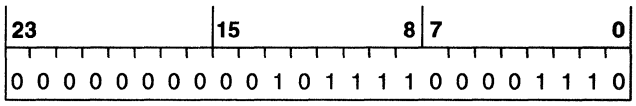
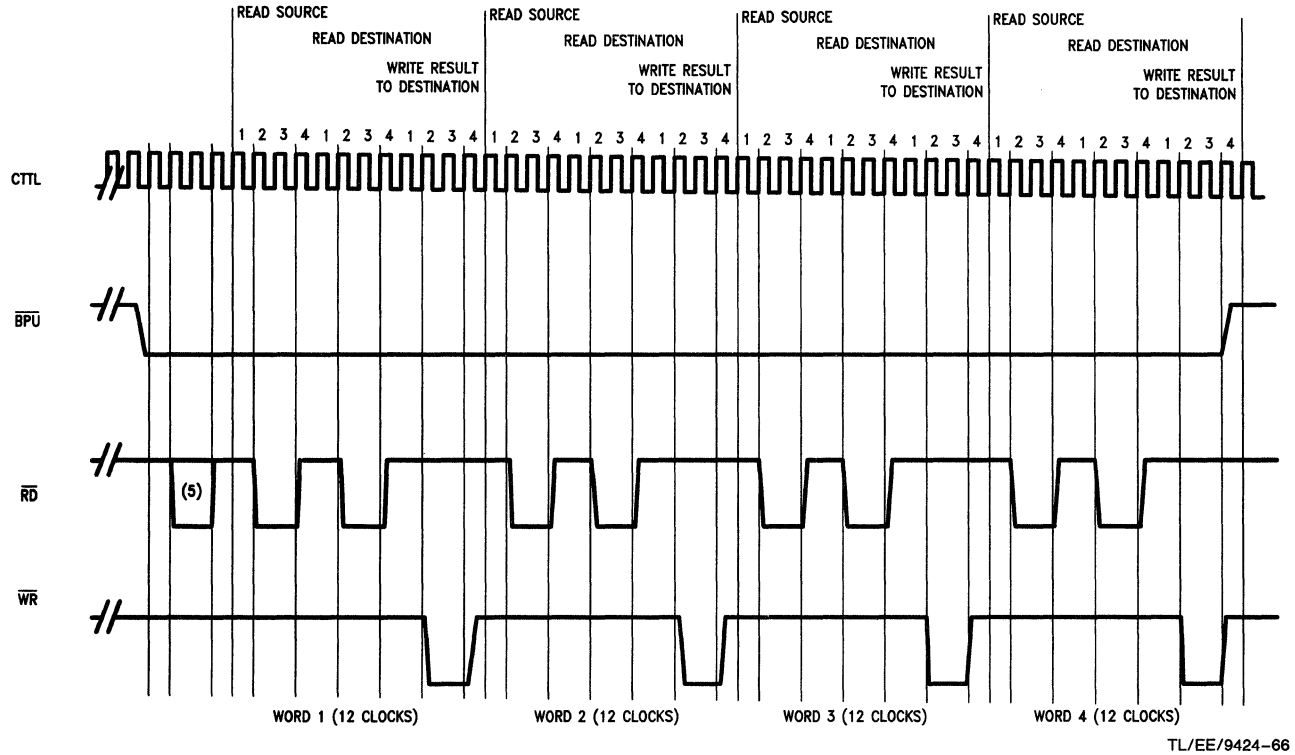


FIGURE 2-18. SBITPS Instruction Format



TL/EE/9424–66

FIGURE 2-19. Bus Activity for a Simple BITBLT Operation

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

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

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

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

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

2.0 Architectural Description (Continued)

B.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

3.1 POWER AND GROUNDING

The NS32CG16 requires a single 5-Volt power supply, applied on 5 pins. The logic voltage pin (V_{CCL}) supplies the power to the on-chip logic. The buffer voltage pins VCCCTTL, VCCFCLK, VCCAD, and VCCIO supply the power to the on-chip output drivers.

Grounding connections are made on 6 pins. The Logic Ground Pin (VSSL) provides the ground connection to the on-chip logic. The buffer ground pins VSSFCLK, VSSNTSO, VSSHAD, VSSLAD, VSSIO are the ground pins for the on-chip output drivers.

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. In addition, a 1.0 μF tantalum capacitor should be connected between V_{CCL} and ground. They should attach to V_{CC}, V_{SS} pairs as close as possible to the NS32CG16.

During prototype using wire-wrap or similar methods, the capacitors should be soldered directly to the power pins of the NS32CG16 socket, or as close as possible, with very short leads.

Recommended bypass for production in printed circuit boards:

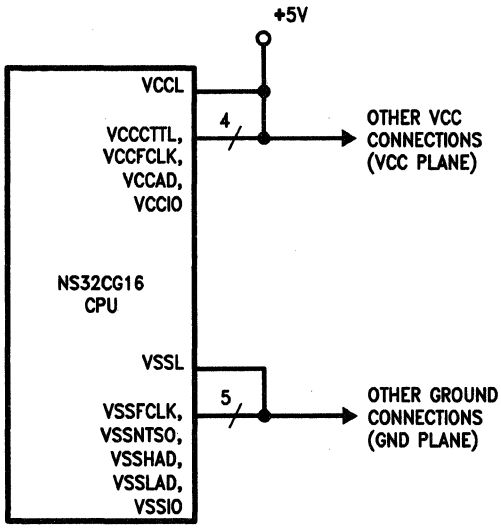
+5	Ground	Capacitors
VCCL	VSSL	0.1 μF Disk Ceramic 1.0 μF Tantulum
VCCIO	VSSIO	0.1 μF
VCCCTTL	VSSNTSO	0.1 μF
VCCAD	VSSLAD	0.1 μF
VCCAD	VSSHAD	None
VCCFCLK	VSSFCLK	0.1 μF

VCCL-VSSL bypass requires a very short lead length and low inductance on the 0.1 μF capacitor.

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 NS32CG16. 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 NS32CG16.

Power and ground connections are shown in *Figure 3-1.*



TL/EE/9424–7

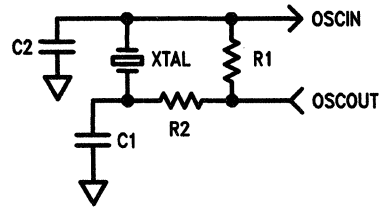
FIGURE 3-1. Power and Ground Connections

3.2 CLOCKING

The NS32CG16 provides an internal oscillator that interacts with an external clock source through two signals; OSCIN and OSCOUT.

Either an external single-phase clock signal or a crystal can be used as the clock source. If a single-phase clock source is used, only the connection on OSCIN is required; OSCOUT should be left open. The voltage level requirements specified in Section 4.3 must also be met for proper operation.

When operation with a crystal is desired, a fundamental mode crystal should be used. In this case, special care should be taken to minimize stray capacitances and inductances, especially when operating at a crystal frequency of 30 MHz. The crystal, as well as the external RC components, should be placed in close proximity to the OSCIN and OSCOUT pins to keep the printed circuit trace lengths to an absolute minimum. *Figure 3-2* shows the external crystal interconnections. Table 3-1 provides the crystal characteristics and the values of the RC components required for various frequencies.



TL/EE/9424–8

FIGURE 3-2. Crystal Interconnections

3.0 Functional Description (Continued)

TABLE 3-1. External Oscillator Specifications

Crystal Characteristics

Type ..... At-Cut
Tolerance ..... 0.005% at 25°C
Stability ..... 0.01% from 0°C to 70°C
Resonance ..... Fundamental (parallel)
Capacitance ..... 20 pF
Maximum Series Resistance 50Ω

RC Component Values

Frequency (MHz)	R1 (kΩ)	R2 (Ω)	C1 (pF)	C2 (pF)
12	470	120	20	30
16	360	100	20	30
20	270	75	20	30
25	220	68	20	30
30	180	51	20	30

3.2.1 Power Save Mode

The NS32CG16 provides a power save feature that can be used to significantly reduce the power consumption at times when the computational demand decreases. The device uses the clock signal at the OSCIN pin to derive the internal clock as well as the external signals PHI1, PHI2, CTTL and FCLK. The frequency of all these clock signals is affected by the clock scaling factor. Scaling factors of 1, 2, 4 or 8 can be selected by properly setting the C and M bits in the CFG register.

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

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

The following table shows the C and M bit settings for the various scaling factors, and the resulting supply current for a crystal frequency of 30 MHz.

Clock Scaling Factor vs Supply Current

C	M	Scaling Factor	CPU Clock Frequency	Typical I_{CC} at +5V
0	0	1	15 MHz	140 mA
0	1	2	7.5 MHz	76 mA
1	0	4	3.75 MHz	42 mA
1	1	8	1.88 MHz	25 mA

3.3 RESETTING

The $\overline{RSTI}$ input pin is used to reset the NS32CG16. The CPU samples $\overline{RSTI}$ on the falling edge of CTTL.

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

On application of power, $\overline{RSTI}$ must be held low for at least 50 μs after V_{CC} is stable. This is to ensure that all on-chip voltages are completely stable before operation. Whenever a Reset is applied, it must also remain active for not less than 64 CTTL cycles. See *Figures 3-3* and *3-4*.

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

The internal CPU clock, PHI1, PHI2 and CTTL all run at half the frequency of the signal on the OSCIN pin. FCLK runs at the same frequency of OSCIN.

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

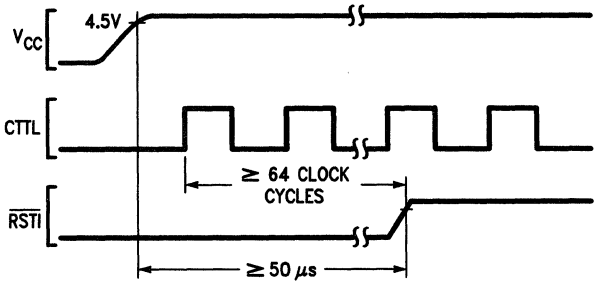
The PSR is reset to 0. The CFG C and M bits are reset to 0. $\overline{NMI}$ is enabled to allow Non-Maskable Interrupts. The following conditions are present after reset due to the PSR being reset to 0:

Tracing is disabled.
Supervisor mode is enabled.
Supervisor stack space is used when the TOS addressing mode is indicated.
No trace traps are pending.
Only $\overline{NMI}$ is enabled. $\overline{INT}$ is not enabled.
$\overline{BPU}$ is inactive high.
The Clock Scaling Factor is set to 1, refer to Section 3.2.1.

Note that vector/non-vectored interrupts have not been selected. While interrupts are disabled, a SETCFG [I] instruction must be executed to declare the presence of the NS32202 if vectored interrupts are desired. If non-vectored interrupts are required, a SETCFG without the [I] must be executed.

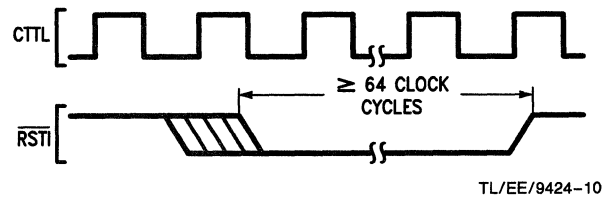
The presence/absence of the NS32081 or NS32381 has also not been declared. If there is a Floating Point Unit, a SETCFG [F] instruction must be executed. If there is no floating point unit, a SETCFG without the [F] must be executed.

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



TL/EE/9424–9

FIGURE 3-3. Power-On Reset Requirements



TL/EE/9424–10

FIGURE 3-4. General Reset Timing

3.0 Functional Description (Continued)

3.4 BUS CYCLES

The CPU will perform a bus cycle for one of the following reasons:

1) To write or read data, to or from memory or peripheral devices. Peripheral input and output are memory-mapped in the Series 32000 family.
2) To fetch instructions into the eight-byte instruction queue. This happens whenever the bus would otherwise be idle and the queue is not already full.
3) To acknowledge an interrupt and allow external circuitry to provide a vector number, or to acknowledge completion of an interrupt service routine.
4) To transfer information to or from a Slave Processor.

In terms of bus timing, cases 1 through 3 above are identical. For timing specifications, see Section 4. The only external difference between them is the four-bit code placed on the Bus Status pins (ST0–ST3). Slave Processor cycles differ in that separate control signals are applied (Section 3.4.7).

3.4.1 Bus Status

The NS32CG16 CPU presents four bits of Bus Status information on pins ST0–ST3. The various combinations on these pins indicate why the CPU is performing a bus cycle, or, if it is idle on the bus, then why it is idle.

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

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

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

0010 — (Reserved for future use.)

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

0100 — Interrupt Acknowledge, Master.

The CPU is performing a Read cycle to acknowledge an interrupt request. See Section 3.4.6.

0101 — Interrupt Acknowledge, Cascaded.

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

0110 — End of Interrupt, Master.

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

0111 — End of Interrupt, Cascaded.

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

1000 — Sequential Instruction Fetch.

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

1001 — Non-Sequential Instruction Fetch.

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

1010 — Data Transfer.

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

1011 — Read RMW Operand.

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

1100 — Read for Effective Address Calculation.

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

1101 — Transfer Slave Processor Operand.

The CPU is either transferring an instruction operand to or from a Slave Processor, or it is issuing the Operation Word of a Slave Processor instruction. See Section 3.9.1.

1110 — Read Slave Processor Status.

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

1111 — Broadcast Slave ID.

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

3.4.2 Basic Read and Write Cycles

The sequence of events occurring during a CPU access to either memory or peripheral device is shown in *Figure 3-6* for a read cycle, and *Figure 3-7* for a write cycle.

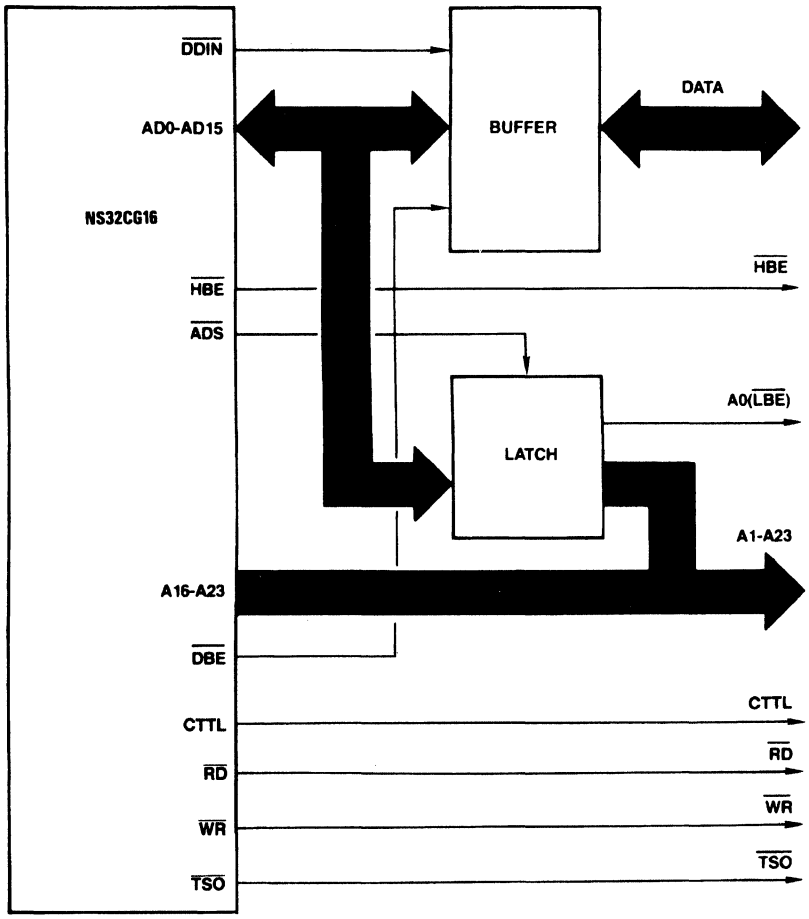
The cases shown assume that the selected memory or peripheral device is capable of communicating with the CPU at full speed. If not, then cycle extension may be requested through $\overline{\text{CWAIT}}$ and/or $\overline{\text{WAIT}}$1–2.

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

During T1, the CPU applies an address on pins AD0–AD15 and A16–A23. It also provides a low-going pulse on the $\overline{\text{ADS}}$ pin, which serves the dual purpose of informing external circuitry that a bus cycle is starting and of providing control to an external latch for demultiplexing Address bits 0–15 from the AD0–AD15 pins. See *Figure 3-5*. During this time also the status signals $\overline{\text{DDIN}}$, indicating the direction of the transfer, and $\overline{\text{HBE}}$, indicating whether the high byte (AD8–AD15) is to be referenced, become valid.

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

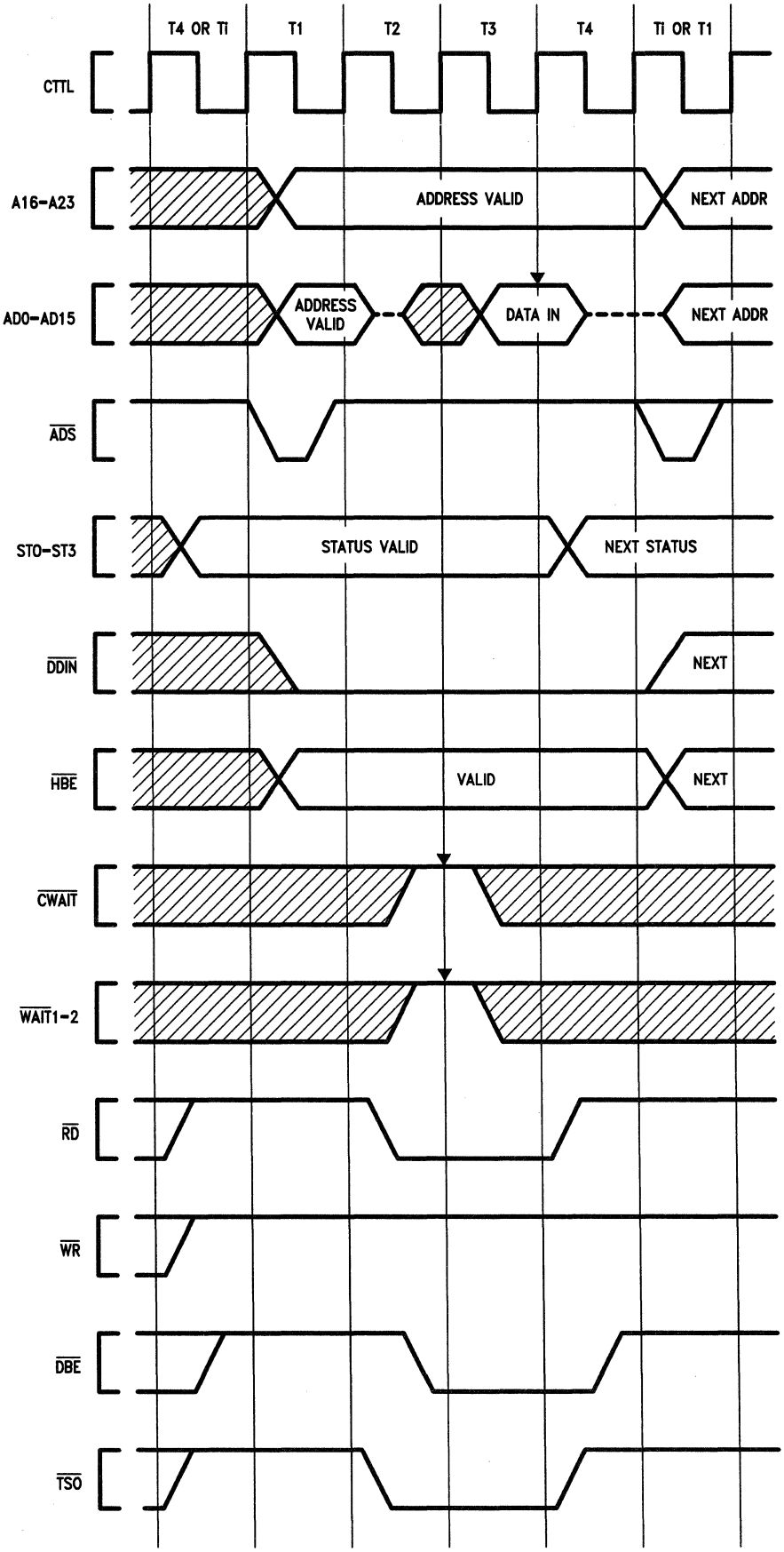
3.0 Functional Description (Continued)



TL/EE/9424–11

FIGURE 3-5. Bus Connections

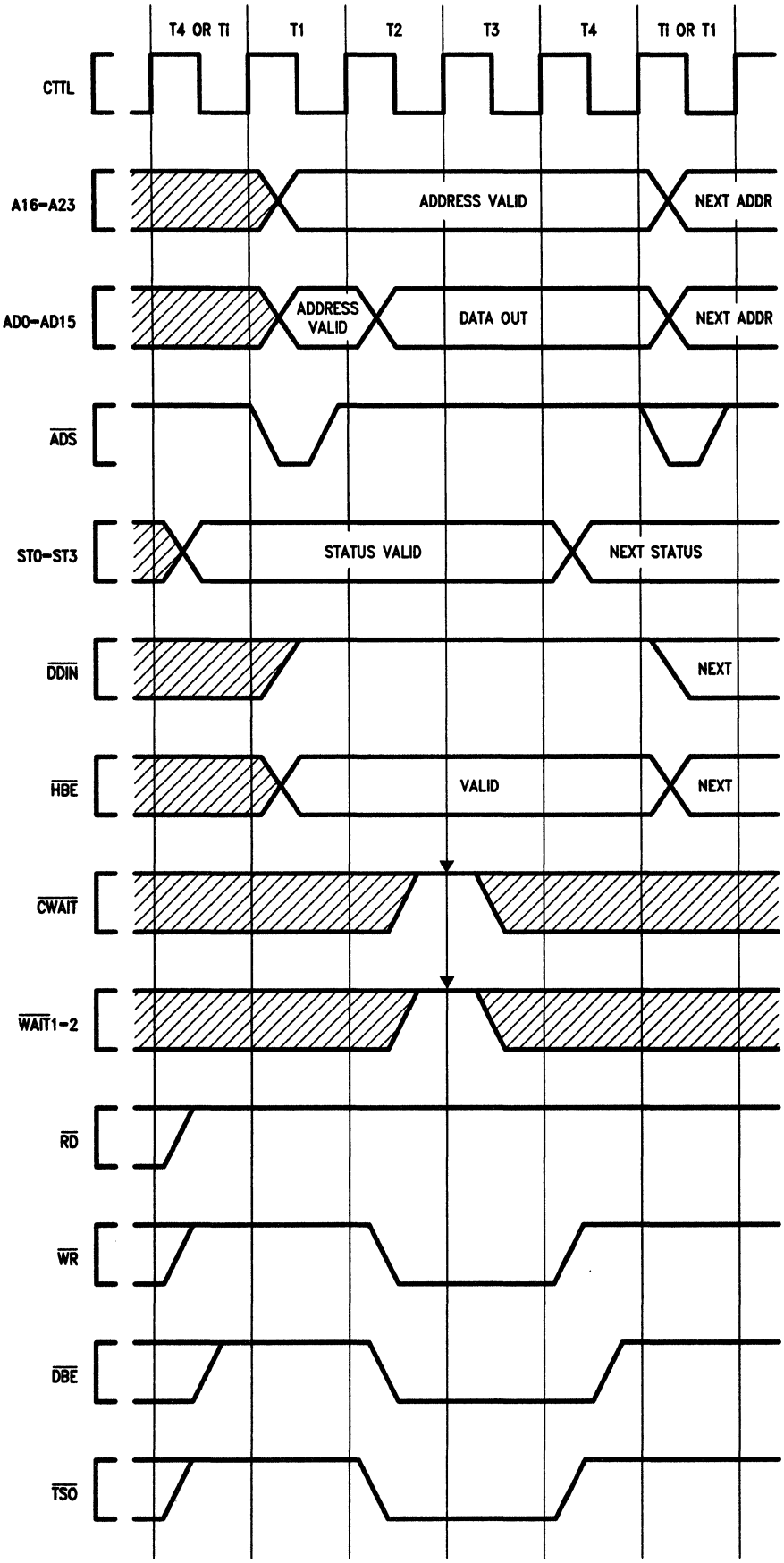
3.0 Functional Description (Continued)



TL/EE/9424–12

FIGURE 3-6. Read Cycle Timing

3.0 Functional Description (Continued)



TL/EE/9424–13

FIGURE 3-7. Write Cycle Timing

3.0 Functional Description (Continued)

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

The T3 state provides for access time requirements, and it occurs at least once in a bus cycle. At the end of T2, on the rising edge of CTTL, the $\overline{CWAIT}$ and $\overline{WAIT}1-2$ signals are sampled to determine whether the bus cycle will be extended. See Section 3.4.3.

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

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

3.4.3 Cycle Extension

To allow sufficient access time for any speed of memory or peripheral device, the NS32CG16 provides for extension of a bus cycle. Any type of bus cycle except a Slave Processor cycle can be extended.

In *Figures 3-6* and *3-7*, note that during T3 all bus control signals from the CPU are flat. Therefore, a bus cycle can be cleanly extended by causing the T3 state to be repeated. This is the purpose of the $\overline{WAIT}1-2$ and $\overline{CWAIT}$ input signals.

At the end of state T2, on the rising edge of CTTL, $\overline{WAIT}1-2$ and $\overline{CWAIT}$ are sampled.

If any of these signals are active, the bus cycle will be extended by at least one clock cycle. Thus, one or more additional T3 state (also called wait state) will be inserted after the next T-State. Any combination of the above signals can be activated at one time. However, the $\overline{WAIT}1-2$ inputs are only sampled by the CPU at the end of state T2. They are ignored at all other times.

The $\overline{WAIT}1-2$ inputs are binary weighted, and can be used to insert up to 3 wait states, according to the following table.

$\overline{WAIT}2$	$\overline{WAIT}1$	Number of Wait States
HIGH	HIGH	0
HIGH	LOW	1
LOW	HIGH	2
LOW	LOW	3

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

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

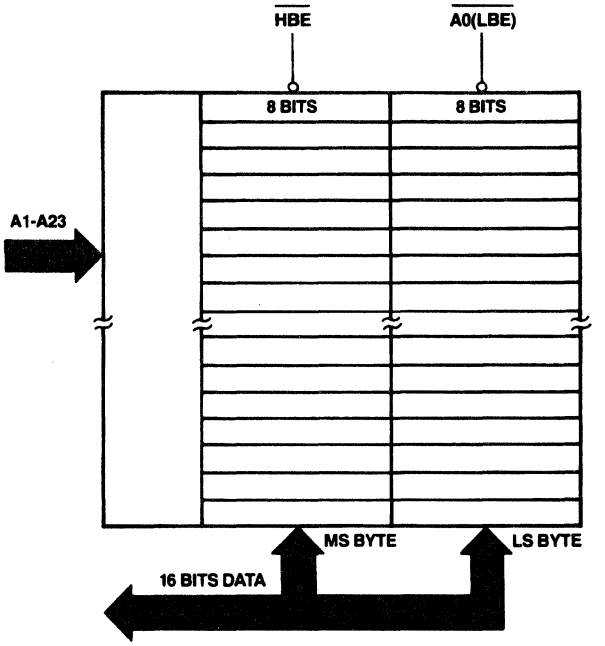
1. Start bus cycle.
2. Sample $\overline{WAIT}1-2$ and $\overline{CWAIT}$ at the end of state T2.
3. If the $\overline{WAIT}1-2$ inputs are both inactive, then go to step 6.
4. Insert the number of wait states selected by $\overline{WAIT}1-2$.
5. Sample $\overline{CWAIT}$ again.
6. If $\overline{CWAIT}$ is not active, then go to step 8.
7. Insert one wait state and then go to step 5.
8. Complete bus cycle.

Figure 3-8 shows a bus cycle extended by three wait states, two of which are due to $\overline{WAIT}2$, and one is due to $\overline{CWAIT}$.

3.4.4 Data Access Sequences

The 24-bit address provided by the NS32CG16 is a byte address; that is, it uniquely identifies one of up to 16,777,216 eight-bit memory locations. An important feature of the NS32CG16 is that the presence of a 16-bit data bus imposes no restrictions on data alignment; any data item, regardless of size, may be placed starting at any memory address. The NS32CG16 provides a special control signal, High Byte Enable ($\overline{HBE}$), which facilitates individual byte addressing on a 16-bit bus.

Memory is organized as two eight-bit banks, each bank receiving the word address (A1–A23) in parallel. One bank, connected to Data Bus pins AD0–AD7, is enabled to respond to even byte addresses; i.e., when the least significant address bit (A0) is low. The other bank, connected to Data Bus pins AD8–AD15, is enabled when $\overline{HBE}$ is low. See *Figure 3-9.*



TL/EE/9424–15

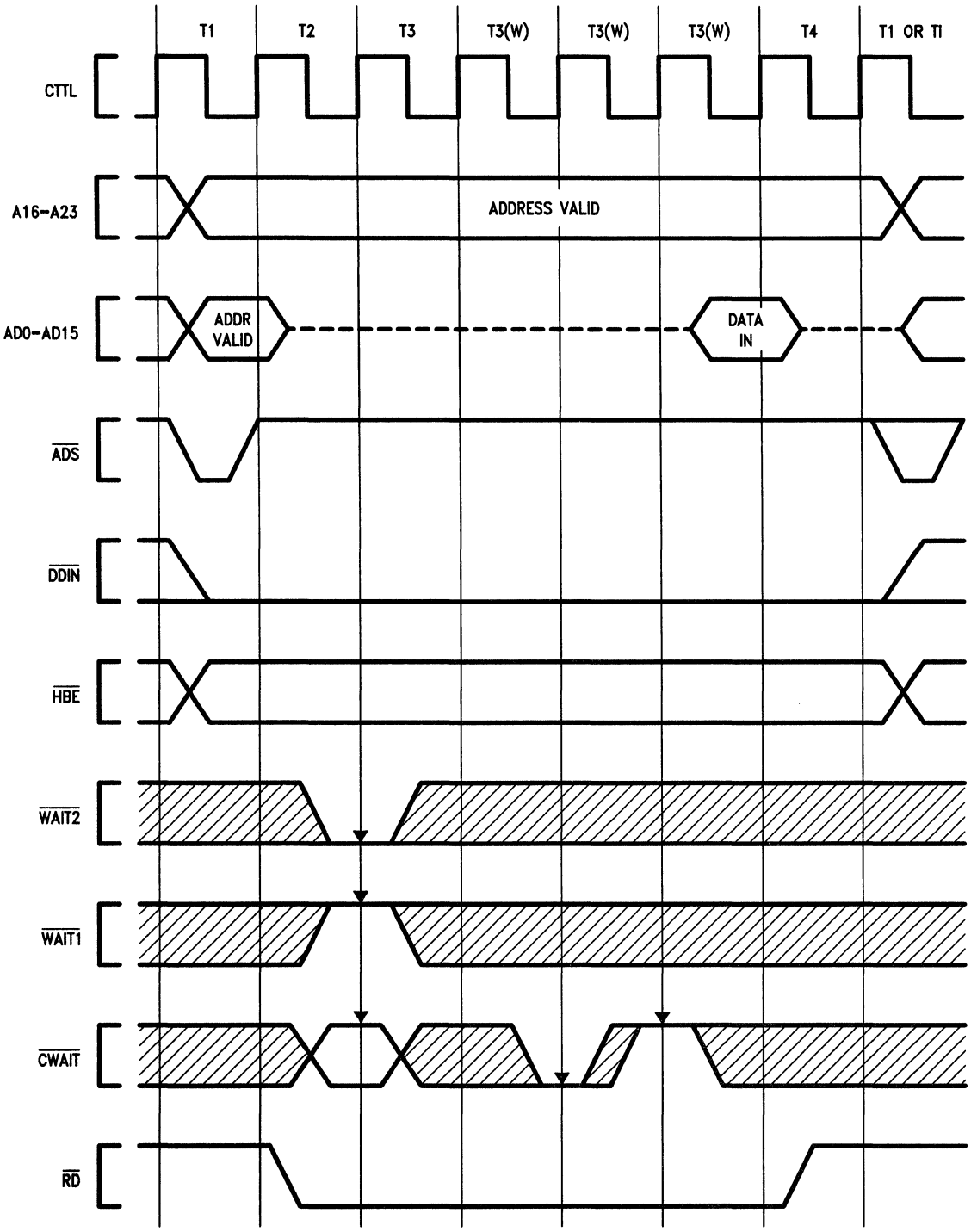
FIGURE 3-9. Memory Interface

Any bus cycle falls into one of three categories: Even Byte Access, Odd Byte Access, and Even Word Access. All accesses to any data type are made up of sequences of these cycles. Table 3-2 gives the state of A0 and $\overline{HBE}$ for each category.

TABLE 3-2. Bus Cycle Categories

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

3.0 Functional Description (Continued)



TL/EE/9424–14

FIGURE 3-8. Cycle Extension of a Read Cycle

3.0 Functional Description (Continued)

Accesses of operands requiring more than one bus cycle are performed sequentially, with no idle T-States separating them. The number of bus cycles required to transfer an operand depends on its size and its alignment (i.e., whether it starts on an even byte address or an odd byte address). Table 3-3 lists the bus cycle performed for each situation. For the timing of A0 and $\overline{HBE}$, see Section 3.4.2.

3.4.4.1 Bit Accesses

The Bit Instructions perform byte accesses to the byte containing the designated bit. The Test and Set Bit instruction (SBIT), for example, reads a byte, alters it, and rewrites it, having changed the contents of one bit.

3.4.4.2 Bit Field Accesses

An access to a Bit Field in memory always generates a Double-Word transfer at the address containing the least significant bit of the field. The Double Word is read by an Extract instruction; an Insert instruction reads a Double Word, modifies it, and rewrites it.

3.4.4.3 Extending Multiply Accesses

The Multiply Extended Integer (MEI) instruction will return a result which is twice the size in bytes of the operand it reads. If the multiplicand is in memory, the most-significant half of the result is written first (at the higher address), then the least-significant half.

3.4.5 Instruction Fetches

Instructions for the NS32CG16 CPU are "prefetched"; that is, they are input before being needed into the next available entry of the eight-byte Instruction Queue. The CPU performs two types of Instruction Fetch cycles: Sequential and Non-Sequential. These can be distinguished from each other by their differing status combinations on pins ST0–ST3 (Section 3.4.1).

A Sequential Fetch will be performed by the CPU whenever the Data Bus would otherwise be idle and the Instruction Queue is not currently full. Sequential Fetches are always Even Word Read cycles (Table 3-2).

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

3.4.6 Interrupt Control Cycles

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

Table 3-4 shows the Interrupt Control sequences associated with each interrupt and with the return from its service routine. For full details of the NS32CG16 interrupt structure, see Section 3.8.

3.0 Functional Description (Continued)

TABLE 3-3. Access Sequences

Cycle	Type	Address	$\overline{HBE}$	A0	High Bus	Low Bus
					BYTE 1	**BYTE 0** ← A
					A. Odd Word Access Sequence	
1	Odd Byte	A	0	1	Byte 0	Don't Care
2	Even Byte	A+1	1	0	Don't Care	Byte 1
					BYTE 3 **BYTE 2** **BYTE 1**	**BYTE 0** ← A
					B. Even Double-Word Access Sequence	
1	Even Word	A	0	0	Byte 1	Byte 0
2	Even Word	A+2	0	0	Byte 3	Byte 2
					BYTE 3 **BYTE 2** **BYTE 1**	**BYTE 0** ← A
					C. Odd Double-Word Access Sequence	
1	Odd Byte	A	0	1	Byte 0	Don't Care
2	Even Word	A+1	0	0	Byte 2	Byte 1
3	Even Byte	A+3	1	0	Don't Care	Byte 3
					BYTE 7 **BYTE 6** **BYTE 5** **BYTE 4** **BYTE 3** **BYTE 2** **BYTE 1**	**BYTE 0** ← A
					D. Even Quad-Word Access Sequence	
1	Even Word	A	0	0	Byte 1	Byte 0
2	Even Word	A+2	0	0	Byte 3	Byte 2
Other bus cycles (instruction prefetch or slave) can occur here.						
3	Even Word	A+4	0	0	Byte 5	Byte 4
4	Even Word	A+6	0	0	Byte 7	Byte 6
					BYTE 7 **BYTE 6** **BYTE 5** **BYTE 4** **BYTE 3** **BYTE 2** **BYTE 1**	**BYTE 0** ← A
					E. Odd Quad-Word Access Sequence	
1	Odd Byte	A	0	1	Byte 0	Don't Care
2	Even Word	A+1	0	0	Byte 2	Byte 1
3	Even Byte	A+3	1	0	Don't Care	Byte 3
Other bus cycles (instruction prefetch or slave) can occur here.						
4	Odd Byte	A+4	0	1	Byte 4	Don't Care
5	Even Word	A+5	0	0	Byte 6	Byte 5
6	Even Byte	A+7	1	0	Don't Care	Byte 7

3.0 Functional Description (Continued)

TABLE 3-4. Interrupt Sequences

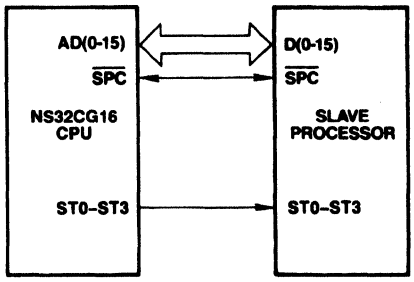
Cycle	Status	Address	$\overline{DDIN}$	$\overline{HBE}$	A0	High Bus	Low Bus
			A. Non-Maskable Interrupt Control Sequence				
Interrupt Acknowledge							
1	0100	$FFFF00_{16}$	0	1	0	Don't Care	Don't Care
Interrupt Return							
None: Performed through Return from Trap (RETT) instruction.							
			B. Non-Vectored Interrupt Control Sequence				
Interrupt Acknowledge							
1	0100	$FFFE00_{16}$	0	1	0	Don't Care	Don't Care
Interrupt Return							
None: Performed through Return from Trap (RETT) instruction.							
			C. Vectored Interrupt Sequence: Non-Cascaded				
Interrupt Acknowledge							
1	0100	$FFFE00_{16}$	0	1	0	Don't Care	Vector: Range: 0–127
Interrupt Return							
1	0110	$FFFE00_{16}$	0	1	0	Don't Care	Vector: Same as in Previous Int. Ack. Cycle
			D. Vectored Interrupt Sequence: Cascaded				
Interrupt Acknowledge							
1	0100	$FFFE00_{16}$	0	1	0	Don't Care	Cascade Index: range −16 to −1
(The CPU here uses the Cascade Index to find the Cascade Address.)							
2	0101	Cascade Address	0	1 or 0*	0 or 1*	Vector, range 0–255; on appropriate half of Data Bus for even/odd address	
Interrupt Return							
1	0110	$FFFE00_{16}$	0	1	0	Don't Care	Cascade Index: same as in previous Int. Ack. Cycle
(The CPU here uses the Cascade Index to find the Cascade Address.)							
2	0111	Cascade Address	0	1 or 0*	0 or 1*	Don't Care	Don't Care

* If the Cascaded ICU Address is Even (A0 is low), then the CPU applies $\overline{HBE}$ high and reads the vector number from bits 0–7 of the Data Bus.
If the address is Odd (A0 is high), then the CPU applies $\overline{HBE}$ low and reads the vector number from bits 8–15 of the Data Bus. The vector number may be in the range 0–255.

3.0 Functional Description (Continued)

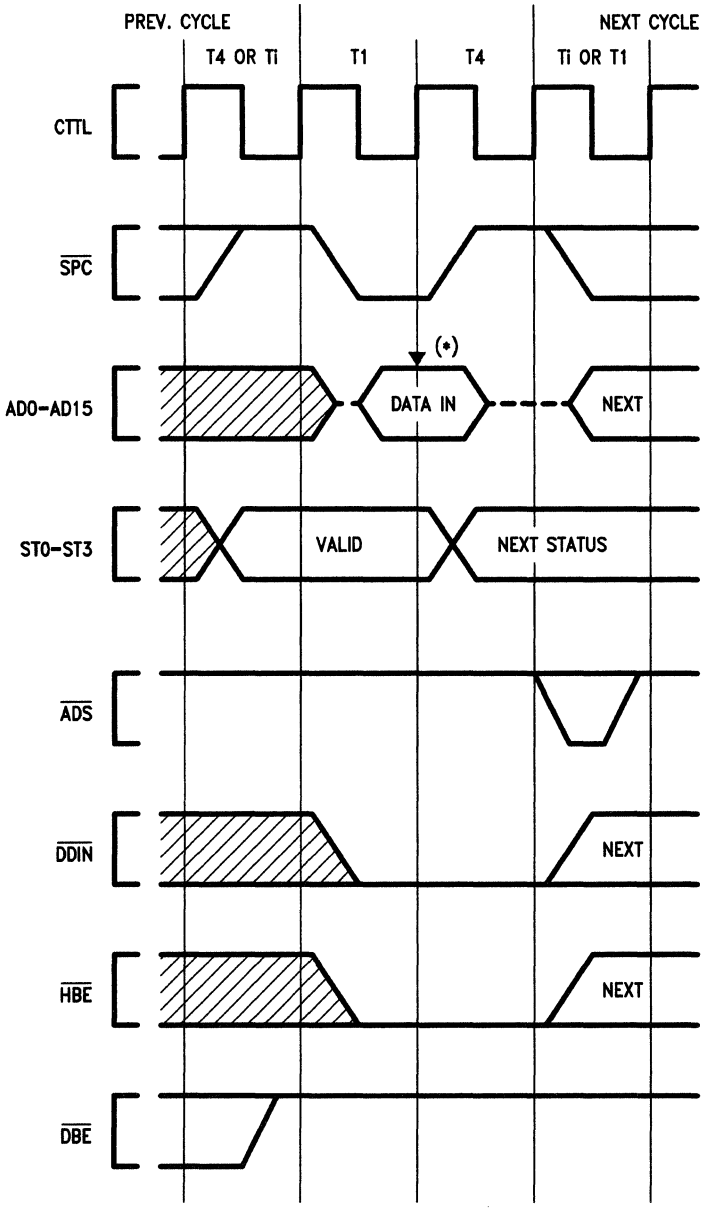
3.4.7 Slave Processor Communication

The $\overline{SPC}$ pin is used as the data strobe for Slave Processor transfers. In a Slave Processor bus cycle, data is transferred on the Data Bus (AD0–AD15), and the status lines ST0–ST3 are monitored by the Slave Processor in order to determine the type of transfer being performed. $\overline{SPC}$ is bidirectional, but is driven by the CPU during all Slave Processor bus cycles. See Section 3.8 for full protocol sequences.



TL/EE/9424–16

FIGURE 3-10. Slave Processor Connections



TL/EE/9424–17

***Note:** CPU samples Data Bus here.

FIGURE 3-11. Slave Processor Read Cycle

3.0 Functional Description (Continued)

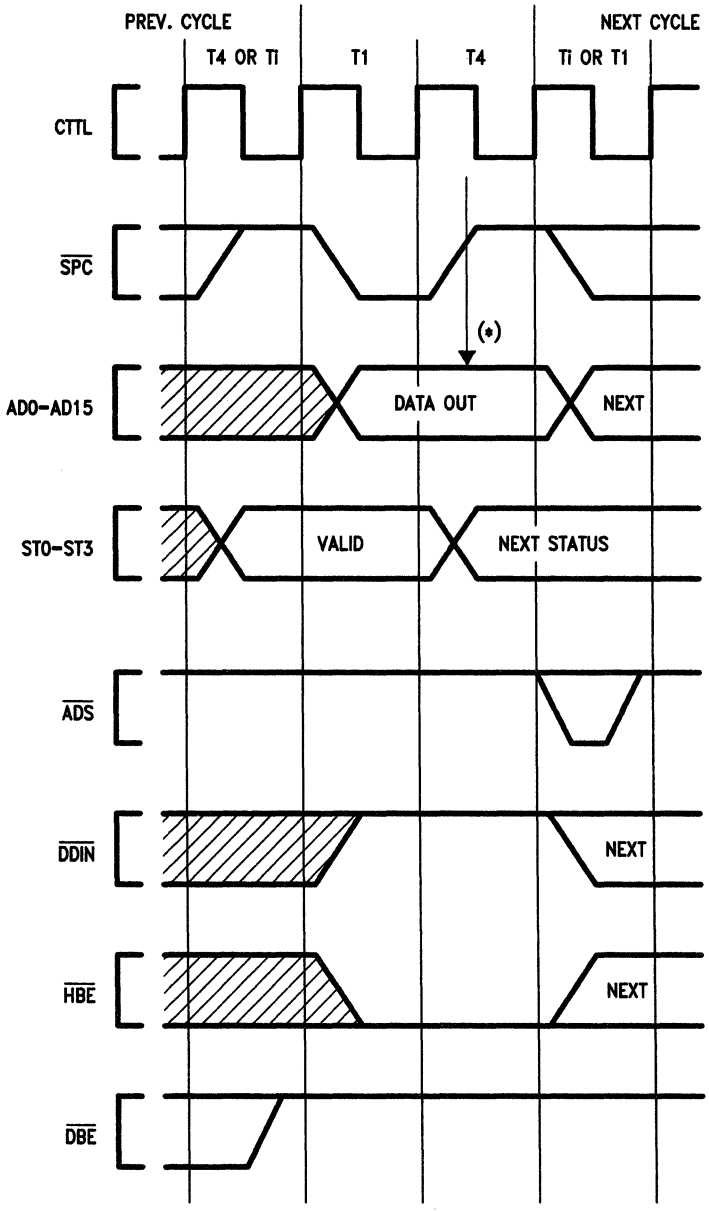
3.4.7.1 Slave Processor Bus Cycles

A Slave Processor bus cycle always takes exactly two clock cycles, labeled T1 and T4 (see *Figures 3-11* and *3-12*). During a Read cycle $\overline{SPC}$ is active from the beginning of T1 to the beginning of T4, and the data is sampled at the end of T1. The Cycle Status pins lead the cycle by one clock period, and are sampled at the leading edge of $\overline{SPC}$. During a Write cycle, the CPU applies data and activates $\overline{SPC}$ at T1, removing $\overline{SPC}$ at T4. The Slave Processor latches status on the leading edge of $\overline{SPC}$ and latches data on the trailing edge.

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

3.4.7.2 Slave Operand Transfer Sequences

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



TL/EE/9424–18

*Note: Slave Processor samples Data Bus here.

FIGURE 3-12. Slave Processor Write Cycle

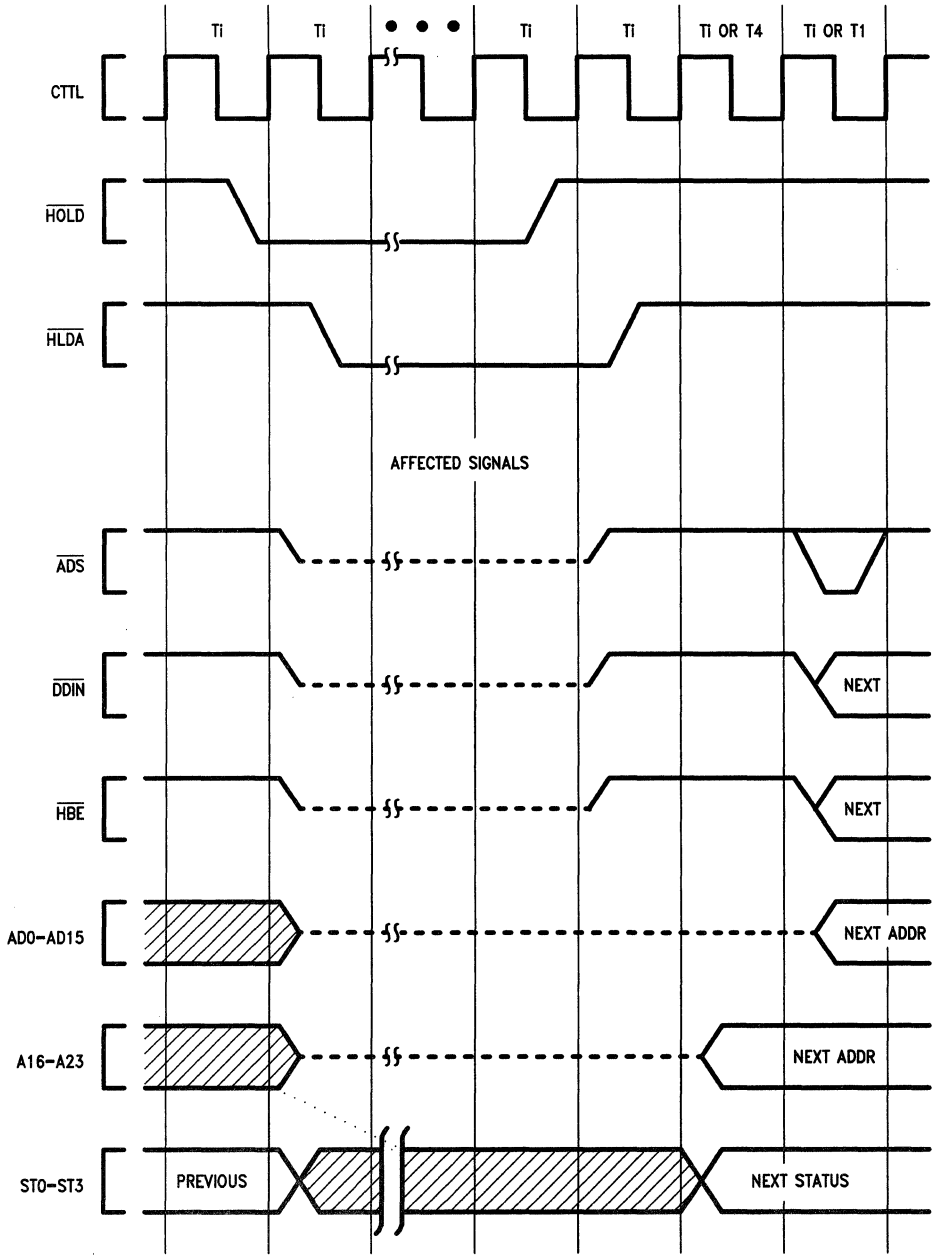
3.0 Functional Description (Continued)

3.5 BUS ACCESS CONTROL

The NS32CG16 CPU has the capability of relinquishing its access to the bus upon request from a DMA controller or another CPU. This capability is implemented on the $\overline{\text{HOLD}}$ (Hold Request) and $\overline{\text{HLDA}}$ (Hold Acknowledge) pins. By asserting $\overline{\text{HOLD}}$ low, an external device requests access to the bus. On receipt of $\overline{\text{HLDA}}$ from the CPU, the device may perform bus cycles, as the CPU at this point has set AD0–AD15, A16–A23 and $\overline{\text{HBE}}$ to the TRI-STATE® condition and has switched $\overline{\text{ADS}}$ and $\overline{\text{DDIN}}$ to the input mode. The CPU now monitors $\overline{\text{ADS}}$ and $\overline{\text{DDIN}}$ from the external device to generate the relevant strobe signals (i.e., $\overline{\text{TSO}}$, $\overline{\text{DBE}}$, $\overline{\text{RD}}$ or $\overline{\text{WR}}$). To return control of the bus to the CPU, the device sets $\overline{\text{HOLD}}$ inactive, and the CPU acknowledges return of the bus by setting $\overline{\text{HLDA}}$ inactive.

How quickly the CPU releases the bus depends on whether it is idle on the bus at the time the $\overline{\text{HOLD}}$ request is made, as the CPU must always complete the current bus cycle. *Figure 3-13* shows the timing sequence when the CPU is idle. In this case, the CPU grants the bus during the immediately following clock cycle. *Figure 3-14* shows the sequence if the CPU is using the bus at the time that the $\overline{\text{HOLD}}$ request is made. If the request is made during or before the clock cycle shown (two clock cycles before T4), the CPU will release the bus during the clock cycle following T4. If the request occurs closer to T4, the CPU may already have decided to initiate another bus cycle. In that case it will not grant the bus until after the next T4 state. Note that this situation will also occur if the CPU is idle on the bus but has initiated a bus cycle internally.

