Hewlett-Packard

This page provides a running history of changes for a multi-page drawing which cannot conveniently be re-issued completely after each change. When making a change, list for each page all before-and-after numbers (within reason; use judgment, and use "extensive" revision note if loss of past history is tolerable, or retype complete page) and associate with each a symbol made up of the change letter and a serial subscript to appear here and on the page involved (there enclosed in the circle, triangle, or other attention-getting outline).

Letter	Revisions	Date	Initials
A	As Issued	7 June	[illegible]

Model No.:	**Stock No.:** 1LY5-0301, -0302
Title: External Reference Specification (ERS)	
Description: Same as title	**Date:** 4/12/90
By: Don Soltis	**Sheet No.** 0 **of** 142
Supersedes: None	**Drawing No.:** A-1LY5-0302-1

HEWLETT PACKARD

Nikki
Series 300 Workstation I/O ASIC
SCSI, High Speed HPIB
RS-232, Low Speed HPIB
and
Bi-directional Parallel Printer Interface
(Centronics)

External Reference Specification

Don Soltis

Hewlett-Packard Co.
Fort Collins Systems Division
Workstation Systems Lab
4-12-90

HEWLETT PACKARD

HEWLETT PACKARD

CONTENTS

LIST OF FIGURES

LIST OF TABLES

HEWLETT PACKARD

1. General Description

This ASIC contains the circuitry necessary to implement a SCSI (Small Computer Systems Interface) and/or High Speed HPIB (HS-HPIB) disc interface, and Low Speed HPIB (LS-HPIB), RS-232 serial, and bi-directional parallel (Centronics) interfaces. These interfaces are software compatible with the following previous products: 98265A SCSI, 98625B HS-HPIB, 98562 System interface built in HPIB and RS-232 (98644). The bi-directional parallel interface has no predecessor, and is for interfacing to HP LaserJet and ScanJet type peripherals as well as most "Centronics" type printers.

In addition, several enhancements were made to the SCSI, HS-HPIB and RS-232 interfaces; and all of the interfaces utilize software "switches" for configuration.

This ASIC (Nikki) implements the glue and control logic to interface 4 VLSI I/O controller chips to a DIO-1 or DIO-II bus. These are the Fujitsu MB87030 or MB87033 SCSI Protocol Controller (SPC), for SCSI, the Medusa (1LT1-9001) HPIB controller for HS-HPIB, and the Texas Instruments TMS9914 GPIB controller for LS-HPIB. The bi-directional parallel printer interface (Centronics) is implemented almost entirely within Nikki.

Because of pin limitations and perceived product mix, the SCSI interface is mutually exclusive with the HS-HPIB interface. These two interfaces share several pins and an internal DMA data FIFO. Although both interfaces supporting circuitry can be present at one time, Nikki will only configure one or the other, and this configuration can only be changed by asserting a power-up reset signal.

1.1 Features

- SCSI
  - Software compatible with current SCSI interfaces (98265A)
  - Circuitry needed to interface to Fujitsu MB87030 or MB87033 SCSI controller
  - DIO Registers and DMA interface
  - † 32 byte FIFO for SCSI DMA transfers
  - † Byte DMA and packer registers for tape drive support
  - † Ability to switch between 8 and 10 MHz Fujitsu frequency "on-the-fly"
  - † Higher DMA performance than 98265A
- HS-HPIB
  - Software compatible with 98625B High Speed HPIB Disc interface
  - Circuitry needed to interface to Medusa (1LT1-9001) HPIB controller
  - DIO Registers and DMA interface
  - 32 byte FIFO for HPIB DMA transfers
  - † 32-bit HPIB DMA transfers in addition to 8- and 16-bit supported by 98625B
  - † Higher DMA performance than 98625B

- LS-HPIB
  - Software compatible with 98625 System Interface Internal HPIB
  - Circuitry needed to interface to TMS9914 GPIB controller
  - DIO Registers and DMA interface
  - Hardware Parallel Poll circuitry
- RS-232
  - Software compatible with the 98625 System Interface internal RS-232 (98644)
  - Circuitry needed to interface to National 16550 or 8250 UART
  - DIO Registers
  - 2.4576 MHz baud rate generator
  - † 7.2727 MHz baud rate generator for high speed communications
- Centronics
  - LaserJet and ScanJet family compatible
  - 300kbyte per second outbound throughput
  - 700kbyte per second inbound throughput
  - Hardware controlled transfers
  - 32 byte FIFO for either DMA or programmed I/O
  - Byte wide DMA interface
  - Supports mixed DMA and programmed transfers
  - BUSY and NACK or BUSY-only hardware handshake
  - 7 maskable interrupting conditions
  - FIFO disable
- General enhancements
  - Software programmable select code.
  - Software programmable "switch" settings
  - Up to 4 Nikki's in one system
  - Individual module disables (Turn off I/O interface)

† *New features*

1.2 Target Application

Nikki is an ASIC for use in HP9000 S300 IO PC boards to implement SCSI or HPIB disc interfaces, Low Speed HPIB, RS-232, and Centronics I/O interfaces. This chip is used to reduce system cost by reducing PC board area and components required to implement the desired functions. Power needs are also reduced.

Each interface can be independently disabled so the I/O subsection being implemented can be easily customized. One could use a Nikki to implement just a DIO-1 Centronics interface, or 4 Nikki's as part of the core I/O for a server with several disc/tape interfaces, 4 RS-232 interfaces, and 4 Centronics printer/scanner interfaces.

HEWLETT PACKARD

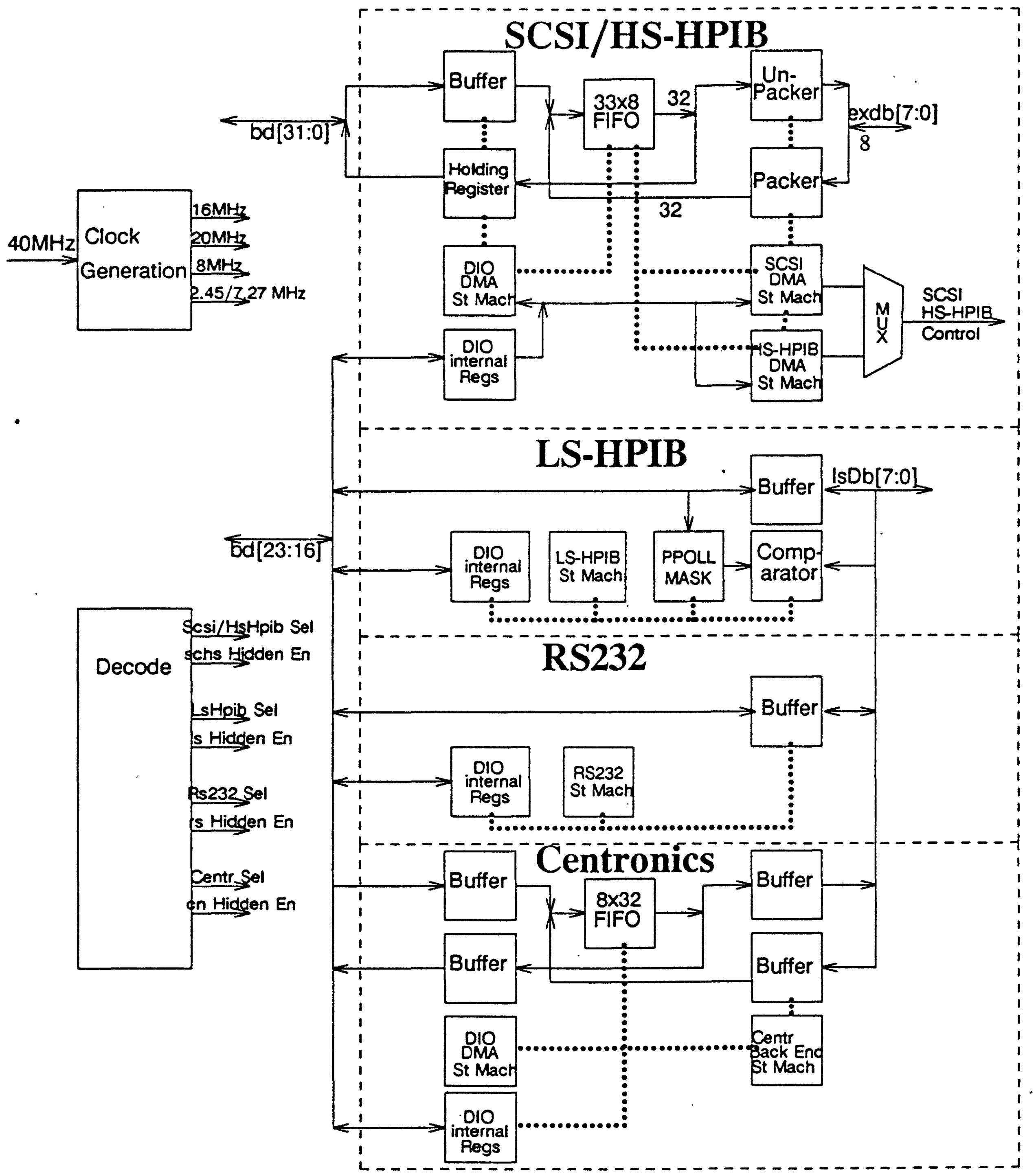

Figure 1. Block Diagram

1.3 External Support Logic Requirements

All pads have n-channel output transistors; however, all but 4 pads have a p-type ESD protection diode. Only the 4 pads, *Ndtack16*, *Ndmrq[1]*, *Ndmrq[0]*, and *Ndmrdy* can be attached directly to a bus where there is the possibility that those bus signals may drive Nikki while power has not yet been applied.

