

# NS32GX320-20/NS32GX320-25/NS32GX320-30

## High-Performance 32-Bit Integrated System Processor

### General Description

The NS32GX320 is a highly-integrated high-performance member of the Series 32000/EPTM family of National's Embedded System Processors™ specifically designed for computation-intensive, embedded applications. It is software compatible with the previous microprocessors in the family while providing new features to better support Graphics and Digital Signal Processing.

The NS32GX320 CPU core incorporates a 4-stage instruction pipeline, on-chip instruction and data caches, a barrel shifter and a hardware multiplier unit. The internal organization allows a high degree of parallelism in executing instructions.

Integrated on the same chip with the CPU are also a 2-channel DMA controller, a 15-level interrupt control unit (ICU) and three 16-bit timers. These make the device extremely attractive for those cost-sensitive applications where a high level of integration is required.

The system interface is also optimized to support applications spanning a wide range, from low-cost, real-time controllers to highly sophisticated, embedded systems.

The NS32GX320 integrates more than 390,000 transistors fabricated in sub-micron, double-metal CMOS technology. The advanced technology and mainframe-like design allow the device to achieve peak performance of 15 million instructions per second.

### Features

- 32-bit architecture and implementation
- 4-GByte uniform addressing space
- 512-Byte on-chip instruction cache
- 1024-Byte on-chip data cache
- Very Efficient DSP Support
  - 32-bit Add 67 ns
  - 16x16-to-32-bit Multiply 200 ns
  - Multiply and Accumulate Double 367 ns
  - Complex Multiply and Accumulate 400 ns
- High-performance/low-cost bus
  - Separate 32-bit address and data lines
  - Burst mode memory accessing
  - Dynamic bus sizing
  - Support for page-mode and static-column DRAMs
  - Idle states for slow peripherals
- On-chip 15-level Interrupt Control Unit
- On-chip 2-channel DMA Controller (30 Mbytes/sec)
- Three on-chip Timers
- Floating-point support via the NS32381
- Software compatible with the Series 32000 architecture
- Double-metal CMOS technology
- 175-Pin Plastic Pin Grid Array (PGA) package

**NS32GX320-20/NS32GX320-25/NS32GX320-30**  
High-Performance 32-Bit Integrated System Processor

### Block Diagram

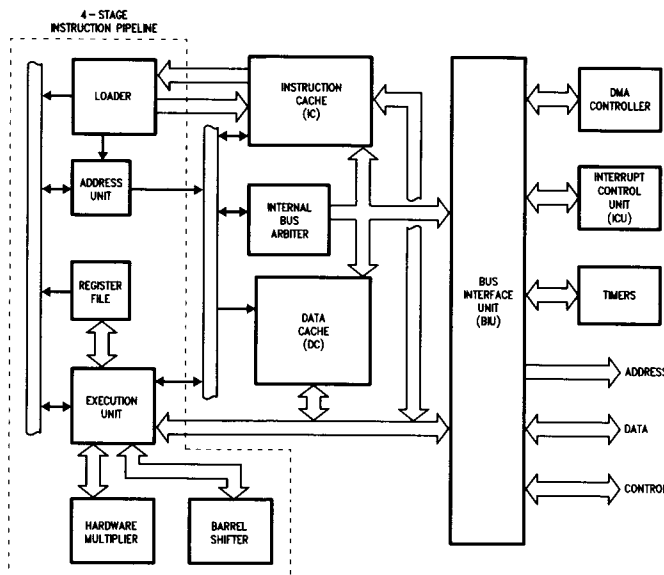


FIGURE 1

TL/EE/10564-1

# Table of Contents

## 1.0 PRODUCT INTRODUCTION

## 2.0 ARCHITECTURAL DESCRIPTION

### 2.1 Register Set

- 2.1.1 General Purpose Registers
- 2.1.2 Address Registers
- 2.1.3 Processor Status Register
- 2.1.4 Configuration Register
- 2.1.5 Debug Registers
- 2.1.6 DMA Controller Registers
- 2.1.7 Interrupt Control Unit (ICU) Registers
- 2.1.8 Timers Registers

### 2.2 Memory Organization

- 2.2.1 Address Mapping

### 2.3 Modular Software Support

### 2.4 Instruction Set

- 2.4.1 General Instruction Format
- 2.4.2 Addressing Modes
- 2.4.3 Instruction Set Summary

## 3.0 FUNCTIONAL DESCRIPTION

### 3.1 Instruction Execution

- 3.1.1 Operating States
- 3.1.2 Instruction Endings
  - 3.1.2.1 Completed Instructions
  - 3.1.2.2 Suspended Instructions
  - 3.1.2.3 Terminated Instructions
  - 3.1.2.4 Partially Completed Instructions
- 3.1.3 Instruction Pipeline
  - 3.1.3.1 Branch Prediction
  - 3.1.3.2 Memory Mapped I/O
  - 3.1.3.3 Serializing Operations
- 3.1.4 Slave Processor Instructions
  - 3.1.4.1 Slave Instruction Protocol
  - 3.1.4.2 Floating-Point Instructions
  - 3.1.4.3 Custom Slave Instructions

### 3.2 Exception Processing

- 3.2.1 Exception Acknowledge Sequence
- 3.2.2 Returning from an Exception Service Procedure
- 3.2.3 Maskable Interrupts
- 3.2.4 Non-Maskable Interrupt
- 3.2.5 Traps
- 3.2.6 Priority Among Exceptions

### 3.2.7 Exception Acknowledge Sequences: Detailed Flow

- 3.2.7.1 Maskable/Non-Maskable Interrupt Sequence
- 3.2.7.2 SLAVE/ILL/SVC/DVZ/FLG/BPT/UND Trap Sequence
- 3.2.7.3 Trace Trap Sequence
- 3.2.7.4 Integer-Overflow Trap Sequence
- 3.2.7.5 Debug Trap Sequence

### 3.3 Debugging Support

- 3.3.1 Instruction Tracing
- 3.3.2 Debug Trap Capability

### 3.4 On-Chip Caches

- 3.4.1 Instruction Cache (IC)
- 3.4.2 Data Cache (DC)
- 3.4.3 Cache Coherence Support

### 3.5 On-Chip Peripherals

#### 3.5.1 DMA Controller

- 3.5.1.1 Flyby (Direct) Transfers
- 3.5.1.2 Indirect (Memory-to-Memory) Transfers
- 3.5.1.3 Single Transfer Operation
- 3.5.1.4 Double Buffer Operation
- 3.5.1.5 Auto Initialize Operation
- 3.5.1.6 Bus Arbitration

#### 3.5.2 Interrupt Control Unit (ICU)

- 3.5.2.1 Interrupt-Acknowledge Processing
- 3.5.2.2 End-of-Interrupt Processing

#### 3.5.3 Timers

- 3.5.3.1 Processor Independent Mode (Mode 1)
- 3.5.3.2 External Event Counter Mode (Mode 2)
- 3.5.3.3 Input Capture Mode (Mode 3)

### 3.6 System Interface

- 3.6.1 Power and Grounding
- 3.6.2 Clocking
- 3.6.3 Resetting
- 3.6.4 Bus Cycles
  - 3.6.4.1 Bus Status
  - 3.6.4.2 Basic Read and Write Cycles
  - 3.6.4.3 Burst Cycles
  - 3.6.4.4 Cycle Extension
  - 3.6.4.5 Interlocked Bus Cycles
  - 3.6.4.6 Special Bus Cycles
  - 3.6.4.7 DMA Controller Bus Cycles
  - 3.6.4.8 Slave Processor Bus Cycles

## Table of Contents (Continued)

### 3.0 FUNCTIONAL DESCRIPTION (Continued)

- 3.6.5 Bus Retry
- 3.6.6 Dynamic Bus Configuration
  - 3.6.6.1 Instruction Fetch Sequences
  - 3.6.6.2 Data Read Sequences
  - 3.6.6.3 Data Write Sequences
- 3.6.7 Bus Access Control
- 3.6.8 Interfacing Memory-Mapped I/O Devices
- 3.6.9 Interrupt and Debug Trap Requests
- 3.6.10 Internal Status
- 3.6.11 Page-Mode and Static-Column DRAM Support

### 4.0 DEVICE SPECIFICATIONS

- 4.1 Pin Descriptions
  - 4.1.1 Supplies
  - 4.1.2 Input Signals
  - 4.1.3 Output Signals
  - 4.1.4 Input/Output Signals
- 4.2 Absolute Maximum Ratings
- 4.3 Electrical Characteristics
- 4.4 Switching Characteristics

### 4.0 DEVICE SPECIFICATIONS (Continued)

- 4.4.1 Definitions
- 4.4.2 Timing Tables
  - 4.4.2.1 Output Signals: Internal Propagation Delays
  - 4.4.2.2 Input Signal Requirements
- 4.4.3 Timing Diagrams

### APPENDIX A: INSTRUCTION FORMATS

#### B: COMPATIBILITY ISSUES

- B.1 Restrictions on Compatibility
- B.2 Architecture Extensions
- B.3 Integer-Overflow Trap
- B.4 Self-Modifying Code
- B.5 Memory-Mapped I/O

#### C: INSTRUCTION SET EXTENSIONS

- C.1 Processor Service Instructions
- C.2 Instruction Definitions

#### D: INSTRUCTION EXECUTION TIMES

- D.1 Internal Organization and Instruction Execution
- D.2 Basic Execution Times
  - D.2.1 Loader Timing
  - D.2.2 Address Unit Timing
  - D.2.3 Execution Unit Timing
- D.3 Instruction Dependencies
  - D.3.1 Data Dependencies
    - D.3.1.1 Register Interlocks
    - D.3.1.2 Memory Interlocks
  - D.3.2 Control Dependencies
- D.4 Storage Delays
  - D.4.1 Instruction Cache Misses
  - D.4.2 Data Cache Misses
  - D.4.3 Instruction and Operand Alignment
- D.5 Execution Time Calculations
  - D.5.1 Definitions
  - D.5.2 Notes on Table Use
  - D.5.3  $T_{eff}$  Evaluation
  - D.5.4 Instruction Timing Example
  - D.5.5 Execution Timing Tables
    - D.5.5.1 Basic and Memory Management Instructions
    - D.5.5.2 Floating-Point Instructions, CPU Portion

## List of Illustrations

CPU Block Diagram .....	1
NS32GX320 Internal Registers .....	2-1
Processor Status Register (PSR) .....	2-2
Configuration Register (CFG) .....	2-3
Debug Condition Register (DCR) .....	2-4
Debug Status Register (DSR) .....	2-5
DMA Controller Registers Address Map .....	2-6
DMA Status Register (STAT) .....	2-7
DMA Interrupt Mask Register (IMSK) .....	2-8
DMA Mode Control Register (MODE) .....	2-9
Channel Control Register (CNTL) .....	2-10
ICU Registers Address Map .....	2-11
Interrupt Vector Register (IVCT) .....	2-12
Timer Registers Address Map .....	2-13
Timer Control Register (TCNTL) .....	2-14
NS32GX320 Address Mapping .....	2-15
NS32GX320 Run-Time Environment .....	2-16
General Instruction Format .....	2-17
Index Byte Format .....	2-18
Displacement Encodings .....	2-19
Operating States .....	3-1
NS32GX320 Internal Instruction Pipeline .....	3-2
Memory References for Consecutive Instructions .....	3-3
Memory References after Serialization .....	3-4
Slave Instruction Protocol: CPU Actions .....	3-5
ID and Operation Word .....	3-6
Slave Processor Status Word .....	3-7
Interrupt Dispatch Table .....	3-8
Exception Acknowledge Sequence: Direct-Exception Mode Disabled .....	3-9
Exception Acknowledge Sequence: Direct-Exception Mode Enabled .....	3-10
Return From Trap (RETTn) Instruction Flow: Direct-Exception Mode Disabled .....	3-11
Return From Interrupt (RETI) Instruction Flow: Direct-Exception Mode Disabled .....	3-12
Exception Processing Flowchart .....	3-13
Service Sequence .....	3-14
Instruction Cache Structure .....	3-15
Data Cache Structure .....	3-16
Timer Block Diagram for the Processor Independent and External Event Counter Mode .....	3-17
Timer Block Diagram for the Input Capture Mode .....	3-18
Power and Ground Connections .....	3-19
Bus Clock Synchronization .....	3-20
Power-On Reset Requirements .....	3-21
General Reset Timing .....	3-22
Basic Read Cycle .....	3-23
Write Cycle .....	3-24
Burst Read cycles .....	3-25
Cycle Extension of a Basic Read Cycle .....	3-26
Type-2 Special Bus Cycle .....	3-27
Flyby DMA Transfer Cycles (Bus Initially Not Idle) .....	3-28
Indirect (Memory-to-Memory) DMA Transfer Cycles (Bus Initially Idle) .....	3-29

## List of Illustrations (Continued)

Successive DMA Transfers on Different Channels .....	3-30
Slave Processor Write Cycle .....	3-31
Slave Processor Read Cycle .....	3-32
Bus Retry During a Basic Read Cycle .....	3-33
Basic Interface for 32-Bit Memories .....	3-34
Basic Interface for 16-Bit Memories .....	3-35
Hold Acknowledge: (Bus Initially Idle) .....	3-36
Read Cycle from Memory-Mapped I/O Device .....	3-37
Write Cycle to Memory-Mapped I/O Device .....	3-38
Multi-Cycle Read from an 8-Bit Memory-Mapped I/O Device .....	3-39
Typical I/O Device Interface .....	3-40
Interrupt Request Encoding Logic .....	3-41
PAGE and PLAT Signals Timing .....	3-42
PAGE Signal Generation Logic .....	3-43
NS32GX320 Interface Signals .....	4-1
175-Pin Plastic PGA Package .....	4-2
Output Signals Specification Standard .....	4-3
Input Signals Specification Standard .....	4-4
Basic Read Cycle Timing .....	4-5
Write Cycle Timing .....	4-6
Interlocked Read and Write Cycles .....	4-7
Burst Read Cycles .....	4-8
External Termination of Burst Cycles .....	4-9
Bus Retry During Burst Cycles .....	4-10
Extended Retry Timing .....	4-11
Type-2 Special Bus Cycle .....	4-12
Flyby DMA Transfer Cycles (Bus Initially Not Idle) .....	4-13
HOLD Timing (Bus Initially Idle) .....	4-14
HOLD Acknowledge Timing (Bus Initially Not Idle) .....	4-15
Slave Processor Read Timing .....	4-16
Slave Processor Write Timing .....	4-17
Slave Processor Done .....	4-18
FSSR Signal Timing .....	4-19
IR0-3 and NMI Signals Sampling .....	4-20
Debug Trap Request .....	4-21
PFS Signal Timing .....	4-22
PAGE and PLAT Signals Timing .....	4-23
Break Point Signal Timing .....	4-24
TXAn and TXBn Input Timing .....	4-25
TXAn Output Timing .....	4-26
TXAn Float/Non-Float Timing .....	4-27
Cache Invalidation Timing .....	4-28
Clock Waveforms .....	4-29
Bus Clock Synchronization .....	4-30
Power-On Reset .....	4-31
Non-Power-On Reset .....	4-32
LPRI/SPRI Instruction Formats .....	C-1
CINV Instruction Format .....	C-2

## List of Illustrations (Continued)

MULWD Instruction Format .....	C-3
CMULD Instruction Format .....	C-4
CMACD Instruction Format .....	C-5
MACTD Instruction Format .....	C-6

## List of Tables

NS32GX320 Addressing Modes .....	2-1
NS32GX320 Instruction Set Summary .....	2-2
Floating-Point Instruction Protocol .....	3-1
Custom Slave Instruction Protocols .....	3-2
Summary of Exception Processing .....	3-3
Maximum Number of Back-to-Back DMA Transfers .....	3-4
Interrupt Sources and Priority Levels .....	3-5
Timer Modes .....	3-6
Interrupt Sequences .....	3-7
Cacheable/Non-Cacheable Instruction Fetches from a 32-Bit Bus .....	3-8
Cacheable/Non-Cacheable Instruction Fetches from a 16-Bit Bus .....	3-9
Cacheable/Non-Cacheable Instruction Fetches from an 8-Bit Bus .....	3-10
Cacheable/Non-Cacheable Data Reads from a 32-Bit Bus .....	3-11
Cacheable/Non-Cacheable Data Reads from a 16-Bit Bus .....	3-12
Cacheable/Non-Cacheable Data Reads from an 8-Bit Bus .....	3-13
Data Writes to a 32-Bit Bus .....	3-14
Data Writes to a 16-Bit Bus .....	3-15
Data Writes to an 8-Bit Bus .....	3-16
LPRI/SPRI New 'Short' Field Encodings .....	C-1
Additional Address Unit Processing Time for Complex Addressing Modes .....	D-1

## 1.0 Product Introduction

The NS32GX320 is an extremely sophisticated microprocessor in the Series 32000/EP family with a full 32-bit architecture and implementation optimized for high-performance embedded applications.

By employing a number of mainframe-like features, the device can deliver 15 MIPS peak performance with no wait states at a frequency of 30 MHz.

In addition to the very powerful CPU core, the NS32GX320 integrates on the same chip a number of peripherals, namely: a 2-channel DMA Controller, a 15-level Interrupt Control Unit (ICU) and three 16-bit timers.

The NS32GX320 is software compatible with the other CPUs of the Series 32000 architecture. The main features of the Series 32000 architecture and particularly the NS32GX320, are described briefly 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 architecture 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 NS32GX320 has 32-bit address pointers that can address up to 4 gigabytes without requiring any segmentation.

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

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

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

- High-level language support
- Easy future growth path
- Application flexibility

## 2.0 Architectural Description

### 2.1 REGISTER SET

The NS32GX320 has 54 internal registers. 21 of them belong to the CPU portion of the device and are addressed either implicitly by specific instructions or through the register addressing mode. The other 33 control the operation of the on-chip peripherals, and are memory mapped. *Figure 2-1* shows the NS32GX320 internal registers.

#### 2.1.1 General Purpose Registers

There are eight registers (R0–R7) used for satisfying the high speed general storage requirements, such as holding temporary variables and addresses. The general purpose registers are free for any use by the programmer. They are 32 bits in length. If a general purpose register is specified for an operand that is eight 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 CPU to implement specific address functions. A description of them 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.

The NS32GX320 also allows the SP1 register to be directly loaded and stored using privileged forms of the LPRi and SPRi instructions, regardless of the setting of the PSR S-bit. When SP1 is accessed in this manner, it is referred to as 'USP Register' or simply 'USP'.

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

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

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

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

## 2.0 Architectural Description (Continued)

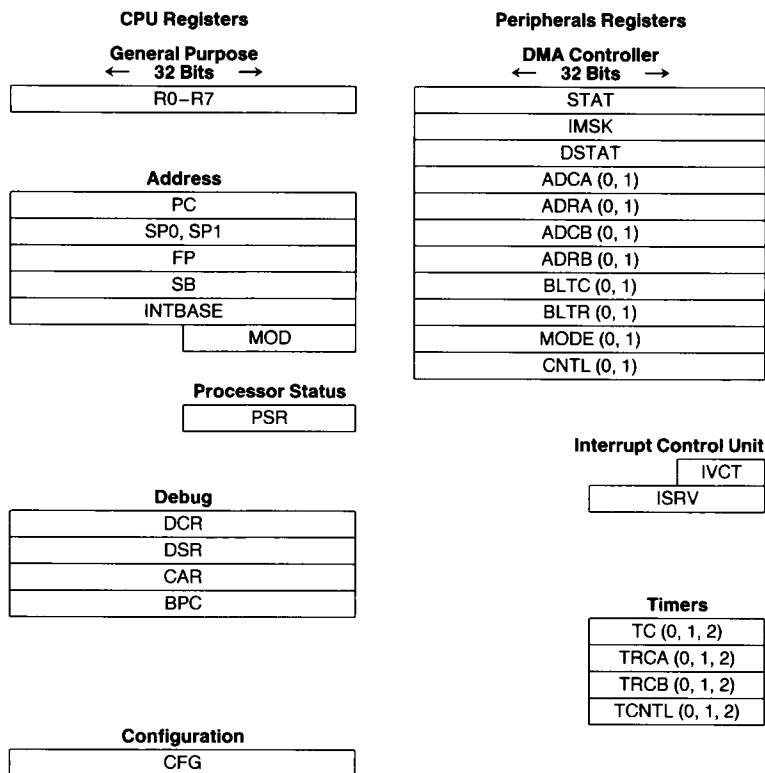


FIGURE 2-1. NS32GX320 Internal Registers

**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.

- 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.

- V** The V-bit enables generation of a trap (OVF) when an integer arithmetic operation overflows.

**Note:** The setting of the V-bit affects the execution times of the multiply and arithmetic shift instructions. Refer to Appendix D for details.

- 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".





## 2.0 Architectural Description (Continued)

### 2.1.5 Debug Registers

The NS32GX320 contains 4 registers dedicated for debug-ging functions.

These registers are accessed using privileged forms of the LPRI and SPRI instructions.

**DCR—Debug Condition Register.** The DCR Register enables detection of debug conditions. The format of the DCR is shown in *Figure 2-4*; the various bits are described below. A debug condition is enabled when the related bit is set to 1.

**CBE0** Compare Byte Enable 0; when set, BYTE0 of an aligned double-word is included in the address comparison

**CBE1** Compare Byte Enable 1; when set, BYTE1 of an aligned double-word is included in the address comparison

**CBE2** Compare Byte Enable 2; when set, BYTE2 of an aligned double-word is included in the address comparison

**CBE3** Compare Byte Enable 3; when set, BYTE3 of an aligned double-word is included in the address comparison

**CWR** Address-compare enable for write references

**CRD** Address-compare enable for read references

**CAE** Address-compare enable

**TR** Enable Trap (DBG) when a debug condition is detected

**PCE** PC-match enable

**UD** Enable debug conditions in User-Mode

**SD** Enable debug conditions in Supervisor Mode

**DEN** Enable debug conditions

The following 2 bits control testing features that can be used during initial system debugging. These features are

unique to the NS32GX320 implementation of the Series 32000 architecture; as such, they may not be supported in future implementations. For normal operation these 2 bits should be set to 0.

**SI** Single-Instruction mode enable. This bit, when set to 1, inhibits the overlapping of instruction's execution.

**BCP** Branch Condition Prediction disable. When this bit is 1, the branch prediction mechanism is disabled. See Section 3.1.3.1.

**DSR—Debug Status Register.** The DSR Register indicates debug conditions that have been detected. When the CPU detects an enabled debug condition, it sets the corresponding bit (BC, BEX, BCA) in the DSR to 1. When an address-compare condition is detected, then the RD-bit is loaded to indicate whether a read or write reference was performed. Software must clear all the bits in the DSR when appropriate. The format of the DSR is shown in *Figure 2-5*; the various fields are described below.

**RD** Indicates whether the last address-compare condition was for a read (RD = 1) or write (RD = 0) reference

**BPC** PC-match condition detected

**BEX** External condition detected

**BCA** Address-compare condition detected

**Note:** If an address compare is detected for a read and write for the same instruction, the RD bit will remain clear.

**CAR—Compare Address Register.** The CAR Register contains the address that is compared to operand reference addresses to detect an address-compare condition. The address must be double-word aligned; that is, the two least-significant bits must be 0. The CAR is 32 bits wide.

**BPC—Breakpoint Program Counter.** The BPC Register contains the address that is compared with the PC contents to detect a PC-match condition. The BPC Register is 32 bits wide.

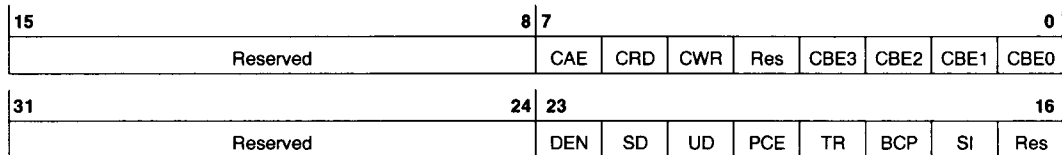


FIGURE 2-4. Debug Condition Register (DCR)

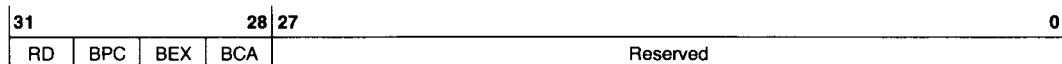


FIGURE 2-5. Debug Status Register (DSR)

## 2.0 Architectural Description (Continued)

### 2.1.6 DMA Controller Registers

The DMA Controller contains 19 32-bit memory mapped registers that are both readable and writable by software. Three of these registers are common to both channels and are used to report various conditions as well as selectively enable or disable interrupt generation corresponding to each condition. The other 16 registers are divided into two sets of eight, with each set associated to one channel.

All of the registers appear as memory locations, and must be accessed by specifying an operand length of 32 bits. Accesses specifying a length other than 32 bits may cause unpredictable results. The registers DSTAT, ADCA, ADCB, BLTC and MODE must not be written into while the associated channel is enabled. Upon reset, the registers STAT, IMSK, BLTC, BLTR, MODE and CNTL are cleared. Figure 2-6 shows the address map of the DMA registers.

**STAT—Status Register.** This register contains status information for the two DMA channels. Its format is shown in Figure 2-7.

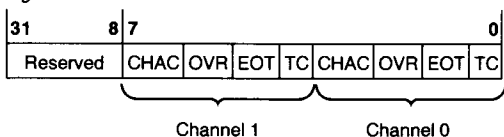
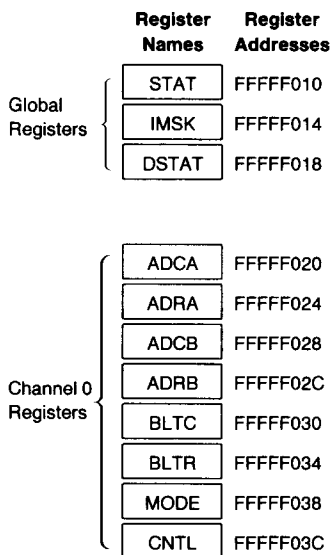


FIGURE 2-7. DMA Status Register (STAT)

**TC** Terminal Count; when set to 1, indicates that the transfer was completed by a terminal count condition (BLTC Register reached zero).



**EOT** External End Of Transfer. This bit is set to 1 when the transfer is externally terminated by the assertion of the EOT signal.

**OVR** Channel Overrun. Used only with double-buffered transfers. OVR is set to 1 when the present transfer is completed (BLTC = 0), but the parameters for the next transfer (addresses and block length) are not valid.

**CHAC** Channel Active. When set to 1, indicates that the channel is active (CHEN bit in register CNTL is 1 and BLTC > 0).

The TC, EOT and OVR bits are sticky. This means that, once set by the occurrence of the specific condition, they will remain set until explicitly cleared by software. These bits can be individually cleared by writing a value into the STAT register with the bit positions to be cleared set to 1. The CHAC bit continuously reflects the active or inactive status of the channel, and therefore, it is READ only.

**IMSK—Interrupt Mask.** This register is used to enable or disable interrupts for various conditions recorded in the STAT register. The format of IMSK is shown in Figure 2-8.

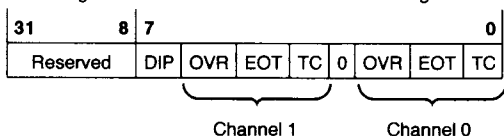


FIGURE 2-8. DMA Interrupt Mask Register (IMSK)

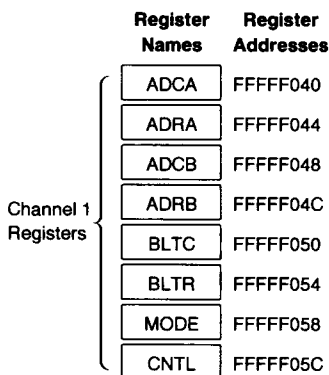


FIGURE 2-6. DMA Controller Registers Address Map

## 2.0 Architectural Description (Continued)

The DIP bit selects the DMA Interrupt Priority level in the Interrupt Control Unit. When a DIP is 0, the priority level is 6; when DIP is 1, the priority level is 14. Bits 0–2 and 4–6 are the Interrupt Mask Bits. An interrupt is enabled when the corresponding mask bit is set to 1.

**DSTAT—DMA Debug Status Register.** DSTAT is an image of the STAT register and provides an alternate means for accessing STAT. Writing a value into DSTAT stores the value itself into the STAT register, as opposed to only clearing bits as when writing directly into STAT. Setting a bit into DSTAT affects the associated interrupt as well as the CHEN bit in the CNTL register, according to the respective mask bit in IMSK.

**ADCA—Device A Address Counter.** In Indirect transfer mode, holds the current address of the source data item to be transferred. In Flyby mode, holds the current address of either the source data item or the destination location in the Addressed Device. If the ADA bit in the MODE register is set to 1, ADCA is incremented after each transfer cycle by the number of bytes transferred.

**ADRA—Device A Address Register.** In Indirect transfer mode, holds the starting address of the next source data block to be transferred. In Flyby mode, holds the starting address of either the source data block or the destination data area in the Addressed Device.

**ADCB—Device B Address Counter.** In Indirect transfer mode, holds the address of the destination location to receive the data item to be transferred. If the ADB bit in the MODE register is set to 1, ADCB is incremented after each transfer cycle by the number of bytes transferred. ADCB is not used in Flyby mode.

**ADRB—Device B Address Register.** In Indirect transfer mode, holds the starting address of the next destination data area to receive the block of data. ADRB is not used in Flyby mode.

**BLTC—Block Length Counter.** Holds the current number of bytes to be transferred. BLTC is decremented after each transfer cycle by the number of bytes transferred.

**BLTR—Block Length Register.** Holds the number of bytes in the next block to be transferred. Writing a zero value into BLTR while the VLD and CHEN bits in CNTL are both set to 1 may cause unpredictable results.

**Note 1:** The ADRA, ADRB and BLTR registers are used to store the transfer parameters (i.e., source address, destination address and block length) for the next data block to be transferred, for either auto-initialize or double-buffer modes of operation.

**Note 2:** The values programmed into ADCA, ADRA, ADCB, ADRB, BLTC and BLTR must be multiples of the smaller of the source and destination bus widths programmed in the mode register.

**MODE—Mode Control Register.** This register is used to specify the channel operating mode. The format of MODE is shown in Figure 2-9. Bit 1 and bits 15 through 31 should be set to 0.

31	15	14	10	9	8	7	6	5	4	3	2	1	0
Reserved			BLT	ADA	BWA	ADB	BWB	DIR	FBY	Res	OT		

FIGURE 2-9. DMA Mode Control Register (MODE)

**OT** Operation Type.

OT = 0 → Auto-Initialize Mode Disabled

OT = 1 → Auto-Initialize Mode Enabled

**FBY** Flyby/Indirect Transfers.

FBY = 0 → Flyby

FBY = 1 → Indirect (memory-to-memory)

**DIR** Flyby Transfer Direction. Specifies the direction of transfer between memory and peripheral device during flyby mode.

DIR = 0 → Peripheral Device is Destination

DIR = 1 → Peripheral Device is Source

**BWB** Device B Bus Width.

BWB = 00 → 8 Bits

BWB = 01 → 16 Bits

BWB = 10 → 32 Bits

BWB = 11 → Reserved

**ADB** Device B Address Control. Enables the incrementing of the device B address after each transfer cycle.

ADB = 0 → Address Unchanged

ADB = 1 → Address Incremented

**BWA** Device A Bus Width.

BWA = 00 → 8 Bits

BWA = 01 → 16 Bits

BWA = 10 → 32 Bits

BWA = 11 → Reserved

**ADA** Device A Address Control. Enables the incrementing of the device A address after each transfer cycle.

ADA = 0 → Address Unchanged

ADA = 1 → Address Incremented

**BLT** Block Transfer Length. This 5-bit field is used by the bus fairness mechanism, and specifies the maximum number of bytes that can be transferred before the DMA channel relinquishes the bus. Refer to Section 3.5.6 for details.

**CNTL—Channel Control Register.** CNTL is used to synchronize the channel operation with the programming of the block transfer parameters. The format of CNTL is shown in Figure 2-10.

31	2	1	0
Reserved		VLD	CHEN

FIGURE 2-10. Channel Control Register (CNTL)

## 2.0 Architectural Description (Continued)

**CHEN** Channel Enable.

CHEN = 0 → Channel Disabled

CHEN = 1 → Channel Enabled

**VLD** Transfer Parameters Valid. Specifies whether the transfer parameters for the next block to be transferred are valid.

VLD = 0 → Parameters Not Valid

VLD = 1 → Parameters Valid

The CHEN bit is set to 0 in the following cases.

- Upon Reset
- Software clears it by writing to the CNTL register
- The EOT bit in STAT is set to 1
- The OVR bit in STAT is set to 1 and is unmasked

In the last two cases the CHEN bit is forced to 0 and cannot be set to 1 by software unless EOT is cleared and OVR is either cleared or masked by clearing the corresponding bit in IMASK.

### 2.1.7 Interrupt Control Unit (ICU) Registers

The interrupt control unit contains two memory-mapped registers: IVCT and ISRV. These registers are 8 bits and 16 bits wide respectively, and must be accessed by specifying operand lengths equal to their widths. Specifying different operand lengths may cause unpredictable results. The address map for IVCT and ISRV is shown in *Figure 2-11*.

Register Names	Register Addresses
IVCT	FFFFFE00
ISRV	FFFFFE04

**FIGURE 2-11. ICU Registers Address Map**

**IVCT—Interrupt Vector Register.** This is a read-only register containing the current interrupt request vector to be used by the CPU in accessing the interrupt dispatch table when the request is acknowledged. Bits 0–3 contain an encoded value representing the interrupt request priority level. The binary value 1111 represents the highest priority level, while the value 0000 indicates that no interrupt request is pending. *Figure 2-12* shows the format of IVCT.

7							0
0	0	0	1	V	V	V	V

**FIGURE 2-12. Interrupt Vector Register (IVCT)**

**ISRV** Interrupt In-Service Register. ISRV is used by the ICU to keep track of the priority levels of interrupts currently being serviced.

A value of 1 in bit position  $i$  (where  $1 \leq i \leq 15$ ), indicates that the  $i$ -th priority level is currently in-service. Bit position 0 is always forced to 0. Upon reset the ISRV register is cleared to 0. Note that a zero value in the ISRV register indicates that there are no in-service interrupts.

### 2.1.8 Timers Registers

Each of the three timers in the NS32GX320 is controlled by a set of four registers. These registers are all 16 bits wide, and are memory-mapped. Their address map is shown in *Figure 2-13*. Accesses to the timer registers must specify operand lengths of 16 bits otherwise undefined results may be obtained. Refer to Section 3.5.3 for details on the timer's operation.

	Register Names	Register Addresses
Timer 0	Reserved	FFFFF800
	Reserved	FFFFF804
	Reserved	FFFFF808
	Reserved	FFFFF80C
	TC	FFFFF810
	TRCA	FFFFF814
	TRCB	FFFFF818
	TCNTL	FFFFF81C
Timer 1	TC	FFFFF820
	TRCA	FFFFF824
	TRCB	FFFFF828
	TCNTL	FFFFF82C
Timer 2	TC	FFFFF830
	TRCA	FFFFF834
	TRCB	FFFFF838
	TCNTL	FFFFF83C

**FIGURE 2-13. Timer Registers Address Map**

**TC—Counter.** TC is a down-counter which, upon underflow, is reloaded with the contents of either TRCA or TRCB for modes 1 and 2, and with all 1's for mode 3.

**TRCA, TRCB—Reload/Capture Registers A and B.** These registers hold either the counter reload values or a snapshot of TC.

## 2.0 Architectural Description (Continued)

**TCNTL—Timer Control Register.** Used to control the operation of each timer, and to enable timer interrupts. The TCNTL format is shown in Figure 2-14.

31	12	11	10	9	8	7	5	4				0
RES	WIS	WIA	WIB	PRC	TMC	TCS	IPFA	IENA	IPFB	IENB		

**FIGURE 2-14. Timer Control Register (TCNTL)**

- IENB** Interrupt Enable Bit B. When set to 1, enables the interrupt from IPFB.
- IPFB** Interrupt Pending Flag B.
- IENA** Interrupt Enable Bit A. When set to 1, enables the interrupt from IPFA. If Mode 3 is selected, it enables the interrupt from TCS as well.
- IPFA** Interrupt Pending Flag A.
- TCS** Timer Control and Status. In modes 1 or 2, this bit is used to start and stop the timer. The timer starts when TCS is 1. In mode 3, TCS is the underflow interrupt pending flag.
- TMC** Timer Mode Control. This three-bit field selects the timer mode of operation. (See Section 3.5.3.)
- PRC** Prescaler Control. Used only in modes 1 and 3. PRC controls the frequency of the timer input clock (TCLK). When PRC = 0, TCLK = BCLK/8; when PRC = 1, TCLK = BCLK/4096.
- WIB** Write Inhibit B. When WIB is set to 1, writing into IPFB is inhibited.
- WIA** Write Inhibit A. When WIA is set to 1, writing into IPFA is inhibited.
- WIS** Write Inhibit S. When WIS is set to 1, writing into TCS is inhibited.

### 2.2 MEMORY ORGANIZATION

The NS32GX320 implements full 32-bit addresses. This allows the device to access up to 4 Gbytes of memory. The memory is a uniform linear address space. Memory locations are numbered sequentially starting at zero and ending at  $2^{32} - 1$ . The number specifying a memory location is called an address. The contents of each memory location is a byte consisting of eight bits. Unless otherwise noted, diagrams in this document show data stored in memory with the lowest address on the right and the highest address on the left. Also, when data is shown vertically, the lowest address is at the top of a diagram and the highest address at the bottom of the diagram. When bits are numbered in a diagram, the least significant bit is given the number zero, and is shown at the right of the diagram. Bits are numbered in increasing significance and toward the left.

7		0
	A	

**Byte at Address A**

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

15		8	7		0
	A + 1			A	
	MSB			LSB	

#### Word at Address A

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

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

#### Double-Word at Address A

Although memory is addressed as bytes, it is actually organized as double-words. Note that access time to a word or a double-word depends upon its address, e.g. double-words that are aligned to start at addresses that are multiples of four will be accessed more quickly than those not so aligned. This also applies to words that cross a double-word boundary.

#### 2.2.1 Address Mapping

Figure 2-15 shows the NS32GX320 address mapping.

The NS32GX320 supports the use of memory-mapped peripheral devices and coprocessors. Such memory-mapped devices can be located at arbitrary locations in the address space except for the upper 8 Mbytes of memory (addresses between FF800000 (hex) and FFFFFFFF (hex), inclusive), which are reserved by National Semiconductor Corporation. Nevertheless, it is recommended that high-performance peripheral devices and coprocessors be located in a specific 8 Mbyte region of memory (addresses between FF000000 (hex) and FF7FFFFF (hex), inclusive), that is dedicated for memory-mapped I/O. This is because the NS32GX320 detects references to the dedicated locations and serializes reads and writes. See Section 3.1.3.3. When making I/O references to addresses outside the dedicated region, external hardware must indicate to the NS32GX320 that special handling is required. In this case a small performance degradation will also result. Refer to Section 3.1.3.2 for more information on memory-mapped I/O.

## 2.0 Architectural Description (Continued)

Address (Hex)

00000000

FF000000

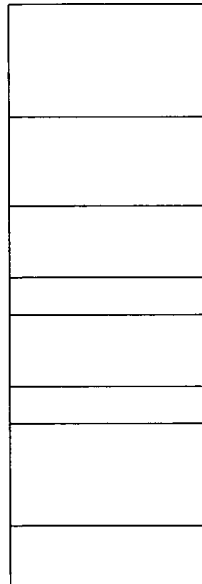
FF800000

FFFFF0XX

FFFFF8XX

FFFFFE00

FFFFFFF



Memory and I/O

Memory-Mapped I/O

Reserved

On-Chip DMA Controller

Reserved

On-Chip Timers

Reserved

On-Chip ICU and  
Interrupt Control

FIGURE 2-15. NS32GX320 Address Mapping

### 2.3 MODULAR SOFTWARE SUPPORT

The NS32GX320 provides special support for software modules and modular programs.

Each module in a NS32GX320 software environment consists of three components:

1. Program Code Segment.

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

2. Static Data Segment.

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

3. Link Table.

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

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

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

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

To handle the transfer of control from one module to another, the NS32GX320 uses a module table in memory and two registers in the CPU.

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

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

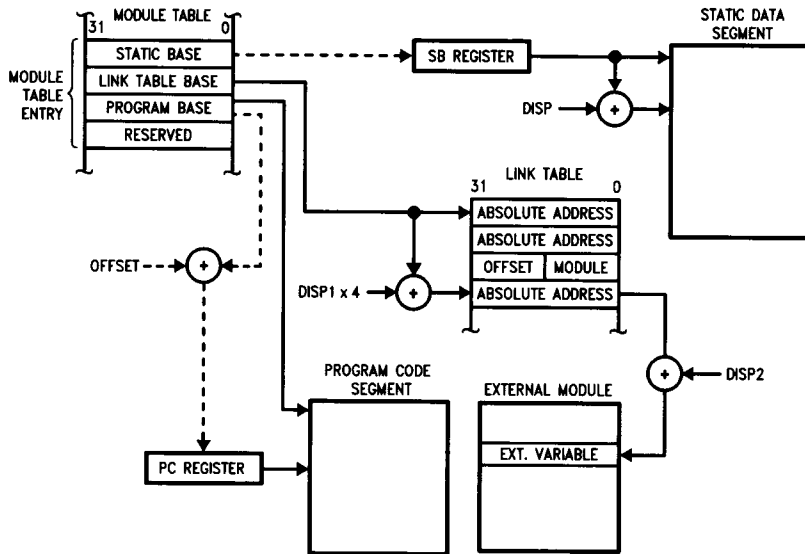
The MOD Register in the CPU contains the address of the Module Descriptor for the currently executing module.

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

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

In an NS32GX320 software environment modules need not be linked together prior to loading. As modules are loaded, a linking loader simply updates the Module Table and fills the Link Table entries with the appropriate values. No modification of a module's code is required. Thus, modules may be stored in read-only memory and may be added to a system independently of each other, without regard to their individual addressing. Figure 2-16 shows a typical NS32GX320 run-time environment.

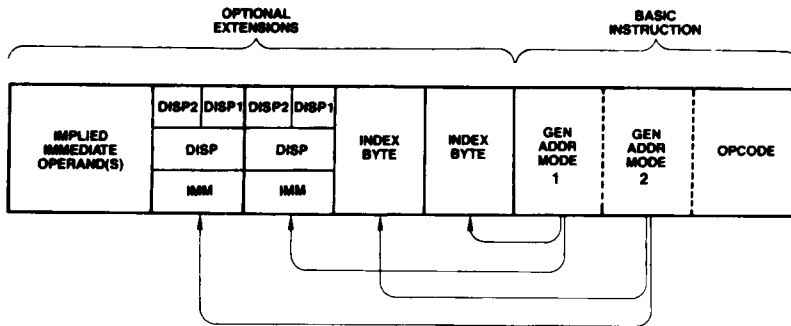
## 2.0 Architectural Description (Continued)



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

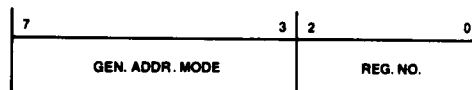
TL/EE/10564-2

**FIGURE 2-16. NS32GX320 Run-Time Environment**



**FIGURE 2-17. General Instruction Format**

TL/EE/10564-3



**FIGURE 2-18. Index Byte Format**

TL/EE/10564-4



## 2.0 Architectural Description (Continued)

### 2.4 INSTRUCTION SET

#### 2.4.1 General Instruction Format

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

Index Bytes appear when either or both Gen fields specify Scaled Index. In this case, the Gen field specifies only the Scale Factor (1, 2, 4 or 8), and the Index Byte specifies which General Purpose Register to use as the index, and which addressing mode calculation to perform before indexing. See Figure 2-18.

