630077-001

National Semiconductor

NS32CG16 Printer/Display Processor Programmer's Reference Supplement

Series 32000

NS32CG16 Printer/Display Processor

Programmer's Reference Supplement

CONTENTS

FIGURES

TABLES

INDEX

PREFACE

This document is intended to serve as a supplement to the *Series 32000® Programmer's Reference Manual* (Copyright 1987, Prentice-Hall, ISBN 0-13-806936-0). It describes the new instructions available on the NS32CG16.

As this is advance information, it is subject to change without notice.

Series 32000 is a registered trademark of National Semiconductor Corporation.

Postscript is a registered trademark of Adobe System, Inc.

VRTX is a registered trademark of Ready Systems.

Chapter 1

NS32CG16 PRINTER/DISPLAY PROCESSOR

Chapter 1

NS32CG16 PRINTER/DISPLAY PROCESSOR

1.1 INTRODUCTION

This document describes the NS32CG16 graphics instructions, provides programmers with the knowledge to use those instructions, and serves as a supplement to the *Series 32000 Programmer's Reference Manual*. Both documents are required to review the complete instruction set of the NS32CG16.

1.2 OVERVIEW OF THE NS32CG16 PRINTER/DISPLAY PROCESSOR

The NS32CG16 is a new member of the *Series 32000* processor family. A full 32-bit CMOS design combines high speed with low-power consumption and provides full compatibility with existing *Series 32000* software. An on-chip clock generator and programmable clock divider (/1, /2, /4, and /8) permit power consumption to be reduced even further in a ''power-down'' mode.

Series 32000's two-address, symmetrical architecture provides optimal support for high-level languages. Fully optimizing compilers are available for C, Pascal, FORTRAN, and Ada®.

Graphics-oriented instructions provide support for Bit-aligned Block Transfer functions, line drawings, pattern replications, and data compression-expansion. Very fast BITBLT operations are supported by the optional DP8510 BITBLT Processing Unit. Full support for the NS32081 and NS32381 floating-point units allows for fast Postscript® Outline Font calculation. Combined with the *Series 32000* general-purpose instructions, the NS32CG16 provides a compact, high-speed solution for many applications.

1.3 APPLICATIONS

The NS32CG16's 32-bit general-purpose instruction set, coupled with the graphics-specific support, gives high performance to many applications including:

- Laser Printers
- LCS Printers
- LED Printers

- Matrix Printers
- Ink Jet Printers
- PC Displays
- Terminals
- Plotters
- FAX
- Scanners
- Embedded Controllers

Chapter 2

NS32CG16 INSTRUCTION SET

Chapter 2

NS32CG16 INSTRUCTION SET

2.1 INTRODUCTION

This chapter describes the NS32CG16 Instruction Set and Instruction Definitions.

2.2 NS32CG16 INSTRUCTION SET

GRAPHICS INSTRUCTIONS

BBAND	Bit-aligned Block Transfer (4-direction AND operation for CRT applications)
BBFOR	Bit-aligned Block Transfer (2-direction OR operation optimized for Printers)
BBOR	Bit-aligned Block Transfer (4-direction OR operation)
BBSTOD	Bit-aligned Block Transfer (4-direction replace operation)
BBXOR	Bit-aligned Block Transfer (4-direction XOR operation)
BITWT	Bit-aligned Word Transfer (for small BITBLT OR operations)
CBITi gen,gen	Clear specified bit
CVTP gen,gen	Convert to bit pointer
EXTBLT	Drives DP8510/DP8511 BITBLT Processing Unit, (4-direction, 16-function, very fast BITBLT)
FFSi gen,gen	Find first set bit
IBITi gen,gen	Invert specified bit
INDEXi reg,gen,gen	X, Y to bit address
MOVMPi	Move Multiple Pattern (pattern file, horizontal line drawing, memory clear)
SBITi gen,gen	Set specified bit

SBITPS	Set Bit Perpendicular String (image expansion, horizontal/vertical/diagonal line drawing)
SBITS	Set Bit String (image expansion, horizontal line drawing)
TBITi gen,gen	Test specified bit
TBITS	Test Bit String (image compression)

EXTBLT (DP8510/11 added functions

0	Zeros to destination
-s AND -d	Not source and not destination
-s AND d	Not source and destination
-s	Not source to destination
s AND -d	Source and not destination
-d	Not destination to destination
s XORD d	Source xor destination
-s or -d	Not source or not destination
s AND d	Source and destination
s XNOR d	Source xnor destination
d	Destination to destination
-s OR d	Not source or destination
s	Source to destination
s OR -d	Source or not destination
s OR d	Source or destination
1	Ones to destination

General Instructions

ABSi gen,gen	Absolute value
ACBi short,gen,disp	Add 4-bit constant, branch if not zero
Addi gen,gen	Add
ADDCi gen,gen	Add with carry
ADDPi gen,gen	Add BCD packed
ADDQi short,gen	Add signed 4-bit constant
ADDR gen,gen	Move effective address
ADJSPi gen	Adjust stack pointer
ANDi gen,gen	Logical AND
ASHi gen,gen	Arithmetic shift (left/right)

Bcond disp	Conditional Branch
BICi gen,gen	Clear selected bits
BICPSRi gen	Clear selected bits in PSR
BISPRSRi gen	Set selected bits in PSR
BPT	Breakpoint trap
BR disp	Branch
BSR disp	Branch to subroutine
CASEi gen	Multiway branch
CHECKi reg,gen,gen	Index bounds check
CMPi gen,gen	Compare
CMPMi gen,gen,disp	Compare multiple
CMPQi short,gen	Compare to signed 4-bit constant
CMPSi (options)	Compare string
CMPSTi (options)	Compare string with translate
COMi gen,gen	Complement all bits
CXP disp	Call external procedure
CXPD gen	Call external procedure using descriptor
DEI gen,gen	Divide extended integer
DIA	Diagnose (hardware breakpoint feature)
DIVi gen,gen	Divide, round down
ENTER [reg list],disp	Save registers, allocate frame
EXIT [reg list]	Restore registers, reclaim frame
EXTi gen,gen,gen,dsp	Extract bit field
EXTSi gen,gen,im,im	Extract bit field
FLAG	Flag trap
INSi reg,gen,gen,disp	Insert bit field
INSSI gen,gen,im,im	Insert bit field
JSR gen	Jump to subroutine
JUMP gen	Jump
LPRi areg,reg	Load dedicated register
LSHi gen,gen	Logical shift (left/right)
MEIi gen,gen	Multiply extended integer
MODi gen,gen	Remainder from DIV
MOVi gen,gen	Move a value
MOVMi gen,gen,disp	Move multiple
MOVQi short,gen	Extend signed 4-bit constant
MOVSi (options)	Move string

MOVSTi (options)	Move string with translate
MOVXBD gen,gen	Move and sign extend
MOVXBW gen,gen	Move and sign extend
MOVXWD gen,gen	Move and sign extend
MOVZBD gen,gen	Move and zero extend
MOVZBW gen,gen	Move and zero extend
MOVZWD gen,gen	Move and zero extend
MULi gen,gen	Multiply
NEGi gen,gen	Negate
NOP	No operation
NOTi gen,gen	Boolean complement
ORi gen,gen	Logical OR
QUOi gen,gen	Divide, round toward zero
RESTORE [reglist]	Restore general registers
RET disp	Return from subroutine
RETI	Return from interrupt
RETT disp	Return from trap
ROTi gen,gen	Rotate (left/right)
RXP disk	Return from external procedure call
SAVE [reglist]	Save Rn registers
Scond gen	Save condition as boolean
SETCFG [options]	Set configuration register, select power-down mode, Floating-point, Interrupt controller
SKPSi (options)	Skip string
SKPSTi (options)	Skip string with translate
SPRi areg,gen	Save dedicated register
SUBi gen,gen	Subtract
SUBCi gen,gen	Subtract with carry
SUBPi gen,gen	Subtract BCD packed
SVC	Supervisor call
WAIT	Wait for interrupt
XORi gen,gen	Logical XOR

Floating-Point Instructions (NS32081)

ABSf gen,gen	Take absolute value
ADDf gen,gen	Add
CMPf gen,gen	Compare

DIVf gen,gen	Divide
FLOORfi gen,gen	Convert to largest integer
LFSR gen	Load FSR
MOVf gen,gen	Move floating-point value
MOVFL gen,gen	Lengthen Float value to Long
MOVif gen,gen	Convert integer to Float/Long
MOVLF gen,gen	Shorten Long value to Float
MULf gen,gen	Negate
ROUNDfi gen,gen	Convert to integer (rounding)
SFSR gen	Store FSR
SUBf gen,gen	Subtract
TRUNCfi gen,gen	Convert to integer (truncating)

Floating-Point Instructions (NS32381 additions)

DOTf gen1,gen2	L0 = L0 + (gen1 * gen2)
LOGBf gen1,gen2	Move exponent of gen1 to gen2
POLYf gen1,gen2	L0 = (L0 * gen1) + gen2
SCALBf gen1,gen2	Move gen2 * 2^{gen1} to gen2

Multiprocessor support

CBITIi gen,gen	Clear bit specified (interlocked)
SBITIi gen,gen	Set bit specified (interlocked)

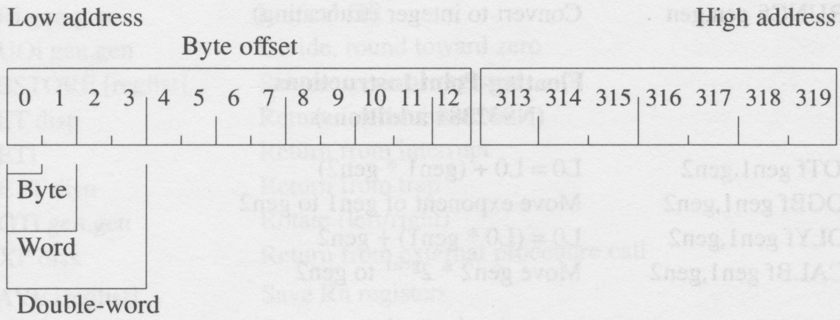
Addressing modes

Register	Rn/Fn/Ln
Register relative	disp(Rn)
Memory relative	disp(disp(FP/SP/SB))
Immediate	value
Absolute	disp
External	EXT(disp1) + disp2
Top of stack	TOS
Memory space	disp(FP/SP/SB/PC)
Scaled index	mode[Rn:B/W/D/Q]
Data types (i/f)	**B**yte (8 bit), **W**ord (16 bit), **D**ouble-word (32 bit), **F**loat (32 bit floating-point), **L**ong (64 bit floating-point)

2.3 CONVENTIONS

This document assumes the displayed image is in a specific format in memory. The following information explains the terminology and directional assumptions, as well as byte and bit ordering conventions in *Series 32000*.

The following diagram represents one scan line of a standard 8½ by 11-inch page on a 300 Dot Per Inch (DPI) laser printer, in the portrait orientation (8½ by 11 inches). There are 3300 such scan lines on each page. The start of the second scan line on the page is at byte offset 320 decimal, 140 hexadecimal. Since the first scan line is at the top of the page, successive scan lines proceed down the page.