The state machine timing is designed to meet DIO protocol when Nikki is completely buffered from the external bus. The buffers to be used in an actual circuit are assumed to be of the 74ALS or 74LS family.

1.3.1 Interfacing to Fujitsu 87030/33 (SCSI)

Reference Document A-98574-66510-9

1.3.2 Interfacing to Medusa (HS-HPIB)

Reference Document A-98574-66510-9

1.3.3 Interfacing to Texas Instruments TMS9914 (LS-HPIB)

Reference Document A-98574-66510-9

1.3.4 Interfacing to National Semiconductor's NS8250/NS16550 (RS-232)

Reference Document A-98574-66510-9

1.3.5 Interfacing to Centronics

Reference Document A-98574-66510-9

2. Hardware ERS and Theory of Operation

This section lightly discusses the architecture and design of Nikki. Refer to the DCS schematics and PD (PAL Descriptor Language) source files in the Appendix (The schematics are not part of this document).

Nikki is implemented CMOS34 Standard Cell technology. It is packaged in a 160 pin plastic quad flat pack (PQFP). There are 114 DCS (Design Capture System) circuit pages that make up the schematic for the 18000+ gate Nikki design. The design utilizes a scan chain configuration for testing purposes. An automatic test generator (ATG) was run to generate a scan based test for "stuck at" fault coverage.

The design was completed using DCS v3.0 on a 370 running version 6.2 HP-UX. The entire design data base exists on two DC-300 tapes presumably in Manufacturing Specifications.

2.1 *Architecture and Functional Descriptions*

The internal design was broken up into several portable modules to ease possible future use. A clock generation block generates internal the clocks and reset signals used by the rest of the chip. There is a scan control block, and a SCSI/HS-HPIB, a LS-HPIB, RS-232, and Centronics block. There is one module that performs powerup configuration, and module address decode. And several small miscellaneous blocks and gate to combine DIO and shared signals. Each of these is discussed below. Also a hierarchy map is shown to assist in following the schematics if necessary.

2.1.1 Hierarchy map

- NikkiChip: Top level. PADS and Nikki
  - Nikki: Main Schematic.
    - Centron: Centronics interface top level
      - cnBEsm: Centronics back end state machine
        - spareG: Spare gates block
        - cnTimer: Centronics 1 microsecond timer
          - cnTimerEQN: Centronics 1 microsecond logic block
        - cnBEsmEQN: State machine logic block
      - cnDIO: Centronics DIO registers and state machine
        - cnDIOreg: Centronics DIO registers
          - strz8: Separate input tri-state 8 bit buffer
          - sssreg8: Separate ouput, synchronous, scannable 8 bit register
          - SSDC1F: synchronous, scannable STD34DC1F
          - SSDP1F: synchronous, scannable STD34DP1F
          - ltf8: 8 bit latch (STD34LTF)
          - cnInterrupt: Centronics interrupting condition register
            - cnEdgeDetect: Centronics Edge detect
        - cnDIOsm: Centronics DIO register access machine
          - spareG: Spare gates block
          - cnDIOsmEQN: Centronics DIO machine logic block
      - cnDMAsm: Centronics DMA state machine
        - spareG: Spare gates block
        - cnDMAsmEQN: Centronics DMA machine logic block
      - cnFifo: Centronics 32x8 FIFO
        - cnFifoCntr: Centronics FIFO pointer counter
          - cnFifoCntrEQN: Centronics counter machine logic block
        - cnFullEmpty: Centronics FIFO full and emtpy generator
          - cnFullEmptyEQN: Centronics full/empty logic block
            - NAND32F: logic block that implements a 32 input NAND
          - cninvf32: 32 bit inverter (STD34INVF)
        - cnRegz32x8: 32 deep 8 bit wide tri-state register array
          - cnFifoRegz8: 8 bit wide tri-state register
          - cnnspbf8: 8 bit wide buffer (STD34SPBF8)
      - regz8: Tri-state 8 bit register
      - trz8: Tri-state 8 bit buffer
    - Clocks:
      - ResetCtl: Chip wide reset generation
      - clk7MHz: 7.2727MHz clock generator
        - clk7MHzEQN: 7.2727MHz generator logic block
      - clkBgen: 2.4576MHz clock generator
        - GclkBgen: 2.4576MHz generator logic block
      - ClkDiv2: 20MHz clock generator
      - a40to16MHz: 16MHz clock generator
    - Decode: Select code decode and select code registers
      - KeyBdEn: Keyboard NMI register enable
        - compare8: 8 bit comparator
      - SelDecEQN: Select code decode logic block
      - SelectReg: Select code register, hidden enable
        - compare8: 8 bit comparator
        - DecodeEQN: decode logic block
        - SelectEQN: hidden enable logic block
      - compare8: 8 bit comparator
      - iSelectReg: LS-HPIB select code register, hidden enable
        - compare8: 8 bit comparator
        - DecodeEQN: decode logic block
        - SelectEQN: hidden enable logic block
      - puCNsc: Centronics powerup select code generator

        - mux4x6: 4 by 6bit MUX
    - puLSsc: LS-HPIB powerupt select code generator
    - puRSsc: RS-232 powerup select code generator
        - mux4x6: 4 by 6bit MUX
    - puSCHSsc: SCSI/HS-HPIB powerup select code generator
        - mux4x6: 4 by 6bit MUX
    - puSel: Power up configuration block
        - hsdcf: Flip flop (ATG hold type)
        - myDelay: delay gates
    - spareG: Spare gates block
- Keep8: 8 bit bus keeper (STD34KEEP)
- LSHPIB: LS-HPIB top level
    - lsHpibDIOreg: LS-HPIB internal DIO registers
        - SSDC1F: Synchronous, scannable flip flop (STD34DC1F)
        - SSDP1F: Synchronous, scannable flip flop (STD34DP1F)
        - strz8: Separate input tri-state 8 bit buffer
    - lsHpibSM: LS-HPIB machine (REG,DMA,PPOLL)
        - fsdcf: Fast scannable flip-flop (STD34FDC)
        - lsHpibSMeqn: LS-HPIB machine logic block
            - NAND12F: logic block that implements a 12 input NAND
        - myDelay: Delay gates
        - spareG: Spare gates
    - ltz8: 8 bit tri-state latch (STD34LTZF)
    - ppollmatch: Parallel poll match
    - trz8: tri-state buffer (STD34TRZF)
- PadControl: Pad drive enable/disable
- SCSIhpib: SCSI/HS-HPIB top level
    - HPIBdio: HS-HPIB DIO registers/machine
        - HPIBdioReg: HS-HPIB internal DIO registers
            - SSDC1F: Synchronous, scannable flip-flop (STD34DC1F)
            - SSDP1F: Synchronous, scannable flip-flop (STD34DP1F)
            - strz8: separate input tri-state 8 bit buffer (STD34TRZF)
        - HPIBdioSM: HS-HPIB DIO register access machine
            - HPIBdioSMeqn: HS-HPIB DIO register access logic block
            - HPIBrstSM: HS-HPIB 500ns reset machine
                - HPIBrstSMeqn: 500ns reset logic block
            - spareG: Spare gates
        - ltz8: 8 bit latch (STD34LTZF)
        - trz8: 8 bit buffer (STD34TRZF)
    - SCSIdio: SCSI DIO registers/machine
        - ltz8: 8 bit latch (STD34LTZF)
        - SCSIdioReg: SCSI internal DIO registers
            - SSDC1F: Synchronous, scannable flip-flop (STD34DC1F)
            - SSDP1F: Synchronous, scannable flip-flop (STD34DP1F)
            - strz8: Separate input tri-state 8 bit buffer (STD34TRZF)
        - SCSIdioSM: SCSI DIO register access machine
            - SCSIdioSMeqn: SCSI DIO register access logic block
            - myDelay: Delay gates
        - spareG: Spare gates
        - trz8: 8 bit tri-state buffer (STD34TRZF)
    - SCSIparity: Parity generator
    - myDelay: Delay gates
    - nkDMA: SCSI/HS-HPIB FIFO, DMA machines
        - DMAdio: DIO-FIFO DMA machine
            - SSDC1F: Synchronous, scannable flip-flop (STD34DC1F)
            - fsdcf: Fast, scannable flip-flop
            - sDMAdioEQN: DIO-FIFO DMA logic block
            - spareG: Spare gates
        - Fifo: 8 deep 33 bit FIFO
            - fifocntr: FIFO pointer counter

```
            fullempty: FIFO full and empty generator
            reg8x32
                nspbf32: 32 bit buffer (STD34NSPBF)
                reg32: 33 bit tri-state register
        HPIBdma: FIFO-Medusa DMA machine
            HPIBdmaEQN: FIFO-Medusa DMA logic block
                NAND11F: logic block that implements a 11 input NAND gate
                NAND12F: logic block that implements a 12 input NAND gate
            SSDC1F: Synchronous, scannable flip-flop (STD34DC1F)
            myDelay: Delay gates
            spareG: Spare gates
        SCSIdma: FIFO-87033 DMA machine
            SCSIdmaEQN: FIFO-87033 DMA logic block
                NAND11F: logic block that implements NAND11F
            spareG: Spare gates
        regz8: 8 bit tri-state register
        sTimeoutCtr: 32 microsecond counter
            sTimeoutEQN: 32 microsecond counter logic block
        trz8: tri-state 8 bit buffer (STD34TRZF)
SChsMUX: SCSI/HS-HPIB shared signal muxing
    SChsMUX8: SCSI/HS-HPIB 8 by 2 to 1 MUX
ScanControl: Scan Test control
irCombine: Interrupt request combine
nkArbiter: lsDb[7:0] data bus arbiter
    arbiterEQN: lsDb arbiter logic block
    spareG: Spare gates
nkRS232: RS-232 top level logic
    ltz8: 8 bit tri-state latch (STD34LTZF)
    nkStRes: 26 microsecond reset counter
        nkSresEqn: 26 microsecond counter logic block
    rs232SM: RS-232 DIO register access machine
        rs232SMeqn: RS-232 DIO register access logic block
    rs232reg: RS-232 internal DIO registers
        SSDC1F: Synchronous, scannable flip-flop (STD34DC1F)
        SSDP1F: Synchronous, scannable flip-flop (STD34DP1F)
        strz8: Separate input 8 bit tri-state buffer (STD34TRZF)
    spareG: Spare gates
    trz8: 8 bit tri-state buffer (STD34TRZF)
ocCombine:
```