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

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

#### 2.4.2 Addressing Modes

The 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."

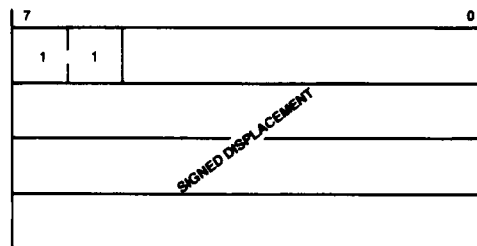
Byte Displacement: Range  $-64$  to  $+63$



Word Displacement: Range  $-8192$  to  $+8191$



Double Word Displacement:  
Range  $-(2^{29} - 2^{24})$  to  $+(2^{29} - 1)^*$



TL/EE/10564-5

**FIGURE 2-19. Displacement Encodings**

\*Note: The pattern "11100000" for the most significant byte of the displacement is reserved by National for future enhancements. Therefore, it should never be used by the user program. This causes the lower limit of the displacement range to be  $-(2^{29} - 2^{24})$  instead of  $-2^{29}$ .

## 2.0 Architectural Description (Continued)

Addressing modes 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.

Addressing Modes fall into nine basic types:

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

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

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

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

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

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

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

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

**Scaled Index:** Although encoded as an addressing mode, Scaled Indexing is an option on any addressing mode except Immediate or another Scaled Index. It has the effect of

calculating an Effective Address, then multiplying any General Purpose Register by 1, 2, 4 or 8 and adding it 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 Instruction Set Reference Manual.

### 2.4.3 Instruction Set Summary

Table 2-2 presents a brief description of the NS32GX320 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 Instruction Set Reference Manual.

#### 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: Address, Debug, Status, Configuration.

creg = A Custom Slave Processor Register (Implementation Dependent).

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

## 2.0 Architectural Description (Continued)

TABLE 2-1. NS32GX320 Addressing Modes

ENCODING	MODE	ASSEMBLER SYNTAX	EFFECTIVE ADDRESS
<b>Register</b>			
00000	Register 0	R0, F0, L0	None: Operand is in the specified register.
00001	Register 1	R1, F1, L1	
00010	Register 2	R2, F2, L2	
00011	Register 3	R3, F3, L3	
00100	Register 4	R4, F4, L4	
00101	Register 5	R5, F5, L5	
00110	Register 6	R6, F6, L6	
00111	Register 7	R7, F7, L7	
<b>Register Relative</b>			
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)	
<b>Memory Relative</b>			
10000	Frame memory relative	disp2(disp1(FP))	Disp2 + Pointer; Pointer found at address Disp1 + 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))	
<b>Reserved</b>			
10011	(Reserved for Future Use)		
<b>Immediate</b>			
10100	Immediate	value	None. Operand is input from instruction queue.
<b>Absolute</b>			
10101	Absolute	@disp	Disp.
<b>External</b>			
10110	External	EXT(disp1) + disp2	Disp2 + Pointer; Pointer is found at Link Table Entry number Disp1.
<b>Top of Stack</b>			
10111	Top of stack	TOS	Top of current stack, using either User or Interrupt Stack Pointer, as selected in PSR. Automatic Push/Pop included.
<b>Memory Space</b>			
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	
<b>Scaled Index</b>			
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)

TABLE 2-2. NS32GX320 Instruction Set Summary

### MOVES

Format	Operation	Operands	Description
4	MOV <sub>i</sub>	gen,gen	Move a value.
2	MOVQ <sub>i</sub>	short,gen	Extend and move a signed 4-bit constant.
7	MOV <sub>Mi</sub>	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	ADD <sub>i</sub>	gen,gen	Add.
2	ADDQ <sub>i</sub>	short,gen	Add signed 4-bit constant.
4	ADDC <sub>i</sub>	gen,gen	Add with carry.
4	SUB <sub>i</sub>	gen,gen	Subtract.
4	SUBC <sub>i</sub>	gen,gen	Subtract with carry (borrow).
6	NEG <sub>i</sub>	gen,gen	Negate (2's complement).
6	ABS <sub>i</sub>	gen,gen	Take absolute value.
7	MUL <sub>i</sub>	gen,gen	Multiply.
7	QUO <sub>i</sub>	gen,gen	Divide, rounding toward zero.
7	REMi	gen,gen	Remainder from QUO.
7	DIV <sub>i</sub>	gen,gen	Divide, rounding down.
7	MOD <sub>i</sub>	gen,gen	Remainder from DIV (Modulus).
7	MEL <sub>i</sub>	gen,gen	Multiply to Extended Integer.
7	DEL <sub>i</sub>	gen,gen	Divide Extended Integer.
18	MULWD	gen,gen	Multiply Word to Double.
18	MACTD	gen,gen	Multiply and Accumulate Twice Double.

### COMPLEX ARITHMETIC

Format	Operation	Operands	Description
18	CMULD	gen,gen	Complex Multiply Double.
18	CMACD	gen,gen	Complex Multiply and Accumulate Double.

### PACKED DECIMAL (BCD) ARITHMETIC

Format	Operation	Operands	Description
6	ADDP <sub>i</sub>	gen,gen	Add Packed.
6	SUBP <sub>i</sub>	gen,gen	Subtract Packed.

### INTEGER COMPARISON

Format	Operation	Operands	Description
4	CMPI	gen,gen	Compare.
2	CMPIQ <sub>i</sub>	short,gen	Compare to signed 4-bit constant.
7	CMPM <sub>i</sub>	gen,gen,disp	Compare Multiple: disp bytes (1 to 16).

### LOGICAL AND BOOLEAN

Format	Operation	Operands	Description
4	AND <sub>i</sub>	gen,gen	Logical AND.
4	OR <sub>i</sub>	gen,gen	Logical OR.
4	BIC <sub>i</sub>	gen,gen	Clear selected bits.
4	XOR <sub>i</sub>	gen,gen	Logical Exclusive OR.
6	COM <sub>i</sub>	gen,gen	Complement all bits.
6	NOT <sub>i</sub>	gen,gen	Boolean complement: LSB only.
2	Scond <sub>i</sub>	gen	Save condition code (cond) as a Boolean variable of size i.

### SHIFTS

Format	Operation	Operands	Description
6	LSH <sub>i</sub>	gen,gen	Logical Shift, left or right.
6	ASH <sub>i</sub>	gen,gen	Arithmetic Shift, left or right.
6	ROT <sub>i</sub>	gen,gen	Rotate, left or right.

## 2.0 Architectural Description (Continued)

TABLE 2-2. NS32GX320 Instruction Set Summary (Continued)

### 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.

### BIT FIELDS

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

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

### ARRAYS

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

### STRINGS

String instructions assign specific functions to the General Purpose Registers:

R4 - Comparison Value

R3 - Translation Table Pointer

R2 - String 2 Pointer

R1 - String 1 Pointer

R0 - Limit Count

Options on all string instructions are:

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

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

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

All string instructions end when R0 decrements to zero.

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.

## 2.0 Architectural Description (Continued)

TABLE 2-2. NS32GX320 Instruction Set Summary (Continued)

### 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 Processor Register. (Privileged if PSR, INTBASE, USP, CFG or Debug Registers).
2	SPRI	areg,gen	Store Processor Register. (Privileged if PSR, INTBASE, USP, CFG or Debug Registers).
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)

### 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.
12	POLYf	gen,gen	Polynomial Step.
12	DOTf	gen,gen	Dot Product.
12	SCALBf	gen,gen	Binary Scale.
12	LOGBf	gen,gen	Binary Log.
9	LFSR	gen	Load FSR.
9	SFSR	gen	Store FSR.

## 2.0 Architectural Description (Continued)

TABLE 2-2. NS32GX320 Instruction Set Summary (Continued)

### MISCELLANEOUS

Format	Operation	Operands	Description
1	NOP		No Operation.
1	WAIT		Wait for interrupt.
1	DIA		Diagnose. Single-byte "Branch to Self" for hardware breakpointing. Not for use in programming.
14	CINV	[options],gen	Cache Invalidate. (Privileged)

### CUSTOM SLAVE

Format	Operation	Operands	Description
15.5	CCAL0c	gen,gen	Custom Calculate.
15.5	CCAL1c	gen,gen	
15.5	CCAL2c	gen,gen	
15.5	CCAL3c	gen,gen	Custom Move.
15.5	CMOV0c	gen,gen	
15.5	CMOV1c	gen,gen	
15.5	CMOV2c	gen,gen	Custom Compare.
15.5	CMOV3c	gen,gen	
15.5	CCMP0c	gen,gen	
15.5	CCMP1c	gen,gen	Custom Convert.
15.1	CCV0ci	gen,gen	
15.1	CCV1ci	gen,gen	
15.1	CCV2ci	gen,gen	
15.1	CCV3ic	gen,gen	
15.1	CCV4DQ	gen,gen	
15.1	CCV5QD	gen,gen	Load Custom Status Register.
15.1	LCSR	gen	
15.1	SCSR	gen	
15.0	LCR	creg,gen	Load Custom Register. (Privileged)
15.0	SCR	creg,gen	Store Custom Register. (Privileged)

## 3.0 Functional Description

This chapter provides details on the functional characteristics of the NS32GX320 microprocessor.

The chapter is divided into six main sections:

Instruction Execution, Exception Processing, Debugging, On-Chip Caches, On-Chip Peripherals and System Interface.

### 3.1 INSTRUCTION EXECUTION

To execute an instruction, the NS32GX320 performs the following operations:

- Fetch the instruction
- Read source operands, if any (1)
- Calculate results
- Write result operands, if any
- Modify flags, if necessary
- Update the program counter

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

Details on the effects of the internal pipelining, as well as the occurrence of exceptions on the instruction execution, are provided in the following sections.

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

#### 3.1.1 Operating States

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

Whenever the  $\overline{RST}$  signal is asserted, the CPU enters the reset state. The CPU remains in the reset state until the  $\overline{RST}$  signal is driven inactive, at which time it enters the Executing-Instructions state. In the Reset state the contents of certain registers are initialized. Refer to Section 3.6.3 for details.

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

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

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

In the Waiting-For-An-Interrupt state, the CPU is idle. A special status identifying this state is presented on the system interface (Section 3.6). When an interrupt or a debug condition is detected, the CPU enters the Processing-An-Exception state.

**Note:** When the Direct-Exception mode is enabled, the CPU does not save the MOD Register contents nor does it read the module linkage information for the exception service procedure. Refer to Section 3.2 for details.

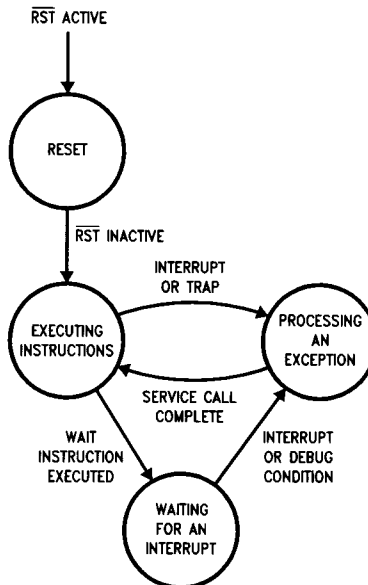


FIGURE 3-1. Operating States

TL/EE/10564-6

#### 3.1.2 Instruction Endings

The NS32GX320 checks for exceptions at various points while executing instructions. Certain exceptions, like interrupts, are in most cases recognized between instructions. Other exceptions, like Divide-By-Zero Trap, are recognized during execution of an instruction. When an exception is recognized during execution of an instruction, the instruction ends in one of four possible ways: completed, suspended, terminated, or partially completed. Each type of exception causes a particular ending, as specified in Section 3.2.

##### 3.1.2.1 Completed Instructions

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



## 3.0 Functional Description (Continued)

### 3.1.2.2 Suspended Instructions

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

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

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

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

### 3.1.2.3 Terminated Instructions

An instruction being executed is terminated when reset occurs. Any result operands and flags due to be affected by the instruction are undefined, as is the contents of the PC. The result operands of other instructions executed since the last serializing operation may not have been written to memory. A terminated instruction cannot be completed.

### 3.1.2.4 Partially Completed Instructions

When an interrupt or debug condition is recognized during execution of a string instruction, the instruction is said to be

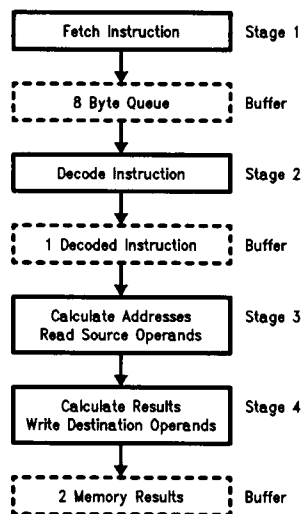
partially completed. A partially completed instruction has not completed, but all other instructions executed since the last exception occurred have been completed. Result operands and flags due to be affected by the instruction may have been modified, but the values stored in the string pointers and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

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

### 3.1.3 Instruction Pipeline

The NS32GX320 executes instructions in a heavily pipelined fashion. This allows a significant performance enhancement since the operations of several instructions are performed simultaneously rather than in a strictly sequential manner.

The CPU provides a four-stage internal instruction pipeline. As shown in Figure 3-2, a write buffer, that can hold up to two operands, is also provided to allow write operations to be performed off-line.



TL/EE/10564-7

**FIGURE 3-2. NS32GX320 Internal Instruction Pipeline**

Due to the pipelining, operations like fetching one instruction, reading the source operands of a second instruction, calculating the results of a third instruction and storing the results of a fourth instruction, can all occur in parallel.

### 3.0 Functional Description (Continued)

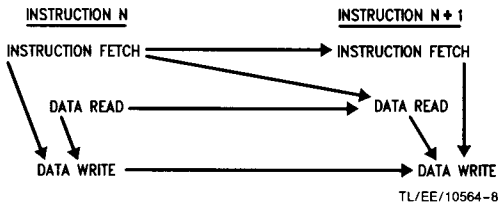
The order of memory references performed by the CPU may also differ from that related to a strictly sequential instruction execution. In fact, when an instruction is being executed, some of the source operands may be read from memory before the instruction is completely fetched. For example, the CPU may read the first source operand for an instruction before it has fetched a displacement used in calculating the address of the second source operand. The CPU, however, always completes fetching an instruction and reading its source operands before writing its results. When more than one source operand must be read from memory to execute an instruction, the operands may be read in any order. Similarly, when more than one result operand is written to memory to execute an instruction, the operands may be written in any order.

An instruction is fetched only after all previous instructions have been completely fetched. However, the CPU may begin fetching an instruction before all of the source operands have been read and results written for previous instructions.

The source operands for an instruction are read only after all previous instructions have been fetched and their source operands read. A source operand for an instruction may be read before all results of previous instructions have been written, except when the source operand's value depends on a result not yet written. The CPU compares the address and length of a source operand with those of any results not yet written, and delays reading the source operand until after writing all results on which the source operand depends. Also, the CPU ensures that the interlocked read and write references to execute an SBITII or CBITII instruction occur after writing all results of previous instructions and before reading any source operands for subsequent instructions.

The result operands for an instruction are written after all results of previous instructions have been written.

The description above is summarized in Figure 3-3, which shows the precedence of memory references for two consecutive instructions.



**FIGURE 3-3. Memory References for Consecutive Instructions**

(An arrow from one reference to another indicates that the first reference always precedes the second.)

Another consequence of overlapping the operations for several instructions, is that the CPU may fetch an instruction and read its source operands, even though the instruction is not executed (e.g., due to the occurrence of an exception).

Special care is needed in the handling of memory-mapped I/O devices. The CPU provides special mechanisms to ensure that the references to these devices are always performed in the order implied by the program. Refer to Section 3.1.3.2 for details.

It is also to be noted that the CPU does not check for dependencies between the fetching of an instruction and the writing of previous instructions' results. Therefore, special care is required when executing self-modifying code.

#### 3.1.3.1 Branch Prediction

One problem inherent to all pipelined machines is what is called "Pipeline Breakage".

This occurs every time the sequentiality of the instructions is broken, due to the execution of certain instructions or the occurrence of exceptions.

The result of a pipeline breakage is a performance degradation, due to the fact that a certain portion of the pipeline must be flushed and new data must be brought in.

The NS32GX320 provides a special mechanism, called branch prediction, that helps minimize this performance penalty.

When a conditional branch instruction is decoded in the early stages of the pipeline, a prediction on the execution of the instruction is performed.

More precisely, the prediction mechanism predicts backward branches as taken and forward branches as not taken, except for the branch instructions BLE and BNE that are always predicted as taken.

Thus, the resulting probability of correct prediction is fairly high, especially for branch instructions placed at the end of loops.

The sequence of operations performed by the loader and execution units in the CPU is given below:

- Loader detects branches and calculates destination addresses
- Loader uses branch opcode and direction to select between sequential and non-sequential streams
- Loader saves address for alternate stream
- Execution unit resolves branch decision

Due to the branch prediction, some special care is required when writing self-modifying code. Refer to the appropriate section in Appendix B for more information on this subject.

#### 3.1.3.2 Memory-Mapped I/O

The characteristics of certain peripheral devices and the overlapping of instruction execution in the pipeline of the NS32GX320 require that special handling be applied to memory-mapped I/O references. I/O references differ from memory references in two significant ways, imposing the following requirements:

1. Reading from a peripheral port can alter the value read on the next reference to the same port or another port in the same device. (A characteristic called here "destructive-reading".) Serial communication controllers and FIFO buffers commonly operate in this manner. As explained in "Instruction Pipeline" above, the NS32GX320 can read the source operands for one instruction while the previous instruction is executing. Because the previous instruction may cause a trap, an interrupt may be recognized, or the flow of control may be otherwise altered, it is a requirement that destructive-reading of source operands before the execution of an instruction be avoided.

## 3.0 Functional Description (Continued)

2. Writing to a peripheral port can alter the value read from another port of the same device. (A characteristic called here "side-effects of writing"). For example, before reading the counter's value from the NS32202 Interrupt Control Unit it is first necessary to freeze the value by writing to another control register.

However, as mentioned above, the NS32GX320 can read the source operands for one instruction before writing the results of previous instructions unless the addresses indicate a dependency between the read and write references. Consequently, it is a requirement that read and write references to peripheral that exhibit side-effects of writing must occur in the order dictated by the instructions.

The NS32GX320 supports 2 methods for handling memory-mapped I/O. The first method is more general; it satisfies both requirements listed above and places no restriction on the location of memory-mapped peripheral devices. The second method satisfies only the requirement for side effects of writing, and it restricts the location of memory-mapped I/O devices, but it is more efficient for devices that do not have destructive-read ports.

The first method for handling memory-mapped I/O uses two signals:  $\overline{\text{IOINH}}$  and  $\overline{\text{IODEC}}$ . When the NS32GX320 generates a read bus cycle, it asserts the output signal  $\overline{\text{IOINH}}$  if either of the I/O requirements listed above is not satisfied. That is,  $\overline{\text{IOINH}}$  is asserted during a read bus cycle when (1) the read reference is for an instruction that may not be executed or (2) the read reference occurs while a write reference is pending for a previous instruction. When the read reference is to a peripheral device that implements ports with destructive-reading or side-effects of writing, the input signal  $\overline{\text{IODEC}}$  must be asserted; in addition, the device must not be selected if  $\overline{\text{IOINH}}$  is active. When the CPU detects that the  $\overline{\text{IODEC}}$  input signal is active while the  $\overline{\text{IOINH}}$  output signal is also active, it discards the data read during the bus cycle and serializes instruction execution. See the next section for details on serializing operations. The CPU then generates the read bus cycle again, this time satisfying the requirements for I/O and driving  $\overline{\text{IOINH}}$  inactive.

The second method for handling memory-mapped I/O uses a dedicated region of memory. The NS32GX320 treats all references to the memory range from address FF000000 to address FFFFFFFF inclusive in a special manner.

While a write to a location in this range is pending, reads from locations in the same range are delayed. However, reads from locations with addresses lower than FF000000 may occur. Similarly, reads from locations in the above range may occur while writes to locations outside of the range are pending.

It is to be noted that the CPU may assert  $\overline{\text{IOINH}}$  even when the reference is within the dedicated region. Refer to Section 3.6.8 for more information on the handling of I/O devices.

### 3.1.3.3 Serializing Operations

After executing certain instructions or processing an exception, the CPU serializes instruction execution. Serializing in-

struction execution means that the CPU completes writing all previous instructions' results to memory, then begins fetching and executing the next instruction.

For example, when a new value is loaded into the PSR by executing an LPRW instruction, the pipeline is flushed and a serializing operation takes place. This is necessary since the privilege level might have changed and the instructions following the LPRW instruction must be fetched again with the new privilege level.

The CPU serializes instruction execution after executing one of the following instructions: BICPSRW, BISPSRW, BPT, CINV, DIA, FLAG (trap taken), LPR (CFG, INTBASE, PSR, UPSR, DCR, BPC, DSR, and CAR only), RETT, RETI, and SVC. Figure 3-4 shows the memory references after serialization.

**Note 1:** LPRB UPSRW can be executed in User Mode to serialize instruction execution.

**Note 2:** After an instruction that writes a result to memory is executed, the updating of the result's memory location may be delayed until the next serializing operation.

**Note 3:** When reset occurs, the CPU discards any results that have not yet been written to memory.

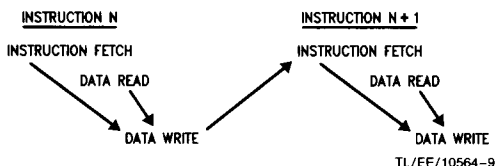


FIGURE 3-4. Memory References after Serialization

### 3.1.4 Slave Processor Instructions

The NS32GX320 recognizes two groups of instructions being executable by external slave processors:

- Floating Point Instructions
- Custom Slave Instructions

Each Slave Instruction Set is enabled by a bit in the Configuration Register (Section 2.1.4). Any Slave Instruction which does not have its corresponding Configuration Register bit set will trap as undefined, without any Slave Processor communication attempted by the CPU. This allows software simulation of a non-existent Slave Processor.

#### 3.1.4.1 Slave Instruction 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 3-5. While applying Status code 1111 (Broadcast ID Section 3.6.4.1), the CPU transfers the ID Byte on bits D24-D31, the operation

### 3.0 Functional Description (Continued)

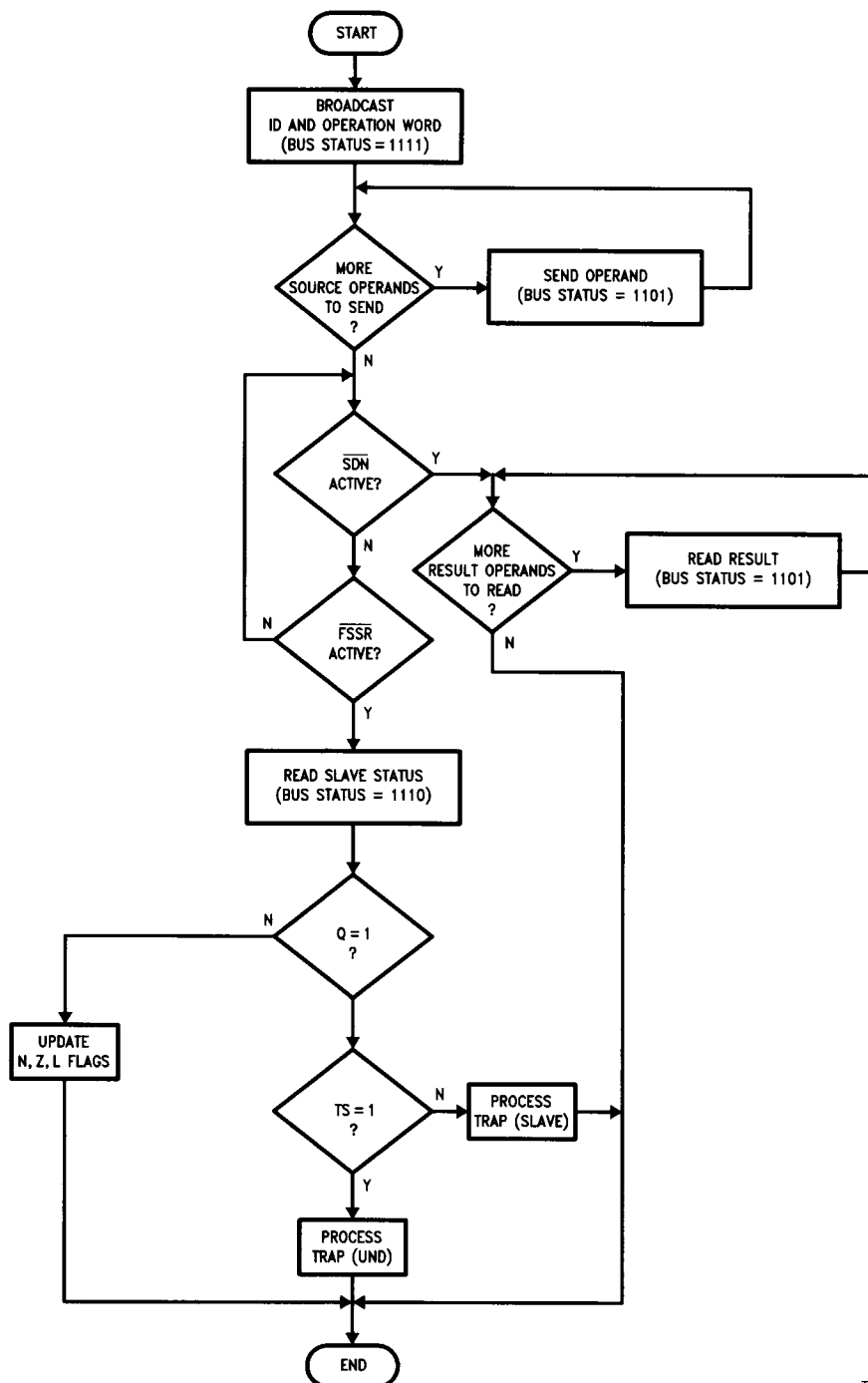


FIGURE 3-5. Slave Instruction Protocol: CPU Actions

TL/EE/10564-10

### 3.0 Functional Description (Continued)

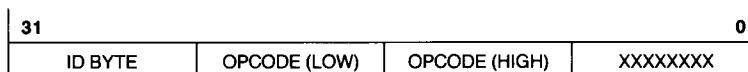


FIGURE 3-6. ID and Operation Word

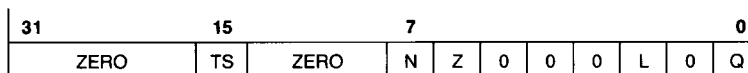


FIGURE 3-7. Slave Processor Status Word

word on bits D8–D23 in a swapped order of bytes and a non-used byte XXXXXXXX (X = don't care) on bits D0–D7 (*Figure 3-6*).

All slave processors observe the bus cycle and inspect the identification code. The slave selected by the identification code continues with the protocol; other slaves wait for the next slave instruction to be broadcast.

After transferring the slave instruction, the CPU sends to the slave any source operands that are located in memory or the General-Purpose registers. The CPU then waits for the slave to assert  $\overline{SDN}$  or  $\overline{FSSR}$ . While the CPU is waiting, it can perform bus cycles to fetch instructions and read source operands for instructions that follow the slave instruction being executed. If there are no bus cycles to perform, the CPU is idle with a special Status indicating that it is waiting for a slave processor. After the slave asserts  $\overline{SDN}$  or  $\overline{FSSR}$ , the CPU follows one of the two sequences described below.

If the slave asserts  $\overline{SDN}$ , then the CPU checks whether the instruction stores any results to memory or the General-Purpose registers. The CPU reads any such results from the slave by means of 1 or 2 bus cycles and updates the destination.

If the slave asserts  $\overline{FSSR}$ , then the NS32GX320 reads a 32-bit status word from the slave. The CPU checks bit 0 in the slave's status word to determine whether to update the PSR flags or to process an exception. *Figure 3-7* shows the format of the slave's status word.

If the Q bit in the status word is 0, the CPU updates the N, Z and L flags in the PSR.

If the Q bit in the status word is set to 1, the CPU processes either a Trap (UND) if TS is 1 or a Trap (SLAVE) if TS is 0.

**Note 1:** Only the floating-point and custom compare instructions are allowed to return a value of 0 for the Q bit when the  $\overline{FSSR}$  signal is activated. All other instructions must always set the Q bit to 1 (to signal a Trap), when activating  $\overline{FSSR}$ .

**Note 2:** While executing CINV instruction, the CPU displays the operation code and source operand using slave processor write bus cycles, as described in the protocol above. Nevertheless, the CPU does not wait for  $\overline{SDN}$  or  $\overline{FSSR}$  to be asserted while executing these instructions. This information can be used to monitor the contents of the on-chip Instruction Cache, and Data Cache.

**Note 3:** The slave processor must be ready to accept new slave instruction at any time, even while the slave is executing another instruction or waiting for the CPU to read results.

## 3.0 Functional Description (Continued)

### 3.1.4.2 Floating Point Instructions

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

The Operand class columns give the Access Class for each general operand, defining how the addressing modes are interpreted (see 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 3-7*).

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

### 3.1.4.3 Custom Slave Instructions

Provided in the NS32GX320 is the capability of communicating with a user-defined, "Custom" Slave Processor. The instruction set provided for a Custom Slave Processor defines the instruction formats, the operand classes and the communication protocol. Left to the user are the interpretations of the Op Code fields, the programming model of the Custom Slave and the actual types of data transferred. The protocol specifies only the size of an operand, not its data type.

Table 3-2 lists the relevant information for the Custom Slave instruction set. The designation "c" is used to represent an operand which can be a 32-bit ("D") or 64-bit ("Q") quantity

in any format; the size is determined by the suffix on the mnemonic. Similarly, an "i" indicates an integer size (Byte, Word, Double Word) selected by the corresponding mnemonic suffix.

Any operand indicated as being of type "c" will not cause a transfer if the register addressing mode is specified. It is assumed in this case that the slave processor is already holding the operand internally.

For the instruction encodings, see Appendix A.

## 3.2 EXCEPTION PROCESSING

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

An interrupt occurs in response to an event generated either internally, by the on-chip DMA Channels or Timers, or externally, by activating NMI or one or more of the  $\overline{IR}0-3$  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 optionally the MOD register contents on the interrupt stack and then it transfers control to an exception service procedure.

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

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

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

Refer to Section 3.6.3 for details on the reset operation.

## 3.0 Functional Description (Continued)

**TABLE 3-1. Floating Point Instruction Protocols**

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

**TABLE 3-2. Custom Slave Instruction Protocols**

Mnemonic	Operand 1 Class	Operand 2 Class	Operand 1 Issued	Operand 2 Issued	Returned Value Type and Dest.	PSR Bits Affected
CCAL0c	read.c	rmw.c	c	c	c to Op.2	none
CCAL1c	read.c	rmw.c	c	c	c to Op.2	none
CCAL2c	read.c	rmw.c	c	c	c to Op.2	none
CCAL3c	read.c	rmw.c	c	c	c to Op.2	none
CMOV0c	read.c	write.c	c	N/A	c to Op.2	none
CMOV1c	read.c	write.c	c	N/A	c to Op.2	none
CMOV2c	read.c	write.c	c	N/A	c to Op.2	none
CMOV3c	read.c	write.c	c	N/A	c to Op.2	none
CCMP0c	read.c	read.c	c	c	N/A	N,Z,L
CCMP1c	read.c	read.c	c	c	N/A	N,Z,L
CCV0ci	read.c	write.i	c	N/A	i to Op.2	none
CCV1ci	read.c	write.i	c	N/A	i to Op.2	none
CCV2ci	read.c	write.i	c	N/A	i to Op.2	none
CCV3ic	read.i	write.c	i	N/A	c to Op.2	none
CCV4DQ	read.D	write.Q	D	N/A	Q to Op.2	none
CCV5QD	read.Q	write.D	Q	N/A	D to Op.2	none
LCSR	read.D	N/A	D	N/A	N/A	none
SCSR	N/A	write.D	N/A	N/A	D to Op.2	none
LCR*	read.D	N/A	D	N/A	N/A	none
SCR*	write.D	N/A	N/A	N/A	D to Op.1	none

**Note:**

D = Double Word

i = Integer size (B,W,D) specified in mnemonic.

c = Custom size (D:32 bits or Q:64 bits) specified in mnemonic.

\* = Privileged instruction: will trap if CPU is in User Mode.

N/A = Not Applicable to this instruction.

## 3.0 Functional Description (Continued)

### 3.2.1 Exception Acknowledge Sequence

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

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

#### Direct-Exception Mode Disabled

The Direct-Exception mode is disabled while the DE bit in the CFG register is 0 (Section 2.1.4). In this case the CPU first pushes the saved PSR copy along with the contents of the MOD and PC registers on the interrupt stack. Then it

reads the double-word entry from the Interrupt Dispatch table at address 'INTBASE + vector × 4'. See Figures 3-8 and 3-9. The CPU uses this entry to call the exception service procedure, interpreting the entry as an external procedure descriptor.

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

#### Direct-Exception Mode Enabled

The Direct-Exception mode is enabled when the DE bit in the CFG register is set to 1. In this case the CPU first pushes the saved PSR copy along with the contents of the PC register on the Interrupt Stack. The word stored on the Interrupt Stack between the saved PSR and PC register is reserved for future use; its contents are undefined. The CPU then reads the double-word entry from the Interrupt Dispatch Table at address 'INTBASE + vector × 4'. The CPU uses this entry to call the exception service procedure, interpreting the entry as an absolute address that is simply loaded into the PC register. Figure 3-10 provides a pictorial of the acknowledge sequence. It is to be noted that while the

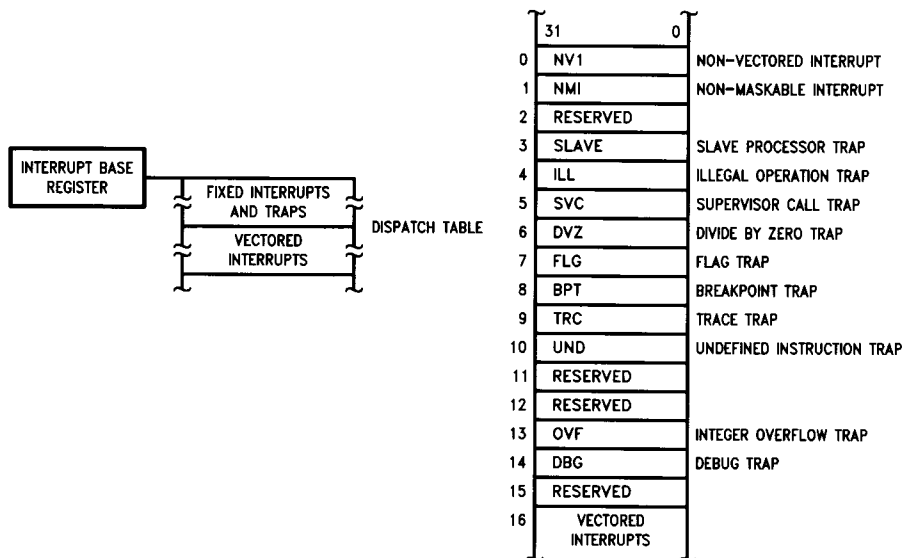
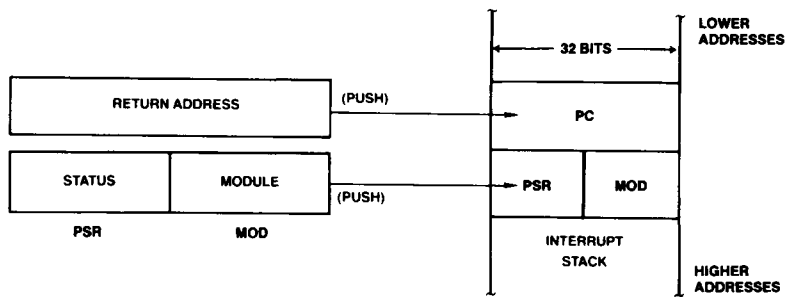


FIGURE 3-8. Interrupt Dispatch Table

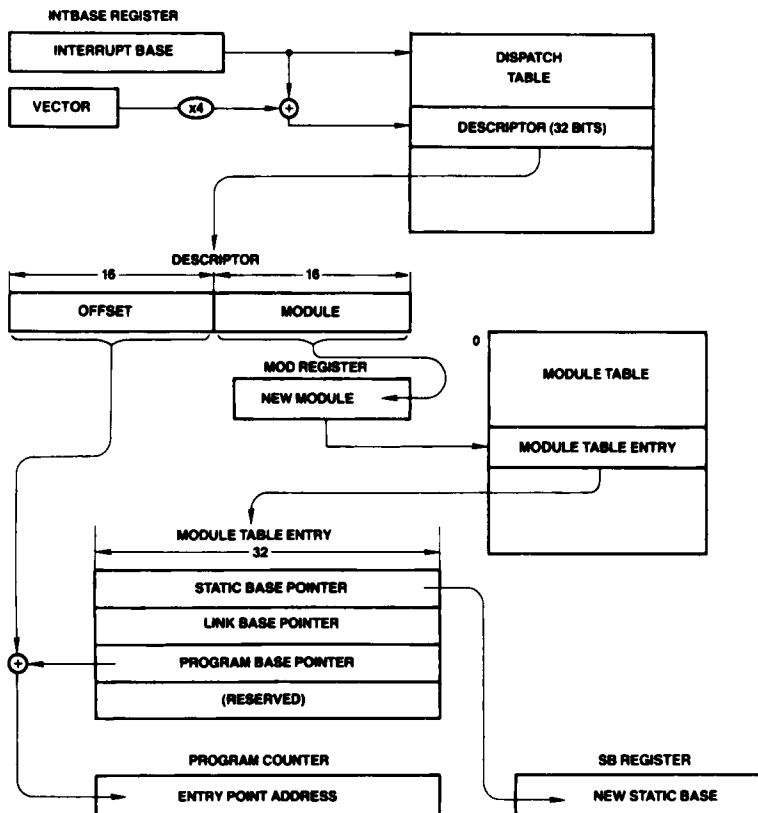
TL/EE/10564-11



### 3.0 Functional Description (Continued)



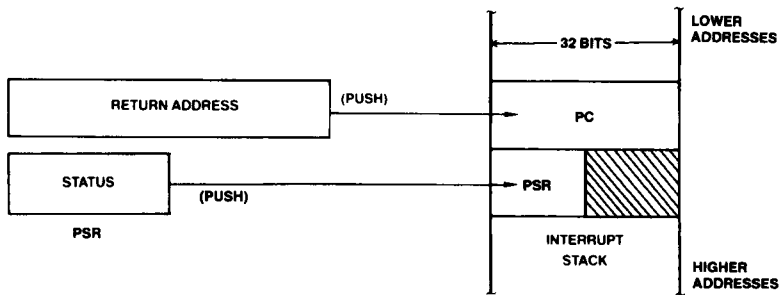
TL/EE/10564-12



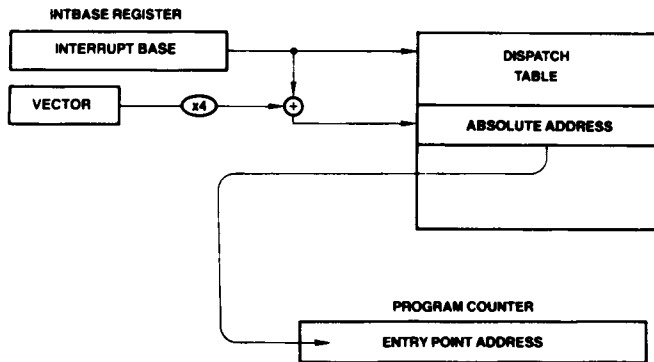
TL/EE/10564-13

**FIGURE 3-9. Exception Acknowledge Sequence. Direct-Exception Mode Disabled.**

### 3.0 Functional Description (Continued)



TL/EE/10564-14



TL/EE/10564-15

**FIGURE 3-10. Exception Acknowledge Sequence.  
Direct-Exception Mode Enabled.**

direct-exception mode is enabled, the CPU can respond more quickly to interrupts and other exceptions because fewer memory references are required to process an exception. The MOD and SB registers, however, are not initialized before the CPU transfers control to the service procedure. Consequently, the service procedure is restricted from executing any instructions, such as CXP, that use the contents of the MOD or SB registers in effective address calculations.

#### 3.2.2 Returning from an Exception Service Procedure

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

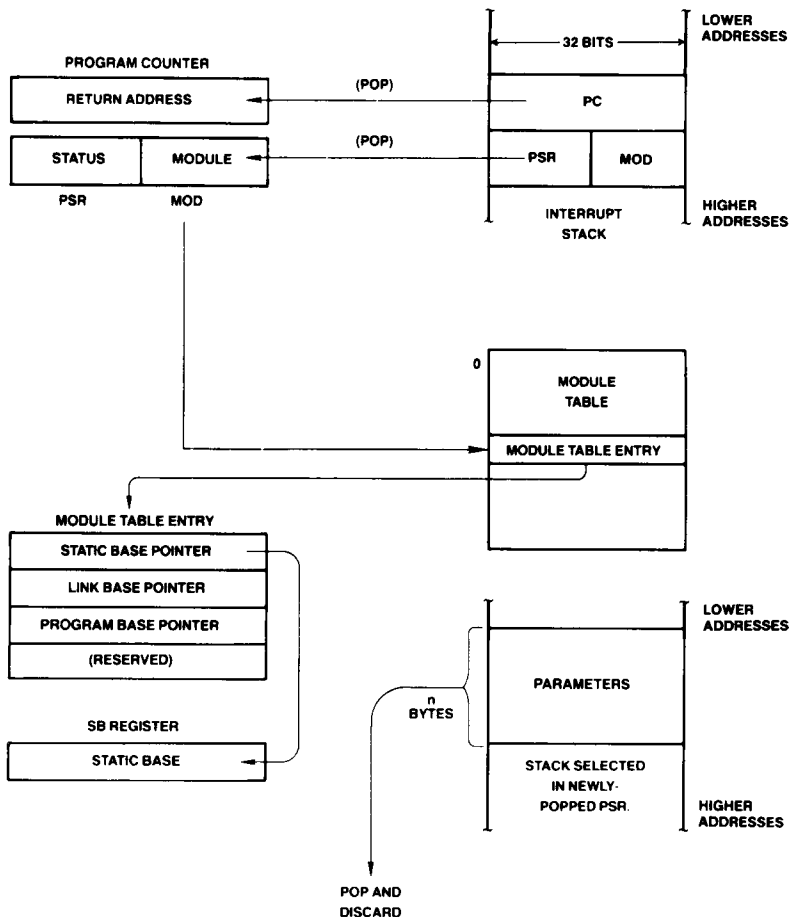
RETT is used to return from any trap, non-maskable interrupt or bus error service procedure. Since some traps are often used deliberately as a call mechanism for supervisor

mode procedures, RETT can also adjust the Stack Pointer (SP) to discard a specified number of bytes from the original stack as surplus parameter space.