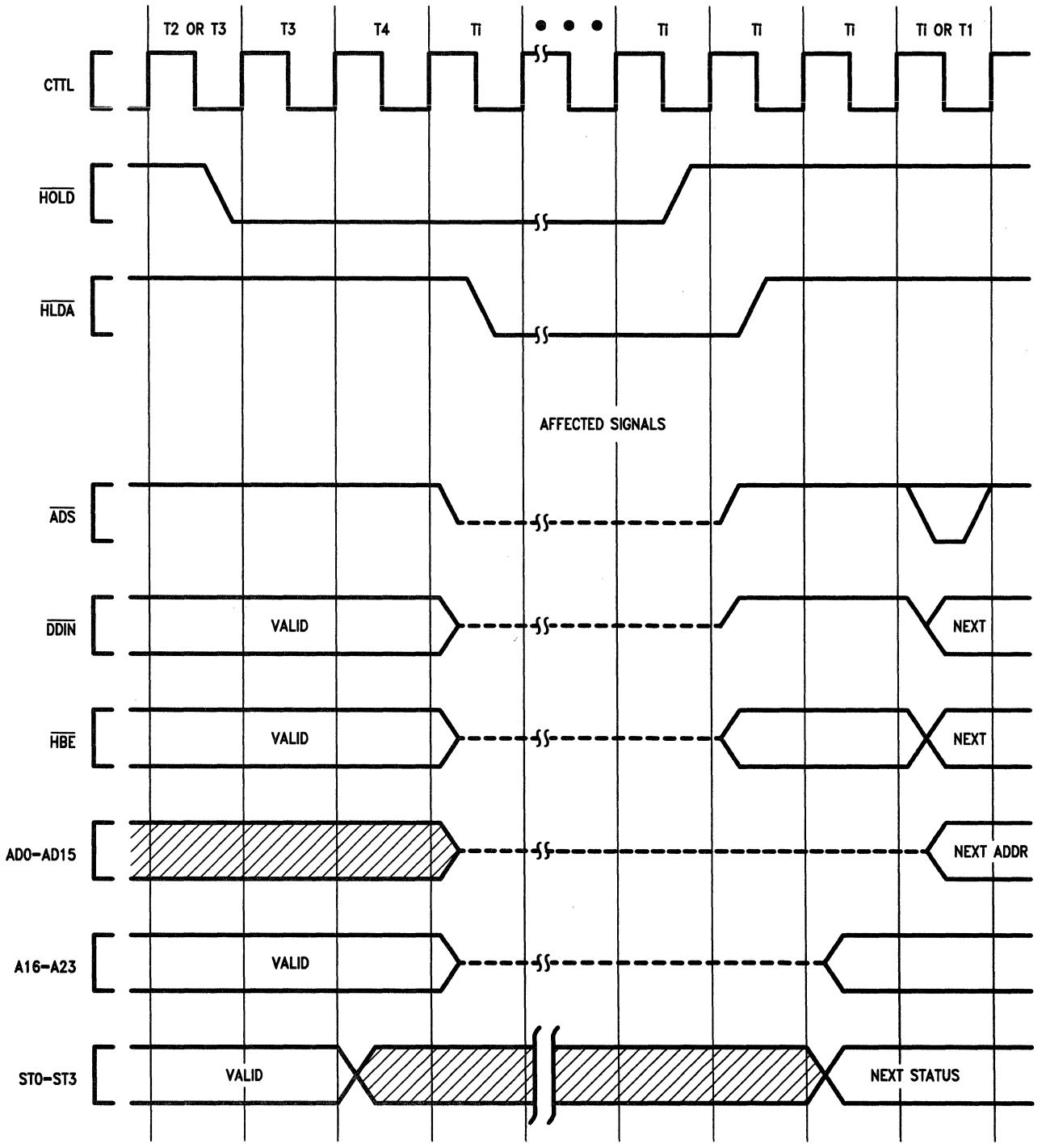
Note: During DMA cycles the $\overline{\text{WAIT}}$1–2 signals should be kept inactive, unless they are also monitored by the DMA controller. If wait states are required, $\overline{\text{CWAIT}}$ should be used.



TL/EE/9424–19

FIGURE 3-13. $\overline{\text{HOLD}}$ Timing, Bus Initially Idle

3.0 Functional Description (Continued)



TL/EE/9424-20

FIGURE 3-14. HOLD Timing, Bus Initially Not Idle

3.6 INSTRUCTION STATUS

In addition to the four bits of Bus Cycle status (ST0–ST3), the NS32CG16 CPU also presents Instruction Status information on three separate pins. These pins differ from ST0–ST3 in that they are synchronous to the CPU's internal instruction execution section rather than to its bus interface section.

PFS (Program Flow Status) is pulsed low as each instruction begins execution. It is intended for debugging purposes.

U/S originates from the U bit of the Processor Status Register, and indicates whether the CPU is currently running in User or Supervisor mode. Although it is not synchronous to bus cycles, there are guarantees on its validity during any given bus cycle. See the Timing Specifications in Section 4.

3.0 Functional Description (Continued)

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

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

3.7 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 signalled by activating the $\overline{NMI}$ or $\overline{INT}$ 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, PSR and the MOD register contents on the interrupt stack and then it transfers control to an exception service procedure.

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

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

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

Refer to Section for details on the reset operation.

3.7.1 Exception Acknowledge Sequence

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

1) Adjustment of Registers.

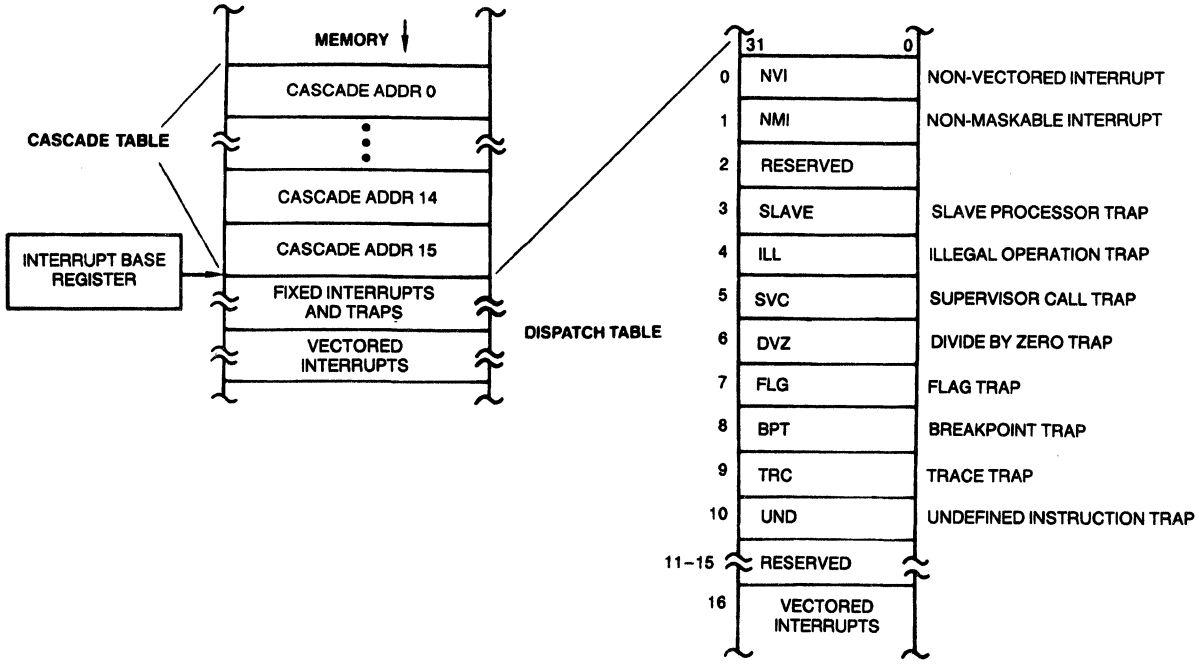
   Depending on the source of the exception, the CPU may restore and/or adjust the contents of the Program Counter (PC), the Processor Status Register (PSR) and the currently-selected Stack Pointer (SP). A copy of the PSR is made, and the PSR is then set to reflect Supervisor Mode and selection of the Interrupt 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 3-15.* A 32-bit External Procedure Descriptor is read from the table entry, and an External Procedure Call is performed using it. The MOD Register (16 bits) and Program Counter (32 bits) are pushed on the Interrupt Stack.



TL/EE/9424-21

FIGURE 3-15. Interrupt Dispatch and Cascade Tables

3.0 Functional Description (Continued)

This process is illustrated in *Figure 3-16*, from the viewpoint of the programmer.

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

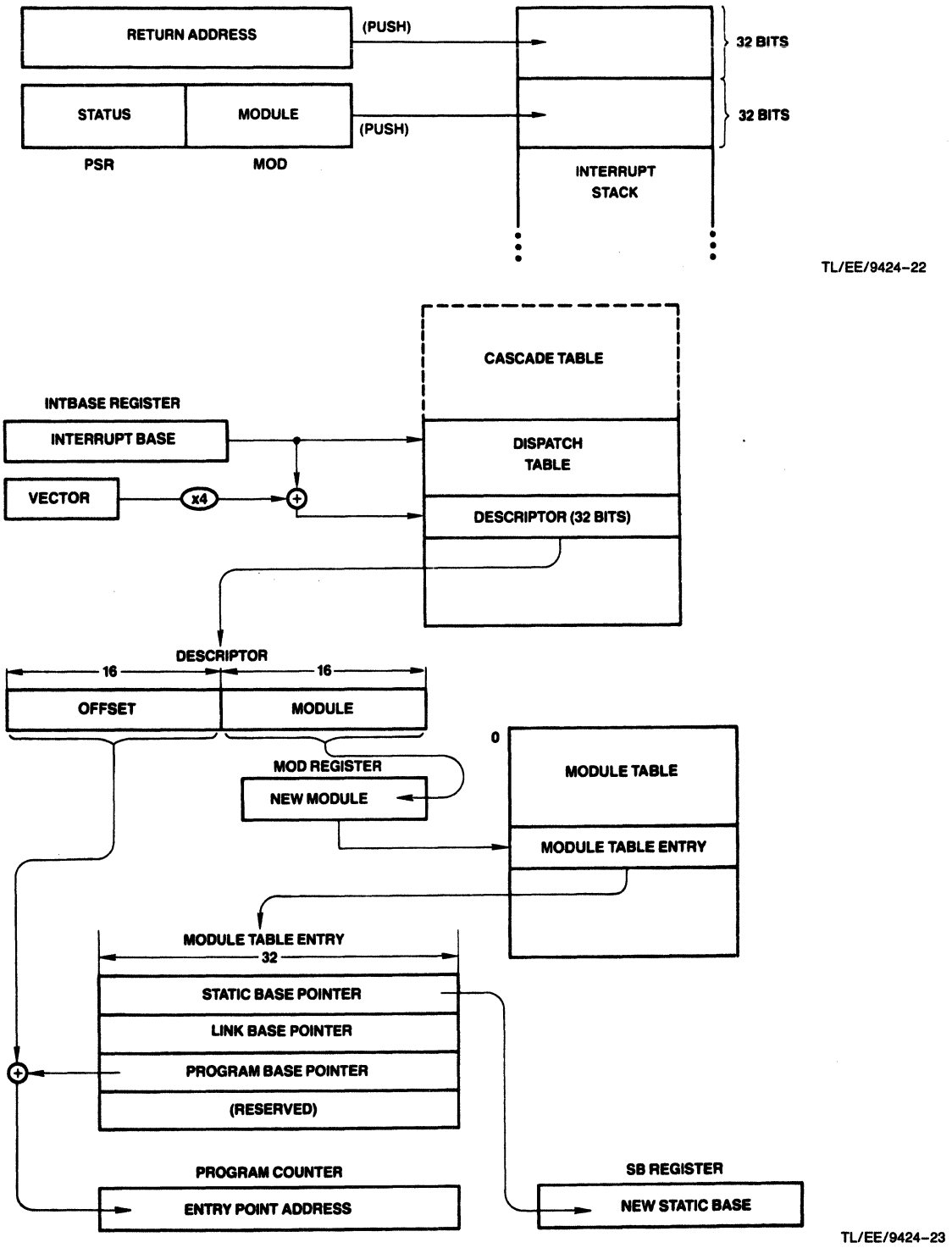


FIGURE 3-16. Exception Acknowledge Sequence

3.0 Functional Description (Continued)

3.7.2 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, MOD, PC and SB registers to their previous contents.

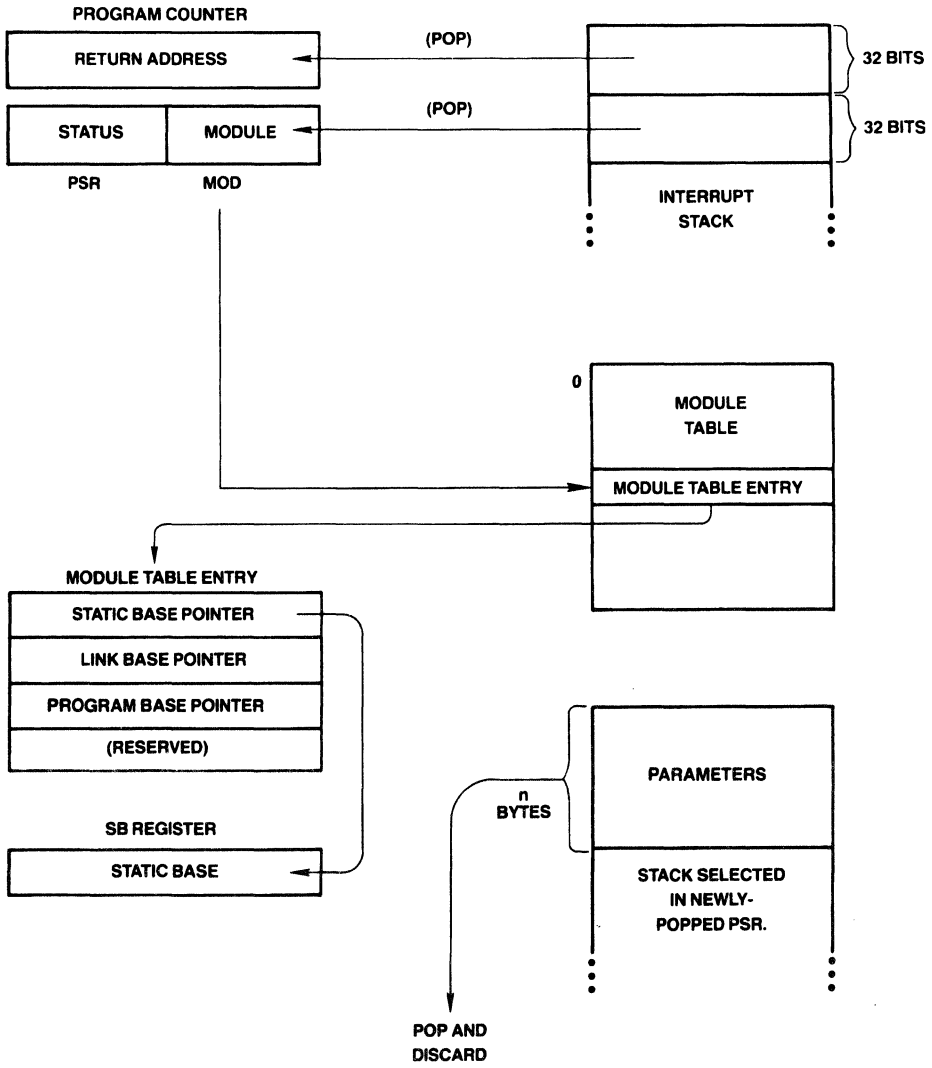
3.7.3 Maskable Interrupts

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

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

3.7.3.1 Non-Vectored Mode

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



TL/EE/9424-24

FIGURE 3-17. Return from Trap (RETT n) Instruction Flow

3.0 Functional Description (Continued)

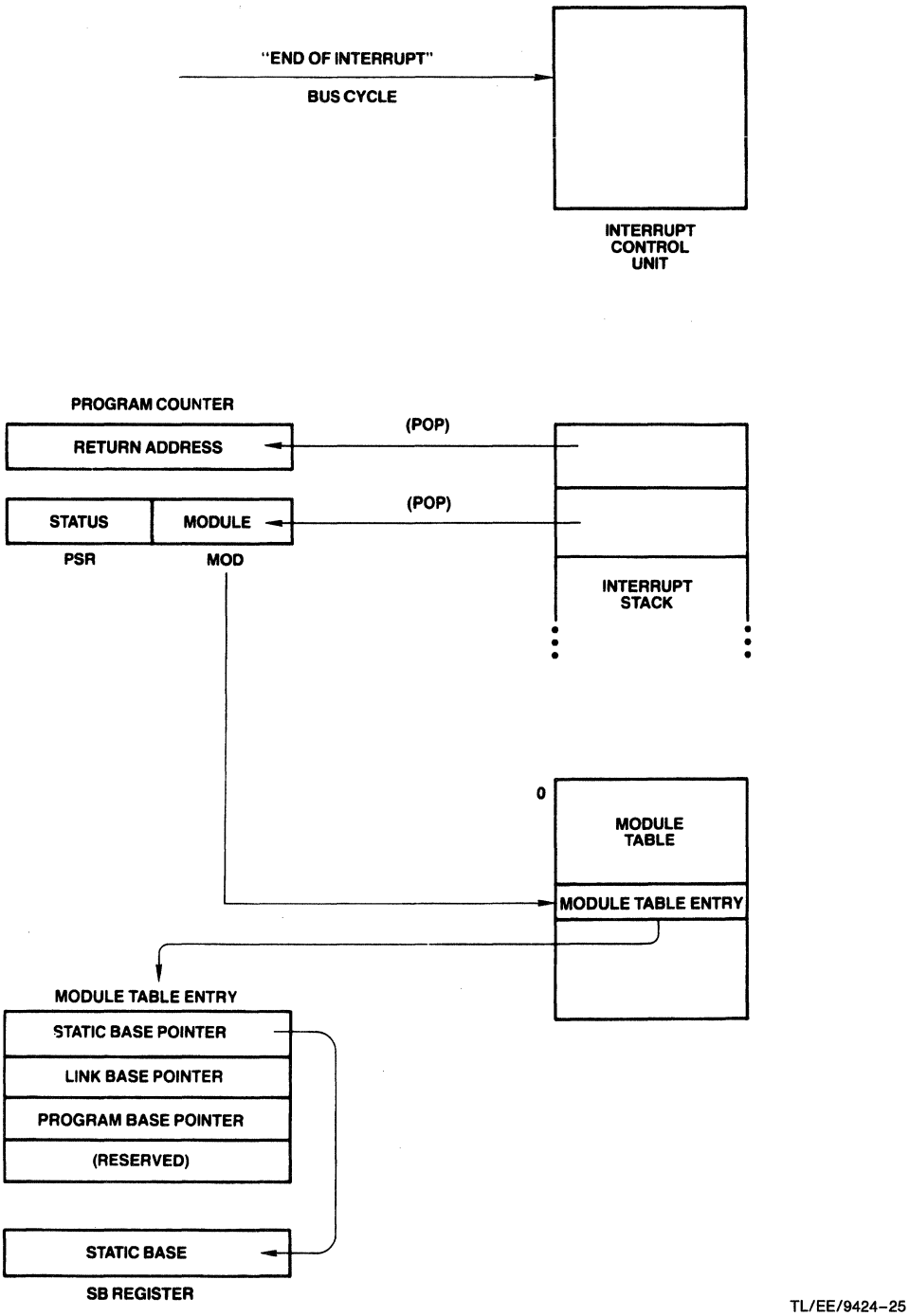


FIGURE 3-18. Return from Interrupt (RETI) Instruction Flow

3.7.3.2 Vectored Mode: Non-Cascaded Case

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

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

3.7.3.3 Vectored Mode: Cascaded Case

In order to allow up to 256 levels of interrupt, provision is made both in the CPU and in the NS32202 Interrupt Control

3.0 Functional Description (Continued)

Unit (ICU) to transparently support cascading. *Figure 3-20* shows a typical cascaded configuration. Note that the Interrupt output from a Cascaded ICU goes to an Interrupt Request input of the Master ICU, which is the only ICU which drives the CPU $\overline{INT}$ pin.

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

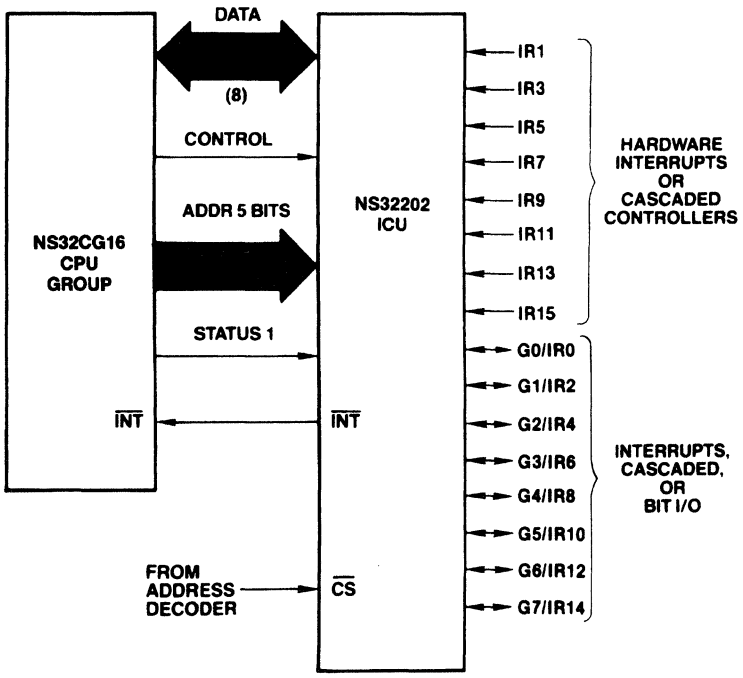
1) For each Cascaded ICU in the system, the Mater ICU must be informed of the line number (0 to 15) on which it receives the cascaded requests.
2) A Cascade Table must be established in memory. The Cascade Table is located in a NEGATIVE direction from the location indicated by the CPU Interrupt Base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

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

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

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

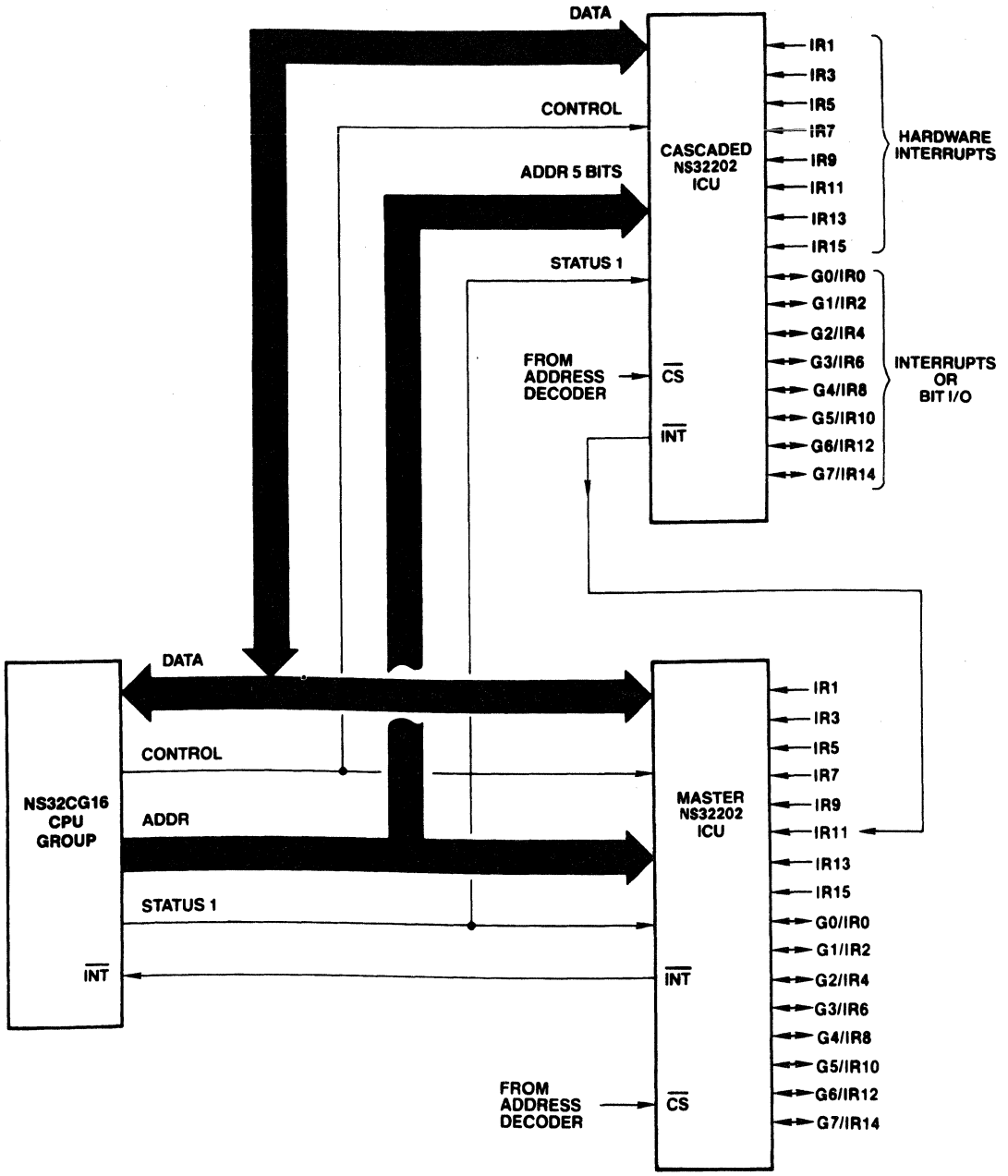
Note: If an interrupt must be masked off, the CPU can do so by setting the corresponding bit in the Interrupt Mask Register of the Interrupt Controller. However, if an interrupt is set pending during the CPU instruction that masks off that interrupt, the CPU may still perform an interrupt acknowledge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



TL/EE/9424–26

FIGURE 3-19. Interrupt Control Unit Connections (16 Levels)

3.0 Functional Description (Continued)



TL/EE/9424–27

FIGURE 3-20. Cascaded Interrupt Control Unit Connections

3.7.4 Non-Maskable Interrupt

The Non-Maskable Interrupt is triggered whenever a falling edge is detected on the $\overline{NMI}$ pin. The CPU performs an "Interrupt Acknowledge, Master" bus cycle when processing of this interrupt actually begins. The Interrupt Acknowledge cycle differs from that provided for Maskable Interrupts in that the address presented is $FFFF00_{16}$. 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.

For the full sequence of events in processing the Non-Maskable Interrupt, see Section 3.7.7.1.

3.7.5 Traps

Traps are processing exceptions that are generated as direct results of the execution of an instruction. The Return Address pushed by any trap except Trap (TRC) is the address of the first byte of the instruction during which the trap occurred. Traps do not disable interrupts, as they are not associated with external events. Traps recognized by NS32CG16 CPU are:

Trap (SLAVE): An exceptional condition was detected by the Floating Point Unit during the execution of a Slave Instruction. This trap is requested via the Status Word returned as part of the Slave Processor Protocol (Section 3.8.1).

3.0 Functional Description (Continued)

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 SLAVE trap is used for Floating Point division by zero.)

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

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

Trap (TRC): The instruction just completed is being traced. See Section 3.7.6.

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

3.7.6 Instruction Tracing

Instruction tracing is a 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.

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

In other cases, it is still possible to guarantee that a Trace Trap occurs at the end of the instruction, provided that special care is taken before returning from the Trace Trap Service Procedure. In case a BICPSRB instruction has been executed, the service procedure should make sure that the T bit in the PSR copy saved on the Interrupt Stack is set before executing the RETT instruction to return to the program begin 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 NS32CG16 instructions which have interior loops (BBOR, BBXOR, BBAND, BBFOR, EXTBLT, MOVMP, SBITPS, TBITS), special care must be taken with the single-step trap. If an interrupt occurs during a single-step of one of the graphics instructions, the interrupt will be serviced. Upon return from the interrupt service routine, the new NS32CG16 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 trapping 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 NS32CG16 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 NS32CG16 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 NS32CG16-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 instruction is executed, the Trap (TRC) occurs after the next interrupt, when the interrupt service procedure has returned.

3.7.7 Priority Among Exceptions

The NS32CG16 CPU internally prioritizes simultaneous interrupt and trap requests as follows:

1) Traps other than Trace (Highest priority)
2) Non-Maskable Interrupt
3) Maskable Interrupts
4) Trace Trap (Lowest priority)

3.7.8 Exception Acknowledge Sequences: Detail Flow

For purposes of the following detailed discussion of interrupt and trap acknowledge sequences, a single sequence called "Service" is defined in *Figure 3-21*. Upon detecting any interrupt request or trap condition, the CPU first performs a sequence dependent upon the type of interrupt or trap. This sequence will include pushing the Processor Status Register and establishing a Vector and a Return Address. The CPU then performs the Service sequence.

3.7.8.1 Maskable/Non-Maskable Interrupt Sequence

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

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

   Otherwise, set "Return Address" to the address of the next instruction.
2. Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits S, U, T, P and I.
3. If the interrupt is Non-Maskable:
   a. Read a byte from address $FFFF00_{16}$, applying Status Code 0100 (Interrupt Acknowledge, Master: Section 3.4.1). Discard the byte read.
   b. Set "Vector" to 1.
   c. Go to Step 8.

3.0 Functional Description (Continued)

4. If the interrupt is Non-Vectored:
   a. Read a byte from address $FFFF00_{16}$, applying Status Code 0100 (Interrupt Acknowledge, Master: Section 3.4.1). Discard the byte read.
   b. Set "Vector" to 0.
   c. Go to Step 8.
5. Here the interrupt is Vectored. Read "Byte" from address $FFFE00_{16}$, applying Status Code 0100 (Interrupt Acknowledge, Master: Section 3.4.1).
6. If "Byte" ≥ 0, then set "Vector" to "Byte" and go to Step 8.
7. If "Byte" is in the range −16 through −1, then the interrupt source is Cascaded. (More negative values are reserved for future use.) Perform the following:
   a. Read the 32-bit Cascade Address from memory. The address is calculated as INTBASE + 4* Byte.
   b. Read "Vector", applying the Cascade Address just read and Status Code 0101 (Interrupt Acknowledge, Cascaded: Section 3.4.1).
8. Push the PSR copy (from Step 2) onto the Interrupt Stack as a 16-bit value.
9. Perform Service (Vector, Return Address), *Figure 3-21.*

Service (Vector, Return Address):

1) Read the 32-bit External Procedure Descriptor from the Interrupt Dispatch Table: address is Vector*4+INTBASE Register contents.
2) Move the Module field of the Descriptor into the MOD Register.
3) Read the new Static Base pointer from the memory address contained in MOD, placing it into the SB Register.
4) Read the Program Base pointer from memory address MOD + 8, and add to it the Offset field from the Descriptor, placing the result in the Program Counter.
5) Flush Queue: Non-sequentially fetch first instruction of Interrupt Routine.
6) Push MOD Register onto the Interrupt Stack as a 16-bit value. (The PSR has already been pushed as a 16-bit value.)
7) Push the Return Address onto the Interrupt Stack as a 32-bit quantity.

FIGURE 3-21. Service Sequence
Invoked during All Interrupt/Trap Sequences

3.7.8.2 Trap Sequence: Traps Other Than Trace

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

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

3) Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits S, U, P and T.
4) Push the PSR copy onto the Interrupt Stack as a 16-bit value.
5) Set "Return Address" to the address of the first byte of the trapped instruction.
6) Perform Service (Vector, Return Address), *Figure 3-21.*

3.7.8.3 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) Push the PSR copy onto the Interrupt Stack as a 16-bit value.
4) Set "Vector" to 9.
5) Set "Return Address" to the address of the next instruction.
6) Perform Service (Vector, Return Address), *Figure 3-21.*

3.8 SLAVE PROCESSOR INSTRUCTIONS

The NS32CG16 supports only one group of instructions, the floating point instruction set, as being executable by a slave processor. The floating point instruction set is validated by the F bit in the CFG register.

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

3.8.1 Slave Processor Protocol

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

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

Upon receiving a Slave Processor instruction, the CPU initiates the sequence outlined in *Figure 3-22.* While applying Status Code 1111 (Broadcast ID, Section 3.4.1), the CPU transfers the ID Byte on the least-significant half of the Data Bus (AD0–AD7). All Slave Processors input this byte and decode it. The Slave Processor selected by the ID Byte is activated, and from this point the CPU is communicating only with it. If any other slave protocol was in progress (e.g., an aborted Slave instruction), this transfer cancels it.

The CPU next sends the Operation Word while applying Status Code 1101 (Transfer Slave Operand, Section 3.4.1). Upon receiving it, the Slave Processor decodes it, and at this point both the CPU and the Slave Processor are aware of the number of operands to be transferred and their sizes. The Operation Word is swapped on the Data Bus; that is, bits 0–7 appear on pins AD8–AD15 and bits 8–15 appear on pins AD0–AD7.

3.0 Functional Description (Continued)

Using the Addressing Mode fields within the Operation Word, the CPU starts fetching operands and issuing them to the Slave Processor. To do so, it references any Addressing Mode extensions which may be appended to the Slave Processor instruction. Since the CPU is solely responsible for memory accesses, these extensions are not sent to the Slave Processor. The Status Code applied is 1101 (Transfer Slave Processor Operand, Section 3.4.1).

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

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

FIGURE 3-22. Slave Processor Protocol

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

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

Upon receiving the pulse on $\overline{SPC}$, the CPU uses $\overline{SPC}$ to read a Status Word from the Slave Processor, applying Status Code 1110 (Read Slave Status). This word has the format shown in *Figure 3-23.* If the Q bit ("Quit", Bit 0) is set, this indicates that an error was detected by the Slave Processor. The CPU will not continue the protocol, but will immediately trap through the Slave vector in the Interrupt Table. Certain Slave Processor instructions cause CPU PSR bits to be loaded from the Status Word.

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

3.8.2 Floating Point Instructions

Table 3-5 gives the protocols followed for each Floating Point instruction. The instructions are referenced by their mnemonics. For the bit encodings of each instruction, see Appendix A.

TABLE 3-5. Floating Point Instruction Protocols

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

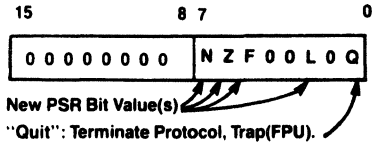
Note:
D = Double Word
i = integer size (B,W,D) specified in mnemonic.
f = Floating Point type (F,L) specified in mnemonic.
N/A = Not Applicable to this instruction.

3.0 Functional Description (Continued)

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

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

The Returned Value Type and Destination column gives the size of any returned value and where the CPU places it. The PSR Bits Affected column indicates which PSR bits, if any, are updated from the Slave Processor Status Word (*Figure 3-23).*



TL/EE/9424-28

FIGURE 3-23. Slave Processor Status Word Format

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

4.0 Device Specifications

4.1 NS32CG16 PIN DESCRIPTIONS

The following is a brief description of all NS32CG16 pins. The descriptions reference portions of the Functional Description, Section 3.

Unless otherwise indicated, reserved pins should be left open.

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

4.1.1 Supplies

V_{CCL} **Logic Power.**
+5V positive supply for on-chip logic.

VCCCTTL, VCCFCLK, VCCAD, VCCIO **Buffers Power.**
+5V positive supplies for on-chip output buffers.

VSSL **Logic Ground.**
Ground reference for on-chip logic.

VSSFCLK, VSSNTSC, VSSHAD, VSSLAD, VSSIO **Buffers Ground.**
Ground reference for on-chip output buffers.

4.1.2 Input Signals

$\overline{\text{RSTI}}$ **Reset Input.**
Schmitt triggered, asynchronous signal used to generate a CPU reset. See Section 3.3.

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

When $\overline{\text{RSTI}}$ changes right before the falling edge of CTTL, and meets the specified set-up time, it will be recognized on that falling edge. Otherwise it will be recognized on the falling edge of CTTL in the following clock cycle.

$\overline{\text{HOLD}}$ **Hold Request.**
When active, causes the CPU to release the bus for DMA or multiprocessing purposes. See Section 3.5.

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

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

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

Note:
If $\overline{\text{INT}}$ is from a asynchronous source, it should be synchronized with CTTL to minimize the possibility of metastable states.

$\overline{\text{NMI}}$ **Non-Maskable Interrupt.**
A High-to-Low transition on this signal requests a non-maskable interrupt

$\overline{\text{CWAIT}}$ **Continuous Wait.**
Causes the CPU to insert continuous wait states if sampled low at the end of T2 and each following T-State. See Section 3.4.3.

$\overline{\text{WAIT}}$1-2 **Two-Bit Wait State Inputs.**
These inputs, collectively called $\overline{\text{WAIT}}$1-2, allow from zero to three wait states to be specified. They are binary weighted. See Section 3.4.3.

Note:
During a DMAC cycle, $\overline{\text{WAIT}}$1-2 should be kept inactive to prevent loss of synchronization. Wait states, in this case, should be generated through $\overline{\text{CWAIT}}$.

OSCIN **Crystal/External Clock Input.**
Input from a crystal or an external clock source. See Section 3.2.

4.1.3 Output Signals

A16-A23 ***High-Order Address Bits.**
These are the most significant 8 bits of the memory address bus.

$\overline{\text{HBE}}$ ***High Byte Enable.**
Status signal used to enable data transfers on the most significant byte of the data bus.

4.0 Device Specifications (Continued)

ST0–3 **Status.**
Bus cycle status code; ST0 is the least significant. Encodings are:

0000—Idle: CPU Inactive on Bus.

0001—Idle: WAIT Instruction.

0010—(Reserved)

0011—Idle: Waiting for Slave.

0100—Interrupt Acknowledge, Master.

0101—Interrupt Acknowledge, Cascaded.

0110—End of Interrupt, Master.

0111—End of Interrupt, Cascaded.

1000—Sequential Instruction Fetch.

1001—Non-Sequential Instruction Fetch.

1010—Data Transfer.

1011—Read Read-Modify-Write Operand.

1100—Read for Effective Address.

1101—Transfer Slave Operand.

1110—Read Slave Status Word.

1111—Broadcast Slave ID.

U/$\overline{S}$ **User/Supervisor.**
User or Supervisor Mode status. High indicates User Mode; low indicates Supervisor Mode.

$\overline{ILO}$ **Interlocked Operation.**
When active, indicates that an interlocked operation is being executed.

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

$\overline{PFS}$ **Program Flow Status.**
A pulse on this signal indicates the beginning of execution of an instruction.

$\overline{BPU}$ **BPU Cycle.**
This signal is activated during a bus cycle to enable an external BITBLT processing unit. The EXTBLT instruction activates this signal.*

$\overline{RSTO}$ **Reset Output.**
This signal becomes active when $\overline{RSTI}$ is low, initiating a system reset.

$\overline{RD}$ **Read Strobe.**
Activated during CPU or DMAC read cycles to enable reading of data from memory or peripherals. See Section 3.4.2.

$\overline{WR}$ **Write Strobe.**
Activated during CPU or DMAC write cycles to enable writing of data to memory or peripherals.

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

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

$\overline{DBE}$ **Data Buffers Enable.**
Used to control external data buffers. It is active when the data buffers are to be enabled.

OSCOUT **Crystal Output.**
This line is used as the return path for the crystal (if used). It must be left open when an external clock source is used to drive OSCIN.

FCLK **Fast Clock.**
This clock is derived from the clock waveform on OSCIN. Its frequency is either the same as OSCIN or is lower, depending upon the scale factor programmed into the CFG register. See Section 3.2.1.

PHI1, PHI2 **Two-Phase Clock.**
These outputs provide a two-phase clock with frequency half that of FCLK. They can be used to clock the DP8510/DP8511 BPU. The trace lengths of PHI1 and PHI2 should be shorter than 4 inches (10 centimeters) when connected to the BPU.

CTTL **System Clock.**
This clock is similar to PHI1 but has a much higher driving capability. The skew between its rising edge and PHI1 rising edge is kept to a minimum.

4.1.4 Input-Output Signals

AD0–15 ***Address/Data Bus.**
Multiplexed Address/Data information. Bit 0 is the least significant bit of each.

$\overline{SPC}$ **Slave Processor Control.**
Used by the CPU as the data strobe output for slave processor transfers; used by a slave processor to acknowledge completion of a slave instruction. See Section 3.4.7.1.

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

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

4.0 Device Specifications (Continued)

4.2 ABSOLUTE MAXIMUM RATINGS

If Military/Aerospace specified devices are required, please contact the National Semiconductor Sales Office/Distributors for availability and specifications.

Temperature Under Bias	0°C to +70°C
Storage Temperature	−65°C to +150°C
All Input or Output Voltages with Respect to GND	−0.5V to +7V

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 ±5%, GND = 0V

Symbol	Parameter	Conditions	Min	Typ	Max	Units
V_{IH}	High Level Input Voltage	(Note 4)	2.0		V_{CC} + 0.5	V
V_{IL}	Low Level Input Voltage	(Note 3)	−0.5		0.8	V
V_{T+}	$\overline{RSTI}$ Rising Threshold Voltage	V_{CC} = 5.0V	2.5		3.5	V
V_{HYS}	$\overline{RSTI}$ Hysteresis Voltage	V_{CC} = 5.0V	0.8		1.8	V
V_{XL}	OSCIN Input Low Voltage				0.3	V
V_{XH}	OSCIN Input High Voltage		V_{CC} − 0.3			V
V_{OH}	Low Level Output Voltage	I_{OH} = −400 μA	0.90 V_{CC}			V
V_{OL}	Low Level Output Voltage	I_{OL} = 4 mA			0.10 V_{CC}	V
I_{ILS}	$\overline{SPC}$ Input Current (low)	V_{IN} = 0.4V, $\overline{SPC}$ in Input Mode	0.05		1.0	mA
I_I	Input Load Current	0 ≤ V_{IN} ≤ V_{CC}, All Inputs except $\overline{SPC}$	−20		20	μA
I_L	Leakage Current Output and I/O Pins in TRI-STATE Input Mode	0.4 ≤ V_{OUT} ≤ V_{CC}	−20		20	μA
I_{CC}	Active Supply Current	I_{OUT} = 0, T_A = 25°C (Note 2)		140	200	mA

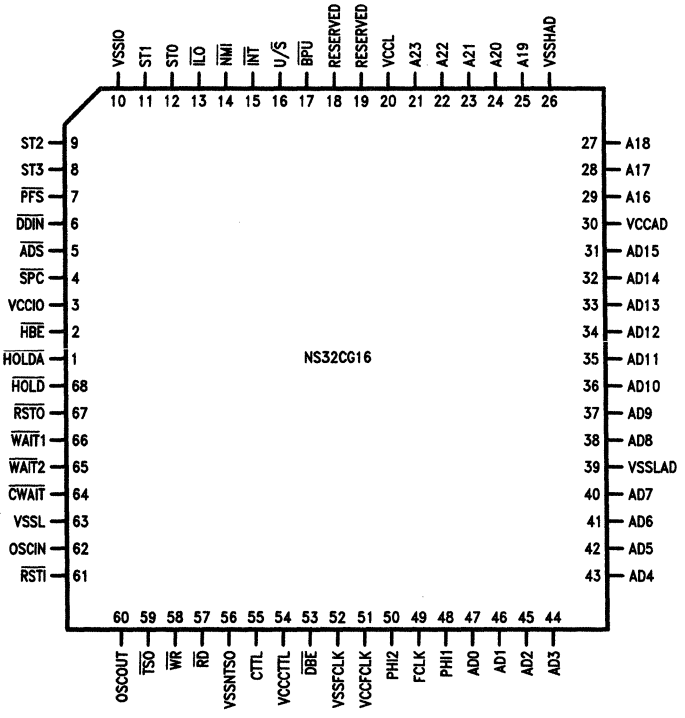
Note 1: Care should be taken by designers to provide a minimum inductance path between the V_{SS} pins and system ground in order to minimize noise.

Note 2: I_{CC} is affected by the clock scaling factor selected by the C and M bits in the CFG register, see Section 3.2.1.

Note 3: $V_{IL\ min}$—in the range of −0.5V to −1.5V, the pulse must be ≤ 20 ns, and the period between pulses ≥ 120 ns.

Note 4: $V_{IH\ max}$—in the range of V_{CC} + 0.5V to V_{CC} + 2.0V, the pulse must be ≤ 25 ns, and the period between pulses ≥ 120 ns.

68-Pin PCC Package



TL/EE/9424-29

Bottom View

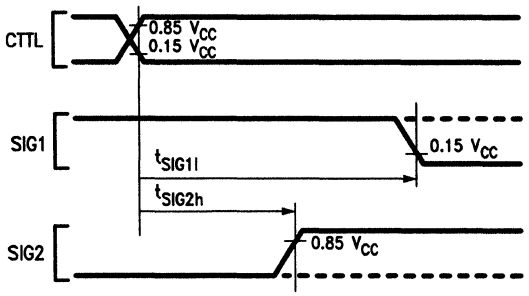
FIGURE 4-1. Connection Diagram

4.0 Device Specifications (Continued)

4.4 SWITCHING CHARACTERISTICS

4.4.1 Definitions

All the timing specifications given in this section refer to 15% or 85% of V_{CC} on the rising or falling edges of CTTL, and all output signals; and to 0.8V or 2.0V on all the TTL input signals as illustrated in *Figures 4-2* and *4-3* unless specifically stated otherwise.



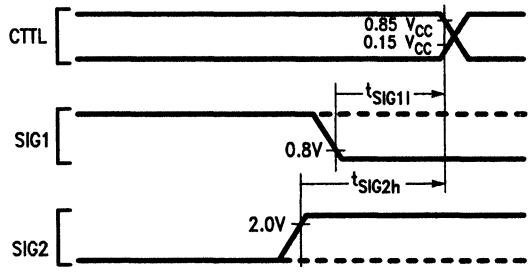
TL/EE/9424–30

FIGURE 4-2. Timing Specification Standard (CMOS Output Signals)

ABBREVIATIONS:

L.E. — leading edge R.E. — rising edge

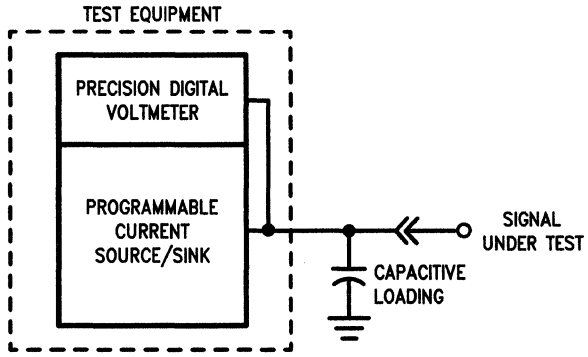
T.E. — trailing edge F.E. — falling edge



TL/EE/9424–31

FIGURE 4-3. Timing Specification Standard (TTL Input Signals)

4.4.2 DEVICE TESTING



TL/EE/9424–65

FIGURE 4.4. Test Loading Configuration

TABLE 4-1. Test Loading Characteristics

Signal Name	Capacitive Loading	High Level Output Voltage (I_{OH} = −400 μA)	Low Level Output Voltage (I_{OL} = 4 mA)	Input Load Current (0 ≤ V_{IN} ≤ V_{CC})	High Level Input Voltage	Low Level Input Voltage
$\overline{HBE}$, ST0–3, U/$\overline{S}$, $\overline{ILO}$, $\overline{HLDA}$, $\overline{PFS}$, $\overline{BPU}$, $\overline{RST0}$, $\overline{RD}$, $\overline{WR}$, $\overline{TSO}$, $\overline{DBE}$, $\overline{OSCOUT}$, FCLK, $\overline{DDIN}$, $\overline{ADS}$	50 pF	≥ 0.90 V_{CC}	≤ 0.10 V_{CC}			
$\overline{RSTI}$, $\overline{HOLD}$, $\overline{INT}$, $\overline{NMI}$, $\overline{CWAIT}$, $\overline{WAIT1–2}$	50 pF			−20 μA ≤ I_I ≤ 20 μA	2.0V ≤ V_{IH} ≤ V_{CC} + 0.5V	−0.5V ≤ V_{IL} ≤ 0.8V
OSCIN	50 pF			−20 μA ≤ I_I ≤ 20 μA	V_{IH} ≥ V_{CC} − 0.3V	V_{IL} ≤ 0.3V
AD0–15, A16–23, CTTL	100 pF	V_{OH} ≥ 0.90 V_{CC}	V_{OL} ≤ 0.10 V_{CC}			
AD0–15	100 pF			−20 μA ≤ I_I ≤ 20 μA	2.0V ≤ V_{IH} ≤ V_{CC} + 0.5V	−0.5V ≤ V_{IL} ≤ 0.8V
PHI1, PHI2	30 pF	V_{OH} ≥ 0.90 V_{CC}	V_{OL} ≤ 0.10 V_{CC}			
$\overline{SPC}$	30 pF	V_{OH} ≥ 0.90 V_{CC}	V_{OL} ≤ 0.10 V_{CC}	50 μA ≤ I_I ≤ 1.0 mA	2.0V ≤ V_{IH} ≤ V_{CC} + 0.5V	V_{IL} = 0.4V

4.0 Device Specifications (Continued)

4.4.3 Timing Tables

4.4.3.1 Output Signals: Internal Propagation Delays, NS32CG16-10 and NS32CG16-15

Name	Figure	Description	Reference/Conditions	NS32CG16-10		NS32CG16-15		Units
				Min	Max	Min	Max	
t_{CTp}	4-20	CTTL Clock Period	R.E., CTTL to Next R.E., CTTL	100	1000	66	1000	ns
t_{CTh}	4-20	CTTL High Time At 1.5V (Both Edges) (See Note 1)	100 pF Capacitive Load	0.5 t_{CTp} − 10 ns	0.5 t_{CTp} + 7 ns	0.5 t_{CTp} −0 ns	0.5 t_{CTp} + 5 ns	
			75 pF Capacitive Load			0.5 t_{CTp} −1 ns	0.5 t_{CTp} +5 ns	
			50 pF Capacitive Load			0.5 t_{CTp} −2 ns	0.5 t_{CTp} +5 ns	
			25 pF Capacitive Load			0.5 t_{CTp} −3 ns	0.5 t_{CTp} +5 ns	
t_{CTl}	4-20	CTTL Low Time	At 0.8V	0.5 t_{CTp} − 8 ns	0.5 t_{CTp} + 6 ns	0.5 t_{CTp} − 6 ns	0.5 t_{CTp} + 2 ns	
t_{CTr}	4-20	CTTL Rise Time	15% to 85% V_{CC} on R.E., CTTL	2	8	2	6	ns
t_{CTf}	4-20	CTTL Fall Time	85% to 15% V_{CC} on F.E., CTTL	2	8	2	6	ns
$t_{CLw(1,2)}$	4-20	PHI1, PHI2 Pulse Width	At 2.0V on PHI1, PHI2 (Both Edges)	0.5 t_{CTp} − 10 ns	0.5 t_{CTp} + 5 ns	0.5 t_{CTp} − 6 ns	0.5 t_{CTp} + 2 ns	
t_{CLh}	4-20	Clock High Time	At 90% V_{CC} on PHI1, PHI2 (Both Edges)	0.5 t_{CTp} − 15 ns	0.5 t_{CTp}	0.5 t_{CTp} − 10 ns	0.5 t_{CTp}	
$t_{nOVL(1,2)}$	4-20	PHI1, PHI2, Non-Overlap Time	At 50% V_{CC} on PHI1, PHI2	3		3		ns
t_{XFr}	4-20	OSCIN to FCLK R.E. Delay	80% V_{CC} on R.E., OSCIN to R.E., FCLK	6	29	2	25	ns
t_{FCr}	4-20	FCLK to CTTL R.E. Delay	R.E., FCLK to R.E., CTTL	0	6	0	6	ns
t_{FCf}	4-20	FCLK to CTTL F.E. Delay	R.E., FCLK to F.E., CTTL	−1	4	−1	4	ns
t_{FPr}	4-20	FCLK to PHI1 R.E. Delay	R.E., FCLK to R.E., PHI1	0	6	0	6	ns
t_{FPf}	4-20	FCLK to PHI1 F.E. Delay	R.E., FCLK to F.E., PHI1	−5	2	−4	2	ns
t_{PCr}	4-20	CTTL and PHI1 Skew	R.E., CTTL to R.E., PHI1	−3	3	−2	2	ns
t_{ALv}	4-5	Address Bits 0–15 Valid	after R.E., CTTL T1		40	10	30	ns
t_{ALh}	4-5	Address Bits 0–15 Hold	after R.E., CTTL T2	5		5		ns
t_{AHv}	4-5	Address Bits 16–23 Valid	after R.E., CTTL T1		40	10	30	ns
t_{AHh}	4-5	Address Bits 16–23 Hold	after R.E., CTTL Next T1 or Ti	5		0		ns
t_{ALnfr}	4-5	AD0–AD15 Active (See Note 2)	Non-Float after R.E., CTTL T1	4	36	4	26	

Note 1: Device testing is performed using the Test Loading Characteristics in Table 4.1. Additional timing data for CTTL with various capacitive loads is not 100% tested.

Note 2: t_{ALnfr} is address bits 0–15 not floating or active after R.E. CTTL T1. This is only valid if the previous CPU cycle was a read *(Figure 4.5)*. A previous write may have "data" active into T1 of the next cycle which then becomes "address" during T1.