2.1.2 SCSI Hardware ERS

The SCSI interface is broken up into three state machines, a FIFO, the data path circuitry and the internal registers and hidden register sections. One machine controls accesses to the DIO and hidden registers, one performs DMA transfers between the Fujitsu and the FIFO, and the last performs DMA transfers between the DIO bus and the FIFO. One should refer to the schematic, timing diagram, and PD state machine source during the following discussion.

2.1.2.1 DIO to FIFO machine (nkDMA)

The nkDMA machine is common to the SCSI interface and the HS-HPIB interface. This machine transfers data to and from the DIO bus and the FIFO. It does NOT master the DIO bus, but uses the 1TQ4 DMA chip or equivalent to control the transfer and supply main memory addresses. The DMA transfer can be byte-wide (8bits), word-wide (16bits) or longword-wide (32bits). For SCSI only word and longword widths are used. The DMA width specifies the amount of data transferred on each cycle. This machine feeds the

32 bit wide FIFO using only 8 bits if in byte mode, 16 bits if in word mode or the entire 32 bits when in longword mode.

During inbound DMA (writing to main memory) the nkDMA machine is enabled with one of the DE1, DE0 bits in register 3 being set. Once set, it waits for the FIFO to become full or a timer to go off. This machine waits for a FIFO full condition so that a minimum of 8 words/longwords will be transferred for each bus arbitration (request, grant, acknowledge, release) cycle. This increases the available DIO bus bandwidth. However, at the end of a transfer the FIFO may be partially full and will never become full. This is why a timer is used. The timer is enabled when one of the DMA enable bits are set and the machine is currently not requesting a transfer. It cyclically times out at 30-32 microseconds intervals and at worst will effect the transfer speed by less than 10% (8kbytes/4Mbytes/sec = 2ms; 32us/2ms = 1.6%). Since the Fujitsu usually interrupts when it has transferred the last byte into Nikki, this will also cause a timeout.

Once the FIFO is full or a timeout has occurred, the FIFO data is latched into a holding register and *Ndmrq[0 or 1]* (DMA request channel 0 or 1) is asserted. When *Ndmack[0 or 1]* (DMA acknowledge channel 0 or 1) is asserted, the holding register data is driven onto the DIO bus and after some hold time, *Ndmrdy* is asserted indicating that data is valid. If the FIFO is not empty *Ndmrq* is held asserted so that the DMA controller does not relinquish the DIO bus; otherwise, *Ndmrq* is deasserted and the machine waits for FIFO full or timeout and the process repeats. Refer to figure titled *DIO Inbound DMA Cycle Timing*

On outbound DMA (read from main memory) the machine asserts *Ndmrq* whenever the FIFO is empty and continues to assert it until the FIFO will become full for the same bandwidth maximization concerns as before. Once *Ndmrq* has been asserted, both *Ndmack and Ndtkall* becoming asserted indicate valid data on the DIO bus. This data is loaded into the FIFO. If *Ndmrq* was deasserted after *Ndmack* was asserted because the FIFO was almost full, then the machine, after completing the current transfer, will again wait for the FIFO to become empty before starting another cycle. Refer to figure titled *DIO Outbound DMA Cycle Timing*

2.1.2.2 Fujitsu 87033 to FIFO machine (SCSIdma)

The SCSIdma block performs byte wide DMA transfers with the Fujitsu 87033 SCSI controller. During inbound DMA (write to main memory) this state machine loads the data bytes from the Fujitsu into the packer until the DIO DMA width is obtained (this is 4 bytes or 2 bytes for longword or word mode). When the FIFO is not full these bytes are loaded into the FIFO and the cycle repeats. The state machine uses the packer clocks to indicate how full the packer is and where the next byte will go. So, after a *scDmrq* assertion, *scDmrsp* is asserted and when data is valid it is clocked into the proper packer byte. Once 2 or 4 bytes have been accumulated the packer contents are loaded into the FIFO. If the data bytes received from the 87033 are numbered 0, 1, 2, and 3 where 0 is the first byte received, then during longword DMA byte 0 will appear on BD[31:24], byte 1 on BD[23:16], byte 2 on BD[15:8] and byte four on BD[7:0]. During word DMA, byte 0 and 1 appear on the same data lanes, and byte 2 and 3 will be transferred on the next DMA cycle on BD[31:24] and BD[23:16] respectively. Refer to figure titled *SCSI-FIFO Inbound DMA Cycle Timing.*

During outbound DMA (read from main memory) the SCSIdma machine unloads a FIFO entry into the unpacker. And enables one byte at a time out to the Fujitsu controller. The byte that was received from the DIO dma cycle on data lane BD[31:24] is sent first and then the byte from BD[23:16]; if this is a longword transfer then the next two bytes will come from data lane BD[15:8] and BD[7:0] respectively. If the transfer is being performed in word mode, the next two bytes will come from the next entry in the FIFO on data lanes BD[31:24] and BD[23:16] respectively. Refer to figure titled *SCSI-FIFO Outbound DMA Cycle Timing.*

The SCSIdma machine has priority over the DIO register access machine (SCSIdioSM) for the use of the shared exDb data bus. Therefore, it relies on the fact that the SCSIdioSM machine will assert *NIOActive* and then wait for *NDMAactive* to deassert before any DIO accesses are performed. If the SCSIdma machine sees that *NIOactive* is asserted, it will deassert *NDMAactive* between cycles and wait for *NIOactive* to deassert before continuing.

2.1.2.3 SCSI DIO Register Access Machine and Registers (SCSIdioSM)

The SCSIdioSM machine controls accesses between DIO read/writes and internal/external/hidden registers. The internal and hidden registers are FLIP-FLOPS and buffers located in SCSIdioReg. There are three types of external registers. Those in the Fujitsu, the wraparound and loopback test registers, and the byte DMA access "register".

The machine starts an access with a "chip select" from the decode block. It asserts *NIOactive* for a minimum of two states before checking *NDMAactive*. This is how it establishes ownership of the *exDb* data bus. Once *NDMAactive* is deasserted, a cycle is performed.

For an internal register read a buffer enable signal is asserted, the BD[23:16] pad output enables are asserted. This state is held for several clocks to allow for setup times and then *Ndtack16* is asserted. Then the machine waits for the "chip select" to be deasserted (corresponds to *Nbas24*), and deasserts *Ndtack16*. *Ndtack16* is also asynchronously gated with *Nbas24* at the top level. The machine deasserts *NIOactive* and returns to the idle state.

For an internal register write, the *NIOactive/NDMAactive* handshake is performed and then after some data setup time, a write enable signal is asserted. This signal gates a MUX to a synchronous FLIP-FLOP in SCSIdioReg to accept data off of the input DIO data bus. At the same time *Ndtack16* is asserted. The same algorithm as in the internal read access ensues until *Nbas24* deassertion. The only time that data is written to the flip flop is the rising clock edge directly preceding the assertion of *Ndtack16*.

An external register read occurs very similarly to that of an internal register read, except *sread* is asserted which asserts *NspcCs* on a falling edge of the clock, then asserts *NscRead* on the next rising edge to ensure *NspcCs* to *NscRead* setup time. Then waits three states with a latch in the transparent mode feeding *exDb[7:0]* to *BD[23:16]*, and *sread* asserted. Once enough data settling and setup time has passed, *sread* is deasserted and that *exDb[7:0]* data is latched internally. The deassertion of *sread* deasserts *NscRead* on a falling clock edge, and deasserts *NspcCs* on the next rising clock edge. This guarantees *NspcCs* hold time after *NscRead* deassertion. Refer to figure titled *SCSI Read Cycle Timing*.

An external register write uses the same mechanism via *swrite* to generate a *NspcCs* assertion followed by a *NscWrite* assertion one half clock later. *exDb[7:0]* is driven at the start of the cycle. The machine holds for three clocks and releases *swrite* which deasserts *NscWrite* on the falling clock edge followed by *NspcCs* deassertion on the next rising edge. Data (*exDb*) is valid until the end of the cycle (*Nbas24* or *Nlds*) Refer to figure titled *SCSI Write Cycle Timing*.

2.1.3 HS-HPIB Hardware ERS

The HS-HPIB interface is broken up into three state machines, a FIFO, the data path circuitry and the internal registers and hidden register sections. One machine controls accesses to the DIO and hidden registers, one performs DMA transfers between the Medusa and the FIFO, and the last performs DMA transfers between the DIO bus and the FIFO. One should refer to the schematic, timing diagram, and PD state machine source during the following discussion.

