NCR

Microelectronics Division
Colorado Springs

NCR 53C700

SCSI I/O Processor

Programmer's Guide

TABLE OF CONTENTS

NCR SCSI I/O Processor (53C700)

1 INTRODUCTION

I/O Performance

The demands on today's I/O interfaces is being pushed by the increasing performance of personal computers and workstations. Extremely fast CPU's, both CISC and RISC, only provide marginal system performance if their I/O interfaces are not properly designed. Faster processors do not equate to higher performance. Amdahl's Law covers this situation. "Assume I/O represents 10% of the system activity and its performance is kept constant. If CPU power is increased by a factor of 10:1, the net improvement is only 5:1. A 100:1 increase in CPU power is valueless if the net improvement in systems performance is only 10:1." Interrupt service routines, which often take in excess of several hundred microseconds to execute, can be a large source of these performance delays. Interrupts may be generated for exception conditions, I/O completion, saving/restoring buffer data pointers (for system check-point/restart), or low probability events available as options in todays SCSI definition. Interrupts can be reduced by using programmed I/O, however, this can be time consuming and requires much of the host computer cycle time. Therefore programmed I/O is not an adequate solution for multi-tasking operations. Another real performance issue is the so called scatter/gather operation. With virtual storage so common today, many I/O's gather the data from several physical addresses in system memory. Latencies inherent in the reinstruct DMA operation can cause serious performance degradation, by allowing the disk drive to slip a latency while the DMA is being reinstructed.

I/O Flexibility

Options in bus protocol allow for increased flexibility. This flexibility is partially responsible for the popularity of the SCSI standard. Flexibility provides users with the ability to configure their systems for a wide range of peripherals (from high performance disk drives to hand held scanners). Additionally, this flexibility offers support for command queueing, asynchronous or synchronous data transfers, caching controllers, peer to peer communication, etc. Unfortunately, flexibility implies firmware complexity, and if these options are not carefully implemented, performance will suffer.

A Better Solution is Required

First generation (NCR5380) SCSI devices were register oriented and required processor intervention even to make the most fundamental protocol decisions. Users liked the flexibility of these devices because the low-level firmware interface provided them with specific real time

information about the SCSI bus and improved testability of the SCSI device. This generation of devices typically requires in excess of 4,000 lines of code to specify a SCSI-1 device implementation.

Second generation (NCR53C90) SCSI devices provide on-chip state machines, allowing some complex SCSI sequences to be performed automatically, thereby reducing protocol overhead. However, these devices have no decision making capability, because the internal sequences are fixed in hardware at VLSI design time. Greater than 2,500 lines of driver software is typically required to support this class of SCSI device.

The flexibility of the SCSI bus has caused system integrators and OEM's alike, to struggle with the decision of using these first or second generation SCSI devices standalone or integrating them into intelligent host adapter boards. Non-intelligent SCSI host ports or host bus adapters require a fair amount of processor intervention, however they are relatively inexpensive to implement. Intelligent host adapters, which are an order of magnitude more expensive than non-intelligent adapters, provide slower decision making capabilities (less powerful CPU's), experience interpretation delays (2-8msec required to start any I/O), and suffer from interprocessor communication delays. For these reasons, non-intelligent host adapters outperform their intelligent counterparts in many systems that do not require some type of complex buffering scheme. On the peripheral controller side, space is at a premium and complex peripheral interfaces require powerful microprocessors to transfer data at the high rates available off the peripheral interface. Therefore, SCSI chips that require intense firmware can cause the controller microprocessor to be overworked and unable to perform the required tasks. Because of the limited space available, adding an extra processor, or replacing it with a more powerful one is not always possible.

With MIPS increasing in the system CPU, the delays cause by intelligent host adapter cards and slow peripheral controllers become painfully obvious to the system integrator. The simple solution to this problem is to build complex, versatile, H/W sequences inside the SCSI components or to add additional CPU power in the SCSI device board. Of course both of these solutions are costly (space and component cost) and do not adequately address the problem.

Third Generation Requirements

To adequately accommodate the flexibility requirements of the SCSI bus (reducing interrupts and controlling board cost), an additional level of intelligence and integration is required for next generation silicon. Third generation SCSI chips must be able to make execution decisions based on phase changes on the SCSI bus, as well as compare for specific incoming data values, resulting in a minimum number of interrupts to the external processor. Thus, a truly programmable SCSI chip that executes SCSI oriented commands is required. Additionally, these new chips must be able to reduce interrupt service routine complexities by providing unique status values to the external processor for the few interrupts that do occur. A fully integrated DMA channel must be provided to allow full use of available host bus bandwidth. With todays virtual memory schemes, the ability to support scatter/gather memory operations without processor intervention is key to overall I/O performance. A few hundred lines of driver code is all that's needed to support third generation SCSI devices. This code is required for exception conditions and for passing addresses of the user data buffer to the chip. Error recovery must occur at the high level interface. In second generation chips, the firmware is required to manage every detail of the error recovery mechanism, because the high level interface is fixed and has only one entry point. Programmable SCSI chips allow error recovery using the high level interface, because the algorithm can be entered at any command, and error specific SCSI SCRIPTS™ can be developed.

The NCR SCSI I/O Processor

The NCR 53C700 is the first truly intelligent SCSI host adapter on a chip. A high-performance re-usable SCSI core and an intelligent 32-bit bus master DMA have been integrated with a SCSI SCRIPTS processor to accommodate the flexibility requirements of SCSI-1, SCSI-2, and eventually SCSI-3. This flexibility is supported while solving the protocol performance problems that have plagued both intelligent and non-intelligent adapter designs.

SCSI Component

In addition to the reliability components of NCR's other SCSI chips:

- 10K volts ESD protection
- >350mV Bus Hysterysis
- immunity to bus reflections due to impedance mismatches
- controlled bus assertion times which reduces generated RFI, improves reliability, and increases the chances for FCC approval
- latch-up protection >100mA
- voltage feed-thru protection

The SCSI core that is in the 53C700 is reusable and designed to migrate to SCSI-2 wide and fast requirements. As part of the 53C700 it offers synchronous transfers up to 6.25MBytes/sec with asynchronous transfers greater than 5MBytes/sec. Synchronous offsets up to 8 are supported.

The SCSI core offers low-level register access as well as the high-level control interface. Like first generation SCSI devices, the 53C700 SCSI core can be accessed as a register oriented device. The ability to sample and assert any signal on the SCSI bus can be useful in manufacturing test as well as in diagnostic procedures. Loopback diagnostics are supported to the extent that the SCSI core may perform a self-selection and operate as both an initiator and a target, verifying that internal data paths are operational. Another important feature is the ability to test the SCSI pins for physical connection to the board or the SCSI bus.

Unlike previous generation devices, the SCSI core is controlled by the integrated DMA through a high-level logical interface. High level programming language commands that control the SCSI core may be chained from main host memory. These commands instruct the SCSI core to select, reselect, disconnect, wait for a disconnect, transfer user data, transfer SCSI information, change bus phases, and in general, implement all aspects of the SCSI protocol. Also, the SCSI SCRIPTS processor will transfer execution control (jump, call,return and interrupt) based on

SCSI bus phase comparisons. A value in the SCSI SCRIPTS command can be compared to the actual data in value on the SCSI bus, allowing the same transfer of control based on input data compares. A special on-chip 2MIPS processor referred to as the SCSI SCRIPTS processor provides this capability.

DMA COMPONENT

The DMA component is a bus master DMA device that is easily attached to the 80486, 80386, 80286, 80386SX, and 80376 processors. It has been designed to 25 Mhz 80386 bus timings and may be externally adapted to ISA (AT), EISA, Micro Channel™ , etc.

The chip supports 16 or 32-bit memory and automatically supports misaligned DMA transfers. As with the 80386, data bus enables are provided for each byte lane. An on-chip 32 byte FIFO allows 2,4, or 8 long words to be burst across the memory bus interface providing memory transfer rates in excess of 50 MBytes/sec.

Since the DMA is tightly coupled to the SCSI core through the SCSI SCRIPTS processor, uninterrupted scatter/gather memory operations are supported, with only a 500 nano second delay between memory segment transfers.

A Watchdog Timer is included as a "bus safety" feature and a flexible arbitration scheme allows for either daisy chained or "ored" memory bus request implementations.

SCSI SCRIPTS™ Processor

The SCSI SCRIPTS processor is a specially designed 2 MIPS processor that allows both DMA and SCSI instructions to be fetched from host memory. Algorithms written in the SCSI SCRIPTS language and then compiled, can control the actions of the SCSI and DMA cores and are executed from 16 or 32-bit system memory. This allows complex SCSI bus sequences to be executed independently of the host CPU.

One of the powerful aspects of the SCSI SCRIPTS processor is the ability to begin a SCSI I/O operation in 500nsec. This compares to the 2-8msec required for traditional intelligent adapters. The SCSI SCRIPTS processor not only offers performance but allows algorithms to be tailored to tune SCSI bus performance, adjust to new bus device types (i.e. scanners, communication gateways, etc.), or adapt to changes in the SCSI logical bus definitions and quickly incorporate new or popular options. Therefore, SCSI flexibility can be implemented without sacrificing I/O performance.

SCSI SCRIPTS are entirely independent of the CPU and system bus being used. This means that scripts for an EISA implementation of a 80386 can be identical to the scripts for a 80386SX Micro Channel™ implementation.

NCR SCSI SCRIPTS™ Description

After power up and initialization of the 53C700, the user may operate the chip in one of two modes; 1) low level register interface, 2) SCSI SCRIPTS chained mode.

In the low level register interface, the user has access to the DMA control logic and the SCSI bus control logic and is able to operate the chip much like an NCR 53C80. An external processor has access to the SCSI bus signals and the low level DMA signals, allowing a user to devise a complicated board level test algorithm. The interface is useful for backward compatibility with SCSI devices that require certain unique timings or bus sequences to operate properly. Another feature allowed at the low level is loop back testing. With the loop back mode, a user can direct the SCSI core to talk to the DMA core for testing internal data paths all the way out to the chip's pad.

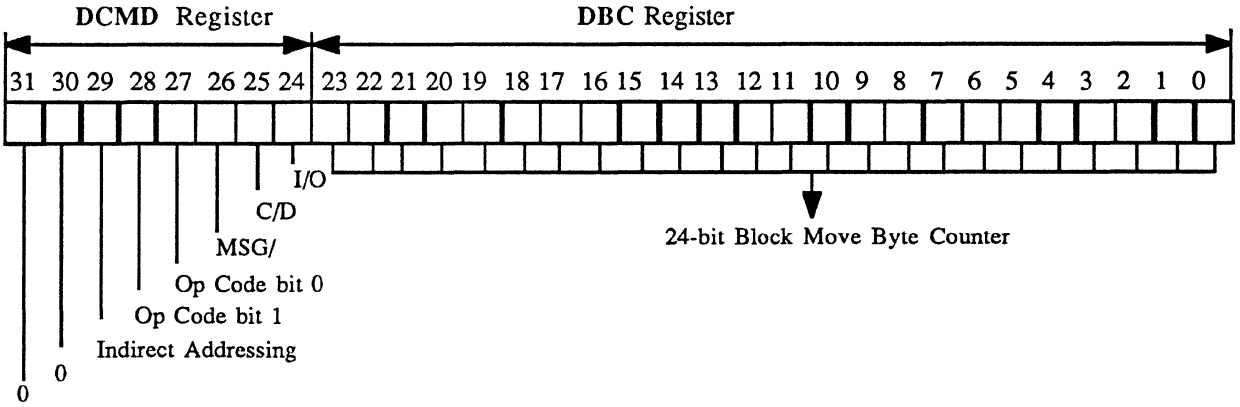
Operating in the chained mode, the 53C700 requires only a SCSI SCRIPTS start address and then all subsequent commands are fetched from external memory. Four bytes (or optionally two) at a time are fetched across the iAPX 286/386 DMA interface and loaded into the command register. Command fetch and decode time is minimal (about 500 ns), so little performance penalty is paid for this feature. Commands will be fetched until an interrupt command is encountered, or until some external, unexpected event (e.g. hardware error detected) causes an interrupt to the external processor. Given the rich set of SCSI oriented features included in the command set, and the ability of the user to re-enter the SCSI algorithm at any point, this high level interface is all that is required for both normal and exception conditions. Therefore the user is never required to switch to a low level mode for error recovery as is the case with today's second generation SCSI VLSI.