HD-01-0-U

All *Series 32000* processors have 32-bit internal data paths, with a ''natural'' size of 32 bits or 4 bytes. The least significant byte is stored at the lowest address. Referring to the previous example, writing a byte of hexadecimal A5 to address zero results in address zero containing hexadecimal A5. Writing a word of hexadecimal A55A to address zero results in address zero containing hexadecimal 5A and address 1 containing hexadecimal A5. Writing a double-word of hexadecimal FFA55A00 to address zero would result in address zero containing hexadecimal 00, address 1 containing hexadecimal 5A, address 2 containing hexadecimal A5, and address 3 containing hexadecimal FF.

Series 32000 does **not** have an alignment restriction since data of byte, word or double-word size need not reside at an even memory address. The Bus Interface Unit, internal to all *Series 32000* processors, requests multiple bus transfers as required, aligning the data automatically.

The bit offset is equally consistent. Bit ordering is always from the least significant to the most significant bit. In the diagram above, bit zero of byte zero is the first pixel imaged on the page. Bit 1 is the next pixel, bit 2 the next, and so on. Bit 2549 is the last pixel imaged on the page in the horizontal direction, since 8½ inches * 300 DPI yields a width of 2550 dots or pixels. Bit 2549 is contained within byte 318, at bit position 5. Both Bit Addressing (*e.g.*, SBITD

2549,page) and Byte Addressing with a byte address and a bit offset (*e.g.*, SBITD 5,page+318) are available in *Series 32000*. The next diagram is an expansion of the first 3 bytes of a scan line, showing bit addressing, as it appears on the page printer or graphics screen.

Byte 0	Byte 1	Byte 2
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23

HD-02-0-U

To clarify these conventions, the following example illustrates how a line 1 dot high and 10 dots wide is drawn. This line appears on scan line one, starting at the ninth pixel, or bit position 8. This results in hexadecimal FF in address 1, and hexadecimal 03 in address 2. This is referred to as the *horizontal* direction.

The memory width of an image is referred to as the image warp (sometimes referred to as raster or pitch). The warp of the page printer image in the previous example is 320 decimal (140 hexadecimal) bytes or 2560 bits. Note that the image width is actually 2550 on this sample page printer at 300 DPI, since 8½ inches * 300 DPI yields a width of 2550 dots. The width is rounded up to 2560 bits (320 bytes) to make memory addressing simpler in a typical hardware design.

When the warp is known, vertical lines can be drawn. A horizontal line 10 dots high and 1 dot wide starting at the first line, ninth pixel, with a warp of 320 (140 hexadecimal) and a base address of 0 results in addresses 1 (1 hexadecimal), 321 (141 hexadecimal), 641 (281 hexadecimal) ... 2881 (B41 hexadecimal), each containing 01 hexadecimal.

To summarize, for portrait applications, the ''top left'' pixel is bit zero. The ''top right'' pixel is bit 2549. The ''bottom left'' pixel is bit 8,445,440. The ''bottom right'' pixel is bit 8,447,989. To calculate x,y bit positions on the page, the formula:

$$\text{Bit offset} = (y * 2560) + x$$

may be used, where y is the scan line number ranging from 0 to 3299, for the sample 8½ by 11 inch page, and x is the pixel displacement across the page from the left-hand edge.

More information on *Series 32000* conventions is located in the *Series 32000 Programmer's Reference Manual* and the *Series 32000 Instruction Set Reference Manual*. See Section 3.5 in the *Instruction Set Reference Manual* for a complete discussion of bit instructions and bit addressing.

All *shifts* in this document are in the increasing bit/byte direction. This is a *Series 32000* Left Shift. However, visually on a page, this is a left-to-right shift. Zeros are inserted in the least-significant bits.

All *rotates* in this document are in the increasing bit/byte direction. This is a *Series 32000* Left Rotate. However, visually on a page, this is a left-to-right shift. Data from the most significant bit position will be transferred to the least significant bit position.

Note that all NS32CG16 new instructions are in *Series 32000* Format 5, shown below.

INSTRUCTION FORMAT 5

0 0 0 0 0	*short*	0	opcode	i	0 0 0 0 1 1 1 0
23	16	15	8	7	0

HD-03-0-U

2.4 NS32CG16 COMMENTS

Since the NS32CG16 is not designed for use with the NS32082 or the 32382 Memory Management Units (MMUs) or with custom slave processors, the following instructions, are not available on the NS32CG16. Execution of these instructions causes an undefined instruction trap. Note that real-time operating systems, such as VRTX™ of Ready Systems, will operate without these instructions.

INSTRUCTION	PURPOSE
LMR	Load MMU register
SMR	Store MMU register
RDVAL	Validate address for reading
WRVAL	Validate address for writing
MOVSUi	Move value from supervisor to user space
MOVUSi	Move value from user to supervisor space
CCAL0c	Custom slave
CCAL1c	Custom slave
CCAL2c	Custom slave
CCAL3c	Custom slave
CMOV0c	Custom slave
CMOV1c	Custom slave
CMOV2c	Custom slave
CMOV3c	Custom slave
CCMP0c	Custom slave compare
CCMP1c	Custom slave compare
CCV0ci	Custom slave
CCV1ci	Custom slave
CCV2ci	Custom slave
CCV3ci	Custom slave
CCV4DQ	Custom slave
CCV5QD	Custom slave
LCSR	Custom slave
SCSR	Custom slave
CATST0	Privileged custom slave
CATST1	Privileged custom slave
LCR	Custom slave
SCR	Custom slave

In addition, four new instructions are present in the NS32CG16 for use with the NS32381 Floating-Point Unit (FPU). These new instructions are SCALBf, LOGB*f*, DOT*f* and POLY*f*. These operators increase the performance of transcendental operations.

The SETCFG instruction now operates as the clock divider controlled by the former C and M bits of the CFG register as follows, assuming an input clock of 15 MHz (30 MHz crystal):

C	M	SCALE	EFFECTIVE CLOCK
0	0	DIV 1	15 MHz
0	1	DIV 2	7.5 MHz
1	0	DIV 4	3.75 MHz
1	1	DIV 8	1.88 MHz

2.5 INSTRUCTION DEFINITIONS

This section defines the individual NS32CG16 instructions. The instructions are ordered alphabetically by their general mnemonic form. For help in interpreting the information presented here, see the beginning of Chapter 6 in the *Series 32000 Programmer's Reference Manual*.

Syntax: **BB***function option(s)*

where *function* is one of the following:

- OR
- FOR (Fast OR)
- AND
- XOR
- STOD

and *option(s)* is one of the following:

D	1 indicates decreasing address (right-to-left)
_	0 indicates increasing address (left-to-right)
S	1 requests inversion of source data
	0 requests no inversion of source data

In the instruction syntax, the D option can be DA for Decreasing Address (right-to-left), or IA for Increasing Address (left-to-right). The default is IA. The S option can be S which causes the source to be operated on as is, or -S which causes the source to be inverted before it is operated upon. The default is S. Options are separated by commas. The DA and IA options must not occur in the same instruction. Similarly, the S and -S must not occur in the same instruction.

BITBLT Series

0 0 0 0 0	0 D X S *short*	0	opcode	i	0 0 0 0 1 1 1 0
23	16	15		8	7 0

HD-04-0-U

The X field is zero for BBOR and BBXOR and one for the BBAND instruction.

The *opcode* and *i* fields indicate the type of BITBLT operation:

FUNCTION	OPTIONS	OPCODE	*i*
BBOR	Yes	0110	01
BBFOR	No	1100	01
BBXOR	Yes	1110	01
BBAND	Yes	1010	11
BBSTOD	Yes	0100	01

Note that no options are available for the BBFOR (Fast Or) instruction. This instruction is optimized for a left-to-right BITBLT operation. Operands of the BITBLT Series instruction are specified in all eight general-purpose registers and on the top of the current stack as follows:

R0 — Base Address of Source Data (32 bits)

R1 — Base Address of Destination Data (32 bits)

R2 — Shift Value in bits (8 bits, unsigned)

R3 — Height in lines (32 bits, unsigned)

R4 — First Mask (16 bits)

R5 — Second Mask (16 bits)

R6 — Adjusted Source Warp in bytes (32 bits)

R7 — Adjusted Destination Warp in bytes (32 bits)

0(sp) — Width in words (16 bits, unsigned)

The *first mask* corresponds to the left mask in a left-to-right operation and a right mask in a right-to-left operation. The *second mask* corresponds to the right mask in a left-to-right operation and a left mask in a right-to-left operation. The first mask is applied to the first word operated on in each line, and the second mask is applied to the last word in each line.

For left-to-right BITBLT operations, the Source Warp must be adjusted to:

(Source Warp - 2 bytes/word * (width - 1))

and the Destination Warp must be adjusted to:

(Destination Warp - 2 bytes/word * (width - 1))

For right-to-left BITBLT operations, the Source Warp must be adjusted to:

(Source Warp + 2 bytes/word * (width - 1))

and the Destination Warp must be adjusted to:

(Destination Warp + 2 bytes/word * (width - 1))

Blocks of data up to 65,535 words wide and up to 4 Gigalines high may be operated on.

Description: The BITBLT Series executes a full two-operand, bit-aligned block transfer (BITBLT) using AND, OR, or XOR as operators. The BBFOR instruction differs only in that it does not have the source inversion or decrementing address options. BBFOR is optimized for speed in printing applications.

The BITBLT instructions have the ability to manipulate in height, based on the maximum values of 0(sp) and R3 respectively. The NS32CG16 External Address lines limit the total number of words operated on to a maximum of 8M Words (16 Mbytes).

A bottom-to-top BITBLT operation can be performed by using negative Source and Destination Warp values and pointing the initial Source and Destination Addresses to the bottom of the block of data. A right-to-left BITBLT operation can be performed by implementing the DA option, as opposed to the default IA option. This means BITBLT instructions can manipulate blocks of data beginning in any of the four corners of the block.

When a BITBLT instruction is initiated, the contents of the Frame Pointer (FP) register are saved on the current stack. The width of each line in words is then loaded into the FP register from 4(sp). It is known as the current width until all BITBLT operations have been performed on the current line. Once the current width has been decremented to zero, the Adjusted Source Warp in R6 is added to the Source Address in R0, the Adjusted Destination Warp in R7 is added to the Destination Address in R1, and the height in R3 is decremented by one. The

BITBLT operation is repeated until the height is zero. At the beginning of each iteration, which corresponds to the beginning of a new line, the Frame Pointer register is reloaded with the original width from 4(sp). After the last word of the block has been manipulated, the contents of the Frame Pointer register are restored.

The following paragraphs describe how data is manipulated by BITBLT instructions within a given line. In FOR, OR and XOR, a word is read from the Source Address, inverted if the -S option is set, zero-extended to 32 bits, ANDed with the first mask, and then shifted to the left the number of bit positions specified in R2. In AND, a word is read from the Source Address, inverted if the -S option is set, combined with a high-order word filled with ones to create a temporary 32-bit value, ORed with the first mask, and rotated to the left the number of bit positions specified in R2. A double-word is read from the Destination Address, and the function specified in the BITBLT instruction is performed on the shifted source and the destination value. The result is then written into the Destination Address.