4.0 Device Specifications (Continued)

4.4.3.1 Output Signals: Internal Propagation Delays, NS32CG16-10 and NS32CG16-15 (Continued)

Name	Figure	Description	Reference/Conditions	NS32CG16-10		NS32CG16-15		Units
				Min	Max	Min	Max	
t_{ALf}	4-7	AD0–AD15 Floating (Caused by $\overline{HOLD}$)	after R.E., CTTL Ti		25		18	ns
t_{AHf}	4-7	A16–A23 Floating	after R.E., CTTL Ti		25		18	ns
t_{ALnf}	4-5, 4-8	Address Bits 0–15 Not Floating	after R.E., CTTL T1	4	36	4	26	ns
t_{AHnf}	4-8	Address Bits 16–23 Not Floating	after R.E., CTTL T4	4	36	4	26	ns
t_{Dv}	4-6, 4-10	Data Valid (Write Cycle)	after R.E., CTTL T2 or T1		50		38	ns
t_{Dh}	4-6, 4-10	Data Hold	after R.E., CTTL Next T1 or Ti	0		0		ns
t_{ADSa}	4-5	$\overline{ADS}$ Signal Active	after R.E., CTTL T1	5	35	5	26	ns
t_{ADSia}	4-5	$\overline{ADS}$ Signal Inactive	after F.E., CTTL T1	5	45	5	25	ns
t_{ADSw}	4-6	$\overline{ADS}$ Pulse Width	at 15% V_{CC} (Both Edges)	30		25		ns
t_{ADSf}	4-7	$\overline{ADS}$ Floating	after R.E., CTTL Ti		55		40	ns
t_{ADSr}	4-8	$\overline{ADS}$ Return from Floating	after R.E., CTTL Ti		55		40	ns
t_{ALADSs}	4-6	Address Bits 0–15 Setup	before $\overline{ADS}$ T.E.	25		20		ns
t_{AHADSs}	4-6	Address Bits 16–23 Setup	before $\overline{ADS}$ T.E.	25		20		ns
t_{ALADSh}	4-5	Address Bits 0–15 Hold	after $\overline{ADS}$ T.E.	15		12		ns
t_{HBEv}	4-5	$\overline{HBE}$ Signal Valid	after R.E., CTTL T1		70		38	ns
t_{HBEh}	4-5	$\overline{HBE}$ Signal Hold	after R.E., CTTL Next T1 or Ti	0		0		ns
t_{HBEf}	4-7	$\overline{HBE}$ Signal Floating	after R.E., CTTL Ti		55		40	ns
t_{HBEr}	4-8	$\overline{HBE}$ Return from Floating	after R.E., CTTL Ti		55		40	ns
t_{DDINv}	4-5	$\overline{DDIN}$ Signal Valid	after R.E., CTTL T1		65		38	ns
t_{DDINh}	4-5	$\overline{DDIN}$ Signal Hold	after R.E., CTTL Next T1 or Ti	0		0		ns
t_{DDINf}	4-7	$\overline{DDIN}$ Floating	after R.E., CTTL Ti		55		40	ns
t_{DDINr}	4-8	$\overline{DDIN}$ Return from Floating	after R.E., CTTL Ti		55		40	ns
t_{SPCa}	4-10	$\overline{SPC}$ Output Active	after R.E., CTTL T1		35	5	26	ns
t_{SPCia}	4-10	$\overline{SPC}$ Output Inactive	after R.E., CTTL T4		35	5	26	ns
t_{SPCnf}	4-12	$\overline{SPC}$ Output Non-Forcing	after T.E., CTTL T4		10		8	ns
t_{HLDAa}	4-7	$\overline{HLDA}$ Signal Active	after R.E., CTTL Ti		50		26	ns
t_{HLDAia}	4-8	$\overline{HLDA}$ Signal Inactive	after R.E., CTTL Ti		50		26	ns
t_{STv}	4-5	Status ST0–ST3 Valid	after R.E., CTTL T4 (before T1, see Note 1)		45		38	ns
t_{STh}	4-5	Status ST0–ST3 Hold	after R.E., CTTL T4	0		0		ns
t_{BPUv}	4-5	$\overline{BPU}$ Signal Valid	after R.E., CTTL T4		45		30	ns
t_{BPUh}	4-5	$\overline{BPU}$ Signal Hold	after R.E., CTTL T4	10		6		ns

Note 1: Every memory cycle starts with T4, during which Cycle Status is applied. If the CPU was idling, the sequence will be: " . . . Ti, T4, T1 . . . ". If the CPU was not idling, the sequence will be: " . . . T4, T1 . . . ".

Note 2: If the CPU is connected directly to the FPU and the CTTL loading is not violated, the CPU and FPU will function correctly together. The CPU and FPU connect directly without buffers. They should be located less than 4 inches (10 centimeters) apart. t_{SPCa} and t_{SPCia} will track each other on all CPU's and therefore it is not possible to have a minimum t_{SPCia} and a maximum t_{SPCa} value. The pulse width minimum, t_{SPCw}, of the FPU will not be violated by the NS32CG16 when connected directly to the FPU.

4.0 Device Specifications (Continued)

4.4.3.1 Output Signals: Internal Propagation Delays, NS32CG16-10 and NS32CG16-15 (Continued)

Name	Figure	Description	Reference/Conditions	NS32CG16-10		NS32CG16-15		Units
				Min	Max	Min	Max	
t_{TSOa}	4-5	$\overline{TSO}$ Signal Active	after R.E., CTTL T2		12	2	10	ns
t_{TSOia}	4-5	$\overline{TSO}$ Signal Inactive	after R.E., CTTL T4		12	0	10	ns
t_{RDa}	4-5	$\overline{RD}$ Signal Active	after R.E., CTTL T2		20		15	ns
t_{RDia}	4-5	$\overline{RD}$ Signal Inactive	after R.E., CTTL T4		20	2	15	ns
t_{WRa}	4-6	$\overline{WR}$ Signal Active	after R.E., CTTL T2		20		15	ns
t_{WRia}	4-6	$\overline{WR}$ Signal Inactive	after R.E., CTTL T4		20	2	15	ns
$t_{DBEa(R)}$	4-5	$\overline{DBE}$ Active (Read Cycle)	after F.E., CTTL T2		21		15	ns
$t_{DBEa(W)}$	4-6	$\overline{DBE}$ Active (Write Cycle)	after R.E., CTTL T2		28		15	ns
t_{DBEia}	4-5, 4-6	$\overline{DBE}$ Inactive	after F.E., CTTL T4		23		15	ns
t_{USv}	4-5	U/$\overline{S}$ Signal Valid	after R.E., CTTL T4		40		30	ns
t_{USh}	4-5	U/$\overline{S}$ Signal Hold	after R.E., CTTL T4	10		6		ns
t_{PFSa}	4-13	$\overline{PFS}$ Signal Active	after F.E., CTTL		50		38	ns
t_{PFSia}	4-13	$\overline{PFS}$ Signal Inactive	after F.E., CTTL		50		38	ns
t_{PFSw}	4-13	$\overline{PFS}$ Pulse Width	at 15% V_{CC} (Both Edges)	70		45		ns
t_{NSPF}	4-16	Nonsequential Fetch to Next $\overline{PFS}$ Clock Cycle	after R.E., CTTL T1	4		4		t_{CTp}
t_{PFNS}	4-15	$\overline{PFS}$ Clock Cycle to Next Nonsequential Fetch	before R.E., CTTL T1	4		4		t_{CTp}
t_{LXPF}	4-14	Last Operand Transfer of an Instruction to Next $\overline{PFS}$ Clock Cycle	before R.E., CTTL T1 of First Bus Cycle of Transfer	0		0		t_{CTp}
t_{ILOs}	4-17	$\overline{ILO}$ Signal Setup	before R.E., CTTL T1 of First Interlocked Write Cycle	30		30		ns
t_{ILOh}	4-18	$\overline{ILO}$ Signal Hold	after R.E., CTTL T3 of Last Interlocked Read Cycle	10		7		ns
t_{ILOa}	4-19	$\overline{ILO}$ Signal Active	after R.E., CTTL		55		35	ns
t_{ILOia}	4-19	$\overline{ILO}$ Signal Inactive	after R.E., CTTL		55		35	ns
t_{RSTOa}	4-22	$\overline{RSTO}$ Signal Active	after R.E., CTTL		21		15	ns
t_{RSTOia}	4-22	$\overline{RSTO}$ Signal Inactive	after R.E., CTTL		21		15	ns
t_{RTOI}	4-22	Reset to Idle	after F.E. of RSTO		10		10	t_{CTp}
t_{RTOF}	4-22	Reset to Fetch	after R.E. of RSTO		8		8	t_{CTp}

4.0 Device Specifications (Continued)

4.4.3.2 Input Signal Requirements: NS32CG16-10 and NS32CG16-15

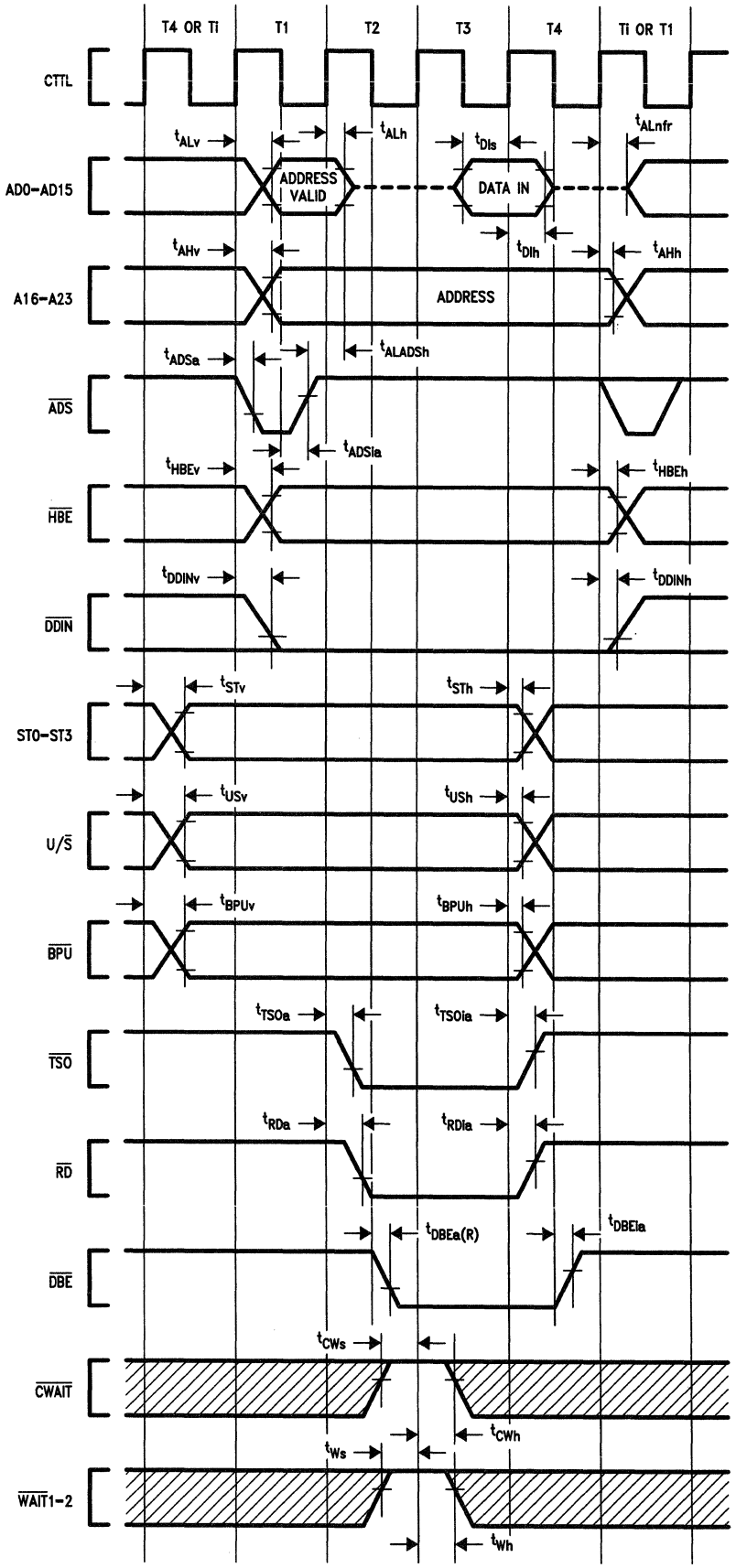
Name	Figure	Description	Reference/Conditions	NS32CG16-10		NS32CG16-15		Units
				Min	Max	Min	Max	
t_{Xp}	4-20	OSCIN Clock Period	R.E., OSCIN to Next R.E., OSCIN	50	500	33	500	ns
t_{Xh}	4-20	OSCIN High Time (External Clock)	at 80% V_{CC} (Both Edges)	16		10		ns
t_{Xl}	4-20	OSCIN Low Time	at 20% V_{CC} (Both Edges)	16		10		ns
t_{DIs}	4-5, 4-11	Data In Setup	before R.E., CTTL T4	18		15		ns
t_{DIh}	4-5, 4-11	Data In Hold (see Note 1)	after R.E., CTTL T4	7		7		ns
t_{CWs}	4-5, 4-6	$\overline{CWAIT}$ Signal Setup	before R.E., CTTL T3 or T3(w)	20		20		ns
t_{CWh}	4-5, 4-6	$\overline{CWAIT}$ Signal Hold	after R.E., CTTL T3 or T3(w)	5		5		ns
t_{Ws}	4-5, 4-6	$\overline{WAIT}$n Signals Setup	before R.E., CTTL T3 or T3(w)	20		20		ns
t_{Wh}	4-5, 4-6	$\overline{WAIT}$n Signals Hold	after R.E., CTTL T3 or T3(w)	5		5		ns
t_{HLDs}	4-7, 4-8	$\overline{HOLD}$ Setup Time	before R.E., CTTL TX2 or Ti	30		22		ns
t_{HLDh}	4-7, 4-8	$\overline{HOLD}$ Hold Time	after R.E., CTTL Ti	0		0		ns
t_{PWR}	4-21	Power Stable to $\overline{RSTI}$ R.E.	after V_{CC} Reaches 4.5V	50		33		μs
t_{RSTs}	4-21, 4-22	$\overline{RSTI}$ Signal Setup	before F.E., CTTL	20		20		ns
t_{RSTw}	4-22	$\overline{RSTI}$ Pulse Width	at 0.8V (Both Edges)	64		64		t_{CTp}
t_{INTs}	4-23	$\overline{INT}$ Signal Setup	before F.E., CTTL	14		14	t_{CTp} − 2 ns	ns
t_{INTh}	4-23	$\overline{INT}$ Signal Hold	after Interrupt Acknowledge		8		8	t_{CTp}
t_{NMIw}	4-24	$\overline{NMI}$ Pulse Width	at 0.8V (Both Edges)	70		50		ns
t_{SPCd}	4-12	$\overline{SPC}$ Pulse Delay from Slave	after F.E., CTTL T4	13		10		ns
t_{SPCs}	4-12	$\overline{SPC}$ Input Setup	before F.E., CTTL	37		30		ns
t_{SPCw}	4-12	$\overline{SPC}$ Pulse Width (from Slave Processor)	at 0.8V (Both Edges)	20		20		ns
t_{ADSs}	4-9	$\overline{ADS}$ Input Setup	before F.E., CTTL	15		10		ns
t_{ADSh}	4-9	$\overline{ADS}$ Input Hold (see Note 2)	after F.E., CTTL T1	10		10		ns
t_{DDINs}	4-9	$\overline{DDIN}$ Input Setup	before F.E., CTTL	15		10		ns
t_{DDINh}	4-9	$\overline{DDIN}$ Input Hold	after R.E., CTTL T4	7		5		ns

Note 1: t_{DIh} is always less than or equal to t_{RDia}.

Note 2: $\overline{ADS}$ must be deasserted before state T4 of the DMA controller cycle.

4.0 Device Specifications (Continued)

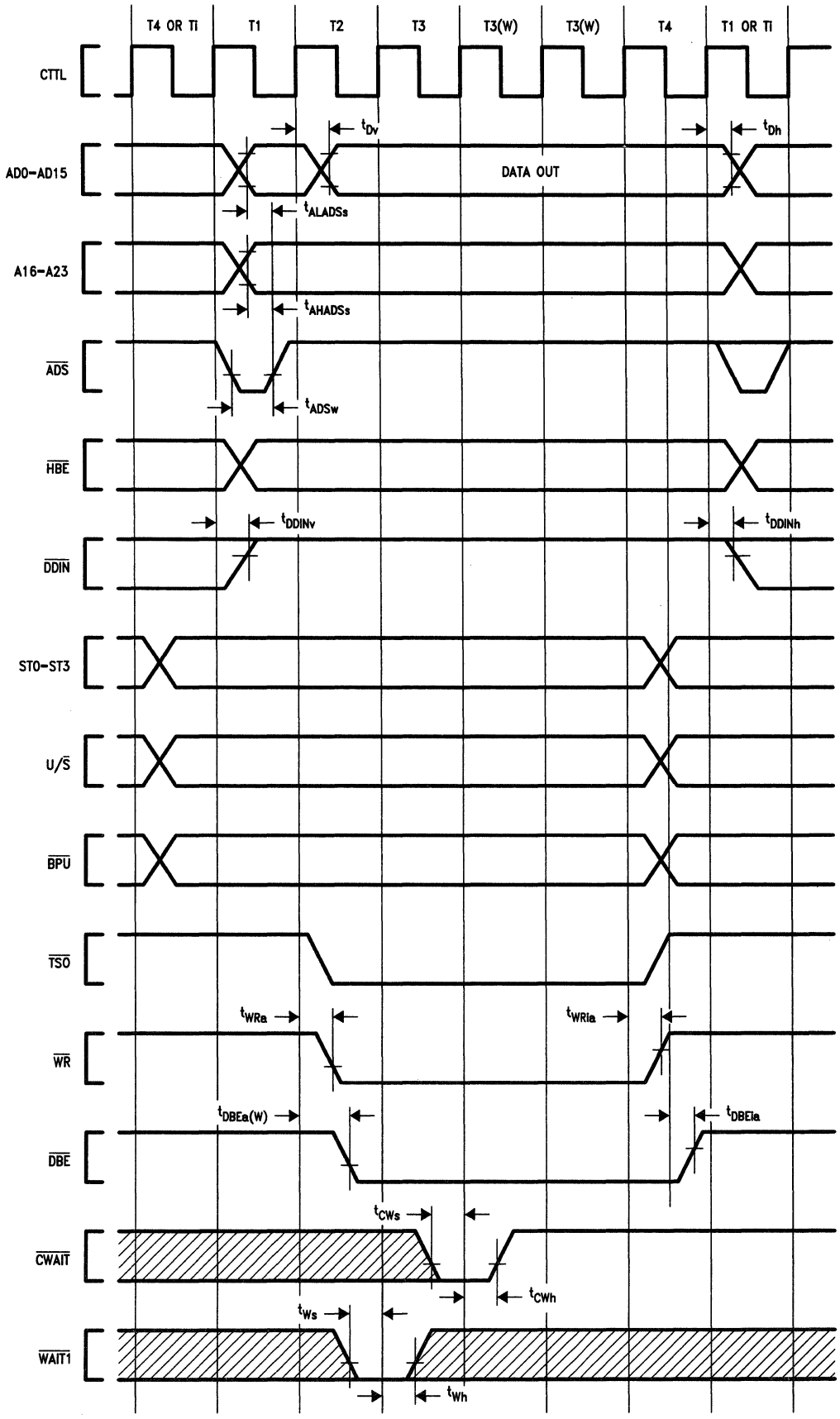
4.4.4 TIMING DIAGRAMS



TL/EE/9424–32

FIGURE 4-5. Read Cycle

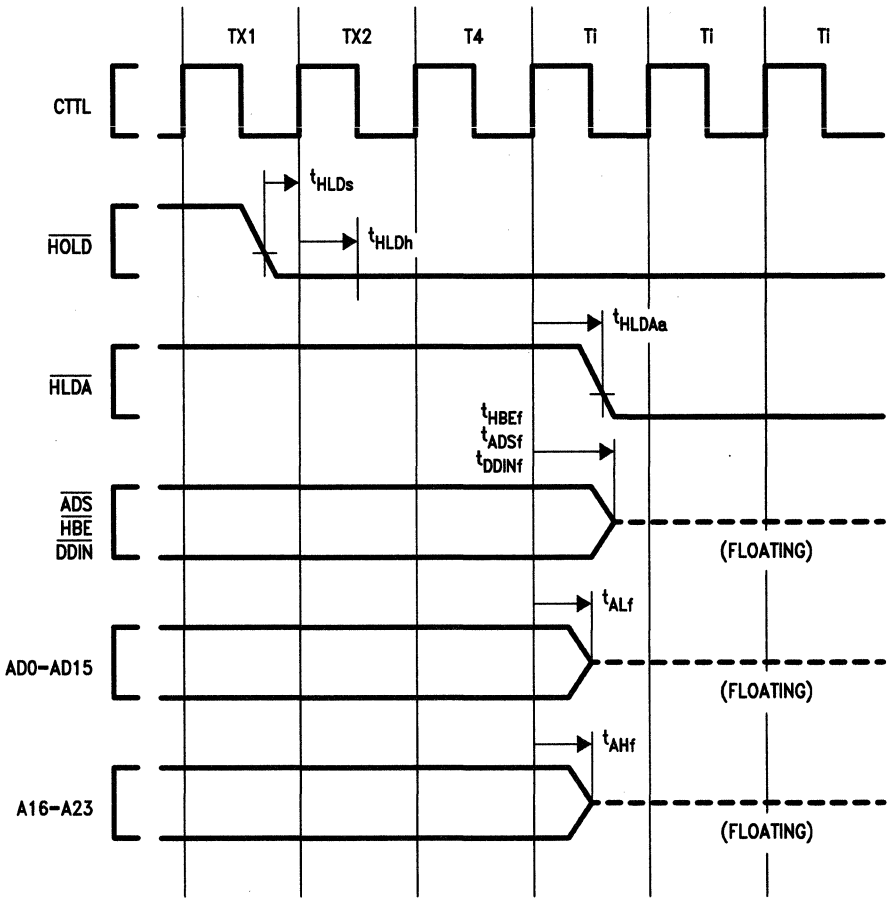
4.0 Device Specifications (Continued)



TL/EE/9424–33

FIGURE 4-6. Write Cycle

4.0 Device Specifications (Continued)

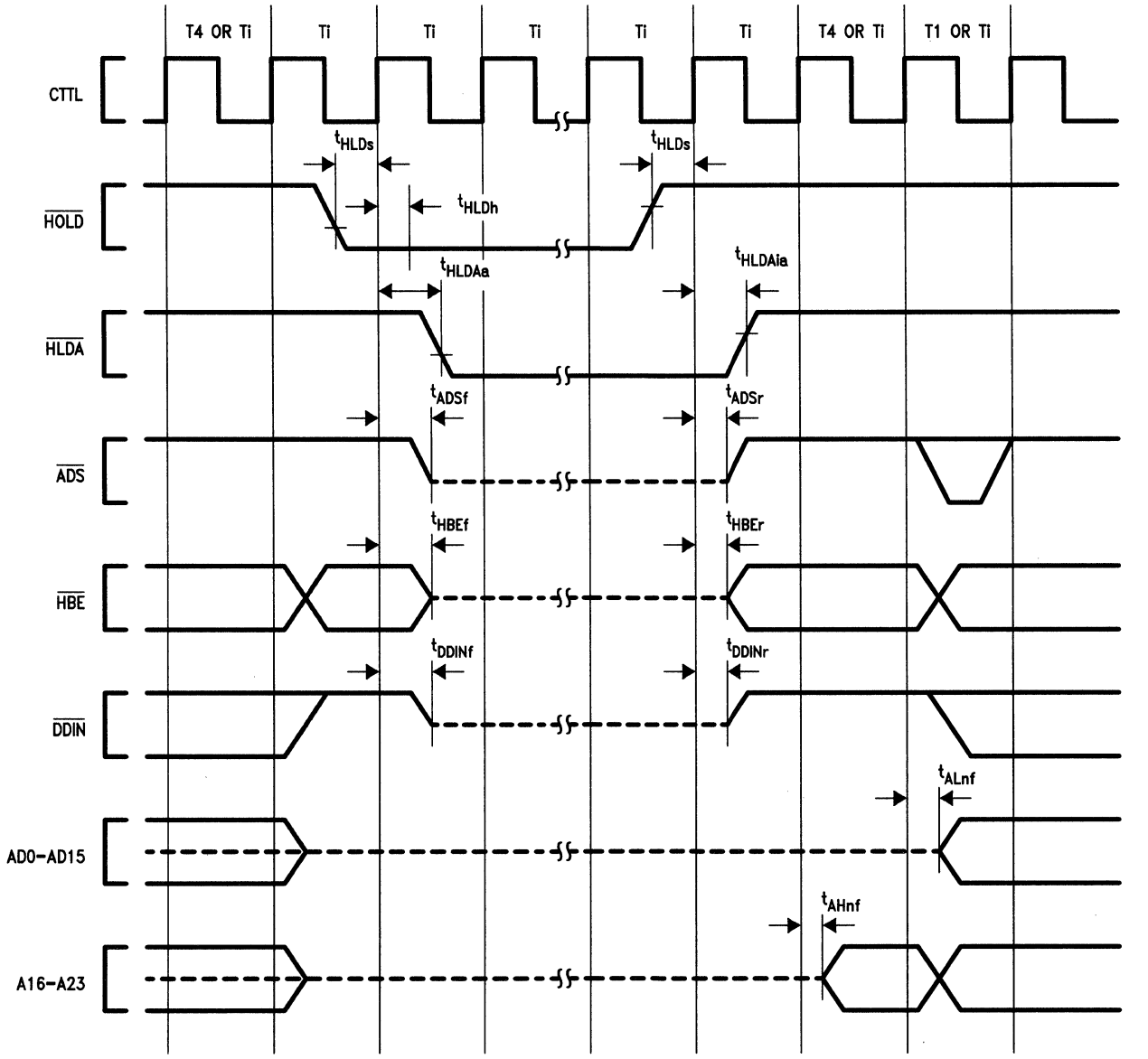


TL/EE/9424–34

FIGURE 4-7. HOLD Acknowledge Timing (Bus Initially Not Idle)

Note: When the bus is not idle, HOLD must be asserted before the rising edge of CTTL of the timing state that precedes state T4 in order for the request to be acknowledged.

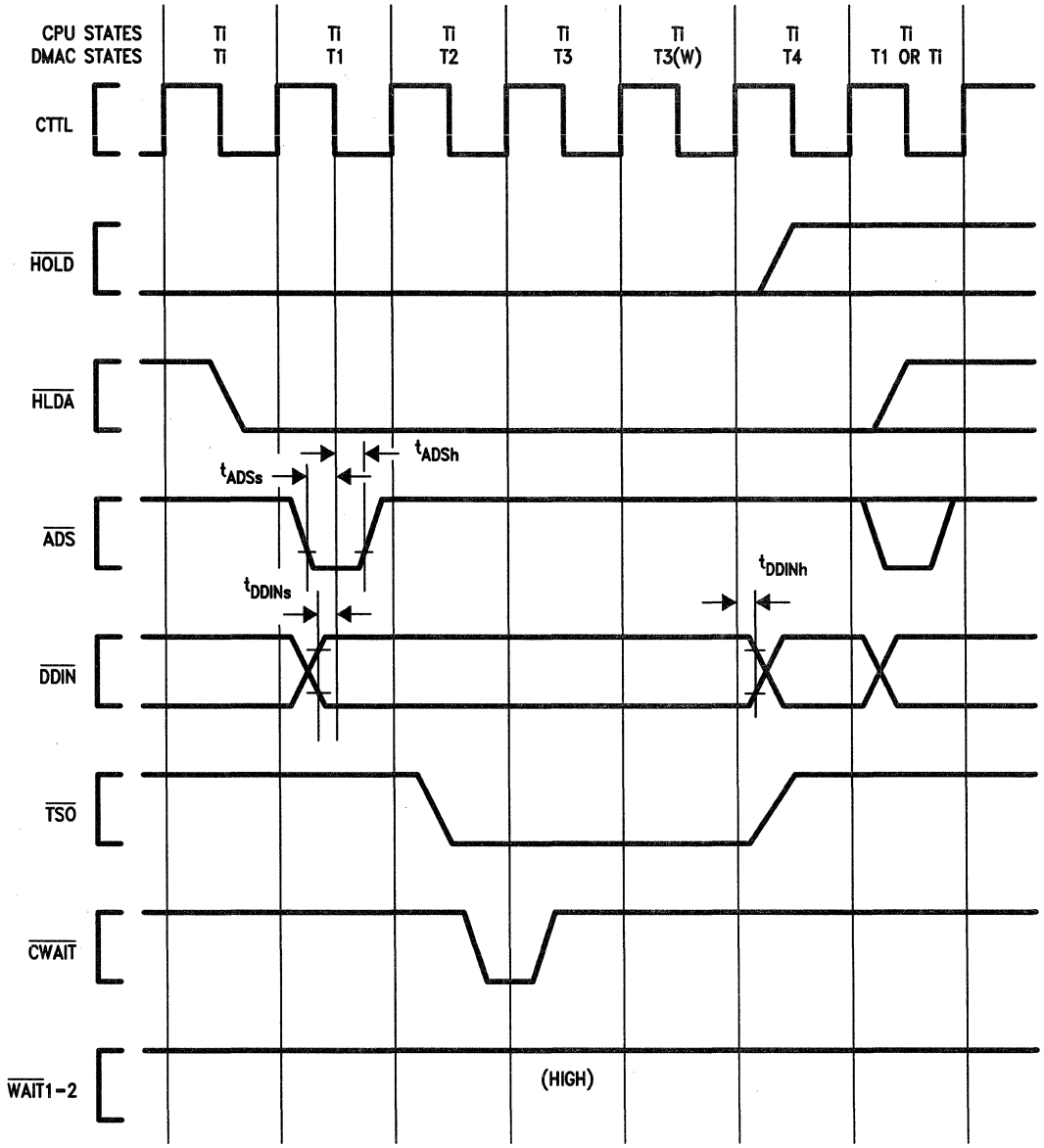
4.0 Device Specifications (Continued)



TL/EE/9424-35

FIGURE 4-8. HOLD Timing (Bus Initially Idle)

4.0 Device Specifications (Continued)



TL/EE/9424-36

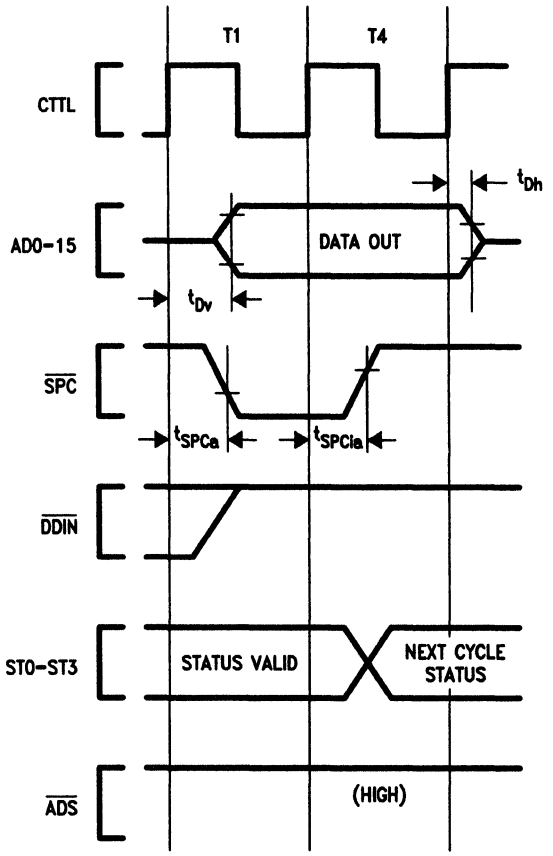
FIGURE 4-9. DMAC Initiated Bus Cycle

Note 1: $\overline{ADS}$ must be deactivated before state T4 of the DMA controller cycle.

Note 2: During a DMAC cycle $\overline{WAIT}1-2$ must be kept inactive to prevent loss of synchronization. A DMAC cycle is similar to a CPU cycle. The NS32CG16 generates $\overline{TSO}$, $\overline{RD}$, $\overline{WR}$ and $\overline{DBE}$. The DMAC drives the address/data lines $\overline{HBE}$, $\overline{ADS}$ and $\overline{DDIN}$.

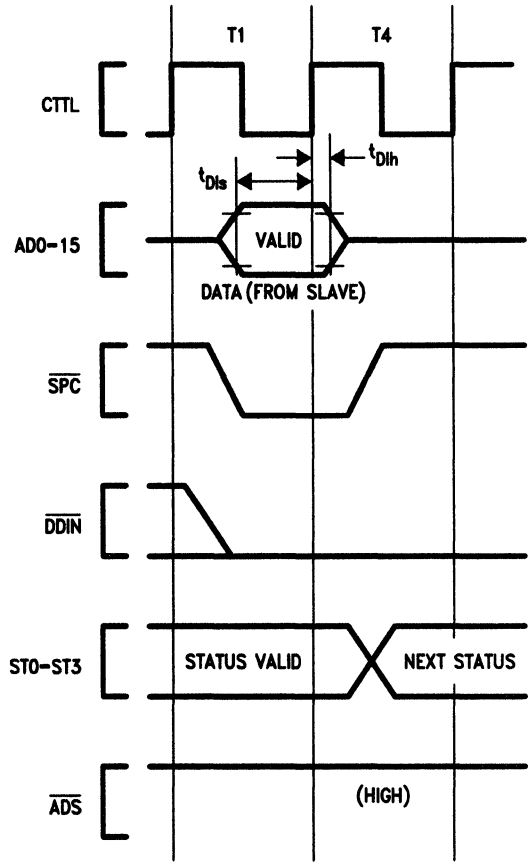
Note 3: During a DMAC cycle, if the $\overline{ADS}$ signal is pulsed in order to initiate a bus cycle, the $\overline{HOLD}$ signal must remain asserted until state T4 of the DMAC cycle.

4.0 Device Specifications (Continued)



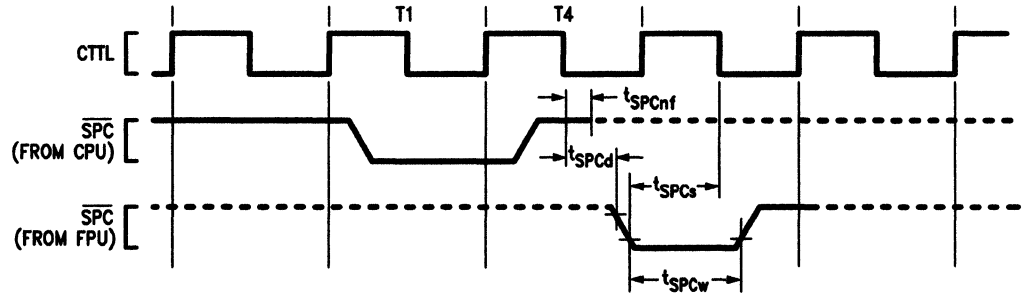
TL/EE/9424-37

FIGURE 4-10. Slave Processor Write Timing



TL/EE/9424-38

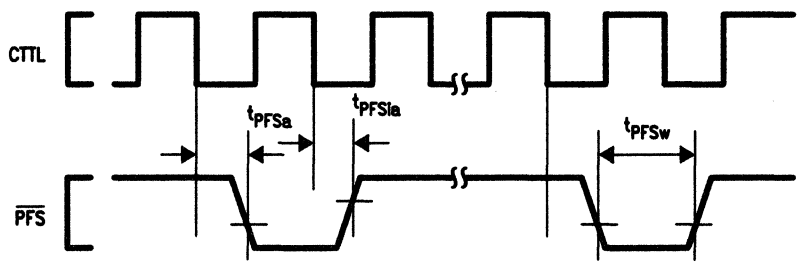
FIGURE 4-11. Slave Processor Read Timing



TL/EE/9424-39

FIGURE 4-12. $\overline{SPC}$ Timing

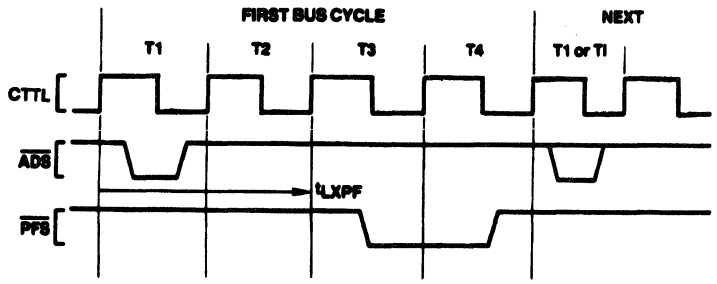
After transferring the last operand to the FPU, the CPU turns OFF the output driver and holds $\overline{SPC}$ high with an internal 5 kΩ pullup.



TL/EE/9424-40

FIGURE 4-13. Relationship of $\overline{PFS}$ to Clock Cycles

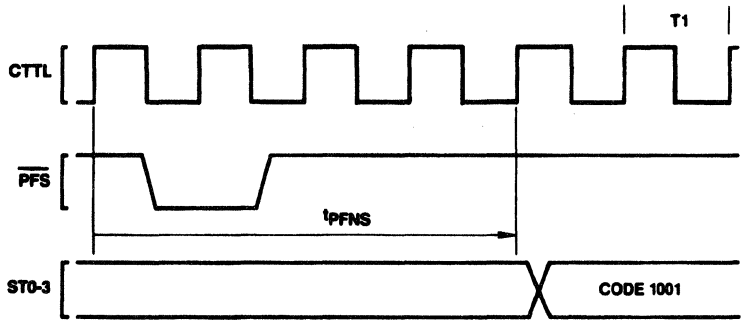
4.0 Device Specifications (Continued)



TL/EE/9424-41

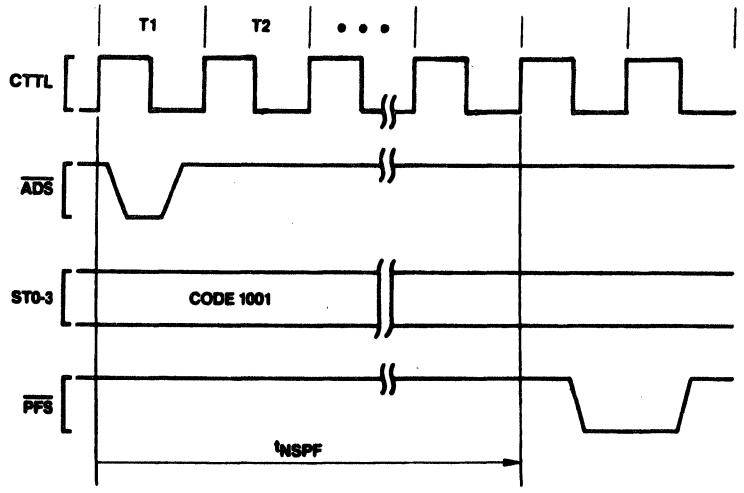
Note: In a transfer of a Read-Modify-Write type operand, this is the Read transfer, displaying RMW Status (Code 1011).

FIGURE 4-14. Relationship Between Last Data Transfer of an Instruction and $\overline{PFS}$ Pulse of Next Instruction



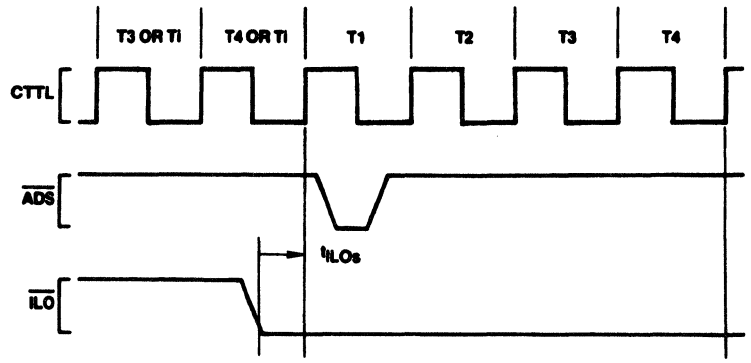
TL/EE/9424-42

FIGURE 4-15. Guaranteed Delay, $\overline{PFS}$ to Non-Sequential Fetch



TL/EE/9424-43

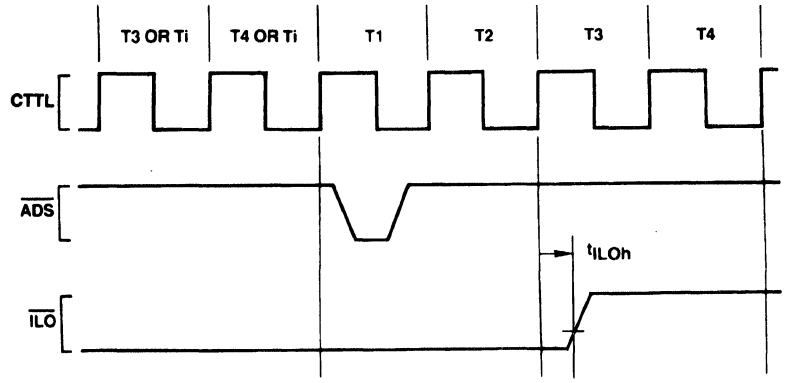
FIGURE 4-16. Guaranteed Delay, Non-Sequential Fetch to $\overline{PFS}$



TL/EE/9424-44

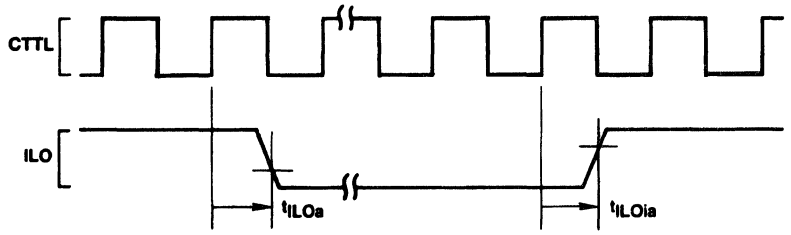
FIGURE 4-17. Relationship of $\overline{ILO}$ to First Operand Cycle of an Interlocked Instruction

4.0 Device Specifications (Continued)



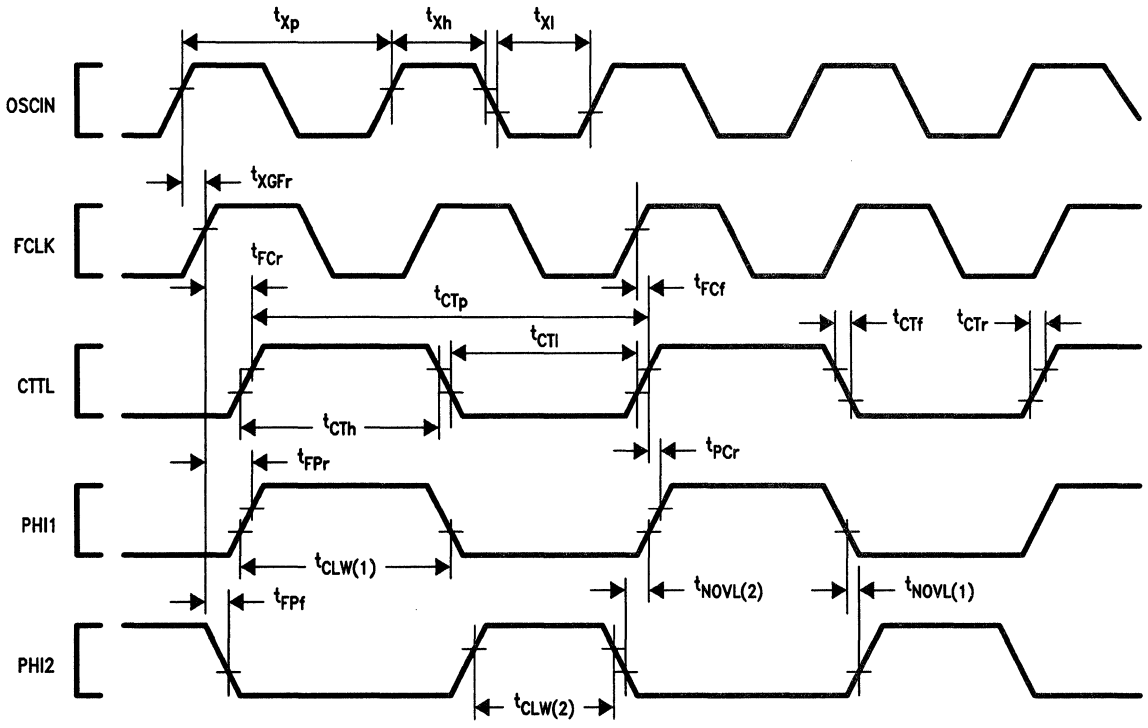
TL/EE/9424-45

FIGURE 4-18. Relationship of $\overline{ILO}$ to Last Operand Cycle of an Interlocked Instruction



TL/EE/9424-46

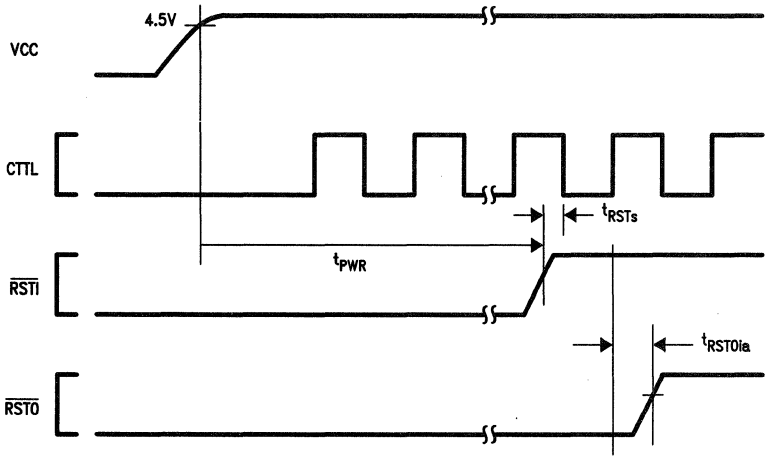
FIGURE 4-19. Relationship of $\overline{ILO}$ to Any Clock Cycle



TL/EE/9424-47

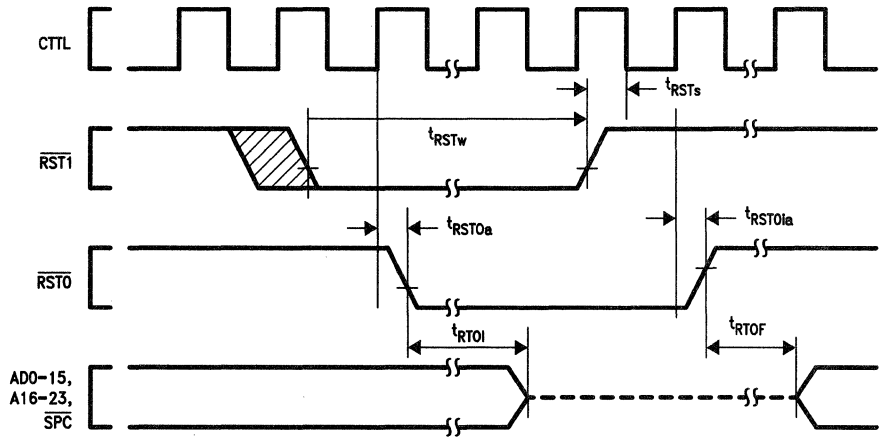
FIGURE 4-20. Clock Waveforms

4.0 Device Specifications (Continued)



TL/EE/9424–48

FIGURE 4-21. Power-On Reset

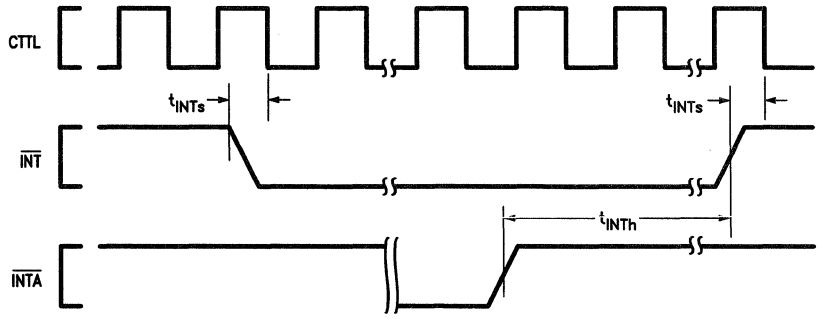


TL/EE/9424–49

FIGURE 4-22. Non-Power-On Reset

Note 1: During Reset the $\overline{HOLD}$ signal must be kept high.

Note 2: After $\overline{RSTI}$ is deasserted the first bus cycle will be an instruction fetch at address zero.

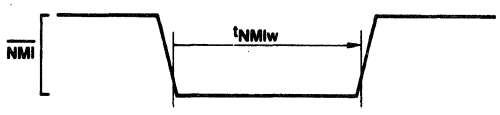


TL/EE/9424–50

FIGURE 4-23. $\overline{INT}$ Interrupt Signal Detection

Note 1: Once $\overline{INT}$ is asserted, it must remain asserted until it is acknowledged.

Note 2: $\overline{INTA}$ is the Interrupt Acknowledge bus cycle (not a CPU signal). Refer to Section 3.4.1 and Table 3.4.



TL/EE/9424–51

FIGURE 4-24. $\overline{NMI}$ Interrupt Signal Timing

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)

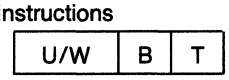
op = Operation Code
- Valid encodings shown with each format.

gen, gen 1, gen 2 = General Addressing Mode Field
- See Sec. 2.2 for encodings.

reg = General Purpose Register Number

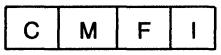
cond = Condition Code Field
- 0000 = EQual: Z = 1
- 0001 = Not Equal: Z = 0
- 0010 = Carry Set: C = 1
- 0011 = Carry Clear: C = 0
- 0100 = Higher: L = 1
- 0101 = Lower or Same: L = 0
- 0110 = Greater Than: N = 1
- 0111 = Less or Equal: N = 0
- 1000 = Flag Set: F = 1
- 1001 = Flag Clear: F = 0
- 1010 = 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)

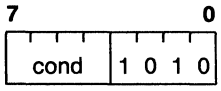
short = Short Immediate value. May contain
- quick: Signed 4-bit value, in MOVQ, ADDQ, CMPQ, ACB.
- cond: Condition Code (above), in Scond.
- areg: CPU Dedicated Register, in LPR, SPR.
  - 0000 = US
  - 0001 – 0111 = (Reserved)
  - 1000 = FP
  - 1001 = SP
  - 1010 = SB
  - 1011 = (Reserved)
  - 1100 = (Reserved)
  - 1101 = PSR
  - 1110 = INTBASE
  - 1111 = MOD
- Options: in String Instructions



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

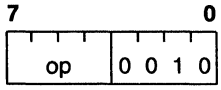
Configuration bits in SETCFG instruction:





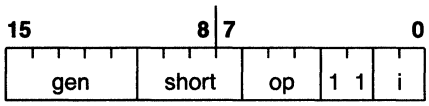
Format 0

Bcond (BR)



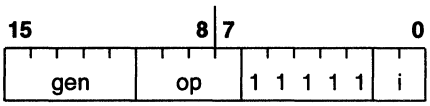
Format 1

BSR	–0000	ENTER	–1000
RET	–0001	EXIT	–1001
CXP	–0010	NOP	–1010
RXP	–0011	WAIT	–1011
RETT	–0100	DIA	–1100
RETI	–0101	FLAG	–1101
SAVE	–0110	SVC	–1110
RESTORE	–0111	BPT	–1111



Format 2

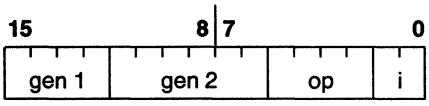
ADDQ	–000	ACB	–100
CMPQ	–001	MOVQ	–101
SPR	–010	LPR	–110
Scond	–011		



Format 3

CXPD	–0000	ADJSP	–1010
BICPSR	–0010	JSR	–1100
JUMP	–0100	CASE	–1110
BISPSR	–0110		

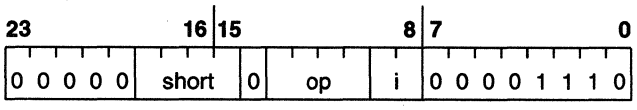
Trap (UND) on XXX1, 1000



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

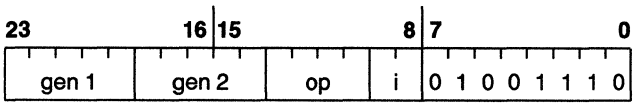
Appendix A: Instruction Formats (Continued)



Format 5

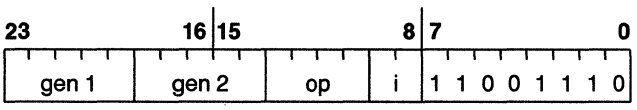
MOVS	−0000	BITWT	−1000
CMPS	−0001	TBITS	−1001
SETCFG	−0010	BBAND	−1010
SKPS	−0011	SBITPS	−1011
BBSTOD	−0100	BBFOR	−1100
EXTBLT	−0101	SBITS	−1101
BBOR	−0110	BBXOR	−1110
MOVMP	−0111		

No Operation on 1111



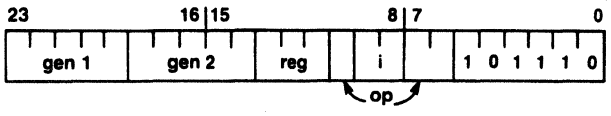
Format 6

ROT	−0000	NEG	−1000
ASH	−0001	NOT	−1001
CBIT	−0010	Trap (UND)	−1010
CBITI	−0011	SUBP	−1011
Trap (UND)	−0100	ABS	−1100
LSH	−0101	COM	−1101
SBIT	−0110	IBIT	−1110
SBITI	−0111	ADDP	−1111



Format 7

MOVM	−0000	MUL	−1000
CMPM	−0001	MEI	−1001
INSS	−0010	Trap (UND)	−1010
EXTS	−0011	DEI	−1011
MOVXBW	−0100	QUO	−1100
MOVZBW	−0101	REM	−1101
MOVZiD	−0110	MOD	−1110
MOVXiD	−0111	DIV	−1111

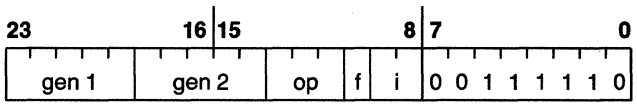


TL/EE/9424-52

Format 8

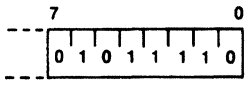
EXT	−0 00	INDEX	−1 00
CVTP	−0 01	FFS	−1 01
INS	−0 10		
CHECK	−0 11		

Trap (UND) on −1 10 and −1 11



Format 9

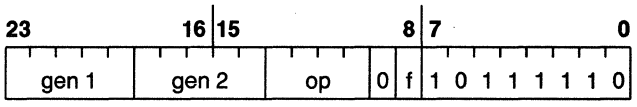
MOVif	−000	ROUND	−100
LFSR	−001	TRUNC	−101
MOVLF	−010	SFSR	−110
MOVFL	−011	FLOOR	−111



TL/EE/9424-53

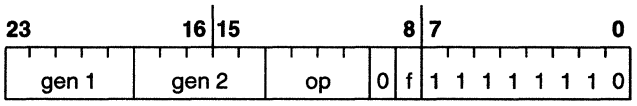
Format 10

Trap (UND) Always



Format 11

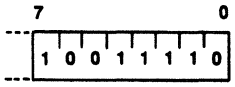
ADDf	−0000	DIVf	−1000
MOVf	−0001	(Note 1)	−1001
CMPf	−0010	Trap (UND)	−1010
(Note 3)	−0011	Trap (UND)	−1011
SUBf	−0100	MULf	−1100
NEGf	−0101	ABSf	−1101
Trap (UND)	−0110	Trap (UND)	−1110
Trap (UND)	−0111	Trap (UND)	−1111



Format 12

(Note 2)	−0000	(Note 2)	−1000
(Note 1)	−0001	(Note 1)	−1001
POLYf	−0010	Trap (UND)	−1010
DOTf	−0011	Trap (UND)	−1011
SCALBf	−0100	(Note 2)	−1100
LOGBf	−0101	(Note 1)	−1101
Trap (UND)	−0110	Trap (UND)	−1110
Trap (UND)	−0111	Trap (UND)	−1111

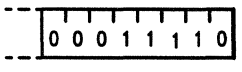
*Instructions with Format 12 are available only when the NS32381 is used.