2.1.3.1 HS-HPIB DIO to FIFO Machine (nkDMA)

This is shared with SCSI, see above section.

2.1.3.2 Medusa to FIFO State Machine (HPIBdma)

The Medusa to FIFO machine works very similarly to the SCSIdma. The HPIBdma block performs byte wide DMA transfers with the Medusa HPIB controller. During inbound DMA (write to main memory) this state machine loads the data bytes from Medusa into the packer until the DIO DMA width is obtained (this is 1 byte, 2 bytes, or 4 bytes for byte, word, or longword mode). When the FIFO is not full these bytes are loaded into the FIFO and the cycle repeats. The state machine uses the packer clocks to indicate how full the packer is and where the next byte will go. If *hsB[0]* is high during a transfer it indicates that this is the last transfer the Medusa will perform. The machine will set an interrupt (EOI) and, if the packer is not full, the odd bits and flush the packer into the FIFO so that the data will get transferred via DMA.

So, after a *NhsDmrq* assertion, a Medusa register 2 read cycle is performed. When data is valid on the *exDb[7:0]* bus it is clocked into the proper packer byte. Once 1, 2 or 4 bytes have been accumulated the packer contents are loaded into the FIFO. If the data bytes received from the Medusa are numbered 0, 1, 2, and 3 where 0 is the first byte received, then during byte DMA the bytes will appear on BD[23:16]. During word DMA, byte 0 and 1 appear on the BD[31:24] and BD[23:16] respectively, and byte 2 and 3 will be transferred on the next DMA cycle on the same data lanes. During longword DMA byte 0 will appear on BD[31:24], byte 1 on BD[23:16], byte 2 on BD[15:8] and byte four on BD[7:0]. Refer to figure titled *Medusa-FIFO Inbound DMA Cycle Timing.*

During outbound DMA (read from main memory) the HPIBdma machine unloads a FIFO entry into the unpacker. And enables one byte at a time out to the Medusa HPIB controller. The byte that was received from the DIO dma cycle on data lane BD[23:16] is sent during byte DMA. During word DMA the byte on data lane BD[31:24] is sent first and then the byte from BD[23:16]; if this is a longword transfer then the next two bytes will come from data lane BD[15:8] and BD[7:0] respectively. Refer to figure titled *Medusa-FIFO Outbound DMA Cycle Timing.*

The HPIBdma machine has priority over the DIO access machine (HPIBdioSM) for the use of the shared exDb data bus. Therefore, it relies on the fact that the HPIBdioSM machine will assert *NIOActive* and then wait two cycles for *NDMAactive* to deassert before any DIO accesses are performed. If the HPIBdma machine sees that *NIOactive* is asserted, it will deassert *NDMAactive* between cycles and wait for *NIOactive* to deassert before continuing.

2.1.3.3 HS-HPIB Register Access State Machine (HPIBdioSM)

The HPIBdioSM machine controls accesses between DIO read/writes and internal/external/hidden registers. The internal and hidden registers are FLIP-FLOPS and buffers located in HPIBdioReg. The external registers are those in the Medusa HPIB controller.

The machine starts an access with a "chip select" from the decode block. It asserts *NIOactive* for a minimum of two states before checking *NDMAactive*. This is how it establishes ownership of the *exDb* data bus. Once *NDMAactive* is deasserted, a cycle is performed.

For an internal register read a buffer enable signal is asserted, the BD[23:16] pad output enables are asserted. This state is held for several clocks to allow for setup times and then *Ndtack16* is asserted. Then the machine waits for the "chip select" to be deasserted (corresponds to *Nbas24*), and deasserts *Ndtack16*. *Ndtack16* is also asynchronously gated with *Nbas24* at the top level. The machine deasserts NIOactive and returns to the idle state.

For an internal register write, the *NIOactive/NDMAactive* handshake is performed and then after some data setup time, a write enable signal is asserted. This signal gates a MUX to a synchronous FLIP-FLOP in HPIBdioReg to accept data off of the input DIO data bus. At the same time *Ndtack16* is asserted. The same algorithm as in the internal read access ensues until *Nbas24* deassertion. The only time that data is written to the flip flop is the rising clock edge directly preceding the assertion of *Ndtack16*.

A write to register 1 will assert *hsPon* for 500 nanoseconds.

An external register read occurs very similarly to that of an internal register read, except the Medusa control signals are asserted and when data is valid on the *exDb[7:0]* data bus, it is latched into an internal latch and *Ndtack16* is asserted. The Medusa cycle is completed by the falling edge of *Ndtack16*. Data is driven on the *BD[23:16]* bus from the internal latch until the deassertion of the internal "chip select" signal. Refer to figure titled *HS-HPIB Read Cycle Timing.*

To perform an external register write the machine asserts the Medusa control signals and data, and when the Medusa accepts data off of the *exDb* bus, *Ndtack16* is asserted and the Medusa cycle is completed. The machine then waits for the deassertion of the internal "chip select" before returning to the idle state. Refer to figure titled *HS-HPIB Write Cycle Timing.*

2.1.4 LS-HPIB Hardware ERS

The LS-HPIB machine controls DMA cycles; internal, hidden, and external register read and writes; and parallel poll operations. A parallel poll operation is basically an extended TMS9914 register 6 read where the data bits are compared to a mask (internal register 9) and when a match occurs the parallel poll is terminated and an interrupt is set. Since a parallel poll requires the use of the share *lsDb* data bus, a parallel poll is canceled any time Centronics, RS-232 needs the bus or a LS-hpib register access occurs. Once the access is completed the parallel poll resumes until a match occurs or the poll is disabled. One should refer to the schematic, timing diagram, and PD state machine source during the following discussion.

2.1.4.1 LS-HPIB Control State Machine (lsHpibSM)

A DMA cycle is performed when the DMA enable bit is set, the TMS9914 asserts *NlsAccRq*, and *Ndmack[0]* is asserted. If *Nwrite* is asserted, an inbound cycle is performed; otherwise, an outbound cycle occurs. During an inbound cycle, the TMS9914 control signals are asserted according to a DMA access and when the data is valid it is latched internally to Nikki. The TMS9914 cycle is completed and *Ndmrdy* is asserted to indicate that *BD[23:16]* is valid. Then the machine waits for the deassertion of *Ndmack[0]* before returning to the idle state. Refer to figure titled *LS-HPIB Inbound DMA Cycle Timing.*

An outbound DMA cycle is initiated in the same manner as the inbound case. During an outbound DMA cycle an outbound DMA TMS9914 cycle is started and when *Ndtkall* is asserted, indicating valid *BD[23:16]* data, the write is completed and *Ndmrdy* is asserted indicating the data has been accepted. Again, the machine waits for the deassertion of *Ndmack[0]* before returning to the idle state. Refer to figure titled *LS-*

HPIB Outbound DMA Cycle Timing.

An internal register read basically gates data on the *BD[23:16]* data bus allows some data setup time and then asserts *Ndtack16* to indicate valid data. The machine waits for *Nbas24* to deassert before the machine returns to the idle state. During an external register read, a TMS9914 read cycle is performed, and when data is valid it is latched internally. The TMS9914 cycle is completed and *Ndtack16* is asserted. Refer to figure titled *LS-HPIB Read Cycle Timing.*

An internal register write enables a set of synchronous FLIP-FLOPS to accept data off of the *BD[23:16]* bus. *Ndtack16* is then asserted to indicate that data was accepted and the machine waits for the deassertion of *Nbas24* to end the cycle. For an external register write, a TMS9914 write cycle is performed and the data on *BD[23:16]* is redirected to *lsDb[7:0]* and when the TMS9914 cycle is completed, *Ndtack16* is asserted. Refer to figure titled *LS-HPIB Write Cycle Timing.*

When parallel polling is enabled, a TMS9914 read cycle is performed. The read is held mid-cycle and the data on *lsDb[7:0]* is repeatedly sampled and compared to a mask value until a match occurs, or someone else (Centronics, RS-232, LS-HPIB) wants to use the *lsDb[7:0]* data bus. If a match occurs, the machine sets an interrupt flag, completes the TMS9914 read cycle and disables further parallel polls. If access to the *lsDb* bus is requested by another device, the TMS9914 cycle is completed and the *lsDb* bus is given up. When the device releases the resource, the parallel poll operation is once again started. Refer to figure titled *LS-HPIB Parallel Poll Cycle Timing.*

2.1.5 RS-232 Hardware ERS

The RS-232 hardware performs only internal, hidden, and external register reads and writes. One should refer to the schematic, timing diagram, and PD state machine source during the following discussion.

2.1.5.1 RS-232 Register Access State Machine (rs232SM)

An internal register read basically gates data on the *BD[23:16]* data bus allows some data setup time and then asserts *Ndtack16* to indicate valid data. The machine waits for *Nbas24* to deassert before the machine returns to the idle state. During an external register read, a NS16550 (NS8250) read cycle is performed, and when data is valid it is latched internally. The NS16550 (NS8250) cycle is completed and *Ndtack16* is asserted. Refer to figure titled *RS-232 Read Cycle Timing.*

An internal register write enables a set of synchronous FLIP-FLOPS to accept data off of the *BD[23:16]* bus. *Ndtack16* is then asserted to indicate that data was accepted and the machine waits for the deassertion of *Nbas24* to end the cycle. For an external register write, a NS16550 (NS8250) write cycle is performed and the data on *BD[23:16]* is redirected to *lsDb[7:0]* and when the NS16550 (NS8250) cycle is completed, *Ndtack16* is asserted. Refer to figure titled *RS-232 Write Cycle Timing.*

A write to register 1 will assert *rsReset* for 26microseconds.