The Source and Destination pointers are incremented or decremented by two, depending on how the DA option is set, and the current width is decremented by one. If the current width is zero, the ''inner loop'' and second mask steps described below are skipped.

If the current width is greater than one, the inner loop of the BITBLT is initiated. In this loop no masks are applied. In FOR, OR and XOR, a word is read from the Source Address, inverted if the -S option is set, zero-extended to 32 bits, and shifted to the left the number of bit positions specified in R2. In AND, a word is read from the Source Address, combined with a high-order word filled with ones to create a temporary 32-bit value, and rotated to the left the number of bit positions specified in R2. A double-word is read from the Destination Address, and the function specified in the BITBLT instruction is performed on the shifted source and the Destination value. The result is then written into the Destination Address.

The Source and Destination pointers are incremented or decremented by two, depending on how the DA option is set, and the current width is decremented by one. This process repeats until the width is one.

When the width is one, indicating that only one word remains to be transferred on the current line, the second mask is applied. In FOR, OR and XOR, a word is read from the Source Address, inverted if the -S option is set, zero-extended to 32 bits, ANDed with the second mask, and shifted to the left the number of bit positions specified in R2. In AND, a word is read from the Source Address, inverted if the -S option is set, combined with a high-order word filled with ones to create a temporary 32-bit value, ORed with the second mask and rotated to the left the number of bit positions specified in R2. A double-word is read from the Destination Address, and the function specified in the BITBLT instruction is performed on the shifted Source and the Destination value. The result is then written into the Destination Address.

The BBSTOD instruction flows like the BITBLT Series instructions. In the BBSTOD instruction, the Destination bits are cleared through masks prior to writing the masked and shifted Source Data. To perform this, three masks internal to BBSTOD instruction are created. The first internal mask is made as follows:

Rotate the contents of R4 by the value in R2 and complement the result.

The second internal mask is FFFF0000, rotated by the value in R2. This mask is used for the inner BBSTOD loop. The third internal mask is made as follows:

Rotate the contents of R5 by the value in R2 and complement it.

The first and third internal masks are used along with the first mask (R4) and the second mask (R5) respectively.

NOTE: The mask registers (R4 and R5) must have the upper 16 bits of the register set to zero. This may be accomplished with the MOVZWD instruction, either when the masks are created (read from a table), or immediately preceding the instruction (with MOVZWD R4,R4; MOVZWD R5,R5).

Example: The following example describes how the data is manipulated by the BBSTOD instruction within a given line.

Operands		Operand Values Before				After			
R0 - Source address		A000				A006			
R1 - Destination address		B000				B280			
R2 - Shift amount		4				4			
R3 - Height		1				0			
R4 - First mask		0FFFC				0FFFC			
R5 - Second mask		03FFF				03FFF			
R6 - Source warp		A				A			
R7 - Destination warp		1A				1A			
0(sp) - Width in words		4				4			
		LSB			MSB	LSB			MSB
	A000 - A003	FF	DD	CC	BB	FF	DD	CC	BB
	A004 - A007	AA	99	88	FF	AA	99	88	FF
	B000 - B003	27	54	32	57	E7	DF	CD	BC
	B004 - B007	39	87	21	E5	AB	9A	89	F8
	B008 - B009	A9	C4			AB	C4		

This example operates on a block of data, four words wide and one line long. Values indicated within parentheses are the running results. Positive Warp and No Inversion are used in this example.

1. A word is read from the Source Address (DDFF), zero extended to 32 bits (0000DDFF), ANDed with the first mask R4 (0000DDFC), and then shifted the number of bit positions specified in R2 (000DDFC0).
2. The first internal mask is made by rotating the first mask R4 by the number of bits positions specified in R2 (000FFFC0) and complementing the result (FFF0003F). A double-word is read from Destination Address (57325427), ANDed with the first internal mask (57300027).
3. The results from step one (000DDFC0) and step two (57300027) are ORed (573DDFE7) and then written into the destination address.

4. The Source and Destination pointers are incremented by two (R0=A002 R1=B002), and the width is decremented by one (FP=3). Since the width is not zero, the block of data operated on is greater than one word. The width is again decremented by one (FP=2) to determine whether the next word operated on is the last word or not. If not, it performs the inner BBSTOD. In this example, since the width is not zero after the second decrement, the instruction performs the inner BBSTOD.

5. A word is read from the Source Address (BBCC), zero extended to 32 bits (0000BBCC), and then shifted the number of bit positions specified in R2 (000BBCC0).

6. The second internal mask is made by rotating FFFF0000 the number of bit positions specified in R2 (FFF0000F) and stored in NS32CG16 internal register for future use. A double-word is read from the Destination Address (8739573D) and ANDed with the second internal mask (8730000D).

7. The results from step five (000BBCC0) and step six (8730000D) are ORed (873BBCCD) and then written into the destination address.

8. The Source and Destination pointers are incremented by two (R0=A004 R1=B004), and the current width is decremented by 1 (FP=1) to determine whether the next word operated on is the last word or not. If not, it performs the inner BBSTOD. In this example, since the width is not zero, the instruction performs the inner BBSTOD.

9. A word is read from the Source Address (99AA), zero extended to 32 bits (000099AA), and then shifted the number of bit positions specified in R2 (00099AA0).

10. The second internal mask is read from the NS32CG16 internal register (FFF0000F). A double-word is read from the Destination Address (E521873B) and then ANDed with the second internal mask (E520000B).

11. The results from step 9 (00099AA0) and step 10 (E520000B) are ORed (E5299AAB) and then written into the destination address.

12. The Source and Destination pointers are incremented by two (R0=A006 R1=B006), and the current width is decremented by zero (FP=0) to determine whether the next word operated on is the last word or not. Since the width is zero, the next word is the last word.

13. A word is read from the Source Address (FF88), zero extended to 32 bits (0000FF88), ANDed with the first mask R5 (00003F88) and then shifted the number of bit positions specified in R2 (0003F880).

14. The third internal mask is made by rotating the third mask R5 by the number of bit positions specified in R2 (0003FFF0) and complementing the result (FFFC000F). A double-word is read from Destination Address (C4A9E529) and then ANDed with the third internal mask (C4A80009).

15. The results from step 13 (0003F880) and step 14 (C4A80009) are ORed (C4ABF889) and then written into the destination address.

16. The Source and Destination pointers are incremented by their corresponding warps (R0=A010 R1=B010), and the height is decremented by one (R3=3). Since the height is zero, the BBSTOD instruction is complete.

Flags Affected: None

Traps: None; however, the instruction is interruptible.

Example: BBAND -S (BITBLT -S AND D)

This example operates on a block of data three bytes wide and two lines high. BBAND masks and shifts the inverted source value, then ANDs it with the destination value.

This example manipulates the block of data from left-to-right and top-to-bottom. However, all BITBLT instructions can begin manipulation from any of the four corners of the block of data.

Operands	Operand Values Before	Operand Values After
R0 - Source address	A000	A006
R1 - Destination address	B000	B280
R2 - Shift amount	2	2
R3 - Height	2	0
R4 - First mask	0003	0003
R5 - Second mask	FFC0	FFC0
R6 - Source warp	1	1
R7 - Destination warp	13E	13E
0(sp) - Width in words	2	2
	LSB MSB	LSB MSB
A000 - A002	12 34 56	12 34 56
A003 - A005	78 9A BC	78 9A BC
B000 - B003	FF FF FF FF	BF 2F A7 FF
B140 - B143	FF FF FF FF	1F 96 0D FF

BBOR (BITBLT S OR D)

This example operates on the same block of Source Data as the previous example. BBOR masks and shifts the Source value and then ORs it with the Destination value. This is the most common BITBLT operation in printer applications.

Operands	Operand Values Before				After			
R0 - Source Address	A000				A006			
R1 - Destination Address	B000				B280			
R2 - Shift amount	2				2			
R3 - Height	2				0			
R4 - First Mask	F800				F800			
R5 - Second Mask	00FF				00FF			
R6 - Source Warp	1				1			
R7 - Destination Warp	13E				13E			
0(sp) - Width in words	2				2			
	LSB			MSB	LSB			MSB
A000 - A002	12	34	56		12	34	56	
A003 - A005	78	9A	BC		78	9A	BC	
B000 - B003	00	00	00	00	00	C0	58	01
B140 - B143	00	00	00	00	00	60	F2	02

Application: These instructions are fundamental to most graphics-oriented programs that run on laser printers and graphics workstations. The instructions make proportional spacing of characters possible. The BBFOR instruction is especially well-suited for high-speed character imaging in the printer.

NOTES: 1. Each BITBLT instruction must be immediately preceded by CMPB R2,0 in order to set the L bit of the PSR correctly. This allows the instructions to optimize a shift of zero bits.

2. Only values from 0 to 15 should be used for the shift amount in R2. Values outside these ranges cause undefined results.

3. A BITBLT operation one word wide applies to only the first mask.

4. BITBLT used in an interrupt routine should be preceded by a BICPSRB $0X18 to clear the J and the K bits of the PSR register.

BITWT Bit Aligned Word Transfer

Syntax: BITWT

		BITWT			
0 0 0 0 0	0 0 0 0 *short*	0	1 0 0 0 opcode	0 1 i	0 0 0 0 1 1 1 0
23	16	15		8	7 0

HD-06-0-U

Operands of the BITWT instruction are specified in the following general purpose registers:

R0 Base Address of the Source Data (32 bits)

R1 Base Address of the Destination Data (32 bits)

R2 Shift Value in bits (8 bits, unsigned)

Description: The BITWT instruction performs a bit-aligned transfer of one word of data. The word at the Source Address is read, zero extended to 32 bits, then shifted to the left the number of bit positions specified in R2. The double-word at the Destination Address is read and ORed with the shifted source. The result is written into the Destination Address. The Source and Destination pointers are then incremented by two to point to the next words to be operated upon. This instruction assumes that the Source Data is ''clean'' and does not require masking.

Flags Affected: None

Traps: None

Example: BITWT

This example shifts the Source word by the value in R2, ORs it with the Destination double-word, then writes the result into the Destination Address.

Operands	Operand Values							
	Before				**After**			
R0	A000				A002			
R1	B000				B002			
R2	2				2			
					LSB			MSB
A000 - A001	51	6A			51	6A		
B000 - B003	30	52	24	62	74	FB	25	62

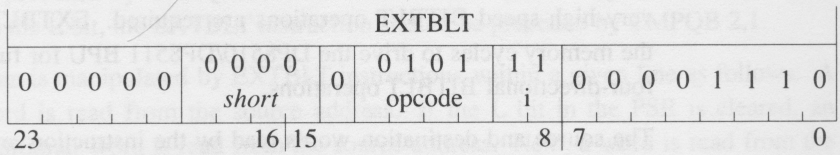
Application: The BITWT instruction is very fast and is intended for use in page-printer applications. Since the source font data can be aligned, no masks are required. A series of BITWT instructions, separated only by simple ADD instructions to adjust the source and destination warps, can implement a high-speed S OR D BITBLT operation by eliminating the speed penalty incurred by masking the data. This instruction is designed for BITBLTing from a Postscript-styled character cache to the composition space.

NOTE: The BITWT instruction must be immediately preceded by CMPB R2,0 in order to correctly set the L bit of the PSR. This allows the instruction to optimize a shift of zero bits.