TL/EE/9424-54

Format 13

Trap (UND) Always

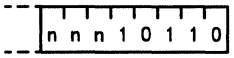


TL/EE/9424-55

Appendix A: Instruction Formats (Continued)

Format 14

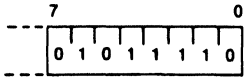
Trap (UND) Always



TL/EE/9424–56

Format 15

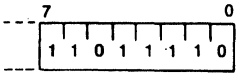
Trap (UND) Always



TL/EE/9424–57

Format 16

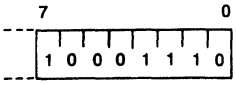
Trap (UND) Always



TL/EE/9424–58

Format 17

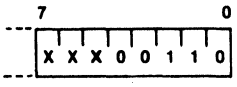
Trap (UND) Always



TL/EE/9424–59

Format 18

Trap (UND) Always

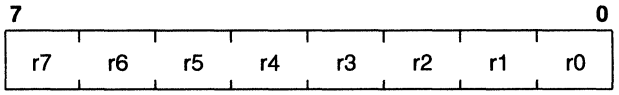


TL/EE/9424–60

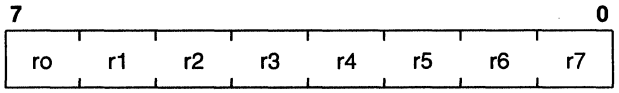
Format 19

Trap (UND) Always

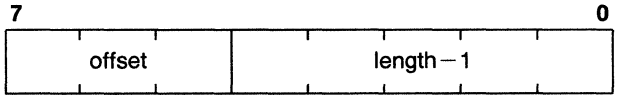
Implied Immediate Encodings:



Register Mask, appended to SAVE, ENTER



Register Mask, appended to RESTORE, EXIT



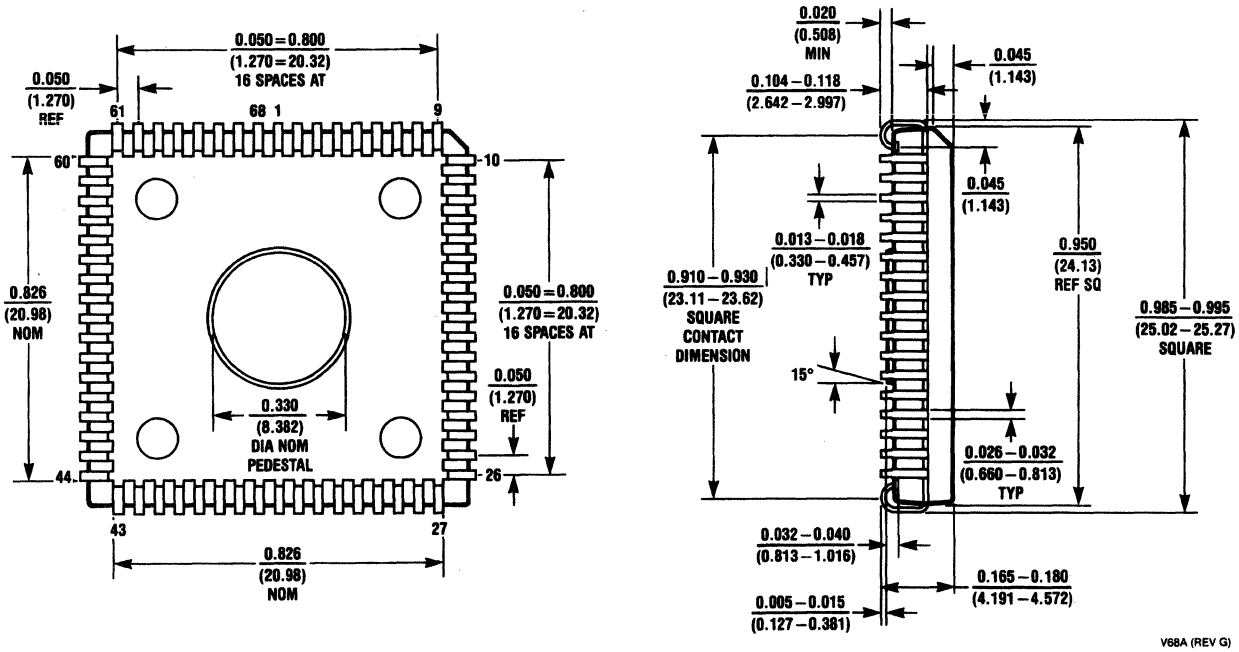
Offset/Length Modifier appended to INSS, EXTS

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: Opcode not defined; CPU treats like CMPf. First operand has access class of read; second operand has access class of read. f-field selects 32-bit or 64-bit data.

Physical Dimensions inches (millimeters)



Plastic Chip Carrier (V)
Order Number NS32CG16V-10 or NS32CG16V-15
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
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: (408) 721-5000
TWX: (910) 339-9240

National Semiconductor GmbH
Westendstrasse 193-195
D-8000 Munchen 21
West Germany
Tel: (089) 5 70 95 01
Telex: 522772

NS Japan Ltd.
Sanseido Bldg. 5F
4-15 Nishi Shinjuku
Shinjuku-Ku,
Tokyo 160, Japan
Tel: 3-299-7001
FAX: 3-299-7000

National Semiconductor Hong Kong Ltd.
Southeast Asia Marketing
Austin Tower, 4th Floor
22-26A Austin Avenue
Tsimshatsui, Kowloon, H.K.
Tel: 3-7231290, 3-7243645
Cable: NSSEAMKTG
Telex: 52996 NSSEA HX

National Semicondutores Do Brasil Ltda.
Av. Brig. Faria Lima, 830
8 Andar
01452 Sao Paulo, SP. Brasil
Tel: (55/11) 212-5066
Telex: 391-1131931 NSBR BR

National Semiconductor (Australia) PTY, Ltd.
21/3 High Street
Bayswater, Victoria 3153
Australia
Tel: (03) 729-6333
Telex: AA32096

National Semiconductor

PRELIMINARY

NS32332-10/NS32332-15
32-Bit Advanced Microprocessors

General Description

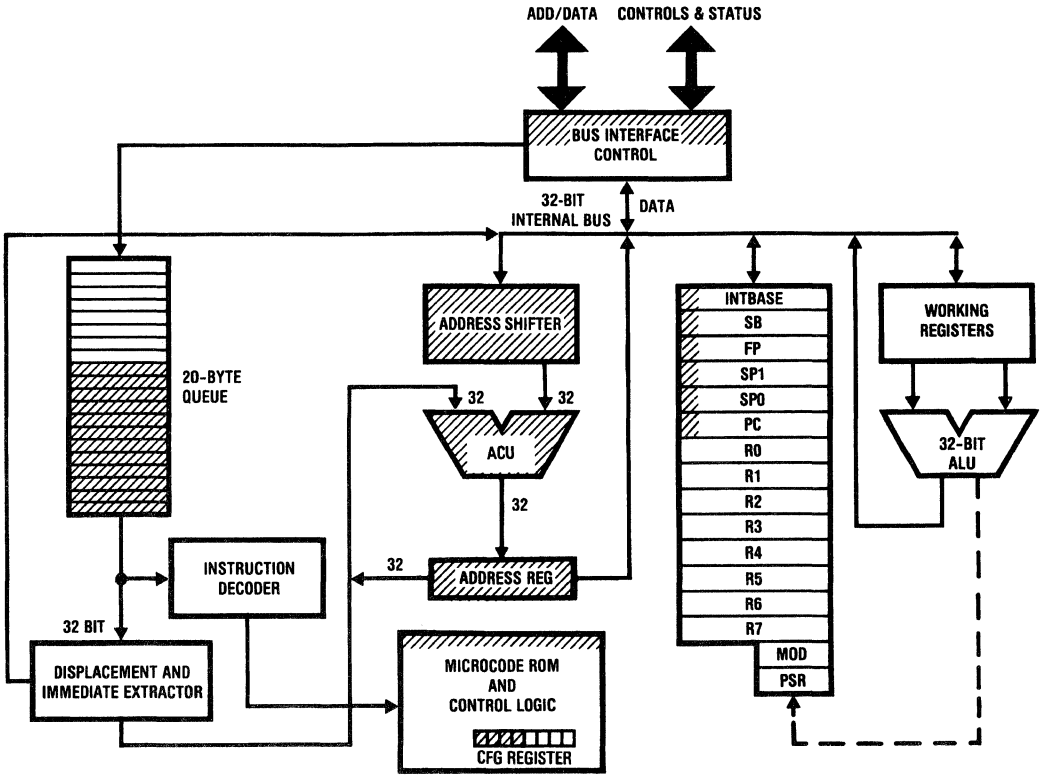
The NS32332 is a 32-bit, virtual memory microprocessor with 4 GByte addressing and an enhanced internal implementation. It is fully object code compatible with other Series 32000® microprocessors, and it has the added features of 32-bit addressing, higher instruction execution throughput, cache support, and expanded bus handling capabilities. The new bus features include bus error and retry support, dynamic bus sizing, burst mode memory accessing, and enhanced slave processor communication protocol. The higher clock frequency and added features of the NS32332 enable it to deliver 2 to 3 times the performance of the NS32032.

The NS32332 microprocessor is designed to work with both the 16- and 32-bit slave processors of the Series 32000 family.

Features

- 32-bit architecture and implementation
- 4 Gbyte uniform addressing space
- Software compatible with the Series 32000 Family
- Powerful instruction set
  - — General 2-address capability
  - — Very high degree of symmetry
  - — Address modes optimized for high level languages
- Supports both 16- and 32-bit Slave Processor Protocol
  - — Memory management support via NS32082 or NS32382
  - — Floating point support via NS32081 or NS32381
- Extensive bus feature
  - — Burst mode memory accessing
  - — Cache memory support
  - — Dynamic bus configuration (8-, 16-, 32-bits)
  - — Fast bus protocol
- High speed XMOS™ technology
- 84 Pin grid array package

Block Diagram



TL/EE/8673–1

FIGURE 1

*Shaded areas indicate enhancements from the NS32032.

National Semiconductor

PRELIMINARY

NS32532-20/NS32532-25/NS32532-30 High-Performance 32-Bit Microprocessor

General Description

The NS32532 is a high-performance 32-bit microprocessor in the Series 32000® family. It is software compatible with the previous microprocessors in the family but with a greatly enhanced internal implementation.

The high-performance specifications are the result of a four-stage instruction pipeline, on-chip instruction and data caches, on-chip memory management unit and a significantly increased clock frequency. In addition, the system interface provides optimal support for applications spanning a wide range, from low-cost, real-time controllers to highly sophisticated, general purpose multiprocessor systems.

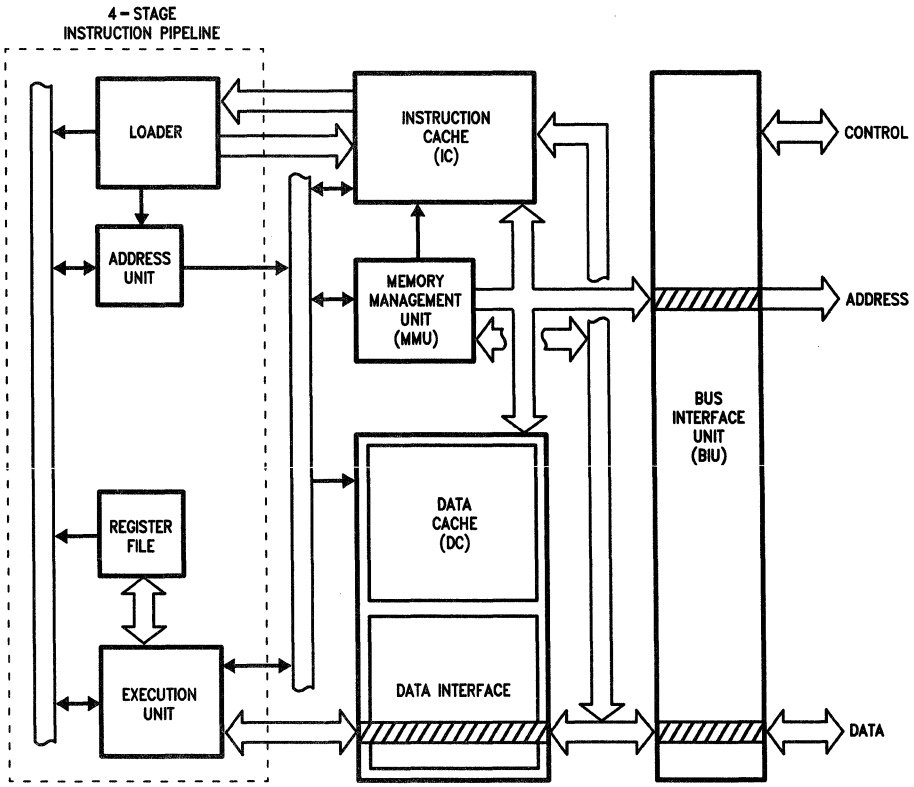
The NS32532 integrates more than 370,000 transistors fabricated in a 1.25 μm double-metal CMOS technology. The advanced technology and mainframe-like design of the device enable it to achieve more than 10 times the throughput of the NS32032 in typical applications.

In addition to generally improved performance, the NS32532 offers much faster interrupt service and task switching for real-time applications.

Features

- Software compatible with the Series 32000 family
- 32-bit architecture and implementation
- 4-GByte uniform addressing space
- On-chip memory management unit with 64-entry translation look-aside buffer
- 4-Stage instruction pipeline
- 512-Byte on-chip instruction cache
- 1024-Byte on-chip data cache
- High-performance bus
  - — Separate 32-bit address and data lines
  - — Burst mode memory accessing
  - — Dynamic bus sizing
- Extensive multiprocessing support
- Floating-point support via the NS32381 or NS32580
- 1.25 μm double-metal CMOS technology
- 175-pin PGA package

Block Diagram



TL/EE/9354–1

FIGURE 1

Section 2
Peripherals

Complete specifications for devices referenced in this section can be found in the 1988 Series 32000 Databook.

National Semiconductor

NS32081-10/NS32081-15 Floating-Point Units

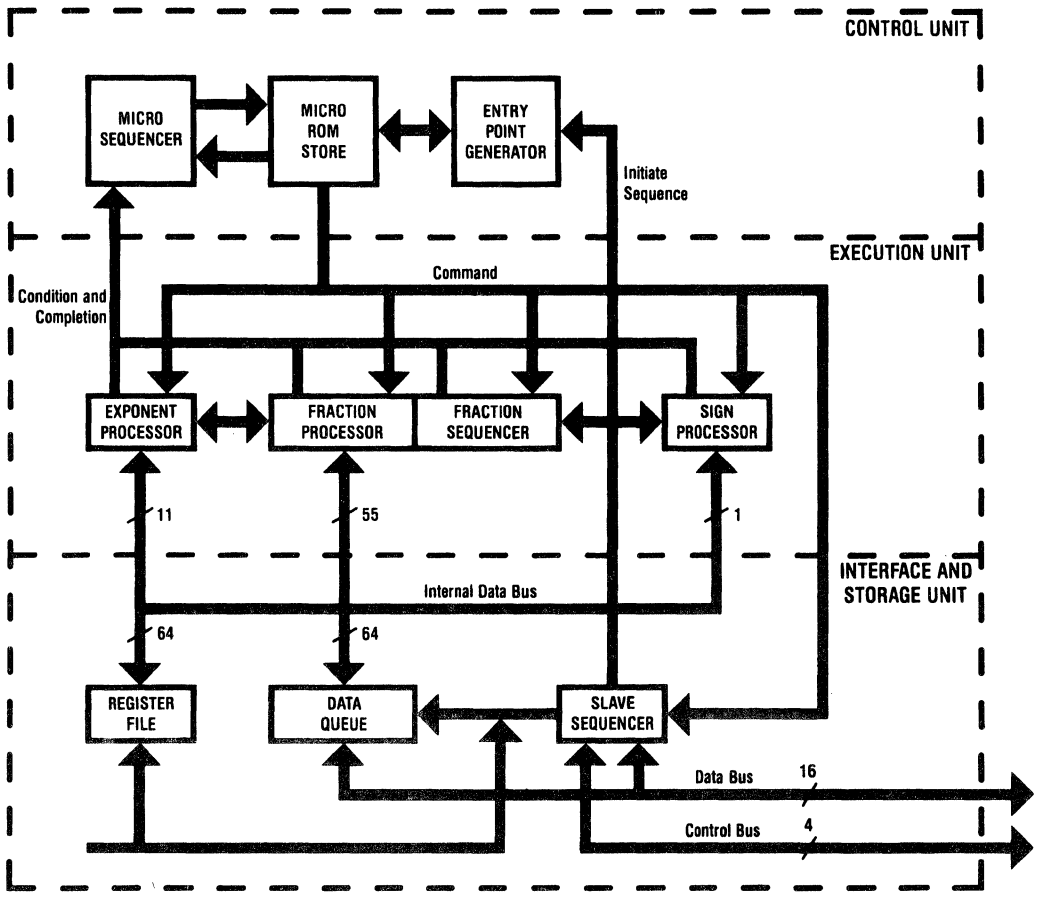
General Description

The NS32081 Floating-Point Unit functions as a slave processor in National Semiconductor's Series 32000® microprocessor family. It provides a high-speed floating-point instruction set for any Series 32000 family CPU, while remaining architecturally consistent with the full two-address architecture and powerful addressing modes of the Series 32000 micro-processor family.

Features

- Eight on-chip data registers
- 32-bit and 64-bit operations
- Supports proposed IEEE standard for binary floating-point arithmetic, Task P754
- Directly compatible with NS32016, NS32008 and NS32032 CPUs
- High-speed XMOS™ technology
- Single 5V supply
- 24-pin dual in-line package

Block Diagram



TL/EE/5234–1

National Semiconductor

PRELIMINARY

NS32381-15/NS32381-20 Floating-Point Unit

General Description

The NS32381 is a second generation, CMOS, floating-point slave processor that is fully software compatible with its forerunner, the NS32081 FPU. The NS32381 FPU functions with any Series 32000 CPU, from the NS32008 to the NS32532, in a tightly coupled slave configuration. The performance of the NS32381 has been increased over the NS32081 by architecture improvements, hardware enhancements, and higher clock frequencies. Key improvements include the addition of a 32-bit slave protocol, an early done algorithm to increase CPU/FPU parallelism, an expanded register set, an automatic power down feature, expanded math hardware, and additional instructions.

The NS32381 FPU contains eight 64-bit data registers and a Floating-Point Status Register (FSR). The FPU executes 20 instructions, and operates on both single and double-precision operands. Three separate processors in the NS32381 manipulate the mantissa, sign, and exponent. The NS32381 FPU conforms to IEEE standard 754-1985 for binary floating-point arithmetic.

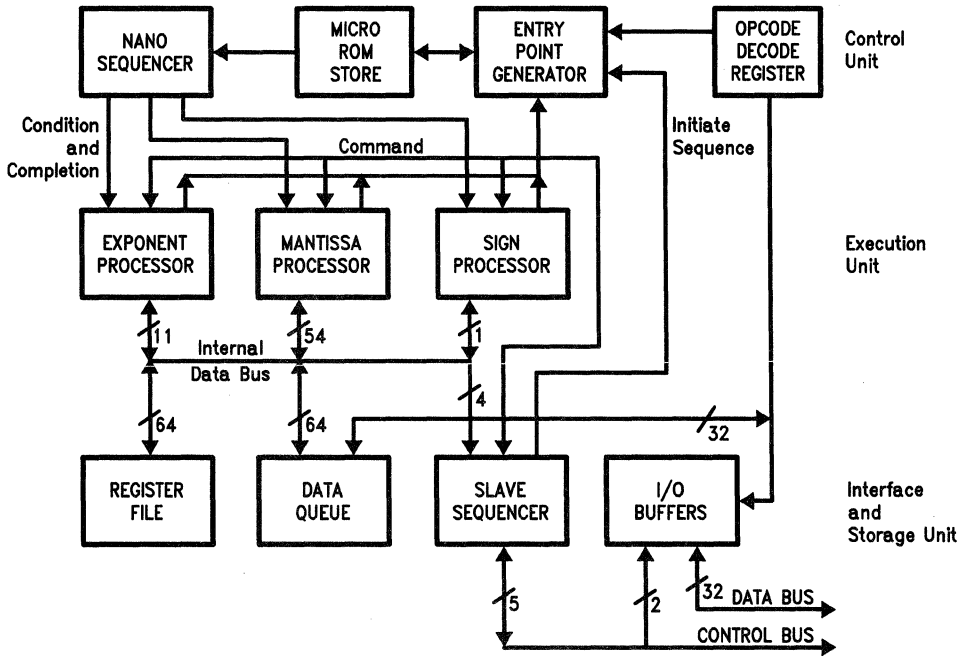
When used with a Series 32000 CPU, the CPU and NS32381 FPU form a tightly coupled computer cluster. This cluster appears to the user as a single processing unit. All addressing modes, including two address operations, are available with the floating-point instructions. In addition, CPU and FPU communication is handled automatically, and is user transparent.

The FPU is fabricated with National's advanced double-metal CMOS process. It is available in a 68-pin Pin Grid Array (PGA) package.

Features

- Directly compatible with NS32008, NS32016, NS32C016, NS32032, NS32C032, NS32332 and NS32532 microprocessors
- Selectable 16-bit or 32-bit Slave Protocol
- Conforms to IEEE standard 754-1985 for binary floating-point arithmetic
- Early done algorithm
- Single (32-bit) and double (64-bit) precision operations
- Eight on-chip (64-bit) data registers
- (Automatic) power down mode
- Full upward compatibility with existing 32000 software
- High speed double-metal CMOS design
- 68-pin PGA package

FPU Block Diagram



TL/EE/9157-1

FIGURE 1-1

National Semiconductor

NS32202-10 Interrupt Control Unit

General Description

The NS32202 Interrupt Control Unit (ICU) is the interrupt controller for the Series 32000® microprocessor family. It is a support circuit that minimizes the software and real-time overhead required to handle multi-level, prioritized interrupts. A single NS32202 manages up to 16 interrupt sources, resolves interrupt priorities, and supplies a single-byte interrupt vector to the CPU.

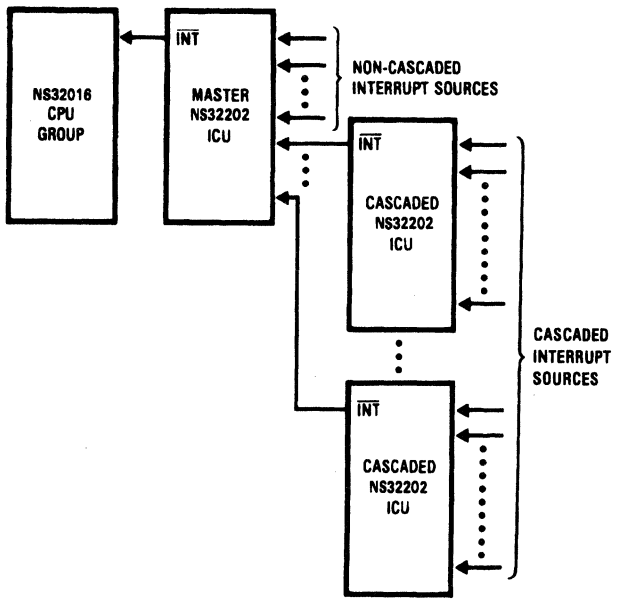
The NS32202 can operate in either of two data bus modes: 16-bit or 8-bit. In the 16-bit mode, eight hardware and eight software interrupt positions are available. In the 8-bit mode, 16 hardware interrupt positions are available, 8 of which can be used as software interrupts. In this mode, up to 16 additional ICUs may be cascaded to handle a maximum of 256 interrupts.

Two 16-bit counters, which may be concatenated under program control into a single 32-bit counter, are also available for real-time applications.

Features

- 16 maskable interrupt sources, cascadable to 256
- Programmable 8- or 16-bit data bus mode
- Edge or level triggering for each hardware interrupt with individually selectable polarities
- 8 software interrupts
- Fixed or rotating priority modes
- Two 16-bit, DC to 10 MHz counters, that may be concatenated into a single 32-bit counter
- Optional 8-bit I/O port available in 8-bit data bus mode
- High-speed XMOS™ technology
- Single, +5V supply
- 40-pin, dual in-line package

Basic System Configuration



TL/EE/5117–1

National Semiconductor

PRELIMINARY

NS32203-10 Direct Memory Access Controller

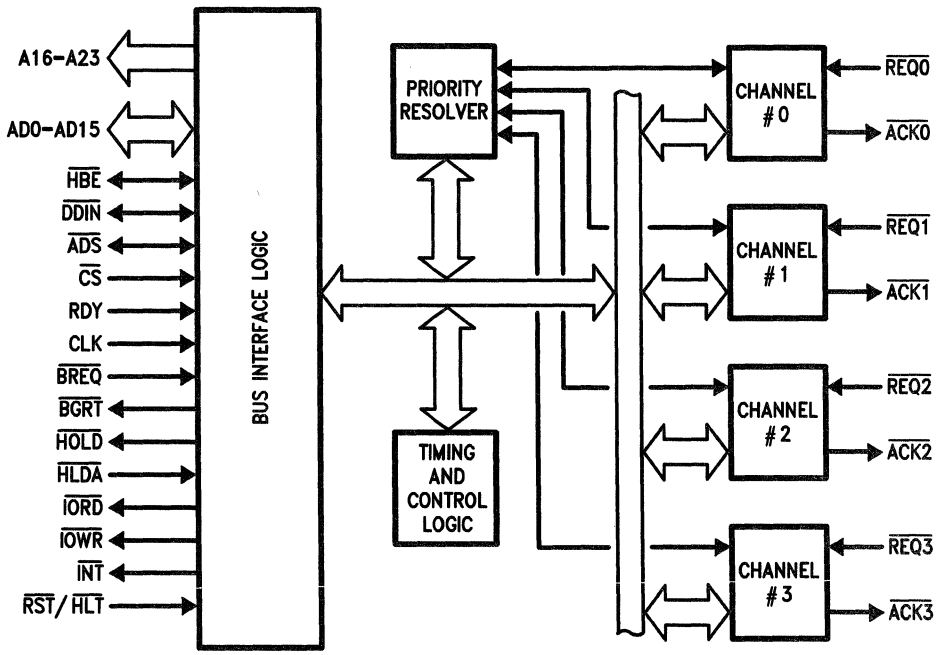
General Description

The NS32203 Direct Memory Access Controller (DMAC) is a support chip for the Series 32000® microprocessor family designed to relieve the CPU of data transfers between memory and I/O devices. The device is capable of packing data received from 8-bit peripherals into 16-bit words to reduce system bus loading. It can operate in local and remote configurations. In the local configuration it is connected to the multiplexed Series 32000 bus and shares with the CPU, the bus control signals from the NS32201 Timing Control Unit (TCU). In the remote configuration, the DMAC, in conjunction with its own TCU, communicates with I/O devices and/or memory through a dedicated bus, enabling rapid transfers between memory and I/O devices. The DMAC provides 4 16-bit I/O channels which may be configured as two complementary pairs to support chaining.

Features

- Direct or Indirect data transfers
- Memory to Memory, I/O to I/O or Memory to I/O transfers
- Remote or Local configurations
- 8-Bit or 16-Bit transfers
- Transfer rates up to 5 Megabytes per second
- Command Chaining on complementary channels
- Wide range of channel commands
- Search capability
- Interrupt Vector generation
- Simple interface with the Series 32000 Family of Microprocessors
- High Speed XMOS™ Technology
- Single +5V Supply
- 48-Pin Dual-In-Line Package

Block Diagram



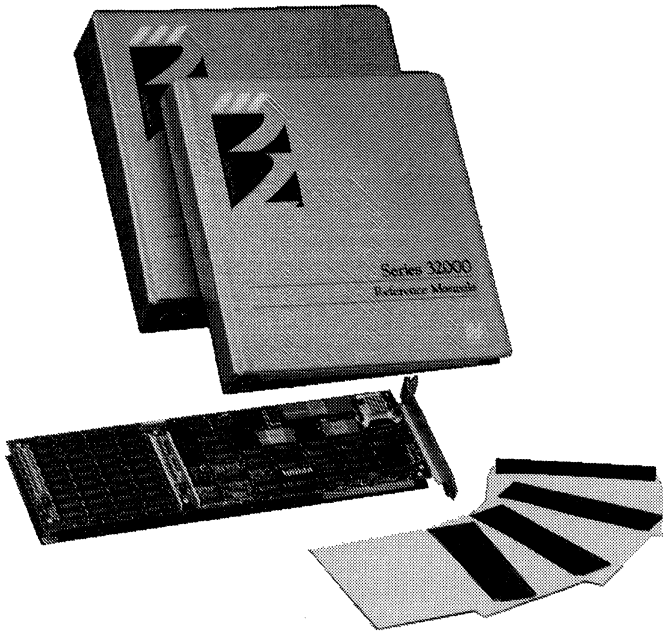
TL/EE/8701–1

Section 3
Development Tools

Complete specifications for devices referenced in this section can be found in the 1988 Series 32000 Databook.

National Semiconductor

SYS32/20 PC Add-In Development Package



TL/C/9250–1

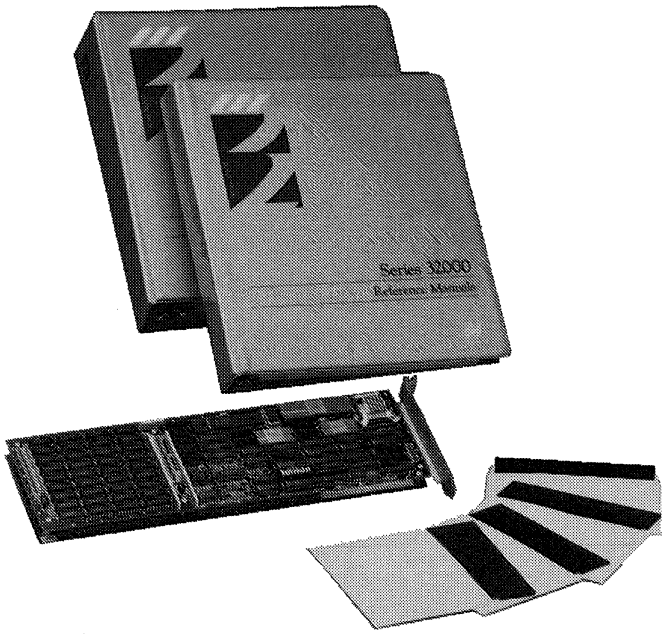
- **High Performance, 10 MHz, no-wait state, 32-bit expansion board for an IBM-PC/AT or compatible system**
- **An Operating System derived from AT&T's UNIX® System V.3**
- **The Series 32000 GNX (GENIX Native and Cross-Support) Language tools including the Series 32000 assembler, linker, monitors and debuggers**
- **Hardware that supports the NS32032 CPU, NS32082 MMU, NS32201 TCU and the NS32081 FPU**
- **Two available on-board memory configurations:**
  - 2-Mbyte RAM
  - 4-MByte RAM
- **Software available on 1.2-MByte floppies**
- **Complete support for the following application tools:**
  - SPLICE
  - National's Series 32000 Development Board Family
  - Compilers for C, FORTRAN77, Pascal and Ada
  - Complete System V Documentation
  - 4.2 "bsd" Utilities
  - Tools for Documentors (TFD), a derivative of AT&T's DWB™ utilities
  - Multiuser environment

Description

National Semiconductor's SYS32/20 is a complete, high performance development package that converts an IBM-PC/AT or compatible system into an ideal environment for the support of Series 32000®-based applications. The SYS32/20 PC Add-In Development Package allows mainframe-size programs to run on a personal computer at speeds similar to those of a VAX 780. The SYS32/20 consists of a 32-bit PC Add-In board based on the Series 32000 chip set, a complete port of AT&T's UNIX® System V.3 specially developed software that integrates the UNIX and DOS operating systems, and National's Series 32000 development tools (GNX).

National Semiconductor

SYS32/30 PC-Add-In Development Package



TL/EE/9420–1

- **15 MHz NS32332/NS32382 Add-In board for an IBM® PC/AT® or compatible system**
- **2–3 MIP system performance**
- **No wait-state, on-board memory in 4-, 8- or 16-Mbyte configurations**
- **Operating system derived from AT&T's UNIX® System V Release 3**
- **Multi-user support**
- **GENIX™ Native and Cross-Support (GNX™) language tools. Includes— assembler, linker, libraries, debuggers**
- **Support for other Series 32000® development products:**
  - **— SPLICE**
  - **— National's Series 32000 Development Board family**
  - **— Compilers: C, FORTRAN77, Pascal, Ada®**
- **Easy to use DOS/UNIX interface**

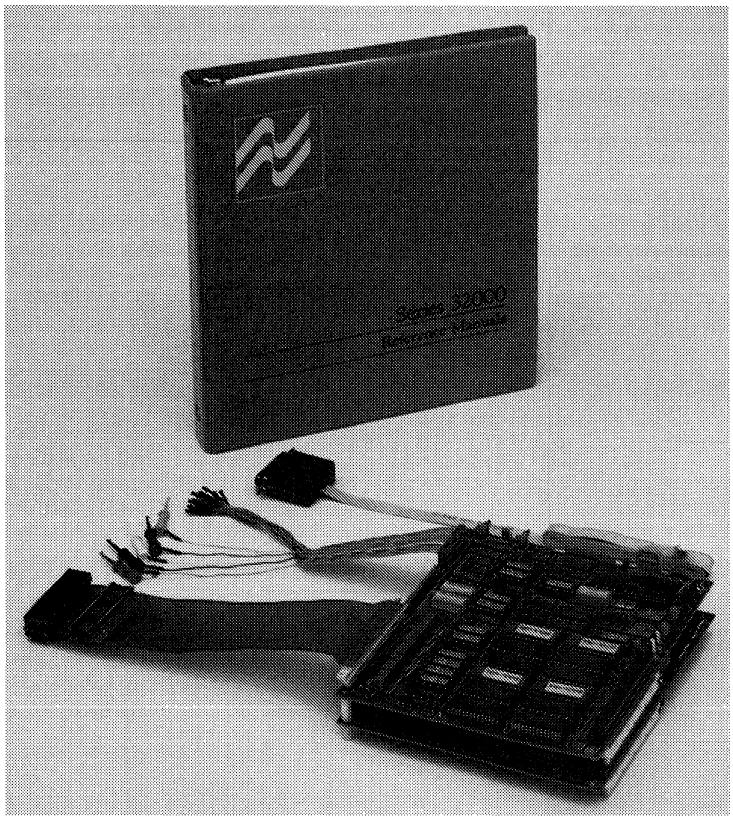
Product Overview

The SYS32™/30 is a complete, high-performance development package that converts an IBM PC/AT or compatible computer into a powerful multi-user system for developing applications that use National Semiconductor Series 32000 microprocessor family components. The SYS32/30 add-in processor board containing the Series 32000 chip cluster with the NS32332 microprocessor allows programs to run on a personal computer at speeds greater than those of a VAX™ 780. The chip cluster on the processor board includes the NS32332 Central Processing Unit, NS32382 Memory Management Unit, NS32C201 Timing Control Unit and the NS32081 Floating-Point Unit.

Along with the processor board, the SYS32/30 package contains the Opus5™ operating system. This operating system is a port of AT&T's UNIX System V Release 3, and is derived from GENIX V.3, National

National Semiconductor

SPLICE Development Tool



TL/R/9347–1

- **Download capabilities via serial connections**
- **256 Kbytes of mappable memory**
- **Optional 1-Mbyte memory board, expands memory up to 8 Mbytes**
- **On-board monitor with power-on diagnostics**
- **Supports Series 32000 CPUs, including: NS32332 NS32CG16 NS32032 NS32C032 NS32016 NS32C016 NS32008**
- **Parallel I/O port reserved for future highspeed download capabilities**
- **Programmable serial port baud rates**
- **CPU bus status test points for logic analyzer connections**
- **4 LED indicators for diagnostic results and general user applications**
- **RESET and NMI push buttons**
- **15 MHz maximum operation**

1.0 Product Overview

The SPLICE Development Tool provides a communication link between a Series 32000 target and a development system host. This connection allows users to download and map their software onto target memory and then debug this software using National Semiconductor's debuggers.

SPLICE includes two RS232 serial ports for the system host/terminal. These ports are particularly useful for target systems that have no serial ports, such as embedded controller designs.

SPLICE is also useful for designs with ROM-based software, or designs whose memory portion has not yet been built. SPLICE provides 256 Kbytes of SRAM which users can map into target memory. Using mapped memory considerably reduces software development time.

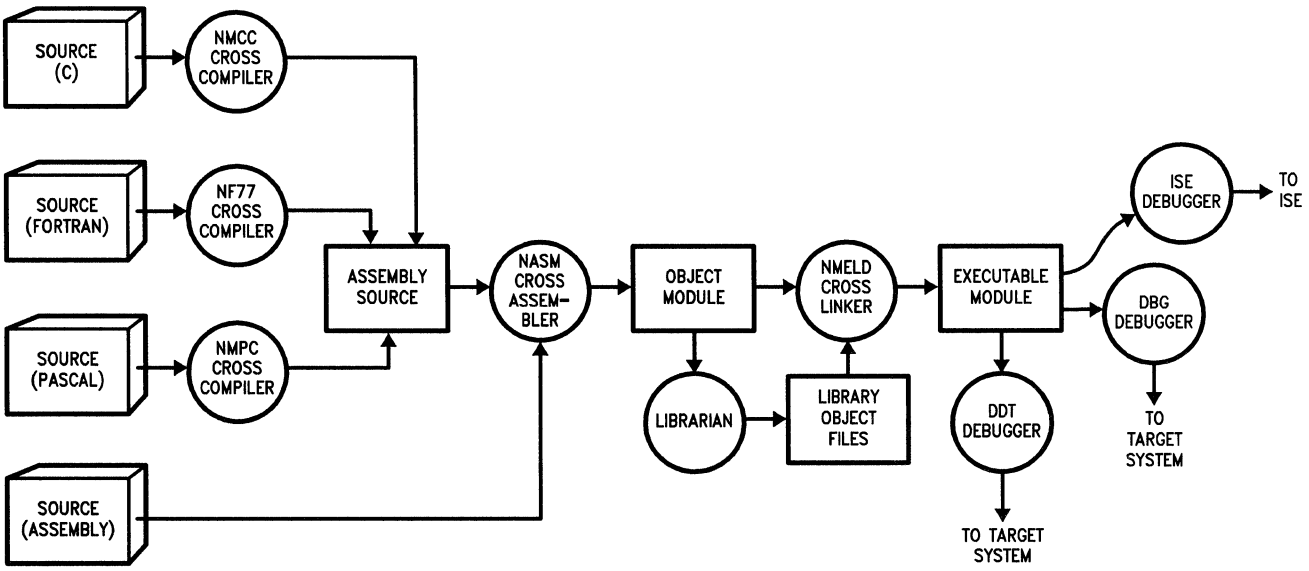
SPLICE also uses the target system's chipset. This cost-effective feature is achieved through the use of CPU and MMU target cables.

Section 4
Software

Complete specifications for devices referenced in this section can be found in the 1988 Series 32000 Databook.

National Semiconductor

Series 32000® GENIX™ Native and Cross-Support (GNX™) Language Tools (Release 2)



TL/GG/8780–1

- **Implements AT&T's standard Common Object File Format (COFF)**
- **Optimizing C Compiler (optional)**
- **Optimizing FORTRAN 77 Compiler (optional)**
- **Pascal Compiler (optional)**
- **Series 32000 assembler and linker**
- **In-System Emulator Support**
- **Interactive remote debugger with helpful command interface**
- **Available in binary for the VAX™ UNIX® 4.3 bsd operating system under derivatives of the Berkeley operating system**
- **Available in binary for the VAX/VMS™ operating system**
- **Available in binary on National Semiconductor Series 32000 Systems**
- **Available in source for porting to other operating system environments**

Product Overview

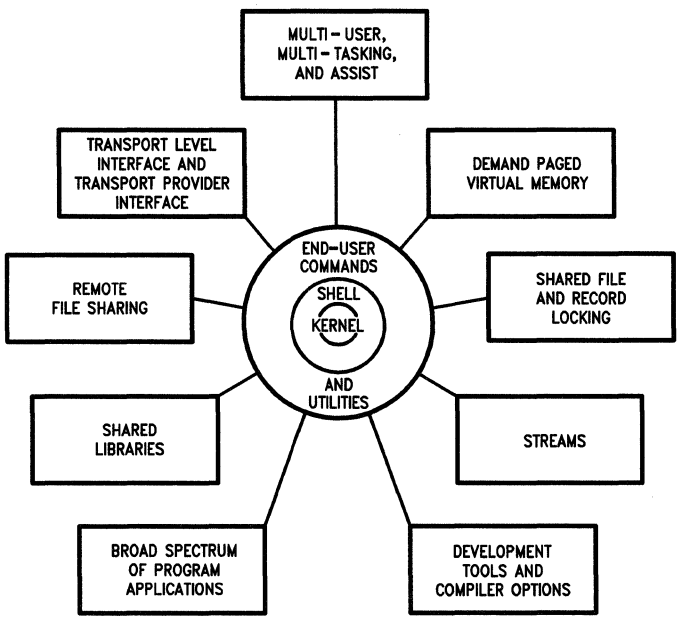
The Series 32000 GNX Language Tools are a set of software development tools for the Series 32000 microprocessor family. Optional high-level language compilers work in conjunction with the standard components to provide tools that can be combined to meet a variety of development needs.

GENIX Native and Cross-Support (GNX) Language Tools

The Series 32000 GNX Language Tools are based on AT&T's Common Object File Format (COFF). With appropriate command-line arguments and when linked with appropriate libraries, code generated by the GNX language tools can be executed in any Series 32000 target environment. In addition, these tools can be used to develop operating-system-independent code or code designed to run in conjunction with real-time kernels, such as National's EXEC and VRTX®/Series 32000. All of National's new language tools conform to the COFF file format, thereby ensuring that modules produced by any one set of tools can be linked with objects produced by any other set of GNX tools.

National Semiconductor

GENIX™/V.3 Operating System



TL/R/9263–1

- **Derived from AT&T's System V, Release 3.0, UNIX® Operating System**
- **Demand-paged Virtual Memory**
- **Mandatory and Advisory File and Record Locking**
- **Streams**
- **Transport Level Interface and Transport Provider Interface**
- **Remote File Sharing**
- **Shared Libraries**
- **Assist**
- **C Compiler and Associated Language Tools**

General Description

GENIX/V.3 is a port of AT&T's System V, Release 3.0, UNIX operating system for the Series 32000® microprocessor family. GENIX/V.3 is available in source form and can be adapted to serve as the operating system on customer-designed Series 32000-based systems.

GENIX/V.3 is a multitasking, multiuser operating system that provides an abundance of programs and utilities for text processing, program development, and system administration. GENIX/V.3 supports a wide variety of applications ranging from databases to graphics packages available from independent software vendors.

GENIX/V.3 carries forward all of the enhancements from Systems V/Series 32000, such as demand-paged virtual memory and file and record locking, while introducing significant new features that support local area networking.

GENIX/V.3 Features

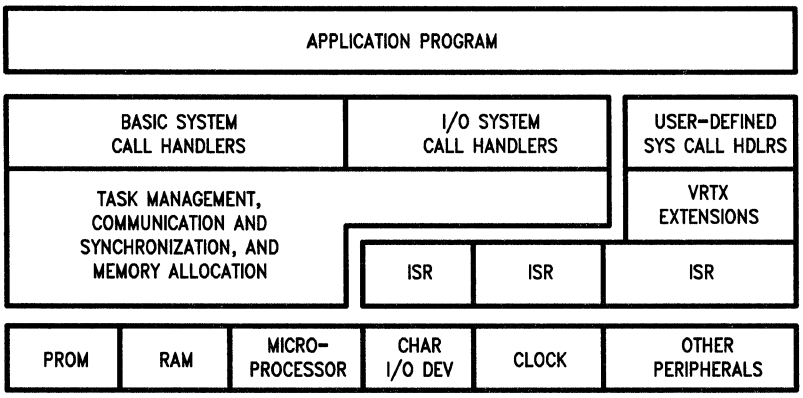
Streams

Streams is a general, flexible facility for the development of communications services within the UNIX operating system. Streams provides a consistent framework for the operation of network services (ranging from local area networks to individual device drivers) under the UNIX kernel.

National Semiconductor

Series 32000® Real-Time Software Components VRTX, IOX, FMX and TRACER

VRTX/Series 32000 R&D Package



TL/GG/8781–1

- **Real-time executive for Series 32000 embedded systems**
- **Can be installed in any Series 32000 hardware environment**
- **Manages multitasking with priority-based scheduler**
- **Manages memory pool, mailboxes, timing and terminal I/O**
- **Can reside in PROM and be located anywhere in memory**
- **No requirements for particular timers, interrupts or busses**
- **Has hooks at key processing points for easy customization**
- **Comprehensive manuals with many examples**
- **Hot-line technical support**
- **Integrated with interactive multitasking debugger (optional)**
- **Integrated with PC-DOS compatible file system (optional)**

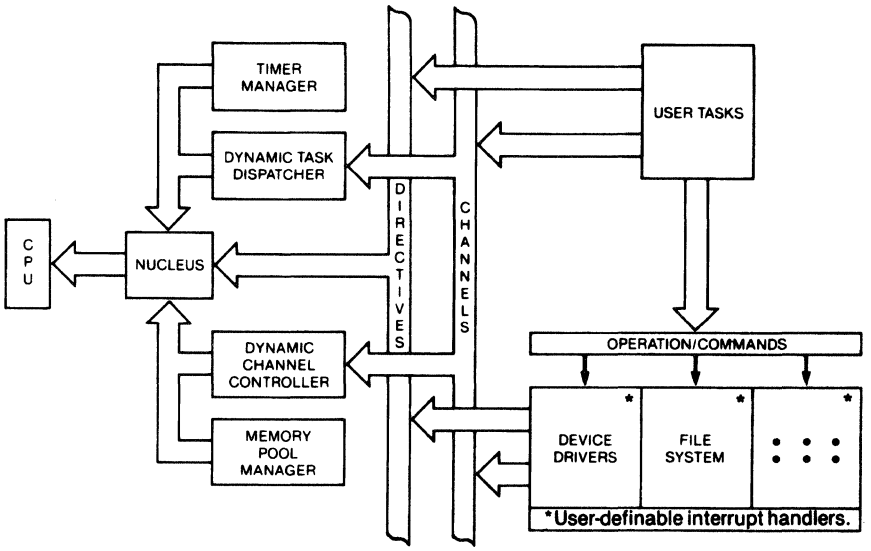
The VRTX®/Series 32000 executive is the central member of a set of silicon software building blocks used in Series 32000-based real-time embedded systems. The executive manages the multitasking environment and responds to operating system service requests from application tasks.

The executive can be used alone or in combination with the other silicon software components to build a more complete operating system. The IOX®/Series 32000 and FMX®/Series 32000 components support a file system that is media-compatible with PC-DOS. The TRACER™/Series 32000 is an interactive multitasking debugger that can be used in VRTX-based systems for debug, download and test.

All the components can reside in PROM's installed in the target system. They can be placed anywhere in the address space and make minimal assumptions about the hardware environment. Small user-written routines supply information about the local implementation of interrupts, timers, I/O devices, etc. Application tasks interface to the components with Series 32000 SVC (Supervisor Call) interrupts, thus code for the components does not require linking with user-written code.

National Semiconductor

Series 32000® EXEC ROMable Real-Time Multitasking EXECUTIVE



TL/GG/7291-1

- **Provides a multitasking executive for real-time applications**
- **Supports all Series 32000 CPUs**
- **Complete Source Code Package**
  - — Fully user configurable
  - — Hardware independent
- **Extensive user implementation support**
  - — Unique demo, program introduction
  - — C and Pascal interface libraries
  - — Sample terminal drivers
  - — Integrated with Series 32000 development boards and monitor
- **ROMable**
- **Reconfigurable**
- **Real-time clock support for time-of-day and event scheduling**
- **Allows up to 256 levels of task priority which can be dynamically assigned**
- **Up to 256 logical channels for task communication**
- **Free-memory pool control**
- **Available for VAX™/VMS™, VAX/UNIX® and SYS32™ development environments**

Product Overview

EXEC is National Semiconductor's real-time, multitasking executive for Series 32000 based applications. Its primary purpose is to simplify the task of designing application software and provides a base upon which users can build a wide range of application systems. EXEC requires only 2K bytes of RAM and only 4K bytes of ROM and is fully compatible with National Semiconductor's Series 32000 family and the Series 32000 development board family.

EXEC allows the user to monitor and control multiple external events that occur asynchronously in real-time, such as intertask communications, system resource access based upon task priority, real-time clock control, and interrupt handling. These functions greatly simplify application development in such areas as instrumentation and control, test and measurement, and data communications. In these applications, EXEC provides an environment in which systems programmers can immediately implement software for their particular application without regard to the details of the system interaction.

Section 5

Application Notes

Line Drawing with the NS32CG16; NS32CG16 Graphics Note 5

National Semiconductor
Application Note 522
Nancy Cossitt

1.0 INTRODUCTION

The Bresenham algorithm, as described in the "Series 32000® Graphics Note 5" is a common integer algorithm used in many graphics systems for line drawing. However, special instructions of the NS32CG16 processor allow it to take advantage of another faster integer algorithm. This application note describes the algorithm and shows an implementation on the NS32CG16 processor using the SBITS (Set BIT String) and SBITPS (Set BIT Perpendicular String) instructions. Timing for the DRAW_LINE algorithm is given in Tables A, B and C of the Timing Appendix. The timing from the original Bresenham iterative method using the NS32CG16 is given in Table D.

The bit map memory conventions followed in this note are the same as those given in the NS32CG16 Reference Manual and Datasheet, and all lines drawn are monochrome. Series 32000 Graphics Note 5, AN-524, is recommended reading.