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

Both of the above instructions always restore the Program Counter (PC) and the Processor Status Register from the interrupt stack. If the Direct-Exception mode is disabled, they also restore the MOD and SB register contents. *Figures 3-11 and 3-12* show the RETT and RETI instruction flows when the Direct-Exception mode is disabled.

### 3.0 Functional Description (Continued)



**FIGURE 3-11. Return from Trap (RETT n) Instruction Flow.  
Direct-Exception Mode Disabled.**

TL/EE/10564-16

#### 3.2.3 Maskable Interrupts

Maskable interrupt requests are generated either externally through the  $\overline{\text{IRQ}}0-3$  pins or internally by the DMA controller or the timers. These requests are enabled to generate an interrupt only while the I-bit in the PSR register is set to 1. The I bit is automatically cleared during service of a maskable interrupt, NMI or Trap (DBG), and is restored to its original setting upon return from the interrupt service routine via the RETT or RETI instruction.

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

If the non-vectored mode is selected, a default vector value of zero is always used. For the vectored mode instead, the on-chip Interrupt Control Unit will provide the CPU with a vector value. This vector value is then used as an index into the Dispatch Table in order to find the 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 on-chip ICU that it may re-prioritize any interrupt requests still pending.

### 3.0 Functional Description (Continued)

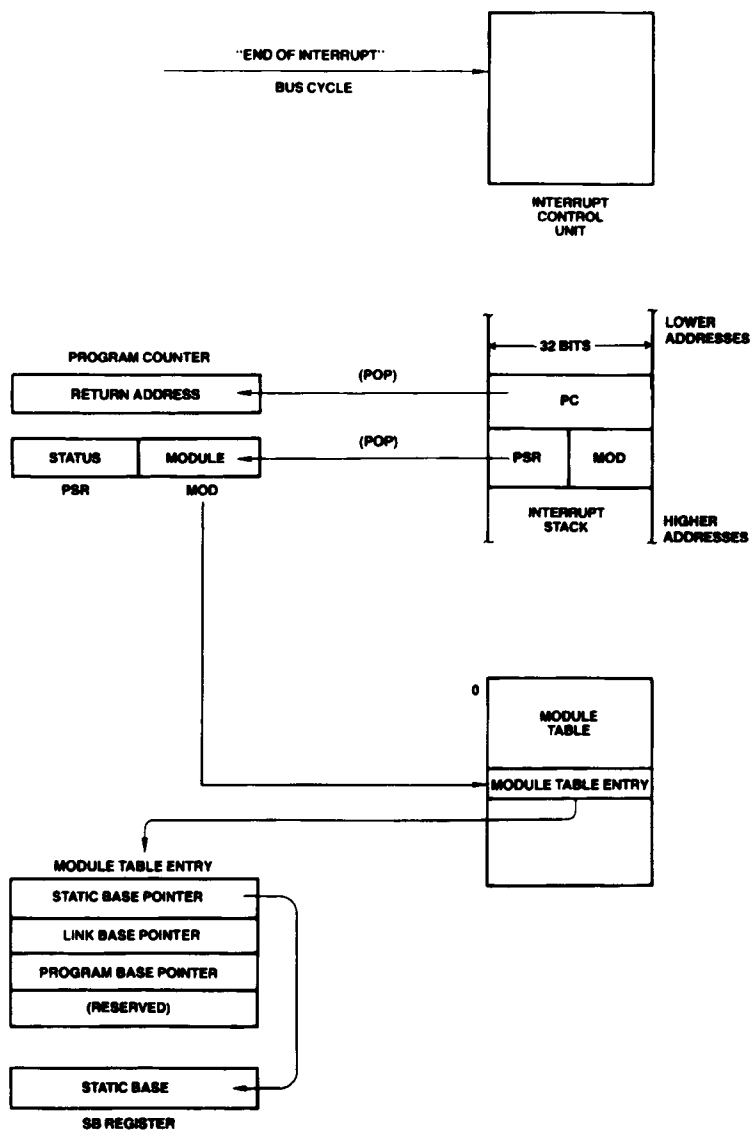


FIGURE 3-12. Return from Interrupt (RETI) Instruction Flow.  
Direct-Exception Mode Disabled.

TL/EE/10564-17

## 3.0 Functional Description (Continued)

### 3.2.4 Non-Maskable Interrupt

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

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

### 3.2.5 Traps

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

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

When a trap is recognized, maskable interrupts are not disabled except for the case of Trap (DBG).

There are 10 trap conditions recognized by the NS32GX320 as described below.

**Trap (SLAVE):** An exceptional condition was detected by the Floating Point Unit or another Slave Processor during the execution of a Slave Instruction. This trap is requested via the Status Word returned as part of the Slave Processor Protocol (Section 3.1.4.1).

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

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

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

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

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

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

**Trap (UND):** An Undefined-Instruction trap occurs when an attempt to execute an instruction is made and one or more of the following conditions is detected:

1. The instruction is undefined. Refer to Appendix A for a description of the codes that the CPU recognizes to be undefined.
2. The instruction is a floating point instruction and the F-bit in the CFG register is 0.
3. The instruction is a custom slave instruction and the C-bit in the CFG register is 0.
4. The reserved general addressing mode encoding (10011) is used.
5. Immediate addressing mode is used for an operand that has access class different from read.
6. Scaled Indexing is used and the basemode is also Scaled Indexing.
7. The instruction is a floating-point or custom slave instruction that the FPU or custom slave detects to be undefined. Refer to Section 3.1.4.1 for more information.

**Trap (OVF):** An Integer-Overflow trap occurs when the V-bit in the PSR register is set to 1 and an Integer-Overflow con-

dition is detected during the execution of an instruction. An Integer-Overflow condition is detected in the following cases:

1. The F-flag is 1 following execution of an ADDi, ADDQi, ADDCi, SUBi, SUBCi, NEGi, ABSi, or CHECKi instruction.
2. The product resulting from a MULi instruction cannot be represented exactly in the destination operand's location.
3. The quotient resulting from a DEli, DIVi, or QUOi instruction cannot be represented exactly in the destination operand's location.
4. The result of an ASHi instruction cannot be represented exactly in the destination operand's location.
5. The sum of the 'INC' value and the 'INDEX' operand for an ACBi instruction cannot be represented exactly in the index operand's location.

**Trap (DBG):** A debug trap occurs when one or more of the conditions selected by the settings of the bits in the DCR register is detected. This trap can also be requested by activating the input signal  $\overline{\text{DBG}}$ . Refer to Section 3.3.2 for more information.

**Note 1:** Following execution of the WAIT instruction, then a Trap (DBG) can be pending for a PC-match condition. In such an event, the Trap (DBG) is processed immediately.

**Note 2:** If an attempt is made to execute a privileged custom instruction while in User-Mode and the C-bit in the CFG register is 0, then Trap (UND) occurs.

**Note 3:** While operating in User-Mode, if an attempt is made to execute a privileged instruction with an undefined use of a general addressing mode (either the reserved encoding is used or else scaled-index or immediate modes are incorrectly used), the Trap (UND) occurs.

**Note 4:** If an undefined instruction or illegal operation is detected, then no data references are performed for the instruction.

**Note 5:** For certain instructions that are relatively long to execute, such as DEID, the CPU checks for pending interrupts during execution of the instruction. In order to reduce interrupt latency, the NS32GX320 can suspend executing the instruction and process the interrupt. Refer to Section B.5 in Appendix B for more information about recognizing interrupts in this manner.

### 3.2.6 Priority Among Exceptions

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

Figure 3-13 shows an exception processing flowchart.

Before executing an instruction, the CPU checks for pending Trap (DBG), interrupts, and Trap (TRC), in that order. If a Trap (DBG) is pending, then the CPU processes that exception, otherwise the CPU checks for pending interrupts. At this point, the CPU responds to any pending interrupt requests; nonmaskable interrupts are reorganized with higher priority than maskable interrupts. If no interrupts are pending, then the CPU checks the P-flag in the PSR to determine whether a Trap (TRC) is pending. If the P-flag is 1, a Trap (TRC) is processed. If no Trap (DBG), interrupt or Trap (TRC) is pending, the CPU begins executing the instruction.

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

1. trap (DBG) or interrupt, if the instruction is interruptible
2. one of 7 mutually exclusive traps: SLAVE, ILL, SVC, DVZ, FLG, BPT, UND

If no exception is detected while the instruction is executing, then the instruction is completed and the PC is updated to point to the next instruction. If a Trap (OVF) is detected, then it is processed at this time.

### 3.0 Functional Description (Continued)

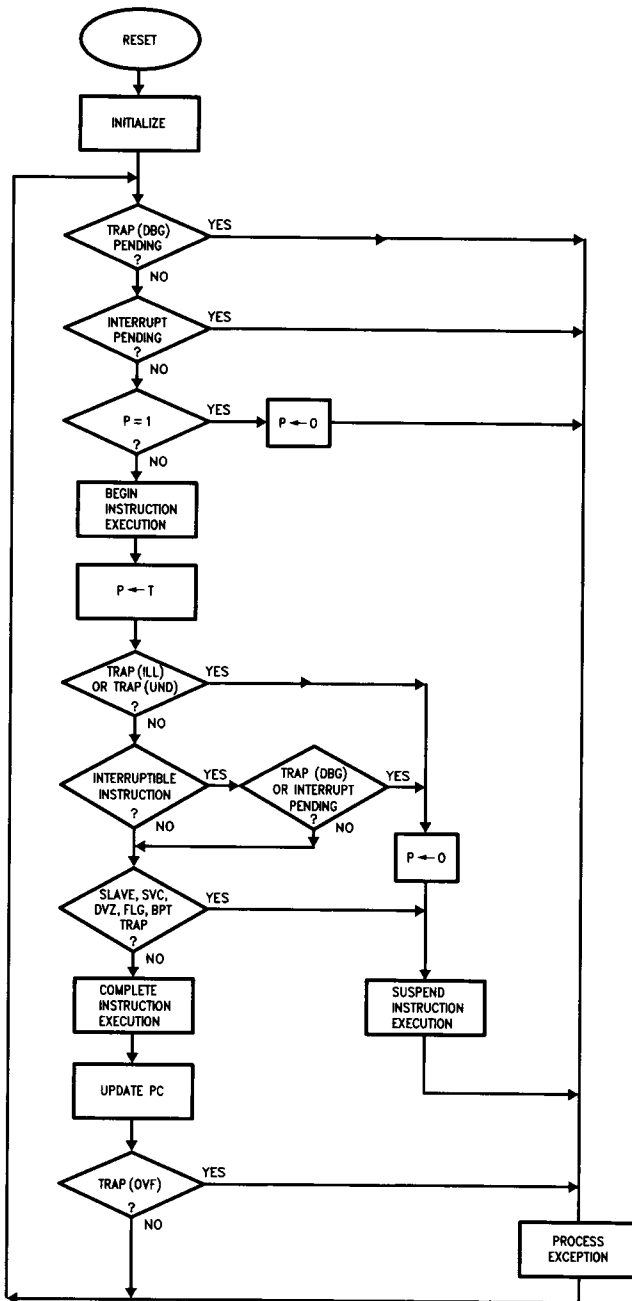


FIGURE 3-13. Exception Processing Flowchart

TL/EE/10564-18

## 3.0 Functional Description (Continued)

While executing the instruction, the CPU checks for enabled debug conditions. If an enabled debug condition is met, a Trap (DBG) is held pending until after the instruction is completed (see Note 3). If another exception is detected before the instruction is completed, the pending Trap (DBG) is removed and the DSR register is not updated.

**Note 1:** Trap (DBG) can be detected simultaneously with Trap (OVF). In this event, the Trap (OVF) is processed before the Trap (DBG).

**Note 2:** An address-compare debug condition can be detected while processing a bus error, interrupt, or trap. In this event, the Trap (DBG) is held pending until after the CPU has processed the first exception.

**Note 3:** Between operations of a string instruction, the CPU responds to pending operand address compare and external debug conditions as well as interrupts. If a PC-match debug condition is detected while executing a string instruction, then Trap (DBG) is held pending until the instruction has completed.

### 3.2.7 Exception Acknowledge Sequences: Detailed Flow

For purposes of the following detailed discussion of exception acknowledge sequences, a single sequence called "service" is defined in *Figure 3-14*.

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

#### 3.2.7.1 Maskable/Non-Maskable Interrupt Sequence

This sequence is performed by the CPU when the  $\overline{\text{NMI}}$  pin receives a falling edge, or an interrupt event is signalled either through  $\overline{\text{IRQ}}-3$  or by the on-chip peripherals and the PSR I bit set. The interrupt sequence begins either at the next instruction boundary or, in the case of an interruptible instruction (e.g., string instruction), at the next interruptible point during its execution.

1. If an interruptible 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 T, V, U, S, P and I.

3. If the interrupt is Non-Maskable:

- a. Do a type-1 special read cycle (Section 3.6.4.6) from address  $\text{FFFFFFF0}_{16}$ , applying Status Code 0100 (Interrupt Acknowledge). Discard the byte read.
- b. Set "Vector" to 1.
- c. Go to Step 6.

4. If the interrupt is Non-Vectored:

- a. Do a type-2 special read cycle (Section 3.6.4.6) from address  $\text{FFFFE00}_{16}$ , applying Status Code 0100 (Interrupt Acknowledge). Discard the byte read.
- b. Set "Vector" to 0.
- c. Go to Step 6.

5. Here the interrupt is Vectored.

- a. Do a type-2 special read cycle from address  $\text{FFFFE00}_{16}$ , applying status code 0100 (Interrupt Acknowledge), and discarding the byte read. This is to notify external circuitry that the interrupt is being acknowledged.

- b. Read vector byte from the IVCT register of the on-chip Interrupt Control Unit.

6. Perform Service (Vector, Return Address), *Figure 3-14*.

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

1. Restore the currently selected Stack Pointer and the Processor Status Register to their original values at the start of the trapped instruction.

2. Set "Vector" to the value corresponding to the trap type.

SLAVE: Vector = 3.

ILL: Vector = 4.

SVC: Vector = 5.

DVZ: Vector = 6.

FLG: Vector = 7.

BPT: Vector = 8.

UND: Vector = 10.

3. If Trap (ILL) or Trap (UND)

- a. Clear the Processor Status Register P bit.

4. Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits T, V, U, S and P.

5. Set "Return Address" to the address of the first byte of the trapped instruction.

6. Perform Service (Vector, Return Address), *Figure 3-14*.

#### 3.2.7.3 Trace Trap Sequence

1. In the Processor Status Register (PSR), clear the P bit.

2. Copy the PSR into a temporary register, then clear PSR bits T, V, U and S.

3. Set "Vector" to 9.

4. Set "Return Address" to the address of the next instruction.

5. Perform Service (Vector, Return Address), *Figure 3-14*.

#### 3.2.7.4 Integer-Overflow Trap Sequence

1. Copy the PSR into a temporary register, then clear PSR bits T, V, U, S and P.

2. Set "Vector" to 13.

3. Set "Return Address" to the address of the next instruction.

4. Perform Service (Vector, Return Address), *Figure 3-14*.

#### 3.2.7.5 Debug Trap Sequence

A debug condition can be recognized either at the next instruction boundary or, in the case of an interruptible instruction, at the next interruptible point during its execution.

1. If PC-match condition, then go to Step 3.

2. 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 instruction.

- c. Go to Step 4.

## 3.0 Functional Description (Continued)

3. Set "Return Address" to the address of the next instruction.
4. Set "Vector" to 14.
5. Copy the Processor Status Register (PSR) into a temporary register, then clear PSR bits T, V, U, S, P and I.
6. Perform Service (Vector, Return Address), *Figure 3-14*.

### 3.3 DEBUGGING SUPPORT

The NS32GX320 provides several features to assist in program debugging.

Besides the Breakpoint (BPT) instruction that can be used to generate soft breaks, the CPU also provides instruction tracing as well as debug trap (or hardware breakpoints) capabilities. Details on these features are provided in the following sub-sections.

#### 3.3.1 Instruction Tracing

Instruction tracing is a very useful feature that can be used during debugging to single-step through selected portions of

a program. Tracing is enabled by setting the T-bit in the PSR Register. When enabled, the CPU generates a Trace Trap (TRC) after the execution of each instruction.

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

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.

TABLE 3-3. Summary of Exception Processing

Exception	Instruction Ending	Cleared before Saving PSR	Cleared after Saving PSR
Interrupt	Before Instruction	None/P*	TVUSPI
ILL, UND	Suspended	P	TVUS
SLAVE, SVC, DVZ, FLG, BPT	Suspended	None	TVUSP
OVF	Completed	None	TVUSP
TRC	Before Instruction	P	TVUS
DBG	Before Instruction	None/P*	TVUSPI

\*Note: The P bit of the saved PSR is cleared in case the exception is acknowledged before the instruction is completed (e.g., interrupted string instruction). This is to avoid a mid-instruction trace trap upon return from the Exception Service Routine.

#### Service (Vector, Return Address):

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

Note: Some of the Memory Accesses indicated in the service sequence may be performed in an order different from the one shown.

FIGURE 3-14. Service Sequence



## 3.0 Functional Description (Continued)

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.

**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.

### 3.3.2 Debug Trap Capability

The CPU recognizes three different conditions to generate a Debug Trap:

- 1) Address Compare
- 2) PC Match
- 3) External

These conditions can be enabled and monitored through the CPU Debug Registers.

An address-compare condition is detected when certain memory locations are either read or written. The double-word address used for the comparison is specified in the CAR Register. The address-compare condition can be separately enabled for each of the bytes in the specified double-word, under control of the CBE bits of the DCR Register. The VNP bit in the DCR controls whether virtual or physical addresses are compared. The CRD and CWR bits in the DCR separately enable the address compare condition for read and write references; the CAE bit in the DCR can be used to disable the compare-address condition independently from the other control bits. The CPU examines the address compare condition for all data reads and writes, reads of memory locations for effective address calculations, Interrupt-Acknowledge and End-of-Interrupt bus cycles, and memory references for exception processing.

The PC-match condition is detected when the address of the instruction equals the value specified in the BPC register. The PC-match condition is enabled by the PCE bit in the DCR.

Detection of address-compare and PC-match conditions is enabled for User and Supervisor Modes by the UD and SD bits in the DCR. The DEN-bit can be used to disable detection of these two conditions independently from the other control bits.

An external condition is recognized whenever the  $\overline{DBG}$  signal is activated.

When the CPU detects an address-compare or PC-match condition while executing an instruction or processing an exception, then Trap (DBG) occurs if the TR bit in the DCR is 1. When an external debug condition is detected, Trap (DBG) occurs regardless of the TR bit. The cause of the Trap (DBG) is indicated in the DSR Register.

When an address-compare or PC-match condition is detected while executing an instruction, the CPU asserts the  $\overline{BP}$

signal at the beginning of the next instruction, synchronously with PFS. If the instruction is not completed because a higher priority trap is detected, the  $\overline{BP}$  signal may or may not be asserted.

**Note 1:** The assertion of  $\overline{BP}$  is not affected by the setting of the TR bit in the DCR register.

**Note 2:** When the LPRI instruction is executed to load a new value into the BPC, CAR or DCR, it is undefined whether the address-compare and PC-match conditions, in effect while executing the instruction, are detected under control of the old or new contents of the loaded register. Therefore, any LPRI instruction that alters the control of the address-compare or PC-match conditions should use register or immediate addressing mode for the source operand.

### 3.4 ON-CHIP CACHES

The NS32GX320 provides two on-chip caches: the Instruction Cache (IC) and the Data Cache (DC).

These are used to hold the contents of frequently used memory locations.

The IC and DC can be individually enabled by setting appropriate bits in the CFG Register (See Section 2.1.4).

The CPU also provides a locking feature that allows the contents of the IC and DC to be locked to specific memory locations. This is accomplished by setting the LIC and LDC bits in the CFG register.

Cache locking can be successfully used in real-time applications to guarantee fast access to critical instruction and data areas.

Details on the organization and function of each of the caches are provided in the following sections.

**Note 1:** Accesses to the on-chip peripherals are not cacheable.

**Note 2:** The size and organization of the on-chip caches may change in future Series 32000 microprocessors. This however, will not affect software compatibility.

#### 3.4.1 Instruction Cache (IC)

The basic structure of the instruction cache (IC) is shown in Figure 3-15.

The IC stores 512 bytes of code in a direct-mapped organization with 32 sets. Direct-mapped means that each set contains only one block, thus each memory location can be loaded into the IC in only one place.

Each block contains a 23-bit tag, which holds the most-significant bits of the physical address for the locations stored in the block, along with 4 double-words and 4 validity bits (one for each double-word).

A 4-double-word instruction buffer is also provided, which is loaded either from a selected cache block or from external memory. Instructions are read from this buffer by the loader unit and transferred to an 8-byte instruction queue.

The IC may or may not be enabled to cache an instruction being fetched by the CPU. It is enabled when the IC bit in the CFG Register is set to 1.

If the IC is disabled, the CPU bypasses it during the instruction fetch and its contents are not affected. The instruction is read directly from external memory into the instruction buffer.

### 3.0 Functional Description (Continued)

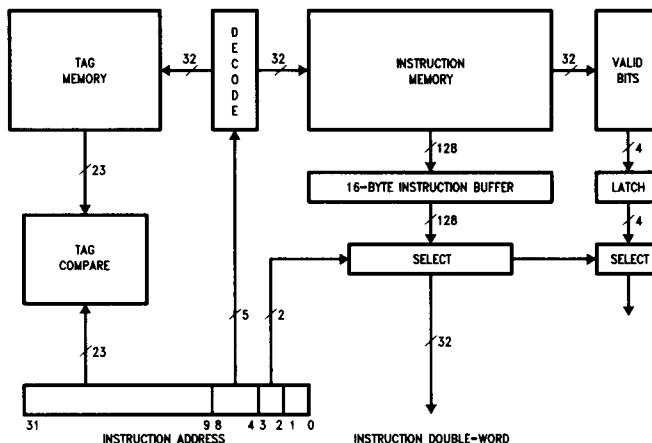


FIGURE 3-15. Instruction Cache Structure

TL/EE/10564-19

When the IC is enabled, the instruction address bits 4 to 8 are used to select the IC set where the instruction may be stored. The tag corresponding to the single block in the set is compared with the 23 most-significant bits of the instruction's physical address. The 4 double-words in this block are loaded into the instruction buffer and the 4 validity bits are also retrieved. Bits 2 and 3 of the instruction's physical address select one of these double-words and the associated validity bit.

If the tag matches and the selected double-word is valid, a cache 'hit' occurs and the double-word is directly transferred to the instruction queue for decoding; otherwise a cache 'miss' will result.

In the latter case, if the cache is not locked, the CPU will take the following actions.

First, if the tag of the selected block does not match, the tag is loaded with the 23 most-significant bits of the instruction address and all the validity bits are cleared. Then, the instruction is read from external memory into the instruction buffer.

If the CIIN input signal is not active during the fetching of the missing instruction, then the IC is updated and the instruction double-words fetched from memory are stored into it with the validity bits set.

If the cache is locked, its contents are not affected, as the CPU reads the missing instruction from external memory.

Whenever the CPU accesses external memory, whether or not the IC is enabled, it always fetches instruction double-words in a non-wrap-around fashion. Refer to Sections 3.6.4.3 and 3.6.6 for more information.

The contents of the instruction cache can be invalidated by software through the CINV instruction. Refer to Section 3.4.3 for details. Clearing the IC bit in the CFG Register also invalidates the instruction cache. Refer to Section C.2 for information on loading the CFG register.

**Note:** If the IC is enabled for a certain instruction and a 'miss' occurs due to a tag mismatch, the CPU will clear all the validity bits of the selected tag before fetching the instruction from external memory. If the CIIN input signal is activated during the fetching of that instruction, the validity bits are not set and the IC is not updated.

#### 3.4.2 Data Cache (DC)

The Data Cache (DC) stores 1,024 bytes of data in a two-way set associative organization as shown in Figure 3-16.

Each of the 32 sets has 2 cache blocks. Each block contains a 23-bit tag, which holds the most-significant bits of the address for the locations stored in the block, along with 4 double-words and 4 validity bits (one for each double-word).

The DC is enabled for a data read when all of the following conditions are satisfied.

- The DC bit in the CFG Register is set to 1.
- The reference is not an interlocked read resulting from executing a CBITI or SBITI instruction.

If the DC is disabled, the CPU bypasses it during the data read and its contents are not affected. The data is read directly from external memory. The DC is also bypassed for Interrupt-Acknowledge and End-of-Interrupt bus cycles.

When the DC is enabled for a data read, the address bits 4 to 8 are used to select the DC set where the data may be stored.

The tags corresponding to the two blocks in the set are compared to the 23 most-significant bits of the address. Bits 2 and 3 of the address select one double-word in each block and the associated validity bit.

If one of the tag matches and the selected double-word in the corresponding block is valid, a cache 'hit' occurs and the data is used to execute the instruction; otherwise a cache 'miss' will result. In the latter case, if the cache is not locked, the CPU will take the following actions.

## 3.0 Functional Description (Continued)

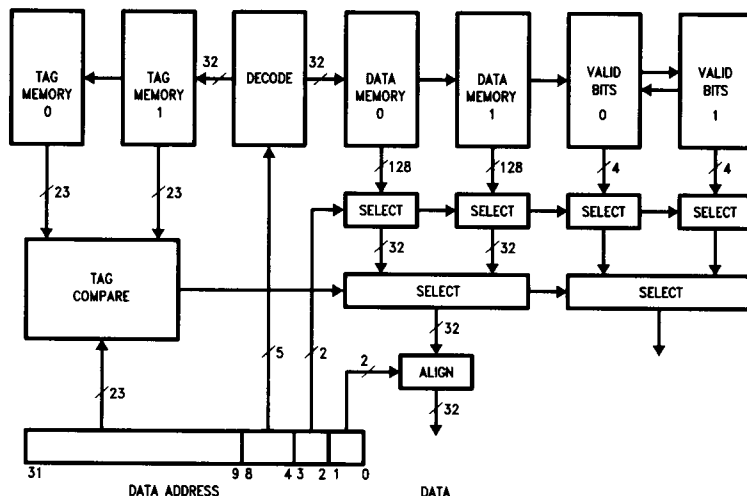


FIGURE 3-16. Data Cache Structure

TL/EE/10564-20

First, if the tag of either block in the set matches the data address, that block is selected for updating. Otherwise, if neither tag matches, then the least recently used block is selected; its tag is loaded with the 23 most-significant bits of the data address, and all the validity bits are cleared.

Then, the data is read from external memory; up to 4 double-word bits are read into the cache in a wrap-around fashion. Refer to Sections 3.6.4.3 and 3.6.6 for more information.

If the CIIN and IODEC input signals are both inactive during the bus cycles performed to read the missing data, then the DC is updated, as each double-word is read from memory, and the corresponding validity bit is set. If the cache is locked, its contents are not affected, as the CPU reads the missing data from external memory.

The DC is enabled for a data write whenever the DC bit in the CFG Register is set to 1, including interlocked writes resulting from executing the CBITI and SBITI instructions.

The DC does not use write allocation. This means that, during a write, if a cache 'hit' occurs, the DC is updated, otherwise it is unaffected. The data is always written through to external memory.

The contents of the data cache can be invalidated by software through the CINV instruction. Clearing the DC bit in the CFG Register also invalidates the data cache. Refer to Section C.2 for information on loading the CFG register.

**Note:** If the DC is enabled for a certain data reference and a "miss" occurs due to tag mismatch, the CPU will clear all the validity bits for the least recently used tag before reading the data from external memory. If either CIIN or IODEC are activated during the data read bus cycles, the validity bits are not set and the DC is not updated.

### 3.4.3 Cache Coherence Support

The NS32GX320 provides means for maintaining coherence between the on-chip caches and external memory. The CINV instruction can be executed to invalidate the In-

struction Cache and/or Data Cache; the CINV instruction can also be executed to invalidate a single 16-byte block in either or both caches.

In hardware, the use of the caches can be inhibited for individual locations using the CIIN input signal. A cache invalidation for both caches can be performed by activating the CINV input signal.

Whenever a CINV instruction is executed, the operation code and operand appear on the system interface using slave processor bus cycles. Thus, invalidations of the on-chip caches by software can be monitored externally.

Note, however, that the software is responsible for communicating to the external circuitry the values of the cache enable and lock bits in the CFG Register, since the CPU does not generate any special cycle (e.g., Slave Cycle) when the CFG Register is loaded.

## 3.5 ON-CHIP PERIPHERALS

Three types of on-chip peripherals are provided in the NS32GX320: a DMA controller, an interrupt control unit and timers.

Details on the operation of these peripherals are provided in the following sections.

### 3.5.1 DMA Controller

The on-chip DMA Controller provides 2 channels for transferring blocks of data between memory and I/O devices with minimal CPU intervention. Source and destination addresses as well as block size and type of operation are set up in advance by programming the appropriate control registers. Actual transfers are handled by the DMA channels in response to external transfer requests. Upon receiving a transfer request from an I/O device, the DMA Controller performs the following operations:

## 3.0 Functional Description (Continued)

1. Acquires control of the bus.
2. Acknowledges the requesting I/O device by asserting the appropriate  $\overline{\text{DAK}}$  signal.
3. Starts executing data transfer cycles according to the values stored into the control registers of the channel being serviced.
4. Terminates the data transfer operation whenever one of the following events occurs:
  - The specified number of bytes has been transferred
  - The  $\overline{\text{EOT}}$  signal is activated during a data transfer cycle
  - The software writes into the appropriate control registers

Each channel can be programmed for Flyby (Direct) or Indirect (Memory-to-Memory) data transfers. In addition, multiple transfer operations can be chained together by programming the appropriate control registers. Detailed descriptions of the different modes of operation are provided in the following sections.

### 3.5.1.1 Flyby (Direct) Transfers

In Flyby mode each data item is transferred using a single bus cycle without reading the data into the DMA controller. This mode offers the fastest transfer rate, but the source and destination bus widths must be the same. Data transfers cannot occur between two memory elements. One of the elements must always be an I/O device selected by the channel's  $\overline{\text{DAK}}$  signal. This device is referred to as the implied I/O device. The other element can be either memory or another I/O device, and is referred to as the addressed I/O device.

Since only one address is required in flyby mode, this address is taken from the corresponding ADCA counter. The DMA Channel generates either a read or a write bus cycle according to the setting of the DIR bit in the MODE register. When the DIR bit is 0, a read bus cycle from the addressed device is performed and the data is written to the implied I/O device. When the DIR bit is 1, a write bus cycle to the addressed device is performed, and the data is read from the implied I/O device.

The number of bytes transferred in each cycle is taken from the BWA field in the MODE register.

After the data element has been transferred, the BLTC counter is decremented by the number of bytes transferred. If the addressed device is memory, the ADCA counter is also incremented by the same amount.

**Note:** On Flyby transfers the channel may change the value of the ADCB register.

### 3.5.1.2 Indirect (Memory-to-Memory) Transfers

In this mode of operation, the transfer of each data item requires two bus cycles. The data is first read into a temporary register, and subsequently it is written into the destination. This mode is slower than the Flyby mode, but it provides support for differing bus widths between source and destination, and also for block transfers between two memory elements. Each element is an addressed device, and can be either memory or an I/O device. The  $\overline{\text{DAK}}$  signal corresponding to the channel is asserted during both bus cycles.

The first bus cycle is used to read data from the source using the ADCA counter, while the second bus cycle is used to write the data into the destination using the address in the ADCB counter. The number of bytes read is determined by the smaller of the BWA and BWB fields in the MODE register.

After the data item has been transferred, the DMA decrements the BLTC counter by the number of bytes transferred. The ADCA and ADCB counters are then incremented by the same amount, provided that the ADA and ADB bits in the MODE register are set to 1.

**Note:** If transfer operations between two memory areas are to be performed, external logic should be provided to activate the  $\overline{\text{DRQ}}$  signal under software control.

### 3.5.1.3 Single Transfer Operation

This mode provides the simplest way to accomplish a single block transfer operation. The block transfer addresses and byte count should be first written into the corresponding ADCA, ADCB and BLTC counters, the OT bit in the MODE register should be programmed for non auto-initialize mode, and the VLD bit in the CNTL register should be set to 0. When the CHEN bit in the CNTL register is set to 1, the channel becomes active and responds to external transfer requests.

When the BLTC counter reaches 0, the transfer operation terminates, the TC and OVR bits in the STAT register are set to 1, the CHAC bit is set to 0 and, if OVR is unmasked, the CHEN bit in the CNTL register is forced to 0. An interrupt can also be generated at the end of the transfer by setting the appropriate bits in the IMASK register.

### 3.5.1.4 Double-Buffer Operation

This mode allows the software to set up the next block transfer specification while the current block transfer is in progress. The operation is initialized by writing the block transfer addresses and byte count into the ADCA, ADCB and BLTC counters, and programming the OT bit in the MODE register for non auto-initialize mode. When the CHEN bit in the CNTL register is set to 1, the channel becomes active and responds to external transfer requests.

While the current block transfer is in progress, the software can write the addresses and byte count for the next block into the ADRA, ADRB and BLTR registers, and then set the VLD bit in the CNTL register to 1.

When the BLTC counter reaches 0, the TC bit is set to 1 and the DMA channel checks the value of the VLD bit. If it is 1, the channel copies the ADRA, ADRB and BLTR values into ADCA, ADCB and BLTC, and becomes ready to start the next block transfer. If the VLD bit is 0, the channel sets the OVR bit in the STAT register to 1, clears the CHAC bit and, if OVR is unmasked, it forces the CHEN bit to 0.

### 3.5.1.5 Auto-Initialize Operation

This mode allows the DMA controller to continuously fill the same memory area without software intervention. The operation is initialized by writing the block addresses and byte count into the ADCA, ADCB and BLTC counters as well as the ADRA, ADRB and BLTR registers, and programming the OT bit in the MODE register for auto-initialize mode. When the CHEN bit in the CNTL register is set to 1, the channel becomes active and responds to external requests.

### 3.0 Functional Description (Continued)

When the BLTC counter reaches 0, the TC bit in the STAT register is set to 1, the contents of the ADRA, ADRB and BLTR registers are copied to the ADCA, ADCB and BLTC counters, and the operation is repeated.

#### 3.5.1.6 Bus Arbitration

Whenever a DMA channel needs to perform a data transfer, it first needs to acquire control of the bus. Bus arbitration is performed according to a fixed priority scheme. An external HOLD request has the highest priority, followed by channel 0, channel 1 and, finally the CPU at the lowest priority. Once the bus is granted in response to a channel request and no higher priority request is pending, the channel can use the bus for a certain number of back-to-back transfers before it is forced to release it. This is controlled by a Bus Fairness mechanism whose purpose is to prevent bus monopolization from a DMA channel. The maximum number of back-to-back transfers can be programmed through the BLT field in the MODE register. When the programmed number of transfers is reached, the channel will release the bus for at least one clock cycle, so that it can be granted to a lower priority requester. Table 3-4 shows the maximum number of back-to-back transfers for different values of the BLT field.

**TABLE 3-4. Maximum Number of Back-to-Back DMA Transfers**

BLT Field	Maximum Number of Transfers		
	Byte Transfers	Word Transfers	Double-Word Transfers
00000	Unlimited	Unlimited	Unlimited
00001	1	1	1
00010	2	1	1
00100	4	2	1
01000	8	4	2
10000	16	8	4

**Note:** The values shown for the BLT field are the only ones allowed. Specifying a different value may cause unpredictable results.

#### 3.5.2 Interrupt Control Unit (ICU)

The on-chip Interrupt Control Unit (ICU) manages up to 15 levels of prioritized interrupt requests. Requests can be generated either externally or internally. External requests are binary encoded as a 4-bit value, and are input to the ICU through the  $\overline{IR}0-3$  pins. Internal requests are generated by the on-chip DMA controller and timers. Table 3-5 shows the possible interrupt sources and related priority levels.

The ICU keeps track of the interrupt priority levels currently in-service, and signals to the CPU only interrupt requests whose priority level is higher than the level of the highest priority interrupt currently being serviced. In addition, the ICU monitors the system bus and responds to Interrupt-Acknowledge and End-of-Interrupt bus cycles, by providing vector values to the CPU and updating the appropriate bits in the ISRV register.

**Note:** The Series 32000 interrupt handling specification remains unchanged, except for the elimination of cascaded maskable interrupt requests and their associated bus cycles. In particular, the vector numbers are always positive, in the range  $11_{16}$  through  $1F_{16}$ . The CPU interpretation and handling of the PSR I-bit and CFG I-bit remains unchanged. From the CPU standpoint, the on-chip ICU can be regarded as an independent module.

**TABLE 3-5. Interrupt Sources and Priority Levels**

Priority Level	Interrupt Source
INT15 (Highest)	External Only
INT14	External or DMA (DIP bit in IMSK is 1)
INT13	External or Timer 0 IPFA or Capture Mode Underflow
INT12	External or Timer 0 IPFB
INT11	External Only
INT10	External or Timer 1 IPFA or Capture Mode Underflow
INT9	External or Timer 1 IPFB
INT8	External Only
INT7	External Only
INT6	External or DMA (DIP bit in IMSK is 0)
INT5	External or Timer 2 IPFA or Capture Mode Underflow
INT4	External Only
INT3	External or Timer 2 IPFB
INT2	External Only
INT1 (Lowest)	External Only
—	No Interrupt

## 3.0 Functional Description (Continued)

### 3.5.2.1 Interrupt-Acknowledge Processing

When an Interrupt acknowledge cycle is performed, the bit in the ISRV register corresponding to the priority level of the current interrupt request is set to 1. This is specified in the least significant 4 bits of the IVCT register and represents the higher of the encoded priority value from the  $\overline{IR}0-3$  pins and the highest priority of any pending internal request. During the acknowledge cycle a special bus cycle is also executed, and the IVCT value is output on the data bus (Section 3.6.4.6).

Note that the IVCT register is not latched by the ICU even after the interrupt request is acknowledged by the CPU. This allows the software to examine the priority level of the current request by reading the IVCT register.

For proper ICU operation, the priority level of an interrupt request must not be decreased unless one of the following conditions is met.

1. The CPU performed an Interrupt-Acknowledge bus cycle
2. Maskable interrupts are currently disabled (PSR I-flag is 0)
3. A higher or same priority level interrupt is currently in-service

The first condition can be used by a requesting I/O device to determine when it is appropriate to remove the interrupt request. The other conditions could be used by the software to remove an interrupt request by accessing the I/O device's control registers.

### 3.5.2.2 End-of-Interrupt Processing

In response to an End-of-Interrupt bus cycle, the ICU clears the bit in the ISRV register corresponding to the highest priority interrupt currently in-service. The CPU is assumed to have returned to the next lower priority interrupt service routine.

In addition, the ICU returns the IVCT register value as data on the system bus. This value is not related to the priority level of the terminated interrupt routine that executed the RETI instruction. The returned IVCT value is ignored by the CPU.

Note that it is also possible to clear bits in the ISRV explicitly by software, for example, inside an interrupt service routine in order to reenale interrupts at the same or lower priority levels. In this case, either the RETT instruction should be used to terminate that interrupt service routine, or else the corresponding ISRV bit should be set to 1 again before executing the RETI instruction.

### 3.5.3 Timers

The NS32GX320 provides three on-chip timer blocks. Since these blocks are identical, the descriptions that follow are equally applicable to any one of them. Each timer block consists of a 16-bit counter TC, a control register TCNTL, and two support registers TRCA and TRCB. Two external signals TXA and TXB are also provided for each block to handle all the interactions with external logic. Each timer can operate in one of three modes: Processor Independent, External Event Counter, and Input Capture. Table 3-6 shows the TMC field encodings for the different modes. Details on the operation of each mode are given in the following sections.

#### 3.5.3.1 Processor Independent Mode (Mode 1)

This mode can be used to generate an output signal with minimal software intervention. The software only needs to define the ON and OFF times for the waveform to be generated. Once started, the timer will generate a periodic waveform without further intervention, except when the parameters need to be updated.

In this mode the timer counts down at the Tclk rate (Section 2.1.8). Upon the occurrence of every underflow the timer is alternately reloaded with the contents of the support registers TRCA and TRCB. The first timer underflow causes a reload from the TRCA register. Reloads from subsequent underflows alternate from the two support registers, starting with TRCB. Each underflow toggles the TXA output pin.

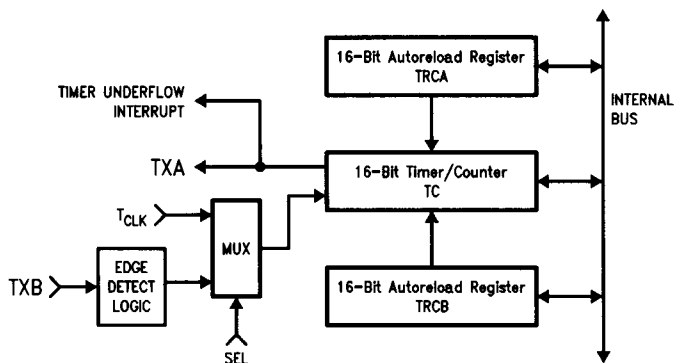
Timer underflows are alternately latched into the IPFA and IPFB flags. The software is responsible for resetting these flags. Two enable bits, IENA and IENB, allow timer underflow interrupts to be enabled or disabled. Setting the IENA bit will cause an interrupt when a timer underflow causes a reload from TRCA, while setting IENB will cause an interrupt when the reload is from TRCB.

The IENA and IENB bits give the user the flexibility to enable or disable interrupts on either or both edges of the timer output waveform.

#### 3.5.3.2 External Event Counter Mode (Mode 2)

This mode is similar to the processor independent mode. The only difference is that the timer is clocked by the rising edge of the signal applied on the TXB input pin.

### 3.0 Functional Description (Continued)



TL/EE/10564-21

**Note 1:**  $T_{clk} = BCLK/8$  if TCNT PRC-bit = 0.

$T_{clk} = BCLK/4096$  if TCNT PRC-bit = 1.

**Note 2:** Specifying a counter value of 0 yields a count of  $2^{16}$ .

**FIGURE 3-17. Timer Block Diagram for the Processor Independent and External Event Counter Modes**

**TABLE 3-6. Timer Modes**

TMC Field	Timer Mode	Interrupt A Source	Interrupt B Source	Timer Counts On
000	IDLE			
001	MODE 1 (Processor Independent)	Autoreload TRCA	Autoreload TRCB	Tclk
010	MODE 2 (External Event Counter)	Autoreload TRCA	Autoreload TRCB	TXB Rising Edge
100	MODE 3 (Input Capture) Captures on: TXA Rising Edge TXB Rising Edge	TXA Rising Edge or Timer Underflow	TXB Rising Edge	Tclk

**Note:** The values shown for the TMC field (Section 2.1.8) are the only ones allowed. Specifying a different value may cause unpredictable results.

#### 3.5.3.3 Input Capture Mode (Mode 3)

This mode allows to perform precise measurements of external frequencies and to time external events.