If the Shift Value is zero, only a word of Destination Data is read, modified and written back, rather than a double-word size.

Only positive values in the range of 0 to 15 should be used for the Shift amount in R2. Values outside this range cause undefined results.

Syntax: EXTBLT



HD-07-0-U

Operands of the EXTBLT instruction are specified in the following general purpose registers:

R0 — Base Address of the Source Data (32 bits)

R1 — Base Address of the Destination Data (32 bits)

R2 — Width (32 bits, signed)

R3 — Height in lines (32 bits, unsigned)

R4 — Horizontal Increment in bytes (32 bits)

R5 — Current Width (32 bits, signed)

R6 — Adjusted Source Warp (32 bits)

R7 — Adjusted Destination Warp (32 bits)

The width must be adjusted to:

(width in words of destination data * horizontal increment [R4])

The source warp must be adjusted to:

(source warp - (width in bytes - 2))

and the destination warp must be adjusted to:

(destination warp - (width in bytes - 2))

Description: This instruction requires the DP8510/DP8511 bit-aligned Block Transfer Processing Unit (BPU) to be in the system in order to execute correctly.

The DP8510/DP8511 BPU and the EXTBLT instruction should be used when very high-speed BITBLT operations are required. EXTBLT implements all of the memory cycles to drive the DP8510/DP8511 BPU for full two-dimensional, four-directional BITBLT operations.

The source and destination words read by the instruction are transferred to the DP8510/8511. The BPU then provides the result that is written to the Destination Address. The NS32CG16 supplies addresses and bus cycles while the DP8510/11 operates on the data.

When the EXTBLT instruction is initiated, the contents of the Frame Pointer (FP) register are saved on the current stack. Next, the NBPU output signal on the NS32CG16 is asserted. The width of each line in bytes is then loaded into R5 from R2. The contents of R5 are known as the Current Width until all EXTBLT operations have been performed on the current line. Once the current width has been decremented to zero, the Adjusted Source Warp in R6 is added to the Source Address in R0, the Adjusted Destination Warp in R7 is added to the Destination Address in R1, and the height in R3 is decremented by one. The EXTBLT operation is repeated until the height is zero. At the beginning of each iteration, which corresponds to the beginning of a new line, R5 is reloaded with the original width from R2. After the last word of the block of data has been transferred, the NBPU signal on the NS32CG16 is deasserted and the contents of the Frame Pointer register are restored.

Because data can be only read and written on word boundaries, if the starting bit offset of the destination data is less than the starting bit offset of the source data, then two Read operations may be required to capture all of the source data, and an operation called preloading should be performed.

Preloading should take place if the starting bit offset of the destination data is less than the starting bit offset of the source data and

$$n + \text{offset} > 16$$

where *n* is the number of bits to be transferred from source to destination, and offset is the starting bit offset of the source data.

If the L bit in the PSR is cleared, preloading occurs, which activates the DP8510/DP8511 pipeline. To clear the L bit, the EXTBLT instruction should be preceded by CPMQB 1,2. If the L bit is set, preloading does not occur. To set the L bit, the EXTBLT instruction should be preceded by CMPQB 2,1.

Data is manipulated by EXTBLT instructions within a given line as follows: A word is read from the source address. If the L bit in the PSR is cleared, an additional word is read from the source address. Next, a word is read from the destination address. These entities are passed to the BPU via hardware. The BPU, in turn, directly presents a word to be written to the destination address. The horizontal increment is then added to both the source address and the destination address and subtracted from the current width. This inner loop is repeated until the current width is zero.

Flags Affected: None

External Signals: NBPU is low only during bus cycles involving pre-fetching instructions and execution of EXTBLT operands. It is recommended that NBPU, NADS and status lines (ST0-3) be used to qualify BPU bus cycles. If a DMA circuit exists in the system, the HLDA signal should be used to further qualify BPU cycles. NBPU may become active during T4 of a non-BPU bus cycle and may become inactive during T4 of a BPU bus cycle. NBPU must be qualified by NADS and status lines (ST0-3) to be used as an external gating signal.

Traps: None; however, the instruction is interruptible.

Interrupts are permitted only at the completion of Read-Read-Write cycles of EXTBLT. When the instruction is interrupted, the NBPU signal is deasserted prior to any non-BPU data Read/Write cycles and remains deasserted for the duration of the interrupt routine. When the interrupt routine terminates, NBPU is asserted prior to the next BPU Read/Write cycle.

Example: EXTBLT

The following example demonstrates how the DP8510/DP8511 may be used to do a bit-aligned block transfer. It assumes that the DP8510/DP8511 has been set up to do an OR operation, to shift one bit, and to not perform masking on a block of data two words wide and two lines high.

Operands	Operand Values Before				Operand Values After			
R0 - Source Address	A000				A008			
R1 - Destination Address	B000				B280			
R2 - Width	4				4			
R3 - Height	2				0			
R4 - Horizontal Increment	2				2			
R5 - Current Width	XXXX				0			
R6 - Source Warp	2				2			
R7 - Destination Warp	13C				13C			
A000 - A003	12	34	56	FF	12	34	56	FF
A004 - A007	78	9A	BC	FF	78	9A	BC	FF
B000 - B003	00	00	00	00	24	68	AC	FE
B140 - B143	00	00	00	00	F1	34	79	FF

Application: The DP8510/DP8511 BPU and the EXTBLT instruction should be used when very high-speed BITBLT operations are required.

The DP8510/DP8511 is a high-performance CMOS device designed for use in raster graphics applications. It implements, with high-speed pipelined logic, the data operations which are fundamental to BITBLT graphics: shifting, masking, and bit-wise logical manipulation of memory.

The 16 classical BITBLT functions supported by the BPU are shown in Table 2-1. In a single 16-bit memory write operation, the NS32CG16 can set the BPU's Function, Shift Amount, and both the first and second masks. External hardware must control the BPU's mask enable signals. Circuit examples are available regarding NS32CG16 and DP8510/11.

TABLE 2-1. DP8510/DP8511 BPU FUNCTIONS

FUNCTION	EXPLANATION
0	Zero source (Zero destination)
-s AND -d	Inverted source ANDed with inverted destination
-s AND d	Inverted source ANDed with destination
-s	Inverted source TO destination
s AND -d	Source ANDed with inverted destination
-d	Inverted destination TO destination
s XOR d	Source XORed with destination
-s OR -d	Inverted source ORed with inverted destination
s AND d	Source ANDed with destination
s XNOR d	Source XNORed with destination
d	Destination to destination
-s OR d	Inverted source ORed with destination
s	Source TO destination
s OR -d	Source ORed with inverted destination
s OR d	Source ORed with destination
1	0xFFFF source (set destination to all ones)

NOTE: 1. The width in bytes field-stored in R2 must be an even value and must be a multiple of the Horizontal Increment field stored in R4. Recommended values for the Horizontal Increment are +2 and -2. The instruction accepts other values, but they may cause undesirable results. When the Horizontal Increment is negative, the width field must also be negative since the Horizontal Increment is always subtracted from the current width in R5. R2 should contain following: (width in words of destination data * horizontal increment [R4]).

2. It's not necessary to initialize the current width stored in R5. The value is initialized prior to each iteration of the BITBLT operation.

3. The EXTBLT instruction should be preceded by CPMQB 1,2 if preloading is required, or by CMPQB 2,1 if preloading is not required.
4. Only word-aligned or even addresses should be used in conjunction with EXTBLT, as the DP8510/11 does not support byte activities on its 16-bit data bus.

Syntax:

MOVMPi					
0 0 0 0 0	0 0 0 0 *short*	0	0 1 1 1 opcode	1 1 i	0 0 0 0 1 1 1 0
23	16	15		8	7 0

HD-08-0-U

Operands of the MOVMPi instruction are specified in the following general-purpose registers:

R0 — Base Address of the Destination Data (32 bits)

R1 — Pointer Increment in bytes (32 bits, signed)

R2 — Number of times pattern is to be moved (32 bits, unsigned)

R3 — Source pattern (may be 8, 16 or 32 bits)

Description: MOVMPi first checks if the contents of R2 are zero. If they are not, the contents of R3 corresponding to size i are copied to the destination address. The i is the integer length suffix and may be B, W, or D, corresponding to Byte, Word and Double-word respectively. The number in R1 is then added to the destination address, and the value in R2 is decremented by one. The instruction repeats this process until R2 equals zero.

Note that while the Pointer Increment stored in R1 is signed, the number of times the pattern is to be moved stored in R2 is unsigned. At the completion of this instruction, the contents of R2 are always zero.

If the output data does not overlap, that is, R1 >= i, then the number of bytes written is n * R2, where n is 1 for i = B, 2 for i = W, and 4 for i = double-word.

MOVMPi **Move Multiple Pattern** (Cont)

Flags Affected: None

Traps: None; however, the instruction is interruptible.

Example: MOVMPW

This example shows how the low-order word of R3 moves to the destination addresses specified by the pointer and increment.

Operands	Operand Values Before	Operand Values After
R0 - Destination Address	A000	A014
R1 - Pointer Increment	2	2
R2 - # Copies of Pattern	A(10)	0 (0)
R3 - Pattern to be Copied 3	AAAA	AAAA
A000 - A001	31 62	AA AA
A002 - A003	A3 12	AA AA
A004 - A005	26 44	AA AA
A006 - A007	41 52	AA AA
A008 - A009	45 F2	AA AA
A00A - A00B	AF CD	AA AA
A00C - A00D	FF 44	AA AA
A00E - A00F	CC 41	AA AA
A010 - A011	52 45	AA AA
A012 - A013	52 66	AA AA
A014 - A015	12 00	12 00

Application: This instruction is designed to clear large blocks of RAM very quickly. It is helpful in printer applications when a page image in RAM must be cleared prior to drawing the next page. It is also useful for drawing patterns and horizontal lines.

Syntax:

SBITS					
0 0 0 0 0	0 0 0 0 *short*	0	1 1 0 1 opcode	1 1 i	0 0 0 0 1 1 1 0
23	16	15		8 7	0

HD-09-0-U

Operands of the SBITS instruction are specified in the following general purpose registers:

R0 — Base Address of the destination (32 bits)

R1 — Starting Bit Offset (32 bits, signed)

R2 — Run Length (32 bits, unsigned)

R3 — Address of string look-up table (32 bits)

Description: SBITS sets the number of bits specified in R2, starting at the bit location determined by adding the Bit Offset in R1 to the Base Address in R0.

Bit Address = Offset MOD 8

within the memory byte whose address is

Base Address + (Offset DIV 8)

where Offset is interpreted as a signed integer.

The Run Length is the number of bits to be set to form the string. Its maximum value, stored in R2, is 25 bits. If R2 is greater than 25, the double-word at (Base Address + (Offset DIV 8)) will be read, but no bits will be set at that address. Instead, the F bit of the PSR will be set. This allows external software to use a more efficient algorithm to set longer strings of bits.

This instruction executes extremely fast by indexing an external table which contains all combinations of offsets from 0 to 7 bits and run lengths from 0 to 25 bits. See Figure 2-1.