2.1.6 Centronics Hardware ERS

The Centronics hardware consists of a DIO-DMA machine for transferring data via DMA into the 32 byte deep FIFO; a DIO register access machine that reads and writes to the FIFO, reads Centronics input signal levels, sets Centronics output signals, and reads and writes various internal control registers; and a Back end machine which interfaces the Centronics bus to the FIFO. One should refer to the schematic, timing diagram, and PD state machine source during the following discussion.

2.1.6.1 Centronics DIO to FIFO DMA State Machine (cnDMAsm)

This machine transfers data to and from the DIO bus and the Centronics FIFO. The DMA transfer can only be byte-wide (8bit).

During inbound DMA (writing to main memory) the cnDMAsm machine is enabled with one of the DE1, DE0 bits in register 3 being set. Once set, it waits for the FIFO to become not empty. The FIFO data is then latched into a holding register and *Ndmrq[0 or 1]* is asserted. When *Ndmack[0 or 1]* is asserted, the holding register data is driven onto the DIO bus and after some hold time, *Ndmrdy* is asserted indicating that data is valid. *Ndmrdy* and data are held until *Ndmack* is deasserted and then the process repeats. Refer to figure titled *DIO Inbound DMA Cycle Timing*

On outbound DMA (read from main memory) the machine asserts *Ndmrq* whenever the FIFO is not full Once *Ndmrq* has been asserted, both *Ndmack and Ndtkall* becoming asserted indicate valid data on the DIO bus. This data is loaded into the FIFO. The machine asserts *Ndmrdy* to indicate that the data was taken, and waits for the deassertion of *Ndmack* before returning to the idle state and repeating the cycle. Refer to figure titled *DIO Outbound DMA Cycle Timing*

2.1.6.2 Centronics Back End (FIFO to Centronics) State Machine (cnBEsm)

This machine basically transfers data out of the FIFO to a Centronics peripheral or from the peripheral into the FIFO.

During inbound transfers, *NcnOut* is asserted by setting the direction bit in register 3, *NcnStrobe* becomes an input and *cnBusy* becomes an output. If the FIFO is not empty *cnBusy* is deasserted and the machine waits for *NcnStrobe* to be asserted by the peripheral indicating that data is valid. A bus request is then asserted to gain control over the *lsDb* bus and *cnDclk* is asserted to indicate that the Centronics data bus should be driven on the *lsDb* data bus. This data is clocked into the FIFO and *cnBusy* is asserted to indicate that data was accepted. The machine then waits for *NcnStrobe* to be deasserted indicating the end of the cycle so that *cnBusy* can be deasserted for the next transfer. Refer to figure titled *Centronics Inbound Back End Cycle Timing*

During outbound transfers *NcnStrobe* is an output and *cnBusy* is an input. The machine waits for the FIFO to become not full and *cnBusy* and *cnAck* to not be asserted. It then arbitrates for *lsDb* and drives the *lsDb* data bus with the FIFO data, waits for 50ns and then asserts *cnDclk* which should latch this data into an external latch that drives the Centronics bus. The machine waits for 1 microsecond for data settling time and asserts *NcnStrobe* indicating to the peripheral that data is valid. It then waits for the peripheral to assert *NcnBusy* and that *NcnStrobe* has been asserted for at least 1 microsecond before deasserting *NcnStrobe.* If the I/O modifier (IOM) bit is not set the machine will then wait for *cnAck* to be asserted to indicate the acceptance of that cycle. It then returns to the idle state to repeat the process. If the I/O modifier bit is set then the machine immediately returns to the idle state without waiting for the assertion of *cnAck.* Refer to figure titled *Centronics Outbound Back End Cycle Timing*

2.1.6.3 Centronics Register Access State Machine (rs232SM)

An internal register read basically gates data on the *BD[23:16]* data bus, allows some data setup time and then asserts *Ndtack16* to indicate valid data. The machine waits for *Nbas24* to deassert before the machine returns to the idle state. When reading from register 5 all of the signals are latched so that the data will not change during the DIO read cycle. When the FIFO is read, its data is latched and driven onto the *BD[23:16]* data bus, and the FIFO un-load clock is pulsed. The data for register 7 is stored internal to the part and is not read from the external latch. Refer to figure titled *Centronics Read Cycle Timing.*

An internal register write enables a set of synchronous FLIP-FLOPS to accept data off of the *BD[23:16]* bus. *Ndtack16* is then asserted to indicate that data was accepted and the machine waits for the deassertion of *Nbas24* to end the cycle. When writing to the FIFO, the FIFO's load clock is asserted and *Ndtack16* is asserted regardless of FIFO full or empty status, software beware. To write to register 7, the machine arbitrates for the *lsDb* data bus, drives *lsDb[7:0]*, waits 50ns and asserts *cnCclk* whose rising edge should latch *lsDb[2:0]* to drive the Centronics control signals NINIT (inverted), NSLCTIN (non-inverted), and AUTOFEEDXT (non-inverted) respectively. Refer to figure titled *Centronics Write Cycle Timing.*

2.1.7 Clock generation ERS

The clock generation block generates a 16 or 20MHz clock, a 20MHz, 8MHz, and 2.4576MHz or 7.2727MHz clock. It also generates several reset signals. If *NTest* is asserted, all of the clock outputs are equivalent to *clk40M*.

This block generates 4 reset signals. There are two types: a power-up only reset, and a power-up and chip reset. Both of the reset signals are synchronized and deglitched. *NPuReset* must be valid for at least 150ns to be recognized, but a value of greater than 300ns is recommended. *NReset* must be valid for at least 500ns. The power-up resets are *sNPuReset* and *rNPuReset*. The first is asserted and then held internally for 300ns. The second deasserts within 40ns of the deassertion of *NPuReset* and is used for latching in configuration pin information. The other 2 resets are *sNReset* and *sNReset16*. These are synchronized versions of *NReset*, the first at 20MHz and the second at 16MHz. These are also asserted any time *NPuReset* is asserted.

2.1.8 Scan Chain Testing ERS

For improving static "stuck-at" fault coverage, scan chain test circuitry exists within the ASIC. The improvement comes from internal controllable and observable nodes implemented as scannable FLIP-FLOPS. An automatic test generation program (ATG) will generate a test vector suite for use on the SENTRY testers that will detect as many stuck-at faults as possible. The basic concept behind scan-chain testing is to set or clear controllability points, let those levels propagate through asynchronous logic to observability points. The observability points are then compared to precomputed values and any discrepancies are flagged as errors.

Since it is very difficult, and thus expensive, to probe within the pad ring of an ASIC, scannable FLIP-FLOPS are used to obtain internal controllability and observability points. The pins are used as external controllability and observability points. To set the internal controllability points the scannable FLIP-FLOPS are tied into a serial shift register. Once all of the controllability points are valid, the shift register mode is disabled and the FLIP-FLOPS drive their typical circuit paths. Then, all of the FLIP-FLOPS are clocked. The data held in each FLIP-FLOP is some combinatorial logic function of all of the controllability points. Thus the controllability FLIP-FLOPS serve as observability points also. The data in the FLIP-FLOPS is then shifted out and compared with the expected pattern. There are then three operating modes of Nikki:

normal, serial scan, and parallel scan. Serial scan mode is used to shift data in and out of the scan chain. Parallel scan mode arranges the inputs/outputs of the FLIP-FLOPS to the same as in normal mode, except there is one common clock and a few other modifications. It is in parallel scan mode that the FLIP-FLOPS are clocked and become observability points.

There are 4 pins used for scan testing. There is the ScanIn pin (*BD[31]*), a ScanOut pin (*cnError*), a ScanMask pin (*BD[30]*), and the ScanMode pin (*NcnPE*). These pins are configured for scan testing when *NTest* and *NcnSelect* are asserted (low). The ScanIn pin is the data in of the shift register, the ScanOut is the data out of the shift register, and the ScanMask pin when high forces the ScanOut pin to a high. ScanMask is used to mask "unknowns (X)" in the scan chain; the tester checks for a high, rather than checking for an unknown.

Serial mode is selected when the ScanMode pin is low. When high, parallel mode is selected. During serial mode the *clk40M*, is tied directly to all scan flops, all data buses are inputs and all other I/O pins are inputs. Each scan flop is tied to the next in a 344 element shift register. Internally all tri-state bus drivers are disabled. In parallel mode there are some exceptions, but the circuit is basically the same as it is during normal operation. The main exceptions are that all flops are clocked by *clk40M* directly (no divide by two or anything else), and the *NReset* and *NPuReset* are completely asynchronous.

The test vectors generated by ATG should be available through NID.

2.2 *Pin List and Signal Descriptions*

The pin names follow a convention which identifies active levels, interface group and special operation. Signal names that begin with an upper case N are active low. For easy identification, interface signals (those other than DIO) begin, or follow the N, with two letters depending on the interface: *sc* for SCSI, *hs* for HS-HPIB, *ls* for LS-HPIB, *rs* for RS-232, *cn* for Centronics, and *kb* for keyboard. The name following the two character id should correspond to the signal name on the interface chip; i.e. 87033, TMS9914, Medusa, NS16550, etc. In addition, two names separated by a slash indicate shared SCSI/HS-HPIB pins; and names within parenthesis "()" identify the configuration purpose of the pin during the powerup phase. The pin number starts in the "upper left" corner and is numbered counter clockwise. A pull-up column of "Y" indicates that this pad has its HIGHBIAS (pull-up transistor) enabled. Die side is just that and has no relation to the pin numbering. PLOC is a number which indicates where a pad is on the die. The x axis is numbered from left to right, and the y axis is from bottom to top.