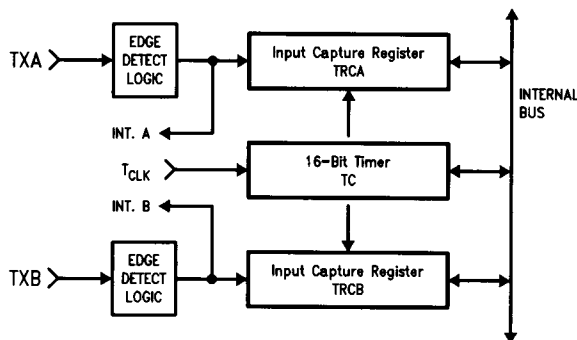
The timer constantly runs at the Tclk rate. The registers TRCA and TRCB act as capture registers, and are controlled by external signals applied on the TXA and TXB pins.

The timer value gets copied into the corresponding register when a trigger event is signaled on either TXA or TXB. A trigger event is specified as a rising edge of the input signal.

Trigger events can be programmed to generate interrupts. The occurrence of a trigger event on either TXA or TXB will be latched into IPFA or IPFB respectively. Interrupts are controlled by the setting of IENA and IENB.

Timer underflows can also generate interrupts. Since the underflow interrupt pending flag TCS has a different function in the other timer modes, the software should always reset it when the Input Capture Mode is selected. Timer underflow interrupts are also enabled by the IENA bit. Therefore, when IENA is set to 1 and an interrupt occurs, both IPFA and TCS should be checked to determine the origin of the interrupt.

### 3.0 Functional Description (Continued)



TL/EE/10564-22

**Note:** When the timer is idle TXA is in input state.

**FIGURE 3-18. Timer Block Diagram for the Input Capture Mode**

### 3.6 SYSTEM INTERFACE

This section provides general information on the NS32GX320 interface to the external world. Descriptions of the CPU requirements as well as the various bus characteristics are provided here. Details on other device characteristics including timing are given in Chapter 4.

#### 3.6.1 Power and Grounding

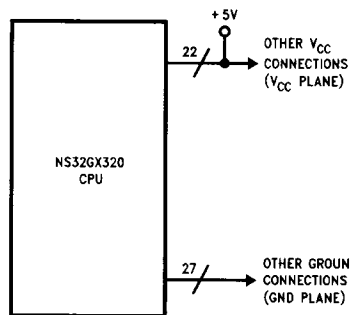
The NS32GX320 requires a single 5V power supply, applied on the  $V_{CC}$  pins. These pins should be connected together by a power ( $V_{CC}$ ) plane on the printed circuit board.

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

Both power and ground connections are shown in *Figure 3-19*.

#### 3.6.2 Clocking

The NS32GX320 requires a single-phase input clock signal (CLK) with frequency twice the CPU's operating frequency.



TL/EE/10564-23

**FIGURE 3-19. Power and Ground Connections**



### 3.0 Functional Description (Continued)

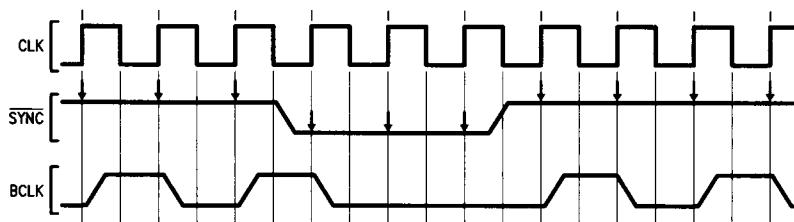


FIGURE 3-20. Bus Clock Synchronization

TL/EE/10564-24

This clock signal is internally divided by two to generate two non-overlapping phases PHI1 and PHI2. One single-phase clock signal BCLK in phase with PHI1 and its complement BCLK, are also generated and output by the CPU for timing reference.

Following power-on, the phase relationship between BCLK and CLK is undefined. Nevertheless, in some systems it may be necessary to synchronize the CPU bus timing to an external reference. The SYNC input signal can be used to initialize the phase relationship between CLK and BCLK. SYNC can also be used to stretch BCLK (Low) while CLK is toggling.

SYNC is sampled on each rising edge of CLK. As shown in Figure 3-20, whenever SYNC is sampled low, BCLK stops toggling and stays low. On the first rising edge that SYNC is sampled high, BCLK is driven high and then toggles on each subsequent rising edge of CLK.

Every rising edge of BCLK defines a transition in the timing state ("T-State") of the CPU.

One T-State represents the execution of one microinstruction within the CPU and/or one step of an external bus transfer.

**Note:** The CPU requirement on the maximum period of BCLK must be satisfied when SYNC is asserted at times other than reset.

#### 3.6.3 Resetting

The  $\overline{RST}$  input pin is used to reset the NS32GX320. The CPU samples  $\overline{RST}$  synchronously on the rising edge of BCLK. 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 latches for the edge-sensitive NMI and DBG signals are cleared.

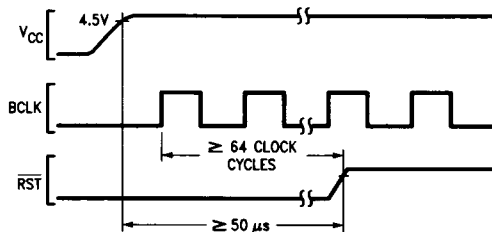
The CPU stores the PC contents in the R0 Register and the PSR contents in the least-significant word of R1, leaving the most-significant word undefined. The PC is then cleared to 0 and so are all the implemented bits in the PSR, CFG and the control registers of the on-chip peripheral devices. The DEN-bit in the DCR Register is also cleared to 0. After reset, the remaining implemented bits in DCR and the contents of all other registers are undefined. The CPU begins executing the instruction at Address 0.

On application of power,  $\overline{RST}$  must be held low for at least 50  $\mu s$  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 BCLK cycles. See Figures 3-21 and 3-22.

While in the Reset state, the CPU drives the signals  $\overline{ADS}$ , BE0-3, BMT, CONF, ICONF, PAGE, HLDA and DAK0-1 inactive. The data bus is floated and the state of all other output signals is undefined.

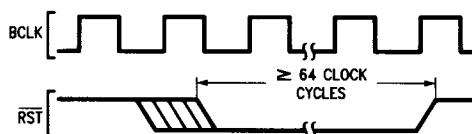
**Note 1:** If HOLD is active at the time  $\overline{RST}$  is deasserted, the CPU acknowledges HOLD before performing any bus cycle.

**Note 2:** If SYNC is asserted while the CPU is being reset, then BCLK does not toggle. Consequently, SYNC must be high for at least 128 CLK cycles while  $\overline{RST}$  is low.



TL/EE/10564-25

FIGURE 3-21. Power-On Reset Requirements



TL/EE/10564-26

FIGURE 3-22. General Reset Timing

#### 3.6.4 Bus Cycles

The NS32GX320 will perform bus cycles for one of the following reasons:

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

## 3.0 Functional Description (Continued)

### 3.6.4.1 Bus Status

The 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 yet need to access the bus.
- 0001** The bus is idle because the CPU is waiting for an interrupt following execution of the WAIT instruction.
- 0010** Reserved
- 0011** The bus is idle because the CPU is waiting for a Slave Processor to complete executing an instruction.
- 0100** Interrupt Acknowledge.  
The CPU is performing a dummy read cycle to inform external circuitry that an interrupt is being acknowledged.
- 0101** Reserved
- 0110** End of Interrupt.  
The CPU is performing a dummy read cycle to indicate that it is executing a Return from Interrupt (RET) instruction at the completion of an interrupt's service procedure.
- 0111** Reserved
- 1000** Sequential Instruction Fetch.  
The CPU is fetching the next double-word in sequence from the instruction stream.
- 1001** Non-Sequential Instruction Fetch.  
The CPU is fetching the first double-word of a new sequence of instruction. This will occur as a result of any JUMP or BRANCH, any exception, or after the execution of certain instructions.
- 1010** Data Transfer.  
The CPU is reading or writing an operand for an instruction, or it is referring to memory while processing an exception.
- 1011** Read RMW Class Operand.  
The CPU is reading an operand with access class of read-modify-write.
- 1100** Read for Effective Address Calculation.  
The CPU is reading a pointer from memory in order to calculate an effective address for Memory Relative or External addressing modes.
- 1101** Transfer Slave Processor Operand.  
The CPU is transferring an operand to or from a Slave Processor.

### 1110 Read Slave Processor Status.

The CPU is reading a status word from a slave processor after the slave processor has activated the FSSR signal.

### 1111 Broadcast Slave Processor ID + OPCODE.

The CPU is initiating the execution of a Slave Instruction by transferring the first 3 bytes of the instruction, which specify the Slave Processor identification and operation.

### 3.6.4.2 Basic Read and Write Cycles

The sequence of events occurring during a basic CPU access to either memory or peripheral device is shown in *Figure 3-23* for a read cycle, and *Figure 3-24* 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 the RDY line. See Section 3.6.4.4.

A full speed bus cycle is performed in two cycles of the BCLK clock, labeled T1 and T2. For both read and write bus cycles the CPU asserts ADS during the first half of T1 indicating the beginning of the bus cycle. From the beginning of T1 until the completion of the bus cycle the CPU drives the Address Bus and other relevant control signals as indicated in the timing diagrams. For cacheable data read cycles the CPU also drives the CASEC signal to indicate the block in the DC set where the data will be stored. If the bus cycle is not cancelled (e.g., state T2 is entered in the next clock cycle), the confirm signal (CONF) is asserted in the middle of T1. Note that due to a bus cycle cancellation, the BMT signal may be asserted at the beginning of T1, and then deasserted before the time in which it is guaranteed valid (see Section 4.4.2).

A confirmed bus cycle is completed at the end of T2, unless a cycle extension is requested. Following state T2 is either state T1 of the next bus cycle, or an idle T-state, if the CPU has no bus cycle to perform.

In case of a read cycle the CPU samples the data bus at the end of state T2.

If a bus exception is detected, the data is ignored.

For write bus cycles, valid data is output from the middle of T1 until the end of the cycle. When a write bus cycle is immediately followed by another write cycle, the CPU keeps driving the bus with the data related to the previous cycle until the middle of state T1 of the second bus cycle.

The CPU always inserts an idle state before a write cycle when the write immediately follows a confirmed read cycle.

**Note:** The CPU can initiate a bus cycle with a T1-state and then cancel the cycle, such as when a Cache hit occurs. In such a case, the CONF signal remains High and the BMT signal is driven High; the T1-state is followed by another T1-state or an idle T-state.

### 3.0 Functional Description (Continued)

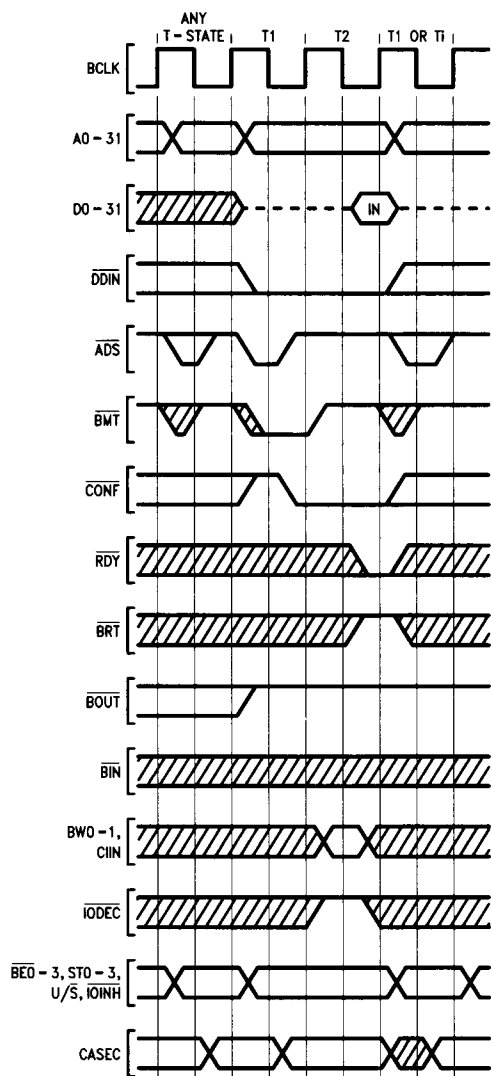


FIGURE 3-23. Basic Read Cycle

TL/EE/10564-27

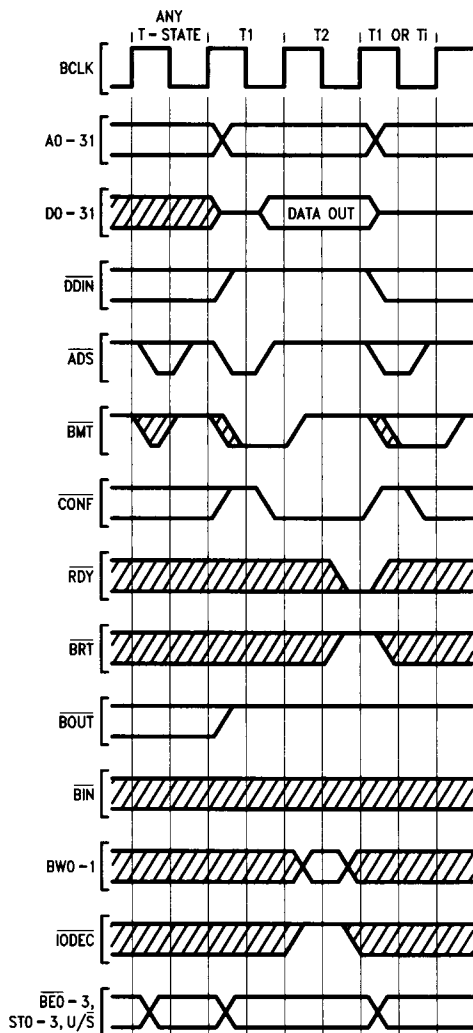


FIGURE 3-24. Write Cycle

TL/EE/10564-28

## 3.0 Functional Description (Continued)

### 3.6.4.3 Burst Cycles

The NS32GX320 is capable of performing burst cycles in order to increase the bus transfer rate. Burst is only available in instruction fetch cycles and data read cycle from 32-bit wide memories. Burst is not supported in operand write cycles or slave cycles.

The sequence of events for burst cycles is shown in *Figure 3-25*. The case shown assumes that the selected memory is capable of communicating with the CPU at full speed. If not, then cycle extension can be requested through the  $\overline{\text{RDY}}$  line. See Section 3.6.4.4.

A Burst cycle is composed of two parts. The first part is a regular cycle (opening cycle), in which the CPU outputs the new status and asserts all the other relevant control signals. In addition, the Burst Out Signal ( $\overline{\text{BOUT}}$ ) is activated by the CPU indicating that the CPU can perform Burst cycles. If the selected memory allows Burst cycles, it will notify the CPU by activating the burst in signal ( $\overline{\text{BIN}}$ ).  $\overline{\text{BIN}}$  is sampled by the CPU in the middle of T2 on the falling edge of BCLK. If the memory does not allow burst ( $\overline{\text{BIN}}$  high), the cycle will terminate at the end of T2 and  $\overline{\text{BOUT}}$  will go inactive immediately. If the memory allows burst ( $\overline{\text{BIN}}$  low), and the CPU has not deasserted  $\overline{\text{BOUT}}$ , the second part of the Burst cycle will be performed and  $\overline{\text{BOUT}}$  will remain active until termination of the Burst.

The second part consists of up to 3 nibbles, labeled T2B. In each of them a data item is read by the CPU. For each nibble in the burst sequence the CPU forces the 2 least-significant bits of the address to 0 and increments address bits 2 and 3 to select the next double-word; all the byte enable signals ( $\overline{\text{BE0}}\text{--}3$ ) are activated.

As shown in *Figures 3-25* and *4-8* (in Section 4), the CPU samples  $\overline{\text{RDY}}$  at the end of each nibble. It extends the access time for the burst transfer if  $\overline{\text{RDY}}$  is inactive.

The CPU initiates burst read cycles in the following cases.

1. An instruction must be fetched (Status = 1000 or 1001), and the instruction address does not fall within the last double-word in an aligned 16-byte block (e.g., address bits 2 and 3 are not both equal to 1).
2. A data item must be read (Status = 1010, 1011 or 1100), and both of the following conditions are met.
  - The data cache is enabled and not locked. (DC = 1 and LDC = 0 in the CFG register.)
  - The bus cycle is not an interlocked data access performed while executing a CBITI or SBITI instruction.

The Burst sequence will be terminated when one of the following events occurs.

1. The last instruction double-word in an aligned 16-byte block has been fetched.
2. The CPU detects that the instructions being prefetched are no longer needed due to an alteration of the flow of control. This happens, for example, when a Branch instruction is executed or an exception occurs.
3. 4 double-words of data have been read by the CPU. The double-words are transferred within an aligned 16-byte block in a wrap-around order. For example, if a source operand is located at address 104<sub>16</sub>, then the burst read cycle transfers the double-words at 104, 108, 10C, and 100, in that order.

3.0 Functional Description (Continued)

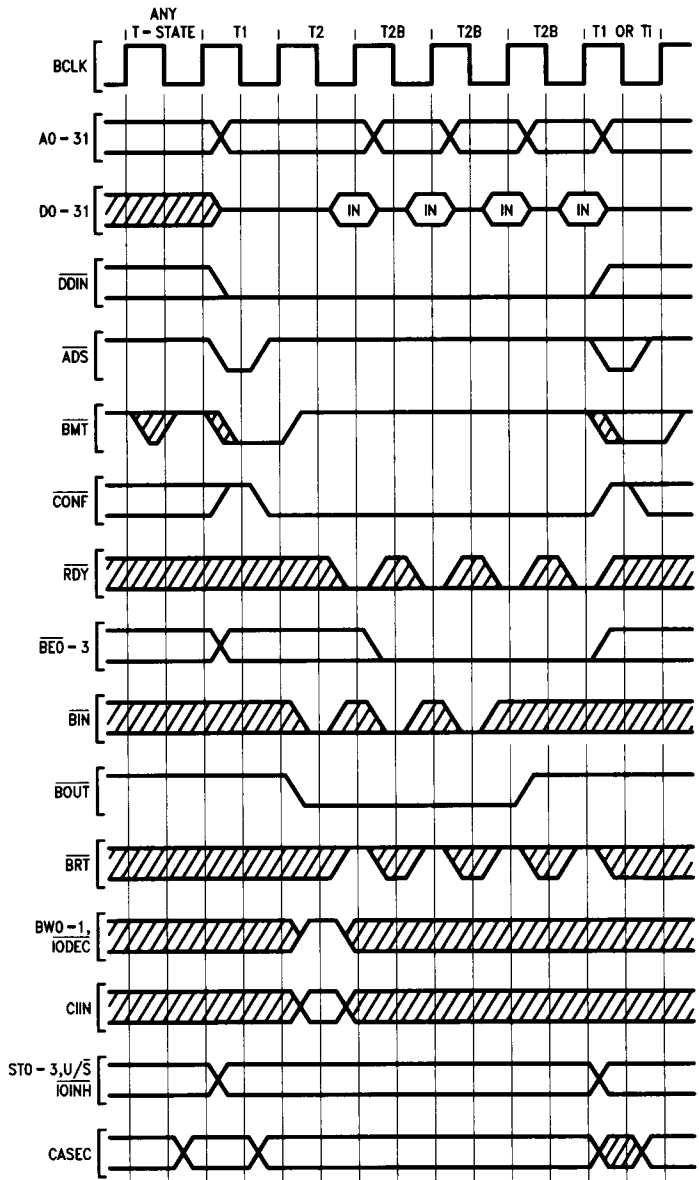


FIGURE 3-25. Burst Read Cycles

TL/EE/10564-29

### 3.0 Functional Description (Continued)

4. The  $\overline{\text{BIN}}$  signal is deasserted.
5.  $\overline{\text{BRT}}$  is asserted to signal a bus retry.
6.  $\overline{\text{IODEC}}$  is asserted or the  $\text{BW0-1}$  signals indicate a bus width other than 32-bits. The CPU samples these signals during state T2 of the opening cycle. During T2B-states  $\text{BW0-1}$  are ignored and  $\overline{\text{IODEC}}$  must be kept HIGH.

The CPU uses only the values of the above signals sampled during the last state of the transfer when the cycle is extended. See Section 3.6.4.4.

**Note:** A burst sequence is not stopped by the assertion of  $\text{CIIN}$ . See Note 3 in Section 3.6.5.

#### 3.6.4.4 Cycle Extension

To allow sufficient access time for any speed of memory or peripheral device, the NS32GX320 provides for extension of a bus cycle. Any type of bus cycle except a slave processor cycle can be extended.

A bus cycle can be extended by causing state T2 for a normal cycle or state T2B for a Burst cycle to be repeated.

At the end of each T2 or T2B state, on the rising edge of  $\text{BCLK}$ , the  $\overline{\text{RDY}}$  line is sampled by the CPU. If  $\overline{\text{RDY}}$  is active, then the transfer cycle will be completed. If  $\overline{\text{RDY}}$  is inactive, then the bus cycle is extended by repeating the T-state for another clock cycle. These additional T-states inserted by the CPU in this manner are called 'WAIT' states.

During a transfer the CPU samples the input control signals  $\overline{\text{BIN}}$ ,  $\overline{\text{BRT}}$ ,  $\text{BW0-1}$ ,  $\overline{\text{PLAT}}$ ,  $\text{EOT}$ ,  $\text{CIIN}$  and  $\overline{\text{IODEC}}$ .

When wait states are inserted, only the values of these signals sampled during the last wait state are significant.

Figure 3-26 illustrates a normal read cycle with wait states added through the  $\overline{\text{RDY}}$  pin.

**Note:** If  $\overline{\text{RST}}$  is asserted during a bus cycle, then the cycle is terminated without regard of  $\overline{\text{RDY}}$ .

#### 3.6.4.5 Interlocked Bus Cycles

The NS32GX320 supports indivisible read-modify-write transactions by asserting the  $\overline{\text{ILO}}$  signal during consecutive read and write operations. See Figure 4-7 in Section 4.

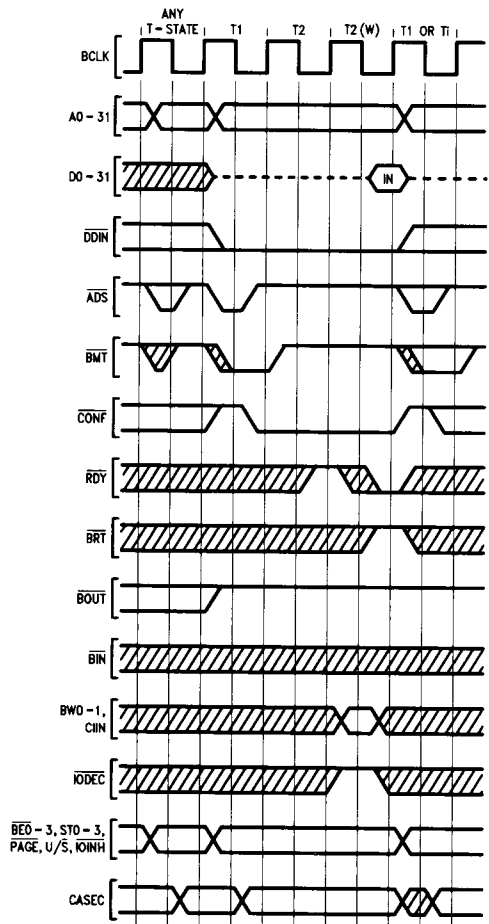
Interlocked transactions are always preceded and followed by one or more idle T-states.

The  $\overline{\text{ILO}}$  signal is asserted in the middle of the idle T-state preceding state T1 of the read operation, and is deasserted in the middle of one of the idle T-states following completion of the write operation, including any retried bus cycles.

No other bus operations (e.g., instruction fetches) will occur while an interlocked transaction is taking place.

Interlocked transactions are required in multiprocessor systems to handle shared resources. The CPU uses them to reference data while executing a  $\text{CBITli}$  or  $\text{SBITli}$  instruction, during which a single byte of data is read and written.

The  $\overline{\text{ILO}}$  signal is always released for one or more clock cycles in the middle of two consecutive interlocked transactions.



TL/EE/10564-30

FIGURE 3-26. Cycle Extension of a Basic Read Cycle

## 3.0 Functional Description (Continued)

### 3.6.4.6 Special Bus Cycles

Special bus cycles are performed during CPU accesses to some locations in the top area of the address space. These cycles may be used by external logic to track CPU activities involving the on-chip I/O devices as well as to monitor interrupt acknowledges and returns. Two types of special bus cycles are provided: Type-1 and type-2. Type-1 special cycles are identical to normal cycles, except that the CPU ignores the data placed on the bus in case of reads. The external logic must assert  $\overline{RDY}$  in order for the cycle to complete. This is the case, for example, of the acknowledge cycle for a Non-Maskable Interrupt.

Type-2 special cycles are significantly different from normal cycles. They still start with the assertion of  $\overline{ADS}$  and take 2 clock cycles to execute, but instead of  $\overline{BMT}$  and  $\overline{CONF}$ , they use the special signal  $\overline{ICONF}$ . The  $\overline{RDY}$ ,  $\overline{BRT}$ ,  $\overline{IODEC}$ ,  $\overline{PLAT}$  and  $\overline{BW0-1}$  signals are ignored, and the CPU always places data on the external data bus, regardless of whether the cycle is a read or a write. Type-2 cycles are performed when the CPU accesses any of the on-chip peripheral devices registers, and during acknowledges and returns of Maskable Interrupts. *Figure 3-27 shows the timing diagram of a type-2 special cycle.*

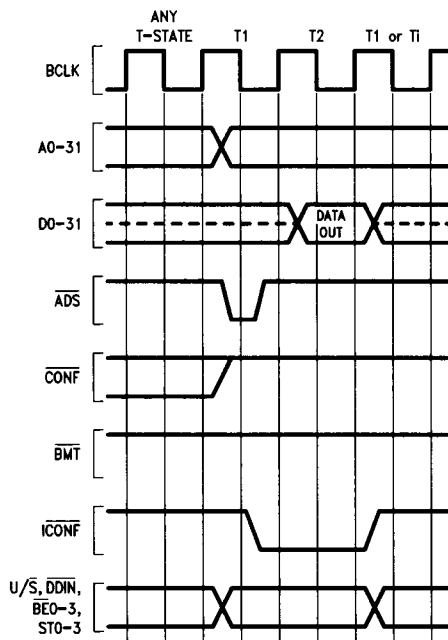


FIGURE 3-27. Type-2 Special Bus Cycle

TL/EE/10564-31

## 3.0 Functional Description (Continued)

### 3.6.4.7 DMA Controller Bus Cycles

DMA bus cycles are executed by the NS32GX320 whenever a data transfer is performed by one of the on-chip DMA channels. These cycles can be differentiated from normal CPU cycles by the fact that either  $\overline{DAK0}$  or  $\overline{DAK1}$  is asserted. The following restrictions apply to DMA cycles:

- Dynamic bus width is not supported, therefore the BW0–1 input signals are ignored.
- Burst mode is not supported.
- The reserved 8 Mbyte address range starting from  $FF800000_{16}$  cannot be accessed. *Figures 3–28 to 3–30* show DMA cycle timings for different situations.

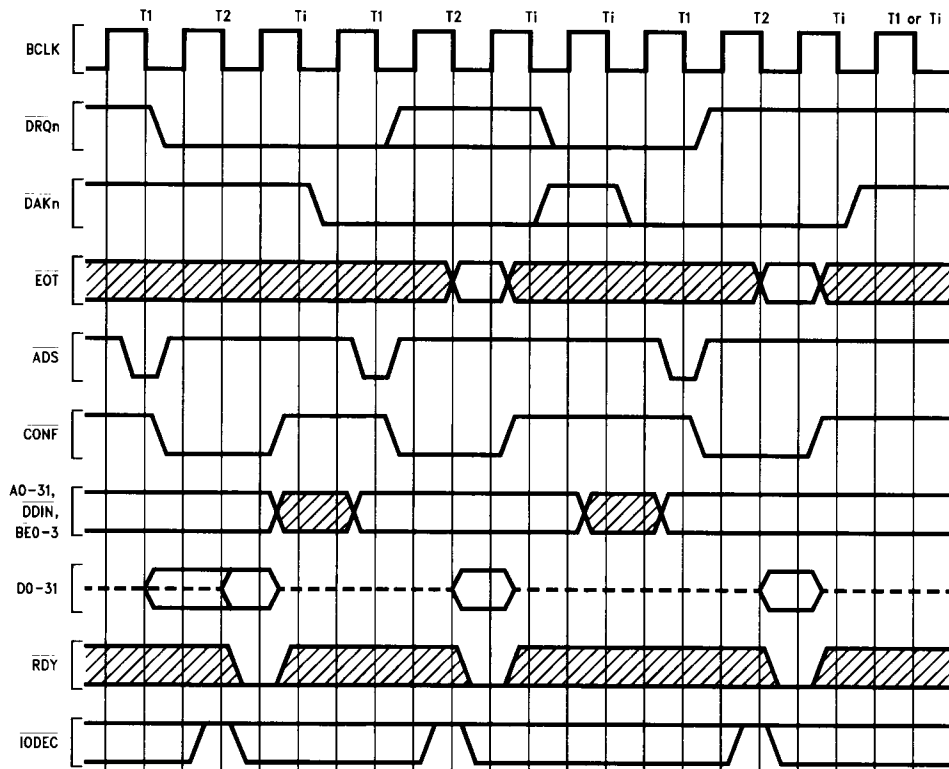


FIGURE 3-28. Flyby DMA Transfer Cycles (Bus Initially Not Idle).

TL/EE/10564–32



### 3.0 Functional Description (Continued)

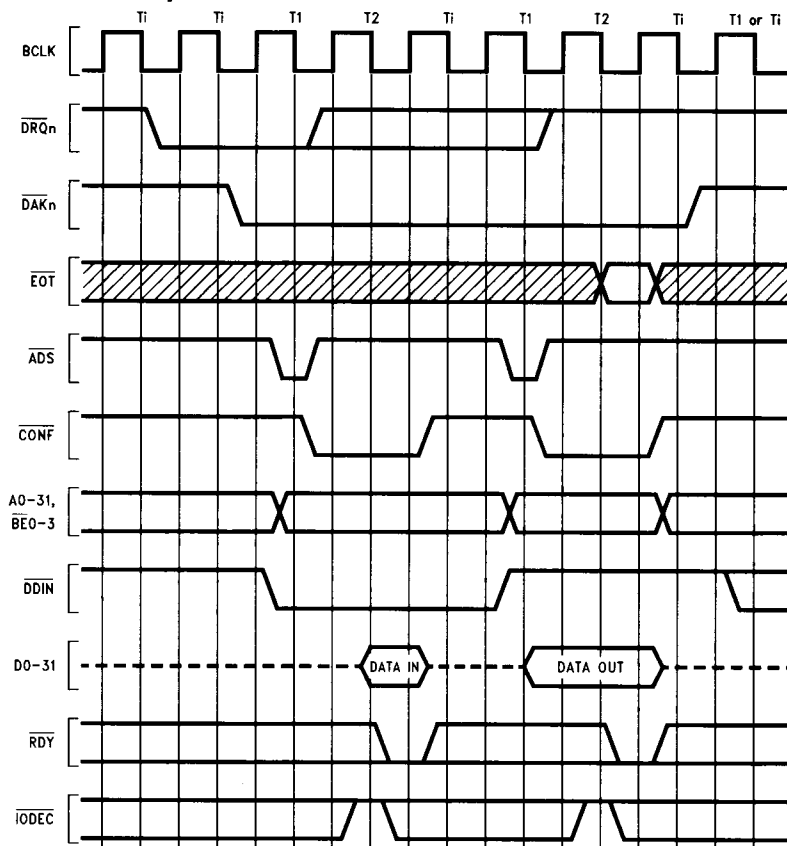


FIGURE 3-29. Indirect (Memory-to-Memory) DMA Transfer Cycles (Bus Initially Idle).

TL/EE/10564-33

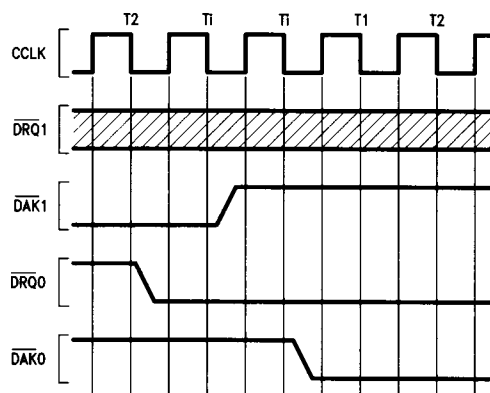


FIGURE 3-30. Successive DMA Transfers on Different Channels

TL/EE/10564-34

## 3.0 Functional Description (Continued)

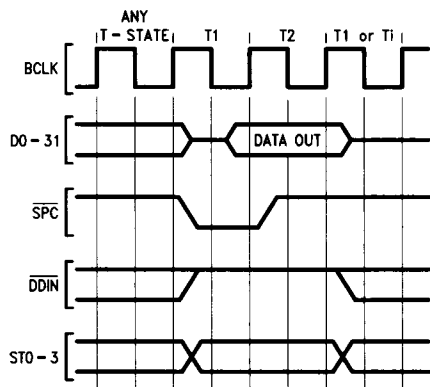
### 3.6.4.8 Slave Processor Bus Cycles

The NS32GX320 performs bus cycles to transfer information to or from slave processors while executing floating-point or custom-slave instructions.

The CPU uses slave write bus cycles to broadcast the identification and operation codes of a slave instruction as well as to transfer operands from memory or general purpose registers to a slave.

Figure 3-31 shows the timing for a slave write bus cycle. The CPU asserts  $\overline{SPC}$  during T1; the status is valid during T1 and T2. The operation code or operand is output on the data bus from the middle of T1 until the end of T2.

The CPU uses a slave read bus cycle to transfer a result operand from a slave to either memory or a general purpose register. A slave read cycle is also used to read a status word when the  $\overline{FSSR}$  signal is asserted. Figure 3-32 shows the timing for a slave read bus cycle.



TL/EE/10564-35

FIGURE 3-31. Slave Processor Write Cycle

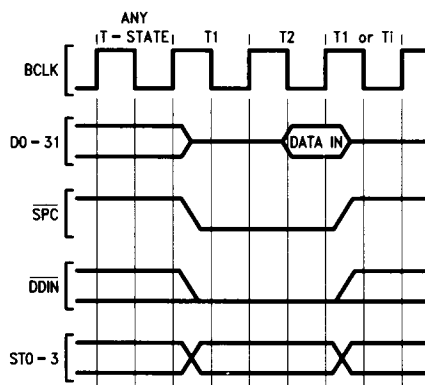
During T1 and T2 the CPU drives the status lines and asserts  $\overline{SPC}$ . The data from the slave is sampled at the end of T2.

The CPU will never perform a slave cycle immediately following a memory read or slave read cycle. In fact, the TRI-STATE following state T2 of a slave read cycle is either an idle TRI-STATE or the T1 state of a memory cycle.

Slave processor data transfers are always 32 bits wide. If the operand is a single byte, then it is transferred on D0 through D7. If it is a word, then it is transferred on D0 through D15.

When two operands are transferred, operand 1 is transferred before operand 2. For double-precision operands, the least-significant double-word is transferred before the most-significant double-word.

During a slave bus cycle the output signals  $\overline{BE0-3}$  are undefined while the input signals  $BW0-1$ ,  $\overline{PLAT}$  and  $\overline{RDY}$  are ignored.  $\overline{BRT}$  must be kept high.



TL/EE/10564-36

FIGURE 3-32. Slave Processor Read Cycle

## 3.0 Functional Description (Continued)

TABLE 3-7. Interrupt Sequences

Cycle	Status	Address	DDIN	BE3	BE2	BE1	BE0	Data Bus			
								Byte 3	Byte 2	Byte 1	Byte 0
A. Non-Maskable Interrupt Control Sequences											
Interrupt Acknowledge											
1	0100	FFFFFF00 <sub>16</sub>	0	1	1	1	0	X	X	X	X
Interrupt Return											
None: Performed through Return from Trap (RETT) instruction.											
B. Non-Vectored Interrupt Control Sequences											
Interrupt Acknowledge											
1	0100	FFFFFE00 <sub>16</sub>	0	1	1	1	0	X	X	X	X
Interrupt Return											
1	0110	FFFFFE00 <sub>16</sub>	0	1	1	1	0	X	X	X	X
C. Vectored Interrupt Sequences											
Interrupt Acknowledge											
1	0100	FFFFFE00 <sub>16</sub>	0	1	1	1	0	X	X	X	Vector: Range: 11 <sub>16</sub> –1F <sub>16</sub>
Interrupt Return											
1	0110	FFFFFE00 <sub>16</sub>	0	1	1	1	0	X	X	X	IVCT Register Content

### 3.6.5 Bus Retry

The NS32GX320 has the capability of handling errors occurring during the execution of a bus cycle.

When an error occurs, the CPU may be requested to repeat the erroneous bus cycle. The request is done by asserting the BRT signal. BRT is sampled at the end of state T2 or T2B.

When the CPU detects that BRT is active, it completes the bus cycle normally, but ignores the data read in case of a read cycle, and maintains a copy of the data to be written in case of a write cycle. Then, after a delay of two clock cycles, it will start executing the bus cycle again.

If the transfer cycle is multiple (e.g., for non-aligned data), only the problematic part will be repeated.

For instance, if a non-aligned double-word is being transferred and the second half of the transfer fails, only the second part will be repeated.

The same applies for a retry during a burst sequence. The repeated cycle will begin where the read operation failed (rather than the first address of the burst) and will finish the original burst.

Figures 3-33 and 4-10 (in Section 4) show the BRT timing for a basic access cycle and for burst cycles respectively.

The CPU always waits for BRT to be HIGH before repeating the bus cycle. While BRT is LOW, the CPU places all the output signals shown in Figure 4-11 in a TRI-STATE® condition.

### 3.6.6 Dynamic Bus Configuration

The NS32GX320 is tuned to operate with 32-bit wide memory and peripheral devices. The bus also supports 8-bit and 16-bit data widths, but at reduced efficiency. The CPU can switch from one bus width to another dynamically; the only restriction is that the bus width cannot change for locations within an aligned 16-byte block.

The CPU determines the bus width in effect for a bus cycle by using the values of the BW0 and BW1 signals sampled during the last T2 state. Values of BW0 and BW1 sampled before the last T2 state or during T2B states are ignored. Whenever a bus width other than 32-bit is detected by the CPU, two idle states are inserted before the next bus cycle is initiated. These idle states are only inserted once during an operand access, even if more than two bus cycles are needed to complete the access.

### 3.0 Functional Description (Continued)

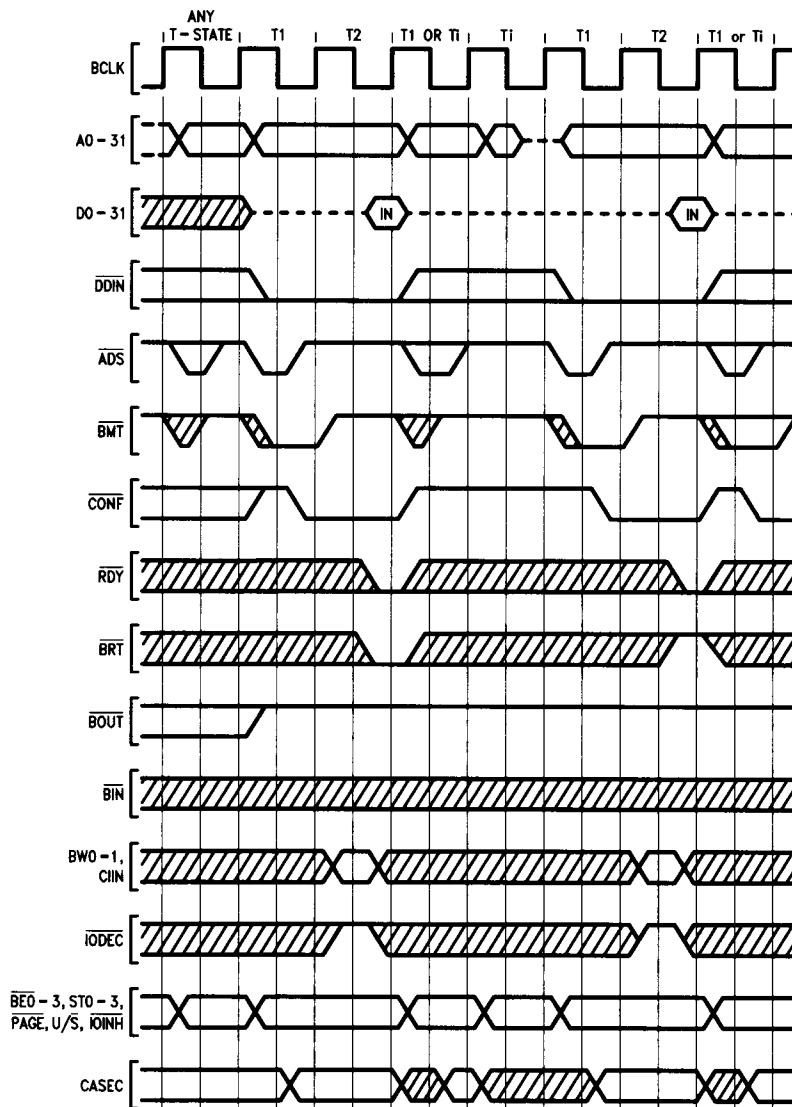


FIGURE 3-33. Bus Retry During a Basic Read Cycle

TL/EE/10564-37

### 3.0 Functional Description (Continued)

The various combinations for BW0 and BW1 are shown below.

BW1	BW0	
0	0	Reserved
0	1	8-Bit Bus
1	0	16-Bit Bus
1	1	32-Bit Bus

The bus width is always 32 bits during slave cycles (see Section 3.6.4.8). An important feature of the NS32GX320 is that it does not impose any restrictions on the data alignment, regardless of the bus width.

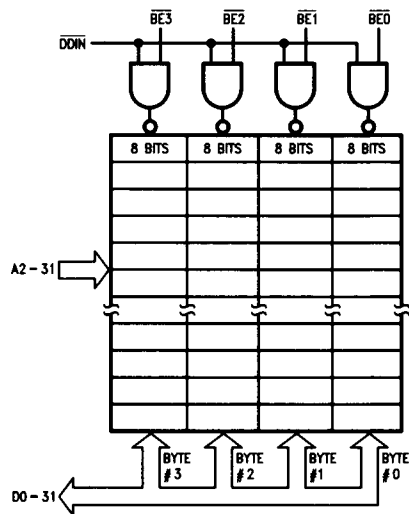
Bus accesses are performed in double-word units. Accesses of data operands that cross double-word boundaries are decomposed into two or more aligned double-word accesses.

The CPU provides four byte enable signals ( $\overline{BE}0-3$ ) which facilitate individual byte accessing on either a 32-bit or a 16-bit bus.

Figures 3-34 and 3-35 show the basic interfaces for 32-bit and 16-bit memories. An 8-bit memory interface (not shown) is even simpler since it does not use any of the  $\overline{BE}0-3$  signals and its single bank is always enabled whenever the memory is selected. Each byte location in this case is selected by address bits A0-31.

The NS32GX320 does not keep track of the bus width used in previous instruction fetches or data accesses. At the beginning of every memory transaction, the CPU always assumes that the bus is 32-bit wide and the  $\overline{BE}0-3$  signals are activated accordingly.