Offset MOD 8 and the Run Length determine which double-word from the table is ORed with the calculated destination double-word. The exact formula used to calculate the table address is:

R3 + 4 bytes/double-word * (R2 + 32 double-words/offset * (R1 MOD 8))

While the instruction does not require the string look-up table to be aligned on a word or double-word boundary, SBITS will achieve maximum speed if the table is aligned on a double-word boundary.

Flags Affected: F Set to 1 if the run length is 26 or greater.
Set to zero if the run length is 25 or smaller.

No other flags affected.

Traps: None

Example: SBITS

This example shows how a string of bits may be set in memory.

Operands	Operand Values Before				After			
R0 - Destination Address	A000				A00A			
R1 - Starting Bit Offset	8				8			
R2 - # Bits to be Set	C	(12)			C	(12)		
R3 - Address of Table	$stab				$stab			
	LSB			MSB	LSB			MSB
A000 - A003	31	62	A3	00	31	FF	AF	00

Application: The SBITS instruction makes it simple to draw horizontal lines and vertical lines which are several bits wide. Since SBITS is passed the run length, it can also be used in pattern expansion. SBITS makes use of an external table and an OR operation to write its result, and thus can be used as a bit-aligned pattern generator.

SBITS can be used to fill a character or graphics object very quickly, and it can also be used to expand fonts or data stored using the TBITS instruction.

NOTE: The $stab label stored in R3 refers to the address of the string look-up table in Figure 2-1. Any programs that contain the SBITS instruction must be able to access this table, or the instruction will not execute properly.

The string look-up table is comprised of 8 groups of 32 double-words, occupying 1024 bytes in all. It may reside in code space, being PC-relative, or in data space, being SB-relative.

For run lengths greater than 25 bits, the BIGSET algorithm which appears in Figure 2-2 may be used. It is written in System V assembly code. SBITS sets the F flag when the run length exceeds 25 bits, and thus the BFS instruction, Branch if Flag Set, may be used to jump to the BIGSET routine.

```
stab:   .double   0x00000000,0x00000001,0x00000003,0x00000007
        .double   0x0000000f,0x0000001f,0x0000003f,0x0000007f
        .double   0x000000ff,0x000001ff,0x000003ff,0x000007ff
        .double   0x00000fff,0x00001fff,0x00003fff,0x00007fff
        .double   0x0000ffff,0x0001ffff,0x0003ffff,0x0007ffff
        .double   0x000fffff,0x001fffff,0x003fffff,0x007fffff
        .double   0x00ffffff,0x01ffffff,0x03ffffff,0x07ffffff
        .double   0x0fffffff,0x1fffffff,0x3fffffff,0x7fffffff
        .double   0x00000000,0x00000002,0x00000006,0x0000000e
        .double   0x0000001e,0x0000003e,0x0000007e,0x000000fe
        .double   0x000001fe,0x000003fe,0x000007fe,0x00000ffe
        .double   0x00001ffe,0x00003ffe,0x00007ffe,0x0000fffe
        .double   0x0001fffe,0x0003fffe,0x0007fffe,0x000ffffe
        .double   0x001ffffe,0x003ffffe,0x007ffffe,0x00fffffe
        .double   0x01fffffe,0x03fffffe,0x07fffffe,0x0ffffffe
        .double   0x1ffffffe,0x3ffffffe,0x7ffffffe,0xfffffffe
        .double   0x00000000,0x00000004,0x0000000c,0x0000001c
        .double   0x0000003c,0x0000007c,0x000000fc,0x000001fc
        .double   0x000003fc,0x000007fc,0x00000ffc,0x00001ffc
        .double   0x00003ffc,0x00007ffc,0x0000fffc,0x0001fffc
        .double   0x0003fffc,0x0007fffc,0x000ffffc,0x001ffffc
        .double   0x003ffffc,0x007ffffc,0x00fffffc,0x01fffffc
        .double   0x03fffffc,0x07fffffc,0x0ffffffc,0x1ffffffc
        .double   0x3ffffffc,0x7ffffffc,0xfffffffc,0xfffffffc
        .double   0x00000000,0x00000008,0x00000018,0x00000038
        .double   0x00000078,0x000000f8,0x000001f8,0x000003f8
        .double   0x000007f8,0x00000ff8,0x00001ff8,0x00003ff8
        .double   0x00007ff8,0x0000fff8,0x0001fff8,0x0003fff8
        .double   0x0007fff8,0x000ffff8,0x001ffff8,0x003ffff8
        .double   0x007ffff8,0x00fffff8,0x01fffff8,0x03fffff8
        .double   0x07fffff8,0x0ffffff8,0x1ffffff8,0x3ffffff8
        .double   0x7ffffff8,0xfffffff8,0xfffffff8,0xfffffff8
        .double   0x00000000,0x00000010,0x00000030,0x00000070
```

Figure 2-1. SBIT TABLE
Sheet 1 of 2

```
.double  0x000000f0,0x000001f0,0x000003f0,0x000007f0
.double  0x00000ff0,0x00001ff0,0x00003ff0,0x00007ff0
.double  0x0000fff0,0x0001fff0,0x0003fff0,0x0007fff0
.double  0x000ffff0,0x001ffff0,0x003ffff0,0x007ffff0
.double  0x00fffff0,0x01fffff0,0x03fffff0,0x07fffff0
.double  0x0ffffff0,0x1ffffff0,0x3ffffff0,0x7ffffff0
.double  0xfffffff0,0xfffffff0,0xfffffff0,0xfffffff0
.double  0x00000000,0x00000020,0x00000060,0x000000e0
.double  0x000001e0,0x000003e0,0x000007e0,0x00000fe0
.double  0x00001fe0,0x00003fe0,0x00007fe0,0x0000ffe0
.double  0x0001ffe0,0x0003ffe0,0x0007ffe0,0x000fffe0
.double  0x001fffe0,0x003fffe0,0x007fffe0,0x00ffffe0
.double  0x01ffffe0,0x03ffffe0,0x07ffffe0,0x0fffffe0
.double  0x1fffffe0,0x3fffffe0,0x7fffffe0,0xffffffe0
.double  0xffffffe0,0xffffffe0,0xffffffe0,0xffffffe0
.double  0x00000000,0x00000040,0x000000c0,0x000001c0
.double  0x000003c0,0x000007c0,0x00000fc0,0x00001fc0
.double  0x00003fc0,0x00007fc0,0x0000ffc0,0x0001ffc0
.double  0x0003ffc0,0x0007ffc0,0x000fffc0,0x001fffc0
.double  0x003fffc0,0x007fffc0,0x00ffffc0,0x01ffffc0
.double  0x03ffffc0,0x07ffffc0,0x0fffffc0,0x1fffffc0
.double  0x3fffffc0,0x7fffffc0,0xffffffc0,0xffffffc0
.double  0xffffffc0,0xffffffc0,0xffffffc0,0xffffffc0
.double  0x00000000,0x00000080,0x00000180,0x00000380
.double  0x00000780,0x00000f80,0x00001f80,0x00003f80
.double  0x00007f80,0x0000ff80,0x0001ff80,0x0003ff80
.double  0x0007ff80,0x000fff80,0x001fff80,0x003fff80
.double  0x007fff80,0x00ffff80,0x01ffff80,0x03ffff80
.double  0x07ffff80,0x0fffff80,0x1fffff80,0x3fffff80
.double  0x7fffff80,0xffffff80,0xffffff80,0xffffff80
.double  0xffffff80,0xffffff80,0xffffff80,0xffffffff
```

Figure 2-1 SBIT Table

Sheet 2 of 2

```
         .align 4
bigset:  save      [r0,r1,r2,r3,r4,r5,r6]  #save registers we will affect
         cmpd      $100,r2                 #are we going to set lots?
         blt       bigs1:b                 #yes, use regular algorithm
         movd      r2,r4                   #copy current length
         addr      25,r2                   #always set 25 bits
         cmpd      r2,r4                   #will we set > 25 bits?
         bge       alp2:b                  #no, exit now
alp1:    sbits     #set the bits
         addd      r2,r1                   #move the bit pointer
         subd      r2,r4                   #subtract to find # left
         cmpd      r2,r4                   #are there bits left?
         blt       alp1:b                  #yes, set them
alp2:    movd      r4,r2                   #move last bits to set
         sbits                             #set the last bits
         restore   [r0,r1,r2,r3,r4,r5,r6]  #restore saved registers
         ret       $0                      #and exit
         .align 4
bigs1:   movd      r1,r4                   #get current bit offset
         ashd      $-3,r4                  #divide by eight to get byte offset
         addd      r4,r0                   #add in base. r0 is new base pointer
         andd      $7,r1                   #mask off msb's of bit pointer to
                                           #get bit = bit offset mod 8
#Now we have true base address and bit offset within base. Now we will move
#to double-word alignment. This speeds up the MOVMPD for long bit sequences.
         movqd     3,r4                    #place mask in r5
         andd      r0,r4                   #get low three bits of address
         xorb      $3,r4                   #and get bytes left to alignment
         addqd     1,r4                    #rem += 1 (for the byte we are on)
         ashd      $3,r4                   #rem *= 8 to get bits to alignment
         subd      r1,r4                   #subtract current bit offset
         cmpd      r4,r2                   #is this more than number of bits left
         bge       shrt:b                  #it is, do it the short way
         cmpd      $32,r4                  #if we are already double aligned, go
                                           #do the MOVMPD
         beq       mvm:b
         movd      r1,r5                   #calculate index into table
         lshd      $5,r5                   #index = 32 * bit offset
         addd      r4,r5                   #index += run length
         ord       r3[r5:d],0(r0)          #or in required bits
         bicb      $3,r0                   #clear last two bits, and
         addqd     4,r0                    #bump to next double
         subd      r4,r2                   #zap sp'd bits off
```

Figure 2-2. BIGSET Algorithm
Sheet 1 of 2

```
mvm:    movd      r2,r4                    #save run length for a minute
        movd      r3,r5                    #and save pointer to table
        ashd      $-5,r2                   #r1 = r1 / 32 = number of doubles
        movd      1020(r3) r3              #get source pattern from table
        movqd     4,r1                     #increment is r1
        movmpd                             #move multiple pattern
        andd      $0x1f,r4                 #mask off all but last 32 bits
        ord       r5[r4:d],0(r0)           #insert the last few bits
        restore   [r0,r1,r2,r3,r4,r5,r6]   #restore saved registers
        ret       $0

        .align 4
shrt:   cmpb      $32,r2                   #check to see if it is exactly
        beq       shrt1:b                  #32 bits. If it is, branch.
        movd      r1,r4                    #calculate index into table
        lshd      $5,r4                    #index = 32 * bit offset
        addd      r2,r4                    #index += run length
        ord       r3[r4:d],0(r0)           #or in required bits
        restore   [r0,r1,r2,r3,r4,r5,r6]   #restore saved registers
        ret       $0
        .align 4
shrt1:  movd      1020(r3),0(r0)           #copy last entry of table
        restore   [r0,r1,r2,r3,r4,r5,r6]   #(all 32 bits) and restore
        ret                                $0
```

Figure 2-2 BIGSET Algorithm
Sheet 2 of 2

SBITPS Set Bit Perpendicular String

Syntax:

SBITPS					
0 0 0 0 0	0000 *short*	0	1 011 opcode	1 1 i	0 0 0 0 1 1 1 0
23	16	15		8	7 0

HD-10-0-U

Operands of the SBITPS instruction are specified in the following general-purpose registers:

R0 — Base Address of the Destination (32 bits)

R1 — Starting Bit Offset (32 bits, signed)

R2 — Run Length (32 bits, unsigned)

R3 — Destination Warp (32 bits, signed)

Description: SBITPS can be used to draw horizontal, vertical and 45 degree lines. SPITPS sets a group of bits in memory, separated by a number of bits known as the Destination Warp. SBITPS implements the following algorithm:

1. If the run length in R2 (the number of bits to be set) is zero, SBITS terminates.

2. Otherwise, a bit is set at the bit location determined by adding the Offset in R1 to the Base Address in R0.

   Bit address = Offset MOD 8

   within the memory byte whose address is

   Base Address + (Offset DIV 8)

   where Offset is interpreted as a signed integer.