2.0 DESCRIPTION

All rasterized lines are formed by sequences of line "slices" which are separated by a unit shift diagonal to the direction of these slices. For example, the line shown in *Figure 1* is composed of 7 slices, each slice separated by a unit diagonal shift in the positive direction. Notice that the slices of the line vary in length. The algorithm presented in this note determines the length of each slice, given the slope and the endpoints of the line.

Depending on the slope of the line, these slices will extend along the horizontal axis, the vertical axis or the diagonal axis with respect to the image plane (i.e., a printed page or CRT screen). If the data memory is aligned with the image plane so that a positive one unit horizontal (x-axis) move in the image plane corresponds to a one bit move within a byte in the data memory, and so that a positive one unit vertical (y-axis) move in the image plane corresponds to a positive one "warp" (warp = the number pixels along the major axis of the bit map) move within the data memory, then the SBITS and SBITPS instructions can be used to quickly set bits within data memory to form the line slices on the image plane, as explained in section 3.1. For long horizontal lines, the MOVMP (MOVe Multiple Pattern) instruction is more efficient than SBITS. This instruction is discussed in section 3.1 and in the NS32CG16 Reference Manual.

2.1 Derivation of the Bresenham SLICE Algorithm

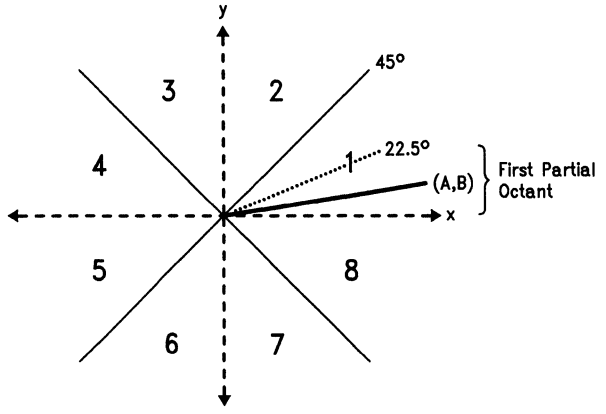
For the moment, consider only those lines in the X-Y coordinate system starting at the origin (0,0), finishing at an integer end point (x,y) and lying in the first partial octant, as in *Figure 2*. (The analysis will be extended for all lines in section 2.2.) The equation for one such line ending at (A,B) is:

$$y = mx,$$

where

$$m = B/A$$

is the slope of the line. Note that because the line lies in the first partial octant, $A > 2B \geq 1$.



TL/EE/9663-2

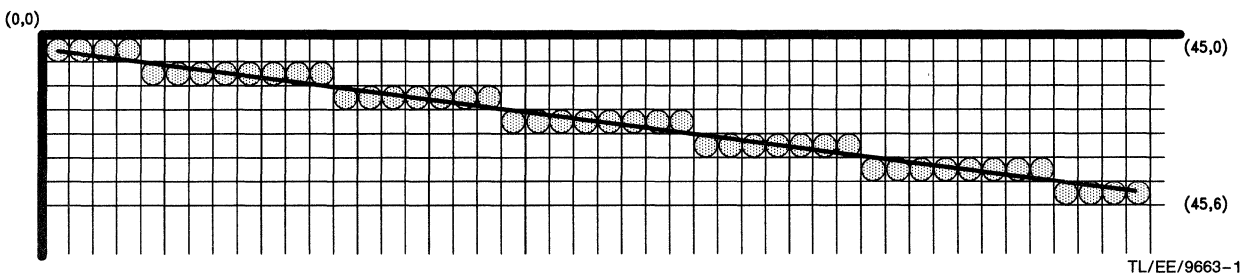
FIGURE 2

Each pixel plotted can be thought of as a unit square area on a Real plane *(Figure 3).* Assume each pixel square is situated so that the center of the square is the integer address of the pixel, and each pixel address is one unit away from its neighbor. Then let A_i represent the X-coordinate of the pixel, as shown in *Figure 3*. The value of Y at A_i is:

$$y = (B/A)A_i$$

where y is Real.

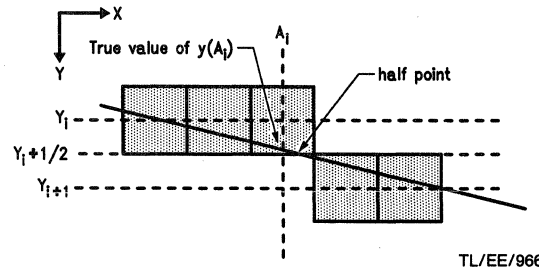
Since the address of each pixel plotted must have corresponding integer coordinates, the closest integer to y is either the upper bound of y or the lower bound. (Recall that upper and lower bounds refer to the smallest integer greater than or equal to y and the largest integer less than or equal to y respectively.) The original Bresenham algorithm was based on this concept, and had a decision variable within the main loop of the algorithm to decide whether the next y_{i+1} was the previous y_i (lower bound) or $y_i + 1$ (upper bound). For the SLICE algorithm, we are only concerned with when the value changes to $y_i + 1$, and the length of the previous slice up to that point.



TL/EE/9663-1

The line from (0,0) to (45,6) is a first octant line with run lengths 3-7-6-7-6-7-3. Notice that a pixel is plotted before the run begins so that the actual number of pixels plotted is equivalent to the run length +1.

FIGURE 1



TL/EE/9663-3

Y is incremented when the location of the half point is beyond A_i, or when the true value of Y at A_{i+1} is greater than $Y_i + \frac{1}{2}$.

FIGURE 3

In order for y_i to be incremented along the Y-axis, the true value of real y at $A_i + 1$ must be greater than or equal to the halfway point between y_i and $y_i + 1$ *(Figure 3)*. If we let i increment along the Y-axis, then this half point occurs when:

$$y = 1/2 + y_i$$

Or, because $y_i = i$ when incrementing along the Y-axis,

$$y = (1 + 2i)/2.$$

The real value of x at this point is:

$$x = A(1 + 2i)/2B$$

using $x = (1/m)y$. The lower bound of this value of x represents the x-coordinate of the pixel square containing the half point.

Letting A_i and A_{i+1} be two integer values of x where the real value of y is greater than or equal to the half point value $y_i + 1/2$ *(Figure 4)*, then the run length extends from $(A_i + 1, i + 1)$ to $(A_{i+1}, i + 1)$. The run length can then be calculated as:

$$H_{i+1} = A_{i+1} - A_i - 1$$

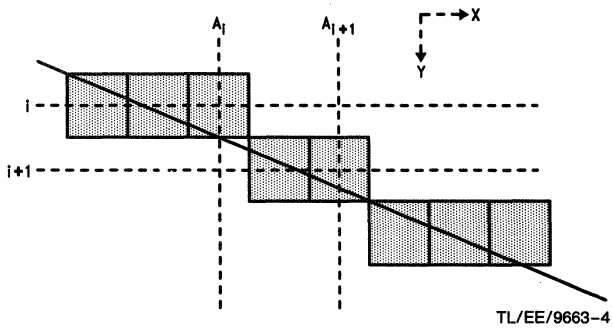
for $i = 0,1, \ldots ,(B-2)$. Using the equation for x above, we can now better define A_i as:

$$A_i = (A/2B) + (iA/B).$$

This equation has two real-valued divisions which are not suitable for an integer algorithm. However, the equation can be broken down so that it only involves an integer-valued division and its integer remainder, which is more efficient for processing. To do this we must define some intermediary integer values:

$\mathbf{Q} = \mathbf{lower}[A/B]$	{Lower bound of inverted slope}
$\mathbf{R} = {}_B\vert A$	{Integer residue of A modulo B}
$\mathbf{M} = \mathbf{lower}[A/2B]$	{Can also be defined as Q/2}
$\mathbf{N} = {}_{2B}\vert A$	{Integer residue of A modulo 2B}
$\mathbf{T_i} = {}_{2B}\vert (N+2iR)$	{Integer residue of (N+2iR) modulo 2B}

Note: ${}_A\vert B = B + A * \mathbf{lower}[A/B]$.



TL/EE/9663-4

Run length is calculated as $A_{i+1} - A_i - 1$. In this example, the run length is 1.

FIGURE 4

Using the above values we can now define A_i as,

$$A_i = (M + N/2B) + (iQ + iR/B)$$

$$A_i = M + iQ + (N + 2iR)/2B$$

Therefore, substituting A_i and A_{i+1} into the equation for H_{i+1}, the intermediate horizontal lengths are,

$$H_{i+1} = A_{i+1} - A_i - 1$$

$$H_{i+1} = \{M + (i+1)Q + \mathbf{lower}[(N + 2(i+1)R)/2B]\} - \{M + iQ + \mathbf{lower}[(N + 2iR)/2B]\} - 1$$

$$H_{i+1} = Q + \mathbf{lower}[(N + 2iR)/2B + 2R/2B] - \mathbf{lower}[(N + 2iR)/2B] - 1$$

$$H_{i+1} = Q - 1 + \mathbf{lower}[(T_i + 2R)/2B]$$

Analyzing the term $\mathbf{lower}[(T_i + 2R)/2B]$ it is shown that if $T_i + 2R \geq 2B$ then the term becomes 1, otherwise it becomes 0. This is due to the definition of residue and modulo. The term T_i is defined as:

$$(N + 2iR) - 2B(\mathbf{lower}[(N + 2iR)/2B]),$$

which means that $0 \leq T_i < 2B$. The same is true for R:

$$R = A - B(\mathbf{lower}[A/B]),$$

so that $0 \leq 2R < 2B$. Therefore,

$$0 \leq T_i + 2R < 4B$$

and,

$$0 \leq (T_i + 2R)/2B < 2.$$

The only possible integer values for this term are 0 and 1. The term will equal 0 if $T_i + 2R < 2B$, and it will equal 1 when $T_i + 2R \geq 2B$, and H_{i+1} will equal Q. The decision variable can now be defined as

$$\text{testvar} = T_i + 2R - 2B.$$

If testvar ≥ 0 then the horizontal run length is Q; if testvar < 0 then the run length is $Q-1$.

Looking again at the definition of T_i, a recursive relationship for the testvar can be formed.

$$T_{i+1} = (N + 2R(i+1)) - 2B(\mathbf{lower}[(N + 2R(i+1))/2B]$$

$$T_{i+1} = (N + 2iR + 2R) - 2B(\mathbf{lower}[(N + 2iR + 2R)/2B]$$

Since, as shown above, $0 < (T_i + 2R)/2B < 2$ then $\mathbf{lower}[(T_i + 2R)/2B] \leq 1$. In fact, if $T_i + 2R < 2B$ then $\mathbf{lower}[(T_i + 2R)/2B] = 0$, and if $T_i + 2R \geq 2B$ then $\mathbf{lower}[(T_i + 2R)/2B] = 1$. Therefore, letting $T_0 = N$,

$$T_{i+1} = T_i + 2R \quad \text{if } (T_i + 2R) < 2B$$

$$T_{i+1} = T_i + 2R - 2B \quad \text{if } (T_i + 2R) \geq 2B.$$

This gives the recursive relationship for testvar:

$$\text{testvar}_{i+1} = \text{testvar}_i + 2R$$

$$H_i = Q - 1$$

if $\text{testvar}_i < 0$. And, if $\text{testvar}_i \geq 0$:

$$\text{testvar}_{i+1} = \text{testvar}_i + 2R - 2B$$

$$H_i = Q.$$

These recursive equations allow the intermediate run lengths to be easily calculated using only a few additions and compare-and-branches.

The initial run length is calculated as follows:

$$H_0 = A_0 = \mathbf{lower}[A/2B] = M + \mathbf{lower}[N/2B] = M.$$

The final run length is similarly calculated as:

$$H_f = M - 1 \quad \text{if } N = 0 \text{ else } H_f = M.$$

Thus, the SLICE algorithm calculates the horizontal run lengths of a line using various parameters based on the first partial octant abscissa and ordinate of the line. The algorithm is efficient because it need only execute its main loop B times, which is a maximum of A/2, if A is normalized for the first partial octant. Compare this with the original Bresenham algorithm which always executes its main loop A times.

2.2 Extended Analysis for All Other Lines

In section 2.1 the SLICE algorithm was derived for lines starting at the origin and contained within the first octant ($B < 2A$). The algorithm is easily extended to encompass lines in all octants starting and ending at any integer coordinates within the pre-defined bit map. The only modifications necessary for this extension are those relating to the direction of movement and in defining the coordinates A and B.

In order to extend the algorithm to cover all classes of lines, the key parameters used by the algorithm must be normalized to the first partial octant. Those parameters are the abscissa and ordinate displacements and the movement of the bit pointer along the line. The abscissa and ordinate displacements of the line are normalized to the first octant by calculating:

$$\text{delta x} = x_f - x_s \text{ and delta y} = y_f - y_s$$

which represent the abscissa (delta x) and ordinate (delta y) displacements of the original line. Then, the first octant equivalents of A and B will be:

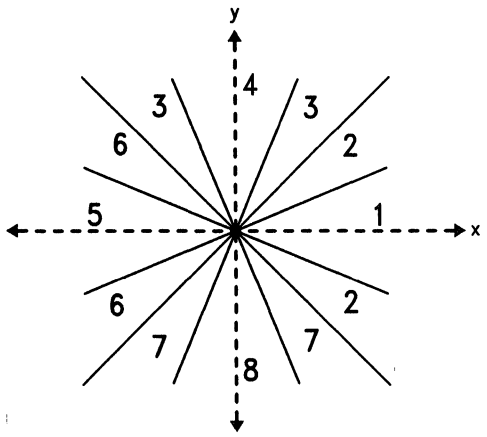
A = maximum $\{|\text{delta x}|,|\text{delta y}|\}$

B' = minimum $\{|\text{delta x}|,|\text{delta y}|\}$

B = minimum $\{B', A - B'\}$

The next step in normalizing the line for the first octant is to assign the correct value to the movement parameters. A line in the first octant and starting at the origin always has horizontal run lengths in the positive direction along the X (major) axis, and has diagonal movement one unit in the positive X direction and one unit in the positive Y (minor) direction. Since the SLICE algorithm calculates the run lengths independent of direction, variables can easily be defined which contain the direction of movement for each slice and each diagonal step within the different octants.

Lines of different angles starting at the origin have slices of different angles. For example, a line of angle between 22.5 degrees and 45 degrees has run lengths that are diagonal, not horizontal, and the direction of the diagonal step is horizontal, not diagonal. Because of this characteristic, it is convenient to break the 8 octants of the X-Y coordinate system into 16 sections, representing all of the partial octants. Then, re-number these partial octants so that they form new octants as in *Figure 5*. These redefined octants represent



TL/EE/9663–5

Redefined octants for SLICE algorithm. Notice that some of the octants are split. The origin is at the center of the drawing. Setting DELX positive on all lines makes opposite octants equivalent in the table below.

FIGURE 5

each of the eight angle classes of lines. For example, the lines in octants 3 and 7 are composed of diagonal (45 degree) slices in either the positive or negative direction, and have diagonal step in the vertical position. Lines in octants 4 and 8 have run length slices in the vertical direction with diagonal steps in the horizontal direction with respect to the X-Y plane.

In conclusion, the SLICE algorithm calculates successive run lengths in the same manner for lines in each octant. The only difference between the octants is the direction of movement of the bit pointer after each successive run length is calculated. The run lengths and diagonal steps for each octant are given in Table I. *Figure 5* shows the octants used by the SLICE algorithm.

3.0 IMPLEMENTATION OF SLICE USING SBITS, SBITPS AND MOVMP

The NS32CG16 features several powerful graphics instructions. The SLICE algorithm described by this application note is implemented with three of these instructions: SBITS, SBITPS and MOVMP. The SBITS instruction allows a horizontal string of bits to be set, while the SBITPS instruction can set vertical or diagonal strings of bits. The MOVMP instruction, not detailed in this application note, can be used to set long strings of bits faster than SBITS when the length is more than 200 bits in the horizontal direction. The BIGSET.S routine given in the appendix uses this instruction in conjunction with SBITS for long lines. These are very useful instructions for the SLICE run length algorithm, as will be shown in section 3.2.

TABLE I

OCTANT	DELA	DELB	DIAGONAL MOVE	RUN LENGTH
1 & 5	DELX	|DELY|	1 + (±WARP)	+HORZ
2 & 6	DELX	DELA-|DELY|	+ 1	±DIAG
3 & 7	|DELY|	DELA-DELX	±WARP	±DIAG
4 & 8	|DELY|	DELX	+ 1	±WARP

If DELX < 0 then the starting and ending coordinates are swapped. This simplifies initialization.

3.1 SBITS and SBITPS Tutorial

SBITS:

SBITS (Set BIT String) sets a string of bits along the horizontal axis of a pre-defined bit map. The instruction sets a string of up to 25 bits in a single execution using four arguments pre-stored in registers R0 through R3.

R0 = (32 bits) Base address of bit-string destination.

R1 = (32 bits, signed) Starting bit-offset from R0.

R2 = (32 bits, unsigned) Run length of the line segment.

R3 = (32 bits) Address of the string look-up table.

The value of the bit offset is used to calculate the bit number within the byte, assuming that the first bit is bit 0 and the last bit is bit 7. A maximum of 7 for the starting bit number added to a maximum of 25 for the run length requires a total of 32 bits. SBITS calculates the destination address of the first byte of the 32-bit double word to contain the string of set bits by the following:

Destination Byte = Base Address + Offset DIV 8.

Then, the starting bit number within the destination byte is:

Starting Bit = Offset MOD 8.

SBITS instruction then calculates the address for the 32-bit double word within the string look-up table (found in the NS32CG16 manual) which will be OR'ed with the 32-bit double word whose starting byte address is Destination Byte, as calculated above. The table is stored as eight contiguous sections, each containing 32 32-bit double words. Each of the eight sections corresponds to a different value of Starting Bit (Offset MOD 8), which has a possible range of 0 through 7. The 32 double words in each section correspond to each value of the run length (up to 25) added to the starting bit offset.

example:

	Register Contents
before	**after**
R0 = 1000	R0 = 1000
R1 = 235	R1 = 235
R2 = 16	R2 = 16
R3 = $stab	R3 = $stab

Destination Address = 1000 + (235 DIV 8) = 1029
Starting Bit = 235 MOD 8 = 3
Table Address = $stab + 4*(16 + (32*3)) = $stab + 448 bytes
32-bit Mask = 0x0007FFF8

This mask value is OR'ed with the 32-bit double word starting at byte address 1029 decimal. Notice that the mask 0x0007FFF8 leaves the first 3 bits and the last 13 bits alone. Thus, a string of 16 bits is set starting at bit number 3 at address 1029 decimal. The contents of the registers are unaffected by the execution of the SBITS instruction.

Since the SBITS instruction can set up to 25 bits in one execution, the run length in R2 can be compared to 25, and a special subroutine executed if it exceeds 25 bits. The subroutine will set the first 25 bits, then subtract 25 from the run length, and compare this to 25 again. This process is repeated until the run length is less than 25, in which case the remaining bits are set and the subroutine returns. The DRAW_LINE algorithm implemented in this application note uses this method for strings of bits to be set less than 200. For horizontal lines greater than 200 pixels in length, the BIGSET routine is more efficient, as described below.

BIGSET:

The utility program BIGSET.S is used to draw longer lines, more than 200 pixels in length, more efficiently than SBITS. BIGSET.S, which is given in the appendix, uses the MOVMP instruction (MOVe Multiple Pattern) to set long strings of bits. Since MOVMP operates on double-word aligned addresses most efficiently, the string is broken up into a starting string within the first byte, a series of bytes to be set, and an ending string which is the leftover bits to be set within the final byte. The starting and ending strings of bits, if any, are set using the SBITS table with an OR instruction.

SBITPS:

SBITPS (Set BIT Perpendicular String) handles both vertical lines and diagonal lines. This instruction also requires four arguments pre-stored in R0 through R3. R0, R1 and R2 are the Base Address, Starting Bit Offset and Run Length respectively, as for SBITS. R3, however, contains the destination warp.

Note: The Destination warp is the number of bits along the horizontal length of the bit map, or the number of bits between scan lines. It is also referred to as the "pitch" of the bit map. Thus, a vertical one-unit move in the positive direction would require adding the value of the warp to the bit pointer. A diagonal or 45 degree line is drawn when the warp is incremented or decremented by one.

The run length is a 32 bit unsigned magnitude.

example:
(Assume that the bit map is a 904 x 904 pixel grid.)

	Register Contents
before	**after**
R0 = 1000	R0 = 1000
R1 = 235	R1 = 235 + (150*904) = 135,835
R2 = 150	R2 = 0
R3 = +904	R3 = +904

Destination Address = 1029
Starting Bit Number = 3
Run Length = 150
Warp = +904

As in the example for SBITS, the Destination Address is 1029, with Starting Bit Number = 3. Since the warp in this example is +904 and the bit map is 904 x 904 bits, the line is vertical, has a length of 150 pixels and starts at bit number 3 within the byte whose address is 1029 decimal. Unlike the SBITS instruction, the SBITPS alters registers R1 and R2 during execution. R1 is set to the position of the last bit set plus the warp. However, this is convenient for drawing the next slice since R1 has been automatically updated to its proper horizontal position for setting the next bit. The bit offset in R1 need only be incremented by +1 or −1 to point to the exact position of the next bit to be set.

Diagonal lines are drawn when the value contained in R3 is an increment of the bit map's warp.

example:
(Assume that the bit map is a 904 x 904 pixel grid.)

Register Contents	
before	**after**
R0 = 1000	R0 = 1000
R1 = 235	R1 = 235 + (150*905) = 135,985
R2 = 150	R2 = 0
R3 = +905	R3 = +905

This example draws a diagonal line with positive slope starting at bit position 3 in byte 1029. Notice that the new value of R1 = 135,985 is exactly 150 pixels offset from the value of R1 in the vertical line drawn in the previous example. Adding +1 to the warp in this example caused the bit position to move not only in the positive vertical direction, but also in the positive horizontal direction, forming a diagonal line.

3.2 Implementation of DRAW_LINE and SLICE on the NS32CG16

Both a C version of the DRAW_LINE algorithm and an NS32CG16 assembly version are given in the appendix. The C program was implemented on SYS32/20 which uses the NS32032 processor. An emulation package developed by the Electronic Imaging Group at National was used to emulate the SBITS and SBITPS instructions in C, and also the MOVMP instruction used for lines longer than 200 pixels. The emulation routines, which cover all NS32CG16 instructions not available on other Series 32000 processors, are available as both C functions and Series 32000 assembly subroutines.

The DRAW_LINE program was first written in C using the emulation functions. Once this version was tested and functional, it was translated into Series 32000 code and further optimized for speed. The assembly version uses the Series 32000 assembly subroutines which emulate the SBITS and SBITPS instructions. NS32CG16 executable code was developed by replacing the emulation subroutine calls with the actual NS32CG16 instruction. The functional and optimized code was finally executed on the NS32CG16 processor with the aid of the DBG16 debugger for downloading the code to an NS32CG16 evaluation board. Timing for lines of various slopes is given in the Timing Appendix.

Most of the optimization efforts are concentrated in the main loop of the SLICE algorithm. Since the use of SBITS or SBITPS for the run length depends on the slope of the line, the code is unrolled for the different octants. This minimizes branching within the main loop, and cuts down on overall execution time. Also, the DRAW_LINE takes advantage of the NS32CG16's ability to draw fast horizontal, vertical and diagonal lines by separating these lines out from the actual Bresenham SLICE algorithm. Therefore, time is not wasted for trivial lines on executing the initialization sections and main loop sections of the SLICE algorithm.

Branching within the initialization section is also minimized by unrolling the code for each octant. Recall from section 2.2 that in order to extend the algorithm over all octants, the abscissa and ordinate displacements must be normalized to the first octant and the run length directions must be modified to preserve the slope of the line. Partitioning the program into "octant" modules makes the initialization for each octant less cluttered with compare-and-branches. Table I shows that each octant has a unique value for DELA and DELB (the normalized abscissa and ordinate displacements). Note that at the beginning of the programs, DELX or $x_f - x_s$ is checked for sign, and if negative, the absolute value function is performed and the starting and ending points are exchanged. This is done because each octant module of the SLICE algorithm only cares about the sign of DELY with respect to coordinate (x_s,y_s). DELX is only important when initializing DELA or DELB, and in this case, only the absolute value is needed.

4.0 SYSTEM SET-UP

NS32CG16 Evaluation Board:

—NS32CG16 with a 30 MHz Clock
—256KB Static RAM Memory (No Wait States)
—2 Serial ports
—MONCG16 Monitor

Host System:

—SYS32/20 running Unix System V
—DBG16 Debugger

Software for Benchmarking:

—START.C	Starts timer and calls DRIVER.
—DRIVER.C	Feeds vectors to DRAW_LINE.
—DRAW_LINE.S	Line drawing routine which includes SLICE.
—BIGSET.S	Uses MOVMPi to set longer lines. Called by DRAW_LINE if length > 200.

4.1 Timing

Timing Assumptions:

1. No wait states are used in the memory.
2. No screen refresh is performed.
3. The overhead referred to as the "driver" overhead is the time it takes to create the endpoints for each vector. This is application dependent, and is not included in the Vector/Sec and Pixel/Sec times.
4. The overhead referred to as the "line drawing" overhead is the time it takes to set up the registers for the actual line drawing routine. This overhead comes from the DRAW_LINE program only and is included in all times.
5. Raw data given in the Timing Appendix for the SBITS, SBITPS and MOVMP is the peak performance for these instructions. These times do not include line drawing overhead or driver overhead.

The timing for this line-drawing application was done so as to give meaningful results for a real graphics application and to allow the reader to calculate additional times if desired. The routines are not optimized for any particular application. All line drawing overhead, such as set-up and branching, is included in the given times for Timing Table A, B and C. The 23 μs driver overhead of the calling routines is not included in the given times for vectors per second and pixels per second. Calculation of these values was done by subtracting the 23 μs out of the average time per vector so that the given times are only for the processing of the vectors. They do not include the overhead of DRIVER.C and START.C (refer to these programs in the appendix).

In addition, the DRAW_LINE algorithm is timed for several test vectors at various strategic points in the code so that

the reader may verify set-up times or calculate other relevant times. The program DRAW__LINE.S in the appendix contains markers (e.g., T1, T2 . . .) for each point at which a particular time was taken. The program was run using a driver program (DRIVER.C in the appendix) which consists of several loops which pass test vectors to the DRAW__LINE routine. A "return" instruction was placed at the time marker so that the execution time was only measured up to that marker. These times are given in the Timing Appendix Table E and include total execution time up to each of the markers.

A millisecond interrupt timer on the NS32CG16 evaluation board was used to time the execution. For each execution, the DRIVER program executed its inner loop over 100 times, and sometimes over 1000 times, so that an accurate reading was obtained from the millisecond timer. The final times were divided by this loop count to obtain a "benchmark" time. This benchmark time was divided by the total number of lines drawn to obtain an average time per vector. The overhead of START.C and DRIVER.C in calling the DRAW__LINE.S routine was not counted in the average time per vector or the average time per pixel calculation. Table E of the Timing Appendix gives the timing for each of the markers and the conditions under which these times were taken.

5.0 CONCLUSION

The timing for the DRAW__LINE algorithm is a good indication of the performance of the NS32CG16 in a real application, something which the datasheet specifications can't always show. The timing clearly shows that the NS32CG16 is well-suited for line-drawing applications. Using the SBITS, SBITPS and the MOVMPi instructions, fast line-drawing is achieved for lines of all slopes and lengths. The NS32CG16 is an ideal processor for taking advantage of the much faster SLICE algorithm.

The SLICE algorithm, which calculates run lengths of line segments to form a complete rasterized line, is much faster than its Bresenham predecessor which calculates the line pixel by pixel. The SLICE algorithm always executes the main loop at least twice as fast as the original Bresenham algorithm, which executes its main loop exactly max{|delx|,|dely|} times for each line.

REFERENCES

J.E. Bresenham, IBM, Research Triangle Park, USA. "Run Length Slice Algorithm for Incremental Lines", **Fundamental Algorithms for Computer Graphics,** Springer-Verlag Berlin Heidelberg 1985.

N.M. Cossitt, National Semiconductor, "Bresenham's Line Algorithm Using the SBIT Instruction", **Series 32000 Graphics Note 5, AN-524**, 1988.

National Semiconductor, **NS32CG16 Supplement to the Series 32000 Programmer's Reference Manual**, 1988.

Bresenham's SLICE Algorithm:

1. INITIALIZE PARAMETERS, MAKE NECESSARY ROTATIONS
2. OUTPUT INITIAL RUN LENGTH (H_O) IN PROPER OCTANT DIRECTION
   MOVE DIAGONALLY IN APPROPRIATE DIRECTION TO START OF NEXT RUN LENGTH
3. OUTPUT INTERMEDIATE RUN LENGTHS
   COUNT=COUNT−1
   IF COUNT ≤ 0 GOTO 4.
   IF TESTVAR < 0 H=Q−1 AND TESTVAR=TESTVAR+2*R
   ELSE H=Q AND TESTVAR=TESTVAR+2*R−2*DELB
   OUTPUT RUN LENGTH OF LENGTH H IN PROPER DIRECTION
   MOVE DIAGONALLY IN PROPER DIRECTION
   GOTO 3.
4. OUTPUT FINAL RUN LENGTH OF LENGTH H_F
5. END

INITIALIZED PARAMETERS

DELA = MAXIMUM OF {|DELX|,|DELY|}

DELB = MINIMUM OF {|DELA|,DELA-MINIMUM{|DELX|,|DELY|}}

Q = LOWER[DELA/DELB]

R = DELA−DELB*Q

M = LOWER[Q/2]

N = R (IF Q EVEN)
N = R+DELB (IF Q ODD)

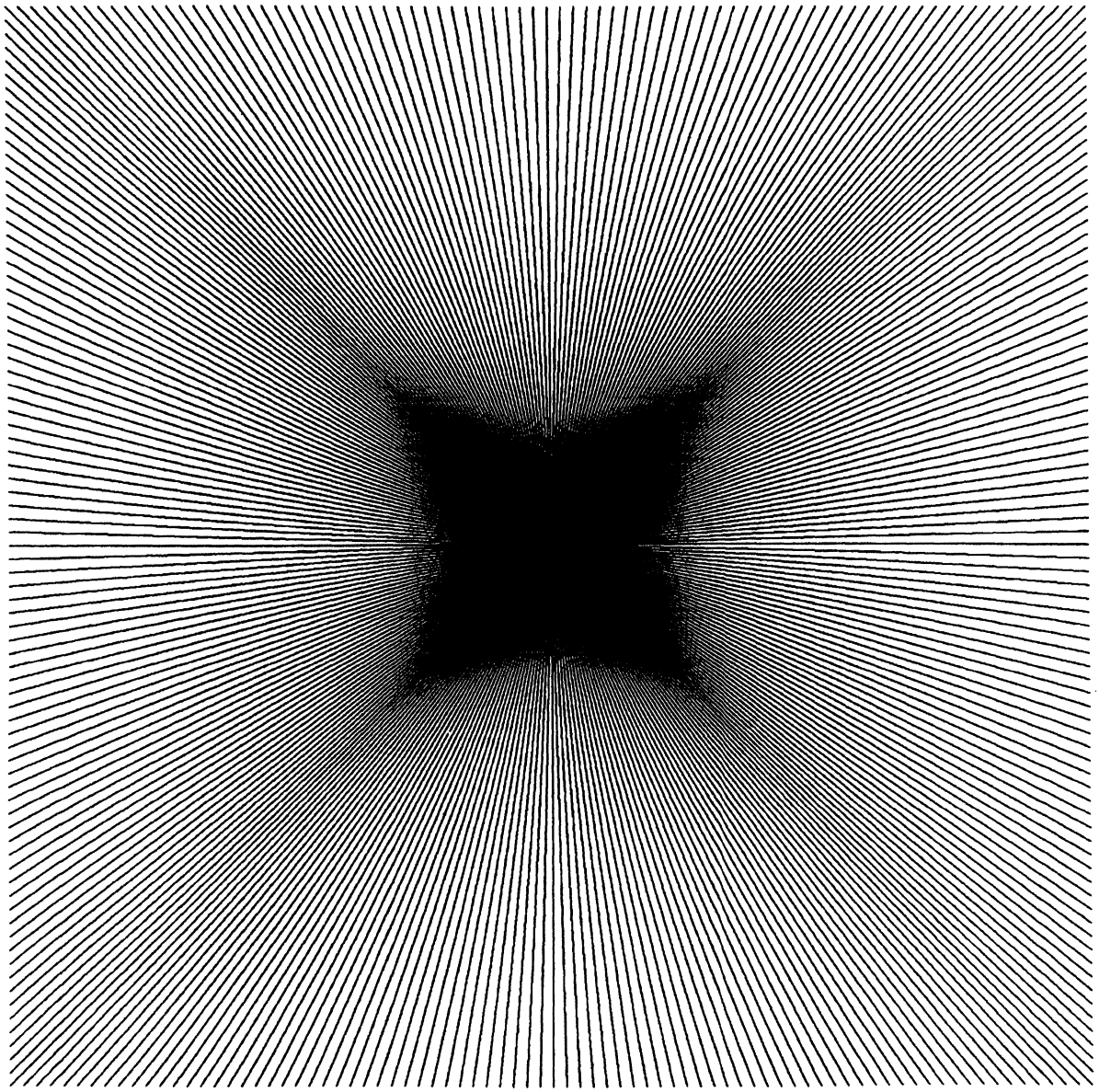
H_O = M (IF DELY≥0 OR N<>0)
H_O = M−1 (IF DELY<0 AND N=0)

H_F = M (IF DELY<0 OR N<>0)
H_F = M−1 (IF DELY ≥0 AND N=0)

COUNT = DELB

$TESTVAR_0$ = N+2*R−2*DELB (IF DELY≥0)
$TESTVAR_0$ = N+2*R−2*DELB−1 (IF DELY<0)

Graphics Image (2000 x 2000 Pixels), 300 DPI



TL/EE/9663–6

FIGURE 6. Star-Burst Benchmark
This Star-Burst image was done on a 2k x 2k pixel bit map. Each line is 2k pixels in length and passes through the center of the image, bisecting the square. The lines are 25 pixel units apart, and are drawn using the DRAW_LINE.S routine. There are a total of 160 lines. The total time for drawing this Star-Burst is 1.0s on 15 MHz NS32CG16.

TIMING APPENDIX

A. PEAK RAW PERFORMANCE AT 15 MHz

Function	Rate*
Horizontal Line (SBITS)	9 MBits/s
Horizontal Line (MOVMP)	60 MBits/s
Vertical Line (SBITPS)	440 kBits/s

*Raw performance does not include any register set-up, branching or other software set-up overhead.

B. TRIVIAL LINES (Using 1k x 1k Bit Map Grid)

	Pixels/Line	Vectors/Sec	Pixels/Sec	Comments**
Horizontal:	1000	13,361	13,361,838	Uses BIGSET.S with MOVMP.
	100	24,136	2,413,593	Uses SBITS only.
	10	45,687	456,870	Uses SBITS only.
Vertical and Diagonal:	1000	424	424,000	Uses SBITPS.
	100	3,975	397,460	
	10	24,491	244,910	

**Pixels/Sec and Vectors/Sec are measured from start of DRAW_LINE.S only. The 23.128 μs driver overhead was not included in these measurements.

C. ALL LINES (Using the "Star-Burst" Benchmark and the SLICE Algorithm)

Pix/Vector	Vectors/Sec	Pixels/Sec	Total Time*	Comments**
1000	318	318,165	0.8s	250 Lines in Star-Burst
100	2,811	281,118	0.019s	50 Lines in Star-Burst
10	14,549	145,490	0.001s	10 Lines in Star-Burst
Avg. Set-up Time Per Line (Measured from Start of DRAW_LINE Only): 37 μs				

D. ALL LINES (Using Original BRESENHAM Iterative Method with SBIT and the Star-Burst Benchmark)

Pix/Vector	Vectors/Sec	Pixels/Sec	Total Time*	Comments**
1000	163	162,746	1.5s	250 Lines in Star-Burst
100	1,568	158,332	0.033s	50 Lines in Star-Burst
10	11,547	127,021	0.001s	10 Lines in Star-Burst
Avg. Set-up Time Per Line (Measured for Line Drawing Routine Only): 30 μs				

The Bresenham program used for the above table can be found in the Series 32000® Graphics Application Note 5.

*Total time is measured from start of execution to finish. It includes all line drawing pre-processing, set-up and branching, and it includes all driver overhead of DRIVER.C and START.C. This time is a good indication of the pages per minute for the complete Star-Burst benchmark. Vectors/Sec and Pixels/Sec are measured from start of DRAW_LINE.S only. The 23.712 μs overhead was not included in these measurements.

**Star-Burst benchmark draws an equal number of lines in each octant. DRIVER.C creates vectors that form the Star-Burst image, passing these vectors to DRAW_LINE.S as they are created. The bit map image can then be downloaded to a printer for a hard copy, as in *Figure 6*.

TIMING APPENDIX TABLE E

Measurement Point	Measured Time/Vector*	Test Vector Used	Octant of Test Vector (Refer to *Figure 5*) And Length of Vector	Comments
T1	23.128 μs	Any Non-Calculated	Any Octant, Any Length	Overhead of entry into DRAW_LINE when not calculating endpoints of line. Application dependent.
	23.712	STAR-BURST	All Octants, 1000 Pixels	Overhead of entry into DRAW_LINE when calculating the STAR-BURST vectors. Application dependent.
T2	40.056	(0,0,0,999)	Vertical, 1000 Pixels/Vector	Average overhead per vertical line to start of line draw instruction (SBITPS).
T3	41.780	(0,999,0,0)	Vertical, 1000 Pixels/Vector	Average overhead per vertical line with negative slope to start of line draw instruction.
T4	40.884	(0,0,999,0)	Horizontal, 1000 Pix/Vect	Average overhead per horizontal line to start of line draw instruction. (SBITS and BIGSET).
	43.912	(999,0,0,0)	Same	Same as above with negative delta × value.
T5	44.532	(0,0,999,999)	Diagonal, 1000 Pix/Vect	Average overhead per diagonal line to start of line draw instruction (SBITPS).
T6	45.356	(0,999,999,0)	Same	Same as above for diagonal line with negative delta × value.
T7	71.164	(0,0,999,10)	Octant 1 1000 Pix/Vect	Average overhead per line to first run length slice of the SLICE algorithm for octant 1.
T8	87.476 75.572 75.568	(0,0,999,10) (0,0,99,10) (0,0,9,2)	Octant 1 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T9	100.348μs 88.444 88.436	(0,0,999,10) (0,0,99,10) (0,0,9,2)	Octant 1 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.
T10	71.856	(0,0,9,8)	Octant 2 10 Pix/Vect	Average overhead per line to first run length. Not dependent on vector length.
T11	79.632 80.040 84.180	(0,0,999,800) (0,0,99,80) (0,0,9,8)	Octant 2 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T12	89.060 89.476 105.376	(0,0,999,800) (0,0,99,80) (0,0,9,8)	Octant 2 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.
T13	73.024	(500,0,700,999)	Octant 3 1000 Pix/Vect	Average overhead per line to first run length. Not dependent on the vector length.
T14	80.736 80.872 80.116	(500,0,700,999) (50,0,70,99) (5,0,7,9)	Octant 3 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T15	89.888 90.020 89.268	(500,0,700,999) (50,0,70,99) (5,0,7,9)	Octant 3 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.
T16	73.712	(10,0,990,999)	Octant 4 1000 Pix/Vect	Average overhead per line to first run length. Not dependent on the vector length.
T17	137.532 81.148 78.256	(10,0,999,999) (10,0,90,99) (2,0,8,9)	Octant 4 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line through first run length of the SLICE algorithm. Dependent on the vector length.
T18	147.236 90.856 87.956	(10,0,999,999) (10,0,90,99) (2,0,8,9)	Octant 4 1000 Pix/Vect 100 Pix/Vect 10 Pix/Vect	Average overhead per 1000, 100 and 10 pixel line to start of main loop of SLICE algorithm. Dependent on the vector length.

*Each time was measured from start of benchmark execution to the Tx marker in the DRAW_LINE.S program. Thus, the overhead of the calling routine to the DRAW_LINE routine is T1 = 23.712 μs for the STAR-BURST benchmark. All programs used for timing are included in the Appendix. All times given above are for a 1k x 1k bit map.

```
/* This program draws a line in a defined bit map using Bresenham's */
/* SLICE algorithm. */

#include<stdio.h>
#define xbytes  25Ø
#define warp    2ØØØ
#define maxy    1999
unsigned char   bit_map[xbytes*maxy];
extern unsigned char    sbitstab[];

draw_line(xs,ys,xt,yt)

int     xs,ys,xt,yt;

{
        int     bit,i,j,delx,dely,dela,delb,
                hf,h,hØ,testvar,q,r,m,
                n,count,xinc,yinc;

        delx=xt-xs;
        dely=yt-ys;

        if (xt-xs<Ø){
                xs=xt;
                ys=yt;
                delx=abs(delx);
                dely= -dely;
        }
        bit=xs+ys*warp;
        if (delx==Ø){
                if (dely>=Ø){
                        sbitps(bit_map,bit,dely,warp);
                        return;
                }
                else(
                        sbitps(bit_map,bit,abs(dely),-warp);
                        return;
                }
        }
        if (dely==Ø){
                sbits(bit_map,bit,delx,sbitstab);
                return;
        }
        if (abs(delx)==abs(dely)){
                if(delx*dely>=Ø){
                        sbitps(bit_map,bit,abs(dely),warp+1);
                        return;
                }
                else {
                        sbitps(bit_map,bit,delx,-warp+1);
                        return;
                }
        }
if (abs(delx)>abs(dely)){

        if (abs(dely)<(delx-abs(dely)))
        {
                dela=delx;
                delb=abs(dely);
                xinc=1;
                if (dely>=Ø)
                        yinc=warp;
                else
                        yinc= -warp;

                q=dela/delb;
```

TL/EE/9663–7

```
        r=dela-delb*q;
        m=q/2;
        if (q-2*(q/2)==Ø)
                n=r;
        else
                n=r+delb;

        if ((dely>=Ø)||(n!=Ø))
                hØ=m;
        else
                hØ=m-1;

        if ((dely<Ø)||(n!=Ø))
                hf=m;
        else
                hf=m-1;

        count=delb;

        if(dely>=Ø)
                testvar=n+2*r-2*delb;
        else
                testvar=n+2*r-2*delb-1;
        sbits(bit_map,bit,hØ+1,sbitstab);
        bit=bit+hØ+yinc+xinc;

        for(i=count-1;i>Ø;i--) {
                if (testvar<Ø){
                        h=q-1;
                        testvar+=2*r;
                }
                else {
                        h=q;
                        testvar+=2*r-2*delb;
                }
                sbits(bit_map,bit,h+1,sbitstab);
                bit=bit+h+yinc+xinc;

        }
        sbits(bit_map,bit,hf,sbitstab);
        return;
}
else{
        dela=abs(delx);
        delb=dela-abs(dely);
        xinc=1;
        if(dely>=Ø)
                yinc=warp;
        else
                yinc= -warp;
        q=dela/delb;
        r=dela-delb*q;
        m=q/2;
        if (q-2*(q/2)==Ø)
                n=r;
        else
                n=r+delb;
        if ((dely>=Ø)||(n!=Ø))
                hØ=m;
        else
                hØ=m-1;

        if ((dely<Ø)||(n!=Ø))
                hf=m;
        else
                hf=m-1;
```

TL/EE/9663-8

```
                count=delb;

                if(dely>=Ø)
                        testvar=n+2*r-2*delb;
                else
                        testvar=n+2*r-2*delb-1;
                sbitps(bit_map,bit,hØ+1,yinc+1);
                bit=bit+hØ+hØ*yinc+1;

                for(i=count-1;i>Ø;i--) {
                        if (testvar<Ø){
                                h=q-1;
                                testvar+=2*r;
                        }
                        else {
                                h=q;
                                testvar+=2*r-2*delb;
                        }
                        sbitps(bit_map,bit,h+1,yinc+1);
                        bit=bit+h+yinc*h+1;
                }
                sbitps(bit_map,bit,hf+1,yinc+1);
                return;
        }
}
else{
        if (abs(delx)<(abs(dely)-abs(delx))){
                dela=abs(dely);
                delb=abs(delx);
                yinc=1;
                if(dely>Ø)
                        xinc=warp;
                else
                        xinc= -warp;

                q=dela/delb;
                r=dela-delb*q;
                m=q/2;
                if (q-2*(q/2)==Ø)
                        n=r;
                else
                        n=r+delb;
                if ((dely>=Ø)||(n!=Ø))
                        hØ=m;
                else
                        hØ=m-1;

                if ((dely<Ø)||(n!=Ø))
                        hf=m;
                else
                        hf=m-1;
                count=delb;

                if(dely>=Ø)
                        testvar=n+2*r-2*delb;
                else
                        testvar=n+2*r-2*delb-1;
                sbitps(bit_map,bit,hØ+1,xinc);
                bit=bit+yinc+(1+hØ)*xinc;
                for(i=count-1;i>Ø;i--) {

                        if (testvar<Ø){
                                h=q-1;
                                testvar+=2*r;
                        }
                        else {
```

TL/EE/9663-9

```
                                h=q;
                                testvar+=2*r-2*delb;
                        }
                        sbitps(bit_map,bit,h+1,xinc);
                        bit=bit+yinc+xinc*(1+h);
                }
                sbitps(bit_map,bit,hf+1,xinc);
                return;
        }
        else{
                dela=abs(dely);
                delb=dela-abs(delx);
                yinc=1;
                if(dely>Ø)
                        xinc=warp;
                else
                        xinc= -warp;

                q=dela/delb;
                r=dela-delb*q;
                m=q/2;
                if (q-2*(q/2)==Ø)
                        n=r;
                else
                        n=r+delb;
                if ((dely>=Ø)||(n!=Ø))
                        hØ=m;
                else
                        hØ=m-1;

                if ((dely<Ø)||(n!=Ø))
                        hf=m;
                else
                        hf=m-1;
                count=delb;

                if(dely>=Ø)
                        testvar=n+2*r-2*delb;
                else
                        testvar=n+2*r-2*delb-1;
                sbitps(bit_map,bit,hØ+1,xinc+1);
                bit=bit+hØ+(1+hØ)*xinc;
                for(i=count-1;i>Ø;i--) {

                        if (testvar<Ø){
                                h=q-1;
                                testvar+=2*r;
                        }
                        else {
                                h=q;
                                testvar+=2*r-2*delb;
                        }
                        sbitps(bit_map,bit,h+1,xinc+1);
                        bit=bit+h+xinc*(1+h);
                }
                sbitps(bit_map,bit,hf,xinc+1);
                return;
        }
}
}
```

TL/EE/9663–10

```
#       National Semiconductor Corporation.
#       CTP version 2.4     -- draw_line.s --      Tue Nov 17 13:2ø:24 1987
#       compilation options: -O -S -KC332 -KFø81 -KB4
#
.file   "draw_line.s"
        .comm _bit_map,49975ø
        .set    WARP,2øøø
        .globl  _draw_line
        .globl  _sbitstab
        .align 4
_draw_line:
        enter    [r3,r4,r5,r6,r7],12
# T1
        movd     16(fp),r4       # xf
        movd     8(fp),r5        # xs
        subd     r5,r4           # delx
        movd     2ø(fp),r6       # yf
        movd     12(fp),r7       # ys
        subd     r7,r6           # dely
        cmpqd    $(ø),r4         # ø>delx
        ble      .VERT
        movd     16(fp),r5       # xf=new xs
        movd     2ø(fp),r7       # yf=new ys
        absd     r4,r4           # delx=|delx|
        negd     r6,r6           # dely=(-dely)
.VERT:
        movd     r7,r1           # ys
        muld     $WARP,r1        # ys*warp
        addd     r5,r1           # bit=ys*WARP+xs
        cmpqd    $(ø),r4         # delx=ø?
        bne      .HORZ
        cmpqd    $(ø),r6         # dely>ø?
        bgt      .VNEG           # if no then warp is neg
        addr     _bit_map,rø     # set registers for sbitps
        movd     r6,r2           # r2=dely=length of line
        movd     $WARP,r3        # r3=warp
# T2
        sbitps                   # draw line
        exit     [r3,r4,r5,r6,r7]
        ret      $(ø)
        .align 4
.VNEG:
        addr     _bit_map,rø     # set reg's for sbitps
        movd     r6,r2           # r2=(-dely)
        absd     r2,r2           # r2=dely=length of line
        movd     $(-WARP),r3     # r3=warp
# T3
        sbitps                   # draw line
        exit     [r3,r4,r5,r6,r7]
        ret      $(ø)
        .align 4
.HORZ:
        cmpqd    $(ø),r6         # dely=ø?
        bne      .DIAG
        addr     _bit_map,rø     # set reg's for sbits
        movd     r4,r2           # r4=delx=length
        addr     _sbitstab,r3    # table pointer
# T4
        sbits                    # try sbits
        bfc      ok              # if not more than 25, skip it
        cmpd    $2øø,r2
        blt     bigs1
        addr    25,r2
        .align  4
alp1:   sbits
        addd    r2,r1
```

TL/EE/9663-11

```
        subd    r2,r4
        cmpd    r2,r4
        blt     alp1
        .align  4
        movd    r4,r2
        sbits
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
bigs1:  bsr     bigset
ok:     exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.DIAG:
        absd    r6,r5           # r5=|dely|
        cmpd    r5,r4           # |dely|=delx?
        bne     .SLOPELT1
        cmpqd   $(Ø),r6         # dely>Ø?
        bgt     .DNEG
        addr    _bit_map,rØ     # set reg's for sbitps
        movd    r4,r2           #  r2=delx=length
        movd    $WARP + 1,r3    # r3=warp+1 for diag
# T5
        sbitps                  # draw line
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.DNEG:
        addr    _bit_map,rØ     # set reg's for sbitps
        movd    r4,r2           # r2=delx=lenght
        movd    $-WARP + 1,r3   # r3=warp-1 for neg slope
# T6
        sbitps                  # draw line
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.SLOPELT1:                      # slope less than 1
        cmpd    r5,r4           # |dely|>delx?
        bgt     .SLOPEGT1
        movd    r4,r2           # r2=delx
        subd    r5,r2           # delx-|dely|
        cmpd    r5,r2           # |dely|>delx-|dely|?
        bgt     .OCTANT2        # if no, start octant1 else octant2
        cmpqd   $(Ø),r6         # dely>Ø?
        bgt     .NEGWARP
        addr    WARP,-4(fp)     # pos slope then warp=positive
        br      .INIT1
        .align 4
.NEGWARP:
        addr    -WARP,-4(fp)    # warp=negative for neg slope
.INIT1:                         # calculate parameters
        movd    r4,r3           # delx=dela  |dely|=delb
        quow    r5,r3           # dela/delb=q
        movd    r3,rØ           # calc m
        ashd    $-1,rØ          # m=q/2
        movd    r3,r2           # calc r
        mulw    r5,r2           # delb*q
        subd    r2,r4           # r=dela-delb*q
        movd    r4,r2           # set r2 = r
        tbitb   $Ø,r3           # is r3 odd?
        bfc     .INIT2          # yes, n = r
        addd    r5,r2           # n=r+delb
        .align 4
.INIT2:
        movd    r2,r7           # pop n
        movd    r3,tos          # push q on stack
        movd    rØ,r2           # r2=m=hØ
```

TL/EE/9663–12

```
        movd    rØ,-8(fp)       # mem=m=hpartb
        cmpqd   $(Ø),r7         # n=Ø?
        bne     .INIT3
        cmpqd   $(Ø),r6         # dely>Ø?
        blt     .INIT4
        addqd   $-1,r2          # hØ=m-1
        br      .INIT3
.INIT4:
        subd    $1,-8(fp)       # hpartb=m-1
.INIT3:
        addqd   $1,r2           # takes care of dashes
        addr    _bit_map,rØ     # set reg's for sbits
        addr    _sbitstab,r3    # hØ=r2 bit=r1
# T7
        sbits
        bfc     .2DONE          # set bits if less than 25
        cmpd    $2ØØ,r2
        blt     BIGSET1
        movd    r5,tos
        movd    r2,r5
        movd    $25,r2
.2DRAW25:
        subd    r2,r5
        sbits
        addd    r2,r1
        cmpd    r2,r5
        blt     .2DRAW25
        movd    r5,r2
        movd    tos,r5
        sbits
        br      .2DONE
BIGSET1:
        bsr     bigset
.2DONE:
# T8
        addd    r2,r1           # bit=bit+hØ+1
        addd    -4(fp),r1       # bit=bit+hØ+1+warp
        addd    r4,r4           # 2*r
        movd    r5,r3           # save delb
        addd    r5,r5           # delb*2
        addd    r4,r7           # n=n+2*r
        subd    r5,r7           # testvar=n+2*r+delb*2
        cmpqd   $(Ø),r6         # dely>Ø
        blt     .INIT5
        addqd   $-1,r7          # testvar-1
.INIT5:
        movd    tos,r2          # r2=q=h=run length
        addqd   $1,r2           # smoothes out line
        movd    r3,tos          # push delb=count
        addr    _sbitstab,r3    # set reg's for sbits
        addr    _bit_map,rØ
        movd    -4(fp),r6       # warp
        addqd   $-1,tos         # count=count-1
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        bge     .LASTRUN
.MAINLOOP:                      # Bresenham slice algorithm
# T9
        cmpqd   $(Ø),r7         # testvar>Ø?
        ble     .CASE2
        addqd   $-1,r2          # h=q-1
        addd    r4,r7           # testvar=testvar+2*r
        sbits
        bfc     .3DRAWLAST      # set bits if less than 25
        cmpd    $2ØØ,r2
        blt     BIGSET3
        movd    r2,tos
```