HEWLETT PACKARD

TABLE 1. Pin List

Nikki Pinout						
Pin	Name	Dir	Pull Up	Die Side	PLOC	Pad Type
1	hsSctrl (scE0)	Output(I)	Y	BOTTOM	10	BUSPAD10
2	scDbP/NhsFast	Tri-state/Open Drain	N	BOTTOM	20	BUSPAD10
3	NscLoopOe/hsB[0]	Output/Tri-state	N	BOTTOM	30	BUSPAD10
4	scWrapLc/hsB[1]	Output/Tri-state	N	BOTTOM	40	BUSPAD10
5	Dirty GND			BOTTOM	50	DGNDPAD
6	NscReset/hsPon	Output	N	BOTTOM	60	BUSPAD10
7	NhsIoend	Input	Y	BOTTOM	70	BUSPAD10
8	NhsIntr	Input	Y	BOTTOM	80	BUSPAD10
9	NhsDmrq	Input	Y	BOTTOM	90	BUSPAD10
10	NSCSIsel/hsA[1]	Input/Output	N	BOTTOM	100	BUSPAD10
11	hsA[2] (scE1)	Output (I)	N	BOTTOM	110	BUSPAD10
12	NMedusaCs (hsEn)	Output (I)	Y	BOTTOM	120	BUSPAD10
13	scHin	Output	N	BOTTOM	130	BUSPAD10
14	Dirty VDD			BOTTOM	140	DVDDPAD
15	NscWrite/NhsIogo	Output	N	BOTTOM	150	pad7
16	Dirty GND			BOTTOM	160	DGNDPAD
17	NscRead/hsWrite	Output	N	BOTTOM	170	pad7
18	NspcCs (scEn)	Output (I)	Y	BOTTOM	180	pad7
	Break VDD			BOTTOM	184	BVDDPAD
	Break GND			BOTTOM	187	BGNDPAD
19	scDmrsp/hsA[0]	Output	N	BOTTOM	190	pad7
20	rsIntr	Input	Y	BOTTOM	200	BUSPAD10
21	rsA[0] (nkSel0)	Output (I)	Y	BOTTOM	210	BUSPAD10
22	rsA[1] (nkSel1)	Output (I)	Y	BOTTOM	220	BUSPAD10
23	rsA[2]	Output	Y	BOTTOM	230	BUSPAD10
24	Dirty GND			BOTTOM	240	DGNDPAD
25	rsReset	Output	N	BOTTOM	250	BUSPAD10
26	NrsRcmEn	Output	N	BOTTOM	260	BUSPAD10
27	NrsIoW	Output	N	BOTTOM	270	BUSPAD10
28	NrsIoR	Output	N	BOTTOM	280	BUSPAD10
29	Dirty VDD			BOTTOM	290	DVDDPAD
30	N8250cs (rsEn)	Output (I)	Y	BOTTOM	300	BUSPAD10
31	NcnStrobe	Input/Output	N	BOTTOM	310	BUSPAD10
32	cnDclk	Output	N	BOTTOM	320	BUSPAD10
33	Dirty GND			BOTTOM	330	DGNDPAD
34	cnCclk	Output	N	BOTTOM	340	BUSPAD10
35	cnError/ScanOut	Input/Output	N	BOTTOM	350	BUSPAD10
36	cnAck	Input	N	BOTTOM	360	BUSPAD10
37	cnBusy	Input/Output	N	BOTTOM	370	BUSPAD10
38	Clean GND			BOTTOM	380	CGNDPAD
39	NcnPE	Input	N	BOTTOM	390	BUSPAD10
40	NcnSelect	Input	Y	BOTTOM	400	BUSPAD10

Nikki Pinout

Pin	Name	Dir	Pull Up	Die Side	PLOC	Pad Type
41	NcnOut (cnEn)	Output (I)	Y	RIGHT	10	BUSPAD10
42	oe	Input	Y	RIGHT	20	BUSPAD10
43	NTest	Input	Y	RIGHT	30	BUSPAD10
44	lsDb[0]	Tri-state	N	RIGHT	40	BUSPAD10
45	lsDb[1]	Tri-state	N	RIGHT	50	BUSPAD10
46	Dirty GND			RIGHT	60	DGNDPAD
47	lsDb[2]	Tri-state	N	RIGHT	70	BUSPAD10
48	lsDb[3]	Tri-state	N	RIGHT	80	BUSPAD10
49	lsDb[4]	Tri-state	N	RIGHT	90	BUSPAD10
50	lsDb[5]	Tri-state	N	RIGHT	100	BUSPAD10
51	Dirty VDD			RIGHT	110	DVDDPAD
52	lsDb[6]	Tri-state	N	RIGHT	120	BUSPAD10
53	lsDb[7]	Tri-state	N	RIGHT	130	BUSPAD10
54	NkbNmi	Input	Y	RIGHT	140	BUSPAD10
55	NlsCont	Input	N	RIGHT	150	BUSPAD10
56	NlsAccRq	Input	Y	RIGHT	160	BUSPAD10
57	NlsIntr	Input	Y	RIGHT	170	BUSPAD10
58	N9914cs (lsEn)	Output (I)	Y	RIGHT	180	BUSPAD10
59	Dirty GND			RIGHT	190	DGNDPAD
60	NlsAccGr	Output	N	RIGHT	200	BUSPAD10
61	VDD (core)			RIGHT	210	VDDPAD
62	lsDbin	Output	N	RIGHT	220	BUSPAD10
63	NlsWe	Output	N	RIGHT	230	BUSPAD10
64	lsSctrl	Output	N	RIGHT	240	BUSPAD10
65	ba[1]	Input	N	RIGHT	250	BUSPAD10
66	ba[2]	Input	N	RIGHT	260	BUSPAD10
67	ba[3]	Input	N	RIGHT	270	BUSPAD10
68	ba[4]	Input	N	RIGHT	280	BUSPAD10
69	ba[5]	Input	N	RIGHT	290	BUSPAD10
70	ba[15]	Input	N	RIGHT	300	BUSPAD10
71	ba[16]	Input	N	RIGHT	310	BUSPAD10
72	ba[17]	Input	N	RIGHT	320	BUSPAD10
73	ba[18]	Input	N	RIGHT	330	BUSPAD10
74	ba[19]	Input	N	RIGHT	340	BUSPAD10
75	ba[20]	Input	N	RIGHT	350	BUSPAD10
76	ba[21]	Input	N	RIGHT	360	BUSPAD10
77	ba[22]	Input	N	RIGHT	370	BUSPAD10
78	ba[23]	Input	N	RIGHT	380	BUSPAD10
	Break GND			RIGHT	384	BGNDPAD
	Break VDD			RIGHT	387	BVDDPAD
79	bd[16]	Tri-State	N	RIGHT	390	BUSPAD7
80	bd[17]	Tri-State	N	RIGHT	400	BUSPAD7

Nikki Pinout

Pin	Name	Dir	Pull Up	Die Side	PLOC	Pad Type
81	bd[18]	Tri-State	N	TOP	400	BUSPAD7
82	Dirty GND			TOP	390	DGNDPAD
83	bd[19]	Tri-State	N	TOP	380	BUSPAD7
84	bd[20]	Tri-State	N	TOP	370	BUSPAD7
85	Dirty VDD			TOP	360	DVDDPAD
86	bd[21]	Tri-State	N	TOP	350	BUSPAD7
87	bd[22]	Tri-State	N	TOP	340	BUSPAD7
88	bd[23]	Tri-State	N	TOP	330	BUSPAD7
89	bd[24]	Tri-State	N	TOP	320	BUSPAD7
90	Dirty GND			TOP	310	DGNDPAD
91	bd[25]	Tri-State	N	TOP	300	BUSPAD7
92	bd[26]	Tri-State	N	TOP	290	BUSPAD7
93	bd[27]	Tri-State	N	TOP	280	BUSPAD7
94	bd[28]	Tri-State	N	TOP	270	BUSPAD7
95	bd[29]	Tri-State	N	TOP	260	BUSPAD7
96	bd[30]	Tri-State	N	TOP	250	BUSPAD7
97	Dirty VDD			TOP	240	DVDDPAD
98	bd[31]	Tri-State	N	TOP	230	BUSPAD7
99	Dirty GND			TOP	220	DGNDPAD
100	bd[0]	Tri-State	N	TOP	210	BUSPAD10
101	bd[1]	Tri-State	N	TOP	200	BUSPAD10
102	bd[2]	Tri-State	N	TOP	190	BUSPAD10
103	bd[3]	Tri-State	N	TOP	180	BUSPAD10
104	bd[4]	Tri-State	N	TOP	170	BUSPAD10
105	bd[5]	Tri-State	N	TOP	160	BUSPAD10
106	Dirty GND			TOP	150	DGNDPAD
107	bd[6]	Tri-State	N	TOP	140	BUSPAD10
108	bd[7]	Tri-State	N	TOP	130	BUSPAD10
109	bd[8]	Tri-State	N	TOP	120	BUSPAD10
110	bd[9]	Tri-State	N	TOP	110	BUSPAD10
111	bd[10]	Tri-State	N	TOP	100	BUSPAD10
112	Dirty VDD			TOP	90	DVDDPAD
113	bd[11]	Tri-State	N	TOP	80	BUSPAD10
114	bd[12]	Tri-State	N	TOP	70	BUSPAD10
115	Dirty GND			TOP	60	DGNDPAD
116	bd[13]	Tri-State	N	TOP	50	BUSPAD10
117	bd[14]	Tri-State	N	TOP	40	BUSPAD10
118	Clean VDD			TOP	30	CVDDPAD
119	bd[15]	Tri-State	N	TOP	20	BUSPAD10
	Break VDD			TOP	17	BVDDPAD
	Break GND			TOP	14	BGNDPAD
120	NPureSet	Input	Y	TOP	10	BUSPAD10