3. Offset = Offset + Destination Warp

4. Run Length = Run Length - 1
5. Go to step 1

Horizontal lines result if the Destination Warp is +1 or -1. Vertical lines result if the Destination Warp is equal to the page warp, which is the distance in bits from the beginning of one scan line to the beginning of the next. Forty-five degree lines result if the Destination Warp is equal to (page warp + 1) or to (page warp - 1).

Flags Affected: None

Traps: None; however, the instruction is interruptible.

Example: SBITPS

This example demonstrates how a vertical line may be drawn.

Operands	Operand Values Before	After
R0	A000	A000
R1	C	320C
R2	5	0
R3	A00	A00
A000	00 00	00 10
A140	00 00	00 10
A280	00 00	00 10
A3C0	00 00	00 10
A500	00 00	00 10
A640	00 00	00 10

Application: The SBITPS instruction speeds the drawing of perpendicular (vertical) lines in the memory space. It has other applications as well, including drawing lines at 45 degrees, expansion/rotation of Run Length Limited fonts (encoded with the TBITS instruction), and filling objects.

Syntax:

Option S is zero for TBITS 0 and one for TBITS 1.

TBITS					
0 0 0 0 0	0 0 0 S *short*	0	1 0 0 1 opcode	1 1 i	0 0 0 0 1 1 1 0
23	16	15		8	7 … 0

HD-11-0-U

Operands of the TBITS instruction are specified in the following general purpose registers:

R0 — Base Address of the source (32-bits)

R1 — Starting Bit Offset (32-bits, signed)

R2 — Run Length (32-bits)

R3 — Maximum Run Length (32-bits)

R4 — Maximum Bit Offset (32-bits)

Description: TBITS counts a series of zeroes if the option chosen is zero. Similarly, it counts a series of ones if the option is one. Counting begins at the bit location determined by adding the offset in R1 to the Base Address in R0, as follows:

Bit address = Offset MOD 8

within the memory byte whose address is

Base Address + (Offset DIV 8)

where Offset is interpreted as a signed integer.

The following seven conditions cause TBITS to terminate:

1. S is zero, and a one is encountered before Run Length equals Maximum Run Length. R1 points to the first set bit upon termination.

2. S is one, and a zero is encountered before Run Length equals Maximum Run Length. R1 points to the first cleared bit upon termination.

3. S is zero, and Run Length equals Maximum Run Length before a one is encountered. R1 equals Maximum Run Length plus Base Address upon termination.

4. S is one, and Run Length equals Maximum Run Length before a zero is encountered. R1 equals Maximum Run Length plus Base Address upon termination.

5. S is zero, and Starting Bit Offset equals the Maximum Bit Offset before a one is encountered.

6. S is one, and Starting Bit Offset equals the Maximum Bit Offset before a zero is encountered.

7. Starting Bit Offset is equal or greater than the Maximum Bit Offset when the instruction begins the execution. Then, the F flag is set to the value of the option for which the instruction has been testing.

Flags Affected:

CONDITION	L	F	NOTES
R2 < R3, bit found	1	F	F reflects last bit tested.
R2 => R3, bit not found	1	F	F reflects last bit tested.
R1 => R4, bit not found	0	F	F reflects last bit tested.
R1 => R4 on entry	0	0/1	F = Value of S

Flags other than L and F are not affected.

Traps: None; however, this instruction is interruptible.

Example: TBITS 0

This example scans a horizontal line in memory to determine where the first set bit is.

Operands	Operand Values Before	After
R0	A000	A000
R1	1F	28
R2	0	9
R3	FF	FF
R4	140	140
A000	ED F8	ED F8
A002	AE 6F	AE 6F
A004	00 EF	00 EF
A006	85 C1	85 C1

Application: This instruction is intended for the compression of font data and other data which benefit from bit-run compression. Several different algorithms may be used in conjunction with TBITS, as it has a flexible maximum run length. Used with the *Series 32000* INS instruction, TBITS provides the basis for an efficient compression algorithm. EXTi and SBITS or SBITPS instructions may be used to decompress the compressed data from TBITS.

NOTE: The current run length stored in R2 should be cleared to zero by executing a MOVQD 0,R2 prior to invoking the TBITS instruction.

Chapter 3

NS32CG16 INSTRUCTION TIMING

Chapter 3

NS32CG16 INSTRUCTION TIMING

3.1 INTRODUCTION

This chapter shows the expected timing of the NS32CG16 graphics instructions. Refer to the *Series 32000 Databook* (1986, Pages 11-30) regarding instructions not listed here.

As this is advance information, it is subject to change without notice.

3.2 ASSUMPTIONS

The cycle time is one T-state of the *Series 32000*, equivalent to one-half of the input clock frequency present on the OSCIN pin of the NS32CG16. A 30-MHz clock source, therefore, yields a cycle time of 66.67 ns. Since the C and M bits of the NS32CG16's configuration register control an on-chip clock divider, setting these bits divides the clock by 1, 2, 4 or 8 to give a cycle time (with a 30-MHz clock source) of 66.67, 133.3 ns, 266.7 ns and 533.3 ns. This first section describes timing of the graphics instructions.

When needed, the entire instruction is assumed to be present in the instruction queue. Interference from instruction prefetches is ignored, with the exception of the BITWT instruction, where a no-wait-state prefetch is included (four clock cycles).

It is assumed that all memory operand transfers are completed before the next instruction begins execution. In the case of an operand of access class RMW in memory, this is pessimistic, as the write transfer occurs in parallel with the execution of the next instruction.

Where possible, the values of operands are taken into consideration when they affect instruction timing, and a range of times is given. Where this is not done, the average case is assumed. All memory accesses are assumed to be word aligned. Non-word-aligned data is acceptable but causes the execution time of a given instruction to increase.

The variety of definitions that follows allows accurate prediction of system performance when, for example, the source data may be in ROM, and the destination may be in RAM, each having a different number of wait states. The number of wait states refers to the number of additional clock cycles requested via the CWAIT or WAIT*n* pins of the NS32CG16, on a given byte or word-memory access.

3.2.1 Definitions

T_{waitrd}	The number of wait states applied for a Read operation.
T_{waitwr}	The number of wait states applied for a Write operation.
$T_{waitrds}$	The number of wait states applied for a Read operation on source data. This also refers to the number of wait states applied for a table memory access (in the SBITS instruction, for example).
$T_{waitrdd}$	The number of wait states applied for a Read operation on destination data.
$T_{waitwrd}$	The number of wait states applied for a Write operation on destination data.
$T_{waitblt}$	$T_{waitrds} + T_{waitrdd} * 2 + T_{waitwrd} * 2$, the value used for BITBLT timing.
width	The width of a BITBLT operation, in words.
height	The height of a BITBLT operation, in scan lines.
shift	The number of bits of shift applied.

The Average Execution Time table (see Table 3-1) lists the execution time for the NS32CG16 instructions based on an average shift of eight bits and a no-wait-state system design. The ''no option'' of each instruction is used.

TABLE 3-1. AVERAGE EXECUTION TIME

INSTRUCTION	NUMBER OF CYCLES
BBOR	42 + (107 + 44 * (*width* - 2)) * *height*
BBXOR	44 + (107 + 44 * (*width* - 2)) * *height*
BBAND	45 + (111 + 44 * (*width* - 2)) * *height*
BBFOR	48 + (61 + 25 * (*width* - 2)) * *height* for *shift*=0 48 + (74 + 32 * (*width* - 2)) * *height* for *shift*=1-8
BBSTOD	66 + (170 + 60 * (*width* - 2)) * *height*
BITWT	if *shift* = 0, 16 if *shift* = 1 - 8, 28
EXTBLT	35 + (19 + 12 * *width*) * *height* (pre-read) 35 + (13 + 12 * *width*) * *height* (no pre-read)
MOVMPB,W MOVMPD	16 + 7 * R2 16 + 8 * R2
TBIT	27 per bit tested
SBITS	if R2 < = 25 39 else 42
SBITP	8 + (34 * R2)
MOVSB	30 + (13 * R0) + (2 * R0 * 4)
MOVSW	30 + (13 * R0) + (2 * R0 * 4)
MOVSD	30 + (10 * R0) + (2 * R0 * 8)
	NOTE: MOVSD is about 12% faster than NS32016

3.2.2 Interpreting the Table

To calculate the execution time for a given instruction, complete the formula and evaluate. For example, to calculate the time for a 10-word wide, 50-line high BBOR operation, the completed formula is:

$$42 + (107 + 44 * (10 - 2)) * 50$$

or

$$42 + (107 + 352) * 50 = 22{,}992 \text{ clocks or } 1.53 \text{ msec at } 15 \text{ MHz, } 0 \text{ wait states.}$$

3.2.3 Calculating the Effects of Shift Values

The formulae in the table give the average execution time for the BITBLT instructions (BBOR, BBXOR, BBAND, BBFOR, BITWT) with a *shift* of 0 to 8 bits. The BITWT and BBFOR instructions, however, optimize a *shift* of 0 bits by reading and writing only a word of the destination data (16 bits). For *shift*s greater than 0, they still must read and write a double-word of data (32 bits). Note that the EXTBLT instruction is not affected by the *shift* amount.

Shifts of greater than 8 bits add 1 clock per bit of *shift* over 8, per word of data read. For example, the BBOR with a *shift* of 15 bits yields the following formula:

$$42 + (107 + 44 * (\mathit{width} - 2)) * \mathit{height} + ((\mathit{shift} - 8) * \mathit{width} * \mathit{height})$$

Inserting the previous example of the 10 by 50 BBOR:

$$45 + (107 + 44 * (\mathbf{10} - 2)) * \mathbf{50} + ((\mathbf{15} - 8) * \mathbf{10} * \mathbf{50})$$

or

$$42 + (107 + 352) * 50 + (7 * 500) = 26{,}492 \text{ clocks or } 1.77 \text{ msec @ } 15 \text{ MHz}$$

This represents the ''worst case'' time for this instruction, since a *shift* of greater than 15 bits can be handled by moving the source and destination pointers by 2 bytes and adjusting the *shift* amount.

The ''best case'' and ''average case'' times for most instructions are the same, due to reading the destination data during the *shift*ing of the source data. This parallel operation is a feature of *Series 32000*.

The ''best case'' for the BITWT and BBFOR instructions is a *shift* of zero bits. This is due to further internal optimization of these instructions, realizing that only a word of the destination data needs to be operated on if a *shift* of zero is specified.