2 SCSI SCRIPTS™ Machine Language Description

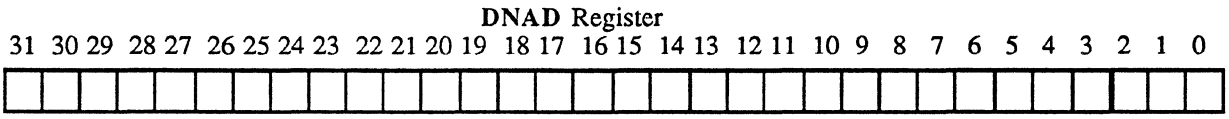
The purpose of this section is to describe the details of each SCSI SCRIPT™ command at a detailed, bit level from a user's (programmer's) point of view. Certainly, a user will normally use the SCSI SCRIPTS compiler described in following sections, but for debugging purposes, the details of each command must be described. Each command description consists of a bit diagram of the command, a brief overview of the command and a description of each field within the command. In the most general case bits 31-30 are SCSI I/O Processor opcodes with 00 equals Block Move Command, 01 equals I/O Command, 10 equals Transfer Control Command and 11 equals NCR Reserved.

BLOCK MOVE COMMAND

First 32-bit word of the Block Move Instructions



Second 32-bit word of the Block Move Instructions



Overview

The Block Move command is used to transfer data to(from) user memory from(to) the SCSI bus. No distinction is made between user data and SCSI information, such as command or message bytes. Thus a series of SCSI SCRIPTS is written to move all types of data, with no requirement for separate firmware to distinguish between user and SCSI data. Also, note that the data may come from any memory address, so scatter/gather operations for user data are transparent to the chip and the external processor. The user need only write a separate Block Move for each piece of data to be moved. To speed data transfers between user memory and the I/O Processor, there is a 32 byte DMA data buffer. An 8 byte FIFO exists for synchronous SCSI data in transfers.

Note: The possible values of each field is given in binary.

BLOCK MOVE COMMAND - FIRST SCRIPTS WORD

Block Move opcode -- 00 Bits 31-30

Indirect data address flag (I) Bit 29

0--SCSI or user data is moved to(from) the 32-bit data start address for the block move. The value is loaded into the chip's address register and incremented as data is transferred.

1--The 32-bit SCSI or user data start address for the Block Move is the address of a pointer to the actual data buffer address. The value at the 32-bit data start address is loaded into the chip's DNAD register via a second long word (four byte transfer across the host computer bus. Note that this option implies three DMA long word transfers, rather than two transfers. Once the actual data buffer address is loaded, the chip executes as if it were in the direct mode. This indirect feature allows a user to specify a table of data buffer addresses. Using the NCR SCSI SCRIPTS compiler, the table offset is placed in the script at compile time. Then at actual data transfer time, the offsets must be added to the base address of the data address table by the external processor. Such a feature allows the logical I/O driver to build a structure of addresses for an I/O rather than having to treat each address individually. Also, having SCSI SCRIPTS in a PROM is possible with this indirect feature.

Block Move Opcodes Bit 28-27

The SCSI role (target or initiator) causes the chip to react differently, with respect to the phase line values. An obvious difference is whether to sense or drive the SCSI phase lines. Also, there are major differences between the two roles when in the command phase. Therefore, the Block Move functions are discussed for each SCSI role.

Target Role Function--00
This operation will DMA user, or SCSI data, with the chip in the target role. First the chip determines whether the previous command has completed, or a reselect has occurred. The SCSI phase bits are asserted to the value requested by the Block Move command. If the command phase has been requested, the chip will:

- Wait for the first byte received.
- Decode the byte to determine the number of SCSI command bytes to receive.
- Write the command length into the DBC register. An invalid group code value causes the chip to use the original value in the DBC register. If this value is zero, the chip will stop, interrupt with the first byte, and stop transferring command bytes.
- Transfer the correct number of bytes into the address designated by the Block Move command.

If any phase (other than command) is requested, the chip will transfer the number of bytes requested to(from) the address requested. Should the initiator turn on attention at any time during the transfer, the transfer will be completed, and then an interrupt will occur.

Target Role Function--01,10, or 11
These are illegal values, and will generate an invalid command interrupt if the chip is in the target role.

Initiator Role Function--00
Reserved

Initiator Role Function--01
This operation will wait for a valid phase and DMA data with the chip in the initiator role. The chip verifies that the previous command is complete or a reselect has occurred, and then waits for a previously unserviced phase before executing the Block Move command. Thus, the user can program the chip to pause until the SCSI device it is communicating with is proceeding to the next phase. A comparison is made between the expected phase bits in the SCSI SCRIPTS and the latched phase value. If the two values are not equal, the chip issues a phase mismatch interrupt and halts execution. One normally uses this wait capability to allow the target to pace the chip in the initiator role. When a phase change is expected, the wait is used to synchronize the expected phase with the Block Move for that phase.

Initiator Role Function--10, or 11
These are illegal values, and will generate an invalid command interrupt if the chip is in the initiator role.

SCSI Phase Lines Bit 26-24

These are the three SCSI phase lines used to compare to the actual SCSI bus phase lines. The SCSI bus phase value is latched when REQ goes active. The value is stored in SSTAT2 (bit 2 through bit 0 -- MSG, C/D, & I/O). Before any data is moved, the chip will compare (or wait for a new phase and compare).

24 Bit Byte Count Bit 23-00

This count value specifies the exact number of data bytes to be moved between the SCSI bus and system memory. As the SCSI SCRIPTS command is decoded, the value is moved into the DBC register. When the user specified burst size of data is available in the DMA FIFO, the SCSI I/O Processor will:

- Gain access to the system bus.
- Transfer the burst size.
- Decrement the byte counter (byte count).
- Increment the next address register (data address).

The process will continue until byte count is zero. At that time, the next SCSI SCRIPTS command will be fetched.

BLOCK MOVE - SECOND SCRIPTS WORD

Data Start Address for the Block Move Bit 31-00

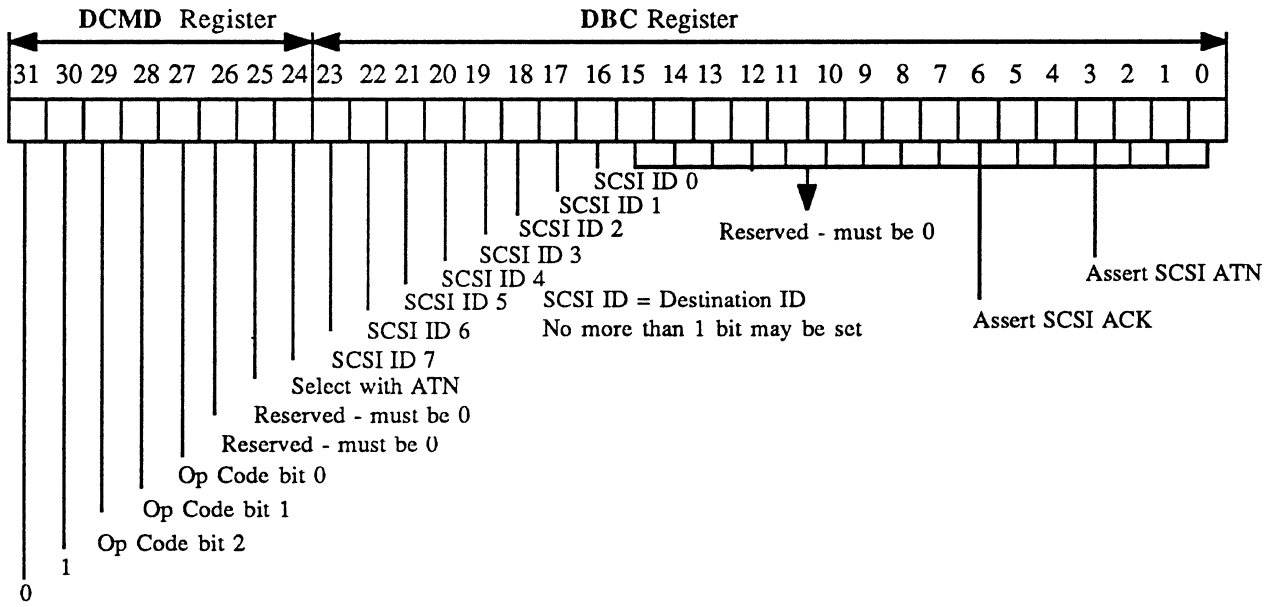
This value specifies the address of data in memory (direct mode) or the address of the actual address (indirect mode). The DNAD register is updated with the address of the actual data and is incremented with each chip DMA transfer.

The Block Move command is very powerful for several reasons. 1)No distinction is made between user data and SCSI command, message, or status data. 2)Data can be stored in any area of system memory with little performance impact (one command fetch). 3)The indirect feature allows a table of addresses instead of requiring the address to be in the command. 4)A scatter/gather operation also has little performance impact, because the only overhead is 500 nano seconds (direct mode) or 750 nano seconds (indirect mode). So, having one Block Move command for each segment of data in memory is economical with the SCSI I/O processor architecture.

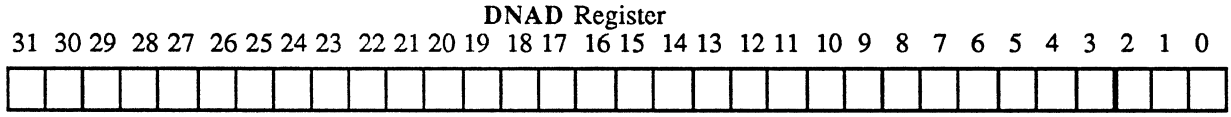
In the initiator role, the Block Move wait feature is useful for high performance SCSI SCRIPTS that do not compare for any unexpected phases before executing a Block Move command. If the phase does not match, then an external interrupt is generated, but in the high performance SCSI SCRIPTS algorithm, exceptions are abnormal and handled by the external processor. Normally the Conditional Transfer command (see below) is used for comparing actual to expected phase. The first Conditional Transfer command must have the "wait" option on (to synchronize the commands with the actual bus phase), and each subsequent command should have the "wait" option turned off.

I/O Command

First 32-bit word of the I/O Instructions



Second 32-bit word of the I/O Instructions



Overview

The I/O command is used to perform certain SCSI operations such as select and reselect. Each function defined is a direct command to the SCSI portion of the 53C700. The functions vary according to whether the chip is in the target or initiator role, so the functions are described separately for each role.

I/O COMMAND - FIRST SCRIPTS WORD

SCSI I/O Processor opcode -- 01 Bits 31-30

I/O command Opcodes Bits 29-27

Five functions have been defined for target and initiator role, with three reserved for future expansion. If reserved function codes are used, an illegal command interrupt is generated and execution will stop.

Target Role -- function 000

Perform reselection -- The chip will arbitrate for the SCSI bus and then perform a reselection. Arbitration continues until the chip is successful, unless there is some bus initiated interrupt (e.g. selection). If arbitration terminates because of a bus initiated interrupt (selection or reselection) the chip will use the 32-bit jump address value to fetch the next instruction and begin execution at that

address. If the command is successful, then the next sequential instruction is fetched and executed. Note that the target/initiator role will automatically change to reflect what is actually happening on the bus. After completion of the bus initiated interrupt processing (sequence goes to bus free), the chip will revert to the role set by the user in the registers. Some caution is required here. If the chip is set to an initiator role, gets selected, changes to the target role automatically, disconnects, does some processing, and then issues a reselect command (without being set to the target role by the external processor), a selection will occur. Because the chip was in the initiator role (at the time of selection), it will revert to that role after the disconnect and bus free.

Target Role -- function 001
Perform disconnect -- The chip will physically disconnect from the SCSI bus.

Target Role -- function 010
Wait for select -- The chip will wait for a SCSI selection by some other device on the SCSI bus. If the chip has already been selected, then the next SCSI SCRIPTS will be fetched and executed. In the case of a bus initiated interrupt of reselect, the chip will change to the initiator role and fetch the next command from the address pointed to by the 32-bit jump address, continuing execution from there.

Target Role -- function 011
Assert bit -- The chip will assert the latches in the SCSI output data register, but nothing will be driven onto the SCSI bus. Consequently, this function should not be used in the target role.

Target Role -- function 100
Reset bit -- The chip will reset the latches in the SCSI output data register, but nothing will be reset on the SCSI bus. Consequently, this function should not be used in the target role.