The  $\overline{BOUT}$  signal is also asserted during instruction fetches or data reads if the conditions for bursting are satisfied. If the bus is other than 32-bit wide, the  $\overline{BIN}$  signal is ignored and  $\overline{BOUT}$  is deasserted at the beginning of the TRI-STATE following T2, since burst cycles are not allowed for 8-bit or 16-bit buses.

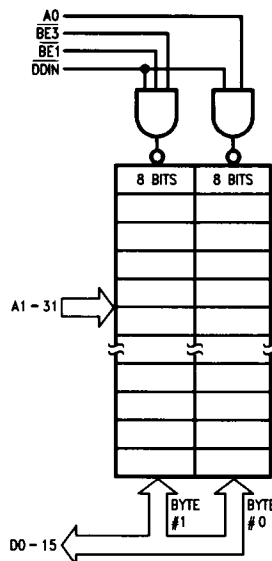


TL/EE/10564-38

FIGURE 3-34. Basic Interface for 32-Bit Memories

The following subsections provide detailed descriptions of the access sequences performed in the various cases.

**Note:** Although the NS32GX320 ignores the  $\overline{BIN}$  signal during DMA cycles, and for 8-bit and 16-bit bus widths, it is recommended that  $\overline{BIN}$  be asserted only if the system supports burst transfers. This is to ensure compatibility with future versions of the NS32GX320 that might support burst transfers in the above cases.



TL/EE/10564-39

FIGURE 3-35. Basic Interface for 16-Bit Memories

#### 3.6.6.1 Instruction Fetch Sequences

The CPU performs two types of instruction fetch cycles: sequential and non-sequential. These can be distinguished from each other by the differing status combinations on pins ST0-4. For non-sequential instruction fetches the CPU presents on the address bus the exact byte address of the first instruction in the instruction stream that is about to begin; for sequential instruction fetches, the address of the next aligned instruction double-word is presented on the address bus. The CPU always activates all byte enable signals ( $\overline{BE}0-3$ ) for both sequential and non-sequential fetches.  $\overline{BOUT}$  is also asserted during T2 if the addressed double-word is not the last in an aligned 16-byte block. Tables 3-8 to 3-10 show the fetch sequence for the various bus widths.

#### 32-Bit Bus Width

The CPU reads the entire double-word present on the data bus into its internal instruction buffer.

If  $\overline{BOUT}$  and  $\overline{BIN}$  are both active, the CPU reads up to 3 consecutive double-words using burst cycles. Burst cycles are used for instruction fetches regardless of whether the accesses are cacheable.

### 3.0 Functional Description (Continued)

Example: JUMP @5

- The CPU performs a fetch cycle at address 5 with  $\overline{BE}0-3$  all active.
- Two burst cycles are then performed and addresses 8 and 12 are output while  $\overline{BE}0-3$  are kept active.

#### 16-Bit Bus Width

The word on the least-significant half of the data bus is read by the CPU. This is either the even or the odd word within the required instruction double-word, as determined by address bit 1.

The CPU then complements address bit 1, clears address bit 0 and initiates a bus cycle to read the other word, while keeping all the  $\overline{BE}0-3$  signals active.

These two words are then assembled into a double-word and transferred into the instruction buffer.

In case of a non-sequential fetch, if the access is not cacheable and the instruction address selects the odd word within the instruction double-word, the even word is not fetched.

Example JUMP @6

- A fetch cycle is performed at address 6 with  $\overline{BE}0-3$  all active.
- The word at address 4 is then fetched if the access is cacheable.

#### 8-Bit Bus Width

The instruction byte on the bus lines D0-7 is fetched. The CPU performs three consecutive cycles to read the remaining bytes within the required double-word, while keeping  $\overline{BE}0-3$  all active. The 4 bytes are then assembled into a double-word and transferred into the instruction buffer. For a non-sequential fetch, if the access is not cacheable, the CPU will only read the upper bytes within the instruction double-word starting with the byte at the instruction address.

Example: JUMP @7

- The CPU performs a fetch cycle at address 7 with  $\overline{BE}0-3$  all active.
- Bytes at addresses 4, 5 and 6 are then fetched consecutively if the access is cacheable.

**TABLE 3-8. Cacheable/Non-Cacheable Instruction Fetches from a 32-Bit Bus**

1. In a burst access four bytes are fetched with the L.S. bits of the address set to 00.
2. A 'C' on the data bus refers to cacheable fetches and indicates that the byte is placed in the instruction cache. An 'I' refers to non-cacheable fetches and indicates that the byte is ignored.

Number of Bytes	Address LSB	Bytes to be Fetched				Address Bus	$\overline{BE}0-3$	Data Bus			
1	11	B0	—	—	—	A	L L L L	B0	C/I	C/I	C/I
2	10	B1	B0	—	—	A	L L L L	B1	B0	C/I	C/I
3	01	B2	B1	B0	—	A	L L L L	B2	B1	B0	C/I
4	00	B3	B2	B1	B0	A	L L L L	B3	B2	B1	B0

**TABLE 3-9. Cacheable/Non-Cacheable Instruction Fetches from a 16-Bit Bus**

1. A bus access marked with '\*' in the 'Address Bus' column is performed only if the fetch is cacheable.

Number of Bytes	Address LSB	Bytes to be Fetched				Address Bus	$\overline{BE}0-3$	Data Bus			
1	11	B0	—	—	—	A *A - 3	L L L L L L L L	—	—	B0 C	C/I C
2	10	B1	B0	—	—	A *A - 2	L L L L L L L L	—	—	B1 C	B0 C
3	01	B2	B1	B0	—	A A + 1	L L L L L L L L	—	—	B0 B2	C/I B1
4	00	B3	B2	B1	B0	A A + 2	L L L L L L L L	—	—	B1 B3	B0 B2

## 3.0 Functional Description (Continued)

TABLE 3-10. Cacheable/Non-Cacheable Instruction Fetches from an 8-Bit Bus

Number of Bytes	Address LSB	Bytes to be Fetched	Address Bus	BE0-3	Data Bus
1	11	B0 — — —	A * A - 3 * A - 2 * A - 1	LLLL LLLL LLLL LLLL	— — — B0 — — — C — — — C — — — C
2	10	B1 B0 — —	A A + 1 * A - 2 * A - 1	LLLL LLLL LLLL LLLL	— — — B0 — — — B1 — — — C — — — C
3	01	B2 B1 B0 —	A A + 1 A + 2 * A - 1	LLLL LLLL LLLL LLLL	— — — B0 — — — B1 — — — B2 — — — C
4	00	B3 B2 B1 B0	A A + 1 A + 2 A + 3	LLLL LLLL LLLL LLLL	— — — B0 — — — B1 — — — B2 — — — B3

### 3.6.6.2 Data Read Sequences

The CPU starts a data read access by placing the exact address of the operand on the address bus. The byte enable lines are activated to select only the bytes required by the instruction being executed. This prevents spurious accesses to peripheral devices that might be sensitive to read accesses, such as those which exhibit the characteristic of destructive reading. If the on-chip data cache is internally enabled for the read access, the  $\overline{BOUT}$  signal is asserted at the beginning of state T2.  $\overline{BOUT}$  will be deasserted if the data cache is externally inhibited (through CIIN or IODEC), or the bus width is other than 32 bits. During cacheable accesses the CPU always reads all the bytes in the double-word, whether or not they are needed to execute the instruction, and stores them into the data cache. The external memory, in this case, must place the data on the bus regardless of the state of the byte enable signals.

If the data cache is either internally or externally inhibited during the access, the CPU ignores the bytes not selected by the  $\overline{BE}0-3$  signals. Data read sequences for the various bus widths are shown in tables 3-11 to 3-13.

#### 32-Bit Bus Width

The entire double-word present on the bus is read by the CPU. If the access is cacheable and the memory allows burst accesses, the CPU reads up to 3 additional double-words within the aligned 16-byte block containing the first byte of the operand. These burst accesses are performed in a wrap-around fashion within the 16-byte block.

Example: MOVW @5, R0

- The CPU reads a double-word at address 5 while keeping  $\overline{BE}1$  and  $\overline{BE}2$  active.
- If the access is not-cacheable,  $\overline{BOUT}$  is deasserted and the data bytes 0 and 3 are ignored.
- If the access is cacheable, the CPU performs burst cycles with  $\overline{BE}0-3$  all active, to read the double-words at addresses 8, 12, and 0.

#### 16-Bit Bus Width

The word on the least-significant half of the data bus is read by the CPU. The CPU can then perform another access cycle with address bit 1 complemented and address bit 0 cleared to read the other word within the addressed double-word.

If the access is cacheable, the entire double-word is read and stored into the cache.

If the access is not cacheable, the CPU ignores the bytes in the double-word not selected by  $\overline{BE}0-3$ . In this case, the second access cycle is not performed, unless selected bytes are contained in the second word.

Example: MOVW @5, R0

- The CPU reads a word at address 5 while keeping  $\overline{BE}1$  active.
- If the access is not cacheable, the CPU ignores byte 0.
- If the access is cacheable, the CPU performs another access cycle, with  $\overline{BE}0-3$  all active, to read the word at address 6.

#### 8-Bit Bus Width

The data byte on the bus lines D0-7 is read by the CPU. The CPU can then perform up to 3 access cycles to read the remaining bytes in the double-word.

If the access is cacheable, the entire double-word is read and stored into the cache.

If the access is not cacheable, the CPU will only perform those access cycles needed to read the selected bytes.

Example: MOVW @5, R0

- The CPU reads the byte at address 5 while keeping  $\overline{BE}1$  and  $\overline{BE}2$  active.
- If the access is not cacheable, the CPU activates  $\overline{BE}2$  and reads the byte at address 6.
- If the access is cacheable, the CPU performs three bus cycles with  $\overline{BE}0-3$  all active, to read the bytes at addresses 6, 7 and 4.

### 3.0 Functional Description (Continued)

**TABLE 3-11. Cacheable/Non-Cacheable Data Reads from a 32-Bit Bus**

1. In a burst access four bytes are read with the L.S. bits of the address set to 00.
2. A 'C' on the data bus refers to cacheable reads and indicates that the byte is placed in the data cache. An 'I' refers to non-cacheable reads and indicates that the byte is ignored.

Number of Bytes	Address LSB	Bytes to be Read	Address Bus	BE0-3	Data Bus
1	00	— — — B0	A	HHHL	C/I C/I C/I B0
1	01	— — B0 —	A	HHLH	C/I C/I B0 C/I
1	10	— B0 — —	A	HLHH	C/I B0 C/I C/I
1	11	B0 — — —	A	LHHH	B0 C/I C/I C/I
2	00	— — B1 B0	A	HHLL	C/I C/I B1 B0
2	01	— B1 B0 —	A	HLLH	C/I B1 B0 C/I
2	10	B1 B0 — —	A	LLHH	B1 B0 C/I C/I
3	00	— B2 B1 B0	A	HLLL	C/I B2 B1 B0
3	01	B2 B1 B0 —	A	LLLH	B2 B1 B0 C/I
4	00	B3 B2 B1 B0	A	LLLL	B3 B2 B1 B0

**TABLE 3-12. Cacheable/Non-Cacheable Data Reads from a 16-Bit Bus**

1. A bus access marked with '\*' in the 'Address Bus' column is performed only if the read is cacheable.

Number of Bytes	Address LSB	Data to be Read	Address Bus	BE0-3		Data Bus
				Cach.	Non Cach.	
1	00	— — — B0	A * A + 2	HHHL LLLL	HHHL	— — C/I B0 — — C C
1	01	— — B0 —	A * A + 1	HHLH LLLL	HHLH	— — B0 C/I — — C C
1	10	— B0 — —	A * A - 2	HLHH LLLL	HLHH	— — C/I B0 — — C C
1	11	B0 — — —	A * A - 3	LHHH LLLL	LHHH	— — B0 C/I — — C C
2	00	— — B1 B0	A * A + 2	HHLL LLLL	HHLL	— — B1 B0 — — C C
2	01	— B1 B0 —	A A + 1	HLLH LLLL	HLLH HLHH	— — B0 C/I — — C/I B1
2	10	B1 B0 — —	A * A - 2	LLHH LLLL	LLHH	— — B1 B0 — — C C
3	00	— B2 B1 B0	A A + 2	HLLL LLLL	HLLL HLHH	— — B1 B0 — — C/I B2
3	01	B2 B1 B0 —	A A + 1	LLLH LLLL	LLLH LLHH	— — B0 C/I — — B2 B1
4	00	B3 B2 B1 B0	A A + 2	LLLL LLLL	LLLL LLHH	— — B1 B0 — — B3 B2



## 3.0 Functional Description (Continued)

TABLE 3-13. Cacheable/Non-Cacheable Data Reads from an 8-Bit Bus D8-12

Number of Bytes	Address LSB	Data to be Read	Address Bus	BE0-3		Data Bus
				Cach.	Non Cach.	
1	00	— — — B0	A *A + 1 *A + 2 *A + 3	HHHL LLLL LLLL LLLL	HHHL	— — — B0 — — — C — — — C — — — C
1	01	— — B0 —	A *A + 1 *A + 2 *A - 1	HHLH LLLL LLLL LLLL	HHLH	— — — B0 — — — C — — — C — — — C
1	10	— B0 — —	A *A + 1 *A - 2 *A - 1	HLHH LLLL LLLL LLLL	HLHH	— — — B0 — — — C — — — C — — — C
1	11	B0 — — —	A *A - 3 *A - 2 *A - 1	LHHH LLLL LLLL LLLL	LHHH	— — — B0 — — — C — — — C — — — C
2	00	— — B1 B0	A A + 1 *A + 2 *A + 3	HHLL LLLL LLLL LLLL	HHLL HHLH	— — — B0 — — — B1 — — — C — — — C
2	01	— B1 B0 —	A A + 1 *A + 2 *A - 1	HLLH LLLL LLLL LLLL	HLLH HLHH	— — — B0 — — — B1 — — — C — — — C
2	10	B1 B0 — —	A A + 1 *A - 2 *A - 1	LLHH LLLL LLLL LLLL	LLHH LHHH	— — — B0 — — — B1 — — — C — — — C
3	00	— B2 B1 B0	A A + 1 A + 2 *A + 3	HLLL LLLL LLLL LLLL	HLLL HLLH HLHH	— — — B0 — — — B1 — — — B2 — — — C
3	01	B2 B1 B0 —	A A + 1 A + 2 *A - 1	LLLH LLLL LLLL LLLL	LLLH LLHH LHHH	— — — B0 — — — B1 — — — B2 — — — C
4	00	B3 B2 B1 B0	A A + 1 A + 2 A + 3	LLLL LLLL LLLL LLLL	LLLL LLLH LLHH LHHH	— — — B0 — — — B1 — — — B2 — — — B3

### 3.6.6.3 Data Write Sequences

In a write access the CPU outputs the operand address and asserts only the byte enable lines needed to select the specific bytes to be written.

In addition, the CPU duplicates the data to be written on the appropriate bytes of the data bus in order to handle 8-bit and 16-bit buses.

The various access sequences as well as the duplication of data are summarized in Tables 3-14 to 3-16.

### 32-Bit Bus Width

The CPU performs only one access cycle to write the selected bytes within the addressed double-word.

Example: MOV B R0, @6

- The CPU duplicates byte 2 of the data bus into byte 0 and performs a write cycle at address 6 with BE2 active.

### 16-Bit Bus Width

Up to two access cycles are needed to complete the write operation.

### 3.0 Functional Description (Continued)

Example: MOVW R0, @5

- The CPU duplicates byte 1 of the data bus into byte 0 and performs a write cycle at address 5 with  $\overline{BE}1$  and  $\overline{BE}2$  active.
- A write at address 6 is then performed with  $\overline{BE}2$  active and the original byte 2 of the data bus placed on byte 0.

#### 8-Bit Bus Width

Up to 4 access cycles are needed in this case to complete the write operation.

Example: MOVB R0, @7

- The CPU duplicates byte 3 of the data bus into bytes 0 and 1, and then performs a write cycle at address 7 with  $\overline{BE}3$  active.

#### 3.6.7 Bus Access Control

The NS32GX320 has the capability of relinquishing its control of the bus upon request from an external DMA device or another CPU. This capability is implemented with the  $\overline{HOLD}$

and  $\overline{HLDA}$  signals. By asserting  $\overline{HOLD}$ , an external device requests access to the bus. On receipt of  $\overline{HLDA}$ , the device may perform bus cycles, as the CPU at this point has placed all the output signals shown in *Figure 3-36* into the TRI-STATE condition.

To return control of the bus to the CPU, the external device sets  $\overline{HOLD}$  inactive, and the CPU acknowledges return of the bus by setting  $\overline{HLDA}$  inactive.

The CPU samples  $\overline{HOLD}$  at the beginning of each TRI-STATE on the rising edge of BCLK. If  $\overline{HOLD}$  is asserted when the bus is idle between access sequences, then the bus is granted immediately (see *Figure 3-36*). If  $\overline{HOLD}$  is asserted during an access sequence, then the bus is granted immediately after the access sequence, including any retried bus cycles, has completed (see *Figure 4-15*). Note that an access sequence can be composed of several bus cycles if the bus width is 8 or 16 bits.

TABLE 3-14. Data Writes to a 32-Bit Bus

1. Bytes on the data bus marked with '•' are undefined.

Number of Bytes	Address LSB	Data to be Written	Address Bus	$\overline{BE}0-3$	Data Bus
1	00	— — — B0	A	H H H L	• • • B0
1	01	— — B0 —	A	H H L H	• • B0 B0
1	10	— B0 — —	A	H L H H	• B0 • B0
1	11	B0 — — —	A	L H H H	B0 • B0 B0
2	00	— — B1 B0	A	H H L L	• • B1 B0
2	01	— B1 B0 —	A	H L L H	• B1 B0 B0
2	10	B1 B0 — —	A	L L H H	B1 B0 B1 B0
3	00	— B2 B1 B0	A	H L L L	• B2 B1 B0
3	01	B2 B1 B0 —	A	L L L H	B2 B1 B0 B0
4	00	B3 B2 B1 B0	A	L L L L	B3 B2 B1 B0

TABLE 3-15. Data Writes to a 16-Bit Bus

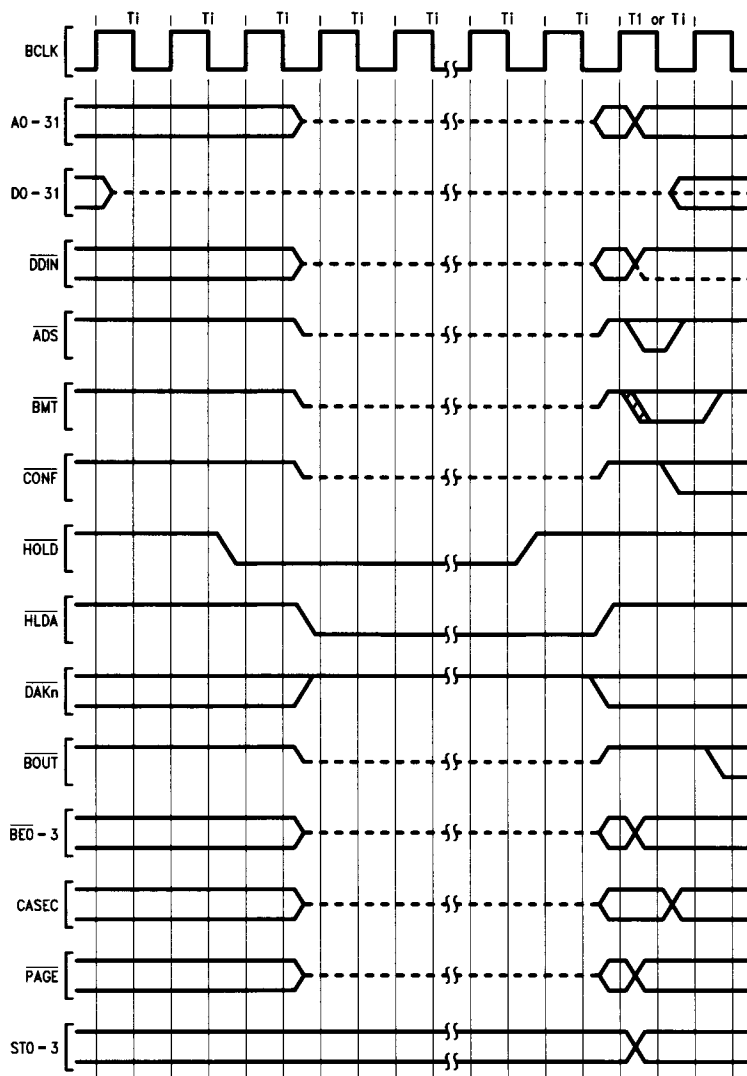
Number of Bytes	Address LSB	Data to be Written	Address Bus	$\overline{BE}0-3$	Data Bus
1	00	— — — B0	A	H H H L	• • • B0
1	01	— — B0 —	A	H H L H	• • B0 B0
1	10	— B0 — —	A	H L H H	• B0 • B0
1	11	B0 — — —	A	L H H H	B0 • B0 B0
2	00	— — B1 B0	A	H H L L	• • B1 B0
2	01	— B1 B0 —	A A + 1	H L L H H L H H	• B1 B0 B0 • • • B1
2	10	B1 B0 — —	A	L L H H	B1 B0 B1 B0
3	00	— B2 B1 B0	A A + 2	H L L L H L H H	• B2 B1 B0 • • • B2
3	01	B2 B1 B0 —	A A + 1	L L L H L L H H	B2 B1 B0 B0 • • B2 B1
4	00	B3 B2 B1 B0	A A + 2	L L L L L L H H	B3 B2 B1 B0 • • B3 B2

### 3.0 Functional Description (Continued)

**TABLE 3-16. Data Writes to an 8-Bit Bus**

Number of Bytes	Address LSB	Data to be Written	Address Bus	$\overline{BE}0-3$	Data Bus
1	00	— — — B0	A	H H H L	• • • B0
1	01	— — B0 —	A	H H L H	• • B0 B0
1	10	— B0 — —	A	H L H H	• B0 • B0
1	11	B0 — — —	A	L H H H	B0 • B0 B0
2	00	— — B1 B0	A A + 1	H H L L H H L H	• • B1 B0 • • • B1
2	01	— B1 B0 —	A A + 1	H L L H H L H H	• B1 B0 B0 • • • B1
2	10	B1 B0 — —	A A + 1	L L H H L H H H	B1 B0 B1 B0 • • • B1
3	00	— B2 B1 B0	A A + 1 A + 2	H L L L H L L H H L H H	• B2 B1 B0 • • • B1 • • • B2
3	01	B2 B1 B0 —	A A + 1 A + 2	L L L H L L H H L H H H	B2 B1 B0 B0 • • • B1 • • • B2
4	00	B3 B2 B1 B0	A A + 1 A + 2 A + 3	L L L L L L L H L L H H L H H H	B3 B2 B1 B0 • • • B1 • • • B2 • • • B3

### 3.0 Functional Description (Continued)



TL/EE/10564-40

**FIGURE 3-36. Hold Acknowledge (Bus Initially Idle)**

**Note:** The status indicates 'IDLE' while the bus is granted. If the cause of the IDLE changes (e.g., CPU starts waiting for an interrupt), the status also changes.

The CPU will never grant the bus between interlocked read and write bus cycles.

**Note:** If an external device requires a very short latency to get control of the bus, the bus retry signal (BRT) can be used instead of hold. See Section 3.6.5.

#### 3.6.8 Interfacing Memory-Mapped I/O Devices

In Section 3.1.3.2 it was mentioned that some special precautions are needed when interfacing I/O devices to the NS32GX320 due to its internal pipelined implementation. Two special signals are provided for this purpose:  $\overline{\text{IOINH}}$  and  $\overline{\text{IODEC}}$ . The CPU asserts  $\overline{\text{IOINH}}$  during a read bus cycle to indicate that the bus cycle should be ignored if an I/O device is selected. The system responds by asserting

$\overline{\text{IODEC}}$  to indicate to the CPU that an I/O device has been selected.  $\overline{\text{IODEC}}$  is sampled by the CPU in the middle of state T2. If the cycle is extended, then the CPU uses the  $\overline{\text{IODEC}}$  value sampled during the last wait state. If a bus retry occurs, the sampled  $\overline{\text{IODEC}}$  value is ignored.  $\overline{\text{IODEC}}$  is ignored during burst transfer cycles.

When  $\overline{\text{IODEC}}$  is active during a bus cycle for which  $\overline{\text{IOINH}}$  is asserted, the CPU discards the data and applies the special handling required for I/O devices. The CPU also adds idle states (at least one) after the transaction, regardless of  $\overline{\text{IOINH}}$ , as long as the  $\overline{\text{IODEC}}$  input is active. Deasserting  $\overline{\text{IODEC}}$  during an idle state allows the CPU to start a new transaction on the next clock cycle.

### 3.0 Functional Description (Continued)

During the idle states the A0–31,  $\overline{\text{DDIN}}$ ,  $\overline{\text{BE0}}\text{--}3$ ,  $\overline{\text{PAGE}}$ , and  $\overline{\text{U/S}}$  signals have the same value they had on the last transaction. If the last bus reference was a write, the value of D0–D31 is also unchanged.

In case of I/O transactions involving two or more bus cycles (bus width smaller than the operand's length), extra idle states can be added only on the last access. In this case the CPU inserts 2 idle cycles after the first accesses and then performs the remaining accesses back-to-back without extra idles, regardless of  $\overline{\text{IODEC}}$ . If on the last access  $\overline{\text{IODEC}}$  is sampled active, extra idle states are added as long as  $\overline{\text{IODEC}}$  is active. I/O devices requiring this "Slow Peripheral" support should always be accessed by specifying operand lengths smaller than or equal to the physical bus width of the device.

Figures 3-37, 3-38 and 3-39 show the timing diagrams for simple read and write accesses as well as for a multi-cycle

read access. Figure 3-40 shows a possible implementation of an I/O device interface where the address mapping of the I/O devices is fixed.

In an open system configuration,  $\overline{\text{IODEC}}$  could be generated by the decoding logic of each I/O device subsystem.

**Note 1:** When  $\overline{\text{IODEC}}$  is active in response to a read bus cycle, the CPU treats the reference as noncacheable.

**Note 2:**  $\overline{\text{IOINH}}$  is kept inactive during write cycles.

**Note 3:** If the CPU samples both  $\overline{\text{HOLD}}$  and  $\overline{\text{IODEC}}$  active,  $\overline{\text{IODEC}}$  has higher priority, and the  $\overline{\text{HOLD}}$  request will be acknowledged after  $\overline{\text{IODEC}}$  is sampled inactive.

**Note 4:** If  $\overline{\text{IODEC}}$  is sampled active in the last T2 state of a bus cycle together with  $\overline{\text{BRT}}$  active,  $\overline{\text{BRT}}$  has the higher priority. In this case the NS32GX320 ignores  $\overline{\text{IODEC}}$ .

**Note 5:** On interlocked bus cycles idle states can be added after each bus access. If  $\overline{\text{IODEC}}$  is sampled active in the last T2 state of the last write reference,  $\overline{\text{ILO}}$  will be deasserted regardless of the value of  $\overline{\text{IODEC}}$ . Idle states will be added until  $\overline{\text{IODEC}}$  is sampled high.

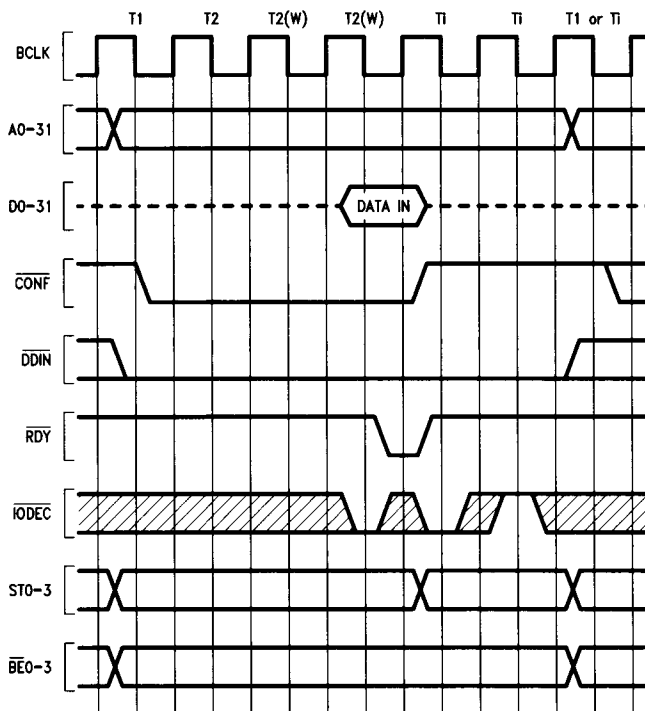


FIGURE 3-37. Read Cycle from a Memory-Mapped I/O Device

TL/EE/10564-42

### 3.0 Functional Description (Continued)

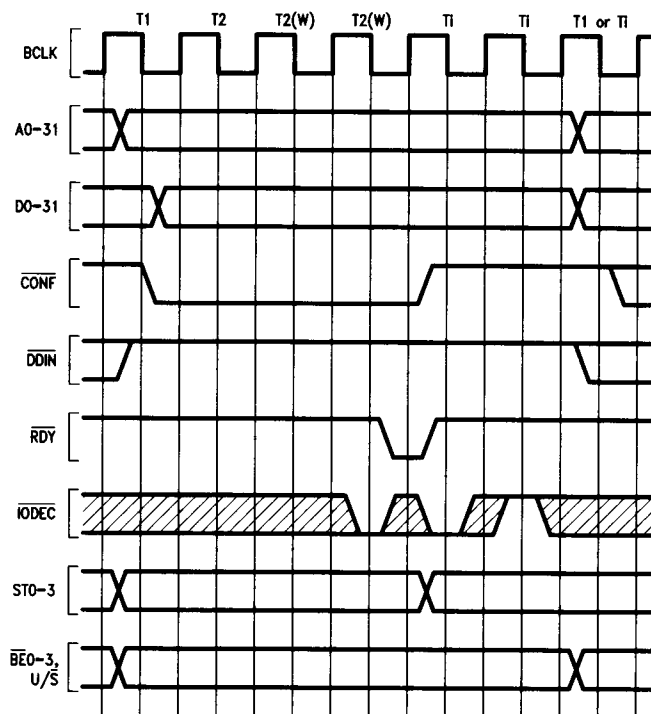
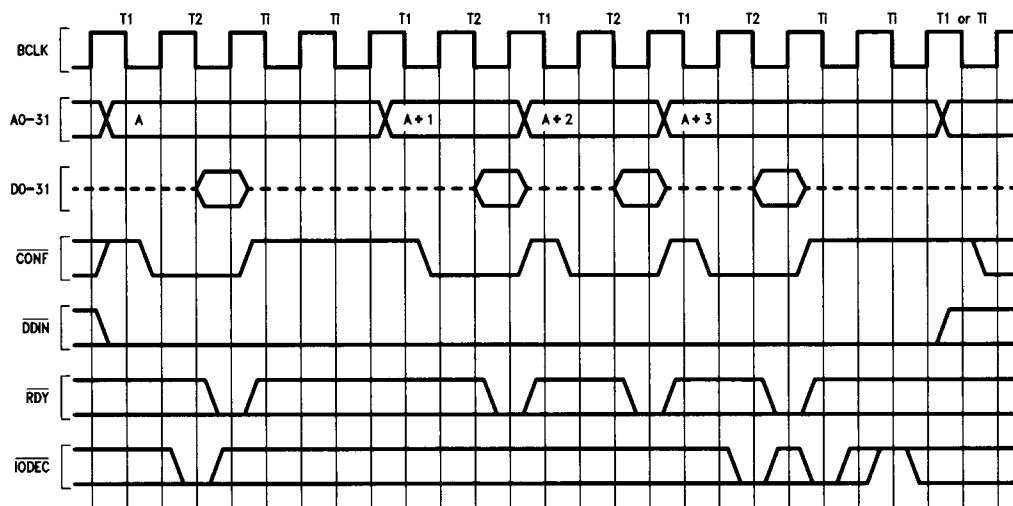


FIGURE 3-38. Write Cycle to a Memory-Mapped I/O Device

TL/EE/10564-43

### 3.0 Functional Description (Continued)

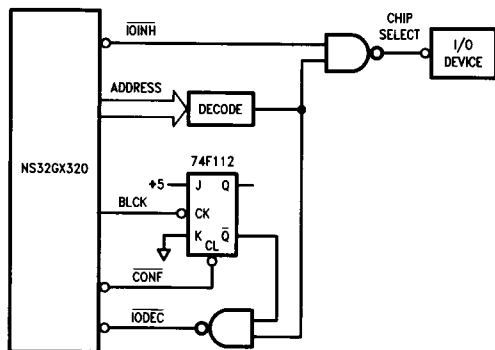


TL/EE/10564-44

**Note:** Two idle states are added after the first reference due to the change in the bus width to 8 bits. Three more bus cycles are needed to complete the double-word reference. These bus cycles are executed back-to-back regardless of the  $\overline{\text{IODEC}}$  state. Idle states can be added after the last reference using  $\overline{\text{IODEC}}$ .

**FIGURE 3-39. Multi-Cycle Read from an 8-Bit Memory-Mapped I/O Device**

### 3.0 Functional Description (Continued)



TL/EE/10564-45

FIGURE 3-40. Typical I/O Device Interface

#### 3.6.9 Interrupt and Debug Trap Requests

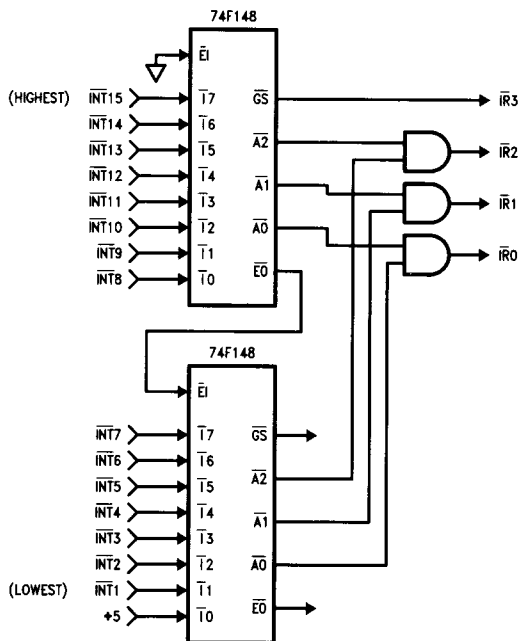
Six signals are provided by the NS32GX320 to externally request interrupts and/or a debug trap.  $\overline{IR}0-3$  and  $\overline{NMI}$  are for maskable and non-maskable interrupts respectively.  $\overline{DBG}$  is used for requesting an external debug trap. All of these signals are sampled on every rising edge of BCLK.

$\overline{IR}0-3$  are level sensitive and, as such, once asserted they must be kept asserted until acknowledged. They can be asynchronous to BCLK, since the NS32GX320 internally synchronizes them. Nevertheless, if  $\overline{IR}0-3$  meet the required setup and hold times, then they are recognized deterministically. The on-chip synchronization circuitry compares the values of  $\overline{IR}0-3$  sampled in two consecutive BCLK edges. An interrupt request that is held constant for two consecutive edges is considered valid. The sampled value of  $\overline{IR}0-3$  indicates an external interrupt request at the encoded priority. When  $\overline{IR}0-3$  are all high, then no external interrupt is requested. When they are all low, then a level-15 request is generated.

$\overline{NMI}$  and  $\overline{DBG}$  are both edge sensitive; a high-to-low transition is detected by the CPU and stored in an internal latch, so that there is not need to keep these signals asserted until the request is acknowledged.  $\overline{NMI}$  and  $\overline{DBG}$  can be asserted asynchronously to BCLK, but they should be at least 2 clock cycles wide in order to be recognized.

If  $\overline{NMI}$  and  $\overline{DBG}$  meet the specified setup and hold times, they will be recognized on the rising edge of BCLK deterministically.

Refer to Figures 4-20 and 4-21 for more details on the timing of the above signals.



TL/EE/10564-47

FIGURE 3-41. Interrupt Request Encoding Logic



## 3.0 Functional Description (Continued)

### 3.6.10 Internal Status

The NS32GX320 provides information on the system interface concerning its internal activity.

The  $U/\overline{S}$  signal will indicate the state of the U bit in the PSR.

The  $PFS$  signal is asserted for one BCLK cycle when the CPU begins executing a new instruction. The  $\overline{BP}$  signal is asserted for one BCLK cycle when an address-compare or PC-match condition is detected. If the  $\overline{BP}$  signal is asserted one BCLK cycle after  $PFS$ , it indicates that an address-compare debug condition has been detected. If  $\overline{BP}$  is asserted at any other time, it indicates that a PC-Match debug condition has been detected.

While executing a CINV instruction, the CPU displays the operation code and source operand using slave processor write bus cycles.

During idle bus cycles, the signals ST0–ST3 indicate whether the CPU is waiting for an interrupt or is waiting for a Slave Processor to complete executing an instruction.

### 3.6.11 Page-Mode and Static-Column DRAM Support

The access time in systems using Page-Mode or Static-Column DRAMs can be shortened on consecutive references to the same DRAM page. The NS32GX320 provides circuitry to detect whether the current access is to the same 8 kbyte page as the previous access. The  $PAGE$  signal is asserted when the above condition is met. The external logic uses the  $\overline{PLAT}$  signal to indicate to the CPU whether to latch the current address for such comparisons. The  $PAGE$  signal can be used even for DRAMs with smaller page by latching and comparing 1 or 2 bits of the address externally. The  $\overline{PLAT}$  signal must not change within an aligned 16-byte block.

When  $PAGE$  is active together with  $\overline{ADS}$ , it indicates that the current address is to the same page as the last valid latched address. Addresses are latched when  $\overline{PLAT}$  is asserted on the last T2 state of the bus access. Whenever a new address is latched a valid bit is set to indicate a valid address in the latch. The valid bit is reset whenever the CPU relinquishes the bus due to HOLD acknowledge, Extended-Retry, or Reset. The reason for invalidating the address is that another bus master may perform memory references to a page different from the one addressed in the last CPU transaction. After clearing the valid bit, the CPU will not assert  $PAGE$  unless a new address is latched using  $\overline{PLAT}$ . The  $PAGE$  signal is floated whenever the NS32GX320 releases the bus.

**Note 1:** The  $\overline{PAGE}$  signal may toggle at the beginning of T1, before the time in which it is guaranteed valid.

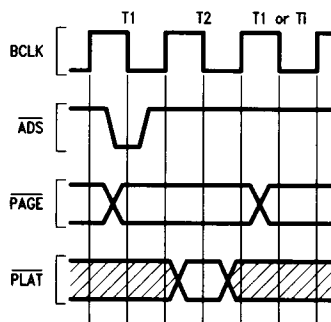
**Note 2:** On multi-cycle references to 8- and 16-bit wide buses,  $PAGE$  is evaluated for each bus access.

A new address is latched using  $\overline{PLAT}$  only in the last T2 state of the first bus cycle.

**Note 3:** On burst accesses the  $PAGE$  signal is set during T1 of the opening cycle and does not change throughout the transaction.

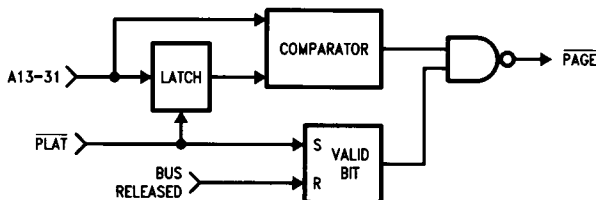
A new address is latched using  $\overline{PLAT}$  only in the last T2 state of the opening cycle.

**Note 4:**  $\overline{PLAT}$  is not sampled during Slave Processor Bus cycles.



TL/EE/10564-48

FIGURE 3-42.  $PAGE$  and  $\overline{PLAT}$  Signals Timing



TL/EE/10564-49

FIGURE 3-43.  $PAGE$  Signal Generation Logic

## 4.0 Device Specifications

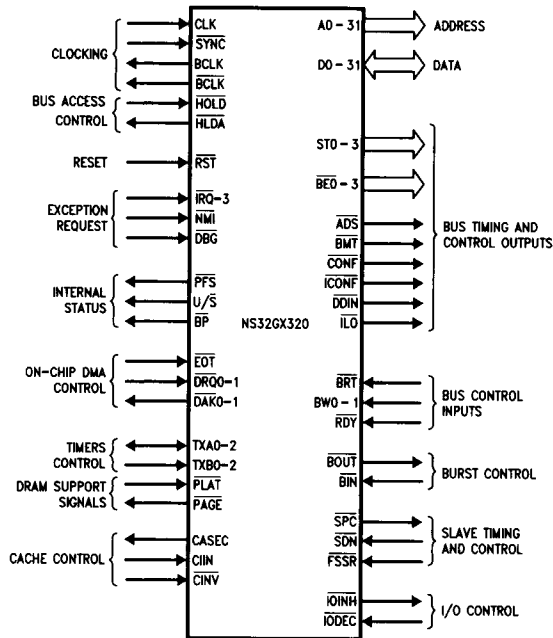


FIGURE 4-1. NS32GX320 Interface Signals

TL/EE/10564-50

### 4.1 NS32GX320 PIN DESCRIPTIONS

Descriptions of the NS32GX320 pins are given in the following sections.

Included are also references to portions of the functional description, Section 3.

Figure 4-1 shows the NS32GX320 interface signals grouped according to related functions.

**Note:** An asterisk next to the signal name indicates a TRI-STATE condition for that signal when **HOLD** is acknowledged or during an extended retry.

#### 4.1.1 Supplies

<b>VCC</b>	<b>Power.</b> ±5V positive supply.
<b>GND</b>	<b>Ground.</b> Ground reference for both on-chip logic and output drivers.

#### 4.1.2 Input Signals

<b>CLK</b>	<b>Clock.</b> Input Clock used to derive all CPU Timing.
<b>SYNC</b>	<b>Synchronize.</b> When <b>SYNC</b> is active, <b>BCLK</b> will stop toggling. This signal can be used to synchronize two or more CPUs (Section 3.6.2). <b>Note:</b> If <b>SYNC</b> is not used, it should be pulled up to $V_{CC}$ .

#### **HOLD**

##### **Hold Request.**

When active, causes the CPU to release the bus for external DMA or multiprocessing purposes (Section 3.6.7).

##### **Note:**

If **HOLD** is generated asynchronously, its set up and hold times may be violated. In this case it is recommended to synchronize it with the rising edge of **BCLK** to minimize the possibility of metastable states.

The CPU provides only one synchronization stage to minimize the **HLDA** latency. This is to avoid speed degradations in cases of heavy **HOLD** activity (i.e. external DMA controller cycles interleaved with CPU cycles).

#### **RST**

##### **Reset.**

When **RST** is active, the CPU is initialized to a known state (Section 3.6.3).

#### **IRQ-3**

##### **Interrupt Request.**

Low levels on these signals request a prioritized interrupt (Section 3.6.9).

#### **NMI**

##### **Nonmaskable Interrupt.**

A High-to-Low transition of this signal requests a nonmaskable interrupt (Section 3.6.9).