TL/EE/9663-13

```
        movd    r5,tos
        movd    r2,r5
        movd    $25,r2
.3DRAW25:
        subd    r2,r5
        sbits
        addd    r2,r1
        cmpd    r2,r5
        blt     .3DRAW25
        movd    r5,r2
        sbits
        addd    r2,r1
        movd    tos,r5
        movd    tos,r2
        br      .3DONE
BIGSET3:
        bsr     bigset
.3DRAWLAST:
        addd     r2,r1          # update bit
.3DONE:
        addd     r6,r1          # bit=bit+warp+h+1
        addd     $1,r2          # exit h
        addqd    $(-1),tos      # count=count-1
        cmpqd    $(Ø),Ø(sp)     # count=Ø?
        blt      .MAINLOOP
        .align 4
.LASTRUN:
        cmpqd    $(Ø),tos       # pop stack
        movd     -8(fp),r2      # hpartb=last run length
        sbits
        bfc      .4DONE         # set bits if less than 25
        cmpd    $2ØØ,r2
        blt     BIGSET4
        movd    r2,tos
        movd    r5,tos
        movd    r2,r5
        movd    $25,r2
.4DRAW25:
        subd    r2,r5
        sbits
        addd    r2,r1
        cmpd    r2,r5
        blt     .4DRAW25
        movd    r5,r2
        sbits
        addd    r2,r1
        movd    tos,r5
        movd    tos,r2
        br      .4DONE
BIGSET4:
        bsr     bigset
.4DONE:
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.CASE2:
        addd     r4,r7          # testvar=testvar+2*r
        subd     r5,r7          # testvar=testvar+2*r-2*delb
        sbits
        bfc      .5DRAWLAST     # SET BITS IF LESS THAN 25
        cmpd    $2ØØ,r2
        blt     BIGSET5
        movd    r2,tos
        movd    r5,tos
        movd    r2,r5
        movd    $25,r2
```

TL/EE/9663-14

```
.5DRAW25:
        subd    r2,r5
        sbits
        addd    r2,r1
        cmpd    r2,r5
        blt     .5DRAW25
        movd    r5,r2
        sbits
        addd    r2,r1
        movd    tos,r5
        movd    tos,r2
        br      .5DONE
BIGSET5:
        bsr     bigset
.5DRAWLAST:
        addd     r2,r1          # update bit
.5DONE:
        addd     r6,r1          # bit=bit+warp+h+1
        addqd    $(-1),tos      # update count
        cmpqd    $(Ø),Ø(sp)     # count=Ø?
        blt      .MAINLOOP
        cmpqd    $(Ø),tos       # pop stack
        movd     -8(fp),r2      # hpartb=last run length
        sbits
        bfc      .6DONE         # set bits if less than 25
        bsr     bigset
.6DONE:
        exit     [r3,r4,r5,r6,r7]
        ret      $(Ø)
        .align 4
.OCTANT2:                       # draw line in octant 2
        cmpqd    $(Ø),r6        # dely>Ø?
        bgt      .2NEGWARP
        addr     WARP,-4(fp)    # pos slope then warp=positive
        br       .2INIT1
.2NEGWARP:
        addr     -WARP,-4(fp)   # warp=negative for neg slope
.2INIT1:                        # calculate parameters
        movd     r4,r3          # dela=delx
        movd     r2,r5          # delb=delx-|dely|
        quow     r5,r3          # dela/delb=q
        movd     r3,rØ          # calc m
        ashd     $-1,rØ         # m=q/2
        movd     r3,r2          # calc r
        mulw     r5,r2          # delb*q
        subd     r2,r4          # r=dela-delb*q
        movd     r4,r2          # push r on stack
        tbitb    $Ø,r3
        bfc      .2INIT2        # then n=r
        addd     r5,r2          # n=r+delb
        .align 4
.2INIT2:
        movd     r2,r7          # pop n
        movd     r3,tos         # push q on stack
        movd     rØ,r2          # r2=m=hØ
        addqd    $1,r2          # set one extra bit for smoothness
        movd     rØ,-8(fp)      # mem=m=hpartb
        cmpqd    $(Ø),r7        # n=Ø?
        bne      .2INIT3
        cmpqd    $(Ø),r6        # dely>Ø?
        blt      .2INIT4
        subd     $1,r2          # hØ=m-1
        br       .2INIT3
.2INIT4:
        subd    $1,-8(fp)       # hpartb=m-1
.2INIT3:
```

TL/EE/9663-15

```
        addr    _bit_map,rØ     # set reg's for sbits
        movd    -4(fp),r3       # warp=r3 hØ=r2 bit=r1
        addqd   $1,r3           # octant 2 needs diag runs
# T1Ø
        sbitps                  # draw first run length
# T11
        addqd   $1,r1           # update bit in x direction
        subd    r3,r1           # sbitps adds extra warp
        addd    r4,r4           # 2*r
        movd    tos,r2          # q=h=next run length
        addqd   $1,r2           # set extra bit for smoothness
        movd    r5,tos          # push delb=count
        addd    r5,r5           # delb*2
        addd    r4,r7           # n=n+2*r
        subd    r5,r7           # testvar=n+2*r+delb*2
        cmpqd   $(Ø),r6         # dely>Ø
        blt     .2INIT5
        subd    $1,r7           # testvar-1
.2INIT5:
        subd    $1,tos          # count=count-1
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        bge     .2LASTRUN
.2MAINLOOP:                     # Bresenham slice algorithm
# T12
        cmpqd   $(Ø),r7         # testvar>Ø?
        ble     .2CASE2
        subd    $1,r2           # h=q-1
        addd    r4,r7           # testvar=testvar+2*r
        movd    r2,tos          # preserve h
        sbitps                  # draw diag line of length h
        movd    tos,r2          # renew h
        addqd   $1,r1           # update bit in x direction
        subd    r3,r1           # sbitps adds one warp extra
        addd    $1,r2           # exit h to q
        subd    $1,tos          # count=count-1
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        blt     .2MAINLOOP
        .align 4
.2LASTRUN:
        cmpqd   $(Ø),tos        # pop stack
        movd    -8(fp),r2       # hpartb=last run length
        sbitps                  # all other reg's set up
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.2CASE2:
        addd    r4,r7           # testvar=testvar+2*r
        subd    r5,r7           # testvar=testvar+2*r-2*delb
        movd    r2,tos          # preserve h
        sbitps                  # draw line of length h=q
        movd    tos,r2          # renew h
        addqd   $1,r1           # update bit in x direction
        subd    r3,r1           # sbitps adds one warp extra
        subd    $(1),tos        # update count
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        blt     .2MAINLOOP
        cmpqd   $(Ø),tos        # pop stack
        movd    -8(fp),r2       # hpartb=last run length
        sbitps                  # all other reg's set up
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.SLOPEGT1:                      # coordinates are rotated for these lines
        movd    r5,r2           # r2=|dely|
        subd    r4,r2           # |dely|-delx
        cmpd    r4,r2           # delx>|dely|-delx?
```

TL/EE/9663–16

```
        bgt     .2OCTANT2       # if no, start octant1 else octant2
        cmpqd   $(Ø),r6         # dely>Ø?
        bgt     .3NEGWARP
        addr    WARP,-4(fp)     # pos slope then warp=positive
        br      .3INIT1
.3NEGWARP:
        addr    -WARP,-4(fp)    # warp=negative for neg slope
.3INIT1:                        # calculate rotated parameters
        movd    r5,r3           # dela=|dely|
        movd    r4,r5           # delb=delx
        movd    r3,r4           # dela in r4
        quow    r5,r3           # dela/delb=q
        movd    r3,rØ           # calc m
        ashd    $-1,rØ          # m=q/2
        movd    r3,r2           # calc r
        mulw    r5,r2           # delb*q
        subd    r2,r4           # r=dela-delb*q
        movd    r4,r2           # push r on stack
        tbitb   $Ø,r3
        bfc     .3INIT2         # then n=r
        addd    r5,r2           # n=r+delb
        .align 4
.3INIT2:
        movd    r2,r7           # pop n
        movd    r3,tos          # push q on stack
        movd    rØ,r2           # r2=m=hØ
        addqd   $1,r2           # set one extra bit for smoothness
        movd    rØ,-8(fp)       # mem=m=hpartb
        cmpqd   $(Ø),r7         # n=Ø?
        bne     .3INIT3
        cmpqd   $(Ø),r6         # dely>Ø?
        blt     .3INIT4
        subd    $1,r2           # hØ=m-1
        br      .3INIT3
.3INIT4:
        subd    $1,-8(fp)       # hpartb=m-1
.3INIT3:
        addr    _bit_map,rØ     # set reg's for sbits
        movd    -4(fp),r3       # warp=r3 hØ=r2 bit=r1
# T13
        sbitps                  # draw first run length
# T14
        addqd   $1,r1           # update bit in x direction
        addd    r4,r4           # 2*r
        movd    tos,r2          # q=h=next run length
        addqd   $1,r2           # set extra bit for smoothness
        movd    r5,tos          # push delb=count
        addd    r5,r5           # delb*2
        addd    r4,r7           # n=n+2*r
        subd    r5,r7           # testvar=n+2*r+delb*2
        cmpqd   $(Ø),r6         # dely>Ø
        blt     .3INIT5
        subd    $1,r7           # testvar-1
.3INIT5:
        subd    $1,tos          # count=count-1
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        bge     .3LASTRUN
.3MAINLOOP:                     # Bresenham slice algorithm
# T15
        cmpqd   $(Ø),r7         # testvar>Ø?
        ble     .3CASE2
        subd    $1,r2           # h=q-1
        addd    r4,r7           # testvar=testvar+2*r
        movd    r2,tos          # preserve h
        sbitps                  # draw vert line of length h
        movd    tos,r2          # renew h
```

TL/EE/9663-17

```
        addqd   $1,r1           # update bit in x direction
        addd    $1,r2           # exit h to q
        subd    $1,tos          # count=count-1
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        blt     .3MAINLOOP
        .align 4
.3LASTRUN:
        cmpqd   $(Ø),tos        # pop stack
        movd    -8(fp),r2       # hpartb=last run length
        sbitps                  # all other reg's set up
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.3CASE2:
        addd    r4,r7           # testvar=testvar+2*r
        subd    r5,r7           # testvar=testvar+2*r-2*delb
        movd    r2,tos          # preserve h
        sbitps                  # draw line of length h=q
        movd    tos,r2          # renew h
        addqd   $1,r1           # update bit in x direction
        subd    $(1),tos        # update count
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        blt     .3MAINLOOP
        cmpqd   $(Ø),tos        # pop stack
        movd    -8(fp),r2       # hpartb=last run length
        sbitps                  # all other reg's set up
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.2OCTANT2:                      # draw line in octant 2
        cmpqd   $(Ø),r6         # dely>Ø?
        bgt     .4NEGWARP
        addr    WARP,-4(fp)     # pos slope then warp=positive
        br      .4INIT1
.4NEGWARP:
        addr    -WARP,-4(fp)    # warp=negative for neg slope
.4INIT1:                        # calculate parameters
        movd    r5,r3           # dela=delx
        movd    r5,r4           # dela into r4
        movd    r2,r5           # delb=delx-|dely|
        quow    r5,r3           # dela/delb=q
        movd    r3,rØ           # calc m
        ashd    $(-1),rØ        # m=q/2
        movd    r3,r2           # calc r
        mulw    r5,r2           # delb*q
        subd    r2,r4           # r=dela-delb*q
        movd    r4,r2           # push r on stack
        tbitb   $Ø,r3
        bfc     .4INIT2         # then n=r
        addd    r5,r2           # n=r+delb
        .align 4
.4INIT2:
        movd    r2,r7           # pop n
        movd    r3,tos          # push q on stack
        movd    rØ,r2           # r2=m=hØ
        addqd   $1,r2           # set one extra bit for smoothness
        movd    rØ,-8(fp)       # mem=m=hpartb
        cmpqd   $(Ø),r7         # n=Ø?
        bne     .4INIT3
        cmpqd   $(Ø),r6         # dely>Ø?
        blt     .4INIT4
        subd    $1,r2           # hØ=m-1
        br      .4INIT3
.4INIT4:
        subd    $1,-8(fp)       # hpartb=m-1
.4INIT3:
```

TL/EE/9663-18

```
        addr    _bit_map,rØ     # set reg's for sbits
        movd    -4(fp),r3       # warp=r3 hØ=r2 bit=r1
        addqd   $1,r3           # octant 2 needs diag runs
# T16
        sbitps                  # draw first run length
# T17
        subd    $1,r1           # update bit
        addd    r4,r4           # 2*r
        movd    tos,r2          # q=h=next run length
        addqd   $1,r2           # set extra bit for smoothness
        movd    r5,tos          # push delb=count
        addd    r5,r5           # delb*2
        addd    r4,r7           # n=n+2*r
        subd    r5,r7           # testvar=n+2*r+delb*2
        cmpqd   $(Ø),r6         # dely>Ø
        blt     .4INIT5
        subd    $1,r7           # testvar-1
.4INIT5:
        subd    $1,tos          # count=count-1
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        bge     .4LASTRUN
.4MAINLOOP:                     # Bresenham slice algorithm
# T18
        cmpqd   $(Ø),r7         # testvar>Ø?
        ble     .4CASE2
        subd    $1,r2           # h=q-1
        addd    r4,r7           # testvar=testvar+2*r
        movd    r2,tos          # preserve h
        sbitps                  # draw diag line of length h
        movd    tos,r2          # renew h
        subd    $1,r1           # sbitps adds one warp extra
        addd    $1,r2           # exit h to q
        subd    $1,tos          # count=count-1
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        blt     .4MAINLOOP
        .align 4
.4LASTRUN:
        cmpqd   $(Ø),tos        # pop stack
        movd    -8(fp),r2       # hpartb=last run length
        addqd   $1,r2
        sbitps                  # all other reg's set up
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.4CASE2:
        addd    r4,r7           # testvar=testvar+2*r
        subd    r5,r7           # testvar=testvar+2*r-2*delb
        movd    r2,tos          # preserve h
        sbitps                  # draw line of length h=q
        movd    tos,r2          # renew h
        subd    $1,r1           # sbitps adds one warp extra
        subd    $(1),tos        # update count
        cmpqd   $Ø,Ø(sp)        # count=Ø?
        blt     .4MAINLOOP
        .align 4
        cmpqd   $(Ø),tos        # pop stack
        movd    -8(fp),r2       # hpartb=last run length
        addqd   $1,r2
        sbitps                  # all other reg's set up
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
```

TL/EE/9663–19

```
# BIGSET.S uses MOVMP and the OR instructions to set long horizontal lines
#
        .globl  bigset
bigset: save    [rØ,r1,r2,r3,r4,r5,r6]  #save registers we will affect
        movd    r1,r4                   #get current bit offset
        ashd    $-3,r4                  #divide by eight to get byte offset
        addd    r4,rØ                   #add in base. rØ is new base pointer
        andd    $7,r1                   #mask off msb's of bit pointer to
                                        #get bit = bit offset mod 8

 #Now we have true base address and bit offset within base. Now we will move
 #to double word alignment. This speeds up the MOVMPD for long bit sequences.

        movqd   3,r4                    #place mask in r4
        andd    rØ,r4                   #get low two bits of address
        xorb    $3,r4                   #and get bytes left to alignment
        addqd   1,r4                    #rem += 1 (for the byte we are on)
        ashd    $3,r4                   #rem *= 8 to get bits to alignment
        subd    r1,r4                   #subtract current bit offset
        cmpd    r4,r2                   #is this more than number of bits left
        bge     shrt                    #it is, do it the short way
        cmpd    $32,r4                  #if we are already double aligned, go
                                        #do the MOVMPD
        beq     mvm
        movd    r1,r5                   #calculate index into table
        lshd    $5,r5                   #index = 32 * bit offset
        addd    r4,r5                   #index += run length
        ord     r3[r5:d],Ø(rØ)          #or in required bits
        bicb    $3,rØ                   #clear last two bits, and
        addqd   4,rØ                    #bump to next double
        subd    r4,r2                   #zap sp'd bits off
mvm:    movd    r2,r4                   #save run length for a minute
        movd    r3,r5                   #and save pointer to table
        ashd    $-5,r2                  #r1 = r1 / 32 = number of doubles
        movd    1Ø2Ø(r3),r3             #get source pattern from table
        movqd   4,r1                    #increment is r1
        movmpd                          # yes, use instruction
        andd    $Øx1f,r4                #mask off all but last 32 bits
        ord     r5[r4:d],Ø(rØ)          #insert the last few bits
        restore [rØ,r1,r2,r3,r4,r5,r6]  #restore saved registers
        ret     $Ø

        .align  4
shrt:   cmpb    $32,r2                  #check to see if it is exactly
        beq     shrt1                   #32 bits. If it is, branch.
        movd    r1,r4                   #calculate index into table
        lshd    $5,r4                   #index = 32 * bit offset
        addd    r2,r4                   #index += run length
        ord     r3[r4:d],Ø(rØ)          #or in required bits
        restore [rØ,r1,r2,r3,r4,r5,r6]  #restore saved registers
        ret     $Ø
        .align  4
shrt1:  movd    1Ø2Ø(r3),Ø(rØ)          #copy last entry of table
        restore [rØ,r1,r2,r3,r4,r5,r6]  #(all 32 bits) and restore
        ret     $Ø
```

TL/EE/9663-20

```
 /* Program driver.c feeds line vectors to LINE_DRAW.S forming Star-Burst.   */

#include <stdio.h>
#define xbytes  25Ø
#define maxx    1999
#define maxy    1999

unsigned char   bit_map[xbytes*maxy];

main()

{
    int i,count;

/* generate Star-Burst image */

        for (count=1;count<=1ØØØ;test++){

                for (i=Ø;i<=maxy;i+=25)
                        draw_line(Ø,i,maxx,maxy-i);
                for (i=Ø;i<=maxx;i+=25)
                        draw_line(i,maxy,maxx-i,Ø);
        }

}
```

TL/EE/9663-21

```
/* Start timer and call main procedure of DRIVER.C to draw lines */

start() {
    long *timer = (long *) Øx6ØØ;
    *timer = Ø;          /* write a zero to timer location */
                main(Ø,Ø);       /* Show argc as zero, argv ->Ø */
    return(*timer);      /* return, in rØ, the current time */
}
```

TL/EE/9663-22

Drawing Circles with the NS32CG16; NS32CG16 Note 1

National Semiconductor
Application Note 523
Dave Rand

1.0 INTRODUCTION

The NS32CG16 is a 32-bit CMOS, graphics oriented processor. It is software compatible with other Series 32000® CPUs, with new instructions for high-speed graphics. The NS32CG16 is designed specifically for page-oriented printing technologies such as laser, LCS, LED, Ion-Deposition, and Ink Jet.

In this applications note, a method for high-speed circle generation will be described, using an optimized version of Bresenham's circle algorithm.

2.0 DESCRIPTION

A circle can be described by the center coordinates (xc, yc), the radius (r), and the width (w). With the Pythagorean theorem, pixels along the path described by the equation:

$$(x - xc)^2 + (y - yc)^2 = r^2$$

can be set for a width of w perpendicular to the tangent of the arc.

This, however, involves substantial computation for each point on the line. Even taking advantage of the symmetry of circles, a large number of instructions must be executed to calculate the path.

Bresenham's circle algorithm works by determining which of two pixels are nearer the actual circle at each step. Then, using symmetry, eight points on the circle's path can be determined. Applying the width (w) to each of these eight points yields a displayed (or imaged) circle. For the actual derivation of Bresenham's algorithm, see *Reference 1*, and *Reference 2*. This derivation was done by J. Michener.

Bresenham's algorithm can be implemented in the following manner:

1. Select the first position for display as
$$(x_1, y_1) = (0, r)$$
2. Calculate the first parameter as
$$p_1 = 3 - 2r$$
If $p_1 < 0$, the next position is $(x_1 + 1, y_1)$. Otherwise, the next position is $(x_1 + 1, y_1 - 1)$.
3. Continue to increment the x coordinate by unit steps, and calculate each succeeding parameter p from the preceding one. If for the previous parameter we found that $p_i < 0$ then
$$p_{i+1} = p_i + 4x_i + 6$$
Otherwise (for $p_i \geq 0$),
$$p_{i+1} = p_i + 4(x_i - y_i) + 10$$
Then, if $p_{i+1} < 0$ the next point selected is $(x_i + 2, y_{i+1})$. Otherwise, the next point is $(x_i + 2, y_{i+1} - 1)$. The y coordinate is $y_{i+1} = y_i$, if $p_i < 0$ or $y_{i+1} = y_i - 1$, if $p_i \geq 0$.
4. Repeat the procedures in step 3 until the x and y coordinates are equal.

3.0 IMPLEMENTATION

With the path of the circle described, the pixels along the path can be set using the basic symmetry of the circle. Following is an example of Bresenham's circle algorithm in the C language, based on Michener's derivation.

```
circle(xc,yc,radius,width)
register unsigned int xc,yc,radius,width;
{
    register int y, x, p;
    x = 0;
    y = radius;
    p = 3 - 2 * radius;
    while (x < y) {
        setgrp(xc,yc,x,y,width);
        if (p < 0)
            p += 4 * x + 6;
        else {
            p += 4 * (x - y) + 10;
            y--;
        }
        x++;
    }
    if (y == x)
        setgrp(xc,yc,x,y,width);
}
```

TL/EE/9664–1

```
setgrp(xc,yc,x,y,width)
register int xc,yc,x,y,width;
{
    if ((y - x) <= (width / 2) {
        hset(xc + y, yc + x,width);
        hset(xc - y, yc + x,width);
        hset(xc + y, yc - x,width);
        hset(xc - y, yc - x,width);
        vset(xc + x, yc + y,width);
        vset(xc - x, yc + y,width);
        vset(xc + x, yc - y,width);
        vset(xc - x, yc - y,width);
    }
    vset(xc + y, yc + x,width);
    vset(xc - y, yc + x,width);
    vset(xc + y, yc - x,width);
    vset(xc - y, yc - x,width);
    hset(xc + x, yc + y,width);
    hset(xc - x, yc + y,width);
    hset(xc + x, yc - y,width);
    hset(xc - x, yc - y,width);
}
```

TL/EE/9664–2

The *setgrp* routine in the previous example uses symmetry to set eight points of the circle. *Setgrp* has a special case to handle the boundaries of the eight sections. When the distance between the boundaries is less than half the width of the circle, both vertical and horizontal lines are imaged for each section. The *vset* routine sets *width* pixels vertically in the image, centered around the second argument. The *hset* routine sets *width* pixels horizontally, centered around the first argument. Since these cases are so well defined, the NS32CG16 instructions *SBITPS* and *SBITS* are used for these routines.

The NS32CG16 implementation is very much like the C version, but is optimized for speed. Note the use of the *ADDR* instruction to do the two p_i computations, each in one line of 32000 assembly code.

```
        .data
xwarp:  equ     2544            #bits of xwarp to get to next scan
        .comm   _page,4
hlfwdth:double  0
        .text
#
#Bresenham's circle algorithm, as expressed in "Computer Graphics" by
#Donald Hearn and M. Pauline Baker (1986, Prentice-Hall,
#ISBN 0-13-165382-2)
#
#       Inputs:
#               r0 = x coodinate of centre of circle
#               r1 = y coodinate of centre of circle
#               r2 = width (in pixels)
#               r3 = radius (in pixels)
#
#       Outputs:
```

TL/EE/9664–3

```
#               no registers altered
#               circle drawn in ram
#
#Notes:
#       This routine uses two special case line drawing routines:
#               a horizontal case (called HLINE)
#               a vertical case (called VLINE)
#       A general purpose line drawing algorithm could be used, however
#       the new 32CG16 instructions are much faster.
#       If the line is to have a width of > 25 pixels, the BIGSET algorithm
#       must be added to the HLINE routine. No other changes are required.
#
circle: save    [r4,r5,r6,r7]   #save our working registers
        movd    r2,r7           #get current width
        lshd    $-1,r7          #divide by two
        movd    r7,hlfwdth      #and store it away
        movqd   0,r4            #x1 = 0
        movd    r3,r5           #y1 = radius
        movqd   3,r6            #p = 3 - (radius * 2)
        subd    r3,r6
        subd    r3,r6
        br      cirtest
        .align  4
cirlp:  bsr     setgrp          #set a group of points
        cmpqd   0,r6            #is P less than zero?
        blt     pge0            #no, it is not. skip
        addr    6(r6)[r4:d],r6  #p += 4 * x1 + 6
        addqd   1,r4            #x1 ++
cirtest:cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
        br      cirot1

        .align  4
pge0:   movd    r4,r7           #t = x1
        subd    r5,r7           #t = x1 - y1
        addr    10(r6)[r7:d],r6 #p += 4 * (x1 - y1) + 10
        addqd   -1,r5           #y1 --
        addqd   1,r4            #x1 ++
        cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
cirot1: bne     cirout          #if x1 != y1, get out
        bsr     setgrp          #else set last group
cirout: restore [r4,r5,r6,r7]   #restore working registers
        ret     0               #and return




#
#Setgrp sets eight points on a circle, given starting x and y, and the
#current xoffset and y offset.
#
#       Inputs:
#               r0 = centerpoint of circle (x coodinate)
```

TL/EE/9664–4

```
#               r1 = centerpoint of circle (y coodinate)
#               r2 = line width
#               r4 = x offset
#               r5 = y offset
#
#       Ouputs:
#               all registers preserved.
#
        .align  4
setgrp: movd    r6,tos          #get two temporary values
        movd    r7,tos
        movd    r0,r6           #save old x
        movd    r1,r7           #and y
        movd    r5,r1
        subd    r4,r1           #r1 = (y1 - x1)
        cmpd    r1,hlfwdth      #if the difference is less than
        ble     sgl:w           #half the width, fill in the edges
        movd    r7,r1           #restore y
        addd    r4,r0           #x += x1
        addd    r5,r1           #y += y1
        bsr     vline           #do a vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r4,r0           #x += x1
        subd    r5,r1           #y -= y1
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        addd    r5,r1           #y += y1
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        subd    r5,r1           #y -= y1
        bsr     vline

        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        addd    r4,r1           #y += x1
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        subd    r4,r1           #y -= x1
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
        addd    r4,r1           #y += x1
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
```

TL/EE/9664–5

```
        subd    r4,r1           #y -= x1
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        movd    tos,r7          #and unstack
        movd    tos,r6
        ret     0

sgl:    movd    r7,r1           #restore y
        addd    r4,r0           #x += x1
        addd    r5,r1           #y += y1
        bsr     hline           #do a hline
        bsr     vline           #and a vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r4,r0           #x += x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        addd    r5,r1           #y += y1
        bsr     hline
        bsr     vline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r4,r0           #x -= x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline

        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        addd    r5,r0           #x += y1
        subd    r4,r1           #y -= x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        subd    r5,r0           #x -= y1
        subd    r4,r1           #y -= x1
        bsr     vline
```

TL/EE/9664–6

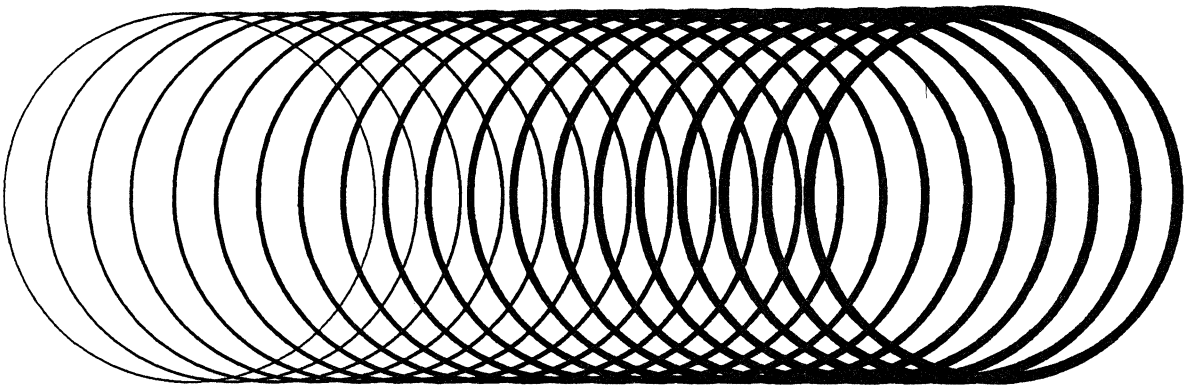
```
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        movd    tos,r7          #and unstack
        movd    tos,r6
        ret     0

#
#A vertical line drawing algorithm, making use of the SBITPS instruction
#
#       Inputs:
#               r0 = x coodinate of line
#               r1 = centerpoint of y coordinate of line
#               r2 = line length
#
#       Outputs:
#               no registers altered.
#               line drawn in memory.
#
        .align  4
vline:  save    [r0,r1,r2,r3]   #save working registers
        subd    hlfwdth,r1      #y -= half of width to centre vline
        addr    @(xwarp-1),r3   #r3 = xwarp -1
        indexd  r1,r3,r0        #bit off = y * (xwarp) + x
        addqd   1,r3            #move to correct warp value
        movd    _page,r0        #page address in r0
        SBITPS                  #set bit perpendicular string
        restore [r0,r1,r2,r3]   #restore registers
        ret     0

#
#A horizontal line drawing algorithm, using SBITS.
#
#       Inputs:
#               r0 = centerpoint of x coordinate
#               r1 = y coodinate of line
#               r2 = line length
#
        .align  4
hline:  save    [r0,r1,r3]      #save working registers
        subd    hlfwdth,r0      #x -= half of width to centre values
        indexd  r1,(xwarp - 1),r0 # bit off = (y * xwarp) + x
        movd    _page,r0        #page address in r0
        addr    stab,r3         #address of sbits table
        SBITS
        restore [r0,r1,r3]
        ret     0
```

TL/EE/9664–7

Figure 1 shows this algorithm 'at work'. 20 circles of radius 350 pixels, and widths of 1 to 20 pixels are shown. A full listing of this test program is shown in *Figure 2*.

4.0 TIMING

The execution speed of this algorithm is dependent on the radius of the circle, and the circle's width. The test program supplied executes in 2.92 seconds on a NS32016 at 10 MHz with no wait states. The execution time on the NS32CG16 at 15 MHz with no wait states is 1.54 seconds. By using macros for the VLINE and HLINE routines, instead of subroutine calls, the time can be further reduced to 1.39 seconds.



TL/EE/9664–8

FIGURE 1

```
        .data
        .set    xwarp,2544      #bits of xwarp to get to next scan
        .comm   _page,4
hlfwdth:.double 0
        .text
#
# Test is a C - callable function that creates Figure 1.
#
        .globl  _test
_test:  save    [r3,r4,r5,r6,r7]
        addr    @400,r0         #start at x=400
        addr    @400,r1         #         y=400
        movqd   1,r2            #width = 1
        addr    @350,r3         #radius = 350
        addr    @20,r7          #we want to do 200 circles
lp:     bsr     circle          #do a circle
        addr    80(r0),r0       #x += 80
        addqd   1,r2            #width += 1
        acbd    -1,r7,lp        #loop for all 200 circles
        restore [r3,r4,r5,r6,r7]
        ret     0               #and return

#
#Bresenham's circle algorithm, as expressed in "Computer Graphics" by
#Donald Hearn and M. Pauline Baker (1986, Prentice-Hall,
#ISBN 0-13-165382-2)
#
#       Inputs:
#               r0 = x coodinate of centre of circle
#               r1 = y coodinate of centre of circle
#               r2 = width (in pixels)
#               r3 = radius (in pixels)
#
#       Outputs:
#               no registers altered
#               circle drawn in ram
#
#Notes:
#       This routine uses two special case line drawing routines:
#               a horizontal case (called HLINE)
#               a vertical case (called VLINE)
#       A general purpose line drawing algorithm could be used, however
#       the new 32CG16 instructions are much faster.
#       If the line is to have a width of > 25 pixels, the BIGSET algorithm
#       must be added to the HLINE routine. No other changes are required.
#
circle: save    [r4,r5,r6,r7]   #save our working registers
        movd    r2,r7           #get current width
```

TL/EE/9664–9

FIGURE 2

```
        lshd    $-1,r7          #divide by two
        movd    r7,hlfwdth      #and store it away
        movqd   0,r4            #x1 = 0
        movd    r3,r5           #y1 = radius
        movqd   3,r6            #p = 3 - (radius * 2)
        subd    r3,r6
        subd    r3,r6
        br      cirtest
        .align  4
cirlp:  bsr     setgrp          #set a group of points
        cmpqd   0,r6            #is P less than zero?
        blt     pge0            #no, it is not. skip
        addr    6(r6)[r4:d],r6  #p += 4 * x1 + 6
        addqd   1,r4            #x1 ++
cirtest:cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
        br      cirout

        .align  4
pge0:   movd    r4,r7           #t = x1
        subd    r5,r7           #t = x1 - y1
        addr    10(r6)[r7:d],r6 #p += 4 * (x1 - y1) + 10
        addqd   -1,r5           #y1 --
        addqd   1,r4            #x1 ++
        cmpd    r4,r5           #is x1 <= y1 ?
        ble     cirlp           #it is. Loop
cirout: restore [r4,r5,r6,r7]   #restore working registers
        ret     0               #and return


#
#Setgrp sets eight points on a circle, given starting x and y, and the
#current xoffset and y offset.
#
#       Inputs:
#               r0 = centerpoint of circle (x coodinate)
#               r1 = centerpoint of circle (y coodinate)
#               r2 = line width
#               r4 = x offset
#               r5 = y offset
#
#       Ouputs:
#               all registers preserved.
#
        .align  4
setgrp: movd    r6,tos          #get two temporary values
        movd    r7,tos
        movd    r0,r6           #save old x
```

TL/EE/9664–10

FIGURE 2 (Continued)

```
movd    r1,r7           #and y
movd    r5,r1
subd    r4,r1           #r1 = (y1 - x1)
cmpd    r1,hlfwdth      #if the difference is less than
ble     sg1             #half the width, fill in the edges
movd    r7,r1           #restore y
addd    r4,r0           #x += x1
addd    r5,r1           #y += y1
bsr     vline           #do a vline
movd    r6,r0           #restore x and y
movd    r7,r1
addd    r4,r0           #x += x1
subd    r5,r1           #y -= y1
bsr     vline
movd    r6,r0           #restore x and y
movd    r7,r1
subd    r4,r0           #x -= x1
addd    r5,r1           #y += y1
bsr     vline
movd    r6,r0           #restore x and y
movd    r7,r1
subd    r4,r0           #x -= x1
subd    r5,r1           #y -= y1
bsr     vline

movd    r6,r0           #restore x and y
movd    r7,r1
addd    r5,r0           #x += y1
addd    r4,r1           #y += x1
bsr     hline
movd    r6,r0           #restore x and y
movd    r7,r1
addd    r5,r0           #x += y1
subd    r4,r1           #y -= x1
bsr     hline
movd    r6,r0           #restore x and y
movd    r7,r1
subd    r5,r0           #x -= y1
addd    r4,r1           #y += x1
bsr     hline
movd    r6,r0           #restore x and y
movd    r7,r1
subd    r5,r0           #x -= y1
subd    r4,r1           #y -= x1
bsr     hline
movd    r6,r0           #restore x and y
movd    r7,r1
movd    tos,r7          #and unstack
```

TL/EE/9664-11

FIGURE 2 (Continued)

```
        movd    tos,r6
        ret     ø

sg1:    movd    r7,r1           #restore y
        addd    r4,rø           #x += x1
        addd    r5,r1           #y += y1
        bsr     hline           #do a hline
        bsr     vline           #and a vline
        movd    r6,rø           #restore x and y
        movd    r7,r1
        addd    r4,rø           #x += x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline
        movd    r6,rø           #restore x and y
        movd    r7,r1
        subd    r4,rø           #x -= x1
        addd    r5,r1           #y += y1
        bsr     hline
        bsr     vline
        movd    r6,rø           #restore x and y
        movd    r7,r1
        subd    r4,rø           #x -= x1
        subd    r5,r1           #y -= y1
        bsr     hline
        bsr     vline

        movd    r6,rø           #restore x and y
        movd    r7,r1
        addd    r5,rø           #x += y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,rø           #restore x and y
        movd    r7,r1
        addd    r5,rø           #x += y1
        subd    r4,r1           #y -= x1
        bsr     vline
        bsr     hline
        movd    r6,rø           #restore x and y
        movd    r7,r1
        subd    r5,rø           #x -= y1
        addd    r4,r1           #y += x1
        bsr     vline
        bsr     hline
        movd    r6,rø           #restore x and y
        movd    r7,r1
        subd    r5,rø           #x -= y1
```

TL/EE/9664–12

FIGURE 2 (Continued)

```
        subd    r4,r1           #y -= x1
        bsr     vline
        bsr     hline
        movd    r6,r0           #restore x and y
        movd    r7,r1
        movd    tos,r7          #and unstack
        movd    tos,r6
        ret     0

#
#A vertical line drawing algorithm, making use of the SBITPS instruction.
#
#       Inputs:
#               r0 = x coodinate of line
#               r1 = centerpoint of y coordinate of line
#               r2 = line length
#
#       Outputs:
#               no registers altered.
#               line drawn in memory.
#
        .align  4
vline:  save    [r0,r1,r2,r3]   #save working registers
        subd    hlfwdth,r1      #y -= half of width to centre vline
        addr    @(xwarp-1),r3   #r3 = xwarp -1
        indexd  r1,r3,r0        #bit off = y * (xwarp) + x
        addqd   1,r3            #move to correct warp value
        movd    _page,r0        #page address in r0
#       SBITPS                  #set bit perpendicular string

# - Start of SBITPS emulation code
        .align  4
sblp:   sbitd   r1,0(r0)        #set required bit
        addd    r3,r1           #add the bit warp
        acbd    -1,r2,sblp      #loop for the rll
# - End of SBITPS emulation code
        restore [r0,r1,r2,r3]   #restore registers
        ret     0

#
#A horizontal line drawing algorithm, using SBITS.
#
#       Inputs:
#               r0 = centerpoint of x coordinate
#               r1 = y coodinate of line
#               r2 = line length
#
        .align  4
```

TL/EE/9664–13

FIGURE 2 (Continued)

```
hline:  save    [r0,r1,r3]      #save working registers
        subd    hlfwdth,r0      #x -= half of width to centre values
        indexd  r1,$(xwarp - 1),r0 # bit off = (y * xwarp) + x
        movd    _page,r0        #page address in r0
#       addr    stab,r3         #address of sbits table
#       SBITS

# - start of SBITS emulation code
        movqd   7,r3
        andd    r1,r3

        addd    r3,r3   #* 2
        addd    r3,r3   #* 4
        addd    r3,r3   #* 8
        addd    r3,r3   #* 16
        addd    r3,r3   #* 32
        addd    r2,r3
        ashd    $-3,r1
        ord     stab[r3:d],0(r0)[r1:b]
# - end of SBITS emulation code
        restore [r0,r1,r3]
        ret     0


        .data
stab:   .double h'00000000,h'00000001,h'00000003,h'00000007
        .double h'0000000f,h'0000001f,h'0000003f,h'0000007f
        .double h'000000ff,h'000001ff,h'000003ff,h'000007ff
        .double h'00000fff,h'00001fff,h'00003fff,h'00007fff
        .double h'0000ffff,h'0001ffff,h'0003ffff,h'0007ffff
        .double h'000fffff,h'001fffff,h'003fffff,h'007fffff
        .double h'00ffffff,h'01ffffff,h'03ffffff,h'07ffffff
        .double h'0fffffff,h'1fffffff,h'3fffffff,h'7fffffff
        .double h'00000000,h'00000002,h'00000006,h'0000000e
        .double h'0000001e,h'0000003e,h'0000007e,h'000000fe
        .double h'000001fe,h'000003fe,h'000007fe,h'00000ffe
        .double h'00001ffe,h'00003ffe,h'00007ffe,h'0000ffffe
        .double h'0001fffe,h'0003fffe,h'0007fffe,h'000ffffe
        .double h'001ffffe,h'003ffffe,h'007ffffe,h'00ffffffe
        .double h'01ffffffe,h'03ffffffe,h'07ffffffe,h'0ffffffe
        .double h'1ffffffe,h'3ffffffe,h'7ffffffe,h'fffffffe
        .double h'00000000,h'00000004,h'0000000c,h'0000001c
        .double h'0000003c,h'0000007c,h'000000fc,h'000001fc
        .double h'000003fc,h'000007fc,h'00000ffc,h'00001ffc
        .double h'00003ffc,h'00007ffc,h'0000ffffc,h'0001ffffc
        .double h'0003fffc,h'0007fffc,h'000ffffc,h'001ffffc
        .double h'003ffffc,h'007ffffc,h'00ffffffc,h'01ffffffc
        .double h'03ffffffc,h'07ffffffc,h'0ffffffc,h'1ffffffc
```

TL/EE/9664–14

FIGURE 2 (Continued)

```
.double h'3ffffffc,h'7ffffffc,h'ffffffc,h'ffffffc
.double h'00000000,h'00000008,h'00000018,h'00000038
.double h'00000078,h'000000f8,h'00000f8,h'00003f8
.double h'00007f8,h'0000ff8,h'0001ff8,h'0003ff8
.double h'0007ff8,h'000fff8,h'001fff8,h'003fff8
.double h'007ffff8,h'00ffff8,h'01ffff8,h'03ffff8
.double h'07fffff8,h'0fffff8,h'1fffff8,h'3fffff8
.double h'7fffff8,h'ffffff8,h'ffffff8,h'ffffff8
.double h'00000000,h'00000010,h'00000030,h'00000070
.double h'000000f0,h'00001f0,h'00003f0,h'00007f0
.double h'0000ff0,h'0001ff0,h'0003ff0,h'0007ff0
.double h'000fff0,h'001fff0,h'003fff0,h'007fff0
.double h'00ffff0,h'01ffff0,h'03ffff0,h'07ffff0
.double h'0fffff0,h'1fffff0,h'3fffff0,h'7fffff0
.double h'ffffff0,h'ffffff0,h'ffffff0,h'ffffff0
.double h'00000000,h'00000020,h'00000060,h'000000e0
.double h'000001e0,h'000003e0,h'000007e0,h'00000fe0
.double h'0001fe0,h'0003fe0,h'0007fe0,h'000ffe0
.double h'001ffe0,h'003ffe0,h'007ffe0,h'00ffffe0
.double h'01ffffe0,h'03ffffe0,h'07ffffe0,h'0ffffe0
.double h'1ffffe0,h'3ffffe0,h'7ffffe0,h'ffffffe0
.double h'ffffffe0,h'ffffffe0,h'ffffffe0,h'ffffffe0
.double h'00000000,h'00000040,h'000000c0,h'000001c0
.double h'000003c0,h'000007c0,h'00000fc0,h'00001fc0
.double h'0003fc0,h'0007fc0,h'000ffc0,h'001ffc0
.double h'003ffc0,h'007ffc0,h'00ffffc0,h'01ffffc0
.double h'03ffffc0,h'07ffffc0,h'0ffffffc0,h'1ffffffc0
.double h'3ffffffc0,h'7ffffffc0,h'ffffffc0,h'ffffffc0
.double h'ffffffc0,h'ffffffc0,h'ffffffc0,h'ffffffc0
.double h'00000000,h'00000080,h'00000180,h'00000380
.double h'00000780,h'00000f80,h'00001f80,h'00003f80
.double h'00007f80,h'0000ff80,h'0001ff80,h'0003ff80
.double h'0007ff80,h'000fff80,h'001fff80,h'003fff80
.double h'007fff80,h'00ffff80,h'01ffff80,h'03ffff80
.double h'07ffff80,h'0fffff80,h'1fffff80,h'3fffff80
.double h'7fffff80,h'ffffff80,h'ffffff80,h'ffffff80
.double h'ffffff80,h'ffffff80,h'ffffff80,h'ffffffff
```

TL/EE/9664–15

FIGURE 2 (Continued)

5.0 CONCLUSIONS

The NS32CG16 provides several instructions that increase the speed of imaging common graphic items such as circles, lines, and ellipses. The NS32CG16's high code density, and fast execution, make it ideal for intensive graphics processing.

This algorithm does, however, show an apparent 'thinning' on the 45° boundaries, when the width of the circle is greater than five pixels. An alternate algorithm will be presented in a future applications note. This algorithm is optimized for speed.

6.0 REFERENCES

1. Hearn, Donald and M. Pauline Baker, (1986). *Computer Graphics*, Englewood Cliffs, N.J., Prentice-Hall, 65–69.

2. Foley, James D. and Van Dam, Andries, (1982). *Fundamentals of Interactive Computer Graphics*, Reading, Massachusetts, Addison-Wesley, 441–446.

Introduction to Bresenham's Line Algorithm Using the SBIT Instruction; Series 32000® Graphics Note 5

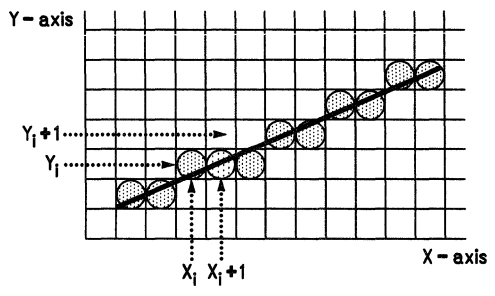
National Semiconductor
Application Note 524
Nancy Cossitt

1.0 INTRODUCTION

Even with today's achievements in graphics technology, the resolution of computer graphics systems will never reach that of the real world. A true real line can never be drawn on a laser printer or CRT screen. There is no method of accurately printing all of the points on the continuous line described by the equation $y = mx + b$. Similarly, circles, ellipses and other geometrical shapes cannot truly be implemented by their theoretical definitions because the graphics system itself is discrete, not real or continuous. For that reason, there has been a tremendous amount of research and development in the area of discrete or raster mathematics. Many algorithms have been developed which "map" real-world images into the discrete space of a raster device. Bresenham's line-drawing algorithm (and its derivatives) is one of the most commonly used algorithms today for describing a line on a raster device. The algorithm was first published in Bresenham's 1965 article entitled "Algorithm for Computer Control of a Digital Plotter". It is now widely used in graphics and electronic printing systems. This application note will describe the fundamental algorithm and show an implementation on National Semiconductor's Series 32000 microprocessor using the SBIT instruction, which is particularly well-suited for such applications. A timing diagram can be found in *Figure 8* at the end of the application note.

2.0 DESCRIPTION

Bresenham's line-drawing algorithm uses an iterative scheme. A pixel is plotted at the starting coordinate of the line, and each iteration of the algorithm increments the pixel one unit along the major, or x-axis. The pixel is incremented along the minor, or y-axis, only when a decision variable (based on the slope of the line) changes sign. A key feature of the algorithm is that it requires only integer data and simple arithmetic. This makes the algorithm very efficient and fast.



TL/EE/9665–1

FIGURE 1

The algorithm assumes the line has positive slope less than one, but a simple change of variables can modify the algorithm for any slope value. This will be detailed in section 2.2.

2.1 Bresenham's Algorithm for 0 < slope < 1

Figure 1 shows a line segment superimposed on a raster grid with horizontal axis X and vertical axis Y. Note that x_i and y_i are the integer abscissa and ordinate respectively of each pixel location on the grid.

Given (x_i, y_i) as the previously plotted pixel location for the line segment, the next pixel to be plotted is either $(x_i + 1, y_i)$ or $(x_i + 1, y_i + 1)$. Bresenham's algorithm determines which of these two pixel locations is nearer to the actual line by calculating the distance from each pixel to the line, and plotting that pixel with the smaller distance. Using the familiar equation of a straight line, $y = mx + b$, the y value corresponding to $x_i + 1$ is

$$y = m(x_i + 1) + b$$

The two distances are then calculated as:

$$d1 = y - y_i$$

$$d1 = m(x_i + 1) + b - y_i$$

$$d2 = (y_i + 1) - y$$

$$d2 = (y_i + 1) - m(x_i + 1) - b$$

and,

$$d1 - d2 = m(x_i + 1) + b - y_i - (y_i + 1) + m(x_i + 1) + b$$

$$d1 - d2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

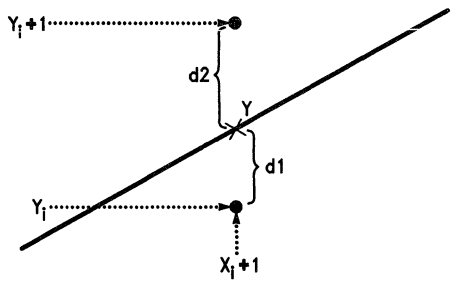
Multiplying this result by the constant dx, defined by the slope of the line $m = dy/dx$, the equation becomes:

$$dx(d1-d2) = 2dy(x_i) - 2dx(y_i) + c$$

where c is the constant $2dy + 2dxb - dx$. Of course, if $d2 > d1$, then $(d1-d2) < 0$, or conversely if $d1 > d2$, then $(d1-d2) > 0$. Therefore, a parameter p_i can be defined such that

$$p_i = dx(d1-d2)$$

$$p_i = 2dy(x_i) - 2dx(y_i) + c$$



TL/EE/9665–2

Distances d1 and d2 are compared.
The smaller distance marks next pixel to be plotted.

FIGURE 2

If $p_i > 0$, then d1 > d2 and y_{i+1} is chosen such that the next plotted pixel is $(x_i + 1, y_i)$. Otherwise, if $p_i < 0$, then d2 > d1 and $(x_i + 1, y_i + 1)$ is plotted. (See *Figure 2.*)

Similarly, for the next iteration, p_{i+1} can be calculated and compared with zero to determine the next pixel to plot. If $p_{i+1} < 0$, then the next plotted pixel is at $(x_{i+1} + 1, y_{i+1})$; if $p_{i+1} > 0$, then the next point is $(x_{i+1} + 1, y_{i+1} + 1)$. Note that in the equation for p_{i+1}, $x_{i+1} = x_i + 1$.

$$p_{i+1} = 2dy(x_i + 1) - 2dx(y_{i+1}) + c$$

Subtracting p_i from p_{i+1}, we get the recursive equation:

$$p_{i+1} = p_i + 2dy - 2dx(y_{i+1} - y_i)$$

Note that the constant c has conveniently dropped out of the formula. And, if $p_i < 0$ then $y_{i+1} = y_i$ in the above equation, so that:

$$p_{i+1} = p_i + 2dy$$

or, if $p_i > 0$ then $y_{i+1} = y_i + 1$, and

$$p_{i+1} = p_i + 2(dy-dx)$$

To further simplify the iterative algorithm, constants c1 and c2 can be initialized at the beginning of the program such that c1 = 2dy and c2 = 2(dy−dx). Thus, the actual meat of the algorithm is a loop of length dx, containing only a few integer additions and two compares *(Figure 3).*

2.2 For Slope < 0 and |Slope| > 1

The algorithm fails when the slope is negative or has absolute value greater than one ($|dy| > |dx|$). The reason for this is that the line will always be plotted with a positive slope if x_i and y_i are always incremented in the positive direction, and the line will always be "shorted" if $|dx| < |dy|$ since the algorithm executes once for every x coordinate (i.e., dx times). However, a closer look at the algorithm must be taken to reveal that a few simple changes of variables will take care of these special cases.

For negative slopes, the change is simple. Instead of incrementing the pixel along the positive direction (+1) for each iteration, the pixel is incremented in the negative direction. The relationship between the starting point and the finishing point of the line determines which axis is followed in the negative direction, and which is in the positive. *Figure 4* shows all the possible combinations for slopes and starting points, and their respective incremental directions along the X and Y axis.

Another change of variables can be performed on the incremental values to accommodate those lines with slopes greater than 1 or less than −1. The coordinate system containing the line is rotated 90 degrees so that the X-axis now becomes the Y-axis and vice versa. The algorithm is then performed on the rotated line according to the sign of its slope, as explained above. Whenever the current position is incremented along the X-axis in the rotated space, it is actually incremented along the Y-axis in the original coordinate space. Similarly, an increment along the Y-axis in the rotated space translates to an increment along the X-axis in the original space. *Figure 4a., g.* and *h.* illustrates this translation process for both positive and negative lines with various starting points.

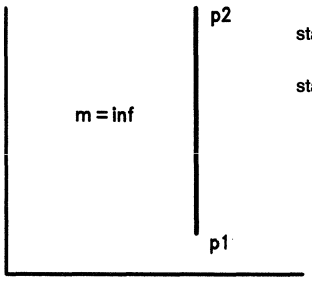
3.0 IMPLEMENTATION IN C

Bresenham's algorithm is easily implemented in most programming languages. However, C is commonly used for many application programs today, especially in the graphics area. The Appendix gives an implementation of Bresenham's algorithm in C. The C program was written and executed on a SYS32/20 system running UNIX on the NS32032 processor from National. A driver program, also written in C, passed to the function starting and ending points for each line to be drawn. *Figure 6* shows the output on an HP laser jet of 160 unique lines of various slopes on a bit map of 2,000 x 2,000 pixels. Each line starts and ends exactly 25 pixels from the previous line.