Target Role -- function 101, 110, 111
These are not currently defined and will cause an illegal command interrupt if they are used.

Initiator Role -- 000
Perform selection -- The chip will arbitrate for the SCSI bus and then perform a selection. Arbitration will continue until the chip is successful, unless there is a bus initiated interrupt (e.g. reselection). If arbitration terminates because of a bus initiated interrupt (as a result of a select or reselect), the chip will use the 32-bit jump address to fetch the next instruction and begin execution at that address. Note that the target/initiator role will automatically change to reflect bus actions. After completion of the bus initiated interrupt processing (sequence goes to bus free), the chip will revert to the role set by the user. If the selection is successful, the next instruction is fetched and executed. If bit 24 (the attention flag) is set, then the chip will perform a select with attention.

User's Note:
Because the chip will automatically change roles and jump to an alternate address if the select or reselect fails, then a bus initiated interrupt can be processed by the chip with no external intervention. The alternate jump address should contain the address of an algorithm for a selection or reselection. That address should contain a wait for selection (target role) command. That command's alternate address is the reselection algorithm (initiator role). In this manner, the chip can determine exactly what happened and transfer control to the appropriate SCSI SCRIPTS algorithm.

Initiator Role -- 001
Wait for disconnect -- The initiator waits for a disconnect from the SCSI bus. A legal disconnect is defined as a loss of busy and select for the specified bus free time following a DISCONNECT message or a COMMAND COMPLETE message. If the disconnect is legal, the next SCSI SCRIPTS command will be executed, otherwise an unexpected disconnect interrupt will be generated.

Initiator Role -- 010
Wait for reselection -- The initiator waits for a reselection from a previously selected SCSI device. If the operation completes as expected, then the next instruction is fetched and executed by the chip. However, if the device is selected, then the alternate jump address should contain the address of an algorithm for a selection. That address should contain a wait for selection (target role) command. That command's alternate address is the error recovery algorithm (for initiator role -- reselect). In this manner, the chip can determine exactly what happened and transfers control to the appropriate SCSI SCRIPTS algorithm.

Users Note:
With the 53C700 byte compare capability of the transfer control command, the SCSI SCRIPTS algorithm can determine which target reselected the initiator and can jump to the correct algorithm for that particular target. Thus SCSI SCRIPTS can be tuned for the various types of targets available and executed with no external processor intervention.

Initiator Role -- function 011
Assert bit -- The chip will assert the SCSI bus bits requested in the flags field. Currently two bits are defined, allowing the SCSI ACK and ATN bits to be set. Bit 6 is for Acknowledge and bit 3 is for Attention.

Initiator Role -- function 100
Reset bit -- The chip will reset the SCSI bus bits requested in the flags field. Currently two bits are defined, allowing the SCSI ACK and ATN bits to be reset. Bit 6 is for Acknowledge and bit 3 is for Attention.

Initiator Role -- function 101, 110, 111
These are not currently defined and will cause an illegal command interrupt if they are used.

SELECT WITH ATN - Bits 26-24
If bit 24 is set, then the initiator SELECT command will cause the SCSI attention line to be set during the SELECT operation. Attention on is valid only during the initiator function 000. The bit is invalid for all other functions, and will cause an interrupt.

SCSI I.D. 7-0 - Bits 23-16
This eight bit field is the I.D. for the SCSI device that is to be selected in the initiator role and reselected in the target role. Only one bit should be set for either of the functions requested. These bits are not used for any function other than select or reselect.

Flags Field - Bits 15-00
These bits are used during the set or clear command. Bit 6 on will cause the SCSI acknowledge to be set/reset, and bit 3 on will cause the SCSI attention to be set/reset. Note that the clear ACK command should be used after the last target message-in byte has been verified for each separate message data Block Move command. The initiator is given the opportunity to set attention before acknowledging the last message byte of a Block Move command. On each byte, if a parity error was detected on the message in operation, then the ASSERT SCSI ATN should be issued

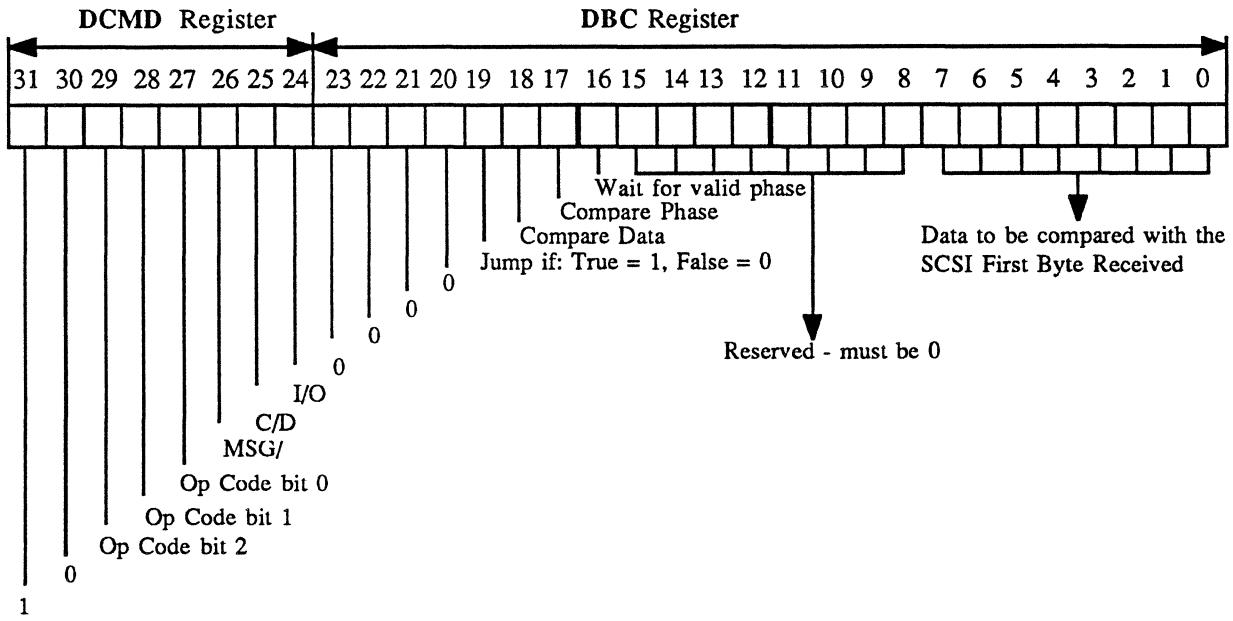
before the clear acknowledge is issued to accept the message. Set Acknowledge can be used to handshake bytes across the SCSI bus, and clear attention should be issued after the target has serviced the request for a message out by the initiator.

I/O COMMAND - SECOND SCRIPTS WORD

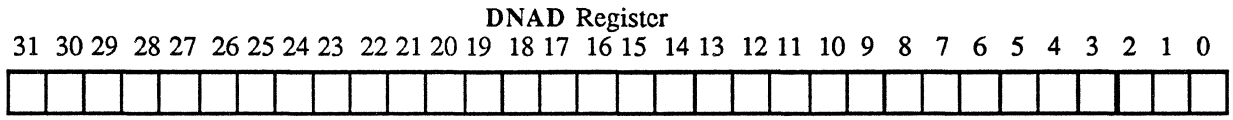
Jump Address - Bit 31-00
If the select, wait reselect, or reselect command fails, this thirty-two bit field specifies the memory address from which the next SCSI SCRIPTS is fetched for execution. Normally, the next instruction is fetched in sequence if the requested operation completes with no bus initiated interrupt.

Transfer Control Command

First 32-bit word of the I/O Instructions



Second 32-bit word of the I/O Instructions



Overview

The Transfer Control Command is comprised of JUMP, CALL, RETRUN, and INTERRUPT operation codes. Each opcode is conditionally performed based on comparers of SCSI phase values and incoming SCIS data values. The purpose of the Transfer control command is to allow the user to compare for current phase values on the SCSI bus or the first byte of data on any incoming bytes and transfer control to another address depending on the results of the test. These commands allow SCSI algorithms to be written in SCSI SCRIPTS and give the 53C700 characteristics of a general purpose SCSI processor. With transfer control commands, the user can truly program the chip, rather than simply buffer commands to be executed in a serial fashion with no real-time decision making capabilities.

Command Fields

TRANSFER CONTROL COMMAND - FIRST SCRIPTS WORD

SCSI I/O Processor opcode -- 10 - Bits 31-30

Transfer Opcodes - Bits 29-27

Four opcodes are currently defined that allow a transfer of control in the SCSI SCRIPTS language. All undefined opcodes cause an interrupt of illegal command.

JUMP Command -- 000
If the condition evaluates according to the sequence control bits so the jump must be taken, the next instruction will be fetched from memory at the 32-bit jump address. Otherwise, the next sequential address will be used as the instruction fetch address.

CALL Command -- 001
If the condition evaluates according to the sequence control bits so the call must be taken, the next instruction will be fetched from memory at the 32-bit call address. Otherwise, the next sequential address will be used as the instruction fetch address. The address of the next sequential command is stored in the chip's TEMP register in anticipation of a subsequent return address. Note that if two CALL instructions are executed without any intervening RETURN instruction, then the first return address in the chip's TEMP register will be overwritten by the second CALL.

RETURN Command -- 010
If the condition evaluates according to the sequence control bits so the return must be taken, the next instruction will be fetched from memory at the 32-bit address contained in the TEMP register, having been stored there by the previous call instruction. Otherwise, the next sequential address will be used as the instruction fetch address. Note that the contents of the TEMP register may be undefined if a call instruction has not been previously executed.

INTERRUPT Command -- 011
If the condition evaluates according to the sequence control bits so the software interrupt must be taken, the chip will halt execution and issue an interrupt request to the external processor. Otherwise, the next sequential address will be used as the instruction fetch address. Note that the 32-bit jump address in the instruction is available in the chip's command register at the time of the interrupt. In this manner, a user can post a four byte, user unique error status to be used by the external processor's interrupt service routine. Thus, the cause of the interrupt can be easily decoded by firmware, thereby reducing firmware interrupt service routine overhead.

SCSI Phase Bits - Bits 26-24

In the SCSI initiator role, these bits are used to compare with the actual SCSI lines (MSG, C/D, and I/O) if the phase compare bit is set in the sequence control field. Actual SCSI lines are a copy of the last valid SCSI phase line values. A user can set these bits in the SCSI SCRIPTS command to compare with the current SCSI bus phase lines, and branch to the SCSI SCRIPT™ that was written to process the particular phase that is currently active. Bit 26 is SCSI MSG, bit 25 is SCSI C/D, and bit 24 is SCSI I/O. In the target role, these bits are ignored because the chip is driving them in the target role.

Bits 23-20

These bits are reserved for future use and must be zero.

Bits 19-16

Sequence Control Bits

SCSI SCRIPTS can use the current conditions on the SCSI bus to determine where to transfer control and execute alternative algorithms, using the sequence control bits. The bits are defined as follows:

- Bit 19 -- Transfer if True/False. If the bit is set to 1, a transfer of control will occur if the phase or data values in the instruction is equal to the actual phase value on the SCSI bus or the first byte of the most recent asynchronous in phase. Note that the byte could be a message in, data in, or status for the initiator and message out, command, or data out for the target role. When the bit is set to zero, the transfer control will occur if the comparison yields a false.

- Bit 18 -- Compare the data byte value (bit 7 - bit 0 in the instruction) to the first byte of the most recent data, message, command, or status byte received. The user's SCSI SCRIPTS program can determine what routine to execute next, based on actual data values received across the SCSI bus. For example, the chip can compare for specific message values and process an extended message in SCSI SCRIPTS, with no external interrupt to the external processor.

- Bit 17 -- In the initiator role, compare the SCSI phase line value (bit 26 - bit 24) to the most recent valid SCSI phase line values saved in the chip. Using this feature, the chip can react to actual bus conditions and determine which routines to execute next based on SCSI bus phase line values. Unexpected phase values can be compared for and error conditions or low probability events can be processed by SCSI SCRIPTS inside the chip. In the Target role, bit 17 on causes the chip to test for the attention line on. If the initiator has set attention, the chip (in the target role) can jump to a message out routine to determine what the initiator needs. Note that this is normally placed after each SCSI phase to allow the initiator to turn on attention if an error is detected during the transfer.