## 4.0 Device Specifications (Continued)

<b>DBG</b>	<b>Debug Trap Request.</b> A High-to-Low transition of this signal requests a debug trap (Section 3.6.9).
<b>EOT</b>	<b>DMA End of Transfer.</b> When asserted in the last T2 state of any DMA bus cycle in Flyby, or the last T2 state of the write cycle in Indirect mode, causes the DMA channel currently in control of the bus to terminate the data transfer.
<b>DRQ0-1</b>	<b>DMA Requests.</b> When active, these signals request DMA service from channels 0 and 1. DRQ0-1 are sampled on each rising edge of BCLK. Once asserted, these signals should not be deasserted until the request has been acknowledged. To avoid multiple DMA cycles, they should be deasserted before the end of each cycle. <b>Note:</b> If DRQ0-1 are generated asynchronously, the set up and hold times may be violated. In this case it is recommended to synchronize them with the rising edge of BCLK to minimize the possibility of metastable states.
<b>TXB0-2</b>	<b>Timer Trigger Signals.</b> A low-to-high transition on any of these pins will signal a trigger event to the corresponding timer. (Section 3.5.3.3).
<b>PLAT</b>	<b>Page Address Latch.</b> When active, the page address for the current access is internally latched. (Section 3.6.11).
<b>CIIN</b>	<b>Cache Inhibit In.</b> When active, indicates that the location referenced in the current bus cycle is not cacheable. CIIN must not change within an aligned 16-byte block.
<b>CINV</b>	<b>Cache Invalidate.</b> When low, both the instruction and data cache contents are invalidated.
<b>IODEC</b>	<b>I/O Decode.</b> Indicates to the CPU that a peripheral device is addressed by the current bus cycle. The value of IODEC must not change within an aligned 16-byte block (Section 3.6.8).
<b>FSSR</b>	<b>Force Slave Status Read.</b> When asserted, indicates that the slave status word should be read by the CPU (Section 3.1.4.1). An external 10 k $\Omega$ resistor should be connected between FSSR and VCC.
<b>SDN</b>	<b>Slave Done.</b> Used by a slave processor to signal the completion of a slave instruction (Section 3.1.4.1). An external 10 k $\Omega$ resistor should be connected between SDN and VCC.
<b>BIN</b>	<b>Burst In.</b> When active, indicates to the CPU that the memory supports burst cycles (Section 3.6.4.3).

<b>RDY</b>	<b>Ready.</b> While this signal is not active, the CPU extends the current bus cycle to support a slow memory or peripheral device.
<b>BW0-1</b>	<b>Bus Width.</b> These lines define the bus width (8, 16 or 32 bits) for each data transfer; BW0 is the least significant bit. The bus width must not change within an aligned 16-byte block—encodings are: 00—Reserved 01—8 Bits 10—16 Bits 11—32 Bits BW0-1 are ignored during DMA transfers.
<b>BRT</b>	<b>Bus Retry.</b> When active, the CPU will reexecute the last bus cycle (Section 3.6.5).

### 4.1.3 Output Signals

<b>BCLK</b>	<b>Bus Clock.</b> Output clock for bus timing (Section 3.6.2).
<b>BCLK</b>	<b>Bus Clock Inverse.</b> Inverted output clock. <b>Note:</b> In the QFP package BCLK and BCLK are each output on two pins that must be externally connected together.
<b>HLDA</b>	<b>Hold Acknowledge.</b> Activated by the CPU in response to the HOLD input to indicate that the CPU has released the bus.
<b>PFS</b>	<b>Program Flow Status.</b> A pulse on this signal indicates the beginning of execution for each instruction (Section 3.6.10).
<b>U/S</b>	<b>User/Supervisor.</b> User or supervisor mode status (Section 3.6.10).
<b>DAK0-1</b>	<b>DMA Acknowledge Signals.</b> Activated in response to DMA requests to notify external devices that the corresponding request has been acknowledged.
<b>PAGE</b>	<b>*In-Page Access.</b> When active, indicates that the current bus access is to the same 8 kbyte page as the previous access.
<b>BP</b>	<b>Break Point.</b> This signal is activated when the CPU detects a PC or operand-address match debug condition (Section 3.3.2).
<b>CASEC</b>	<b>*Cache Section.</b> For cacheable data read bus cycles indicates the Section of the on-chip Data Cache where the data will be placed; undefined for other bus cycles.

## 4.0 Device Specifications (Continued)

<b><math>\overline{\text{IOINH}}</math></b>	<b>I/O Inhibit.</b> Indicates that the current bus cycle should be ignored if a peripheral device is addressed. $\overline{\text{IOINH}}$ is always inactive during DMA bus cycles.
<b><math>\overline{\text{SPC}}</math></b>	<b>Slave Processor Control.</b> Data strobe for slave processor transfers.
<b><math>\overline{\text{BOUT}}</math></b>	<b>*Burst Out.</b> When active, indicates that the CPU is requesting to perform burst cycles.
<b><math>\overline{\text{ILO}}</math></b>	<b>Interlocked Operation.</b> When active, indicates that interlocked cycles are being performed (Section 3.6.4.5).
<b><math>\overline{\text{DDIN}}</math></b>	<b>*Data Direction.</b> Indicates the direction of a data transfer. It is low for reads and high for writes.
<b><math>\overline{\text{ICONF}}</math></b>	<b>Confirm Special Bus Cycle.</b> When active, indicates a type-2 special bus cycle. (Section 3.6.4.6).
<b><math>\overline{\text{CONF}}</math></b>	<b>*Confirm Bus Cycle.</b> When active, indicates that a bus cycle initiated by $\overline{\text{ADS}}$ is valid; that is, the bus cycle has not been cancelled (Section 3.6.4.2).
<b><math>\overline{\text{BMT}}</math></b>	<b>*Begin Memory Transaction.</b> When Stable Low indicates that the current bus cycle is valid; that is, the bus cycle has not been cancelled (Section 3.6.4.2).
<b><math>\overline{\text{ADS}}</math></b>	<b>*Address Strobe.</b> When active, indicates that a bus cycle has begun and a valid address is on the address bus.
<b><math>\overline{\text{BE0-3}}</math></b>	<b>*Byte Enables.</b> Used to selectively enable data transfers on bytes 0-3 of the data bus.

<b>ST0-3</b>	<b>Status.</b> 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: The bus is idle while the slave processor is executing an instruction. 0100—Interrupt Acknowledge. 0101—Reserved. 0110—End of Interrupt. 0111—Reserved. 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.
<b>A0-31</b>	<b>*Address Bus.</b> Used by the CPU to output a 32-bit address at the beginning of a bus cycle. A0 is the least significant.

### 4.1.4 Input/Output Signals

<b>TXA0-2</b>	<b>Timer Control Signals.</b> These signals are used to either output the timer waveforms or to signal trigger events. (Section 3.5.3.)
<b>D0-31</b>	<b>*Data Bus.</b> Used by the CPU to input or output data during a read or write cycle respectively.

## 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.

Case Temperature Under Bias  $0^{\circ}\text{C}$  to  $+95^{\circ}\text{C}$

Storage Temperature  $-65^{\circ}\text{C}$  to  $+150^{\circ}\text{C}$

All Input or Output Voltages with

Respect to GND

$-0.5\text{V}$  to  $+7\text{V}$

Power Dissipation

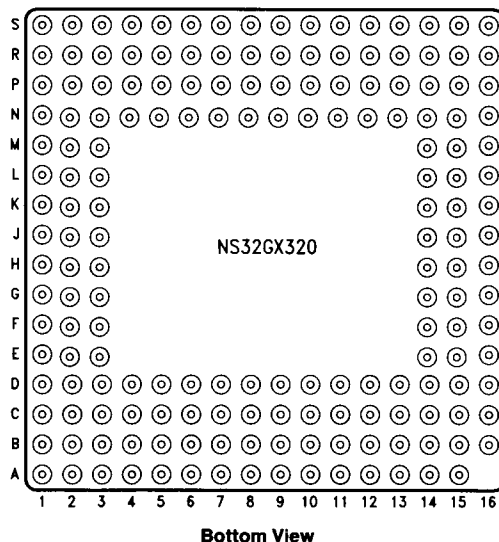
4 W

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** NS32GX320-20, 25:  $T_{\text{CASE}} = 0^{\circ}\text{C}$  to  $+95^{\circ}\text{C}$ ,  $V_{\text{CC}} = 5\text{V} \pm 10\%$ ,  $\text{GND} = 0\text{V}$   
 NS32GX320-30:  $T_{\text{CASE}} = 0^{\circ}\text{C}$  to  $+95^{\circ}\text{C}$ ,  $V_{\text{CC}} = 5\text{V} \pm 5\%$ ,  $\text{GND} = 0\text{V}$ .

Symbol	Parameter	Conditions	Min	Typ	Max	Units
$V_{\text{IH}}$	High Level Input Voltage		2.0		$V_{\text{CC}} + 0.5$	V
$V_{\text{IL}}$	Low Level Input Voltage		-0.5		0.8	V
$V_{\text{OH}}$	High Level Output Voltage	$I_{\text{OH}} = -400\ \mu\text{A}$	2.4			V
$V_{\text{OL}}$	Low Level Output Voltage A0-11, D0-31, $\overline{\text{DIN}}$ CONF, BMT BCLK, $\overline{\text{BCLK}}$ All Other Outputs	$I_{\text{OL}} = 4\ \text{mA}$			0.45	V
		$I_{\text{OL}} = 6\ \text{mA}$			0.45	V
		$I_{\text{OL}} = 16\ \text{mA}$			0.45	V
		$I_{\text{OL}} = 2\ \text{mA}$			0.45	V
$I_{\text{I}}$	Input Load Current	$0 \leq V_{\text{IN}} \leq V_{\text{CC}}$	-20		20	$\mu\text{A}$
$I_{\text{L}}$	Leakage Current (Output and I/O pins in TRI-STATE/Input Mode)	$0.4 \leq V_{\text{IN}} \leq V_{\text{CC}}$	-20		20	$\mu\text{A}$
$C_{\text{IN}}$	CLK Input Capacitance			15		pF
$I_{\text{CC}}$	Active Supply Current	$I_{\text{OUT}} = 0$ , $T_{\text{A}} = 25^{\circ}\text{C}$ $V_{\text{CC}} = 5\text{V}$		700 @ 30 MHz 600 @ 25 MHz 470 @ 20 MHz	800 @ 30 MHz 700 @ 25 MHz 575 @ 20 MHz	mA

## Connection Diagram



Bottom View

FIGURE 4-2. 175-Pin Plastic PGA Package

TL/EE/10564-89

## 4.0 Device Specifications (Continued)

### NS32GX320 Pinout Descriptions

Desc	Pin	Desc	Pin	Desc	Pin
GND	A1	D26	B16	GND	D14
VCC	A2	GND	C1	VCC	D15
VCC	A3	GND	C2	D23	D16
BP	A4	VCC	C3	U/S	E1
PFS	A5	VCC	C4	A31	E2
RST	A6	VCC	C5	GND	E3
NMI	A7	FSSR	C6	D22	E14
GND	A8	RESERVED	C7	D21	E15
IR0	A9	BCLK	C8	D20	E16
IR3	A10	VCC	C9	CASEC	F1
HOLD	A11	IR1	C10	A30	F2
TXA1	A12	DRQ1	C11	IOINH	F3
D31	A13	DAK1	C12	D19	F14
D29	A14	RESERVED	C13	D18	F15
VCC	A15	D27	C14	D17	F16
GND	B1	D25	C15	A28	G1
VCC	B2	D24	C16	A29	G2
GND	B3	GND	D1	VCC	G3
VCC	B4	IL0	D2	GND	G14
VCC	B5	GND	D3	D16	G15
DBG	B6	GND	D4	D15	G16
RESERVED	B7	GND	D5	A26	H1
SYNC	B8	SDN	D6	A27	H2
GND	B9	RESERVED	D7	GND	H3
IR2	B10	BCLK	D8	VCC	H14
CINV	B11	GND	D9	D14	H15
TXA0	B12	CLK	D10	D13	H16
TXA2	B13	DRQ0	D11	A23	J1
D30	B14	DAK0	D12	A24	J2
D28	B15	GND	D13	A25	J3

Desc	Pin	Desc	Pin	Desc	Pin
GND	J14	GND	N9	ST3	R6
VCC	J15	CONF	N10	VCC	R7
D12	J16	RDY	N11	BOUT	R8
VCC	K1	PLAT	N12	BE3	R9
A22	K2	EOT	N13	BE1	R10
GND	K3	GND	N14	VCC	R11
GND	K14	D0	N15	BW1	R12
D11	K15	D5	N16	ICONF	R13
D10	K16	A15	P1	CIIN	R14
A20	L1	VCC	P2	TXB1	R15
A21	L2	GND	P3	D3	R16
VCC	L3	A5	P4	A11	S1
D7	L14	A3	P5	A10	S2
D8	L15	A1	P6	A7	S3
D9	L16	ST0	P7	A6	S4
A19	M1	GND	P8	A0	S5
GND	M2	VCC	P9	ST2	S6
A18	M3	BE0	P10	PAGE	S7
D1	M14	GND	P11	HLDA	S8
D6	M15	BIN	P12	DDIN	S9
VCC	M16	IODEC	P13	BE2	S10
A17	N1	RESERVED	P14	ADS	S11
A16	N2	TXB2	P15	BMT	S12
A12	N3	D4	P16	BRT	S13
A8	N4	A14	R1	BW0	S14
VCC	N5	A13	R2	TXB0	S15
GND	N6	A9	R3	D2	S16
ST1	N7	A4	R4		
SPC	N8	A2	R5		

## 4.4 SWITCHING CHARACTERISTICS

### 4.4.1 Definitions

All the timing specifications given in this section refer to 0.8V or 2.0V on all the signals as illustrated in *Figures 4-3 and 4-4*, unless specifically stated otherwise.

The capacitive load is assumed to be 100 pF on the clock signals and 50 pF on all the other output signals. A minimum capacitive load of 50 pF on BCLK when BCLK is also assumed.

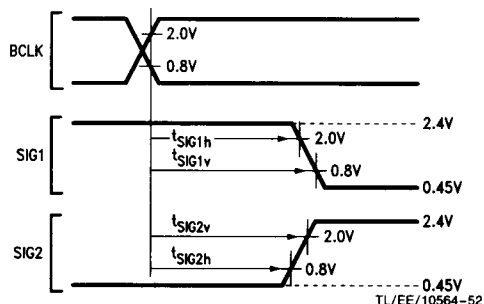


FIGURE 4-3. Output Signals Specification Standard

### ABBREVIATIONS:

L.E.—leading edge R.E.—rising edge  
T.E.—training edge F.E.—falling edge

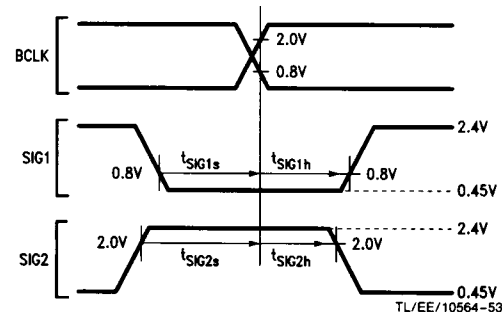


FIGURE 4-4. Input Signals Specification Standard

#### 4.4.2 Timing Tables

##### 4.4.2.1 Output Signals: Internal Propagation Delays, NS32GX320-20, NS32GX320-25, NS32GX320-30

- The output to input timings (e.g., Address to  $\overline{\text{RDY}}$  are at least 2 ns better than the worst case values calculated from the output valid and input setup times relative to BCLK or  $\overline{\text{BCLK}}$ .

Name	Figure	Description	Reference/Conditions	NS32GX320-20		NS32GX320-25		NS32GX320-30		Units
				Min	Max	Min	Max	Min	Max	
$t_{\text{BCp}}$	4-29	Bus Clock Period	R.E., BCLK to Next R.E., BCLK	50	100	40	100	33.3	100	ns
$t_{\text{BCh}}$	4-29	BCLK High Time	At 2.0V on BCLK (Both Edges)	$0.5 t_{\text{BCp}} - 5$		$0.5 t_{\text{BCp}} - 4$		$0.5 t_{\text{BCp}} - 3.65$		ns
$t_{\text{BCl}}$	4-29	BCLK Low Time	At 0.8V on BCLK (Both Edges)	$0.5 t_{\text{BCp}} - 5$		$0.5 t_{\text{BCp}} - 4$		$0.5 t_{\text{BCp}} - 3.65$		ns
$t_{\text{BCr}}$ (Note 1)	4-29	BCLK Rise Time	0.8V to 2.0V on R.E., BCLK		5		4		3	ns
$t_{\text{BCf}}$ (Note 1)	4-29	BCLK Fall Time	2.0V to 0.8V on F.E., BCLK		5		4		3	ns
$t_{\text{NBCh}}$	4-29	$\overline{\text{BCLK}}$ High Time	At 2.0V on $\overline{\text{BCLK}}$ (Both Edges)	$0.5 t_{\text{BCp}} - 5$		$0.5 t_{\text{BCp}} - 4$		$0.5 t_{\text{BCp}} - 3.65$		ns
$t_{\text{NBCl}}$	4-29	$\overline{\text{BCLK}}$ Low Time	At 0.8V on $\overline{\text{BCLK}}$ (Both Edges)	$0.5 t_{\text{BCp}} - 5$		$0.5 t_{\text{BCp}} - 4$		$0.5 t_{\text{BCp}} - 3.65$		ns
$t_{\text{NBCr}}$ (Note 1)	4-29	$\overline{\text{BCLK}}$ Rise Time	0.8V to 2.0V on R.E., $\overline{\text{BCLK}}$		5		4		3	ns
$t_{\text{NBCf}}$ (Note 1)	4-29	$\overline{\text{BCLK}}$ Fall Time	2.0V to 0.8V on F.E., $\overline{\text{BCLK}}$		5		4		3	ns
$t_{\text{CBCdr}}$	4-29	CLK to BCLK R.E. Delay	2.0V on R.E., CLK to 2.0V on R.E., BCLK		20		17		15	ns
$t_{\text{CBCdf}}$	4-29	CLK to BCLK F.E. Delay	2.0V on R.E., CLK to 0.8V on F.E., BCLK		20		17		15	ns
$t_{\text{CNBCdr}}$	4-29	CLK to $\overline{\text{BCLK}}$ R.E. Delay	2.0V on R.E., CLK to 0.8V on R.E., $\overline{\text{BCLK}}$		20		17		15	ns
$t_{\text{CNBCdf}}$	4-29	CLK to $\overline{\text{BCLK}}$ F.E. Delay	2.0V on R.E., CLK to 0.8V on F.E., $\overline{\text{BCLK}}$		20		17		15	ns
$t_{\text{BCNBCrf}}$ (Note 1)	4-29	Bus Clocks Skew	2.0V on R.E., BCLK to 0.8V on F.E., $\overline{\text{BCLK}}$	-2	+2	-2	+2	-2	+2	ns
$t_{\text{BCNBCrf}}$ (Note 1)	4-29	Bus Clocks Skew	0.8V on F.E., BCLK to 2.0V on R.E., BCLK	-2	+2	-2	+2	-2	+2	ns
$t_{\text{Av}}$	4-5, 4-6	Address Bits 0-31 Valid	After R.E., BCLK T1		11		9		9	ns
$t_{\text{Ah}}$	4-5, 4-6	Address Bits 0-31 Hold	After R.E., BCLK T1 or Ti	0		0		0		ns
$t_{\text{Af}}$	4-11, 4-14	Address Bits 0-31 Floating	After F.E., BCLK Ti		21		17		13	ns
$t_{\text{Anf}}$	4-11, 4-14	Address Bits 0-31 Not Floating	After F.E., BCLK Ti	0		0		0		ns

**Note 1:** Guaranteed by characterization. Due to tester conditions, this parameter is not 100% tested.

## 4.0 Device Specifications (Continued)

### 4.4.2.1 Output Signals: Internal Propagation Delays, NS32GX320-20, NS32GX320-25, NS32GX320-30 (Continued)

Name	Figure	Description	Reference/Conditions	NS32GX320-20		NS32GX320-25		NS32GX320-30		Units
				Min	Max	Min	Max	Min	Max	
$t_{ABV}$	4-8	Address Bits A2, A3 Valid (Burst Cycle)	After R.E., BCLK T2B		11		9		9	ns
$t_{ABh}$	4-8	Address Bits A2, A3 Hold (Burst Cycle)	After R.E., BCLK T2B	0		0		0		ns
$t_{DOV}$	4-6, 4-17	Data Out Valid	After R.E., BCLK T1	$0.5 t_{BCp}$	$0.5 t_{BCp} + 13$	$0.5 t_{BCp}$	$0.5 t_{BCp} + 12$	$0.5 t_{BCp}$	$0.5 t_{BCp} + 11$	ns
$t_{IDOV}$	4-12	Internal Data Out Valid	After R.E., BCLK T2		$0.5 t_{BCp} + 15$		$0.5 t_{BCp} + 13$		$0.5 t_{BCp} + 11$	ns
$t_{DOh}$	4-6, 4-17	Data Out Hold	After R.E., BCLK T1 or Ti	0		0		0		ns
$t_{DOspc}$	4-17	Data Out Setup (Slave Write)	Before $\overline{SPC}$ T.E.	8		6		5		ns
$t_{DOf}$	4-7	Data Bus Floating	After R.E., BCLK T1 or Ti		21		17		13	ns
$t_{DONf}$	4-7	Data Bus Not Floating	After F.E., BCLK T1	0		0		0		ns
$t_{BMTV}$	4-5, 4-7	BMT Signal Valid	After R.E., BCLK T1		32		27		22	ns
$t_{BMT_h}$	4-5, 4-7	BMT Signal Hold	After R.E., BCLK T2	0		0		0		ns
$t_{BMT_f}$	4-11, 4-14	BMT Signal Floating	After F.E., BCLK Ti		21		17		13	ns
$t_{BMT_{hf}}$	4-11, 4-14	BMT Signal Not Floating	After F.E., BCLK Ti	0		0		0		ns
$t_{CONF_a}$	4-5, 4-8	CONF Signal Active	After R.E., BCLK T1	$0.5 t_{BCp}$	$0.5 t_{BCp} + 11$	$0.5 t_{BCp}$	$0.5 t_{BCp} + 9$	$0.5 t_{BCp}$	$0.5 t_{BCp} + 9$	ns
$t_{CONF_{ia}}$	4-5, 4-8	CONF Signal Inactive	After R.E., BCLK T1 or Ti		11		9		9	ns
$t_{CONF_f}$	4-11, 4-14	CONF Signal Floating	After F.E., BCLK Ti		21		17		13	ns
$t_{CONF_{nf}}$	4-11, 4-14	CONF Signal Not Floating	After F.E., BCLK Ti	0		0		0		ns
$t_{ICONF_a}$	4-12	ICONF Signal Active	After R.E., BCLK T1	$0.5 t_{BCp}$	$0.5 t_{BCp} + 11$	$0.5 t_{BCp}$	$0.5 t_{BCp} + 9$	$0.5 t_{BCp}$	$0.5 t_{BCp} + 9$	ns
$t_{ICONF_{ia}}$	4-12	ICONF Signal Inactive	After R.E., BCLK next T1, Ti		11		9		9	ns
$t_{ADS_a}$	4-5, 4-8	ADS Signal Active	After R.E., BCLK T1		11		9		9	ns
$t_{ADS_{ia}}$	4-5, 4-8	ADS Signal Inactive	After F.E., BCLK T1		11		9		9	ns
$t_{ADSw}$	4-6	ADS Pulse Width	At 0.8V (Both Edges)	16		12		10		ns
$t_{ADS_f}$	4-11, 4-14	ADS Signal Floating	After F.E., BCLK Ti		21		17		13	ns
$t_{ADS_{nf}}$	4-11, 4-14	ADS Signal Not Floating	After F.E., BCLK Ti	0		0		0		ns
$t_{BE_v}$	4-6, 4-8	$\overline{BE}_n$ Signals Valid	After R.E., BCLK T1		11		9		9	ns
$t_{BE_h}$	4-6, 4-8	$\overline{BE}_n$ Signals Hold	After R.E., BCLK T1, Ti or T2B	0		0		0		ns
$t_{BE_f}$	4-11, 4-14	$\overline{BE}_n$ Signals Floating	After F.E., BCLK Ti		21		17		13	ns



## 4.0 Device Specifications (Continued)

### 4.4.2.1 Output Signals: Internal Propagation Delays, NS32GX320-20, NS32GX320-25, NS32GX320-30 (Continued)

Name	Figure	Description	Reference/Conditions	NS32GX320-20		NS32GX320-25		NS32GX320-30		Units
				Min	Max	Min	Max	Min	Max	
$t_{BE_{nf}}$	4-11, 4-14	$\overline{BE}_n$ Signals Not Floating	After F.E., BCLK T <sub>i</sub>	0		0		0		ns
$t_{DDIN_v}$	4-5, 4-6	DDIN Signal Valid	After R.E., BCLK T <sub>1</sub>		11		9		9	ns
$t_{DDIN_h}$	4-5, 4-6	DDIN Signal Hold	After R.E., BCLK T <sub>1</sub> or T <sub>i</sub>	0		0		0		ns
$t_{DDIN_f}$	4-11, 4-14	DDIN Signal Floating	After F.E., BCLK T <sub>i</sub>		21		17		13	ns
$t_{DDIN_{nf}}$	4-11, 4-14	DDIN Signal Not Floating	After F.E., BCLK T <sub>i</sub>	0		0		0		ns
$t_{PAGE_v}$	4-23	PAGE Signal Valid	After R.E., BCLK T <sub>1</sub>		22		18		15	ns
$t_{PAGE_h}$	4-23	PAGE Signal Hold	After R.E., BCLK next T <sub>1</sub> , T <sub>i</sub>	0		0		0		ns
$t_{PAGE_f}$	4-14	PAGE Signal Floating	After F.E., BCLK T <sub>i</sub>		21		17		13	ns
$t_{PAGE_{nf}}$	4-14	PAGE Signal Not Floating	After F.E., BCLK T <sub>i</sub>	0		0		0		ns
$t_{SPC_a}$	4-16, 4-17	SPC Signal Active	After R.E., BCLK T <sub>1</sub>		19		15		12	ns
$t_{SPC_{ia}}$	4-16, 4-17	SPC Signal Inactive	After R.E., BCLK T <sub>i</sub> , T <sub>1</sub> or T <sub>2</sub>		19		15		12	ns
$t_{DDSPC}$ (Note 2)	4-16	DDIN Valid to SPC Active	Before SPC L.E.	0		0		0		ns
$t_{HLDA_a}$	4-14, 4-15	HLDA Signal Active	After F.E., BCLK T <sub>i</sub>		15		11		10	ns
$t_{HLDA_{ia}}$	4-14	HLDA Signal Inactive	After F.E., BCLK T <sub>i</sub>		15		11		10	ns
$t_{DAK_a}$	4-13	$\overline{DAK}_n$ Signals Active	After F.E., BCLK T <sub>i</sub>		15		11		10	ns
$t_{DAK_{ia}}$	4-13	$\overline{DAK}_n$ Signal Inactive	After F.E., BCLK T <sub>i</sub>		15		11		10	ns
$t_{ST_v}$	4-5, 4-16	Status (ST0-4) Valid	After R.E., BCLK T <sub>1</sub>		11		9		9	ns
$t_{ST_h}$	4-5, 4-16	Status (ST0-4) Hold	After R.E., BCLK T <sub>1</sub> or T <sub>i</sub>	0		0		0		ns
$t_{BOUT_a}$	4-8, 4-9	BOUT Signal Active	After R.E., BCLK T <sub>2</sub>		15		12		11	ns
$t_{BOUT_{ia}}$	4-8, 4-9	BOUT Signal Inactive	After R.E., BCLK Last T <sub>2B</sub> , T <sub>1</sub> or T <sub>i</sub>		15		12		11	ns
$t_{BOUT_f}$	4-11, 4-14	BOUT Signal Floating	After F.E., BCLK T <sub>i</sub>		21		17		13	ns
$t_{BOUT_{nf}}$	4-11, 4-14	BOUT Signal Not Floating	After F.E., BCLK T <sub>i</sub>	0		0		0		ns
$t_{ILO_a}$	4-7	Interlock Signal Active	After F.E., BCLK T <sub>i</sub>		11		9		9	ns
$t_{ILO_{ia}}$	4-7	Interlock Signal Inactive	After F.E., BCLK T <sub>i</sub>		11		9		9	ns
$t_{PFS_a}$	4-22	PFS Signal Active	After F.E., BCLK		15		11		10	ns
$t_{PFS_{ia}}$	4-22	PFS Signal Inactive	After F.E., Next BCLK		15		11		10	ns
$t_{BP_a}$	4-22	$\overline{BP}$ Signal Active	After F.E., BCLK		15		11		10	ns
$t_{BP_{ia}}$	4-22	$\overline{BP}$ Signal Inactive	After F.E., Next BCLK		15		11		10	ns
$t_{US_v}$	4-5	U/ $\overline{S}$ Signal Valid	After R.E., BCLK T <sub>1</sub>		11		9		9	ns
$t_{US_h}$	4-5	U/ $\overline{S}$ Signal Hold	After R.E., BCLK T <sub>1</sub> or T <sub>i</sub>	0		0		0		ns
$t_{CAS_v}$	4-5	CAS $\overline{C}$ Signal Valid	After F.E., BCLK T <sub>1</sub>		15		11		10	ns

## 4.0 Device Specifications (Continued)

### 4.4.2.1 Output Signals: Internal Propagation Delays, NS32GX320-20, NS32GX320-25, NS32GX320-30 (Continued)

Name	Figure	Description	Reference/Conditions	NS32GX320-20		NS32GX320-25		NS32GX320-30		Units
				Min	Max	Min	Max	Min	Max	
$t_{CASh}$	4-5	$\overline{CASEC}$ Signal Hold	After R.E., BCLK T1 or Ti	0		0		0		ns
$t_{CASf}$	4-11, 4-14	$\overline{CASEC}$ Signal Floating	After F.E., BCLK Ti		21		17		13	ns
$t_{CASnf}$	4-11, 4-14	$\overline{CASEC}$ Signal Not Floating	After F.E., BCLK Ti	0		0		0		ns
$t_{IOlv}$	4-5	$\overline{IOINH}$ Signal Valid	After R.E., BCLK T1		15		11		10	ns
$t_{IOlh}$	4-5	$\overline{IOINH}$ Signal Hold	After R.E., BCLK T1 or Ti	0		0		0		ns
$t_{TXAv}$	4-26	TXA <sub>n</sub> Outputs Valid	After R.E., BCLK		15		11		10	ns
$t_{TXAh}$	4-26	TXA <sub>n</sub> Outputs Hold	After R.E., BCLK	0		0		0		ns
$t_{TXAf}$	4-27	TXA <sub>n</sub> Signals Floating	After F.E., BCLK		21		17		13	ns
$t_{TXAnf}$	4-27	TXA <sub>n</sub> Signals Not Floating	After F.E., BCLK	0		0		0		ns

**Note 1:** Guaranteed by characterization. Due to tester conditions, this parameter is not 100% tested.

**Note 2:**  $t_{DQSPC}$  and all the parameters related to the "Floating/Not Floating" conditions are guaranteed by characterization. Due to tester conditions, these parameters are not 100% tested.

### 4.4.2.2 Input Signal Requirements: NS32GX320-20, NS32GX320-25, NS32GX320-30

Name	Figure	Description	Reference/Conditions	NS32GX320-20		NS32GX320-25		NS32GX320-30		Units
				Min	Max	Min	Max	Min	Max	
$t_{Cp}$	4-29	Input Clock Period	R.E., CLK to Next R.E., CLK	25	50	20	50	16.6	50	ns
$t_{Ch}$ (Note 3)	4-29	CLK High Time	At 2.0V on CLK (Both Edges)	$0.5 t_{Cp} - 5$		$0.5 t_{Cp} - 5$		$0.5 t_{Cp} - 4$		ns
$t_{Cl}$ (Note 3)	4-29	CLK Low Time	At 0.8V on CLK (Both Edges)	$0.5 t_{Cp} - 5$		$0.5 t_{Cp} - 5$		$0.5 t_{Cp} - 4$		ns
$t_{Cr}$ (Note 3)	4-29	CLK Rise Time	0.8V to 2.0V on R.E., CLK		5		4		3	ns
$t_{Cf}$ (Note 3)	4-29	CLK Fall Time	2.0V to 0.8V on F.E., CLK		5		4		3	ns
$t_{DIs}$	4-5, 4-16	Data In Setup	Before R.E., BCLK T1 or Ti	13		11		8		ns
$t_{DIh}$	4-5, 4-16	Data In Hold	After R.E., BCLK T1 or Ti	1		1		1		ns
$t_{RDYs}$	4-5	$\overline{RDY}$ Setup Time	Before R.E., BCLK T2(W), T1 or Ti	22		18		12		ns
$t_{RDYh}$	4-5	$\overline{RDY}$ Hold Time	After R.E., BCLK T2(W), T1 or Ti	1		1		1		ns
$t_{BW_s}$	4-5	BW0-1 Setup Time	Before F.E., BCLK T2 or T2(W)	21		17		13		ns
$t_{BW_h}$	4-5	BW0-1 Hold Time	After F.E., BCLK T2 or T2(W)	1		1		1		ns
$t_{PLAT_s}$	4-23	$\overline{PLAT}$ Setup Time	Before F.E., BCLK T2	21		17		13		ns
$t_{PLAT_h}$	4-23	$\overline{PLAT}$ Hold Time	After F.E., BCLK T2	1		1		1		ns
$t_{DRQ_s}$	4-13	$\overline{DRQ_n}$ Setup Time	Before R.E., BCLK	21		17		14		ns
$t_{DRQ_h}$	4-13	$\overline{DRQ_n}$ Hold Time	After R.E., BCLK	1		1		1		ns
$t_{EOT_s}$	4-13	$\overline{EOT}$ Setup Time	Before R.E., BCLK next T1, Ti	23		19		16		ns

## 4.0 Device Specifications (Continued)

### 4.4.2.2 Input Signal Requirements: NS32GX320-20, NS32GX320-25, NS32GX320-30 (Continued)

Name	Figure	Description	Reference/Conditions	NS32GX320-20		NS32GX320-25		NS32GX320-30		Units
				Min	Max	Min	Max	Min	Max	
$t_{EOT_h}$	4-13	$\overline{EOT}$ Hold Time	After R.E., BCLK	1		1		1		ns
$t_{HOLD_s}$	4-14, 4-15	HOLD Setup Time	Before R.E., BCLK	19		15		12		ns
$t_{HOLD_h}$	4-14	HOLD Hold Time	After R.E., BCLK	1		1		1		ns
$t_{BIN_s}$	4-8	$\overline{BIN}$ Setup Time	Before F.E., BCLK T2 or T2(W)	21		17		13		ns
$t_{BIN_h}$	4-8	$\overline{BIN}$ Hold Time	After F.E., BCLK T2 or T2(W)	1		1		1		ns
$t_{BRT_s}$	4-6, 4-8	$\overline{BRT}$ Setup Time	Before R.E., BCLK T1 or T1	21		17		13		ns
$t_{BRT_h}$	4-6, 4-8	$\overline{BRT}$ Hold Time	After R.E., BCLK T1 or T1	1		1		1		ns
$t_{IOD_s}$	4-5	$\overline{IO\overline{DEC}}$ Setup Time	Before F.E., BCLK T2 or T2(W)	21		17		13		ns
$t_{IOD_h}$	4-5	$\overline{IO\overline{DEC}}$ Hold Time	After F.E., BCLK T2 or T2(W)	1		1		1		ns
$t_{PWR}$ (Note 3)	4-31	Power Stable to R.E. of $\overline{RST}$	After VCC Reaches 4.5V	50		40		30		$\mu s$
$t_{RST_s}$	4-32	$\overline{RST}$ Setup Time	Before R.E., BCLK	14		12		11		ns
$t_{RST_w}$	4-32	$\overline{RST}$ Pulse Width	At 0.8V (Both Edges)	64		64		64		$t_{BCP}$
$t_{CII_s}$	4-5	$\overline{CII\overline{N}}$ Setup Time	Before F.E., BCLK T2	21		17		13		ns
$t_{CII_h}$	4-5	$\overline{CII\overline{N}}$ Hold Time	After F.E., BCLK T2	1		1		1		ns
$t_{CINV_s}$	4-28	$\overline{CINV}$ Signal Setup	Before R.E., BCLK	14		12		11		ns
$t_{CINV_h}$	4-28	$\overline{CINV}$ Signal Hold	After R.E., BCLK	1		1		1		ns
$t_{IR_s}$	4-20	$\overline{IR_n}$ Setup Time	Before R.E., BCLK	14		12		11		ns
$t_{IR_h}$	4-20	$\overline{IR_n}$ Hold Time	After R.E., BCLK	1		1		1		ns
$t_{NMI_s}$	4-20	$\overline{NMI}$ Setup Time	Before R.E., BCLK	20		17		16		ns
$t_{NMI_h}$	4-20	$\overline{NMI}$ Hold Time	After R.E., BCLK	1		1		1		ns
$t_{SD_s}$	4-18	$\overline{SDN}$ Setup Time	Before R.E., BCLK	14		12		11		ns
$t_{SD_h}$	4-18	$\overline{SDN}$ Hold Time	After R.E., BCLK	1		1		1		ns
$t_{FSSR_s}$	4-19	$\overline{FSSR}$ Setup Time	Before R.E., BCLK	14		12		11		ns
$t_{FSSR_h}$	4-19	$\overline{FSSR}$ Hold Time	After R.E., BCLK	1		1		1		ns
$t_{SYNC_s}$	4-30	$\overline{SYNC}$ Setup Time	Before R.E., CLK	10		8		7		ns
$t_{SYNC_h}$	4-30	$\overline{SYNC}$ Hold Time	After R.E., CLK	1		1		1		ns
$t_{DBG_s}$	4-21	$\overline{DBG}$ Setup Time	Before R.E., BCLK	14		12		11		ns
$t_{DBG_h}$	4-21	$\overline{DBG}$ Hold Time	After R.E., BCLK	1		1		1		ns
$t_{TXAB_s}$	4-25	$TXA_n$ and $TXB_n$ Signals Setup	Before F.E., BCLK	14		12		11		ns
$t_{TXAB_h}$	4-25	$TXA_n$ and $TXB_n$ Signals Hold	After F.E., BCLK	1		1		1		ns

**Note 3:** Guaranteed by characterization. Due to tester conditions, this parameter is not 100% tested.

## 4.0 Device Specifications (Continued)

### 4.4.3 Timing Diagrams

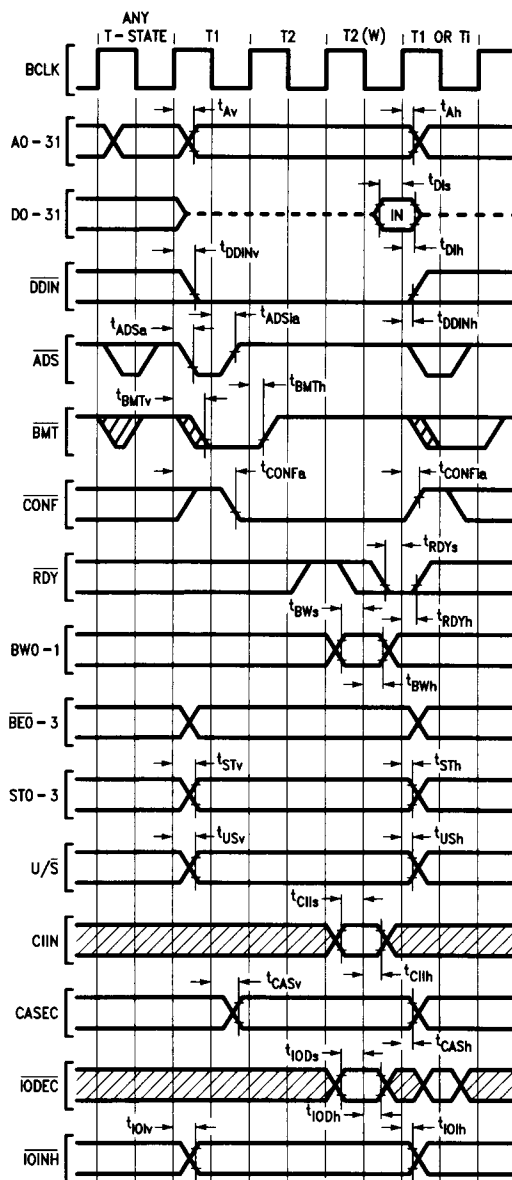
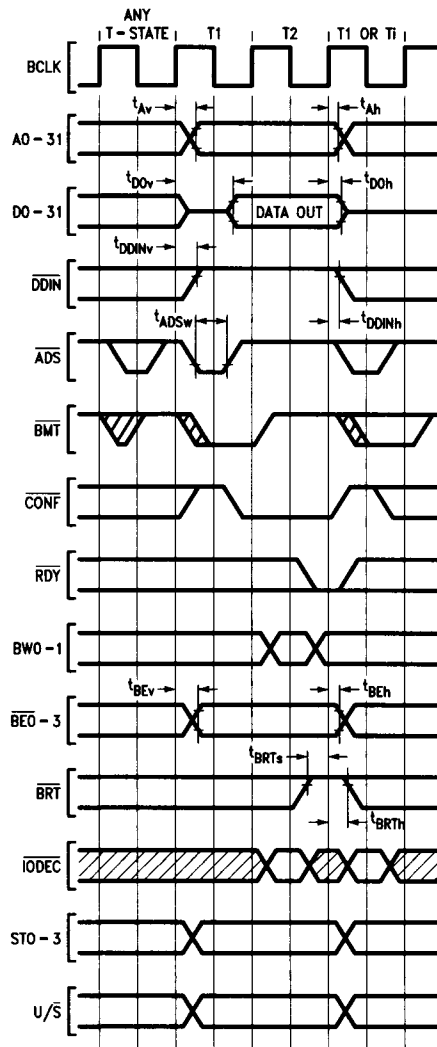


FIGURE 4-5. Basic Read Cycle Timing

TL/EE/10564-54

## 4.0 Device Specifications (Continued)



TL/EE/10564-55

**Note:** An Idle State is always inserted before a Write Cycle when the Write immediately follows a confirmed Read Cycle. A0-31, DDIN, BE0-3, ST0-3 and U/S remain unchanged during this idle state.