Table 3-2 shows the expected timing information for *shift*s greater than 8 bits.

TABLE 3-2. SHIFT > 8

INSTRUCTION	NUMBER OF CYCLES
BBOR	42 + (107 + 44 * (*width* - 2)) * *height* + ((*shift* - 8) * *width* * *height*)
BBXOR	44 + (107 + 44 * (*width* - 2)) * *height* + ((*shift* - 8) * *width* * *height*)
BBAND	45 + (111 + 44 * (*width* - 2)) * *height* + ((*shift* - 8) * *width* * *height*)
BBFOR	48 + (74 + 32 * (*width* - 2)) * *height* + ((*shift* - 8) * *width* * *height*)
BBSTOD	66 + (170 + 60 * (*width* - 2)) * *height* + ((*shift* - 8) * *width* * *height*)
BITWT	28 + (*shift* - 8)
EXTBLT	35 + (17 + 13 * *width*) * *height* (pre-read)
	35 + (11 + 13 * *width*) * *height* (no pre-read)

3.2.4 Calculating the Effects of Wait States

Since the new NS32CG16 instructions make use of the pipelined reads and writes of the *Series 32000*, calculation of the effect of wait states is rather difficult. As an example, in the MOVSi instruction group, each wait state on read operations adds 1 clock cycle per read bus access. Each wait state on write operations subtracts 1 clock cycle per write bus access, from the TCY of the instruction, since updating the pointers occurs in parallel with the write operation. This means that wait states can be added to write cycles without changing the execution time of the instruction, up to a maximum of 13 wait states on writes for MOVSB and MOVSW, and 4 wait states on writes for MOVSD. At zero wait states, a MOVSD of 1,056,000 bytes (264,000 double-words) executes in

$$30 + (10 * 264{,}000) + (2 * 264{,}000 * 8)$$

or

$$30 + 2{,}640{,}000 + 4{,}224{,}000 = 6{,}864{,}030 \text{ clocks or } 458 \text{ msec @ } 15 \text{ MHz}$$

With two wait states on read and write (T_{waitrd} = 2 and T_{waitwr} = 2), see Table 3-3, the time becomes

30 + (10 * **264,000**) + (2 * **264,000** * 8) + 0 + (2 * 2) * **264,000**

or

30 + 2,640,000 + 4,224,000 + 1,056,000 = 7,920,030 clocks or 528 msec @ 15 MHz

Instructions that have *shift* amounts, such as BBOR, BBXOR, BBAND, BBFOR and BITWT, make use of the parallel nature of the *Series 32000* by doing the actual *shift* during the reading of the double-word destination data. This means that if there are wait states on read operations, these instructions are able to *shift* further, without impacting the overall time. For example, the total execution time for a BBFOR operation, *shift*ing 8 bits, with 2 wait states on read operations, is the same execution time as for a BBFOR operation *shift*ing by 12 bits. This is because a destination read takes 4 clock cycles longer than a no-wait-state double-word read does. Note that this effect is not valid for more than 4 wait states because at 4 wait states, all possible *shift* values (0 - 15) are ''hidden'' during the destination read.

Note that in Table 3-3, $T_{waitrds}$ refers to the reading of the Source data, or of the table data used for a particular instruction. $T_{waitrdd}$ refers to the reading of the Destination data, or of the data on which to be operated. Table 3-3 shows the expected timing of instructions with wait states. Again, this is the average execution time, using a *shift* of eight (where appropriate).

$T_{waitblt}$ is equal to ($T_{waitrds}$ + 2 * $T_{waitrdd}$ + 2 * $T_{waitwrd}$). This represents one BITBLT transfer, which makes the following equations easier.

TABLE 3-3. AVERAGE EXECUTION WITH WAIT STATES

INSTRUCTION	NUMBER OF CYCLES
BBOR	$42 + ((107 + 2 * T_{waitblt}) + (44 + T_{waitblt}) * (\mathit{width} - 2)) * \mathit{height}$
BBXOR	$44 + ((107 + 2 * T_{waitblt}) + (44 + T_{waitblt}) * (\mathit{width} - 2)) * \mathit{height}$
BBAND	$45 + ((111 + 2 * T_{waitblt}) + (44 + T_{waitblt}) * (\mathit{width} - 2)) * \mathit{height}$
BBFOR	$48 + ((74 + 2 * T_{waitblt}) + (32 + T_{waitblt}) * (\mathit{width} - 2)) * \mathit{height}$
BBSTOD	$66 + ((170 + 2 * T_{waitblt}) + (60 + \mathrm{Twatiblt}) * (\mathit{width} - 2)) * \mathit{height}$
BITWT	$\mathit{shift} = 0,\ 16 + T_{waitrds} + T_{waitrdd} + T_{waitwrd}$ $\mathit{shift} = 1 - 8,\ 28 + T_{waitblt}$
EXTBLT	$35 + (19 + (12 + (T_{waitrds} + T_{waitrdd} + T_{waitwrd}))* \mathit{width})* \mathit{height}$ (pre-read) $35 + (13 + (12 + (T_{waitrds} + T_{waitrdd} + T_{waitwrd}))* \mathit{width})* \mathit{height}$ (no pre-read)
MOVMPB,W	if $(T_{waitwr} > 1)$ $16 + 7 * R2 + (T_{waitwr} - 1) * R2$ else $16 + 7 * R2 + 0$
MOVMPD	$16 + 8 * R2 + T_{waitwr} * R2$
TBIT	$(27 + T_{waitrd})$ per bit tested
SBITS	if $R2 <= 25$ $39 + (2 * T_{waitrdd} + 2 * T_{waitwrd} + 2 * T_{waitrds})$ else $42 + (2 * T_{waitrdd} + 2 * T_{waitrds})$
SBITP	$8 + (34 * R2) + ((T_{waitrdd} + T_{waitwrd}) * R2)$
MOVSB	if $T_{waitwr} > 13$ $30 + (13 * R0) + ((T_{waitwr} - 13) + T_{waitrd}) * R0$ else $30 + (13 * R0) + 0 + (T_{waitrd} * R0)$
MOVSW	if $T_{waitwr} > 13$ $30 + (13 * R0) + ((T_{waitwr} - 13) + T_{waitrd}) * R0$ else $30 + (13 * R0) + 0 + (T_{waitrd} * R0)$
MOVSD	if $T_{waitwr} > 4$ $30 + (10 * R0) + (2 * R0 * 8) + (((T_{waitwr} - 4) * 2) + (T_{waitrd} * 2)) * R0$ else $30 + (10 * R0) + (2 * R0 * 8) + 0 + ((T_{waitrd} * 2) * R0)$

3.3 NS32CG16 GENERAL INSTRUCTION TIMING

3.3.1 Assumptions

The entire instruction, with all displacements and immediate operands, is assumed to be present in the instruction queue when needed.

Interference from instruction prefetches, which is very dependent upon the preceding instruction(s), is ignored. This assumption tends to affect the timing estimate in an optimistic direction.

It is assumed that all memory operand transfers are completed before the next instruction begins execution. In the case of an operand of access class rmw in memory, this is pessimistic, as the Write transfer occurs in parallel with the execution of the next instruction.

It is assumed that there is no overlap between the fetch of an operand and the following sequences of microcode. This is pessimistic, as the fetch of Operand A generally occurs in parallel with the effective adddress calculation of Operand B, and the fetch of Operand B occurs in parallel with the execution phase of the instruction.

Where possible, the values of operands are taken into consideration when they affect instruction timing, and a range of times is given. Where this is not done, the worst case is assumed.

3.3.2 Definitions

TEA — The time required to calculate an operand's effective address. For a Register or Immediate operand, this includes the fetch of that operand.

TOPB — The time needed to read or write a memory byte.

TOPW — The time needed to read or write a memory word.

TOPD — The time needed to read or write a memory double-word.

TOPi — The time needed to read or write a memory operand, where the operand size is given by the operation length of the instruction. It is always equivalent to either TOPB, TOPW or TOPD.

TCY — Internal processing overhead, in clock cycles.

L — Internal processing whose duration depends on the operation length. The number of clock cycles is derived by multiplying this value by the number of bytes in the operation length.

3.3.3 Equations

TOPB	If operand is in a register or is immediate then TOP= 0 else TOP = 3 + TMMU
TOPW	If operand is in a register or is immediate then TOPW = 0 else if word-aligned (even address) then TOPW = 3 else TOPW = 7
TOPD	If operand is in a register or is immediate then TOPD = 0 else if word-aligned (even address) then TOPD = 7 else TOPD = 11
TOPi	If operand is in a register or is immediate then TOPi = 0 else if i = byte then TOPi = TOPB else if i = word then TOPi = TOPW else (i = double-word) then TOPi = TOPD
TCY	TCY = 1
L	If i (operation length) = byte then L = 1 else if i = word then L = 2 else (i = double-word) L = 4
TEA	If REGISTER addressing then TEA = 2 If IMMEDIATE or ABSOLUTE addressing then TEA = 4 If REGISTER RELATIVE or MEMORY SPACE addressing then TEA = 5 If MEMORY RELATIVE addressing then TEA = 7 + TOPD If TOP OF STACK addressing then if access class = write then TEA = 4 if access class = read then TEA = 2 else TEA = 3 if EXTERNAL addressing then TEA = 11 + 2 * TOPD if SCALED INDEXED addressing then TEA = TI1 + TI2

where TI1 depends on scale factor:

if byte indexing TI1 = 5
if word indexing TI1 = 7
if double-word indexing TI1 = 8
if quad-word indexing TI1 = 10

and TI2 = TEA of the basemode except:

if basemode is REGISTER then TI2 = 5
if basemode is TOP OF STACK then TI2 = 4

3.3.4 Calculation of Total Execution Time (TEX)

TEX is obtained by performing the following steps:

1. Find the desired instruction in the table.
2. Calculate the values of TEA, TOPB, etc., using the numbers in the table and the equations given on the preceding page.
3. The result derived by adding together these values is the execution time (TEX) in clock cycles.

3.3.5 Notes on Table Use

Values in the TEA column (see Tables 3-4 and 3-5) indicate the number of effective addresses to be calculated. If the value in this column is less than the number of general operands in the instruction, this is because one or both operands are in registers and that instruction has an optimized form which eliminates TEA for such operands.

In the L column, multiply the entry by the operation length in bytes (1, 2 or 4).

In the TCY column, special notations sometimes appear:

n1-n2 means n1 minimum, n2 maximum.

n1%n2 means that the instruction flushes the instruction queue after n1 clock cycles and nonsequentially fetches the next instruction. The value n2, indicating the total number of clock cycles in internally executing the instruction (including n1), is not generally useful. The most accurate technique for determining such timing depends on the size and alignment of the basic instruction portion of the next instrution, plus index bytes. If this portion can be read in one memory cycle, then the execution time is n1 + 10 (including the memory cycle). If more memory cycles are required, the value is n1 + 5 + 4*m, where m is the number of memory cycles required.