HEWLETT PACKARD

Nikki Pinout

Pin	Name	Dir	Pull Up	Die Side	PLOC	Pad Type
121	NReset	Input	Y	LEFT	400	BUSPAD10
122	Ndone	Input	N	LEFT	390	BUSPAD10
123	Ndmack[0]	Input	N	LEFT	380	BUSPAD10
124	Ndmack[1]	Input	N	LEFT	370	BUSPAD10
125	Nlds	Input	N	LEFT	360	BUSPAD10
126	Nbas24	Input	N	LEFT	350	BUSPAD10
127	Ndtkall	Input	N	LEFT	340	BUSPAD10
128	Ndmrdy	Open Drain	N	LEFT	330	BUSPAD7
129	Ndmrq[0]	Open Drain	N	LEFT	320	BUSPAD7
130	Dirty GND			LEFT	310	DGNDPAD
131	Ndmrq[1]	Open Drain	N	LEFT	300	BUSPAD7
132	NdmaActive	Open Drain	N	LEFT	290	BUSPAD10
133	Nwrite	Input	N	LEFT	280	BUSPAD10
134	Dirty VDD			LEFT	270	DVDDPAD
135	Ndtack16	Open Drain	N	LEFT	260	BUSPAD7
136	Nir[3]	Open Drain	Y	LEFT	250	BUSPAD10
137	Nir[4]	Open Drain	Y	LEFT	240	BUSPAD10
138	Nir[5]	Open Drain	Y	LEFT	230	BUSPAD10
139	GND (core)			LEFT	220	GNDPAD
140	Dirty GND			LEFT	210	DGNDPAD
141	Nir[6]	Open Drain	Y	LEFT	200	BUSPAD10
142	Nima	Open Drain	N	LEFT	190	BUSPAD7
143	clk40M	Input	N	LEFT	180	BUSPAD10
	Break GND			LEFT	177	BGNDPAD
	Break VDD			LEFT	174	BVDDPAD
144	clk8M	Output	N	LEFT	170	BUSPAD7
145	Dirty VDD			LEFT	160	DVDDPAD
146	Dirty GND			LEFT	150	DGNDPAD
147	clk24576/7.27	Output	N	LEFT	140	BUSPAD7
	Break GND			LEFT	137	BGNDPAD
	Break VDD			LEFT	134	BVDDPAD
148	scDmrq	Input	Y	LEFT	130	BUSPAD10
149	scIntr	Input	Y	LEFT	120	BUSPAD10
150	scTrmPwr	Input	N	LEFT	110	BUSPAD10
151	exDb[0]	Tri-state	N	LEFT	100	BUSPAD10
152	exDb[1]	Tri-state	N	LEFT	90	BUSPAD10
153	exDb[2]	Tri-state	N	LEFT	80	BUSPAD10
154	exDb[3]	Tri-state	N	LEFT	70	BUSPAD10
155	Dirty GND			LEFT	60	DGNDPAD
156	exDb[4]	Tri-state	N	LEFT	50	BUSPAD10
157	Dirty VDD			LEFT	40	DVDDPAD
158	exDb[5]	Tri-state	N	LEFT	30	BUSPAD10
159	exDb[6]	Tri-state	N	LEFT	20	BUSPAD10
160	exDb[7]	Tri-state	N	LEFT	10	BUSPAD10

2.2.1 *Signal Descriptions*

TABLE 2. DIO signals

Pin Name	Type	Description (DIO Signal Name)
oe	I	Output enable, when low, all outputs are tristated, and the pad pullup transistors are disabled on signals *NTest, hsSctrl,* and *hsA2.*
NTest	I	Test signal, when low chip is in special test mode. When asserted, *NcnSelect* and *NcnPE* select Serial scan mode (both low) or Parallel scan mode (*NcnSelect* low and *NcnPE* high); also, *bd[31]* becomes the scan in pin, *cnError/ScanOut* becomes the scan out pin, and *bd[30]* becomes the scan mask pin. See section on test for further details.
NPuReset	I	Power-up/Hard Reset (PURESET*). When asserted (low) the entire chip is reset, including hidden registers, to its default state and on the rising edge of *NPuReset* the configuration input pins latch their current values.
NReset	I	DIO Bus Reset (RESET*). When asserted most of the chip is reset. The current configuration and hidden register values are preserved. For each module the assertion of *NReset* is analogous to writing to the soft reset register.
Ndtkall	I	Logical OR of all DIO BUS DTACKs. This signal is used during outbound DMA. The DMA machines wait for the assertion of this signal to indicate that valid data exists on the *bd* bus.
Ndtack16	Open Drain	16 bit active low data transfer acknowledge signal (DTACK16*). This signal is asserted when Nikki has either received the data or presented valid data on the *bd[23:16]* bus. This signal can drive an expandable DIO bus directly.
Nwrite	I	Active low write, active high read (BR/W*). When asserted during NON-DMA cycles data is being written into Nikki, else it is being read. When asserted during LS-HPIB DMA cycles, it indicates that memory is being written, Nikki is driving the *bd[23:16]* bus, otherwise main memory is supplying the data, Nikki is receiving data.
Nbas24	I	Active low 24 bit address strobe (BAS24*). When asserted this signal indicates that the address, *ba*, and the control, *Nwrite*, signals are valid. This signal is not used for DMA cycles.
Nlds	I	Active low lower data strobe (BLDS*). When asserted, this signal indicates either that data is valid on the bus or that the bus is ready for Nikki to drive the data bus during DIO register accesses. This signal is not used for DMA cycles.

ba[23:15,5:1]	I	Bus address (BA23..BA1). These lines are decoded to select the various modules/registers within Nikki. The upper 8 bits *ba[23:16]* indicate a particular select code within which exist registers selected by the lower bits.
bd[31:0]	I/O	Bus data (BD15..BD0,XD15..XD0). All non-DMA accesses will use data lane 1 which comprise *bd[23:16]*. Longword DMA uses all lanes, *bd[31:0]*, word DMA uses lanes 0 and 1, *bd[31:16]*, and byte DMA uses lane 1, *bd[23:16]*.
Nir[6:3]	Open Drain	Active low interrupting level output (IR6*, IR5*, IR4*, IR3*). When asserted, some component of Nikki is indicating an interrupt on one of the four levels. Each module's interrupting level can be modified through its first hidden register.
Nima	O	Active low output "I am active" (IMA*). When asserted, some component of Nikki is going to perform a read or write cycle starting when *Nlds* is asserted. This signal will be asserted as long as the address and *Nbas24* are asserted. This signal should be used in conjunction with *Nwrite* to control the data buffer direction between Nikki and the DIO data bus.
Ndmrq[1:0]	Open Drain	Active low DMA request for channel 1 and 0 (DMRQ1*, DMRQ0*). When one of the modules within Nikki wants to perform a DMA cycle, one of these signals will be asserted.
Ndmack[1:0]	I	Active low DMA acknowledge for channel 1 and 0 (DMACK1*, DMACK0*). These signals are used analogously to *Nbas24* but for DMA cycles. When asserted, they indicate that the DMA controller is performing a cycle.
Ndmrdy	Open Drain	Active low DMA ready (DMARDY*). When asserted, Nikki has either presented valid data on, or has accepted data off of the *bd* data bus during a DMA cycle. This signal is analogous to *Ndtack16* but for DMA cycles.
Ndone	I	Active low DMA done (DONE*). This signal is asserted by the DMA controller during the last DMA cycle. It is only used by the HS-HPIB module within Nikki.
NdmaActive	O	Active low "I am DMAing" output. This signal is asserted in response to a *Ndmack* assertion to indicate that some module within Nikki will perform a DMA cycle. It is asserted the entire time *Ndmack* is asserted (less asynchronous delays, of course). It should be used in conjunction with *Nwrite* to control the direction of the data buffers between Nikki and the DIO data bus.

TABLE 3. Clock signals

Pin Name	Type	Description
clk40M	I	Input 40MHz clock. All internal clocks are generated from this input.
clk8M	O	Output 8MHz or 10MHz clock for the MB37030/3. This frequency can be changed between 8MHz and 10MHz at any time through a bit in the SCSI hidden register set.
clk24576	O	Output 2.4567MHz or 7.2727MHz clock for Baud rate generation by the NS16550. This frequency can be changed between 2.4567MHz and 7.2727MHz at any time through a bit in the RS232 hidden register set.