**FIGURE 4-6. Write Cycle Timing**

## 4.0 Device Specifications (Continued)

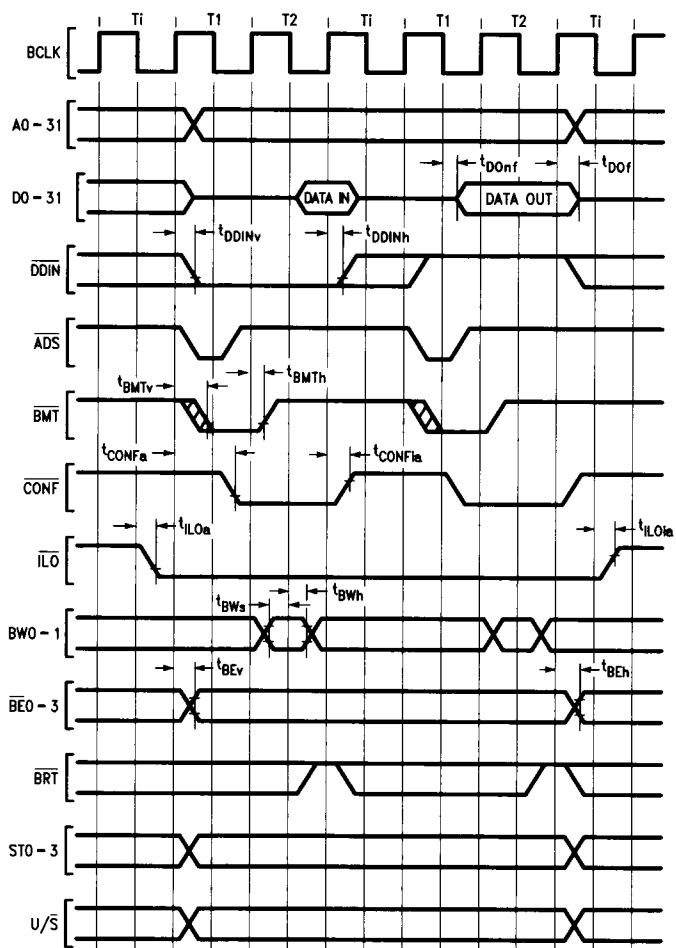


FIGURE 4-7. Interlocked Read and Write Cycles

TL/EE/10564-56

## 4.0 Device Specifications (Continued)

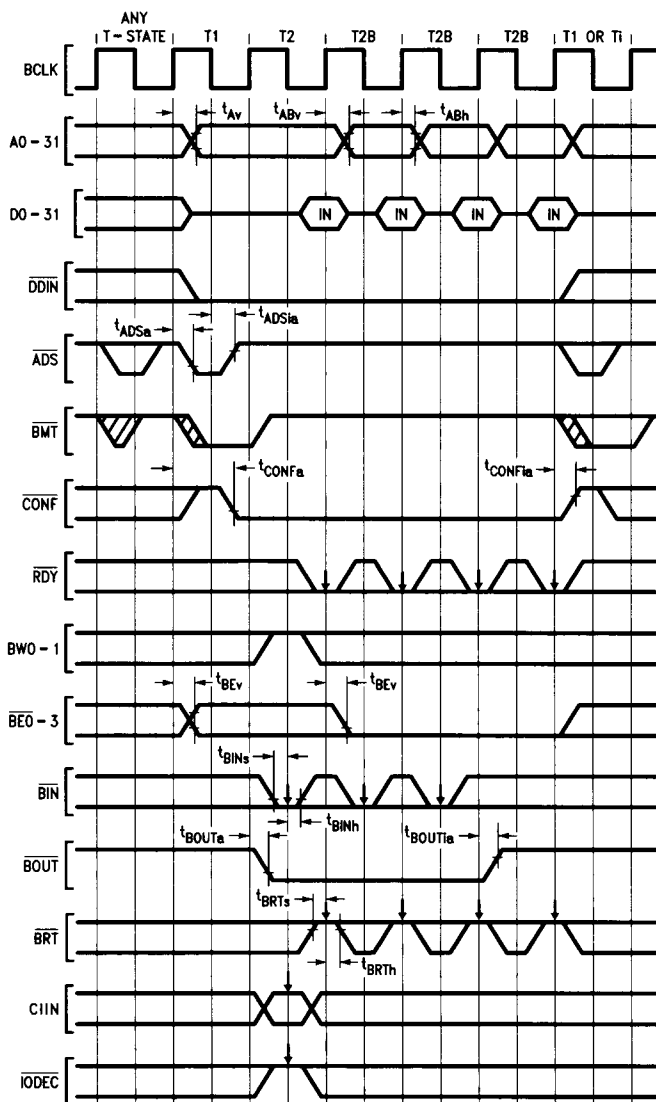


FIGURE 4-8. Burst Read Cycles

TL/EE/10564-57

## 4.0 Device Specifications (Continued)

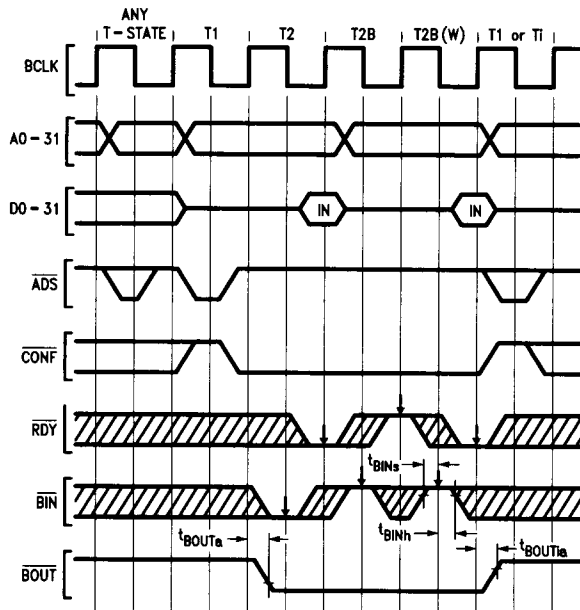


FIGURE 4-9. External Termination of Burst Cycles

TL/EE/10564-58

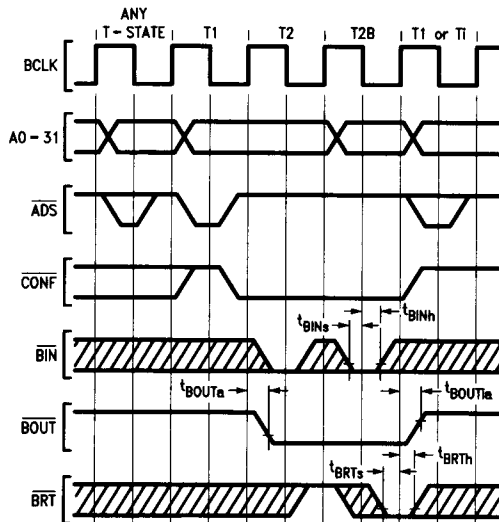


FIGURE 4-10. Bus Retry during Burst Cycles

TL/EE/10564-59

**Note:** Two idle state are always inserted by the CPU following the assertion of BRT.



## 4.0 Device Specifications (Continued)

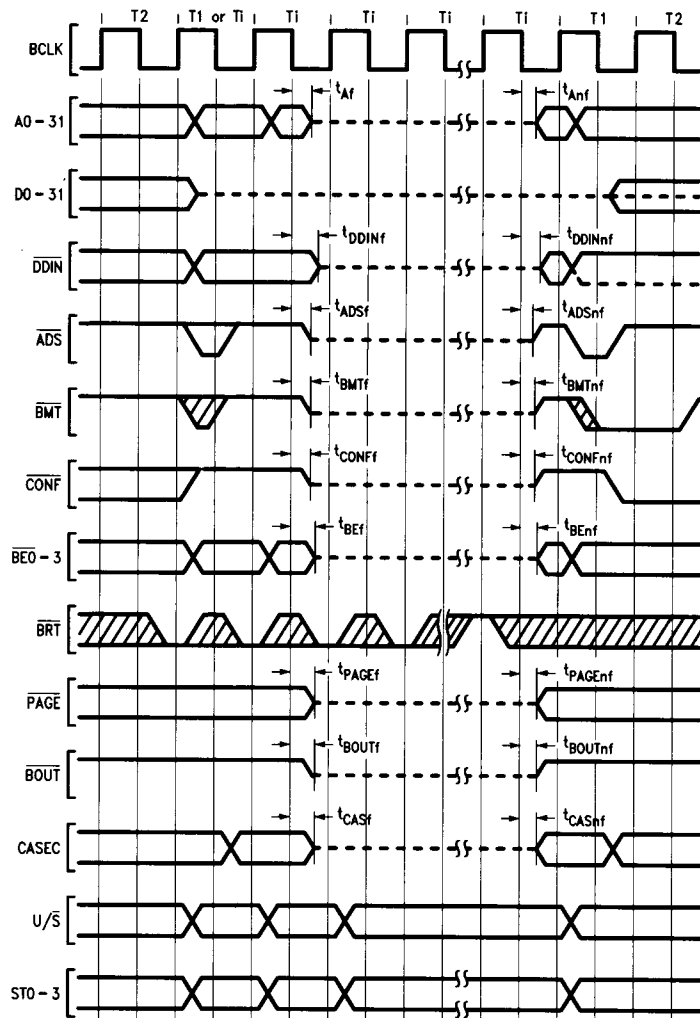
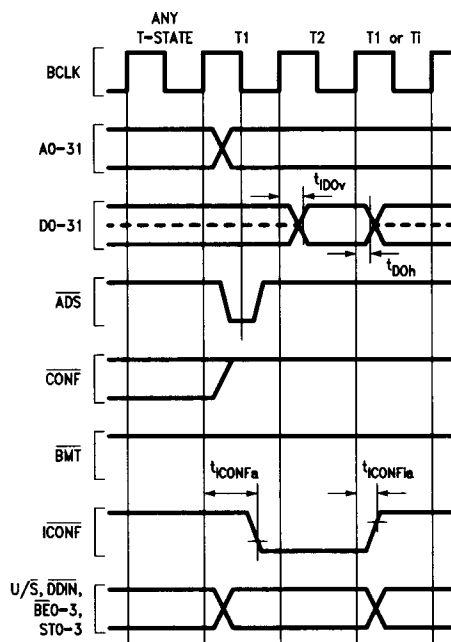


FIGURE 4-11. Extended Retry Timing

TL/EE/10564-60

## 4.0 Device Specifications (Continued)



TL/EE/10564-61

FIGURE 4-12. Type-2 Special Bus Cycle

## 4.0 Device Specifications (Continued)

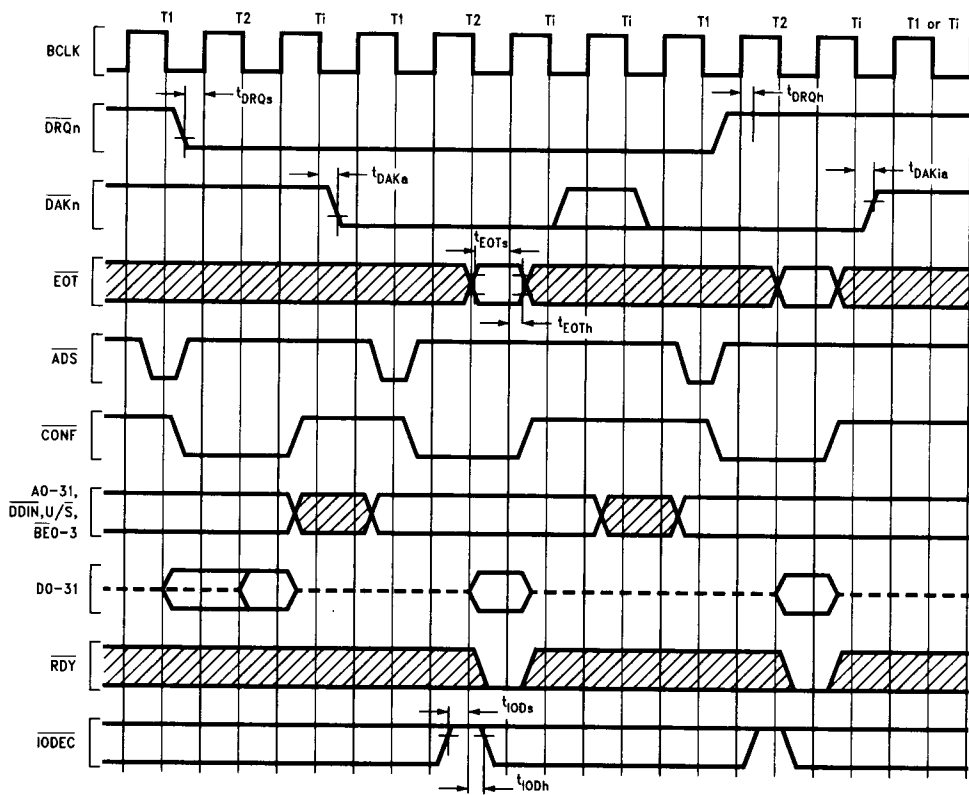


FIGURE 4-13. Flyby DMA Transfer Cycles (Bus Initially Not Idle)

TL/EE/10564-62

## 4.0 Device Specifications (Continued)

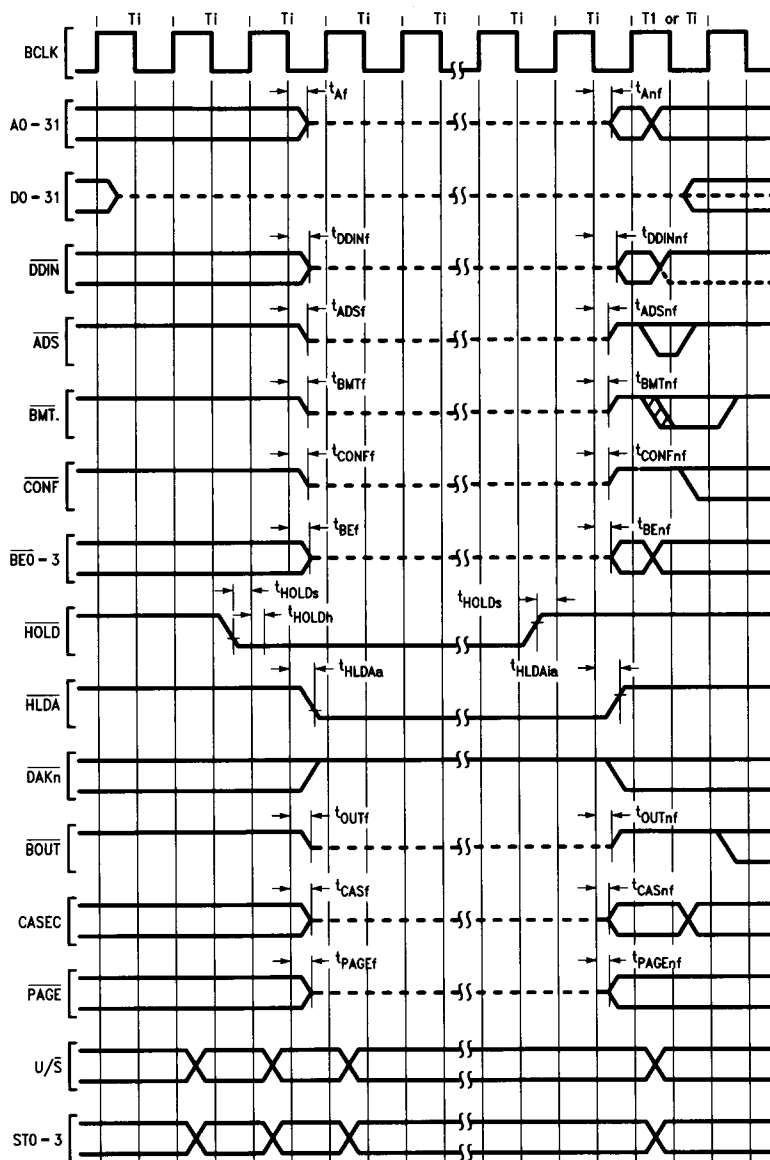
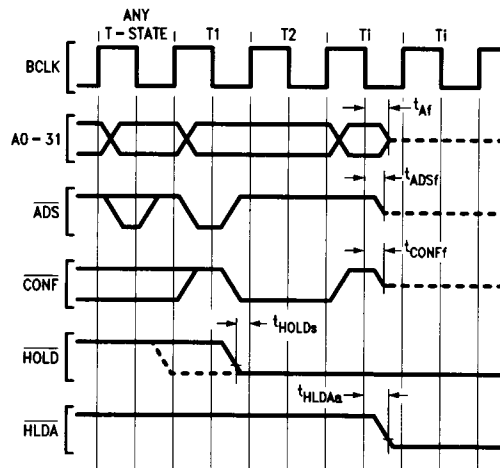


FIGURE 4-14. Hold Timing (Bus Initially Idle)

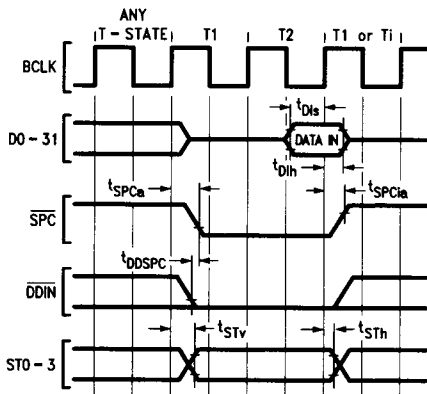
TL/EE/10564-63

## 4.0 Device Specifications (Continued)



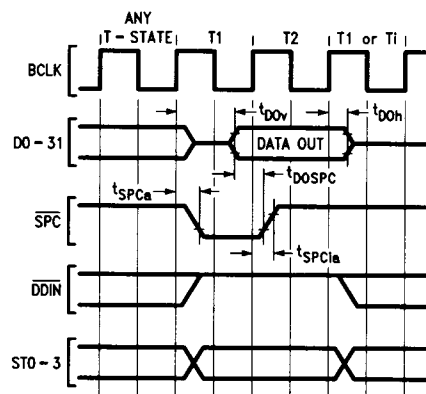
TL/EE/10564-64

FIGURE 4-15.  $\overline{\text{HOLD}}$  Acknowledge Timing (Bus Initially Not Idle)



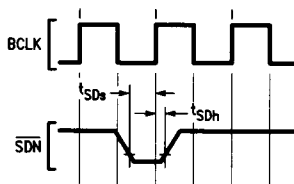
TL/EE/10564-65

FIGURE 4-16. Slave Processor Read Timing



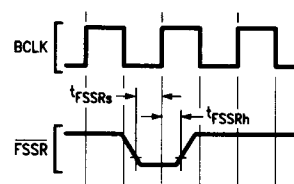
TL/EE/10564-66

FIGURE 4-17. Slave Processor Write Timing



TL/EE/10564-67

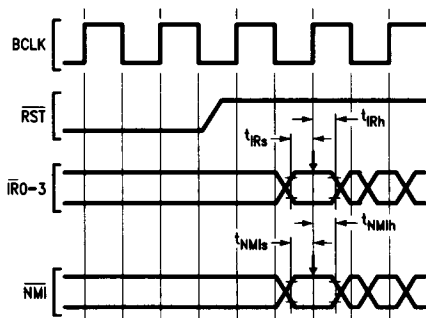
FIGURE 4-18. Slave Processor Done



TL/EE/10564-68

FIGURE 4-19.  $\overline{\text{FSSR}}$  Signal Timing

## 4.0 Device Specifications (Continued)

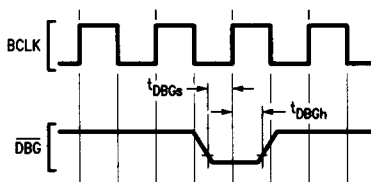


TL/EE/10564-69

**FIGURE 4-20. IR0-3 and NMI Signals Sampling**

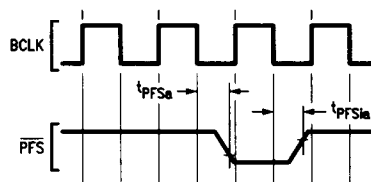
**Note 1:** IR0-3 and NMI are sampled on every rising edge of BCLK.

**Note 2:** IR0-3 are level sensitive, and once asserted, they should not be deasserted until they are acknowledged.



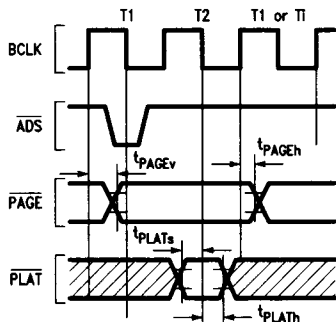
TL/EE/10564-70

**FIGURE 4-21. Debug Trap Request**



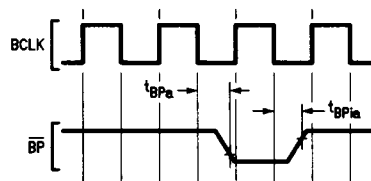
TL/EE/10564-71

**FIGURE 4-22. PFS Signal Timing**



TL/EE/10564-72

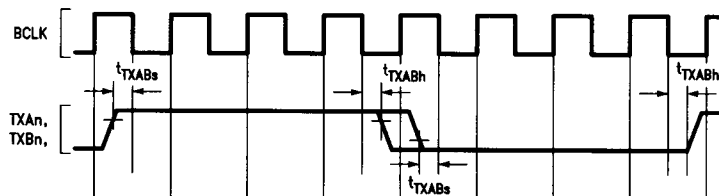
**FIGURE 4-23. PAGE and PLAT Signals Timing**



TL/EE/10564-73

**FIGURE 4-24. Break Point Signal Timing**

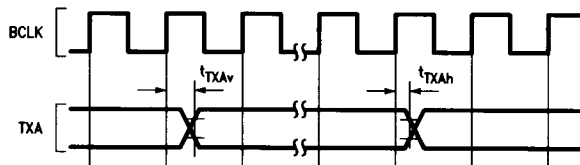
## 4.0 Device Specifications (Continued)



TL/EE/10564-74

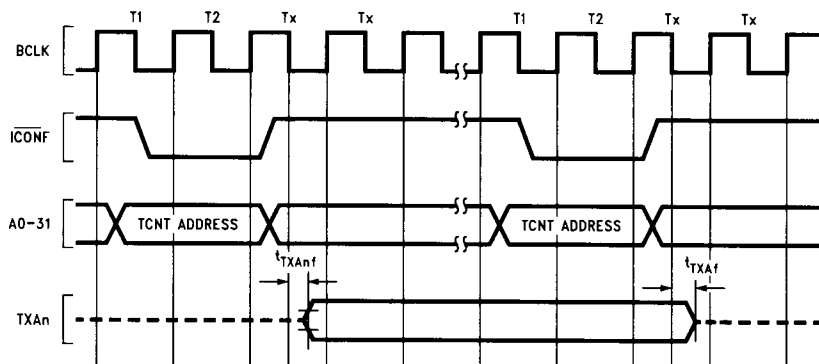
**Note:** Once TXAn and TXBn are changed, they must stay high or low for at least 4 BCLK cycles.

**FIGURE 4-25. TXAn and TXBn Input Timing**



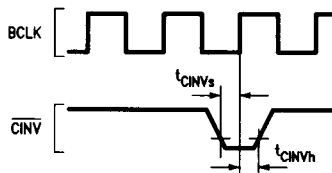
TL/EE/10564-75

**FIGURE 4-26. TXAn Output Timing**



TL/EE/10564-76

**FIGURE 4-27. TXAn Float/Non-Float Timing**



TL/EE/10564-77

**FIGURE 4-28. Cache Invalidation Timing**

## 4.0 Device Specifications (Continued)

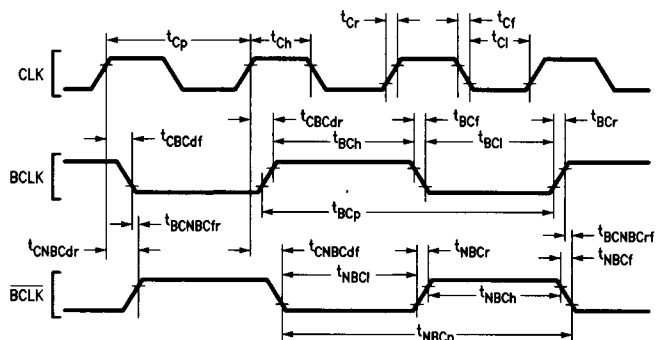


FIGURE 4-29. Clock Waveforms

TL/EE/10564-78

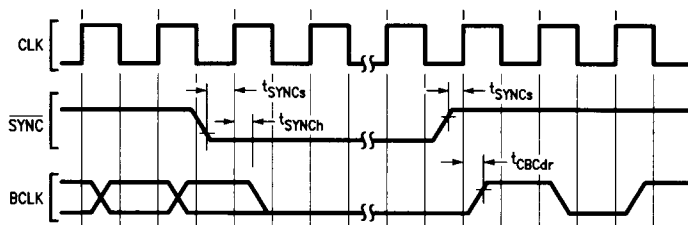


FIGURE 4-30. Bus Clock Synchronization

TL/EE/10564-79

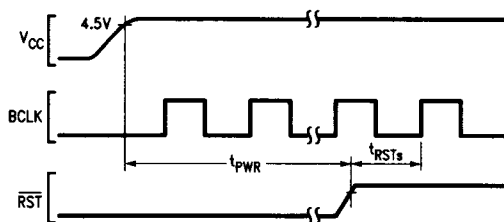


FIGURE 4-31. Power-On Reset

TL/EE/10564-80

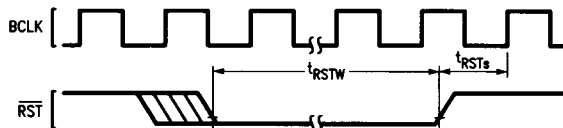


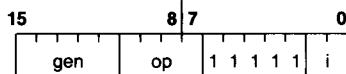
FIGURE 4-32. Non-Power-On Reset

TL/EE/10564-81



## 97

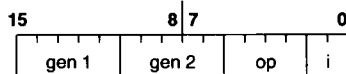
# Appendix A: Instruction Formats (Continued)



**Format 3**

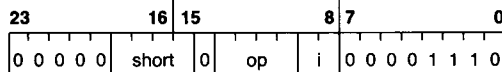
CXPD	-0000	ADJSP	-1010
BICPSR	-0010	JSR	-1100
JUMP	-0100	CASE	-1110
BISPSR	-0110		

Trap (UND) on XXX1, 1000



**Format 4**

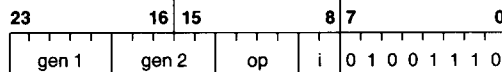
ADD	-0000	SUB	-1000
CMP	-0001	ADDR	-1001
BIC	-0010	AND	-1010
ADDC	-0100	SUBC	-1100
MOV	-0101	TBIT	-1101
OR	-0110	XOR	-1110



**Format 5**

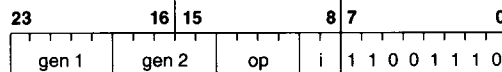
MOVS	-0000	SETCFG	-0010
CMPS	-0001	SKPS	-0011

Trap (UND) on 1XXX, 01XX



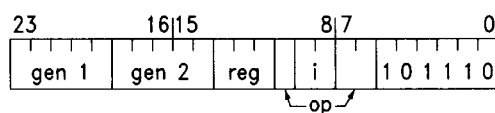
**Format 6**

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

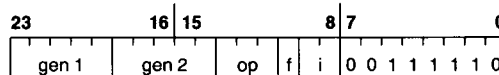


**Format 8**

TL/EE/10564-82

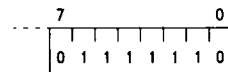
EXT	-0 00	INDEX	-1 00
CVTP	-0 01	FFS	-1 01
INS	-0 10		
CHECK	-0 11		
MOVSU	-110, reg = 001		
MOVUS	-110, reg = 011		

Trap (UND) on 110, 111



**Format 9**

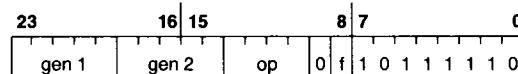
MOVif	-000	ROUND	-100
LFSS	-001	TRUNC	-101
MOVLf	-010	SFSR	-110
MOVFL	-011	FLOOR	-111



TL/EE/10564-83

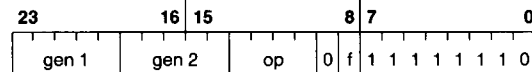
**Format 10**

Trap (UND) Always



**Format 11**

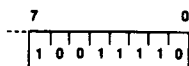
ADDf	-0000	DIVf	-1000
MOVf	-0001	Note 1	-1001
CMPf	-0010	Note 3	-1010
Note 3	-0011	Note 1	-1011
SUBf	-0100	MULf	-1100
NEGf	-0101	ABSf	-1101
Note 2	-0110	Note 2	-1110
Note 1	-0111	Note 1	-1111



**Format 12**

Note 2	-0000	Note 2	-1000
Note 1	-0001	Note 1	-1001
POLYf	-0010	Note 3	-1010
DOTf	-0011	Note 1	-1011
SCALBf	-0100	Note 2	-1100
LOGBf	-0101	Note 1	-1101
Note 2	-0110	Note 2	-1110
Note 1	-0111	Note 1	-1111

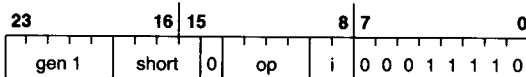
## Appendix A: Instruction Formats (Continued)



TL/EE/10564-84

**Format 13**

Trap (UND) Always



**Format 14**

CINV - 1001

Trap (UND) on 00XX, 01XX, 1000, 101X, 11XX



Operation Word

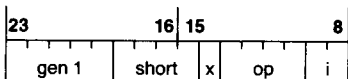
ID Byte

**Format 15**

(Custom Slave)

Operation Word Format

nnn



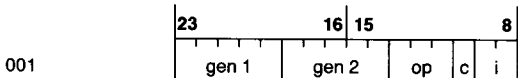
000

**Format 15.0**

LCR -0010

SCR -0011

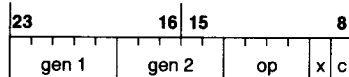
Trap (UND) on all others



**Format 15.1**

CCV3	-000	CCV2	-100
LCSR	-001	CCV1	-101
CCV5	-010	SCSR	-110
CCV4	-011	CCV0	-111

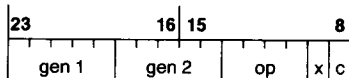
101



**Format 15.5**

CCAL0	-0000	CCAL3	-1000
CMOV0	-0001	CMOV3	-1001
CCMP0	-0010	Note 3	-1010
CCMP1	-0011	Note 1	-1011
CCAL1	-0100	CCAL2	-1100
CMOV2	-0101	CMOV1	-1101
Note 2	-0110	Note 2	-1110
Note 1	-0111	Note 1	-1111

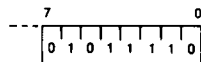
111



**Format 15.7**

Note 2	-0000	Note 2	-1000
Note 1	-0001	Note 1	-1001
Note 3	-0010	Note 3	-1010
Note 3	-0011	Note 1	-1011
Note 2	-0100	Note 2	-1100
Note 1	-0101	Note 1	-1101
Note 2	-0110	Note 2	-1110
Note 1	-0111	Note 1	-1111

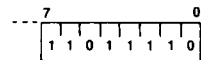
If nnn = 010, 011, 100, 110 then Trap (UND) Always.



TL/EE/10564-85

**Format 16**

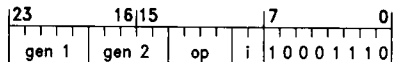
Trap (UND) Always



TL/EE/10564-86

**Format 17**

Trap (UND) Always



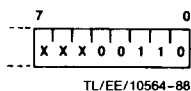
TL/EE/10564-87

**Format 18**

MULWD	-0000	CMACD	-0010
CMULD	-0001	MACTD	-0011

Trap (UND) on 1XXX, 01XX

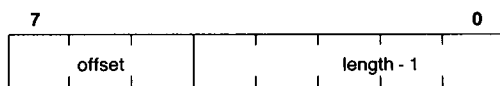
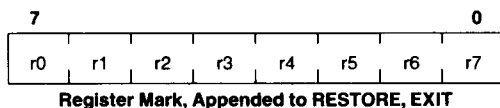
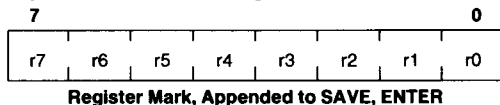
## Appendix A: Instruction Formats (Continued)



**Format 19**

Trap (UND) Always

**Implied Immediate Encodings:**



**Offset/Length Modifier Appended to INSS, EXTS**

**Note 1:** Opcode not defined; CPU treats like MOV<sub>r</sub> or CMOV<sub>c</sub>. First operand has access class of read; second operand has access class of write; f or c field selects 32- or 64-bit data.

**Note 2:** Opcode not defined; CPU treats like ADD<sub>r</sub> or CCAL<sub>c</sub>. First operand has access class of read; second operand has access class of read-modify-write; f or c field selects 32- or 64-bit data.

**Note 3:** Opcode not defined; CPU treats like CMP<sub>r</sub> or CCMP<sub>c</sub>. First operand has access class of read; second operand has access class of read; f or c field selects 32- or 64-bit data.

## Appendix B. Compatibility Issues

The NS32GX320 is compatible with the Series 32000 architecture implemented by the NS32532, NS32GX32 and previous microprocessors. Compatibility means that within certain limited constraints, programs that execute on one of the earlier Series 32000 microprocessors will produce identical results when executed on the NS32GX320. Compatibility applies to privileged operating systems programs, as well as to non-privileged applications programs. This appendix explains both the restrictions on compatibility with previous Series 32000 microprocessors and the extensions to the architecture that are implemented by the NS32GX320.

### B.1 RESTRICTIONS ON COMPATIBILITY

If the following restrictions are observed, then a program that executes on an earlier microprocessor of the Series 32000 architecture will produce identical results when executed on the NS32GX320 in an appropriately configured system:

1. The program is not time-dependent. For example, the program should not use instruction loops to control real-time delays.
2. The program does not use any encodings of instructions, operands, addresses, or control fields identified to be reserved or undefined. For example, if the count operand's value for an LSHi instruction is not within the range specified by the *Series 32000 Instruction Set Ref-*

erence Manual, then the results produced by the NS32GX320 may differ from those of the NS32032.

3. The program does not depend on the use of a Memory Management Unit (MMU).
4. The program does not depend on the detection of bus errors according to the implementation of the NS32332. For example, the NS32GX320 distinguishes between restartable and nonrestartable bus errors by transferring control to the appropriate bus-error exception service procedure through one of two distinct entries in the Interrupt Dispatch Table. In contrast, the NS32332 uses a single entry in the Interrupt Dispatch Table for all bus errors.
5. The program does not modify itself. Refer to Section B.4 for more information.
6. The program does not depend on the execution of certain complex instructions to be non-interruptible. Refer to Section B.5 on "Memory-Mapped I/O" for more information.
7. The program does not use the custom slave instructions CATSTO and CATST1, as they are not supported by the NS32GX320 and will result in a Trap (UND) when their execution is attempted.

### B.2 ARCHITECTURE EXTENSIONS

The NS32GX320 implements the following extensions of the Series 32000 architecture using previously reserved control bits, instruction encodings, and memory locations. Extensions implemented earlier in the NS32332, such as 32-bit addressing, are not listed.

1. The DC, LDC, IC, and LIC bits in the CFG register have been defined to control the on-chip Instruction and Data Caches. The DE-bit in the CFG register has been defined to enable Direct-Exception Mode.
2. The V-flag in the PSR register has been defined to enable the Integer-Overflow Trap.
3. The DCR, BPC, DSR, and CAR registers have been defined to control debugging features. Access to these registers has been added to the definition of the LPR and SPR instructions.
4. Access to the CFG and SP1 registers has been added to the definition of the LPR and SPR instructions.
5. The CINV instruction has been defined to invalidate control of the on-chip Instruction and Data Caches.
6. The instructions MULWD, CMULD, CMACD and MACTD have been defined to improve DSP support.
7. Direct-Exception Mode has been added to support faster interrupt service time and systems without module tables.
8. A new entry has been added to the Interrupt Dispatch Table for supporting vectors to distinguish between restartable and nonrestartable bus errors. Two additional entries support Trap (OVF) and Trap (DBG).

### B.3 INTEGER OVERFLOW TRAP

A new trap condition is recognized for integer arithmetic overflow. Trap (OVF) is enabled by the V-flag in the PSR. This new trap is important because detection of integer overflow conditions is required for certain programming languages, such as ADA, and the PSR flags do not indicate the occurrence of overflow for ASHi, Divi and MULi instructions.

## Appendix B. Compatibility Issues (Continued)

More details on integer overflow are given in Section 3.2.5, where a description of all the cases in which an overflow condition is detected is also provided.

### INTEGER ARITHMETIC

The V-flag in the PSR enables Trap (OVF) to occur following execution of an integer arithmetic instruction whose result cannot be represented exactly in the destination operand's location.

If the number of bits required to represent the resulting quotient of a DEI instruction exceeds half the number of bits of the destination, then the contents of both the quotient and remainder stored in the destination are undefined.

The ADDR instruction can be used in place of integer arithmetic instructions to perform certain calculations. In this case however, integer overflow is not detected by the CPU.

### LOGICAL INSTRUCTIONS

The V-flag in the PSR enables Trap (OVF) to occur following execution of an ASHi instruction whose result cannot be represented exactly in the destination operand's location.

### ARRAY INSTRUCTIONS

The V-flag in the PSR enables Trap (OVF) to occur following execution of a CHECKi instruction whose source operand is out of bounds.

### PROCESSOR CONTROL INSTRUCTIONS

The V-flag in the PSR enables Trap (OVF) to occur following execution of an ACBi instruction if the sum of the "inc" value and the "index" operand cannot be represented exactly in the "index" operand's location.

### DSP INSTRUCTIONS

The V-flag in the PSR enables Trap (OVF) to occur following execution of a CMULD, CMACD or MACTD instruction if either the final result or the result of any intermediate calculation is longer than a double word.

### B.4 SELF-MODIFYING CODE

The Series 32000 architecture does not have special provisions to optimally support self-modifying programs. Nevertheless, on the NS32332 and previous Series 32000 microprocessors it is possible to execute self-modifying code according to the following sequence:

1. Modify the appropriate instruction.
2. Execute a JUMP instruction or other instruction that causes the microprocessor's instruction queue to be flushed.
3. Execute the modified instruction.

For example, an interactive debugger may follow the sequence above after reaching a breakpoint in a program being monitored.

The same program may not produce identical results when executed on the NS32GX320 due to effects of the Instruction Cache and branch prediction. In order to execute self-modifying code on the NS32GX320 it is necessary to do the following:

1. Modify the appropriate instruction.
2. If the modified instruction is on a cacheable page, execute CINV to invalidate the contents of the Instruction Cache.

3. Execute an instruction that causes a serializing operation. See Section 3.1.3.3.
4. Execute the modified instruction.

### B.5 MEMORY-MAPPED I/O

As was mentioned in Section 3.1.3.2, certain peripheral devices exhibit characteristics identified as "destructive-reading" and "side-effects of writing" that impose requirements for special handling of memory-mapped I/O references. The NS32GX320 supports two methods to use on references to memory-mapped peripheral devices that exhibit either or both of these characteristics.

For peripheral devices that exhibit only side-effects of writing, correct operation can be ensured either by locating the device between addresses FF000000 (hex) and FF7FFFFF (hex) in the address space or by observing the first 2 restrictions listed below. For peripheral devices that exhibit destructive-reading, all the following restrictions must be observed to ensure correct operation:

1. References to the device must be inhibited while the CPU asserts the output signal  $\overline{IOINH}$ .
2. The input signal  $\overline{IODEC}$  must be asserted by the system on references to the device.
3. The device cannot be used for instruction fetches, reads of effective addresses.
4. If an instruction that reads a source operand from the device crosses a page boundary, then no Trap (ABT) or restartable bus error can occur during fetches from the page with higher addresses.
5. The device can be used as a source operand only for instructions in the list below.

ABSi	CBITi	MOVMi	SBITi
ADDi	CBITi	MOVXi	SUBi
ADDCi	CMPI	MOVZi	SUBCi
ADDPi	CMPI	NEGi	SUBPi
ADDQi	COMi	NOTi	TBITi
ANDi	IBITi	ORi	XORi
ASHi	LSHi	ROTi	
BICi	MOVi	SBITi	

This restriction arises because the CPU can respond to interrupt requests during the execution of complex instruction in order to reduce interrupt latency. Thus, the CPU may read the source operands for a DEID instruction (extended-precision divide), begin calculating the instruction's results, and then respond to an interrupt request before completing the instruction. In such an event, the instruction can be executed again and completed correctly after the interrupt service procedure returns unless one of the source operands was altered by destructive-reading.

## Appendix C. Instruction Set Extensions

The following sections describe the differences and extensions to the Series 32000 instruction set (as presented in the "Series 32000 Instruction Set Reference Manual") implemented by the NS32GX320.

## Appendix C. Instruction Set Extensions (Continued)

### C.1 PROCESSOR SERVICE INSTRUCTIONS

The CFG register, User Stack Pointer (SP1), and Debug Registers can be loaded and stored using privileged forms of the LPRi and SPRI instructions.

When the SETCFG instruction is executed, the CFG register bits 0 through 3 are loaded from the instruction's short field, bits 4 through 7 are forced to 1, and bits 8 through 12 are forced to 0.

The contents of the on-chip Instruction Cache and Data Cache can be invalidated by executing the privileged instruction CINV. While executing the CINV instruction, the CPU generates 2 slave bus cycles on the system interface to display the first 3 bytes of the instruction and the source operand.

### C.2 INSTRUCTION DEFINITIONS

This section provides a description of the operations and encodings of the new NS32GX320 privileged instructions.

#### Load and Store Processor Registers

**Syntax:** LPRi      procreg,      src  
                                 short              gen  
   read.i

                 SPRI      procreg      dest  
                                 short              gen  
   write.i

The LPRi and SPRI instructions can be used to load and store the User Stack Pointer (USP or SP1), the Configuration Register (CFG) and the Debug Registers in addition to the Processor Registers supported by the previous Series 32000 CPUs. Access to these registers is privileged.

Figure C-1 and Table C-1 show the instruction formats and the new 'short' field encodings for LPRi and SPRI.

**Flags Affected:** No flags affected by loading or storing the USP, CFG, or Debug Registers.

**Traps:** Illegal Instruction Trap (ILL) occurs if an attempt is made to load or store the USP, CFG or Debug Registers while the U-flag is 1.

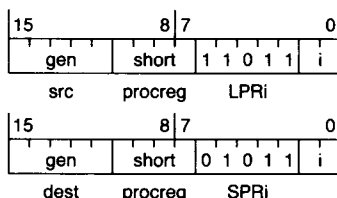


FIGURE C-1. LPRi/SPRI Instruction Formats

TABLE C-1. LPRi/SPRI New 'Short' Field Encodings

Register	procreg	short field
Debug Condition Register	DCR	0001
Breakpoint Program Counter	BPC	0010
Debug Status Register	DSR	0011
Compare Address Register	CAR	0100
User Stack Pointer	USP	1011
Configuration Register	CFG	1100

#### Cache Invalidate

**Syntax:** CINV      [options], src  
                                 gen  
                                 read. D

The CINV instruction invalidates the contents of locations in the on-chip Instruction Cache and Data Cache. The instruction can be used to invalidate either the entire contents of the on-chip caches or only a 16-byte block. In the latter case, the 28 most-significant bits of the source operand specify the physical address of the aligned 16-byte block; the 4 least-significant bits of the source operand are ignored. If the specified block is not located in the on-chip caches, then the instruction has no effect. If the entire cache contents is to be invalidated, then the source operand is read, but its value is ignored.

Options are specified by listing the letters A (invalidate All), I (Instruction Cache), and D (Data Cache). If neither the I nor D option is specified, the instruction has no effect.