- Bit 16 -- In the initiator role, wait for a previously unserviced phase change. Thus, the user can program the chip to pause until the SCSI device it is communicating with has proceeded to the next phase. One normally uses this wait capability to pace the chip in the initiator role. When a phase change is expected, the wait is used to synchronize the expected phase with the actual phase detected on the SCSI bus. Note that if both data and phase compare bits are set, the compare must be both true or both false for the transfer to occur.

Bits 15-8
Reserved

Bits 7-0
Data Byte -- Compare this data byte value to the first byte of the most recent asynchronous data, message, command, or status byte received. The user's SCSI SCRIPTS program can determine what routine to execute next based on actual data values received. Using a series of these compares, the algorithm can process complex sequences with no intervention required by the external processor.

TRANSFER CONTROL COMMAND - SECOND SCRIPTS WORD

Data Jump Address - Bit 31-00
This value specifies the address of the next instruction in memory that control should be transferred to. The value is ignored in a return command and in the interrupt command. However, it is loaded into the chip's command register and is available to be read by firmware in the case of an interrupt command.

Several points should be noted about the transfer control command. If both data compare and phase compare bits are set, then both comparisons must equate to true or both must equate to false before the requested transfer will occur. There is no way to test one for false and the other for true. If neither the phase or data bit is set, and if the true/false bit is 1, the operation is executed unconditionally. If neither the phase nor the data bit is set and the true/false bit is 0, then the command is a no operation and can be used for a delay function, or to reserve SCSI SCRIPTS patch area.

3 DEVELOPING NCR SCSI SCRIPTS™

NCR Microelectronics is plannig to support the development of SCSI SCRIPTS with an integrators developer's kit. This kit includes:

- Sample SCRIPTS
- SCRIPTS Utilities
- Test/Diagnostic SCRIPTS
- A SCRIPTS compiler
- Hardware Test Support

Your local NCR Sales Office or Factory Representative will inform you of the current software release and current board level options.

To develop an executable SCSI SCRIPT, the user must first define the SCSI functions required, paying careful attention to what functions are to be executed in SCRIPTS and what functions must be contained in system firmware. Then the specific algorithms must be designed for the functions that are to be executed in the SCSI SCRIPTS portion of the SCSI logical I/O driver. Using the SCRIPTS compiler, the algorithms are coded in SCRIPTS and compiled to create the object code required as input by the 53C700. The compiler output is much like an object module, because it includes relocation information required to load the SCRIPTS object module into main memory. At load time, the SCRIPTS jump addresses must be resolved using one of the utilities furnished in the software package, and at start I/O time, another utility must be used to patch in the correct buffer addresses, byte counts, destination I.D., etc.

The 53C700 allows a logical I/O driver to be written very easily. The first SCSI SCRIPTS example illustrates how easy the task actually is. This code will perform a read or write function, using the 53C700 in the high level chained mode. Because SCSI algorithms are so simple when written in SCSI SCRIPTS, the user can rapidly prototype SCSI algorithms for a proof of concept and then concentrate on more complicated, realistic algorithms sooner.

A SCSI SCRIPTS is comprised of two areas:

1.) Definition area
2.) SCRIPT area

In this example, the definition area is comprised of variable and absolute values. These values may describe a variable memory address location, variable byte count or a fixed status byte value.

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* The following are variable data values provided  *
;* external to the compiler and resolved at run-time *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
;       Definition area INITIATOR ROLE

;       Target Device I.D. to be fixed at Start I/O time.
EXTERNAL device

;       Ten byte buffer for sending messages
EXTERNAL sendmsg

;       Ten byte buffer for receiving messages
```

```
EXTERNAL rcvmsg

;       Number of message bytes to send after selection
EXTERNAL idcount

;       Number of command bytes
EXTERNAL cmd_count

;       Buffer for the SCSI command
EXTERNAL cmd_adr

;       Number of user data bytes
EXTERNAL data_count

;       Address of user data buffer
EXTERNAL data_adr
;       Error -- not message out after selection


;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*  Absolute values are stored in DNAD Register      *
;*  for purposes of interrupt processing               *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*  Note that 0X0 precedes the interrupt status        *
;*  values and designates a hex value                  *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *


ABSOLUTE err1 = 0x0ff01

;       Error -- unexpected SCSI phase before command phase
ABSOLUTE err2 = 0x0ff02

;       Error -- unexpected SCSI phase after a command transfer
ABSOLUTE err3 = 0x0ff03

;       Error -- expected status phase
ABSOLUTE err4 = 0x0ff04

;       No Error -- good I/O
ABSOLUTE ok = 0x0ff00

;       Error -- expected message outphase
ABSOLUTE err5 = 0x0ff05

;       Error -- expected message command complete
ABSOLUTE err6 = 0x0ff06
```

The following is a simple SCSI SCRIPT that performs a single-tasking SCSI operation without disconnecting. If an unpredictable event occurs on the SCSI bus, a unique interrupt status value is DNAI stored in the 53C700's register and is available for interrupt processing.

```
; select the device with attention on
select atn device resel_adr

;  if the next phase is not message out, interrupt
int err1 when not MSG_OUT

; sent the i.d. message out to the target
move idcount sendmsg when MSG_OUT

; if the next phase is not command, interrupt
int err2 when not CMD

; send the command bytes
move cmd_count cmd_adr when CMD

; go to process cleanup if status phase
jump end when STATUS

; process data in phase
jump input_data if DATA_IN

; or data out phase
jump output_data if DATA_OUT

; unexpected phase if here
int err3

; process the data in phase
input_data:
move data_count data_adr when DATA_IN

; and go process status
jump end

; process the data out phase
output_data:
move data_count data_adr when DATA_OUT

; interrupt if not status phase
end:
int err4 when not STATUS

; move the status byte into memory
move 1 status_adr when STATUS

; interrupt if message in is not next
int err5 when not MSG_IN

; move the command complete byte in
move 1 rcvmsg when MSG_IN

; interrupt if it is not a command complete message
int err6 if not 00
```

```
; accept the message if there are no problems
clear ack

; wait for a physical disconnect
wait disconnect

; interrupt with an I/O complete
int ok
```

4 NCR SCSI SCRIPTS™ Utilities

The following utilities will be provided as part of the integrators development package with the SCSI SCRIPTS Compiler for each user.

Initialize_IOP()
Sets up the 53C700 for operation after power up.

Save_IOP_State(savearea*)
Takes as an argument the pointer to a save area and stores the status of the 53C700 at that address. Information saved includes SCSI SCRIPT pointer, current data counter, buffer address, and all registers required to allow the state of the chip to be restored later. This routine is used during SCSI disconnect handling, or save data pointer operations.

Restore_IOP_State(savearea*)
Takes as an argument the pointer to the save area where Save_IOP_State has stored the 53C700 status data. Restores the chip state so an interrupted I/O can be resumed after a reselection or restore pointers operation.

IOP_Interrupt Status()
After the 53C700 experiences an interrupt, this routine is called to decode the chip's interrupt status information and report the reason for the interrupt.

Relocate_Script_Address(rel_info,base)
Relocates all transfer control addresses in a SCSI SCRIPT to the specified base address. Rel_info is a data structure produced by the back end of the SCSI SCRIPTS compiler.

Patch_Script(rel_info,symbol,value)
Using the symbol name for byte count, device i.d., or data address, and the rel_info data structure, this routine will patch the locations where the symbol is used with the input value.

5 The NCR SCSI SCRIPTS™ Language Syntax

Notation:

When something is enclosed in curly braces, it is optional.

If the end curly brace is followed by "...", then it means that the item enclosed in the curly braces can be repeated as often as is desired.

Something that is entirely in upper case is a keyword. Case is ignored by the compiler when looking for keywords.

Phase must be replaced with exactly one of the following keywords: MSG_IN, MSG_OUT, DATA_IN, DATA_OUT, CMD, STATUS, RES4, RES5

The word **'address'** means a 32-bit number.

The word **'value'** means a 32-bit number.

The word **'count'** means a 24 bit number.

The word **'id'** means an eight bit number that has exactly one bit set.

The word **'data'** means an eight bit number.

The word **'expression'** denotes a mathematical expression in the form of <identifier> [<addop> <identifier>]*, where <identifier> is any valid variable name or a numeric constant, and <addop> is either the '+' or '-' characters, denoting addition or subtraction respectively. Note that an 'expression' may be used in any place where one would normally use a numeric value. The value of all 'expressions' will be extended to 32-bit s. If an expression is used in a context in which the evaluated value of it must be less then 32-bit s, the least significant bits will be used. For instance, if an 'expression' is used to represent a count for a move instruction, the evaluated value will be truncated to 24 bits. The user will not be notified of the occurrence of this truncation unless it entails changing the value of the expression.

The word **'name'** means a string of one or more consecutive characters chosen from the letters, the numbers, the underscore, and the dollar sign. Note that names used for labels, for externals, and for variables in the relative data area will be passed on to the Host development system. If the Host development system has restrictions on the format of such names, it is up to the SCSI SCRIPTS writer to avoid using such names. For example, Turbo C, which is used as the Host development system for this proposal, does not allow names to begin with a digit or to contain a dollar sign. Thus, the SCSI SCRIPTS writer for DOS and Turbo C should avoid names of this form.

INPUT FORMAT

SCSI SCRIPTS consists of a series of lines. Blank lines, lines that contain only whitespace, and anything after a semi-colon on an input line is ignored by the front end.

The compiler is "token" oriented. It reads the input stream and splits it up into tokens. White space and anything from a semicolon to the end of the line is not part of any token, and is thus ignored by the first pass of the compiler.

There are two types of tokens. Any string of consecutive letters, numbers, dollar signs, and underscores is a token. Any given character can be part of no more than one token, and the input stream is split into tokens in such a way as to minimize the number of tokens. Thus, the string "abc" would be treated as one token ("abc") rather than multiple tokens ("a" and "bc", for example).

The second type of token consists of characters that are not part of other tokens. For example, anything that is not a letter, a digit, an underscore, or a dollar sign, will become a token.

For example, the string "xxx=0x123 ; assign value to xxx" contains three tokens. "xxx" is a token, "=" is a token, and "0x123" is a token.

Numeric values may be specified in decimal, hexadecimal, octal, or binary. Decimal numbers are specified by a string of digits that does not begin with a zero. Hex numbers are specified by a string consisting of "0x" or "0X" and the hex digits of the number. Both upper and lower case are allowed. A binary number is similar to a hex number, except that "0b" or "0B" is used instead of "0x" or "0X". An octal number is specified by a "0" followed by the octal digits.

Language Directives

There are several keywords that are used to provide information to the front end concerning the compilation of the SCSI SCRIPTS. These are used to define symbolic names, and to indicate certain things that need to be passed to the second pass of the compiler.

ENTRY label {,label...}

The ENTRY keyword indicates that the specified labels are SCSI SCRIPTS entry points. Their names and values will be made available to the back end, which will make them available to the Host development system.

ABSOLUTE name = expression {,name = expression...}

This declares symbolic names for numeric values. For example, "ABSOLUTE bad_cmd = 0x1200" allows the name "bad_cmd" to be used wherever a number would be allowed in the SCSI SCRIPTS. The SCSI SCRIPTS will be compiled as if the number 0x1200 had been specified instead of the name "bad_cmd" in every instruction that uses "bad_cmd".

EXTERNAL name {,name...}

This informs the compiler that the SCSI SCRIPTS will refer to variables with the specified names that are declared outside of the SCSI SCRIPTS. Note that some host development systems are not able to support this usage. SCSI SCRIPTS that need this feature may not be portable to all hosts.

RELATIVE name = expression {,name = expression...}

This is used to declare variables in the relative data area. "name" is the name of the variable, and "expression" is the offset from the start of the relative data area that the variable is located.

A name followed by a colon signifies a label. The name of a label can be used wherever an address is called for.

The SCSI SCRIPTS Instructions

When an instruction calls for a count to be specified, a 24 bit number may be used, or a symbolic constant (declared with the ABSOLUTE keyword) may be used.

When an instruction calls for an address, a 32-bit number may be used, the name of a label may be used, the name of a variable in the relative data area (declared with the RELATIVE keyword previously) can be used, or the name of an external variable (declared previously with the EXTERNAL keyword) can be used.

Labels, external variables, and relative variables all share the same name space. If a given name is declared more than once, the front end is free to resolve the conflict in any way it sees fit. It will issue a warning to let the user know that there is a possible problem.

If the address field of an instruction contains a name that has not been defined, then the front end will assume that it refers to a label that will be defined later. This is called forward referencing. If the name is later defined as an external or relative variable, this shall be considered a name conflict and the front end may resolve it in any way it wishes. It will issue a warning in this case.

BLOCK MOVE COMMAND

These are the various forms of the Block Move instruction.The 'address' and 'count' specify the address and byte count fields of the instruction. If the optional keyword 'PTR' is present, then the indirect bit will be set. 'Phase' specifies the phase field of the instruction. WITH or WHEN are used to specify the Block Move function codes. WITH is used to signal the target role which sets the phase values, and WHEN is the initiator "test for phase" feature.

MOVE count, { PTR } address, WITH Phase

MOVE count, { PTR } address, WHEN Phase

JUMP COMMAND

The conditional JUMP instructions all have the same general form. 'Address' is the SCSI SCRIPTS address that will be transferred to if the JUMP is taken. WHEN means that the Wait bit in the SEQ CNTL field is to be set. IF means that the Wait bit is not to be set. If the WHEN or IF is followed by NOT, then the True/False bit of the SEQ CNTL field is not set, otherwise, the bit will be set. If 'Phase' is present, then the compare Phase bit of SEQ CNTL will be set, otherwise, it will be cleared. If 'data' is present, the compare Data bit of SEQ CNTL will be set, otherwise, it will be cleared. If both 'Phase' and 'data' are specified, they must be in that order and they must be separated by the keyword AND. ATN is the target role version and required to test whether the initiator has set ATN on the bus. NOT is used for the inverse test of WHEN and IF. "NOT Phase OR data" is the negation of "Phase AND data".