In the Notes column, notations held within angle brackets <> indicate alternative in the form of the instruction which affect the execution time. A table entry which is affected by the form of the instruction may have multiple values, separated by slashes, corresponding to the alternatives. The notations are:

<M>	Memory form
<R>	Register form
<MM>	Memory-to-Memory form
<RM>	Register-to-Memory form
<MR>	Memory-to-Register form
<RR>	Register-to-Register form
x	either Register to Memory

3.3.6 Example of Table Usage

Calculate TEX for the instruction: CMPW R0, TOS

Operand A is in a register: Operand B is in memory. The table values must be used corresponding to the <xM> case as given in the Notes column (<xM> meaning ''anything to memory'').

Only the TEA, TOPi and TCY columns have values assigned for the CMPi instruction; therefore, they are the only ones that need to be calculated to find TEX. The blank columns are irrelevant to this instruction.

The TEA column contains 2 for the <xM> case. This means that effective address times have to be calculated for both operands. (For the <MR> case, the Register operand requires no TEA time; therefore, only the Memory operand TEA is necessary.) From the equations:

TEA (Register mode) = 2.
TEA (Top-of-Stack mode, read access class) = 2.

Total TEA = 2 + 2 = 4.

The TOPi column represents potential operand transfers to or from memory. For a Compare instruction, each operand is read once, for a total of two operand transfers.

TOPi (Word, Register) = 0.

TOPi (Word, TOS) = TOPW = 3 (assuming aligned, no waits)

Total TOPi = 3

TCY is the time required for internal operation within the CPU. The TCY value for this case is 3.

TEX = TEA + TOPi + TCY = 4 + 3 + 3 = 10 machine cycles.

If the CPU is running at 15 MHz, then a machine cycle (clock cycle) is 66 ns. Therefore, this instruction takes 10 x 66 ns, or 660 ns to execute.

TABLE 3-4. INSTRUCTION TIMING PARAMETERS
Sheet 1 of 3

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
ABSi	2	-	-	-	2	9/8	-	src <0/src> = 0
ACBi	1	-	-	-	2	16/15%20	-	<M>, no branch/branch
ACBi	-	-	-	-	-	18/17%22	-	<R>, no branch/branch
ADDi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
ADDCi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
ADDPi	2	-	-	-	3	16/18	-	no carry/carry
ADDQi	1/0	-	-	-	2/0	6/4	-	<M>/<R>
ADDR	2/1	-	-	1/0	-	2/3	-	<xM>/<xR>
ADJSPi	1	-	-	-	1	6	-	<xM>/<MR>/<RR>
ANDi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
ASHi	2	1	-	-	2	14-45	-	
Bcond	-	-	-	-	-	7/6%10	-	no branch/branch
BICi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
BICPSRB	1	1	-	-	-	18%22	-	
BICPSRW	1	-	1	-	-	30%34	-	
BISPSRB	1	1	-	-	-	18%22	-	
BISPSRW	1	-	1	-	-	30%34	-	
BPT	-	-	4	3	-	40	-	
BR	1	-	-	-	1	4%0	-	
BSR	-	-	-	1	-	6%16	-	
CASEi	1	-	-	-	1	4%9	-	
CBITi	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
CBITii	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
CHECKi	2	-	-	-	3	7/10/11	-	high/low/ok
CMPi	2/1/0	-	-	-	2/1/0	3	-	<xM>/<MR>/<RR>
CMPMi	2	-	-	-	2*n	9*n + 24	-	n = # of delements in block
CMPQi	1/0	-	-	-	1/0	3	-	<M>/<R>
CMPSi	-	-	-	-	2*n	35*n+53	-	n = # of elements, not Translated
CMPST	-	n	-	-	2*n	38*n+53	-	Translated
COMi	2	-	-	-	2	7	-	
CVTP	2	-	-	1	-	7	-	
CXP	-	-	3	4	-	16%21	-	
CXPD	1	-	3	3	-	13%18	-	
DELi	2/1	-	-	-	5/1	38/31	16	<xM>/<xR>
DIA	-	-	-	-	-	3%7	-	

TABLE 3-4 INSTRUCTION TIMING PARAMETERS
Sheet 2 of 3

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
DIVi	2	-	-	-	3	58-68	16	
ENTER	-	-	-	n + 1	-	4*n + 18	-	n = # of general registers saved
EXIT	-	-	-	n + 1	-	5*n + 17	-	n = # of general registers restored
EXTi	2	-	-	1	1	19-29	-	field in memory
EXTi	2	-	-	-	1	17-51	-	field in register
EXTSi	2	-	-	1	1	26-36	-	
FFSi	2	2	-	-	1	24-28	24	
FLAG	-	-	0/4	0/3	-	6/44	-	no trap/trap
IBITi	2/1	2/0	-	-	1	17/9	-	<xM>/<xR>
INDEXi	2	-	-	-	2	25	16	
INSi	2	-	-	2	1	29-39	-	field in memory
INSi	1	-	-	-	1	28-96	-	field in register
INSSi	2	-	-	2	1	39-49	-	
JSR	1	-	-	1	1	5%15	-	
JUMP	1	-	-	-	-	2%6	-	
LPRi	1	-	-	-	1	19-33	-	
LSHi	2	1	-	-	2	14-45	-	
MELi	2	-	-	-	4	23	16	
MODi	2	-	-	-	3	54-73	16	
MOVi	2/1/0	-	-	-	2/1/0	1/3/3	-	<xM>/<MR>/<RR>
MOVMi	2	-	-	-	2*n	3*n+20	-	n = # of elements in block
MOVQi	1/0	-	-	-	1/0	2/3	-	<M>/<R>
MOVSi	-	-	-	-	-	-	-	See graphics instructions
MOVSi	-	-	-	-	2*n	24*n+54	-	B, W and/or M option in effect
MOVST	-	n	-	-	2*n	27*n+54	-	Translated
MOVXBD	2	1	-	1	-	6	-	
MOVXBW	2	1	1	-	-	6	-	
MOVXWD	2	-	1	1	-	6	-	
MOVZBD	2	1	-	1	-	5	-	
MOVZBW	2	1	1	-	-	5	-	
MOVZWD	2	-	1	1	-	5	-	
MULi	2	-	-	-	3	15	16	
NEGi	2	-	-	-	2	5	16	
NOP	-	-	-	-	-	3	-	
NOTi	2	-	-	-	2	5	-	

TABLE 3-4 INSTRUCTION TIMING PARAMETERS

Sheet 3 of 3

MNEMONIC	TEA	TOPB	TOPW	TOPD	TOPi	TCY	L	NOTES
ORi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
QUOi	2	-	-	-	3	49-55	16	
REMi	2	-	-	-	3	57-62	16	
RESTORE	-	-	-	n	-	5*n+12	-	n=# of general registers restored
RET	-	-	-	1	-	2%8	-	
RETi	-	1	3	3	-	39%45	-	
RETT	-	-	2	2	-	35%41	-	
ROTi	2	1	-	-	2	14-45	-	
RXP	-	-	1	2	-	2%6	-	
Scondi	1	-	-	-	-	1	9/10	False/True
SAVE	-	-	-	n	-	4*n+13	-	n=# of general registered saved
SBITi	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
SBITii	2/1	2/0	-	-	1	15/7	-	<xM>/<xR>
SETCFG	-	-	-	-	-	15	-	
SKPSi	-	-	-	-	n	27*n+51	-	n=# of elements, not translated
SKPST	-	n	-	-	n	30*n+51	-	Translated
SPRi	1	-	-	-	1	21-27	-	
SUBi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>
SUBCi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<R>/<RR>
SUBPi	2	-	-	-	3	16/18	-	no carry/carry
SVC	-	-	4	3	-	40	-	
TBITi	2/1	1/0	-	-	1	14/4	-	<xM>/<xR>
WAIT	-	-	-	-	-	6-?	-	? - until an interrupt/reset
XORi	2/1/0	-	-	-	3/1/0	3/4/4	-	<xM>/<MR>/<RR>

3.3.7 NS32081 Floating-Point Execution Times

The following section gives execution timing information for Floating-Point instructions. Some additional timing definitions are used, as given below:

f The floating-point operation length.

Standard Floating (32 bits): f = 1
Long Floating (64 bits): f = 2

Tf The time required to transfer 32 bits of a floating-point value to or from the NS32081 Floating-Point unit.

Tf = 4 always

Ti The time required to transfer an integer value to or from the NS32081 Floating-Point Unit.

Byte: Ti = 2

Word: Ti = 2

Double-word: T1 = 4

TABLE 3-5. FLOATING-POINT INSTRUCTION EXECUTION TIMES

MNEMONIC	CASE	TEA	TOPD	TOPi	Ti	Tf	TCY
MOVf	<MM>	2	2f	-	-	2f	23
	<RR>	-	-	-	-	-	27
	<MR>	1	f	-	-	f	23
	<RM>	-	f	-	-	f	27
ADDf, SUBf	<MM>	2	3f	-	-	3f	70
	<RR>	-	-	-	-	-	74
	<MR>	1	f	-	-	f	70
	<RM>	1	2f	-	-	2f	70
MULf	<MM>	2	3f	-	-	3f	30+14f
	<RR>	-	-	-	-	-	34+14f
	<MR>	1	f	-	-	f	30+14f
	<RM>	1	2f	-	-	2f	30+14f
DIVf	<MM>	2	3f	-	-	3f	55_30f
	<RR>	-	-	-	-	-	59+30f
	<MR>	1	f	-	-	f	55+30f
	<RM>	1	2f	-	-	2f	55+30f
ABSf, NEGf	<MM>	1	2f	-	-	2f	20
	<RR>	-	-	-	-	-	24
	<MR>	1	f	-	-	f	20
	<RM>	-	f	-	-	f	24
CMPf	<MM>	1	2f	-	-	2f	45
	<RR>	-	-	-	-	-	49
	<MR>	1	f	-	-	f	45
	<RM>	1	f	-	-	f	45
MOVLF	<MM>	1	3	-	-	3	23
	<RR>	-	-	-	-	-	27
	<MR>	1	2	-	-	2	23
	<RM>	-	1	-	-	1	27
MOVFL	<MM>	1	3	-	-	3	22
	<RR>	-	-	-	-	-	26
	<MR>	1	1	-	-	1	22
	<RM>	-	2	-	-	2	26
MOVif	<MM>	1	f	1	1	f	53
	<MR>	1	-	1	1	-	53
ROUNDfi, TRUNCfi,	<MM>	1	f	1	1	f	53
FLOORfi	<RM>	-	-	1	1	-	66
SFSR	<M>	-	1	-	-	1	13
LFSR	<M>	1	1	-	-	1	18

NS32CG16 Printer/Display Processor Programmer's Reference Supplement

The NS32CG16 is a new member of the *Series 32000* processor family. A full 32-bit CMOS design combines high speed with low power consumption and provides full compatibility with existing *Series 32000* software.

This document describes the NS32CG16 graphics instructions, provides programmers with the knowledge to use those instructions, and serves as a supplement to the *Series 32000 Programmer's Reference Manual*. Both documents are required to review the complete instruction set of the NS32CG16.

Chapter 1 provides an introduction to the NS32CG16 printer/display processor.

Chapter 2 provides a complete description of each graphic instruction.

Chapter 3 shows the expected timing of the NS32CG16 graphics instructions.

As this is advance information, it is subject to change without notice.

Publication number 424511080-001