The program uses the variable *bit* to keep track of the current pixel position within the 2,000 x 2,000 bit map *(Figure 5).* When the Bresenham algorithm requires the current position to be incremented along the X-axis, the variable *bit* is incremented by either +1 or −1, depending on the sign of the slope. When the current position is incremented along the Y-axis (i.e., when p > 0) the variable *bit* is incremented by +warp or −warp, where *warp* is the vertical bit displacement of the bit map. The constant *last bit* is compared with *bit* during each iteration to determine if the line is complete. This ensures that the line starts and finishes according to the coordinates passed to the function by the driver program.

```
do while count < > dx
   if (p < 0) then p+ = c1
   else

                    p+ = c2
                    next_y = prev_y + y_inc
   next_x = prev_x + x_inc
   plot(next_x,next_y)
   count + = 1

/* PSEUDO CODE FOR BRESENHAM LOOP */
```

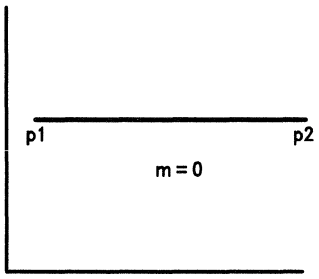
FIGURE 3



start p1: x_inc = y'_inc = 0
y_inc = x'_inc = +1
start p2: x_inc = y'_inc = 0
y_inc = x'_inc = − 1

TL/EE/9665-3

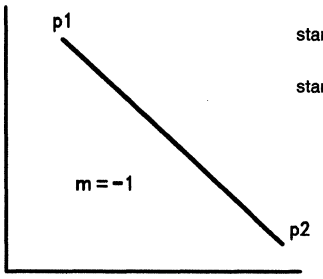
a.



start p1: x_inc = +1
y_inc = 0
start p2: x_inc = −1
y_inc = 0

TL/EE/9665-4

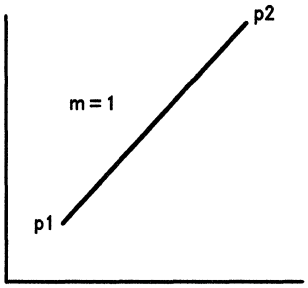
b.



start p1: x_inc = +1
y_inc = −1
start p2: x_inc = −1
y_inc = +1

TL/EE/9665-5

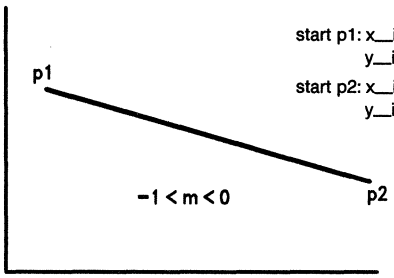
c.



start p1: x_inc = +1
y_inc = +1
start p2: x_inc = −1
y_inc = −1

TL/EE/9665-6

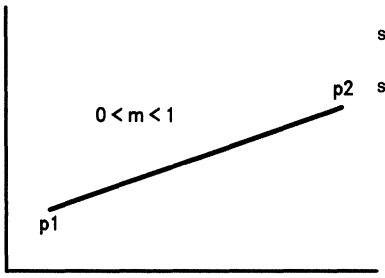
d.



start p1: x_inc = +1
y_inc = −1
start p2: x_inc = −1
y_inc = +1

TL/EE/9665-7

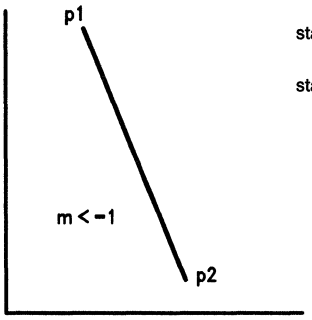
e.



start p1: x_inc = +1
y_inc = +1
start p2: x_inc = −1
y_inc = −1

TL/EE/9665-8

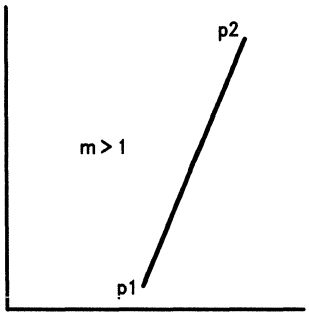
f.



start p1: x_inc = y'_inc = +1
y_inc = x'_inc = −1
start p2: x_inc = y'_inc = −1
y_inc = x'_inc = +1

TL/EE/9665-9

g.



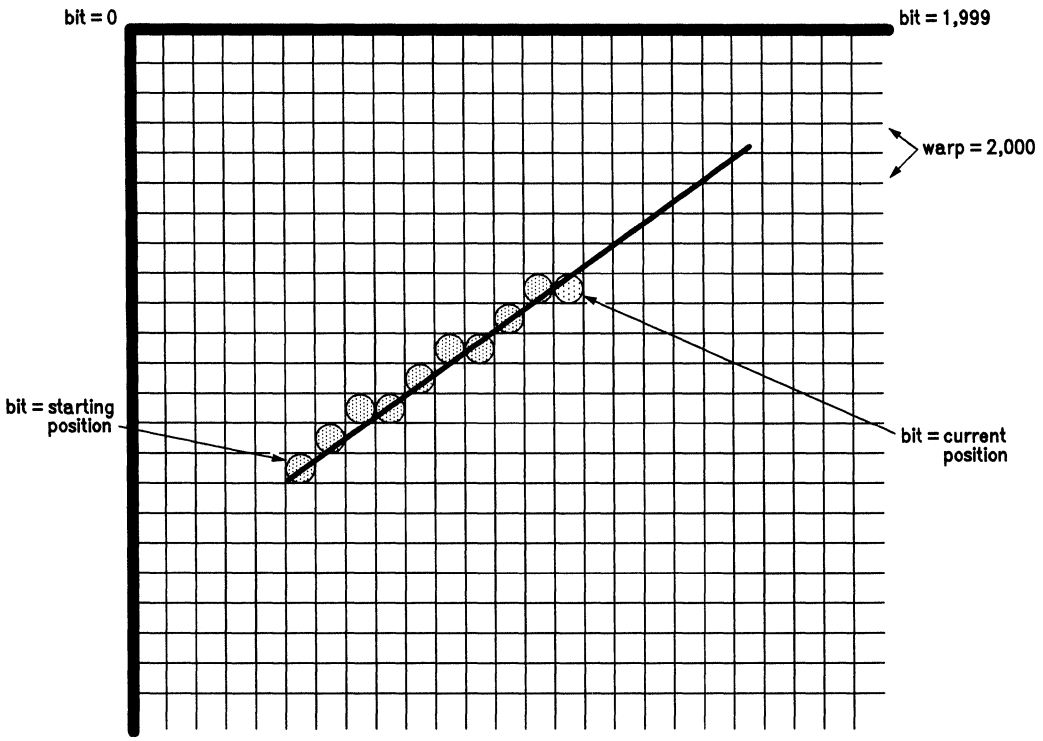
start p1: x_inc = y'_incl = −1
y_inc = x'_inc = +1
start p2: x_inc = y'_inc = +1
y_inc = x'_inc = −1

TL/EE/9665-10

h.

Note: a., g., and h. are rotated 90 degrees left and x', y' refer to the original axis.

FIGURE 4

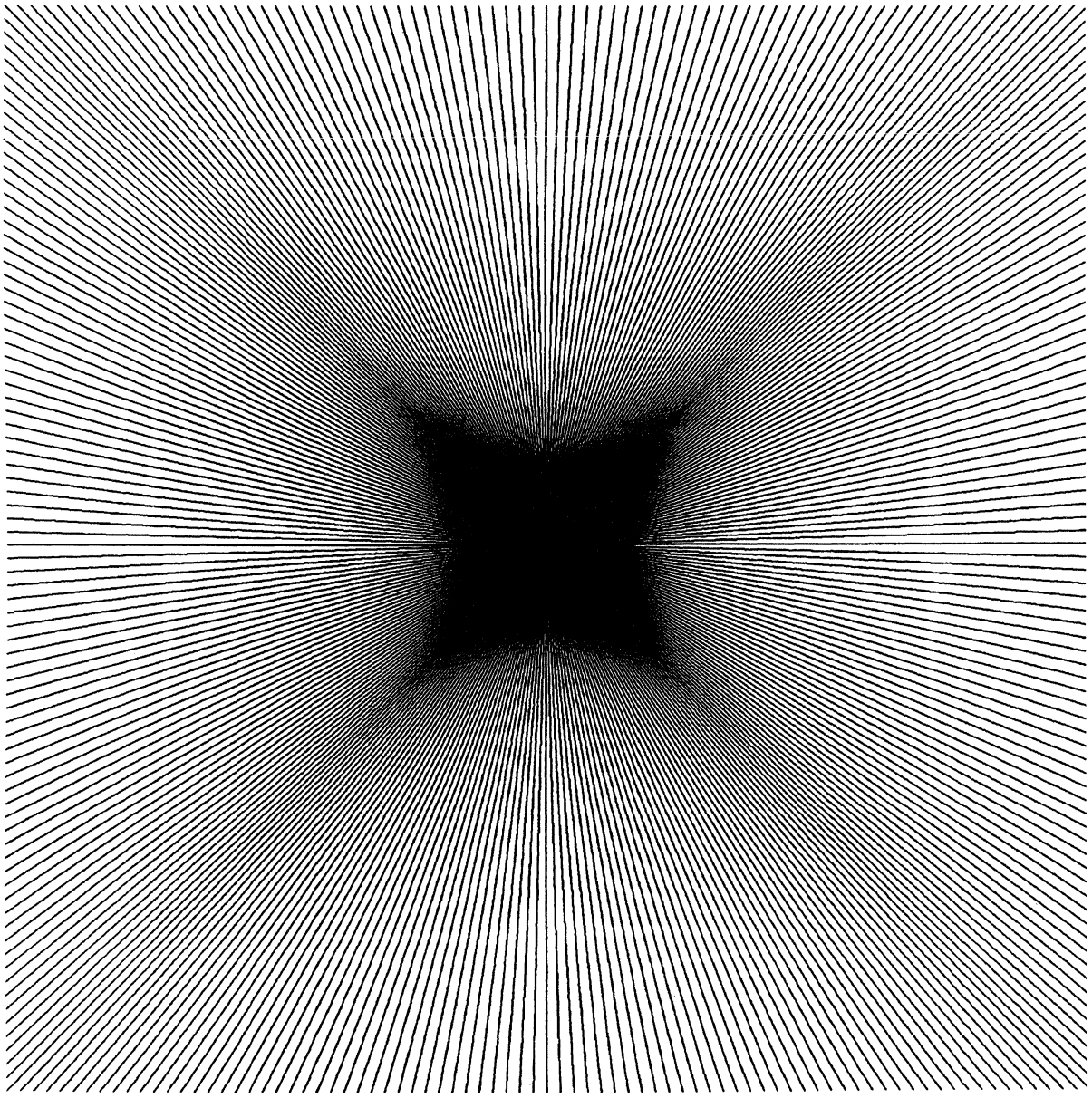


TL/EE/9665–11

Bit Map is 500 kbytes, 2k x 2k Bits
Base Address of Bit Map is 'Bit_Map'

FIGURE 5

Graphics Image (2000 x 2000 Pixels), 300 DPI



TL/EE/9665–12

FIGURE 6. Star-Burst Benchmark—This Star-Burst image was done on a 2k x 2k pixel bit map. Each line is 2k pixels in length and passes through the center of the image, bisecting the square. The lines are 25 pixel units apart, and are drawn using the LINE_DRAW.S routine. There are a total of 160 lines. The total time for drawing this Star-Burst is 2.9 sec on 10 MHz NS32C016.

4.0 IMPLEMENTATION IN SERIES 32000 ASSEMBLY: THE SBIT INSTRUCTION

National's Series 32000 family of processors is well-suited for the Bresenham's algorithm because of the SBIT instruction. *Figure 7* shows a portion of the assembly version of the Bresenham algorithm illustrating the use of the SBIT instruction. The first part of the loop, handles the algorithm for $p < 0$ and .CASE2 handles the algorithm for $p > 0$. The main loop is unrolled in this manner to minimize unnecessary branches (compare loop structure of *Figure 7* to *Figure 3*). The SBIT instruction is used to plot the current pixel in the line.

The SBIT instruction uses *bit_map* as a base address from which it calculates the bit position to be set by adding the offset *bit* contained in register r1. For example, if *bit*, or R1, contains 2,000*, then the instruction:

sbitd r1,@ *bit_map*

will set the bit at position 2,000, given that *bit_map* is the memory location starting at bit 0 of this grid. In actuality, if *base* is a memory address, then the bit position set is:

offset MOD 8

within the memory byte whose address is:

base + (*offset* DIV 8)

So, for the above example,

2,000 MOD 8 = 0

bit_map + 2,000 DIV 8 = *bit_map* + 250

Thus, bit 0 of byte (*bit_map* + 250) is set. This bit corresponds to the first bit of the second row in *Figure 5*.

*All numbers are in decimal.

```
# Main loop of Bresenham algorithm
.LOOP: #p < 0: move in x direction only
              cmpqd           $0,r4
              ble             .CASE2
              addd            r0,r4
              addd            r5,r1
              sbitd           r1,@_bit_map
              cmpd            r3,r1
              bne             .LOOP
              exit            [r3,r4,r5,r6,r7]
              ret             $0
              .align 4
.CASE2: #P > 0: move in x and y direction
              addd            r2,r4
              addd            r7,r1
              addd            r5,r1
              sbitd           r1,@_bit_map
              cmpd            r1,r3
              bne             .LOOP
              exit            [r3,r4,r5,r6,r7]
              ret             $0
```

FIGURE 7

Note: Instructions followed by the letter 'd' indicate "double word" operations.

The SBIT instruction greatly increases the speed of the algorithm. Notice the method of setting the pixel in the C program given in the Appendix:

bit_map[*bit*/8] | = *bit_pos*[(bit & 7)]

This line of code contains a costly division and several other operations that are eliminated with the SBIT instruction. The SBIT instruction helps optimize the performance of the program. Notice also that the algorithm can be implemented using only 7 registers. This improves the speed performance by avoiding time-consuming memory accesses.

5.0 CONCLUSION

An optimized Bresenham line-drawing algorithm has been presented using the SYS32/20 system. Both Series 32000 assembly and C versions have been included. *Figure 8* presents the various timing results of the algorithm. Most of the optimization efforts have been concentrated in the main loop of the program, so the reader may spot other ways to optimize, especially in the set-up section of the algorithm.

Several variations of the Bresenham algorithm have been developed. One particular variation from Bresenham himself relies on "run-length" segments of the line for speed optimization. The algorithm is based on the original Bresenham algorithm, but uses the fact that typically the decision variable p has one sign for several iterations, changing only once in-between these "run-length" segments to make one vertical step. Thus, most lines are composed of a series of horizontal "run-lengths" separated by a single vertical jump. (Consider the special cases where the slope of the line is exactly 1, the slope is 0 or the slope is infinity.) This algorithm will be explored in the NS32CG16 Graphics Note 5, AN-522, "Line Drawing with the NS32CG16", where it will be optimized using special instructions of the NS32CG16.

```
   Register and Memory
        Contents
r0 = c1 constant
r1 = bit current
     position
r2 = c2 constant
r3 = last_bit
r4 = p decision var
r5 = x_inc increment
r6 = unused register
r7 = y_inc increment
_bit_map = address of
first byte in bit map
```

Timing Performance
2k x 2k Bit Map
2k Pix/Vector 160 Lines per Star-Burst

Version	NS32000 Assembly with SBIT	
Parameter	**NS32C016-10**	**NS32C016-15**
Set-up Time Per Vector	45 μs	30 μs
Vectors/Sec	54	82
Pixels/Sec	109,776	164,771
Total Time Star-Burst Benchmark	2.9s	1.9s

FIGURE 8

Set-up time per line is measured from the start of LINE_DRAW.S only. The overhead of calling the LINE_DRAW routine, starting the timer and creating the endpoints of the vector are not included in this time. Set-up time does include all register set-up and branching for the Bresenham algorithm up to the entry point of the main loop.

Vectors/Second is determined by measuring the number of vectors per second the LINE_DRAW routine can draw, not including the overhead of the DRIVER.C and START.C routines, which start the timer and calculate the vector endpoints. All set-up of registers and branching for the Bresenham algorithm are included.

Pixels/Second is measured by dividing the Vectors/Second value by the number of pixels per line.

Total Time for the Star-Burst benchmark is measured from start of benchmark to end. It does include all overhead of START.C and DRIVER.C and all set-up for LINE_DRAW.S. This number can be used to approximate the number of pages per second for printing the whole Star-Burst image.

```
#       National Semiconductor Corporation.
#       CTP version 2.4     -- line_draw.s --

.file   "line_draw.s"
        .comm   _bit_map,49975Ø
        .globl  _line_draw
        .set    WARP,2ØØØ
        .align 4
_line_draw:                             # initialize
        enter   [r3,r4,r5,r6,r7],12
        movd    12(fp),r5               # r5=ys
        movd    8(fp),r6                # r6=xs
        movd    r5,r1                   # initialize starting 'bit'
        muld    $(WARP),r1              # bit=warp*ys+xs
        addd    r6,r1                   # r1=bit
        movd    2Ø(fp),r4               # r4=yf
        subd    r5,r4                   # r4=dy
        absd    r4,r3                   # r3=|dy|
        movd    16(fp),r2               # r2=xf
        subd    r6,r2                   # r2=dx
        absd    r2,r6                   # r6=|dx|
        cmpd    r3,r6                   # branch if slope<1
        ble     .LL1                    # must rotate axis for slope>1
        cmpqd   $(Ø),r4                 # if dy<Ø want x_inc<Ø
        bge     .LL2                    # else x_inc is pos
        addr    WARP,r5                 # x_inc=+/-warp because of rotate
        br      .LL3
        .align 4
.LL2:
        addr    -WARP,r5
.LL3:
        cmpqd   $(Ø),r2                 # if dx<Ø want y_inc<Ø
        bge     .LL4                    # else y_inc is pos
        movqd   $(1),r7                 # y_inc=+/-1 becaue of rotate
        br      .LL5
        .align 4
.LL4:
        movqd   $(-1),r7
.LL5:                                   # calculate c1,c2 and p
        movd    r6,rØ
        addd    rØ,rØ                   # rØ=c1=2*|dx| because of rotate
        subd    r3,r6                   # r6=|dx-dy|    r2=2*r6=c2
        addr    Ø[r6:w],r2              # this muls r6 by 2 and puts in r2
        movd    rØ,r4
        subd    r3,r4                   # r4=c2-|dy|=p in rotated space
        movd    2Ø(fp),r3               # calculate last_bit
        muld    $(WARP),r3
        addd    16(fp),r3               # r3=last_bit
        br      .LL6
        .align 4
.LL1:                                   # slope<1 use original axis
        cmpqd   $(Ø),r4                 # dy determines y_inc
        bge     .LL7
        addr    WARP,r7                 # dy>Ø then y_inc=+warp
        br      .LL8
        .align 4
.LL7:
        addr    -WARP,r7                # dy<Ø then y_inc=-warp
.LL8:
        cmpqd   $(Ø),r2
        bge     .LL9
        movqd   $(1),r5                 # dx>Ø then x_inc=+1
        br      .LL1Ø
        .align 4
.LL9:
        movqd   $(-1),r5                # dx<Ø then x_inc=-1
```

TL/EE/9665-13

```
.LL1Ø:                                  # calculate c1,c2,p
        addr    Ø[r3:w],rØ              # rØ=2*r3=c1
        movd    r3,r2
        subd    r6,r2
        addd    r2,r2                   # r2=2*|dy-dx|=c2
        movd    rØ,r4
        subd    r6,r4                   # p=2*dy-dx=r4
        movd    2Ø(fp),r3               # calculate last_bit=r3
        muld    $(WARP),r3
        addd    16(fp),r3
.LL6:                                   # main loop for algorithm
        cmpqd   $(Ø),r4                 # check sign of p
        ble     .LL11                   # branch if pos
        addd    rØ,r4                   # add c1 to p
        addd    r5,r1                   # inc bit by x_inc only
        sbitd   r1,@_bit_map            # plot bit
        cmpd    r3,r1                   # end only if bit=last_bit
        bne     .LL6
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
        .align 4
.LL11:                                  # p>Ø then inc in y dir
        addd    r2,r4                   # add c2 to p
        addd    r7,r1                   # add y_inc to bit
        addd    r5,r1                   # add x_inc to bit
        sbitd   r1,@_bit_map            # plot bit
        cmpd    r1,r3                   # end only when bit=last_bit
        bne     .LL6
        exit    [r3,r4,r5,r6,r7]
        ret     $(Ø)
```

TL/EE/9665-14

```
/* This program calculates points on a line using Bresenham's iterative */
/* method.  */

#include<stdio.h>
#define xbytes  25Ø             /* number of bytes along x-axis*/
#define warp    xbytes * 8      /* number of bits along x_axis*/
#define maxy    1999            /* number of lines in y_axis*/
unsigned char   bit_map[xbytes*maxy];   /* array contains bit map*/
static unsigned char    bit_pos[]={1,2,4,8,16,32,64,128};
                                /* look-up table for setting bit */

line_draw(xs,ys,xf,yf)          /* starting (s) and finishing (f) points */

int     xs,ys,xf,yf;

{
        int     dx,dy,x_inc,y_inc,      /* deltas and increments */
                bit,last_bit,           /* current and last bit positions */
                p,c1,c2;                /* decision variable p and constants */

        dx=xf-xs;
        dy=yf-ys;
        bit=(ys*warp)+xs;               /* initialize bit to first bit pos */
        last_bit=(yf*warp)+xf;          /* calculate last bit on line */

        if (abs(dy) > abs(dx))
        {                               /* abs(slope)>1 must rotate space */
                                        /* see Figure 5 a.,g.,and h. */
                if (dy>Ø)
                        x_inc=warp;     /* x_axis is now original y_axis */
                else
                        x_inc= -warp;
                if (dx>Ø)
                        y_inc=1;        /* y_axis is now original x_axis */
                else
                        y_inc= -1;
                c1=2*abs(dx);           /* calculate Bresenham's constants */
                c2=2*(abs(dx)-abs(dy));
                p=2*abs(dx)-abs(dy);    /* p is decision variable now rotated */
        }
        else {                          /* abs(slope)<1 use original axis */
                if (dy>Ø)
                        y_inc=warp;     /* y_inc is +/-warp number of bits */
                else
                        y_inc= -warp;
                if (dx>Ø)
                        x_inc=1;        /* move forward one bit */
                else
                        x_inc= -1;      /* or backward one bit */
                c1=2*abs(dy);           /* calculate constants and p */
                c2=2*(abs(dy)-abs(dx));
                p=2*abs(dy)-abs(dx);
        }

/* Bresenham's Algorithm */
                do      /* do once for each x increment, i.e. dx times */
                {
                        if (p<Ø)                /* no y movement if p<Ø */
                                p+=c1;
                        else {                  /* move in y dir if p>Ø */
                                p+=c2;
                                bit+=y_inc;
                        }
                        bit+=x_inc;             /* always increment x */

                        /* bit is set by calculating bit MOD 8, which is */
```

TL/EE/9665-15

```
                        /* same as bit & 7, then looking up appropriate  */
                        /* bit in table bit_pos.  This bit pos is then set */
                        /* in byte bit/8 */

                        bit_map[bit/8] |= bit_pos[(bit&7)];
                } while (bit!=last_bit);
}
```

TL/EE/9665-16

```
 /* Program driver.c feeds line vectors to LINE_DRAW.S forming Star-Burst.   */

#include <stdio.h>
#define xbytes  25ø
#define maxx    1999
#define maxy    1999

unsigned char   bit_map[xbytes*maxy];

main()

{
    int i,count;

/* generate Star-Burst image */

        for (count=1;count<=1øøø;test++){

                for (i=ø;i<=maxy;i+=25)
                        line_draw(ø,i,maxx,maxy-i);
                for (i=ø;i<=maxx;i+=25)
                        line_draw(i,maxy,maxx-i,ø);
        }

}
```

TL/EE/9665-17

```
/* Start timer and call main procedure of DRIVER.C to draw lines */

start() {
    long *timer = (long *) øx6øø;
    *timer = ø;             /* write a zero to timer location */
                main(ø,ø);      /* Show argc as zero, argv ->ø */
    return(*timer);     /* return, in rø, the current time */
}
```

TL/EE/9665-18

Block Move Optimization Techniques Series 32000® Graphics Note 2

National Semiconductor
Application Note 526
Dave Rand

1.0 INTRODUCTION

This application note discusses fast methods of moving data in printer applications using the National Semiconductor Series 32000. Typically this data is moved to or from the band of RAM representing a small portion (or slice) of the total image. The length of data is fixed. The controller design may require moving data every few milliseconds to image the page, until a total of 1 page has been moved. This may be (at 300 DPI, for example) (8.5 × 300) × (11 × 300), or 1,051,875 bytes. In current controller designs the width is often rounded to a word boundary (usually 320 bytes at 300 DPI). This technique uses 1,056,000 bytes, or 528,000 words.

2.0 DESCRIPTION

The move string instructions (MOVSi) in the 32000 are very powerful, however, when all that is needed is a string copy, they may be overkill. The string instructions include string translation, conditionals and byte/word/double sizes. If the application needs only to move a block of data from one location to another, and that data is a known size (or at least a multiple of a known size), using unrolled MOVD instructions is a faster way of moving the data from A to B on the NS32032 and NS32332.

3.0 IMPLEMENTATION

A code sample follows which makes use of a block size of 128 bytes. To move 256 bytes, for example, R0 should contain 2 on entry.

```
; Version 1.0  Sun Mar 29 12:57:20 1987
;
;A subroutine to move blocks of memory. Uses a granularity of
;128 bytes.
;
;       Inputs:
;               r0 = number of 128 byte blocks to move
;               r1 = source block address
;               r2 = destination block address
;
;Listing continues on following page
;
```

TL/EE/9696-1

```
;       Outputs:
;               r0 = 0
;               r1 = source block address + (128 * blocks)
;               r2 = destination block address + (128 * blocks)
;
;Notes:
;       This algorithm corresponds closely to the MOVSD instruction,
;       except that r0 contains the number of 128 byte blocks, not
;       4 byte double words. The output values are the same as if a
;       MOVSD instruction were used.
;
movmem: cmpqd   0,r0            ;if no blocks to move
        beq     mvexit          ;exit now.
        .align  4
mvlp1:  movd    0(r1),0(r2)     ;move one block of data
        movd    4(r1),4(r2)
        movd    8(r1),8(r2)
        movd    12(r1),12(r2)
        movd    16(r1),16(r2)
        movd    20(r1),20(r2)
        movd    24(r1),24(r2)
        movd    28(r1),28(r2)
        movd    32(r1),32(r2)
        movd    36(r1),36(r2)
        movd    40(r1),40(r2)
        movd    44(r1),44(r2)
        movd    48(r1),48(r2)
        movd    52(r1),52(r2)
        movd    56(r1),56(r2)
        movd    60(r1),60(r2)
        movd    64(r1),64(r2)
        movd    68(r1),68(r2)
        movd    72(r1),72(r2)
        movd    76(r1),76(r2)
        movd    80(r1),80(r2)
        movd    84(r1),84(r2)
        movd    88(r1),88(r2)
        movd    92(r1),92(r2)
        movd    96(r1),96(r2)
        movd    100(r1),100(r2)
        movd    104(r1),104(r2)
        movd    108(r1),108(r2)
        movd    112(r1),112(r2)
        movd    116(r1),116(r2)
        movd    120(r1),120(r2)
        movd    124(r1),124(r2)
        addr    128(r1),r1      ;quick way of adding 128
        addr    128(r2),r2
        acbd    -1,r0,mvlp1     ;loop for rest of blocks
mvexit: ret     $0
```

TL/EE/9696-2

4.0 TIMING

All timing assumes word aligned data (double word aligned for 32-bit bus). Unaligned data is permitted, but will reduce the speed.

On the 32532 (no wait states, @ 30 MHz, 32-bit bus), this code executes in 204 clocks, assuming burst mode access is available. To move 256 bytes, this routine would take 13.6 μs. The MOVSD instruction takes about 156 clocks to move a 128-byte block. The MOVSD instruction is the best choice, therefore, on the 32532.

On the 32332 (no wait states, @ 15 MHz, 32-bit bus), this code executes in 458 clocks per 128-byte block. Thus, to move 256 bytes, this algorithm takes 61.1 μs. The loop overhead (the ADDR and ACBD instructions) is about 10%. Doubling the block size (to 256 bytes) would reduce the loop overhead to 5%, and reducing the block size (to 64 bytes) would increase the loop overhead to 20%. In comparison, the 32332 MOVSD instruction takes about 721 clocks to move a 128-byte block.

On the 32032 (no wait states. @ 10 MHz, 32-bit bus), this code executes in 634 clocks per 128-byte block. Thus, to move 256 bytes, this algorithm takes 126.8 μs. The loop overhead (the ADDR and ACBD instructions) is about 5%. Doubling the block size (to 256 bytes) would reduce the loop overhead to 2.5%, and reducing the block size (to 64 bytes) would increase the loop overhead to 10%. In comparison, the 32032 MOVSD instruction takes about 690 clocks to move a 128-byte block.

On the 32016 (1 wait state. @ 10 MHz, 16-bit bus), this code executes in 1150 clocks per 128-byte block. Thus, to move 256 bytes, this algorithm takes 230.0 μs. The loop overhead on the 32016 is about 2.5%. In comparison, the 32016 MOVSD instruction would take about 1,074 clocks. Thus, the MOVSD instruction is faster, and makes better use of the available bus bandwidth of the NS32016.

5.0 CONCLUSIONS

The MOVSi instructions on the NS32016 provide a very fast memory block move capability, with variable size. On the NS32332 and NS32032, however, unrolled MOVD instructions are faster due to the larger bus bandwidth of the NS32332 and NS32032.

Clearing Memory with the 32000; Series 32000® Graphics Note 3

National Semiconductor
Application Note 527
Dave Rand

1.0 INTRODUCTION

In printer applications, large amounts of RAM may need to be initialized to a zero value. This application note describes a fast method.

2.0 DESCRIPTION

While several different methods of initializing memory to all zeros are available, here is one that works very well on the Series 32000. While the current version clears memory only in blocks of 128 bytes, other block sizes are possible by extending the algorithm.

3.0 IMPLEMENTATION

This routine is written to clear blocks of 128 bytes. This provides an optimal tradeoff between loop size (granularity) and loop overhead. This can be modified to use a different size. For example, to use a block size of 64 bytes, simply delete 16 of the MOVQD 0,TOS instructions from the listing. As well, since the value of r1 is now the number of 64 byte groups, one of the ADDD R2,R2 instructions (prior to the loading of the stack pointer) must be removed. Since the 32000 has two stacks, interrupts will be handled properly using this code. If only a fixed buffer size needs to be cleared, the code can be further unrolled to clear that area (i.e., increase the number of MOVQD 0,TOS instructions.)

```
; Version 1.1  Sun Mar 29 10:22:19 1987
;
;Subroutine to clear a block of memory. The granularity of this
;algorithm is 128 bytes, to reduce the looping overhead.
;
;       Inputs:
;               r0 = start of block
;               r1 = number of 128-byte groups to clear
;
;       Outputs:
;               All registers preserved.
;
;
;Listing continues on following page
;
```

TL/EE/9697-1

```
clram:  cmpqd   0,r1              ;any blocks to clear?
        beq     clexit:w          ;no, exit now.
        save    [r0,r1,r2]        ;save our working registers
        movd    r1,r2             ;here we set r0 = r0 + (r1 * 128) + 4
        addd    r2,r2             ;length *= 2
        addd    r2,r2             ;*4
        addd    r2,r2             ;*8
        addd    r2,r2             ;*16
        addr    4(r0)[r2:q],r0    ;get starting point + 4
        sprd    sp,r2             ;save current stack
        lprd    sp,r0             ;move to last double
        .align  4
cl2:    movqd   0,tos             ;clear a double
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        movqd   0,tos
        acbd    -1,r1,cl2
        lprd    sp,r2             ;restore stack pointer
        restore [r0,r1,r2]        ;restore our saved registers
clexit: ret     0
```

TL/EE/9697-2

FIGURE 1

```
clram:  cmpqd   0,r1            ;any blocks to clear?
        beq     clexit:w        ;no, exit now.
        .align  4
cl2:    movqd   0,00(r0)                ;clear a double
        movqd   0,04(r0)
        movqd   0,08(r0)
        movqd   0,12(r0)
        movqd   0,16(r0)
        movqd   0,20(r0)
        movqd   0,24(r0)
        movqd   0,28(r0)
        movqd   0,32(r0)
        movqd   0,36(r0)
        movqd   0,40(r0)
        movqd   0,44(r0)
        movqd   0,48(r0)
        movqd   0,52(r0)
        movqd   0,56(r0)
        movqd   0,60(r0)
        movqd   0,64(r0)
        movqd   0,68(r0)
        movqd   0,72(r0)
        movqd   0,76(r0)
        movqd   0,80(r0)
        movqd   0,84(r0)
        movqd   0,88(r0)
        movqd   0,92(r0)
        movqd   0,96(r0)
        movqd   0,100(r0)
        movqd   0,104(r0)
        movqd   0,108(r0)
        movqd   0,112(r0)
        movqd   0,116(r0)
        movqd   0,120(r0)
        movqd   0,124(r0)
        addd    $128,r0
        acbd    -1,r1,cl2
clexit: ret     0
```

TL/EE/9697-3

FIGURE 2

4.0 TIMING RESULTS

On the NS32016, NS32032 and NS32332, 4 clock cycles per write are required. To clear one page of 300 DPI 8½ x 11 (1,056,000 bytes), for example, requires 264,000 double words to be written. The optimal time for this, using 100% of the bus bandwidth on a 16 bit bus, would be 528,000 * 400 ns, or 211.2 ms, @ 10 MHz. All timing data assumes word aligned data (double word aligned for 32 bit bus). Unaligned data is permitted, but will reduce the speed somewhat.

On the NS32332 (no wait states. @15 MHz, 32 bit bus), this code clears the full page image in 178 ms.

On the NS32032 (no wait states. @10 MHz, 32 bit bus), this code clears the full page image in 324 ms.

On the NS32016 (1 wait state. @10 MHz, 16 bit bus), this code clears the full page image in 509 ms.

Doubling the block size (to 256 bytes) would increase the speed by 1%–2%, on the code sample.

On the NS32532, a better approach is to use the register indirect method of referencing memory, as is shown in *Figure 2*. With this approach, the page memory can be cleared in 19 ms, assuming a no wait state 30 MHz system, with a 32 bit bus. The optimal time, using 100% of the bus bandwidth of the NS32532 (2 clock bus cycle) would be 264,000 * 66.6 ns, or 17.6 ms.

Image Rotation Algorithm Series 32000® Graphics Note 4

National Semiconductor
Application Note 528
Dave Rand

1.0 INTRODUCTION

Fast image rotation of 90 and 270 degrees is important in printer applications, since both Portrait and Landscape orientation printing may be desired. With a fast image rotation algorithm, only the Portrait orientation fonts need to be stored. This minimizes ROM storage requirements.

This application note shows a fast image rotation algorithm that may be used to rotate an 8 pixel by 8 line image. Larger image sizes may be rotated by successive application of the rotation primitive.

2.0 DESCRIPTION

This Rotate Image algorithm (developed by the Electronic Imaging Group at National Semiconductor) does a very fast 8 by 8 (64 bit) rotation of font data. Note also that this algorithm does not exclusively deal with fonts, but any 64 bit image. Larger images can be rotated by breaking the image down into 8 x 8 segments, and using a 'source warp' constant to index into the source data.

The source data is pointed to by R0 on entry. A 'source warp' is contained in R1, and is added to R0 after each read of the source font. This allows the rotation of 16 by 16, 32 by 32 and larger fonts.

ROTIMG deals with the 8 by 8 destination character as 8 sequential bytes in two registers (R2 and R3), as follows:

Destination Font Matrix

Low Address

1
2
3
4
5
6
7
8

= R2	4	3	2	1
= R3	8	7	6	5

High Address

ROTIMG uses an external table (a pointer to the start of the table is located in register R4) to speed the rotation and to minimize the code. This table consists of 256 64 bit entries, or a total of 2,048 bytes. The table may be located code (PC) or data (SB) relative. The complete table is at the end of this document (see *Figure 1*). A few entries of the table are reproduced above.

Entry	Definition
0	0x00000000 00000000
1	0x00000000 00000001
2	0x00000000 00000100
3	0x00000000 00000101
...	
253	0x01010101 01010001
254	0x01010101 01010100
255	0x01010101 01010101

The bytes in the table are standard LSB to MSB format. Since there is no quad-byte assembler pseudo-op (other than LONG, which is floating point), we must reverse the 'double' declaration to get the correct byte ordering, as is shown below:

Entry	Definition
0	double 0,0
1	double 1,0
2	double 256,0
3	double 257,0
...	
253	double 16842753,16843009
254	double 0x01010100,0x01010101
255	double 0x01010101,0x01010101

Each byte within each eight byte table entry represents one bit of output data. By indexing into the table, and ORing the table's contents with R2 and R3, we set the destination byte if the corresponding source bit is set. In this manner, the character is rotated.

3.0 IMPLEMENTATION

What we are doing is setting the LS Bit of the destination byte if the source bit corresponding to that byte is set. We then shift the entire 64 bit destination left one bit, and repeat this process until we have set all eight bits, and processed all eight bytes of source information.

The source data for an 8 by 8 character ">" appears below:

Character Table for '>'

	Bit Number	Hex Value
	01234567	
Byte	0 0 1 0 0 0 0 0 0	02
	1 0 0 1 0 0 0 0 0	04
	2 0 0 0 1 0 0 0 0	08
	3 0 0 0 0 1 0 0 0	10
	4 0 0 0 0 1 0 0 0	10
	5 0 0 0 1 0 0 0 0	08
	6 0 0 1 0 0 0 0 0	04
	7 0 1 0 0 0 0 0 0	02

The ROTIMG algorithm, expressed in 32000 code, appears below:

```
#
#
#Rotate image emulation code
#
#    Inputs:
#         R0 = Source font address
#         R1 = Source font warp
#         R4 = Rotate table address
#
#    Outputs:
#         R2 = Destination font low 4 bytes (lsb->msb, 0 - 3)
#         R3 = Destination font high 4 bytes (lsb->msb, 4 - 7)
#
ROTIMG:  save [r0,r5,r6,r7]      #save registers we will use
         movqd    0,r2           #clear destination font
         movd     r2,r3          #clear high bits of dest.
         movd     r2,r5          #clear high bits of temp.
         addr     8,r6           #deal with 8 bytes of src.
rotlp:   movb     0(r0),r5       #get a byte of source
         addd     r1,r0          #add source warp
         addd     r2,r2          #shift destination left one bit
         addd     r3,r3          #top 32 bits too
         addrd    r4[r5:q],r7    #get pointer to table
         ord      0(r7),r2       #or in low bits
         ord      4(r7),r3       #or in high bits
         acbd     -1,r6,rotlp    #and back for more
         restore  [r0,r5,r6,r7]  #restore registers
         ret      $0             #and return
```

TL/EE/9698-1

Now, let's look at what happens to the data, given the example font of '>'.

Loop #	Source Font	R3	R2	
0	—	00000000	00000000	;0 destination
1	02 hex	00000000	00000100	;first bits in
2	04	00000000	00010200	;next bits in
3	08	00000000	01020400	;and so on
4	10	00000001	02040800	
5	10	00000003	04081000	
6	08	00000006	09102000	
7	04	0000000C	12214000	
8	02	00000018	24428100	;last iteration

Now, arranging this in the appropriate order gives us:

Destination Character Table for '>', 90 degree

	Bit Number	Hex Value
	01234567	
Byte	0 0 0 0 0 0 0 0 0	00
	1 1 0 0 0 0 0 0 1	81
	2 0 1 0 0 0 0 1 0	42
	3 0 0 1 0 0 1 0 0	24
	4 0 0 0 1 1 0 0 0	18
	5 0 0 0 0 0 0 0 0	00
	6 0 0 0 0 0 0 0 0	00
	7 0 0 0 0 0 0 0 0	00

Destination Character Table for '>', 270 degree

	Bit Number	Hex Value
	01234567	
Byte	0 0 0 0 0 0 0 0 0	00
	1 0 0 0 0 0 0 0 0	00
	2 0 0 0 0 0 0 0 0	00
	3 0 0 0 1 1 0 0 0	18
	4 0 0 1 0 0 1 0 0	24
	5 0 1 0 0 0 0 1 0	42
	6 1 0 0 0 0 0 0 1	81
	7 0 0 0 0 0 0 0 0	00

Note that by re-ordering the output data, we may rotate 90 or 270 degrees. This may also be accomplished by using a different table (see *Figure 2*).

4.0 TIMING

With unrolled 32000 code, the time for this algorithm is about 588 clocks on the 32016. Subtracting the font read time from this (about 113 clocks), the actual time for rotation is 475 clocks. On the 32332, the time is about 388 clocks. On the 32532, the unrolled loop time is 120–180 clocks, depending on burst mode availability. Repetition of the character data also affects the 32532, due to the data cache. See *Figure 3* for an unrolled code listing.

This table is used for the ROTIMG code. It is 256 entries of 64 bits each (8 bytes * 256 = 2048 bytes). There are two entries per line. This table is used for 90° rotation.

```
rottab1:  .double   0x00000000,0x00000000,0x00000001,0x00000000 ;0,1
          .double   0x00000100,0x00000000,0x00000101,0x00000000 ;2,3
          .double   0x00010000,0x00000000,0x00010001,0x00000000 ;4,5
          .double   0x00010100,0x00000000,0x00010101,0x00000000 ;6,7
          .double   0x01000000,0x00000000,0x01000001,0x00000000 ;...
          .double   0x01000100,0x00000000,0x01000101,0x00000000
          .double   0x01010000,0x00000000,0x01010001,0x00000000
          .double   0x01010100,0x00000000,0x01010101,0x00000000
          .double   0x00000000,0x00000001,0x00000001,0x00000001
          .double   0x00000100,0x00000001,0x00000101,0x00000001
          .double   0x00010000,0x00000001,0x00010001,0x00000001
          .double   0x00010100,0x00000001,0x00010101,0x00000001
          .double   0x01000000,0x00000001,0x01000001,0x00000001
          .double   0x01000100,0x00000001,0x01000101,0x00000001
          .double   0x01010000,0x00000001,0x01010001,0x00000001
          .double   0x01010100,0x00000001,0x01010101,0x00000001
          .double   0x00000000,0x00000100,0x00000001,0x00000100
          .double   0x00000100,0x00000100,0x00000101,0x00000100
          .double   0x00010000,0x00000100,0x00010001,0x00000100
          .double   0x00010100,0x00000100,0x00010101,0x00000100
          .double   0x01000000,0x00000100,0x01000001,0x00000100
          .double   0x01000100,0x00000100,0x01000101,0x00000100
          .double   0x01010000,0x00000100,0x01010001,0x00000100
          .double   0x01010100,0x00000100,0x01010101,0x00000100
          .double   0x00000000,0x00000101,0x00000001,0x00000101
          .double   0x00000100,0x00000101,0x00000101,0x00000101
          .double   0x00010000,0x00000101,0x00010001,0x00000101
          .double   0x00010100,0x00000101,0x00010101,0x00000101
          .double   0x01000000,0x00000101,0x01000001,0x00000101
          .double   0x01000100,0x00000101,0x01000101,0x00000101
          .double   0x01010000,0x00000101,0x01010001,0x00000101
          .double   0x01010100,0x00000101,0x01010101,0x00000101
          .double   0x00000000,0x00010000,0x00000001,0x00010000
          .double   0x00000100,0x00010000,0x00000101,0x00010000
          .double   0x00010000,0x00010000,0x00010001,0x00010000
          .double   0x00010100,0x00010000,0x00010101,0x00010000
          .double   0x01000000,0x00010000,0x01000001,0x00010000
          .double   0x01000100,0x00010000,0x01000101,0x00010000
          .double   0x01010000,0x00010000,0x01010001,0x00010000
          .double   0x01010100,0x00010000,0x01010101,0x00010000
          .double   0x00000000,0x00010001,0x00000001,0x00010001
          .double   0x00000100,0x00010001,0x00000101,0x00010001
          .double   0x00010000,0x00010001,0x00010001,0x00010001
          .double   0x00010100,0x00010001,0x00010101,0x00010001
          .double   0x01000000,0x00010001,0x01000001,0x00010001
          .double   0x01000100,0x00010001,0x01000101,0x00010001
          .double   0x01010000,0x00010001,0x01010001,0x00010001
          .double   0x01010100,0x00010001,0x01010101,0x00010001
          .double   0x00000000,0x00010100,0x00000001,0x00010100
```

TL/EE/9698–2

FIGURE 1

```
.double  0x00000100,0x00010100,0x00000101,0x00010100
.double  0x00010000,0x00010100,0x00010001,0x00010100
.double  0x00010100,0x00010100,0x00010101,0x00010100
.double  0x01000000,0x00010100,0x01000001,0x00010100
.double  0x01000100,0x00010100,0x01000101,0x00010100
.double  0x01010000,0x00010100,0x01010001,0x00010100
.double  0x01010100,0x00010100,0x01010101,0x00010100
.double  0x00000000,0x00010101,0x00000001,0x00010101
.double  0x00000100,0x00010101,0x00000101,0x00010101
.double  0x00010000,0x00010101,0x00010001,0x00010101
.double  0x00010100,0x00010101,0x00010101,0x00010101
.double  0x01000000,0x00010101,0x01000001,0x00010101
.double  0x01000100,0x00010101,0x01000101,0x00010101
.double  0x01010000,0x00010101,0x01010001,0x00010101
.double  0x01010100,0x00010101,0x01010101,0x00010101
.double  0x00000000,0x01000000,0x00000001,0x01000000
.double  0x00000100,0x01000000,0x00000101,0x01000000
.double  0x00010000,0x01000000,0x00010001,0x01000000
.double  0x00010100,0x01000000,0x00010101,0x01000000
.double  0x01000000,0x01000000,0x01000001,0x01000000
.double  0x01000100,0x01000000,0x01000101,0x01000000
.double  0x01010000,0x01000000,0x01010001,0x01000000
.double  0x01010100,0x01000000,0x01010101,0x01000000
.double  0x00000000,0x01000001,0x00000001,0x01000001
.double  0x00000100,0x01000001,0x00000101,0x01000001
.double  0x00010000,0x01000001,0x00010001,0x01000001
.double  0x00010100,0x01000001,0x00010101,0x01000001
.double  0x01000000,0x01000001,0x01000001,0x01000001
.double  0x01000100,0x01000001,0x01000101,0x01000001
.double  0x01010000,0x01000001,0x01010001,0x01000001
.double  0x01010100,0x01000001,0x01010101,0x01000001
.double  0x00000000,0x01000100,0x00000001,0x01000100
.double  0x00000100,0x01000100,0x00000101,0x01000100
.double  0x00010000,0x01000100,0x00010001,0x01000100
.double  0x00010100,0x01000100,0x00010101,0x01000100
.double  0x01000000,0x01000100,0x01000001,0x01000100
.double  0x01000100,0x01000100,0x01000101,0x01000100
.double  0x01010000,0x01000100,0x01010001,0x01000100
.double  0x01010100,0x01000100,0x01010101,0x01000100
.double  0x00000000,0x01000101,0x00000001,0x01000101
.double  0x00000100,0x01000101,0x00000101,0x01000101
.double  0x00010000,0x01000101,0x00010001,0x01000101
.double  0x00010100,0x01000101,0x00010101,0x01000101
.double  0x01000000,0x01000101,0x01000001,0x01000101
.double  0x01000100,0x01000101,0x01000101,0x01000101
.double  0x01010000,0x01000101,0x01010001,0x01000101
.double  0x01010100,0x01000101,0x01010101,0x01000101
.double  0x00000000,0x01010000,0x00000001,0x01010000
.double  0x00000100,0x01010000,0x00000101,0x01010000
.double  0x00010000,0x01010000,0x00010001,0x01010000
.double  0x00010100,0x01010000,0x00010101,0x01010000
.double  0x01000000,0x01010000,0x01000001,0x01010000
.double  0x01000100,0x01010000,0x01000101,0x01010000
.double  0x01010000,0x01010000,0x01010001,0x01010000
.double  0x01010100,0x01010000,0x01010101,0x01010000
```

TL/EE/9698–3

FIGURE 1 (Continued)

```
.double  0x00000000,0x01010001,0x00000001,0x01010001
.double  0x00000100,0x01010001,0x00000101,0x01010001
.double  0x00010000,0x01010001,0x00010001,0x01010001
.double  0x00010100,0x01010001,0x00010101,0x01010001
.double  0x01000000,0x01010001,0x01000001,0x01010001
.double  0x01000100,0x01010001,0x01000101,0x01010001
.double  0x01010000,0x01010001,0x01010001,0x01010001
.double  0x01010100,0x01010001,0x01010101,0x01010001
.double  0x00000000,0x01010100,0x00000001,0x01010100
.double  0x00000100,0x01010100,0x00000101,0x01010100
.double  0x00010000,0x01010100,0x00010001,0x01010100
.double  0x00010100,0x01010100,0x00010101,0x01010100
.double  0x01000000,0x01010100,0x01000001,0x01010100
.double  0x01000100,0x01010100,0x01000101,0x01010100
.double  0x01010000,0x01010100,0x01010001,0x01010100
.double  0x01010100,0x01010100,0x01010101,0x01010100
.double  0x00000000,0x01010101,0x00000001,0x01010101
.double  0x00000100,0x01010101,0x00000101,0x01010101
.double  0x00010000,0x01010101,0x00010001,0x01010101
.double  0x00010100,0x01010101,0x00010101,0x01010101
.double  0x01000000,0x01010101,0x01000001,0x01010101
.double  0x01000100,0x01010101,0x01000101,0x01010101 ;250,251
.double  0x01010000,0x01010101,0x01010001,0x01010101 ;252,253
.double  0x01010100,0x01010101,0x01010101,0x01010101 ;254,255
```

TL/EE/9698–4

FIGURE 1 (Continued)

This table is used for the ROTIMG code. It is 256 entries of 64 bits each (8 bytes * 256 = 2048 bytes). There are two entries per line. This gives a 270° rotation.

```
rottab2:  .double  0x00000000,0x00000000,0x00000000,0x01000000
          .double  0x00000000,0x00010000,0x00000000,0x01010000
          .double  0x00000000,0x00000100,0x00000000,0x01000100
          .double  0x00000000,0x00010100,0x00000000,0x01010100
          .double  0x00000000,0x00000001,0x00000000,0x01000001
          .double  0x00000000,0x00010001,0x00000000,0x01010001
          .double  0x00000000,0x00000101,0x00000000,0x01000101
          .double  0x00000000,0x00010101,0x00000000,0x01010101
          .double  0x01000000,0x00000000,0x01000000,0x01000000
          .double  0x01000000,0x00010000,0x01000000,0x01010000
          .double  0x01000000,0x00000100,0x01000000,0x01000100
          .double  0x01000000,0x00010100,0x01000000,0x01010100
          .double  0x01000000,0x00000001,0x01000000,0x01000001
          .double  0x01000000,0x00010001,0x01000000,0x01010001
          .double  0x01000000,0x00000101,0x01000000,0x01000101
          .double  0x01000000,0x00010101,0x01000000,0x01010101
          .double  0x00010000,0x00000000,0x00010000,0x01000000
          .double  0x00010000,0x00010000,0x00010000,0x01010000
          .double  0x00010000,0x00000100,0x00010000,0x01000100
          .double  0x00010000,0x00010100,0x00010000,0x01010100
          .double  0x00010000,0x00000001,0x00010000,0x01000001
          .double  0x00010000,0x00010001,0x00010000,0x01010001
          .double  0x00010000,0x00000101,0x00010000,0x01000101
          .double  0x00010000,0x00010101,0x00010000,0x01010101
```