```
NOP

JUMP address

JUMP address, IF ATN

JUMP address, IF Phase

JUMP address, IF data

JUMP address, IF ATN AND data

JUMP address, IF Phase AND data

JUMP address, WHEN Phase

JUMP address, WHEN data

JUMP address, WHEN Phase AND data

JUMP address, IF NOT ATN

JUMP address, IF NOT Phase

JUMP address, IF NOT data

JUMP address, IF NOT ATN OR data

JUMP address, IF NOT Phase OR data

JUMP address, WHEN NOT Phase

JUMP address, WHEN NOT data

JUMP address, WHEN NOT Phase OR data
```

CALL COMMAND

The conditional CALL instructions all have the same general form. 'Address' is the SCSI SCRIPTS address that will be transferred to if the JUMP is taken. WHEN means that the Wait bit in the SEQ CNTL field is to be set. IF means that the Wait bit is not to be set. If the WHEN or IF is followed by NOT, then the True/False bit of the SEQ CNTL field is not set, otherwise, the bit will be set. If 'Phase' is present, then the compare Phase bit of SEQ CNTL will be set, otherwise, it will be cleared. If 'data' is present, the compare Data bit of SEQ CNTL will be set, otherwise, it will be cleared. If both 'Phase' and 'data' are specified, they must be in that order and they must be separated by the keyword AND. ATN is the target role version and required to test whether the initiator has set ATN on the bus. NOT is used for the inverse test of WHEN and IF. "NOT Phase OR data" is the negation of "Phase AND data".

```
CALL address

CALL address, IF ATN

CALL address, IF Phase

CALL address, IF data

CALL address, IF ATN AND data

CALL address, IF Phase AND data

CALL address, WHEN Phase

CALL address, WHEN data

CALL address, WHEN Phase AND data

CALL address, IF NOT ATN

CALL address, IF NOT Phase

CALL address, IF NOT data

CALL address, IF NOT ATN OR data

CALL address, IF NOT Phase OR data

CALL address, WHEN NOT Phase

CALL address, WHEN NOT data

CALL address, WHEN NOT Phase OR data
```

RETURN COMMAND

The conditional RETURN instructions all have the same general form. 'Address' is the SCSI SCRIPTS address that will be transferred to if the JUMP is taken. WHEN means that the Wait bit in the SEQ CNTL field is to be set. IF means that the Wait bit is not to be set. If the WHEN or IF is followed by NOT, then the True/False bit of the SEQ CNTL field is not set, otherwise, the bit will be set. If 'Phase' is present, then the compare Phase bit of SEQ CNTL will be set, otherwise, it will be cleared. If 'data' is present, the compare Data bit of SEQ CNTL will be set, otherwise, it will be cleared. If both 'Phase' and 'data' are specified, they must be in that order and they must be separated by the keyword AND. ATN is the target role version and required to test whether the initiator has set ATN on the bus. NOT is used for the inverse test of WHEN and IF. "NOT Phase OR data" is the negation of "Phase AND data".

```
RETURN

RETURN, IF ATN

RETURN, IF Phase

RETURN, IF data

RETURN, IF ATN AND data

RETURN, IF Phase AND data

RETURN, WHEN Phase

RETURN, WHEN data

RETURN, WHEN Phase AND data

RETURN, IF NOT ATN

RETURN, IF NOT Phase

RETURN, IF NOT data

RETURN, IF NOT ATN OR data

RETURN, IF NOT Phase OR data

RETURN, WHEN NOT Phase

RETURN, WHEN NOT data

RETURN, WHEN NOT Phase OR data
```

INTERRUPT COMMAND

The conditional INT instructions all have the same general form. 'Address' is the SCSI SCRIPTS address that will be transferred to if the JUMP is taken. WHEN means that the Wait bit in the SEQ CNTL field is to be set. IF means that the Wait bit is not to be set. If the WHEN or IF is followed by NOT, then the True/False bit of the SEQ CNTL field is not set, otherwise, the bit will be set. If 'Phase' is present, then the compare Phase bit of SEQ CNTL will be set, otherwise, it will be cleared. If 'data' is present, the compare Data bit of SEQ CNTL will be set, otherwise, it will be cleared. If both 'Phase' and 'data' are specified, they must be in that order and they must be separated by the keyword AND. ATN is the target role version and required to test whether the initiator has set ATN on the bus. NOT is used for the inverse test of WHEN and IF. "NOT Phase OR data" is the negation of "Phase AND data".

```
INT address

INT address, IF ATN

INT address, IF Phase

INT address, IF data

INT address, IF ATN AND data

INT address, IF Phase AND data

INT address, WHEN Phase

INT address, WHEN data

INT address, WHEN Phase AND data

INT address, IF NOT ATN

INT address, IF NOT Phase

INT address, IF NOT data

INT address, IF NOT ATN OR data

INT address, IF NOT Phase OR data

INT address, WHEN NOT Phase

INT address, WHEN NOT data

INT address, WHEN NOT Phase OR data
```

SCSI I/O COMMANDS - SELECT {ATN} ID Address

Initiator mode function 0. if ATN is present, the "select with ATN" bit is turned on. 'Id' specifies the destination SCSI id.

RESELECT id address

Target mode function 0

WAIT DISCONNECT

Initiator mode function 1

DISCONNECT

Target mode function 1

WAIT RESELECT address

Initiator mode function 2

WAIT SELECT address

Target mode function 2

The following set and clear commands have no meaning in the SCSI target role and should not be used.

SET ACK

Function 3 with the ACK bit set in the Flags field.

SET ATN

Function 3 with the ATN bit set in the Flags field

SET ACK AND ATN

Function 3 with both ACK and ATN bits set in the flag field

CLEAR ACK

Function 4 with the ACK bit set in the Flags field.

CLEAR ATN

Function 4 with the ATN bit set in the Flags field

CLEAR ACK AND ATN

Function 4 with both ACK and ATN bits set in the Flags field

6 SCSI SCRIPTS™ To Support Use of Scatter/Gather

Virtual memory schemes are very common in todays systems, and user data is kept in small, manageable pages in main memory. Memory management units keep track of actual, physical locations. Because user data is scattered through memory and gathered for a write to disk, this situation is termed the scatter/gather requirement. One I/O may include several pages, so current SCSI ports must reinstruct the DMA controller at the beginning of each page of user data. The extra time required to reinstruct for each page always causes some delay for the external processor interrupt and DMA setup time, but a worse side effect is that the delay may cause the disk to slip a revolution because there is no place to put data coming off the media.

Fortunately, the 53C700 addresses this scatter/gather performance degradation in a most efficient way. Each page of user data is represented by a Block Move command. The only overhead required to move to the next page of data is a SCSI SCRIPTS fetch (500 nanoseconds). No firmware interrupt is required (normally a **minimum** of 80 microseconds in a system environment), and no firmware to reinstruct a DMA controller is required.

A SCSI SCRIPTS model for the scatter/gather situation is as follows. First, separate the set of Block Move commands that are required to process the user data and code the SCSI SCRIPTS to call this user data move section. A maximum number of pages per I/O should be determined, and one SCSI SCRIPTS Block Move coded for each possible page. At start I/O time, the logical I/O routine determines exactly how many block moves are required and patches a return command over the next SCSI SCRIPTS command after the last Block Move command required. The group of Block Move commands is called, the correct number of moves is performed, and the return is executed. At the completion of the I/O, the return is overwritten with a Block Move to prepare the set of Block Move commands for the next I/O.

With this simple mechanism, the 53C700 can process scatter/gather requests in a very simple manner and at the same time, dramatically reduce I/O overhead.

7 NCR SCSI SCRIPTS™ FOR AN INITIATOR

```
;       Definition area INITIATOR ROLE

;       Target Device I.D. to be fixed at Start I/O time.
EXTERNAL device

;       Ten byte buffer for sending messages
EXTERNAL sendmsg

;       Ten byte buffer for receiving messages
EXTERNAL rcvmsg

;       Number of message bytes to send after selection
EXTERNAL idcount

;       Number of command bytes
EXTERNAL cmd_count

;       Buffer for the SCSI command
EXTERNAL cmd_adr

;       Number of user data bytes
EXTERNAL data_count

;       Address of user data buffer
EXTERNAL data_adr

;       Error -- not message out after selection
ABSOLUTE err1 = 0x0ff01

;       Error -- unexpected SCSI phase before command phase
ABSOLUTE err2 = 0x0ff02

;       Error -- unexpected SCSI phase after a command transfer
ABSOLUTE err3 = 0x0ff03

;       Error -- not msg in phase after status phase
ABSOLUTE err4 = 0x0ff04

;       No Error -- good I/O
ABSOLUTE ok = 0x0ff00

;       SCSI status returned is check condition
ABSOLUTE check_cond = 0x0fffe

;       SCSI status returned is busy
ABSOLUTE busy = 0x0fffd

;       SCSI status returned is reservation conflict
ABSOLUTE reserved = 0x0fffc
```

```
;       SCSI status returned is unknown
ABSOLUTE bad_status = 0x0fffb

;       Error -- unexpected phase after a data transfer
ABSOLUTE err5 = 0x0ff05

;       Error -- unexpected msg in phase before command phase
ABSOLUTE err6 = 0x0ff06

;       Error -- extended msg present before a command phase
ABSOLUTE err7 = 0x0ff07

;       Error -- save data pointers before a command phase
ABSOLUTE err8 = 0x0ff08

;       Error -- disconnect before command phase
ABSOLUTE err9 = 0x0ff09

;       Error -- save data pointers after the command phase
ABSOLUTE err10 = 0x0ff10

;       Error -- unexpected msg after command phase
ABSOLUTE err11 = 0x0ff11

;       Error -- extended message present after the command phase
ABSOLUTE err12 = 0x0ff12

;       Error -- disconnect after a command phase
ABSOLUTE err13 = 0x0ff13

;       Error -- save data pointers after a data transfer
ABSOLUTE err14 = 0x0ff14

;       Error -- unexpected message after a data transfer
ABSOLUTE err15 = 0x0ff15

;       Error -- extended message after a data transfer
ABSOLUTE err16 = 0x0ff16

;       Error -- disconnect after a data transfer
ABSOLUTE err17 = 0x0ff17

;       Error -- Message in not received after reselection
ABSOLUTE err18 = 0x0ff18

;       Error -- Data in phase after reselection and i.d. msg rcvd
ABSOLUTE err19 = 0x0ff19

;       Error -- Data out phase after reselection and i.d. msg rcvd
ABSOLUTE err20 = 0x0ff20

;       Error -- Msg in phase after reselection and i.d. msg rcvd
ABSOLUTE err21 = 0x0ff21
```