In the instruction encoding, the options are represented in the A, I, and D fields as follows:

- A: 0—invalidate only a 16-byte block  
      1—invalidate the entire cache
- I: 0—do not affect the Instruction Cache  
      1—invalidate the Instruction Cache
- D: 0—do not affect the Data Cache  
      1—invalidate the Data Cache

**Flags Affected:** None

**Traps:** Illegal Operation Trap (ILL) occurs if an attempt is made to execute this instruction while the U-flag is 1.

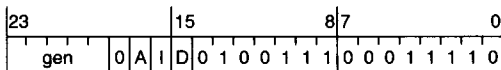
#### Examples:

1. CINV [A, D, I], R3      1E A7 1B
2. CINV [I], R3              1E 27 19

Example 1 invalidates the entire Instruction Cache and Data Cache.

Example 2 invalidates the 16-byte block whose physical address in the Instruction Cache is contained in R3.

## Appendix C. Instruction Set Extensions (Continued)



src options CINV  
FIGURE C-2. CINV Instruction Format

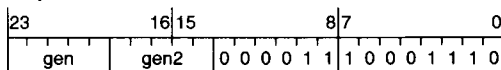
### Multiply Word to Double

**Syntax:** MULWD src, dest  
gen gen  
read. W rmw. D

The MULWD instruction reads a word source operand and a double-word destination operand, and multiplies source and the lower 16 bits of destination signed 16-bit numbers to form a signed 32-bit result in destination. Overflow does not occur in this instruction.

**Flags Affected:** None

**Traps:** None



src DST MULWD  
FIGURE C-3. MULWD Instruction Format

### Complex Multiply Double

**Syntax:** CMULD src1, src2  
gen gen  
read. D read. D

The CMULD instruction reads two double-word source operands, representing complex numbers. Each operand consists of a signed 16-bit real part in the low-order word, and a signed 16-bit imaginary part in the high-order word. The result consists of a signed 32-bit real part in R0, and a signed 32-bit imaginary part in R1, as follows:

**Assuming:** src1 = (A2, A1)  
src2 = (B2, B1)

**The CMULD result is:**  $R0 \leftarrow A1*B1 - A2*B2$   
 $R1 \leftarrow A1*B2 + A2*B1$

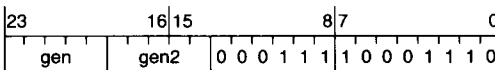
If the result of any of the operations shown above is longer than a double-word, then the high-order bits are truncated, and Overflow Trap (OVF) occurs if the V-flag in the PSR is set to 1. The F-flag is not affected. Note that the order of the instruction's intermediate operations is implementation-dependent, and is not necessarily as presented above.

### Restrictions:

The two instructions following a CMULD, CMACD or MACTD instruction (see below) must not access the R0 or R1 registers, unless there was either: 1) a serializing operation after the execution of the instruction and before the access of any of these registers, or 2) the instruction is followed by another CMULD, CMACD or MACTD that does not use any of these registers as a source operand. If this condition is violated, the result is unpredictable. Note that interrupt handlers can access R0 and R1 upon entry, because an interrupt acknowledge involves a serializing operation.

**Flags Affected:** None

**Traps:** Trap (OVF) occurs if the V-flag is set to 1, and an overflow occurred in any of the intermediate operations performed by the instruction.



src1 src2 CMULD  
FIGURE C-4. CMULD Instruction Format

### Complex Multiply and Accumulate Double

**Syntax:** CMACD src1, src2  
gen gen  
read. D read. D

The CMACD instruction reads two double-word source operands, representing complex numbers. Each operand consists of a signed 16-bit real part in the low-order word, and a signed 16-bit imaginary part in the high-order word. The result consists of a signed 32-bit real part in R0, and a signed 32-bit imaginary part in R1, as follows:

**Assuming:** src1 = (A2, A1)  
src2 = (B2, B1)

**The CMACD result is:**  $R0 \leftarrow R0 + A1*B1 - A2*B2$   
 $R1 \leftarrow R1 + A1*B2 + A2*B1$

If the result of any of the operations shown above is longer than a double-word, then the high-order bits are truncated, and Overflow Trap (OVF) occurs if the V-flag is set to 1. The F-flag is not affected. Note that the order of the instruction's intermediate operations is implementation-dependent, and is not necessarily as presented above.

## Appendix C. Instruction Set Extensions (Continued)

**Restrictions:** Same as for the CMULD instruction above.

**Flags Affected:** None

**Traps:** Trap (OVF) occurs if PSR V-bit is set to 1, and an overflow occurred in any of the intermediate operations performed by the instruction.

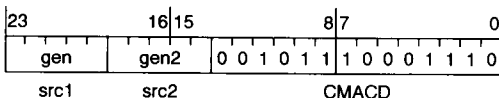


FIGURE C-5. CMACD Instruction Format

### Multiply and Accumulate Twice Double

**Syntax:** MACTD src1, src2  
gen gen  
read. D read. D

The MACTD instruction reads two double-word source operands, where each operand consists of two signed 16-bit values. The result is a signed 32-bit value in R0, as follows:

**Assuming:** src1 = (A2, A1)  
src2 = (B2, B1)

**The MACTD result is:**  $R0 \leftarrow R0 + A1*B1 + A2*B2$   
If the result of any of the operations shown above is longer than a double-word, then the high-order bits are truncated, and Overflow Trap (OVF) occurs if the V-flag is set to 1. The F-flag is not affected. Note that the order of the instruction's intermediate operations is implementation-dependent, and is not necessarily as presented above.

**Restrictions:** Same as for the CMULD instruction above.

**Flags Affected:** None

**Traps:** Trap (OVF) occurs if the V-flag is set to 1, and an overflow occurred in any of the intermediate operations performed by the instruction.

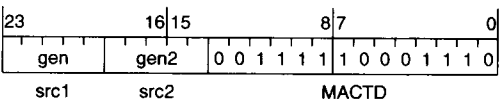


FIGURE C-6. MACTD Instruction Format

## Appendix D. Instruction Execution Times

The NS32GX320 achieves its performance by using an advanced implementation incorporating a 4-stage Instruction Pipeline, an Instruction Cache and a Data Cache into a single integrated circuit.

As a consequence of this advanced implementation, the performance evaluation for the NS32GX320 is more complex than for the previous microprocessors in the Series 32000 family. In fact, it is no longer possible to determine the execution time for an instruction using only a set of tables for operations and addressing modes. Rather, it is necessary to consider dependencies between the various instructions executing in the pipeline, as well as the occurrence of misses for the on-chip caches.

The following sections explain the method to evaluate the performance of the NS32GX320 by calculating various timing parameters for an instruction sequence. Due to the high degree of parallelism in the NS32GX320, the evaluation techniques presented here include some simplifications and approximations.

### D.1 INTERNAL ORGANIZATION AND INSTRUCTION EXECUTION

The NS32GX320 is organized internally as 8 functional units as shown in Figure 1. The functional units operate in parallel to execute instructions in the 4-stage pipeline. The structure of this pipeline is shown in Figure 3-2. The Instruction Fetch and Instruction Decode pipeline stages are implemented in the loader along with the 8-byte instruction queue and the buffer for a decoded instruction. The Address Calculation pipeline stage is implemented in the address unit. The Execute pipeline stage is implemented in the Execution Unit along with the write data buffer that holds up to two results directed to memory.

The Address Unit and Execution Unit can process instructions at a peak rate of 2 clock cycles per instruction, enabling a sustained pipeline throughput at 30 MHz of 15 MIPS (million instructions per second) for sequences of register-to-register, immediate-to-register, memory-to-register instructions and register-to-memory. Nevertheless, the execution of instructions in the pipeline is reduced from the peak throughput of 2 cycles by the following causes of delay:

1. Complex operations, like division, require more than 2 cycles in the Execution Unit, and complex addressing modes, like memory relative, require more than 2 cycles in the Address Unit.
2. Dependencies between instructions can limit the flow through the pipeline. A data dependency can arise when the result of one instruction is the source of a following instruction. Control dependencies arise when branching instructions are executed. Section D.3 describes the types of instruction dependencies that impact performance and explains how to calculate the pipeline delays.
3. Cache misses can cause the flow of instructions through the pipeline to be delayed, as can non-aligned references. Section D.4 explains the performance impact for these forms of storage delays.

The effective time  $T_{eff}$  needed to execute an instruction is given by the following formula:

$$T_{eff} = T_e + T_d + T_s$$

$T_e$  is the execution time in the pipeline in the absence of data dependencies between instructions and storage delays,  $T_d$  is the delay due to data dependencies, and  $T_s$  is the effect of storage delays.

### D.2 BASIC EXECUTION TIMES

Instruction flow in sequence through the pipeline stages implemented by the Loader, Address Unit, and Execution Unit. In almost all cases, the Loader is at least as fast at decoding an instruction as the Address Unit is at processing the instruction. Consequently, the effects of the Loader can be ignored when analyzing the smooth flow of instructions in the pipeline, and it is only necessary to consider the times for the Address Unit and Execution Unit. The time required by the Loader to fetch and decode instructions is significant



## Appendix D. Instruction Execution Times (Continued)

only when there are control dependencies between instructions or Instruction Cache misses, both of which are explained later.

The time for the pipeline to advance from one instruction to the next is typically determined by the maximum time of the Address Unit and Execution Unit to complete processing of the instruction on which they are operating. For example, if the Execution Unit is completing instruction  $n$  in 2 cycles and the Address Unit is completing instruction  $n+1$  in 4 cycles, then the pipeline will advance in 4 cycles. For certain instructions, such as RESTORE, the Address Unit waits until the Execution Unit has completed the instruction before proceeding to the next instruction. When such an instruction is in the Execution Unit, the time for the pipeline to advance is equal to the sum of the time for the Execution Unit to complete instruction  $n$  and the time for the Address Unit to complete instruction  $n+1$ . The processing times for the Loader, Address Unit, and Execution Unit are explained below.

### D.2.1 Loader Timing

The Loader can process an instruction field on each clock cycle, where a *field* is one of the following:

- An opcode of 1 to 3 bytes including addressing mode specifiers.
- Up to 2 index bytes, if scaled index addressing mode is used.
- A displacement.
- An immediate value of 8, 16 or 32 bits.

The Loader requires additional time in the following cases:

- 1 additional cycle when 2 consecutive double-word fields begin at an odd address.
- 2 cycles in total to process a double-precision floating-point immediate value.

### D.2.2 Address Unit Timing

The processing time of the Address Unit depends on the instruction's operation and the number and type of its general addressing modes. The basic time for most instructions is 2 cycles. A relatively small number of instructions require an additional address unit time, as shown in the timing tables in Section D.5.5. Floating-point instructions as well as Custom-Slave instructions require an additional 3 cycles plus 2 cycles for each quad-word operand in memory.

For instructions with 2 general addressing modes, 2 additional cycles are required when both addressing modes refer to memory. Certain general addressing modes require an additional processing time, as shown in Table D-1. For example, the instruction **MOVD 4(8(FP)), TOS** requires 7 cycles in the Address Unit; 2 cycles for the basic time, an additional 2 cycles because both modes refer to memory, and an additional 3 cycles for Memory Relative addressing mode.

**TABLE D-1. Additional Address Unit Processing Time for Complex Addressing Modes**

Mode	Additional Cycles
Memory Relative	3
External	8
Scaled Indexing	2

### D.2.3 Execution Unit Timing

The Execution Unit processing times for the various NS32GX320 instructions are provided in Section D.5.5. Certain operations cause a break in the instruction flow through the pipeline.

Some of these operation simply stop the Address Unit, while others flush the instruction queue as well. The information on how to evaluate the penalty resulting from instruction flow breaks is provided in the following sections.

## D.3 INSTRUCTION DEPENDENCIES

Interactions between instructions in the pipeline can cause delays. Two types of interactions can arise, as described below.

### D.3.1 Data Dependencies

In certain circumstances the flow of instructions in the pipeline will be delayed when the result of an instruction is used as the source of a succeeding instruction. Such interlocks are automatically detected by the microprocessor and handled with complete transparency to software.

#### D.3.1.1 Register Interlocks

When an instruction uses a base register that is the destination of either of the previous 2 instructions, a delay occurs. Modifications of the Stack Pointer resulting from the use of TOS addressing mode do not cause any delay. Also, there is no delay for a data dependency when the instruction that modifies the register is one for which the Address Unit stops. The delay is 3 cycles when, as in the following example, the base register is modified by the immediately preceding instruction.

```
n:  ADDD R1,R0      ; modify R0
n+1: MOVD 4(R0),R2  ; R0 is base register,
                    delay 3 cycles
```

The delay is 1 cycle when the register is modified 2 instructions before its use as a base register, as shown in this example.

```
n:  ADDD R1,R0      ; modify R0
n+1: MOVD 4(SP),R3  ; R0 not used
n+2: MOVD 4(R0),R2  ; R0 is base register,
                    delay 1 cycle
```

## Appendix D. Instruction Execution Times (Continued)

When an instruction uses an index register that is the destination of the previous instruction, a delay of 1 cycle occurs, as shown in the example below. If the register is modified 2 or more instructions prior to its use as an index register, then no delay occurs.

```
n:  ADDD R1,R0      ; modify R0
n+1: MOVD 4(SP)[R0:B],R2
                        ; R0 is index register
                        delay 1 cycle
```

Bypass circuitry in the Execution Unit generally avoids delay when a register modified by one instruction is used as the source operand of the following instruction, as in the following example.

```
n:  ADDD R1,R0      ; modify R0
n+1: MOVD R0,R2      ; R0 is source register,
                        no delay
```

For the uncommon case where the operand in the source register is larger than the destination of the previous instruction, a delay of 2 cycles occurs. Here is an example.

```
n:  ADDB R1,R0      ; modify byte in R0
n+1: MOVD R0,R2      ; R0 dw source operand,
                        2 cycle delay
```

**Note:** The Address Unit does not make any differentiation between CPU and FPU registers. Therefore, register interlocks can occur between integer and floating-point instructions.

### D.3.1.2 Memory Interlocks

When an instruction reads a source operand (or address for effective address calculation) from memory that depends on the destination of either of the previous 2 instructions, a delay occurs. The CPU detects a dependency between a read and a write reference in the following cases, which include some false dependencies in addition to all actual dependencies:

- Either reference crosses a double-word boundary
- Address bits 0 through 11 are equal
- Address bits 2 through 11 are equal and either reference is for a word
- Address bits 2 through 11 are equal and either reference is for a double-word

The delay for a memory interlock is 4 cycles when, as in the following example, the memory location is modified by the immediately preceding instruction.

```
n:  ADDQD 1,4(SP)    ; modify 4(SP)
n+1: CMPFD 10,4(SP)  ; read, 4(SP),
                        4 cycle delay
```

The delay is 2 cycles when the memory location is modified 2 instructions before its use as a source operand or effective address, as shown in this example.

```
n:  ADDQD 1,4(SP)    ; modify 4(SP)
n+1: MOVD R0,R1      ; no reference to 4(SP)
n+2: CMPFD 10, 4(SP) ; read 4(SP),
                        2 cycles delay
```

Certain sequences of read and write references can cause a delay of 1 cycle although there is no data dependency between the references. This arises because the Data Cache is occupied for 2 cycles on write references. In the absence of data dependencies, read references are given priority over write references. Therefore, this delay only occurs when an instruction with destination in memory is followed 2 instructions later by an instruction that refers to memory (read or write) and 3 instructions later by an instruction that reads from memory. Here is an example:

```
n:  MOVD R0,4(SP)    ; memory write
n+1: MOVD R6,R7      ; any instruction
n+2: MOVD 8(SP),R0    ; memory read or write
n+3: MOVD 12(SP),R1   ; memory read
                        delayed 1 cycle
```

### D.3.2 Control Dependencies

The flow of instructions through the pipeline is delayed when the address from which to fetch an instruction depends on a previous instruction, such as when a conditional branch is executed. The Loader includes special circuitry to handle branch instructions (ACB, BR, Bcond, and BSR) that serves to reduce such delays. When a branch instruction is decoded, the Loader calculates the destination address and selects between the sequential and non-sequential instruction streams. The non-sequential stream is selected for unconditional branches. For conditional branches the selection is based on the branch's direction (forward or backward) as well as the tested condition. The branch is predicted taken in any of the following cases.

- The branch is backward.
- The tested condition is either NE or LE.

Measurements have shown that the correct stream is selected for 64% of conditional branches and 71% of total branches.

If the Loader selects the non-sequential stream, then the destination address is transferred to the Instruction Cache. For conditional branches, the Loader saves the address of the alternate stream (the one not selected). When a conditional branch instruction reaches the Execution Unit, the condition is resolved, and the Execution Unit signals the Loader whether or not the branch was taken. If the branch had been incorrectly predicted, the Instruction Cache begins fetching instructions from the correct stream.

The delay for handling a branch instruction depends on whether the branch is taken and whether it is predicted correctly. Unconditional branches have the same delay as correctly predicted, taken conditional branches.

Another form of delay occurs when 2 consecutive conditional branch instructions are executed. This delay of 2 cycles arises from contention for the register that holds the alternate stream address in the Loader.

Control dependencies also arise when JUMP, RET, and other non-branch instructions alter the sequential execution of instructions.

### D.4 STORAGE DELAYS

The flow of instructions in the pipeline can be delayed by off-chip memory references that result from misses in the on-chip storage buffers and by misalignment of instructions

## Appendix D. Instruction Execution Times (Continued)

and operands. These considerations are explained in the following sections. The delays reported assume no wait states on the external bus and no interference between instruction and data references.

### D.4.1 Instruction Cache Misses

An Instruction Cache miss causes a 5 cycle gap in the fetching of instructions. When the miss occurs for a non-sequential instruction fetch, the pipeline is idle for the entire gap, so the delay is 5 cycles. When the miss occurs for a sequential fetch, the pipeline is not idle for the entire gap because instructions that have been prefetched ahead and buffered can be executed. The delay for misses on non-sequential instruction fetches can be estimated to be approximately half the gap, or 2.5 cycles.

### D.4.2 Data Cache Misses

A Data Cache miss causes a delay of 2 cycles. When a burst read cycle is used to fill the cache block, then 3 additional cycles are required to update the Data Cache. In case a burst cycle is used and either of the 2 instructions following the instruction that caused the miss also reads from memory, then an additional delay occurs: 3 cycle delay when the instruction that reads from memory immediately follows the miss, and 2 cycle delay when the memory read occurs 2 instructions after the miss.

### D.4.3 Instruction and Operand Alignment

When a data reference (either read or write) crosses a double-word boundary, there is a delay of 2 cycles.

When the opcode for a non-sequential instruction crosses a double-word boundary, there is a delay of 1 cycle. No delay occurs in the same situation for a sequential instruction.

## D.5 EXECUTION TIME CALCULATIONS

This section provides the necessary information to calculate the  $T_e$  portion of the effective time required by the CPU to execute an instruction.

The effects of data dependencies and storage delays are not taken into account in the evaluation of  $T_e$ , rather, they should be separately evaluated through a careful examination of the instruction sequence.

The following assumptions are made:

- The entire instruction, with displacements and immediate operands, is present in the instruction queue when needed.
- All memory operands are available to the Execution Unit and Address Unit when needed.
- Memory writes are performed at full speed through the write buffer.
- Where possible, the values of operands are taken into consideration when they affect instruction timing, and a range of times is given. When this is not done, the worst case is assumed.

### D.5.1 Definitions

$T_{eu}$  Time required by the Execution Unit to execute an instruction.

$T_{au}$  Total processing time in the Address Unit.

$T_{ad}$  Extra time needed by the Address Unit, in addition to the basic time, to process more complex cases.  $T_{ad}$  can be evaluated as follows:

$$T_{ad} = T_x + T_{y1} + T_{y2}$$

$T_x = 2$  if the instruction has two general operands and both of them are in memory.

0 otherwise.

$T_{y1}$  and  $T_{y2}$  are related to operands 1 and 2 respectively. Their values are given below.

$T_{y(1,2)} = 3$  if Memory Relative

8 if External

2 if Scaled Indexing

0 if any other addressing mode

The following parameters are only used for floating-point execution time calculations.

$T_{anp}$  Additional Address Unit time needed to process floating-point instructions (Section D.2.2).  $T_{anp}$  can be calculated as follows:

$$T_{anp} = 3 + 2 * (\text{Number of 64-bit operands in memory})$$

$T_{1cs}$  Time required to transfer ID and Opcode, if no operand needs to be transferred to the slave. Otherwise, it is the time needed to transfer the last 32 bits of operand data to the slave. In the latter case the transfer of ID and Opcode as well as any operand data except the last 32 bits is included in the Execution Unit timing.

$T_{1sc}$  Time required by the CPU to complete the floating-point instruction upon receiving the DONE signal from the slave. This includes the time to process the DONE signal itself in addition to the time needed to read the result (if any) from the slave.

$I$  This parameter is related to the floating-point operand size as follows:

Standard floating (32 bits):  $I = 0$

Long floating (64 bits):  $I = 1$

### D.5.2 Notes on Table Use

1. In the  $T_{eu}$  column the notation  $n1 \rightarrow n2$  means  $n1$  minimum,  $n2$  maximum.

2. In the notes column, notations held within angle brackets <> indicate alternatives in the operand addressing modes which affect the execution time. A table entry which is affected by the operand addressing may have multiple values, corresponding to the alternatives. This addressing notations are:

<I> Immediate

<R> CPU register

<M> Memory

<F> FPU register, either 32 or 64 bits

<m> Memory, except Top of Stack

<T> Top of Stack

<x> Any addressing mode

## Appendix D. Instruction Execution Times (Continued)

<ab> a and b represent the addressing modes of operands 1 and 2 respectively. Both of them can be any addressing mode. (e.g., <MR> means memory to CPU register).

3. The notation 'Break K' provides pipeline status information after executing the instruction to which 'Break K' applies. The value of K is interpreted as follows:

K = 0 The Address Unit was stopped by the instruction but the pipeline was not flushed. The Address Unit can start processing the next instruction immediately.

K > 0 The pipeline was flushed by the instruction. The Address Unit must wait for K cycles before it can start processing the next instruction.

K < 0 The Address Unit was stopped at the beginning of the instruction but it was restarted |K| cycles before the end of it. The Address Unit can start processing the next instruction |K| cycles before the end of the instruction to which 'Break K' applies.

4. Some instructions must wait for pending writes to complete before being able to execute. The number of cycles that these instructions must wait for, is between 6 and 7 for the first operand in the write buffer and 2 for the second operand, if any.

5. The CBITli and SBITli instructions will execute a RMW access after waiting for pending writes. The extra time required for the RMW access is only 3 cycles since the read portion is overlapped with the time in the Execution Unit.

6. The keyword defined for the Bcond instruction have the following meaning:

BTPC Branch Taken, Predicted Correctly

BTPI Branch Taken, Predicted Incorrectly

BNTPC Branch Not Taken, Predicted Correctly

BNTPI Branch Not Taken, Predicted Incorrectly

### D.5.3 T<sub>eff</sub> Evaluation

The T<sub>e</sub> portion of the effective execution time for a certain instruction in an instruction sequence is obtained by performing the following steps:

1. Label the current and previous instruction in the sequence with n and n-1 respectively.
2. Obtain from the tables the values of T<sub>eu</sub> and T<sub>au</sub> for instruction n and T<sub>eu</sub> for instruction n-1.
3. For floating-point instructions, obtain the values of T<sub>tcs</sub> and T<sub>tsc</sub>.
4. Use the following formula to determine the execution time T<sub>e</sub>.

$$T_e = \text{func}(T_{au}(n), T_{eu}(n-1), T_{flt}(n-1), \text{Break}(n-1)) + T_{eu}(n) + T_{flt}(n)$$

func provides the amount of processing time in the Address Unit that cannot be hidden. Its definition is given below.

$$0 \quad \text{if } T_{au}(n) \leq (T_{eu}(n-1) + T_{flt}(n-1))$$

AND NOT Break (n-1)

$$T_{au}(n) - T_{eu}(n-1) \quad \text{if } T_{au}(n) > (T_{eu}(n-1) + T_{flt}(n-1))$$

AND NOT Break (n-1)

$$T_{au}(n) + K \quad \text{if } (T_{au}(n) + K) > 0$$

AND Break (n-1)

$$0 \quad \text{if } (T_{au}(n) + K) \leq 0$$

AND Break (n-1)

K is the value associated with Break (n-1).

T<sub>flt</sub> only applies to floating-point instructions and is always 0 for other instructions. It is evaluated as follows:

$$T_{flt} = t_{tcs} + T_{tsc} + T_{fpu}$$

T<sub>fpu</sub> is the execution time in the Floating-Point Unit.

5. Calculate the total execution time T<sub>eff</sub> by using the following formula:

$$T_{eff} = T_e + T_d + T_s$$

Where T<sub>d</sub> and T<sub>s</sub> are dependent on the instruction sequence, and can be obtained using the information provided in Section D.4.

### D.5.4 Instruction Timing Example

This section presents a simple instruction timing example for a procedure that recursively evaluates the Fibonacci function. In this example there are no data dependencies or storage buffer misses; only the basic instruction execution times in the pipeline, control dependencies, and instruction alignment are considered.

The following is the source of the procedure in C.

```
unsigned fib(x)
int x;
{
    if (x > 2)
        return (fib(x-1) + fib(x-2));
    else
        return(1);
}
```

## Appendix D. Instruction Execution Times (Continued)

The assembly code for the procedure with comments indicating the execution time is shown below. The procedure requires 26 cycles to execute when the actual parameter is less than or equal to 2 (branch taken) and 99 cycles when the actual parameter is equal to 3 (recursive calls).

```
_fib:  movd    r3,tos    ; 2 cycles
      movd    r4,tos    ; 2 cycles
      movd    r1,r3     ; 2 cycles
      cmpq    $(2),r3   ; 2 cycles
      bge     .L1       ; 2 cycles, Break 2 If Branch Taken
      movd    r3,r1     ; 2 cycles
      addq    $(-2),r1  ; 2 cycles
      bsr     _fib      ; 3 cycles
      movd    r0,r4     ; 2 cycles + 4 Cycles due to RET
      movd    r3,r1     ; 2 cycles
      addq    $(-1),r1  ; 2 cycles
      bsr     _fib      ; 3 cycles
      add     r4,r0     ; 2 cycles + 1 cycle alignment + 4 cycles due to RET
      movd    tos,r4    ; 2 cycles
      movd    tos,r3    ; 2 cycles
      ret     $(0)      ; 4 cycles, break 4
      .align 4

.L1:  movq    $(1),r0    ; 4 cycles + 4 cycles due to BGE
      movd    tos,r4     ; 2 cycles
      movd    tos,r3     ; 2 cycles
      ret     $(0)      ; 4 cycles, Break 4
```

## Appendix D. Instruction Execution Times (Continued)

### D.5.5 Execution Timing Tables

The following tables provide the execution timing information for all the NS32GX320 instructions. The table for the floating-point instructions provides only the CPU portion of the total execution time. The FPU execution times can be found in the NS32381 datasheet.

#### D.5.5.1 Basic Instructions

Mnemonic	T <sub>eu</sub>	T <sub>au</sub>	Notes
ABSi	5	2 + T <sub>ad</sub>	
ACBi	5	2 + T <sub>ad</sub>	If incorrect prediction then Break 1
ADDi	2	2 + T <sub>ad</sub>	
ADDCi	2	2 + T <sub>ad</sub>	
ADDPi	9	2 + T <sub>ad</sub>	
ADDQi	2	2 + T <sub>ad</sub>	
ADDR	2	4 + T <sub>ad</sub>	
ADJSPi	5 3	2 + T <sub>ad</sub> 2 + T <sub>ad</sub>	i = B, W Break 0 i = D Break 0
ANDi	2	2 + T <sub>ad</sub>	
ASHi	7 5 9	2 + T <sub>ad</sub> 2 + T <sub>ad</sub> 2 + T <sub>ad</sub>	Right Left .AND. PSR-V Bit = 0 Left .AND. PSR-V Bit = 1
B <sub>COND</sub>	2 → 3 2 2 2	2 2 2 2	BTPC BTPI Break 2 BNTPC BNTPI Break 2 (see Note 5 in Section D.5.2)
BICi	2	2 + T <sub>ad</sub>	
BICPSRi	6	2 + T <sub>ad</sub>	Wait for pending writes. Break 5
BISPSRi	6	2 + T <sub>ad</sub>	Wait for pending writes. Break 5
BPT	30 21	2 2	Modular Direct Break 5
BR	2 → 3	2	
BSR	2 → 3	3 + T <sub>ad</sub>	
CASEi	7	2 + T <sub>ad</sub>	Break 5
CBITi	8 9	2 2 + T <sub>ad</sub>	<R> <M> Break 0
CBITli	18	2 + T <sub>ad</sub>	<M> Wait for pending writes. Execute interlocked RMW access. Break 5

Mnemonic	T <sub>eu</sub>	T <sub>au</sub>	Notes
CHECKi	10	2 + T <sub>ad</sub>	Break - 3. If SRC is out of bounds and the V bit in the PSR is set, then add trap time.
CINV	10	2 + T <sub>ad</sub>	Wait for pending writes. Break 5
CMACD	12	2 + T <sub>ad</sub>	
CMPi	2	2 + T <sub>ad</sub>	
CMPMi	6 + 8 * n		n = number of elements. Break 0
CMPQi	2	2 + T <sub>ad</sub>	
CMPSi	7 + 13 * n	2 + T <sub>ad</sub>	n = number of elements. Break 0
CMPST	6 + 20 * n	2 + T <sub>ad</sub>	n = number of elements. Break 0
CMULD	12	2 + T <sub>ad</sub>	
COMi	2	2 + T <sub>ad</sub>	
CVTP	5	4 + T <sub>ad</sub>	
CXP	17	13	Break 5
CXPD	21	11 + T <sub>ad</sub>	Break 5
DEli	28 + 4 * i	5 + T <sub>ad</sub>	i = 0/4/12 for B/W/D. Break 0
DIA	3	2	Break 5
DIVi	(30 → 40) + 4 * i	2 + T <sub>ad</sub>	i = 0/4/12 for B/W/D
ENTER	15 + 2 * n	3	n = number of registers saved. Break 0
EXIT	8 + 2 * n	2	n = number of registers restored

## Appendix D. Instruction Execution Times (Continued)

### D.5.5.1 Basic Instructions (Continued)

Mnemonic	T <sub>eu</sub>	T <sub>au</sub>	Notes
EXTi	12 13	8 8 + T <sub>ad</sub>	<R> <M> Break -3
EXSi	11 14	6 6 + T <sub>ad</sub>	<R> <M> Break -3
FFSi	11 + 3 * i	2 + T <sub>ad</sub>	i = number of bytes
FLAG	4 32 21	2 2 2	No trap Trap, Modular Trap, Direct If trap then: {wait for pending writes; Break 5}
IBITi	10 14	2 2 + T <sub>ad</sub>	<R> <M> If <M> then Break 0
INDEXi	16	5 + T <sub>ad</sub>	
INSi	15 18	8 8 + T <sub>ad</sub>	<R> <M>
INSSi	14 19	6 6 + T <sub>ad</sub>	<R> <M> Break 0
JSR	3	9 + T <sub>ad</sub>	Break 5
JUMP	3	4 + T <sub>ad</sub>	Break 5
LPRI	6  5  7	2 + T <sub>ad</sub>  2 + T <sub>ad</sub>  2 + T <sub>ad</sub>	CPU Reg = FP, SP, USP, SP, MOD. Break 0 CPU Reg = CFG, INTBASE, DSR, BPC, UPSR. Wait for pending writes. Break 5 CPU Reg = DCR, PSR CAR. Wait for pending writes. Break 5
LSHi	3	2 + T <sub>ad</sub>	
MACTD	11	2 + T <sub>ad</sub>	
MEIi	13 + 2 * i	5 + T <sub>ad</sub>	i = 0/4/12 for B/W/D. Break 0
MODi	(34 → 49) + 4 * i	2 + T <sub>ad</sub>	i = 0/4/12 for B/W/D
MOVi	2	2 + T <sub>ad</sub>	
MOVMI	5 + 4 * n	2 + T <sub>ad</sub>	n = number of elements. Break 0
MOVQi	2	2 + T <sub>ad</sub>	

Mnemonic	T <sub>eu</sub>	T <sub>au</sub>	Notes
MOVSi	12 + 4 * n 14 + 8 * n	2 + T <sub>ad</sub> 2 + T <sub>ad</sub>	n = number of elements. No options. B, W and/or U Options in effect. Break 0
MOVST	16 + 9 * n	2 + T <sub>ad</sub>	n = number of elements. Break 0
MOVXii	2	2 + T <sub>ad</sub>	
MOVZii	2	2 + T <sub>ad</sub>	
MULB	6 13	2 + T <sub>ad</sub> 2 + T <sub>ad</sub>	PSR-V Bit = 0 PSR-V Bit = 1
MULW	6 21	2 + T <sub>ad</sub> 2 + T <sub>ad</sub>	PSR-V Bit = 0 PSR-V Bit = 1
MULD	11 37 24	2 + T <sub>ad</sub> 2 + T <sub>ad</sub> 2 + T <sub>ad</sub>	PSR-V Bit = 0 PSR-V Bit = 1 PSR-V Bit = 1 .AND. 0 ≤ src ≤ 255
MULWD	6	2 + T <sub>ad</sub>	
NEGi	2	2 + T <sub>ad</sub>	
NOP	2	2	
NOTi	3	2 + T <sub>ad</sub>	
ORi	2	2 + T <sub>ad</sub>	
QUOI	(30 → 40) + 4 * i	2 + T <sub>ad</sub>	i = 0/4/12 for B/W/D
REMI	(32 → 42) + 4 * i	2 + T <sub>ad</sub>	i = 0/4/12 for B/W/D
RESTORE	7 + 2 * n	2	n = number of registers restored. Break 0
RET	4	3	Break 4
RETI	19 13 29 22	5 5 5 5	Noncascaded, Modular Noncascaded, Direct Cascaded, Modular Cascaded, Direct  Wait for pending writes. Break 5
RETT	14 8	5 5	Modular Direct  Wait for pending writes. Break 5
ROTi	5	2 + T <sub>ad</sub>	
RXP	8	5	Break 5
SCONDi	3	2 + T <sub>ad</sub>	
SAVE	8 + 2 * n	2	n = number of registers. Break 0
SBITi	8 9	2 2 + T <sub>ad</sub>	<R> <M> Break 0

## Appendix D. Instruction Execution Times (Continued)

### D.5.5.1 Basic Instructions (Continued)

Mnemonic	T <sub>eu</sub>	T <sub>au</sub>	Notes
SBITli	10 18	2 2 + T <sub>ad</sub>	<R> <M>  Wait for pending writes. Execute interlocked RMW access. Break 5
SETCFG	6	2	Break 5
SKPSi	8 + 6 * n	2 + T <sub>ad</sub>	n = number of elements. Break 0
SKPST	6 + 20 * n	2 + T <sub>ad</sub>	n = number of elements. Break 0
SPRi	5 3	2 + T <sub>ad</sub> 2 + T <sub>ad</sub>	CPU Reg = PSR, CAR CPU Reg = all others

Mnemonic	T <sub>eu</sub>	T <sub>au</sub>	Notes
SUBi	2	2 + T <sub>ad</sub>	
SUBCi	2	2 + T <sub>ad</sub>	
SUBPi	6	2 + T <sub>ad</sub>	
SVC	32 21	2 2	Modular Direct  Wait for pending writes. Break 5
TBITi	7 7	2 2 + T <sub>ad</sub>	<R> <M> If <M> then break 0
WAIT	3	2	Wait for pending writes. Wait for interrupt
XORi	2	2 + T <sub>ad</sub>	



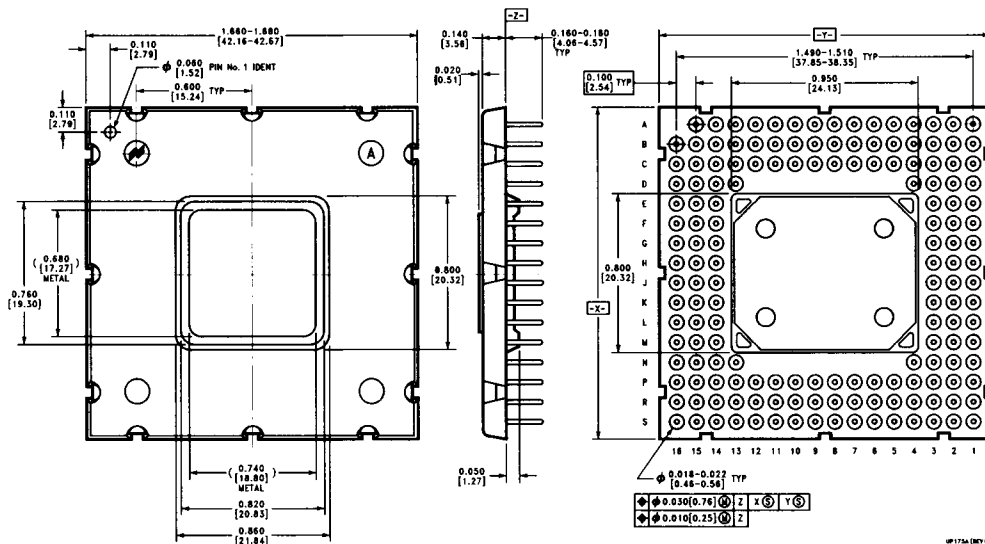
## Appendix D. Instruction Execution Times (Continued)

### D.5.5.2 Floating-Point Instructions, CPU Portion

Mnemonic	$T_{eu}$	$T_{au}$	$T_{tcs}$	$T_{tsc}$	Notes
MOVf, NEGf, ABSf, LOGBf	2	$2 + T_{anp}$	2	1	<FF>
	$4 + 3 * I$	$2 + T_{anp} + T_{ad}$	2	1	<MF>
	$6 + 3 * I$	$2 + T_{anp}$	2	1	<IF>
	$6 + 3 * I$	$2 + T_{anp}$	2	1	<TF>
	$11 + 4 * I$	$2 + T_{anp} + T_{ad}$	2	$3 + 2 * I$	<FM> Break – (1 + I)
	$13 + 7 * I$	$2 + T_{anp} + T_{ad}$	2	$3 + 2 * I$	<MM>, <IM> Break – (1 + I)
ADDf, SUBf, MULf, DIVf, SCALBf	2	$2 + T_{anp}$	2	1	<FF>
	$4 + 3 * I$	$2 + T_{anp}$	2	1	<MF>
	$6 + 3 * I$	$2 + T_{anp}$	2	1	<IF>
	$6 + 3 * I$	$2 + T_{anp}$	2	1	<TF>
	$17 + 7 * I$	$2 + T_{anp} + T_{ad}$	2	$3 + 2 * I$	<FM> Break – (1 + I)
	$19 + 10 * I$	$2 + T_{anp} + T_{ad}$	2	$3 + 2 * I$	<MM>, <IM> Break – (1 + I)
ROUNDfi, TRUNCfi, FLOORfi	11	$2 + T_{anp}$	2	$3 + 2 * I$	<FR> Break – 1
	$11 + 4 * I$	$2 + T_{anp} + T_{ad}$	2	$3 + 2 * I$	<FM> Break – (1 + I)
	13	$2 + T_{anp} + T_{ad}$	2	$3 + 2 * I$	<MR>, <IR> Break – 1
	$13 + 7 * I$	$2 + T_{anp} + T_{ad}$	2	$3 + 2 * I$	<MM>, <IM> Break – (1 + I)
CMPf	18	$2 + T_{anp}$	2		<FF>
	$20 + 3 * I$	$2 + T_{anp} + T_{ad}$	2		<MF>
	$23 + 3 * I$	$2 + T_{anp} + T_{ad}$	2		<FM>
	$25 + 6 * I$	$2 + T_{anp} + T_{ad}$	2		<MM>, <IM>, <MI>, <II> Break 3
POLYf, DOTf	2	$2 + T_{anp}$	2	1	<FF>
	$4 + 3 * I$	$2 + T_{anp} + T_{ad}$	2	1	<MF>
	$6 + 3 * I$	$2 + T_{anp}$	2	1	<IF>, <TF>
	$11 + 4 * I$	$2 + T_{anp} + T_{ad}$	2	1	<FM> Break – (1 + I)
	$13 + 7 * I$	$2 + T_{anp} + T_{ad}$	2	1	<MM>, <MI>, <IM>, <II> Break – (1 + I)
MOVfi	6	$2 + T_{anp}$	2	1	<RF>
	13	$2 + T_{anp} + T_{ad}$	2		<RM> Break – 1
	$6 + 3 * I$	$2 + T_{anp} + T_{ad}$	2	1	<MF>, <IF>, <TF>
	$13 + 7 * I$	$2 + T_{anp} + T_{ad}$	2		<MM>, <IM> Break – (1 + I)
LFSR	6	$2 + T_{anp}$	2	1	<R>
	$6 + 3 * I$	$2 + T_{anp} + T_{ad}$	2	1	<M>
	$6 + 3 * I$	$2 + T_{anp}$	2	1	<I>
	$6 + 3 * I$	$2 + T_{anp}$	2	1	<T>
SFSR	11	$2 + T_{anp} + T_{ad}$	2	3	Break – 1
MOVFL	4	$2 + T_{anp}$	2	1	<FF>
	6	$2 + T_{anp} + T_{ad}$	2	1	<MF>, <IF>, <TF>
	15	$2 + T_{anp} + T_{ad}$	2		<FM> Break 0
	17	$2 + T_{anp} + T_{ad}$	2		<MM>, <IM> Break 0
MOVLf	4	$2 + T_{anp}$	2	1	<FF>
	9	$2 + T_{anp} + T_{ad}$	2	1	<MF>, <IF>, <TF>
	15	$2 + T_{anp} + T_{ad}$	2		<FM> Break 0
	20	$2 + T_{anp} + T_{ad}$	2		<MM>, <IM> Break 0

# Physical Dimensions inches (millimeters)

Lit. # 114303



## Plastic Pin Grid Array (NU)

Order Number NS32GX320NU-20, NS32GX320NU-25 or NS32GX320NU-30  
NS Package Number NU175

## 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.

023740



**National Semiconductor Corporation**  
2900 Semiconductor Drive  
P.O. Box 58090  
Santa Clara, CA 95052-8090  
Tel: (800) 272-9959  
TWX: (910) 339-9240

**National Semiconductor GmbH**  
Industriestrasse 10  
D-8080 Furstenfeldbruck  
West Germany  
Tel: (041-41) 103-0  
Telex: 527-649  
Fax: (08141) 103554

**National Semiconductor Japan Ltd.**  
Saneido Bldg. 5F  
4-15 Nishi Shinjuku  
Shinjuku-Ku,  
Tokyo 160, Japan  
Tel: 3-299-7001  
FAX: 3-299-7000

**National Semiconductor Hong Kong Ltd.**  
Suite 513, 5th Floor  
Chinachem Golden Plaza,  
77 Mody Road, Tsimshatsui East,  
Kowloon, Hong Kong  
Tel: 3-7231290  
Telex: 52996 NSSEA HX  
Fax: 3-3112536

**National Semicondutores Do Brasil Ltda.**  
Av. Brig. Faria Lima, 1383  
6.0 Andor-Conj. 62  
01451 Sao Paulo, SP, Brasil  
Tel: (55/11) 212-5066  
Fax: (55/11) 211-1181 NSBR BR

**National Semiconductor (Australia) PTY, Ltd.**  
1st Floor, 441 St. Kilda Rd.  
Melbourne, 3004  
Victoria, Australia  
Tel: (63) 267-5006  
Fax: 61-3-2677458