TL/EE/9698–5

FIGURE 2

```
.double  0x01010000,0x00000000,0x01010000,0x01000000
.double  0x01010000,0x00010000,0x01010000,0x01010000
.double  0x01010000,0x00000100,0x01010000,0x01000100
.double  0x01010000,0x00010100,0x01010000,0x01010100
.double  0x01010000,0x00000001,0x01010000,0x01000001
.double  0x01010000,0x00010001,0x01010000,0x01010001
.double  0x01010000,0x00000101,0x01010000,0x01000101
.double  0x01010000,0x00010101,0x01010000,0x01010101
.double  0x00000100,0x00000000,0x00000100,0x01000000
.double  0x00000100,0x00010000,0x00000100,0x01010000
.double  0x00000100,0x00000100,0x00000100,0x01000100
.double  0x00000100,0x00010100,0x00000100,0x01010100
.double  0x00000100,0x00000001,0x00000100,0x01000001
.double  0x00000100,0x00010001,0x00000100,0x01010001
.double  0x00000100,0x00000101,0x00000100,0x01000101
.double  0x00000100,0x00010101,0x00000100,0x01010101
.double  0x01000100,0x00000000,0x01000100,0x01000000
.double  0x01000100,0x00010000,0x01000100,0x01010000
.double  0x01000100,0x00000100,0x01000100,0x01000100
.double  0x01000100,0x00010100,0x01000100,0x01010100
.double  0x01000100,0x00000001,0x01000100,0x01000001
.double  0x01000100,0x00010001,0x01000100,0x01010001
.double  0x01000100,0x00000101,0x01000100,0x01000101
.double  0x01000100,0x00010101,0x01000100,0x01010101
.double  0x00010100,0x00000000,0x00010100,0x01000000
.double  0x00010100,0x00010000,0x00010100,0x01010000
.double  0x00010100,0x00000100,0x00010100,0x01000100
.double  0x00010100,0x00010100,0x00010100,0x01010100
.double  0x00010100,0x00000001,0x00010100,0x01000001
.double  0x00010100,0x00010001,0x00010100,0x01010001
.double  0x00010100,0x00000101,0x00010100,0x01000101
.double  0x00010100,0x00010101,0x00010100,0x01010101
.double  0x01010100,0x00000000,0x01010100,0x01000000
.double  0x01010100,0x00010000,0x01010100,0x01010000
.double  0x01010100,0x00000100,0x01010100,0x01000100
.double  0x01010100,0x00010100,0x01010100,0x01010100
.double  0x01010100,0x00000001,0x01010100,0x01000001
.double  0x01010100,0x00010001,0x01010100,0x01010001
.double  0x01010100,0x00000101,0x01010100,0x01000101
.double  0x01010100,0x00010101,0x01010100,0x01010101
.double  0x00000001,0x00000000,0x00000001,0x01000000
.double  0x00000001,0x00010000,0x00000001,0x01010000
.double  0x00000001,0x00000100,0x00000001,0x01000100
.double  0x00000001,0x00010100,0x00000001,0x01010100
.double  0x00000001,0x00000001,0x00000001,0x01000001
.double  0x00000001,0x00010001,0x00000001,0x01010001
.double  0x00000001,0x00000101,0x00000001,0x01000101
.double  0x00000001,0x00010101,0x00000001,0x01010101
.double  0x01000001,0x00000000,0x01000001,0x01000000
.double  0x01000001,0x00010000,0x01000001,0x01010000
.double  0x01000001,0x00000100,0x01000001,0x01000100
.double  0x01000001,0x00010100,0x01000001,0x01010100
.double  0x01000001,0x00000001,0x01000001,0x01000001
.double  0x01000001,0x00010001,0x01000001,0x01010001
.double  0x01000001,0x00000101,0x01000001,0x01000101
```

TL/EE/9698–6

FIGURE 2 (Continued)

```
.double  0x01000001,0x00010101,0x01000001,0x01010101
.double  0x00010001,0x00000000,0x00010001,0x01000000
.double  0x00010001,0x00010000,0x00010001,0x01010000
.double  0x00010001,0x00000100,0x00010001,0x01000100
.double  0x00010001,0x00010100,0x00010001,0x01010100
.double  0x00010001,0x00000001,0x00010001,0x01000001
.double  0x00010001,0x00010001,0x00010001,0x01010001
.double  0x00010001,0x00000101,0x00010001,0x01000101
.double  0x00010001,0x00010101,0x00010001,0x01010101
.double  0x01010001,0x00000000,0x01010001,0x01000000
.double  0x01010001,0x00010000,0x01010001,0x01010000
.double  0x01010001,0x00000100,0x01010001,0x01000100
.double  0x01010001,0x00010100,0x01010001,0x01010100
.double  0x01010001,0x00000001,0x01010001,0x01000001
.double  0x01010001,0x00010001,0x01010001,0x01010001
.double  0x01010001,0x00000101,0x01010001,0x01000101
.double  0x01010001,0x00010101,0x01010001,0x01010101
.double  0x00000101,0x00000000,0x00000101,0x01000000
.double  0x00000101,0x00010000,0x00000101,0x01010000
.double  0x00000101,0x00000100,0x00000101,0x01000100
.double  0x00000101,0x00010100,0x00000101,0x01010100
.double  0x00000101,0x00000001,0x00000101,0x01000001
.double  0x00000101,0x00010001,0x00000101,0x01010001
.double  0x00000101,0x00000101,0x00000101,0x01000101
.double  0x00000101,0x00010101,0x00000101,0x01010101
.double  0x01000101,0x00000000,0x01000101,0x01000000
.double  0x01000101,0x00010000,0x01000101,0x01010000
.double  0x01000101,0x00000100,0x01000101,0x01000100
.double  0x01000101,0x00010100,0x01000101,0x01010100
.double  0x01000101,0x00000001,0x01000101,0x01000001
.double  0x01000101,0x00010001,0x01000101,0x01010001
.double  0x01000101,0x00000101,0x01000101,0x01000101
.double  0x01000101,0x00010101,0x01000101,0x01010101
.double  0x00010101,0x00000000,0x00010101,0x01000000
.double  0x00010101,0x00010000,0x00010101,0x01010000
.double  0x00010101,0x00000100,0x00010101,0x01000100
.double  0x00010101,0x00010100,0x00010101,0x01010100
.double  0x00010101,0x00000001,0x00010101,0x01000001
.double  0x00010101,0x00010001,0x00010101,0x01010001
.double  0x00010101,0x00000101,0x00010101,0x01000101
.double  0x00010101,0x00010101,0x00010101,0x01010101
.double  0x01010101,0x00000000,0x01010101,0x01000000
.double  0x01010101,0x00010000,0x01010101,0x01010000
.double  0x01010101,0x00000100,0x01010101,0x01000100
.double  0x01010101,0x00010100,0x01010101,0x01010100
.double  0x01010101,0x00000001,0x01010101,0x01000001
.double  0x01010101,0x00010001,0x01010101,0x01010001
.double  0x01010101,0x00000101,0x01010101,0x01000101
.double  0x01010101,0x00010101,0x01010101,0x01010101
```

TL/EE/9698–7

FIGURE 2 (Continued)

The following is an unrolled version of the rotate image algorithm. For the NS32532, the address computation, currently done with a separate addr instruction, may be done with the ORD instruction. This makes the execution time slightly faster.

```
#
#
#Rotate image emulation code
#
#     Inputs:
#          R0 = Source font address
#          R1 = Source font warp
#          R4 = Rotate table address
#
#     Outputs:
#          R2 = Destination font low 4 bytes (lsb->msb, 0 - 3)
#          R3 = Destination font high 4 bytes (lsb->msb, 4 - 7)
#
ROTIMG:
          movqd    0,r2           #clear destination font
          movd     r2,r3          #clear high bits of dest.
          movd     r2,r5          #clear high bits of temp.
          movb     0(r0),r5       #get a byte of source
          addd     r1,r0          #add source warp
          addd     r2,r2          #shift destination left one bit
          addd     r3,r3          #top 32 bits too
          addr     r4[r5:q],r6    #get pointer to table
          ord      0(r6),r2       #or in low bits
          ord      4(r6),r3       #or in high bits
          movb     0(r0),r5       #get a byte of source
          addd     r1,r0          #add source warp
          addd     r2,r2          #shift destination left one bit
          addd     r3,r3          #top 32 bits too
          addr     r4[r5:q],r6    #get pointer to table
          ord      0(r6),r2       #or in low bits
          ord      4(r6),r3       #or in high bits
          movb     0(r0),r5       #get a byte of source
          addd     r1,r0          #add source warp
          addd     r2,r2          #shift destination left one bit
          addd     r3,r3          #top 32 bits too
          addr     r4[r5:q],r6    #get pointer to table
          ord      0(r6),r2       #or in low bits
          ord      4(r6),r3       #or in high bits
          movb     0(r0),r5       #get a byte of source
          addd     r1,r0          #add source warp
          addd     r2,r2          #shift destination left one bit
          addd     r3,r3          #top 32 bits too
          addr     r4[r5:q],r6    #get pointer to table
          ord      0(r6),r2       #or in low bits
          ord      4(r6),r3       #or in high bits
          movb     0(r0),r5       #get a byte of source
          addd     r1,r0          #add source warp
```

TL/EE/9698-8

FIGURE 3

```
addd    r2,r2           #shift destination left one bit
addd    r3,r3           #top 32 bits too
addr    r4[r5:q],r6     #get pointer to table
ord     0(r6),r2        #or in low bits
ord     4(r6),r3        #or in high bits
movb    0(r0),r5        #get a byte of source
addd    r1,r0           #add source warp
addd    r2,r2           #shift destination left one bit
addd    r3,r3           #top 32 bits too
addr    r4[r5:q],r6     #get pointer to table
ord     0(r6),r2        #or in low bits
ord     4(r6),r3        #or in high bits
movb    0(r0),r5        #get a byte of source
addd    r1,r0           #add source warp
addd    r2,r2           #shift destination left one bit
addd    r3,r3           #top 32 bits too
addr    r4[r5:q],r6     #get pointer to table
ord     0(r6),r2        #or in low bits
ord     4(r6),r3        #or in high bits
movb    0(r0),r5        #get a byte of source
addd    r1,r0           #add source warp
addd    r2,r2           #shift destination left one bit
addd    r3,r3           #top 32 bits too
addr    r4[r5:q],r6     #get pointer to table
ord     0(r6),r2        #or in low bits
ord     4(r6),r3        #or in high bits
ret     $0              #and return
```

TL/EE/9698-9

FIGURE 3 (Continued)

80x86 to Series 32000® Translation; Series 32000 Graphics Note 6

National Semiconductor
Application Note 529
Dave Rand

1.0 INTRODUCTION

This application note discusses the conversion of Intel 8088, 8086, 80188 and 80186 (referred to here as 80x86) source assembly language to Series 32000 source code. As this is not intended to be a tutorial on Series 32000 assembly language, please see the Series 32000 Programmers Reference Manual for more information on instructions and addressing modes.

2.0 DESCRIPTION

The 80x86 model has 6 general purpose registers (AX, BX, CX, DX, SI, DI), each 16 bits wide. 4 of these registers can be further addressed as 8-bit registers (AL, AH, BL, BH, CL, CH, DL, DH). Series 32000 has 8 general purpose registers (R0–R7), each 32 bits wide. Each Series 32000 register may be accessed as an 8-, 16- or 32-bit register. Two special purpose registers on the 80x86, SP and BP, are 16-bit stack and base pointers. These are represented in Series 32000 with the SP and FP registers, each 32-bit.

The 80x86 model is capable of addressing up to 1 Megabyte of memory. Since the 16-bit register pointers are only capable of addressing 64 kbytes, 4 segment registers (CS, DS, ES, SS) are used in combination with the basic registers to point to memory. Series 32000 registers and addressing modes are all full 32-bit, and may point anywhere in the 16 Megabyte (or 4 Gigabyte, depending on processor model) addressing range.

Device ports are given their own 16-bit address on the 80x86, and there is a complement of instructions to handle input and output to these ports. Device ports on Series 32000 are memory mapped, and all instructions are available for port manipulation.

There are 6 addressing modes for data memory on the 80x86: Immediate, Direct, Direct indexed, Implied, Base relative and Stack. There are 9 addressing modes on Series 32000: Register, Immediate, Absolute, Register-relative, Memory space, External, Top-of-stack and Scaled index. Scaled index may be applied to any of the addressing modes (except scaled index) to create more addressing modes. The following figure shows the 80x86 addressing modes, and their Series 32000 counterparts.

Series 32000 assembly code reads left-to-right, meaning source is on the left, destination on the right. As you can see, most of the 80x86 addressing modes fall into the register-relative class of Series 32000. Also note that the ADDW could have been ADDD, performing a 32-bit add instead of only a 16-bit.

Series 32000 also permits memory-to-memory (two address) operation. A common operation like adding two variables is easier in Series 32000. Series 32000 has the same form for all math operations (multiply, divide, subtract), as well as all logical operators.

80x86		Series 32000
ADD AX,1234	Immediate	ADDW $1234,R0
ADD AX,LAB1	Direct	ADDW LAB1,R0
ADD AX,16[SI]	Direct Indexed	ADDW 16(R6),R0
ADD AX,[SI]	Implied	ADDW 0(R6),R0
ADD AX,[BX]	Base Relative	ADDW 0(R1),R0
ADD AX,[BX+SI]	Base Relative Implied	ADDW R1[R6:B],R0
ADD AX,12[BX+SI]	Base Relative Implied Indexed	ADDW 12(R1)[R6:B],R0
ADD AX,4[BP]	Stack (Relative)	ADDW 4(FP),R0
PUSH AX	Stack	MOVW R0,TOS

80x86	Series 32000	
MOV AL,LAB1 ADD LAB2,AL	ADDB LAB1,LAB2	8-Bit Add Operation
MOV AX,LAB3 ADD LAB4,AX	ADDW LAB3,LAB4	16-Bit Add Operation

MOV AX,LAB5L
ADD LAB6L,AX
MOV AX,LAB5H
ADDC LAB6H,AX
	ADDD LAB5,LAB6	32-Bit Add Operation

Most 80x86 instructions have direct Series 32000 equivalents—with a major difference. Most 80x86 instructions affect the flags. Most Series 32000 instructions do not affect the flags in the same manner. For example, the 80x86 ADD instruction affects the Overflow, Carry, Arithmetic, Zero, Sign and Parity flags. The Series 32000 ADD instruction affects the Overflow and Carry flags. Programs that rely on side-effects of instructions which set flags must be changed in order to work correctly on Series 32000.

Table I gives a general guideline of instruction correlation between 80x86 and Series 32000. Many of the common subroutines in 80x86 may be replaced by a single instruction in Series 32000 (for example, 32-bit multiply and divide routines). Many special purpose instructions exist in Series 32000, and these instructions may help to optimize various algorithms.

3.0 IMPLEMENTATION

As an example, we will show some small 80x86 programs which we wish to convert to Series 32000. The first program reads a number of bytes from a port, waiting for status information. Below is the program in 80x86 assembly language:

```
;This program reads count bytes from port ioport, waiting for bit 7 of
;statport to be active (1) before reading each byte.
        xor     bx,bx           ;zero checksum
        mov     cx,count        ;get count of bytes
        mov     es,bufseg       ;get buffer segment
        lea     di,buffer       ;point to buffer offset
l1:     mov     dx,statport     ;get status port address
l2:     in      al,dx           ;read status port
        rcl     al,1            ;move bit 7 to carry
        jnc     l2              ;loop until status available
        mov     dx,ioport       ;point to data port
        in      al,dx           ;read port
        stosb                   ;store byte
        xor     ah,ah           ;zero high part of ax
        add     bx,ax           ;add to checksum
        loop    l1              ;loop for all bytes
        ret
```

TL/EE/9699-1

A direct translation of this program to Series 32000 using Table I, appears below. Note that this program will not work directly, due to the side effect of the rcl instruction being used.

```
#This program reads count bytes from port ioport, waiting for bit 7 of
#statport to be active (1) before reading each byte.
#
# Before optimization

        xord    r1,r1           # zero checksum
        movw    $count,r2       # get count of bytes
        addr    buffer,r5       # point to buffer
l11:    addr    statport,r3     # get status port address
l12:    movb    0(r3),r0        # read status port
        rotb    $1,r0           # move bit 7 to carry <<- does not work
        bcc     l12             # branch if carry clear
        addr    ioport,r3       # point to data port
        movb    0(r3),r0        # read port
        movb    r0,0(r5)        # store byte
        addqd   1,r5
        movzbw  r0,r0           # zero high part of ax
        addw    r0,r1           # add to checksum
        acbw    -1,r2,l11       # loop for all bytes
        ret     $0
```

TL/EE/9699-2

By using some of the special Series 32000 instructions, we can make this program much faster. The ROTB wil not work to test status, so we will replace that with a TBITB instruction. Since TBITB can directly address the port, there is no need to read the status port value at all. We will remove the read status port line, and the register load of r3. Reading the IO port as well can be done directly now, and we use a zero extension to ensure the high bits are cleared in preparation for the checksum addition. Note that it is easy to do a 32-bit checksum instead of only a 16-bit. Below is the 'optimized' code:

```
#This program reads count bytes from port ioport, waiting for bit 7 of
#statport to be active (1) before reading each byte.
#
# After optimization

        xord    r1,r1           # zero checksum
        movw    $count,r2       # get count of bytes
        addr    buffer,r5       # point to buffer
l11:
l12:    tbitb   $7,statport     # is bit 7 of status port valid?
        bfc     l12             # no, loop until it is
        movzbd  ioport,r0       # read io port
        movb    r0,0(r5)        # store in buffer
        addqd   1,r5
        addw    r0,r1           # add to checksum
        acbw    -1,r2,l11       # loop for all bytes
        ret     $0
```

TL/EE/9699-3

A second program shows, in 80x86 assembler, a method to copy and convert a string from mixed case ASCII to all upper case ASCII. This program is shown below:

```
;This program translates a null terminated ASCII string to uppercase
;
        mov     ds,buf1seg      ;point to input segment
        lea     si,buf1         ;point to input string
        mov     es,buf2seg      ;point to output segment
        lea     di,buf2         ;point to output string
        cld                     ;clear direction flag (increasing add)
l1:     lodsb                   ;get a byte
        cmp     al,'a'          ;is the char less than 'a'?
        jb      l2              ;yes, branch out
        cmp     al,'z'          ;is the char greater than 'z'?
        ja      l2              ;yes, branch out
        and     al,5fh          ;and with 5f to make uppercase
l2:     stosb                   ;store the character
        or      al,al           ;is this the last char?
        jnz     l1              ;no, loop for more
        ret                     ;yes, exit
```

TL/EE/9699-4

A direct translation to Series 32000 works fine, as is shown below:

```
#This program translates a null terminate ASCII string to uppercase
#
# Before optimization

        addr    buf1,r4         # point to input string
        addr    buf2,r5         # point to output string
111:    movb    0(r4),r0        # get a byte
        addqd   1,r0
        cmpb    $'a',r0         # is the char less than 'a'?
        blo     112             # yes, branch out
        cmpb    $'z',r0         # is the char greater than 'z'?
        bhi     112             # yes, branch out
```

TL/EE/9699-5

```
        andb    $0x5f,r0        # and with 5f to make uppercase
112:    movb    r0,0(r5)        # store the character
        addqd   1,r5
        cmpqb   0,r0            # is this the last char?
        bne     111             # no, loop for more
        ret     $0
```

TL/EE/9699-6

This program allows us to exploit another Series 32000 instruction, the MOVST (Move and String Translate). With a 256 byte external table, we can translate any byte to any other byte. In this example, we simply use the full range of ASCII values in the translation table, with the lower case entries containing uppercase values.

Watch for other optimization opportunities, especially with multiply and add sequences (the INDEXi instruction could be used), and possible memory to memory sequence changes. When optimizing Series 32000 code, it is important to fully utilize the Complex Instruction Set. Allow the fewest number of instructions possible to do the work. Use the advanced addressing modes where possible. Try to employ larger data types in programs (Series 32000 takes the same number of clocks to add Bytes, Words or Double words).

4.0 CONCLUSION

Series 32000 assembly language offers a much richer complement of instructions when compared to the 80x86 assembly language. Translation from 80x86 to Series 32000 is made much easier by this full instruction set.

```
#This program translates a null terminate ASCII string to uppercase
#
# After optimization

        movqd   -1,r0           # number of bytes in string max.
        addr    buf1,r1         # point to input string
        addr    buf2,r2         # point to output string
        addr    ctable,r3       # address of conversion table
        movqd   0,r4            # match on a zero
        movst   u               # move string, translate, until 0
        movqb   0,0(r2)         # move a zero to output string
        ret     $0
```

TL/EE/9699-7

TABLE I

The following is a conversion table from 80x86 mnemonics to Series 32000. Note that many of the conversions are not exact, as the 80x86 instructions may affect flags that Series 32000 instructions do not. A * marks those instructions that may be affected most by this change in flags. The i in the Series 32000 instructions refers to the size of the data to be operated on. It may be B for Byte, W for Word or D for Double. Most arithmetic instructions also support F for single-precision Floating Point, and L for double-precision Floating-Point.

80x86	Series 32000	Comments
AAA	—	Suggest changing algorithm to use ADDPi
AAD	—	Suggest changing algorithm to use ADDPi/SUBPi
AAM	—	"
AAS	—	Suggest changing algorithm to use SUBPi
ADC	ADDCi	
ADD	ADDi	
AND	ANDi	
BOUND	CHECKi	
CALL	BSR/JSR	
CBW	MOVXBW	You may directly sign-extend data while moving
CLC	BICPSRB $1	Usually not required
CLD	—	Direction encoded within string instructions
CLI	BICPSRW $0x800	Supervisor mode instruction
CMC	—	Usually not required
CMP	CMPi	
CMPS	CMPSi	Many options available
CWD	MOVXWD	You may directly sign-extend data while moving
DAA	—	Suggest changing algorithm to use ADDPi
DAS	—	Suggest changing algorithm to use SUBPi
DEC	ADDQi-1*	Watch for flag usage
DIV	DIVi	Note: Series 32000 uses signed division
ENTER	ENTER[reglist],d	Builds stack frame, saves regs, allocates stack space
ESC	—	Usually used for Floating Point-see Series 32000 FP instructions
HLT	WAIT	
IDIV	DIVi/QUOi	DIVi rounds towards -infinity, QUOi to zero
IMUL	MULi	
IN	—	Series 32000 uses memory-mapped I/O
INC	ADDQi 1*	Watch for flag usage
INS	—	Series 32000 uses memory mapped I/O
INT	SVC	Not exact conversion, but usually used to call O/S
INTO	FLAG	Trap on overflow
IRET	RETI $0	Causes Interrupt Acknowledge cycle
JA/JNBE	BHI	Unsigned comparison
JAE/JNB	BHS	Unsigned comparison
JB/JNAE	BLT	Unsigned comparison
JBE/JNA	BLS	Unsigned comparison
JCXZ	—	Use CMPQi 0, followed by BEQ
JE/JZ	BEQ	Equal comparison
JG/JNLE	BGT	Signed comparison
JGE/JNL	BGE	Signed comparison
JL/JNGE	BLT	Signed comparison
JLE/JNG	BLE	Signed comparison
JMP	BR/JUMP	
JNE/JNZ	BNE	Not Equal comparison
JNO	—	Subroutines should be used for these instructions
JNP	—	as most Series 32000 code will not need these
JNS	—	operations.
JO	—	"
JP	—	"
JPE	—	"
JPO	—	"
JS	—	"
LAHF	—	SPRB UPSR,xxx may be useful
LDS	—	Segment registers not required on Series 32000
LEA	ADDR	
LEAVE	EXIT[reglist]	Restores regs, unallocates frame and stack
LES	—	Segment registers not required
LOCK	—	SBITIi, CBITIi interlocked instructions
LODS	MOVi/ADDQD	MOV instruction followed by address increment
LOOP	ACBi-1	ACBi may use memory or register

TABLE I (Continued)

80x86	Series 32000	Comments
LOOPE	—	BEQ followed by ACBi may be used
LOOPNE	—	BNE followed by ACBi may be used
LOOPNZ	—	BNE followed by ACBi may be used
LOOPZ	—	BEQ followed by ACBi may be used
MOV	MOVi	
MOVS	MOVSi	Many options available
MUL	MULi	Series 32000 uses signed multiplication
NEG	NEGi	Two's complement
NOP	NOP	
NOT	COMi	One's complement
OR	ORi	
OUT	—	Series 32000 uses memory mapped I/O
OUTS	—	Series 32000 uses memory mapped I/O
POP	MOVi TOS,	TOS addressing mode auto increments/decrements SP
POPA	RESTORE [r0,r1 . . r7]	Restores list of registers
POPF	LPRB UPSR,TOS	User mode loads 8 bits, supervisor 16 bits of PSR
PUSH	MOVi xx,TOS	Any data may be moved to TOS
PUSHA	SAVE [r0,r1 . . r7]	Saves list of registers
PUSHF	SPRB UPSR,TOS	User mode stores 8 bits, supervisor 16 bits of PSR
RCL	ROTi*	Does not rotate through carry
RCR	ROTi*	Does not rotate through carry
REP	—	Series 32000 string instructions use 32-bit counts
RET	RET	
ROL	ROTi	
ROR	ROTi	Rotates work in both directions
SAHF	—	LPRB UPSR,xx may be useful
SAL	ASHi	Arithmetic shift
SAR	ASHi	Arithmetic shift works both directions
SBB	SUBCi	
SCAS	SKPSi	Many options available
SHL	LSHi	Logical shift
SHR	LSHi	Logical shift works both directions
STC	BISPSRB $1	
STD	—	Direction is encoded in string instructions
STI	BISPSRW $0x800	Supervisor mode instruction
STOS	MOVi/ADDQD	MOV instruction followed by address increment
SUB	SUBi	
TEST	—	TBITi may be used as a substitute
WAIT	—	
XCHG	—	MOVi x,temp; MOVi y,x; MOVi temp,y
XLAT	MOVi x[R0:b],	Scaled index addressing mode
XOR	XORi	

Bit Mirror Routine; Series 32000® Graphics Note 7

National Semiconductor
Application Note 530
Dave Rand

1.0 INTRODUCTION

The bit mirror routine is designed to reorder the bits in an image. The bits are swapped around a fixed point, that being one half of the size of the data, as is shown for the byte mirror below. These routines can be used for conversion of 68000 based data.

2.0 DESCRIPTION

	Bit Number								Hex Value
	7	6	5	4	3	2	1	0	
Source	1	0	1	1	0	0	1	0	B2
Result of Mirror	0	1	0	0	1	1	0	1	4D

The "mirror", in this case, is between bits 3 and 4.

Several different algorithms are available for the mirror operation. The best algorithm to mirror a byte takes 20 clocks on a NS32016 (about 2.5 clocks per bit), and uses a 256 byte table to do the mirror operation. The table is reproduced at the end of this document. To perform a byte mirror, the following code may be used. The byte to be mirrored is in R0, and the destination is to be R1.

```
MOVB mirtab[r0:b],r1     #Mirror a byte
```

TL/EE/9700–1

An extension of this algorithm is used to mirror larger amounts of data. To mirror a 32-bit block of data from one location to another, the following code may be used. Register R0 points to the source block, register R1 points to the destination. R2 is used as a temporary value.

```
MOVZBD    0(r0),r2              #get first byte
MOVB      mirtab[r2:b],3(r1)    #store in last place
MOVB      1(r0),r2              #get next byte
MOVB      mirtab[r2:b],2(r1)    #store in next place
MOVB      2(r0),r2              #get the third byte
MOVB      mirtab[r2:b],1(r1)    #store in next place
MOVB      3(r0),r2              #get the last byte
MOVB      mirtab[r2:b],0(r1)    #first place
```

TL/EE/9700–2

This code uses 33 bytes of memory, and just 169 clocks to execute. Larger blocks of data can be mirrored with this method as well, with each additional byte taking about 40 clocks.

Registers can also be mirrored with this method, with just a few more instructions. To mirror R0 to R1, for example, the following code could be used. R2 is used as a temporary variable.

```
MOVZBD    r0,r2           #get lsbyte
MOVB      mirtab[r2:b],r1 #mirror the byte
LSHD      $8,r1           #move into higher byte of destination
LSHD      $-8,r0          #and of source
MOVB      r0,r2           #get lsbyte
MOVB      mirtab[r2:b],r1 #mirror the byte
LSHD      $8,r1           #move into higher byte of destination
LSHD      $-8,r0          #and of source
MOVB      r0,r2           #get lsbyte
MOVB      mirtab[r2:b],r1 #mirror the byte
LSHD      $8,r1           #move into higher byte of destination
LSHD      $-8,r0          #and of source
MOVB      r0,r2           #get lsbyte
MOVB      mirtab[r2:b],r1 #mirror the byte
```

TL/EE/9700–3

This code occupies 49 bytes, and executes in 286 clocks on an NS32016.

If space is at a premium, a shorter table may be used, at the expense of time. Each nibble (4 bits) instead of each byte is processed. This means that the table only requires 16 entries. To mirror a byte in R0 to R1, the following code can be used. R2 is used as a temporary variable.

```
MOVB    r0,r2               #get lsbyte
ANDD    $15,r2              #mask to get ls nibble
MOVB    mirtb16[r2:b],r1    #mirror the nibble
LSHD    $4,r1               #high nibble of destination
LSHD    $-4,r0              #and of source
MOVB    r0,r2               #get lsbyte
ANDD    $15,r2              #mask to get ls nibble
ORB     mirtb16[r2:b],r1    #mirror the nibble
```

TL/EE/9700–4

This code requires 32 bytes of memory, and executes in 125 clock cycles on an NS32016. A slightly faster time (100 clocks) may be obtained by adding a second table for the high nibble, and eliminating the LSHD 4,r1 instruction.

TABLES

MIRTAB is a table of all possible mirror values of 8 bits, or 256 bytes. MIRTB16 is a table of all possible mirror values of 4 bits, or 16 bytes. These tables should be aligned for best performance. They may reside in code (PC relative), or data (SB relative) space.

```
mirtab:
        .byte   0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50
        .byte   0xd0,0x30,0xb0,0x70,0xf0
        .byte   0x08,0x88,0x48,0xc8,0x28,0xa8,0x68,0xe8,0x18,0x98,0x58
        .byte   0xd8,0x38,0xb8,0x78,0xf8
        .byte   0x04,0x84,0x44,0xc4,0x24,0xa4,0x64,0xe4,0x14,0x94,0x54
        .byte   0xd4,0x34,0xb4,0x74,0xf4
        .byte   0x0c,0x8c,0x4c,0xcc,0x2c,0xac,0x6c,0xec,0x1c,0x9c,0x5c
        .byte   0xdc,0x3c,0xbc,0x7c,0xfc
        .byte   0x02,0x82,0x42,0xc2,0x22,0xa2,0x62,0xe2,0x12,0x92,0x52
        .byte   0xd2,0x32,0xb2,0x72,0xf2
        .byte   0x0a,0x8a,0x4a,0xca,0x2a,0xaa,0x6a,0xea,0x1a,0x9a,0x5a
        .byte   0xda,0x3a,0xba,0x7a,0xfa
        .byte   0x06,0x86,0x46,0xc6,0x26,0xa6,0x66,0xe6,0x16,0x96,0x56
        .byte   0xd6,0x36,0xb6,0x76,0xf6
        .byte   0x0e,0x8e,0x4e,0xce,0x2e,0xae,0x6e,0xee,0x1e,0x9e,0x5e
        .byte   0xde,0x3e,0xbe,0x7e,0xfe
        .byte   0x01,0x81,0x41,0xc1,0x21,0xa1,0x61,0xe1,0x11,0x91,0x51
        .byte   0xd1,0x31,0xb1,0x71,0xf1
        .byte   0x09,0x89,0x49,0xc9,0x29,0xa9,0x69,0xe9,0x19,0x99,0x59
        .byte   0xd9,0x39,0xb9,0x79,0xf9
        .byte   0x05,0x85,0x45,0xc5,0x25,0xa5,0x65,0xe5,0x15,0x95,0x55
        .byte   0xd5,0x35,0xb5,0x75,0xf5
        .byte   0x0d,0x8d,0x4d,0xcd,0x2d,0xad,0x6d,0xed,0x1d,0x9d,0x5d
        .byte   0xdd,0x3d,0xbd,0x7d,0xfd
        .byte   0x03,0x83,0x43,0xc3,0x23,0xa3,0x63,0xe3,0x13,0x93,0x53
        .byte   0xd3,0x33,0xb3,0x73,0xf3
        .byte   0x0b,0x8b,0x4b,0xcb,0x2b,0xab,0x6b,0xeb,0x1b,0x9b,0x5b
        .byte   0xdb,0x3b,0xbb,0x7b,0xfb
        .byte   0x07,0x87,0x47,0xc7,0x27,0xa7,0x67,0xe7,0x17,0x97,0x57
        .byte   0xd7,0x37,0xb7,0x77,0xf7
        .byte   0x0f,0x8f,0x4f,0xcf,0x2f,0xaf,0x6f,0xef,0x1f,0x9f,0x5f
        .byte   0xdf,0x3f,0xbf,0x7f,0xff

mirtb16:
        .byte   0x0,0x8,0x4,0xc,0x2,0xa,0x6,0xe,0x1,0x9,0x5
        .byte   0xd,0x3,0xb,0x7,0xf
```

TL/EE/9700–5

Instruction Execution Times of FPU NS32081 Considered for Stand-Alone Configurations

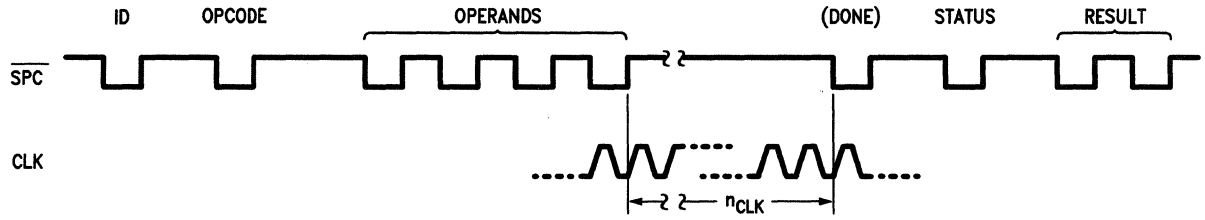
National Semiconductor
Application Brief 26
Systems & Applications Group

The table below gives execution timing information for the FPU NS32081.

The number of clock cycles nCLK is counted from the last SPC pulse, strobing the last operation word or operand into the FPU, and the Done-SPC pulse, which signals the CPU that the result is available (see *Figure 1*). The values are therefore independent of the operand's addressing modes and do not include the CPU/FPU protocol time. This makes it easy to determine the FPU execution times in stand-alone configurations.

The values are derived from measurements, the worst case is always assumed. The results are given in clock cycles (CLK).

Operation	Number of Clock-Cycles nCLK
Add, Subtract	63
Multiply Float	37
Multiply Long	51
Divide Float	78
Divide Long	108
Compare	38



TL/EE/8760–1

FIGURE 1

NATIONAL SEMICONDUCTOR CORPORATION DISTRIBUTORS

ALABAMA
Huntsville
Arrow Electronics
(205) 837-6955
Bell Industries
(205) 837-1074
Hamilton/Avnet
(205) 837-7210
Pioneer
(205) 837-9300

ARIZONA
Phoenix
Arrow Electronics
(602) 437-0750
Tempe
Anthem Electronics
(602) 966-6600
Bell Industries
(602) 966-7800
Hamilton/Avnet
(602) 961-6400

CALIFORNIA
Agora Hills
Zeus Components
(818) 889-3838
Anaheim
Time Electronics
(714) 934-0911
Zeus Components
(714) 921-9000
Chatsworth
Anthem Electronics
(818) 700-1000
Arrow Electronics
(818) 701-7500
Hamilton Electro Sales
(818) 700-6500
Time Electronics
(818) 998-7200
Costa Mesa
Avnet Electronics
(714) 754-6050
Hamilton Electro Sales
(714) 641-4159
Garden Grove
Bell Industries
(714) 895-7801
Gardena
Bell Industries
(213) 515-1800
Hamilton Electro Sales
(213) 217-6751
Irvine
Anthem Electronics
(714) 768-4444
Ontario
Hamilton/Avnet
(714) 989-4602
Rocklin
Bell Industries
(916) 969-3100
Roseville
Bell Industries
(916) 969-3100
Sacramento
Anthem Electronics
(916) 922-6800
Hamilton/Avnet
(916) 925-2216
San Diego
Anthem Electronics
(619) 453-9005
Arrow Electronics
(619) 565-4800
Hamilton/Avnet
(619) 571-7510
Time Electronics
(619) 586-1331
San Jose
Anthem Electronics
(408) 295-4200
Zeus Components
(408) 998-5121
Sunnyvale
Arrow Electronics
(408) 745-6600
Bell Industries
(408) 734-8570
Hamilton/Avnet
(408) 743-3355
Time Electronics
(408) 734-9888
Thousand Oaks
Bell Industries
(805) 499-6821
Torrance
Time Electronics
(213) 320-0880
Tustin
Arrow Electronics
(714) 838-5422
Yorba Linda
Zeus Components
(714) 921-9000

COLORADO
Englewood
Anthem Electronics
(303) 790-4500
Arrow Electronics
(303) 790-4444
Hamilton/Avnet
(303) 799-9998
Wheatridge
Bell Industries
(303) 424-1985

CONNECTICUT
Cheshire
Time Electronics
(203) 271-3200
Danbury
Hamilton/Avnet
(203) 797-2800
Meridan
Anthem Electronics
(203) 237-2282
Norwalk
Pioneer Northeast
(203) 853-1515
Wallingford
Arrow Electronics
(203) 265-7741

FLORIDA
Altamonte Springs
Arrow/Kierulff Electronics
(305) 682-6923
Pioneer
(305) 834-9090
Deerfield Beach
Arrow Electronics
(305) 429-8200
Bell Industries
(305) 421-1997
Pioneer
(305) 428-8877
Fort Lauderdale
Hamilton/Avnet
(305) 971-2900
Lake Mary
Arrow Electronics
(407) 323-0252
Largo
Bell Industries
(813) 541-4434
Orlando
Chip Supply
(305) 298-7100
Oviedo
Zeus Components
(407) 365-3000
Palm Bay
Arrow Electronics
(305) 725-1480
St. Petersburg
Hamilton/Avnet
(813) 576-3930
Winter Park
Hamilton/Avnet
(407) 628-3888

GEORGIA
Norcross
Arrow Electronics
(404) 449-8252
Bell Industries
(404) 662-0923
Hamilton/Avnet
(404) 447-7500
Pioneer
(404) 448-1711

ILLINOIS
Bensenville
Hamilton/Avnet
(312) 860-7780
Elk Grove Village
Anthem Electronics
(312) 640-6066
Bell Industries
(312) 640-1910
Itasca
Arrow Electronics
(312) 250-0500
Urbana
Bell Industries
(217) 328-1077

INDIANA
Carmel
Hamilton/Avnet
(317) 844-9333
Fort Wayne
Bell Industries
(219) 423-3422
Indianapolis
Advent Electronics Inc.
(317) 872-4910
Arrow Electronics
(317) 243-9353
Bell Industries
(317) 634-8202
Pioneer
(317) 849-7300

IOWA
Cedar Rapids
Advent Electronics
(319) 363-0221
Arrow Electronics
(319) 395-7230
Bell Industries
(319) 395-0730
Hamilton/Avnet
(319) 362-4757

KANSAS
Lenexa
Arrow Electronics
(913) 541-9542
Overland Park
Hamilton/Avnet
(913) 888-8900

MARYLAND
Columbia
Anthem Electronics
(301) 995-6640
Arrow Electronics
(301) 995-0003
Hamilton/Avnet
(301) 995-3500
Lionex Corp.
(301) 964-0040
Time Electronics
(301) 964-3090
Zeus Components
(301) 997-1118
Gaithersburg
Pioneer
(301) 921-0660

MASSACHUSETTS
Lexington
Pioneer Northeast
(617) 861-9200
Zeus Components
(617) 863-8800
Norwood
Gerber Electronics
(617) 769-6000
Peabody
Hamilton/Avnet
(617) 531-7430
Sertech Laboratories
(617) 532-5105
Time Electronics
(617) 532-6200
Wilmington
Anthem Electronics
(617) 657-5170
Arrow Electronics
(617) 935-5134
Lionex Corporation
(617) 657-5170

MICHIGAN
Ann Arbor
Arrow Electronics
(313) 971-8220
Bell Industries
(313) 971-9093
Grand Rapids
Arrow Electronics
(616) 243-0912
Hamilton/Avnet
(616) 243-8805
Pioneer Standard
(616) 698-1800
Livonia
Hamilton/Avnet
(313) 522-4700
Pioneer
(313) 525-1800
Wyoming
R. M. Michigan, Inc.
(616) 531-9300

MINNESOTA
Eden Prairie
Anthem Electronics
(612) 944-5454
Pioneer-Twin Cities
(612) 944-3355
Edina
Arrow Electronics
(612) 830-1800
Minnetonka
Hamilton/Avnet
(612) 932-0600

MISSOURI
Earth City
Hamilton/Avnet
(314) 344-1200
St. Louis
Arrow Electronics
(314) 567-6888
Time Electronics
(314) 391-6444

NEW HAMPSHIRE
Hudson
Bell Industries
(603) 882-1133
Manchester
Arrow Electronics
(603) 668-6968
Hamilton/Avnet
(603) 624-9400

NEW JERSEY
Cherry Hill
Hamilton/Avnet
(609) 424-0100
Fairfield
Hamilton/Avnet
(201) 575-3390
Lionex Corporation
(201) 227-7960
Marlton
Arrow Electronics
(609) 596-8000
Parsippany
Arrow Electronics
(201) 538-0900
Pine Brook
Nu Horizons Electronics
(201) 882-8300
Pioneer
(201) 575-3510

NEW MEXICO
Albuquerque
Alliance Electronics Inc.
(505) 292-3360
Arrow Electronics
(505) 243-4566
Bell Industries
(505) 292-2700
Hamilton/Avnet
(505) 765-1500

NEW YORK
Amityville
Nu Horizons Electronics
(516) 226-6000
Binghamton
Pioneer
(607) 722-9300
Buffalo
Summit Distributors
(716) 887-2800
Fairport
Pioneer Northeast
(716) 381-7070

NATIONAL SEMICONDUCTOR CORPORATION DISTRIBUTORS (Continued)

NEW YORK (Continued)
Hauppauge
Anthem Electronics
(516) 273-1660
Arrow Electronics
(516) 231-1000
Hamilton/Avnet
(516) 434-7413
Lionex Corporation
(516) 273-1660
Time Electronics
(516) 273-0100
Port Chester
Zeus Components
(919) 937-7400
Rochester
Arrow Electronics
(716) 427-0300
Hamilton/Avnet
(716) 475-9130
Summit Electronics
(716) 334-8100
Ronkonkoma
Zeus Components
(516) 737-4500
Syracuse
Hamilton/Avnet
(315) 437-2641
Time Electronics
(315) 432-0355
Westbury
Hamilton/Avnet
(516) 997-6868

NORTH CAROLINA
Charlotte
Pioneer
(704) 527-8188
Durham
Pioneer Technology
(919) 544-5400
Raleigh
Arrow Electronics
(919) 876-3132
Hamilton/Avnet
(919) 878-0810
Winston-Salem
Arrow Electronics
(919) 725-8711

OHIO
Centerville
Arrow Electronics
(513) 435-5563
Cleveland
Pioneer
(216) 587-3600
Dayton
Bell Industries
(513) 435-8660
Bell Industries
(513) 434-8231
Hamilton/Avnet
(513) 439-6700
Pioneer
(513) 236-9900
Zeus Components
(914) 937-7400
Highland Heights
CAM/Ohio Electronics
(216) 461-4700
Solon
Arrow Electronics
(216) 248-3990
Hamilton/Avnet
(216) 831-3500
Westerville
Hamilton/Avnet
(614) 882-7004

OKLAHOMA
Tulsa
Arrow Electronics
(918) 252-7537
Hamilton/Avnet
(918) 252-7297
Quality Components
(918) 664-8812
Radio Inc.
(918) 587-9123

OREGON
Beaverton
Almac-Stroum Electronics
(503) 629-8090
Anthem Electronics
(503) 643-1114
Arrow Electronics
(503) 645-6456
Lake Oswego
Bell Industries
(503) 241-4115
Hamilton/Avnet
(503) 635-7850

PENNSYLVANIA
Horsham
Anthem Electronics
(215) 443-5150
Lionex Corp.
(215) 443-5150
Pioneer
(215) 674-4000
King of Prussia
Time Electronics
(215) 337-0900
Monroeville
Arrow Electronics
(412) 856-7000
Pittsburgh
Hamilton/Avnet
(412) 281-4150
Pioneer
(412) 782-2300
CAM/RPC Ind. Elec.
(412) 782-3770

TEXAS
Addison
Quality Components
(214) 733-4300
Austin
Arrow Electronics
(512) 835-4180
Hamilton/Avnet
(512) 837-8911
Pioneer
(512) 835-4000
Quality Components
(512) 835-0220
Minco Technology Labs
(512) 834-2022
Carrollton
Arrow Electronics
(214) 380-6464
Dallas
Pioneer
(214) 386-7300
Houston
Arrow Electronics
(713) 530-4700
Pioneer
(713) 988-5555
Irving
Hamilton/Avnet
(214) 550-7755
Richardson
Anthem Electronics
(214) 238-0237
Zeus Components
(214) 783-7010
Stafford
Hamilton/Avnet
(713) 240-7733
Sugarland
Quality Components
(713) 240-2255

UTAH
Midvale
Bell Industries
(801) 972-6969
Salt Lake City
Anthem Electronics
(801) 973-8555
Arrow Electronics
(801) 973-6913
Bell Industries
(801) 972-6969
Hamilton/Avnet
(801) 972-4300

WASHINGTON
Bellevue
Almac-Stroum Electronics
(206) 643-9992
Hamilton/Avnet
(206) 453-5844
Kent
Arrow Electronics
(206) 575-4420
Redmond
Anthem Electronics
(206) 881-0850
Hamilton/Avnet
(206) 867-0148

WISCONSIN
Brookfield
Arrow Electronics
(414) 792-0150
Mequon
Taylor Electric
(414) 241-4321
Waukesha
Bell Industries
(414) 547-8879
Hamilton/Avnet
(414) 784-4516

CANADA
WESTERN PROVINCES
Burnaby
Hamilton/Avnet
(604) 437-6667
Semad Electronics
(604) 438-2515
Calgary
Hamilton/Avnet
(403) 250-9380
Semad Electronics
(403) 252-5664
Zentronics
(403) 272-1021
Edmonton
Zentronics
(403) 468-9306
Richmond
Zentronics
(604) 273-5575
Saskatoon
Zentronics
(306) 955-2207
Winnipeg
Zentronics
(204) 694-1957

EASTERN PROVINCES
Brampton
Zentronics
(416) 451-9600
Mississauga
Hamilton/Avnet
(416) 677-7432
Nepean
Hamilton/Avnet
(613) 226-1700
Zentronics
(613) 226-8840
Ottawa
Semad Electronics
(613) 727-8325
Pointe Claire
Semad Electronics
(514) 694-0860
St. Laurent
Hamilton/Avnet
(514) 335-1000
Zentronics
(514) 737-9700
Waterloo
Zentronics
(800) 387-2329
Willowdale
ElectroSound Inc.
(416) 494-1666

National Semiconductor

National Semiconductor
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: (408) 721-5000
TWX: (910) 339-9240

SALES OFFICES

ALABAMA
Huntsville
(205) 837-8960
(205) 721-9367

ARIZONA
Tempe
(602) 966-4563

B.C.
Burnaby
(604) 435-8107

CALIFORNIA
Encino
(818) 888-2602
Inglewood
(213) 645-4226
Roseville
(916) 786-5577
San Diego
(619) 587-0666
Santa Clara
(408) 562-5900
Tustin
(714) 259-8880
Woodland Hills
(818) 888-2602

COLORADO
Boulder
(303) 440-3400
Colorado Springs
(303) 578-3319
Englewood
(303) 790-8090

CONNECTICUT
Fairfield
(203) 371-0181
Hamden
(203) 288-1560

FLORIDA
Boca Raton
(305) 997-8133
Orlando
(305) 629-1720
St. Petersburg
(813) 577-1380

GEORGIA
Atlanta
(404) 396-4048
Norcross
(404) 441-2740

ILLINOIS
Schaumburg
(312) 397-8777

INDIANA
Carmel
(317) 843-7160
Fort Wayne
(219) 484-0722

IOWA
Cedar Rapids
(319) 395-0090

KANSAS
Overland Park
(913) 451-8374

MARYLAND
Hanover
(301) 796-8900

MASSACHUSETTS
Burlington
(617) 273-3170
Waltham
(617) 890-4000

MICHIGAN
W. Bloomfield
(313) 855-0166

MINNESOTA
Bloomington
(612) 835-3322
(612) 854-8200

NEW JERSEY
Paramus
(201) 599-0955

NEW MEXICO
Albuquerque
(505) 884-5601

NEW YORK
Endicott
(607) 757-0200
Fairport
(716) 425-1358
(716) 223-7700
Melville
(516) 351-1000
Wappinger Falls
(914) 298-0680

NORTH CAROLINA
Cary
(919) 481-4311

OHIO
Dayton
(513) 435-6886
Highland Heights
(216) 442-1555
(216) 461-0191

ONTARIO
Mississauga
(416) 678-2920
Nepean
(404) 441-2740
(613) 596-0411
Woodbridge
(416) 746-7120

OREGON
Portland
(503) 639-5442

PENNSYLVANIA
Horsham
(215) 675-6111
Willow Grove
(215) 657-2711

PUERTO RICO
Rio Piedias
(809) 758-9211

QUEBEC
Dollard Des Ormeaux
(514) 683-0683
Lachine
(514) 636-8525

TEXAS
Austin
(512) 346-3990
Houston
(713) 771-3547
Richardson
(214) 234-3811

UTAH
Salt Lake City
(801) 322-4747

WASHINGTON
Bellevue
(206) 453-9944

WISCONSIN
Brookfield
(414) 782-1818
Milwaukee
(414) 527-3800

INTERNATIONAL OFFICES

Electronica NSC de Mexico SA
Juventino Rosas No. 118-2
Col Guadalupe Inn
Mexico, 01020 D.F. Mexico
Tel: 52-5-524-9402

National Semicondutores Do Brasil Ltda.
Av. Brig. Faria Lima, 1409
6 Andor Salas 62/64
01451 Sao Paulo, SP, Brasil
Tel: (55/11) 212-5066
Telex: 391-1131931 NSBR BR

National Semiconductor GmbH
Industriestrasse 10
D-8080 Furstenfeldbruck
West Germany
Tel: 49-08141-103-0
Telex: 527 649

National Semiconductor (UK) Ltd.
301 Harpur Centre
Horne Lane
Bedford MK40 ITR
United Kingdom
Tel: (02 34) 27 00 27
Telex: 826 209

National Semiconductor Benelux
Vorstlaan 100
B-1170 Brussels
Belgium
Tel: (02) 6725360
Telex: 61007

National Semiconductor (UK) Ltd.
1, Bianco Lunos Alle
DK-1868 Fredriksberg C
Denmark
Tel: (01) 213211
Telex: 15179

National Semiconductor
Expansion 10000
28, rue de la Redoute
F-92260 Fontenay-aux-Roses
France
Tel: (01) 46 60 81 40
Telex: 250956

National Semiconductor S.p.A.
Strada 7, Palazzo R/3
20089 Rozzano
Milanofiori
Italy
Tel: (02) 8242046/7/8/9

National Semiconductor AB
Box 2016
Stensatravagen 13
S-12702 Skarholmen
Sweden
Tel: (08) 970190
Telex: 10731

National Semiconductor
Calle Agustin de Foxa, 27
28036 Madrid
Spain
Tel: (01) 733-2958
Telex: 46133

National Semiconductor Switzerland
Alte Winterthurerstrasse 53
Postfach 567
Ch-8304 Wallisellen-Zurich
Switzerland
Tel: (01) 830-2727
Telex: 59000

National Semiconductor
Kauppakartanonkatu 7
SF-00930 Helsinki
Finland
Tel: (0) 33 80 33
Telex: 126116

National Semiconductor Japan Ltd.
Sanseido Bldg. 5F
4-15 Nishi Shinjuku
Shinjuku-ku
Tokyo 160 Japan
Tel: 3-299-7001
Fax: 3-299-7000

National Semiconductor Hong Kong Ltd.
Southeast Asia Marketing
Austin Tower, 4th Floor
22-26A Austin Avenue
Tsimshatsui, Kowloon, H.K.
Tel: 852 3-7243645
Cable: NSSEAMKTG
Telex: 52996 NSSEA HX

National Semiconductor (Australia) PTY, Ltd.
1st Floor, 441 St. Kilda Rd.
Melbourne, 3004
Victory, Australia
Tel: (03) 267-5000
Fax: 61-3-2677458

National Semiconductor (PTE), Ltd.
200 Cantonment Road 13-01
Southpoint
Singapore 0208
Tel: 2252226
Telex: RS 33877

National Semiconductor (Far East) Ltd.
Taiwan Branch
P.O. Box 68-332 Taipei
7th Floor, Nan Shan Life Bldg.
302 Min Chuan East Road,
Taipei, Taiwan R.O.C.
Tel: (86) 02-501-7227
Telex: 22837 NSTW
Cable: NSTW TAIPEI

National Semiconductor (Far East) Ltd.
Korea Office
Room 612,
Korea Fed. of Small Bus. Bldg.
16-2, Yoido-Dong,
Youngdeungpo-Ku
Seoul, Korea
Tel: (02) 784-8051/3 – 785-0696-8
Telex: K24942 NSRKLO

 TL/2522 RRD-RRD15M108/Printed in U.S.A.