```
;       Error -- Status phase after reselection and i.d. msg rcvd
ABSOLUTE err22 = 0x0ff22

;       Error -- Msg out phase after reselection and i.d. msg rcvd
ABSOLUTE err23 = 0x0ff23

;       Error -- Unknown phase after reselection and i.d. msg rcvd
ABSOLUTE err24 = 0x0ff24

;       Error -- Selected as a target
ABSOLUTE err25 = 0x0ff25

;       Error -- Unexpected message rcvd instead of command complete
ABSOLUTE err26 = 0x0ff26

;       SCSI I/O entry point.  This address must be loaded into the
;       53C700 before initiating a SCSI I/O.
ENTRY start_up
```

```
;       SCRIPTS AREA

;       ************************************
;       This is the entry point for a SCSI I/O
;       ************************************
start_up:

;       This is the SCRIPT for a standard SCSI I/O

;       First, select the device with attention and go to an
;       alternate reselect address.  If a reselection or selection
;       happens before the selection can execute, the chip will
;       change roles if required.
SELECT ATN device resel_adr

;       If the next phase is status, go to end.  Wait for valid
;       phase before performing the comparison.
JUMP end WHEN STATUS

;       If not msg out phase, interrupt.  Do not wait for phase.
INT err1 IF NOT MSG_OUT




;       ************************************************
;       Label for retry loop to resend I.D. msg on error
;       ************************************************
retry:

;       The expected case after selection is I.D. message out to the
;       device.  Move the I.D. message from the send message buffer.
;       Do not wait for a phase change.
MOVE idcount sendmsg when MSG_OUT

;       If the target remains in the message out phase after the
;       initial messages have been sent to the device, retransfer
;       the messages.  Wait for a valid phase (req asserted).
JUMP retry WHEN MSG_OUT

;       Now check for all expected phases.
JUMP end IF STATUS

;       Process a message in before the command phase here
JUMP msg1 IF MSG_IN

;       If it is not status, msg in, or command, stop
;       Interrupt if not command phase
INT err2 IF NOT CMD

;       Transfer command bytes to the host
MOVE cmd_count cmd_adr when CMD
```

```
;       Determine what is coming next.  Is there a message in after
;       the command phase?
JUMP msg2 WHEN MSG_IN

;       Status phase after the command?
JUMP end IF STATUS

;       Check for data in phase
JUMP input_data IF DATA_IN

;       Is this a data out phase?
JUMP output_data IF DATA_OUT

;       Error -- an unexpected phase after a command transfer
INT err3



;       ********************************
;       Label to process the status phase
;       ********************************
end:

;       Move the status byte in to the buffer area
MOVE 1 status_adr when STATUS

;       NOTE: an alternative at this point is to determine what the
;       status byte is and jump to a set of routines that will
;       process the command complete message, physical disconnect,
;       and then interrupt with the appropriate status byte error
;       value.  Here, the algorithm interrupts if good I/O is not
;       the status byte returned by the target.

;       Was there a check condition
INT check_cond IF 0x02

;       Is the device busy
INT busy IF 0x08

;       Is the device reserved
INT reserved IF 0x018

;       Interrupt for unknown state
INT bad_status IF NOT 0x00

;       Status value is good I/O, so process the command complete
;       Stop if the next phase is not message in.
INT err4 WHEN NOT MSG_IN

;       Message in if here.  It should be a command complete.
MOVE 1 rcvmsg when MSG_IN

;       Process the message if it is not a command complete
```

```
INT IF NOT 0x00

;       At this point, instead of interrupting, the best course
;       would be to examine the message received and react, or to
;       interrupt with a more specific error code.

;       Command complete was received, acknowledge it
CLEAR ACK

;       A physical disconnect should be next
WAIT DISCONNECT

;       Good I/O if here
INT ok


;       ****************************************
;       This the data out section of the algorithm
;       ****************************************
output_data:

MOVE data_count data_adr when DATA_OUT

;       If a scatter/gather requirement exists, then this section
;       can be multiple block moves to allow for multiple segments
;       of data.  Also, this section could actually be a jump to a
;       group of block moves that can be patched appropriately at
;       start I/O for the number of segments needed.  The overhead
;       between segment block moves is 500-600 nano seconds.



;       ****************************************
;       Process what comes after the data transfer
;       ****************************************
check_out:

;       Status phase is the normal next step
JUMP end WHEN STATUS

;       Is there a message in phase after data transfer
JUMP msg3 IF MSG_IN

;       Unexpected phase detected after data transfer
INT err5



;       ************************************************
;       This is the data in phase portion of the algorithm
;       ************************************************
input_data:
```

```
;       If a scatter/gather requirement exists, then this section
;       can be multiple block moves to allow for multiple segments
;       of data.  Also, this section could actually be a jump to a
;       group of block moves that can be patched appropriately at
;       start I/O for the number of segments needed.  The overhead
;       between segment block moves is 500-600 nano seconds.

MOVE data_count data_adr when DATA_IN

;       Go check the phase after data in
JUMP check_it




;       ********************************************
;       Process a message in before the command phase
;       ********************************************
msg1:

MOVE 1 rcvmsg when MSG_IN

;       Is this an extended message?
JUMP ext_msg1 IF 0x01

;       Is this save data pointers?  Interrupt with ACK set.
INT err8 IF 0x02

;       Is this a disconnect?
JUMP disc1 IF 0x04

;       Interrupt if any other message with ACK set
INT err6

;       Message is an extended message
ext_msg1:
;       Acknowledge the message just received
CLEAR ACK

;       Move two more messages into the buffer to get the extended
;       message length and opcode for the processor to have
;       available on the interrupt.
MOVE 2 ext_buf when MSG_IN

;       Interrupt the processor
INT err7

;       Message is a disconnect
disc1:
;       Acknowledge the disconnect message
CLEAR ACK

;       Disconnect before the command if here
```

```
WAIT DISCONNECT

;       Interrupt the processor on a disconnect
INT err9



;       **********************************
;       Message in after the command phase
;       **********************************
msg2:
MOVE 1 rcvmsg when MSG_IN

;       Is this an extended message?
JUMP ext_msg2 IF 0x01

;       Is this save data pointers?  Interrupt with ACK set.
INT err10 IF 0x02

;       Is this a disconnect?
JUMP disc2 IF 0x04

;       Interrupt if any other message with ACK set
INT err11

;       Message is an extended message
ext_msg2:
;       Acknowledge the message just received
CLEAR ACK

;       Move two more messages into the buffer to get the extended
;       message length and opcode for the processor to have
;       available on the interrupt.
MOVE 2 ext_buf when MSG_IN

;       interrupt the processor
INT err12

;       Message is a disconnect
disc2:
;       Acknowledge the message
CLEAR ACK

;       Disconnect after the command if here
WAIT DISCONNECT

;       Interrupt the processor on a disconnect
INT err13


*****************************************
;       Message in after the data transfer phase
*****************************************
```

```
msg3:
MOVE 1 rcvmsg when MSG_IN

;       Is this an extended message?
JUMP ext_msg3 IF 0x01

;       Is this save data pointers?  Interrupt with ACK set.
INT err14 IF 0x02

;       Is this a disconnect?
JUMP disc3 IF 0x04

;       Interrupt if any other message with ACK set
INT err15

;       Message is an extended message
ext_msg3:
;       Acknowledge the message just received
CLEAR ACK

;       Move two more messages into the buffer to get the extended
;       message length and opcode for the processor to have
;       available on the interrupt.
MOVE 2 ext_buf when MSG_IN

;       Interrupt the processor
INT err16

;       Message is a disconnect
disc3:
;       Acknowledge the message
CLEAR ACK

;       Disconnect before the data transfer if here
WAIT DISCONNECT

;       Interrupt the processor on a disconnect
INT err17



;       ******************************************
;       This is the section of code to process a reselect or select
;       when a select I/O command was executed
;       ******************************************
;
resel_adr:

;       Wait for reselect as the most probable event
WAIT RESELECT select_adr

;       The initiator was reselected, so process the possibilities
INT err18 WHEN NOT MSG_IN
```

```
;       I.D. message in is the only expected SCSI phase here
MOVE 1 rcvmsg when MSG_IN

;       At this point, if the system integrator knows the possible
;       SCSI device I.D's possible, the algorithm can compare for
;       each known I.D. and react accordingly.  An I/O could even be
;       restarted if the SCSI bus configuration is exactly known.

;       Data in phase after reselection and i.d. transfer
INT err19 WHEN DATA_IN

;       Data out phase after reselection and i.d. transfer
INT err20 IF DATA_OUT

;       Message in phase after reselection and i.d. transfer
INT err21 IF MSG_IN

;       Status phase after reselection and i.d. transfer
INT err22 IF STATUS

;       Message out phase after reselection and i.d. transfer
INT err23 IF MSG_OUT

;       Unknown phase after reselection and i.d. transfer
INT err24


;       ********************************************
;       The chip was in an initiator role, but it has been selected
;       by another device on the SCSI bus.  It is now in the target
;       role.  One could implement the complete SCSI SCRIPTS target
;       algorithm here, or simply interrupt with an error message.
        ********************************************
select_adr:

INT err25
```

```
;       Definition Area TARGET ROLE

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;* The following are variable data values provided  *
;* external to the compiler and resolved at run-time  *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * *

;       Buffer area where the initiator device i.d. is kept
EXTERNAL device

;       Message out buffer area
EXTERNAL msg_buf

;       Command byte buffer area
EXTERNAL cmd_buf

;       Input message buffer
EXTERNAL msg_buf2

;       Buffer address for the initiator i.d.
EXTERNAL initiator

;       Count of user data bytes to be moved
EXTERNAL data_count

;       Address of the user data buffer
EXTERNAL data_addr
;       Target got reselected

;       Address of the status buffer
EXTERNAL stat_adr
```

```
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *
;*  Absolute values are stored in DNAD Register      *
;*  for purposes of interrupt processing             *
;* * * * * * * * * * * * * * * * * * * * * * * * * * * *

ABSOLUTE error1 = 0x0ff01

;       ATN is on after the i.d. message is sent in to the initiator
ABSOLUTE error2 = 0x0ff02

;       ATN is on after the command bytes are sent to the initiator
ABSOLUTE error3 = 0x0ff03

;       Atn is on after the disconnect message is sent to the ;initiator
ABSOLUTE error4 = 0x0ff04

;       ATN on after i.d. message sent to the initiator after a
;       reselect operation is complete
ABSOLUTE error5 = 0x0ff05

;       ATN is on after user data is sent into the initiator
ABSOLUTE error6 = 0x0ff06

;       ATN is on after the status byte is sent
ABSOLUTE error7 = 0x0ff07

;       ATN is on after the command complete message is sent
ABSOLUTE error8 = 0x0ff08

;       Entry Point for the target role
ENTRY start_up

;       Entry point for a target reselect
ENTRY resel_in


;       SCRIPTS AREA

;       ******************************************
;       This is the entry point for a SCSI target I/O
;       ******************************************
start_up:

;       First wait for a selection by the initiator and jump to the
;       alternate address if reselected.
WAIT SELECT resel_adr

;       Move the i.d. message into the message buffer
retry_id:
MOVE 1 msg_buf WITH MSG_OUT
```

```
;       If the initiator sets ATN, go process that condition
JUMP id_atn IF ATN

continue_id:

;       Move the command bytes in to the target buffer
MOVE 1 cmd_buf WITH CMD

;       Note that though a 1 is in the command count field, the chip
;       will automatically transfer in the correct number of bytes
;       based on the SCSI command op code.
;       If the initiator sets ATN, go process that condition
JUMP cmd_atn IF ATN

continue_cmd:

;       In this algorithm, an automatic disconnect is assumed after
;       the SCSI command is received into the buffer.  However, the
;       first byte of the command may be compared against a set of
;       opcode values to determine if this specific command should
;       disconnect or not.

;       Send in the disconnect message
MOVE 1 msg_buf2 WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP disc_atn IF ATN

continue_disc:

;       Now get off the bus
DISCONNECT

;       ****************************************
;       Entry point for reselecting the initiator
;       ****************************************
resel_in:

;       Perform the reselect and jump to resel_adr if a reselection
;       happens while trying to do the reselect
RESELECT initiator resel_adr

;       Move the reselect i.d. message into the initiator
retry_resel:
MOVE 1 msg_buf2 WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP resel_atn IF ATN

continue_resel:

;       Now move the data bytes into the initiator
MOVE data_count data_adr WITH DATA_IN
```

```
;       Note that this could easily be changed to a data out command
;       by patching the phase section of the command, or using a
;       jump command that can be patched to transfer control to a
;       section of code that is either the data out or data in algorithm.
;       If the initiator sets ATN, go process that condition.
JUMP data_atn IF ATN

continue_data:

;       ****************************************
;       If a scatter/gather requirement exists, then this data
;       transfer section can be multiple block moves for the
;       multiple segments of data.  Also, the section could be a
;       jump to a group of block moves that had been patched
;       appropriately at start I/O for the exact number of segments desired.
;       ****************************************************

;       Now move in the status byte
MOVE 1 stat_adr WITH STATUS

;       If the initiator sets ATN, go process that condition
JUMP stat_atn IF ATN

continue_stat:

;       Move the command complete message in
MOVE 1 msg_buf2 WITH MSG_IN

;       If the initiator sets ATN, go process that condition
JUMP cc_atn IF ATN

continue_cc:

;       Now physically disconnect
DISCONNECT

;       ****************************************************
;       If the wait for select or reselect fails, this is the label
;       for the alternate address
;       ****************************************************
resel_adr:

INT error1

;       ********************************************
;       If the initiator turns on ATN after the i.d. message comes
;       out, this is the code for processing what comes next.
;       ********************************************
id_atn:

;       Move the message byte from the initiator out to the message buffer
MOVE 1 msg_buf WITH MSG_OUT

;       At this point, the user may decide to use scripts to program
```

```
;       at a very detailed level or simply interrupt with one user
;       error code.  Scripts may be used to check for:
;               • no-op message -- ignore and jump to continue
;               • initiator detected error -- jump to retry
;               • message parity error -- jump to retry
;               • extended message -- as a minimum, get the opcode and
;                       byte count before interrupting the processor

INT error2


;       All the ATN subroutines have the same basic function

cmd_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error3


disc_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error4


resel_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error5


data_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error6


stat_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error7


cc_atn:
MOVE 1 msg_buf WITH MSG_OUT
INT error8
```

8 Unique Initiator Sequences For the 53C700

I. Disk Drive Initiator Sequence

Arbitrate and Select With Atn
Transfer the I.D. message
Transfer the command bytes
Accept the message in -- DISCONNECT
Reselected -- I.D. message in
Data transfer of 1 - 4 user data blocks
Accept SCSI status byte, COMMAND COMPLETE message and wait for bus free

53C700 strengths in the disk drive environment are:

- A large number of commands are typically issued to the disk, and the 53C700 offers very little SCSI bus overhead and a minimum of time to initiate an I/O in the host computer.

- The 53C700 can continue to the next scheduled SCSI I/O within SCRIPTS with no interrupt to the external processor in the following manner:
  - Compare for Good I/O status byte
  - Interrupt if non-zero
  - Jump to the next scheduled I/O if the status is zero (Good I/O)

- The 53C700 can mask certain disk idiosyncrasies. For example if the disk does a SAVE DATA POINTERS before the first DISCONNECT message after the command bytes are transferred, the 53C700 can be programmed to absorb this message with no interrupt to the external processor.

II. Tape Drive Initiator Sequence

Arbitrate and Select With Atn
Transfer the I.D. message
Transfer the command bytes
Accept the message in -- DISCONNECT
Relelected -- I.D. message in
Data transfer of 16k of user data
Accept the message in -- SAVE DATA POINTERS followed by DISCONNECT.
Reselected -- I.D. message in
Data transfer of 16k of user data
*
*
*
Reselected -- I.D. message in
Data transfer of 16k of user data
Accept SCSI status byte, COMMAND COMPLETE message and wait for bus free

Each of the disconnects (on a 16k boundary) must cause an interrupt to the external processor if there are multiple SCSI devices on the SCSI bus. Also, the reselect must cause an interrupt in the general case. If this were a single device bus, or the system was designed to perform tape only activity on the SCSI bus during backup, then the 53C700 could be programmed specifically for this system. Knowing the tape drive was alone on the bus, the 53C700 could be programmed to:

1) Absorb the SAVE DATA POINTERS.
2) Execute a SCRIPTS command of wait for reselect.
3) Process the SCSI reselect sequence with no interrupts.
4) Initiate the next 16k user data block move.
5) If there is ever a restore pointers, the 53C700 interrupts to allow the external processor to restart the tape I/O.

The 53C700 can allow a systems integrator to design using the SCSI bus in a versatile fashion with no performance impact to I/O throughput.

III. SCSI Character Oriented Device in the Initiator Role

The system designer can dedicate a SCSI port to terminal control. First, a SCSI read command is transferred to the target terminal controller. Coming back to the initiator is a stream of user data typed in at the terminals, plus the inserted control bytes in the stream. A SCRIPT can be written to look at the byte stream coming in and send line control bytes to the processing buffer and data bytes to the data buffer. When certain control bytes are received, the 53C700 can terminate the READ operation and generate a unique interrupt to the external processor.

Writes to the terminal controller can begin automatically when a certain read threshold is reached. The 53C700 can process the READ command cleanup, jump to the WRITE command portion of the SCRIPTS, and automatically start sending data to the terminal controller. Thus the 53C700 can be used in unusual areas to offload any processor and improve performance.

Certainly another area of opportunity for the 53C700 is in the design of SCSI printers, where WRITE is the only operation and control characters play in important role also.

9 Special Scripts™ Situations for the User's Guide

I. A SAVE DATA POINTERS message that can be ignored.

CASE 1 -- Unexpected Phase change in the middle of a data transfer. The Block Move command was constructed for a 4k transfer of user data, and after 2k of data, the chip gets an unexpected phase change. Because there may be data left in the chip on a data out phase, an interrupt is required to:

1) Clean up the chip on Data Out Phase
2) Change the data address and byte count in the active SCSI SCRIPTS
3) Receive the message byte via SCSI SCRIPTS (e.g. load the new entry point for resumption of the message in operation). This routine will receive the message byte, verify that the message byte is a SAVE DATA POINTERS (if not, interrupt the external processor), and jump to the SCSI SCRIPTS entry point that will resume the data transfer previously interrupted.

CASE 2 -- The burst size expected is known ahead of time and is extremely predictable. At systems integration time, this burst size must be set, so that each Block Move command can be made exactly equal to the burst size. The SCSI SCRIPTS logic becomes:

```
        *
        *
Block Move  of burst size.
Call subroutine (after waiting) if the next phase is not a data phase. (The subroutine should
process the SAVE DATA POINTERS message in and return.)
Block Move  of burst size
Call subroutine (after waiting) if the next phase is not a data phase.
        *
        *
```

Using this logic, all phase changes are assumed to come on a Block Move command boundary, so no bytes can be left in the chip when a phase change occurs. Certainly, there is an extra penalty for fetching the call subroutine command (500 nsec per SCSI SCSI SCRIPTS), but a system interrupt (minimum 80 microseconds) will be saved by avoiding the extra interrupt.

CASE 3 -- The burst size is not known. Use the same logic as in case 2, but the Block Move byte count should be equal to the device block size. The assumption is that a phase change will come only on the device's block boundary. There is even more SCSI SCRIPTS fetching overhead, depending on the ratio of device block size to burst size, however, even an extra 10 microseconds is minor when compared to the external processor interrupt time of at least 80 microseconds.

II. A SAVE DATA POINTERS message that must be processed by the initiator.

CASE 1 -- The message comes in during a Block Move command. The two possibilities that exist are that the chip is in the data in phase or the data out phase. If it is in the data in phase, all the bytes in the 53C700 are sent to the DMA core and on into system memory. When no bytes are left in the chip, all execution stops, and an interrupt is generated to the external processor. To save the state of the I/O, update the current SCSI SCRIPTS with memory address and byte count that are in the 53C700. The user should then save a pointer to this current SCSI SCRIPTS in some system I/O structure so the I/O can be easily rescheduled later. The chip's SCSI SCRIPTS pointer value is actually the current SCSI SCRIPTS address plus eight, so the saved value must be the SCSI SCRIPTS pointer value minus eight.

If the phase is data out, the 53C700 is full of data bytes going out to the SCSI bus. Execution stops after the phase change, and then an interrupt is generated to the external processor. At that time, the processor should calculate the number of bytes in the chip, and use this value to add to the chip's byte count, subtract from the chip's memory address pointer, and store these values in the current SCSI SCRIPTS. A pointer to the SCSI SCRIPTS (minus eight) must be saved in some I/O structure for later rescheduling. This saved value is the entry point for a resumption of the data transfer portion of the I/O, depending on the outcome of the phase change.

CASE 2 -- The message comes in on a Block Move command boundary. If no test for data phase was placed between Block Move commands, then the 53C700 will fetch the next command ans start processing it. When the phase change actually occurs, the 53C700 may have data in it, so the processing is exactly the same as CASE 1 above.

However, if a wait and test for data phase command is inserted between each Block Move (burst size is known or the block size is used in each Block Move command), then one interrupt is generated to signal the processor to save a pointer to the next Block Move command. A SCSI SCRIPTS to receive message bytes is executed, and the I/O can be resumed by reloading the saved SCSI SCRIPTS pointer. Also, the message processing SCSI SCRIPTS could have a jump command as its last command. The jump to address would be the entry point of the resume SCSI SCRIPTS pointer so that the interrupted I/O can easily start up again.

10 Multi-Tasking I/O Using SCSI SCRIPTS™

In order to accommodate multi-tasking I/O entirely within SCSI SCRIPTS, some special techniques are required. First, a standard SCSI SCRIPTS algorithm (the I/O descriptor) must be developed for each concurrent I/O that is desired. I/O's can then then be linked together by having the last SCSI SCRIPTS command of each scheduled I/O descriptor be a jump to the next scheduled I/O descriptor. This last command address is effectively a mailbox for communication between the host computer and the 53C700. It allows the external processor to patch the last command to be a jump command if the next I/O descriptor has been scheduled by the logical I/O, or to be an interrupt command if it has not been scheduled. The 53C700 will fetch a complete SCSI SCRIPTS (all 8 bytes), blocking out the processor. The iAPX 286/386 can write 4 bytes, blocking out the 53C700. The patch must be to the four byte opcode to allow a test/set capability. Thus, the second four bytes must be the SCSI SCRIPTS jump address and the interrupt value. All of the SCSI SCRIPTS algorithms are arranged in memory in a linked list, and to schedule an I/O, the host processor must:

- Find the address of the first open I/O descriptor (SCSI SCRIPTS program).

- Update the variable addresses of user and SCSI data within the I/O descriptor.

- Change the last command of the I/O descriptor (currently an interrupt command) to an interrupt with the I/O descriptor I.D. as the last four bytes. This information at interrupt time allows a fast identification of which I/O just completed when no other I/O is scheduled.

- Change the last command of the previous I/O descriptor (currently an interrupt command) to a jump to the beginning of the newly scheduled I/O descriptor.

In this manner, a series of I/O's can be scheduled by the host processor. When an I/O completes, if there is no other scheduled I/O, then the 53C700 will simply interrupt with the I/O descriptor I.D. in the command register, otherwise it will jump to the next I/O that has been scheduled. The processor knows that an I/O has completed when:

- An interrupt occurs for a given I/O because of an error.

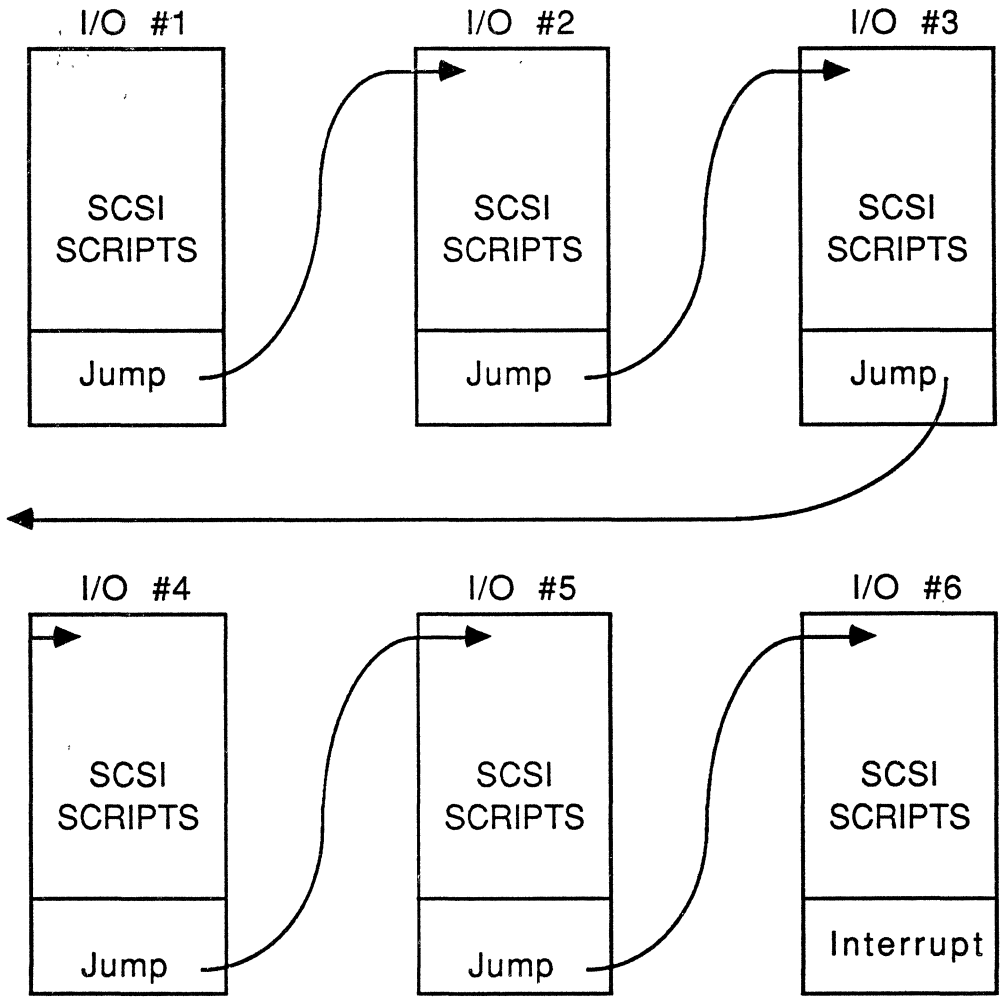
- The address of the current SCSI SCRIPTS command being executed is outside the address space of the I/O descriptor that is being tested for completion. Wrap around must be considered in this test. A timer interrupt or a polling scheme can be used to trigger this test by the host processor.

- The SCSI status byte has been written into a known address to flag that the I/O is complete. A timer interrupt or a polling scheme can be used to trigger this test by the host processor.

- Some type of hardware assist (wherein the SCSI status byte being written into a main memory address causes an interrupt to the host processor) generates an interrupt at the completion of the I/O.

At the completion of an I/O, the last SCSI SCRIPTS command of the I/O descriptor must be changed to an interrupt command to initialize it for the next I/O to be scheduled (the i.d. value is set to an invalid value). The I/O driver must take care that no infinite loop is ever established with the SCSI SCRIPTS jump command. Using this simple software mechanism, the 53C700 can be used to schedule I/O requests without the requirement for a sophisticated host bus adapter.

Using SCSI SCRIPTS to Implement Multi-Threaded I/O



In this example, all six I/O's have been scheduled by patching the last SCSI SCRIPT in each I/O descriptor to jump to the next scheduled I/O descriptor. When each I/O is complete, the linked iist is simply broken by patching out the jump instruction to the next I/O descriptor.

Appendix 1

High Performance Considerations When Using the NCR53C700 vs. the NCR53C90

The purpose of this section is to compare firmware required for the 53C700 and the 53C90 to determine how much of a performance boost the 53C700 can offer at a system level (I/O's per second). One micro second is the time assumed for execution of each external processor instruction.

Sample Input Data Structure
To perform an I/O, the following data structure could be expected at the SCSI H/W driver level.

I.D. message buffer address
Input message buffer address
SCSI command byte count
SCSI command buffer address
User data byte count 1
User data buffer address 1
*
*
User data byte count 'n'
User data buffer address 'n'
SCSI status buffer address
Command Complete message address

Description of the firmware required to initialize the SCSI SCSI SCRIPTS™ for an I/O and the operations to start the I/O.

Refer to the sample initiator SCSI SCSI SCRIPTS for more detail concerning the exact sequence and the values to be updated. At the firmware level, the Initiator SCSI SCSI SCRIPTS must be updated with the address and count for the various pieces of SCSI data and user data required to perform an I/O. In the sample initiator algorithm, there are fifteen values to be updated. Basically, all the BLOCK MOVE commands must be altered. The firmware sequence involved requires:

Load Address Of Data/Count In Main Memory
Load Value Desired From Main Memory
Move Value To SCSI SCRIPTS™ Offset

Assuming about three micro seconds for the above sequence, the total time is **45** micro seconds. To execute the initiator algorithm requires approximately 30 SCSI SCRIPTS fetches and decodes for a total of **15** micro seconds overhead. Assuming a disconnect from the target after the SCSI command has been transferred across, there will be two interrupts to the host processor. Each of these should take about 80 micro seconds for a total of **160** micro seconds. Note that the SCSI portion of the interrupt service routine is only two or three lines of code because both interrupting situations are under control of SCSI SCRIPTS and a read of the interrupt code is all that is required. Combining all of these times:

45 + 15 + 160 = 220 micro seconds overhead.

53C90 Algorithm Description.

The firmware must begin the sequence by preloading the 53C90 FIFO with the SCSI I.D. message followed by a ten byte SCSI command. The firmware sequence involved requires:

Loop: Read Next Byte
Write Next Byte
Go To Loop If Count Not Zero

For eleven bytes, the above sequence would require about 33 micro seconds. Once the SCSI operation begins, the 53C90 requires the following overhead. (Note that each interrupt requires some reads and processing to determine the exact cause of the chip's interrupt.) Assume that an extra 20 micro seconds is required for each interrupt for a total of 100 (80 + 20) microseconds. The following sequence is required to perform a SCSI operation.

Send the SCSI command	033 micro seconds
Interrupt -- msg in phase	100
Interrupt -- msg accepted	100
Interrupt -- physical disc	100
Interrupt -- reselected	100
Initialize DMA Logic	025
Interrupt -- transfer complete	100
Interrupt -- completion seq	100
Interrupt -- msg accepted	100
Interrupt -- physical disc	100
Total time	858 micro seconds

Conclusion:

The 53C700 requires about 25% of the normal firmware overhead associated with a 53C90, in the simplest case. To further compare the chips, note that a SAVE DATA POINTER operation in the 53C90 requires two processor interrupts (200 micro seconds) and only one interrupt using the 53C700 chip (80 -90 micro seconds). Each data segment (in a scatter gather situation) requires 125 micro seconds on the 53C90 (one interrupt plus DMA initialize) but only .5 micro seconds on the 53C700 (500 nano second instruction fetch). So, an I/O that required four data segments in a scatter/gather mode would require 500 micro seconds on the 53C90 and 2 micro seconds on the 53C700 for user data transfer. Combining all this, a four segment data transfer requires:

53C90 1233 micro seconds per I/O

53C700 222 micro seconds per I/O

Translating this improvement into I/O's per second, assume a 4k data transfer size, consisting of four 1k segments in host memory, a target overhead of one millisecond (excluding seek times), and a 4 megabyte per second user data transfer rate on the SCSI bus.

Function	**53C90**	**53C700**
Data Transfer Time	1.00 millisecond	1.00 millisecond
Target overhead	1.00 millisecond	1.00 millisecond
Host Overhead	1.25 milliseconds	0.22 milliseconds
Total times	3.25 milliseconds	2.22 milliseconds

I/O's Per Second (4k transfers/second)	**307**	**450**

In this projected environment, a system can increase its throughput rate by fifty percent simply by using the 53C700 and reducing host computer firmware overhead. Assuming currently available buffered SCSI disk drives, the 53C700 eliminates the host computer firmware as the high performance bottleneck. Note that a 125 microsecond delay between user data segments may cause disk drives to slip a revolution, which implies a dramatic decrease in data throughput. To increase system level performance, designers must patiently eliminate each delay. The 53C700 can remove a large portion of the host overhead associated with each I/O.

Appendix 2

Appendix 2

53C700 System Bus Utilization

The 53C700 has been observed in the laboratory environment transfering 512 bytes of user data at the rate of 6,666 transfers per second (150 micro seconds per I/O). The synchronous SCSI burst rate is set at 5 Mbytes per second. This I/O's per second rate is a limit for the 53C700, because no firmware intervention is required. A real concern is host bus utilization, or "Does the 53C700 affect host computer performance significantly?" This report will discuss how the host bus is used when the SCSI bus is saturated at a block size of 512 bytes.

Host Bus Time To Fetch A SCSI SCRIPTS Command

80 nsec -- Arbitrate and bus settle
80 nsec -- Fetch 4 bytes
80 nsec -- Fetch 4 bytes
40 nsec -- Bus settle time
280 nsec -- Total time

To complete an I/O requires 14 SCSI SCRIPTS.

```
select with ATN
jump error, when not MSG_OUT
move, msg_buf, when MSG_OUT
jump error, when not CMD phase
move, cmd_buf, when CMD
jump error, when not DATA_IN
move, data_buf, when DATA_IN
jump error, when not STATUS
move, status_buf, when STATUS
jump error, when not MSG_IN
move, msg_buf, when MSG_IN
clear ack
wait disconnect
int 0x001
error:
int 0x0ff
```

The time required to execute them with no exception conditions is:

14 X 280 = 3.92 micro seconds

6,666 X 3.92 = 26.13 msec total fetch time per second

Which implies that fetch time is 2.6% of the available system bus time (one second).

Fetching data across the system bus requires:

200 nsec -- I.D. msg fetch ==> 80 (data fetch)
+ 80 (arbitrate)
+ 40 (settle)
360 nsec -- command fetch ==> 240 (three data fetches)
+ 120 (arbitrate + settle)
200 nsec -- Status byte fetch
200 nsec -- COMMAND COMPLETE message
960 nsec -- **Total time per SCSI command**

Total SCSI related data fetch time is:

6,666 X 960 = 6.4 msec

which is 0.64% of the available system time (one second).

Total overhead time is 0.64% + 2.6% = 3.24% of the time available.

Effective user data transfer rate is 3.333Mbytes per seconds plus bus arbitration, etc. comes to 4.0 Mbytes per second. In a 50.0 Mbyte per second system, the user data consumes 8% of the available bandwidth.

So the total time to saturate the SCSI bus takes 11.2% of the iAPX 286/386 system bus available with a blocksize of 512 bytes per SCSI command. Using larger block sizes will lower SCSI command overhead (fewer commands per second), and increase the data transfer rates. For example, a 1k block implies 250 micro seconds per I/O (50 -- SCSI overheadas measured in a lab environment and 200 for user data), which is 4000 I/O's per second or 4 Mbytes per second. The total 386 bus overhead is reduced to about 1.95% of the available time (4000/6666 X 3.24%). As the block size increases, the overhead decreases

NCR MICROELECTRONICS
1635 Aeroplaza Drive
Colorado Springs, CO 80916
(719) 596-5612, (800) 525-2252

NORTH WESTERN SALES OFFICE
Suite 209
3130 De La Cruz Boulevard
Santa Clara, CA 95054-2410
(408) 727-6575

NORTH CENTRAL SALES OFFICE
Suite 4050
South Barrington Office Center
33 West Higgins Road
South Barrington, IL 60010
(312) 426-4600

NORTH EASTERN SALES OFFICE
Suite 4000
500 West Cummings Park
Woburn, MA 01801-6336
(617) 933-0778

SOUTH WESTERN SALES OFFICE
1940 Century Park East
Los Angeles, CA 90067
(213) 556-5231, (213) 556-5395

Suite 255
3300 Irvine Avenue
Newport Beach, CA 92660
(714) 474-7264

SOUTH CENTRAL SALES OFFICE
Suite 100
400 Chisholm Place
Plano, TX 75075
(214) 578-9113

SOUTH EASTERN SALES OFFICE
Suite 250
700 Old Roswell Lake Parkway
Roswell, GA 30076
(404) 587-3136

EUROPEAN SALES OFFICE
NCR GMBH
Gustav-Heinemann-Ring 133
8000 Munchen 83
West Germany
(49) 89-632-202

ASIA/PACIFIC
2501 Vicwood Plaza
199 Des Voeux Road
Central
Hong Kong
852-5-859-6888