TABLE 4. SCSI support signals

Pin Name	Type	Description (87033 pin name)
scDmrsp	O	SPC (87030/33 SCSI Protocol Controller) DMA Response (DRESP). Active high output to indicate that Nikki is ready to transfer DMA data to/from the SPC (87030/33).
scDmrq	I	SPC DMA Request (DREQ). Active high input when asserted indicates that the SPC is requesting a DMA transfer.
scHin	O	SPC DMA data bus direction signal (HIN). When asserted indicates that DMA transfers will be inbound.
exDb[7:0]	I/O	SPC Data Bus bits 7-0 (D[7:0] and HDB[7:0]).
scDbP	I/O	SPC Data Bus parity (DBP, HDBP).
NspcCs	O(I)	SPC Chip Select Output ($\overline{CS}$). This active low signal is asserted when SPC registers are accessed. If asserted, as an input, during *NPuReset* the SCSI module will be disabled completely.
NscWrite	O	SPC WRITE ($\overline{WT}$). This active low output is asserted during SPC register writes.
NscRead	O	SPC READ ($\overline{RD}$ and $\overline{RDG}$). This active low output is asserted during SPC register reads.
scIntr	I	SPC Interrupt input (INTR). This input is used to propagate SPC interrupts to *Nir[6:3]* if so enabled.
NscReset	O	SPC reset output ($\overline{RESET}$). This active low output is asserted when *Nreset* is asserted or during a write to SCSI register 1.
scWrapLe	O	SCSI Wraparound Register Latch Enable. This active high signal when asserted indicates the *exDb[7:0]* holds data for the wraparound register. The wraparound register drives SCSI control lines NIO, NCD, NMSG, NBSY, no-connect, no-connect, NACK, and NREQ via *exDb[0:7]* inverted respectively.
NscLoopOe	O	SCSI Loopback Output Enable. This active low signal when asserted should enable the SCSI data bus on to *exDb[7:0]*.
scTrmPwr	I	SCSI Terminator Resistor Power. This input should be asserted if the SCSI terminator resistors are powered up.

NSCSIsel	I	SCSI/HS-HPIB selection. If asserted (low) during the deassertion of *NPuReset* then the SCSI module is enabled, assuming *NspcCs* is high (floating) during this period, otherwise HS-HPIB is enabled, again assuming *NMedusaCs* is high (floating).

TABLE 5. High Speed HPIB Support Signals

Note: The following signals indicate only the HS-HPIB name of the signal. In the case of shared SCSI/HS-HPIB pins, these signals indicate the pin usage during HS-HPIB mode and may be redundant with the SCSI section. Use the Nikki *pinout table to cross reference to actual signal name.*

Pin Name	Type	Description (Medusa pin name)
NhsDmrq	I	Medusa DMA Request ($\overline{\text{DMARQ}}$). Active low input when asserted indicates that Medusa is ready for a DMA transfer.
NhsIogo	O	Medusa transfer handshake signal ($\overline{\text{IOGO}}$). Active low output is asserted for all Medusa register accesses including DMA transfers.
NhsIoend	I	Medusa transfer handshake signal $\overline{\text{IOEND}}$). Active low input is asserted by Medusa to signal acceptance of a register read or write.
exDb[7:0]	I/O	Medusa data lines (D[8:15] respectively).
hsB[1:0]	I/O	Medusa D[1:0] 'data' lines. These are used to indicate what type of information *exDb[7:0]* contains.
NMedusaCs	O(I)	Medusa Chip Select ($\overline{\text{CHSEL}}$). Active low output to select Medusa. If asserted, as an input, during *NPuReset*, the HS-HPIB module is completely disabled.
hsA[2:0]	O	Medusa address lines (A[13:15] respectively). During register accesses these lines are identical to *ba[3:1]*. During DMA transfers these lines specify access of the Medusa FIFO register.
hsWrite	O	Medusa write/read signal (W). Active high signal when asserted specifies a write operation, otherwise its a read operation.
NhsIntr	I	Medusa interrupt signal ($\overline{\text{INT}}$). Active low input when asserted indicates an interrupting condition of Medusa.
hsSctrl	O	HPIB System Controller. Active high output when asserted indicates that this HPIB controller is a system controller. This signal level is specified in a HS-HPIB hidden register.
hsPon	O	Medusa Power on (PON). Active high output when not-asserted indicates a reset condition.
NhsFast	open drain	HPIB FAST select. This output should ultimately vary the Medusa timing resistor for tuned or "not-tuned" operation. This signal level is specified in a HS-HPIB hidden register.

TABLE 6. Low Speed HPIB Support Signals

Note: The TMS9914 address lines (RS2-0) are shared with RS-232 on pins rsA[2:0]

Pin Name	Type	Description (TMS9914 pin name)
NlsIntr	I	Active low LS-HPIB interrupt signal from TMS9914 ($\overline{\text{INT}}$).
NlsAccRq	I	TMS9914 support signal, DMA ACCESS REQUEST ($\overline{\text{ACCRQ}}$).
lsSctrl	O	If asserted, LS-HPIB is system controller. This signal should be used to control the direction of the HPIB control signals.
NlsCont	I	TMS9914 support signal, Controller in Charge ($\overline{\text{CONT}}$).
NlsWe	O	TMS9914 support signal, WRITE ENABLE ($\overline{\text{WE}}$).
lsDbin	O	TMS9914 support signal, DATA BUS IN (DBIN).
NlsAccGr	O	TMS9914 support signal, ACCESS GRANTED ($\overline{\text{ACCGR}}$).
lsDb[7:0]	O	Shared LS-HPIB, RS232, and Centronics databus (D7-D0).
NkbNmi	I	Keyboard NMI signal reported in register 5 of LS-HPIB register set.
N9914cs	O(I)	TMS9914 chip enable ($\overline{\text{CE}}$). If asserted (low), as an input, during *NPuReset* the LS-HPIB module is completely disabled except for register 5 if Nikki #1. Register 5 still exists for keyboard support when LS-HPIB is disabled. But, register 5 (along with the rest of LS-HPIB) does not exist if Nikki is configured as #2, #3, or #4.

HEWLETT PACKARD

TABLE 7. Internal RS-232 support signals

Note: The data bus of the NS16550 is shared with LS-HPIB and Centronics on pins lsDb[7:0].

Pin Name	Type	Description (NS16550 pin name)
rsIntr	I	Active high interrupt request from RS-232 controller (INTR).
NrsIoR	O	I/O read signal for RS-232 controller ($\overline{\text{RD}}$).
NrsIoW	O	I/O write signal for RS-232 controller ($\overline{\text{WR}}$).
rsReset	O	RS-232 reset, active high (MR).
rsA[2:0]	O	Register select (address) for RS-232 controller (A2-0), and LS-HPIB. The signal level on rsA[1:0] when *NPuReset* is deasserted selects one of four Nikki configurations.

rsA[1]	rsA[0]	
1	1	Nikki #1
1	0	Nikki #2
0	1	Nikki #3
0	0	Nikki #4

NrsRemEn	O	RS-232 support signal for MODEM HANDSHAKE. When asserted this signal should assert the following NS16550 signals: $\overline{\text{DSR}}$, $\overline{\text{DCD}}$, $\overline{\text{CTS}}$, $\overline{\text{RI}}$
N8250cs	O(I)	NS16550 (NS8250) chip select, active low ($\overline{\text{CS2}}$). If asserted, as an input, during *NPuReset* the RS-232 module will be disabled completely.

TABLE 8. Centronics Support Signals

Pin Name	Type	Description
NcnStrobe	I/O	This is the Centronics NSTROBE control signal. It should be buffered from the Centronics bus with an open collector output gate and a schmidt triggered input gate.
cnAck	I	This is the Centronics NACK (acknowledge) control signal. It should be buffered from the Centronics bus with a schmidt triggered *inverting* input gate.
cnBusy	I/O	This is the Centronics BUSY control signal. It should be buffered from the Centronics bus with an open collector output gate and a schmidt triggered input gate.
NcnPE	I	This is the Centronics PE (paper error) status signal. It should be buffered from the Centronics bus with a schmidt triggered *inverting* input gate.
NcnSelect	I	This is the Centronics SELECT status signal. It should be buffered from the Centronics bus with a schmidt triggered *inverting* input gate.
cnError/ScanOut	I(O)	When *NTest* is not asserted (high) this is the Centronics NERROR status signal. It should be buffered from the Centronics bus with a schmidt triggered *inverting* input gate. When *NTest* is asserted (low) this is the scan chain output pin *ScanOut*.
NcnOut	I	This is the Centronics direction control signal. When asserted (low) **Nikki** is driving *NcnStrobe* and receiving *cnBusy*. The Centronics data bus register (74ALS654, see below) should also be driving the Centronics data bus. When deasserted (high), **Nikki** is driving *cnBusy* and receiving *NcnStrobe*. The Centronics data bus register should be enabled to drive *lsDb[7:0]* from the Centronics data bus when *NcnOut* is deasserted and *cnDclk* is asserted. While *NcnOut* is deasserted, the data register should not drive the Centronics data bus.
cnCclk	I	Centronics control clock. The rising edge of this signal should be used to latch *lsDb[2:0]* which are the repective Centronics control signals: NINIT, NSLCTIN, and WRnRD (AUTOFEEDXT). WRnRD and NSLCTIN should be inverted before driving the Centronics bus. NINIT is the same sense on *lsDb[2]* as it is on the Centronics bus.

cnDclk	I/O	This is the Centronics data control signal. When *NcnOut* is asserted (low) this signal should be used to latch data from the *lsDb[7:0]* data bus into a register (74ALS654) that drives the Centronics data bus DATA[7:0]. The register should latch on the rising edge of *cnDclk* When *NcnOut* is not asserted (high) this signal should be used to enable data buffers onto the *lsDb[7:0]* data bus from the Centronics data bus DATA[7:0]. The buffers should drive when *cnDclk* is asserted (high).

132	signal pins
11	power pins
17	ground pins
160	total pins

TABLE 9. Absolute Maximum Ratings

Vdd	5.5V
Supply current	2000 mA
DC input voltage	Vdd
DC input current	+/- 100 mA
Storage temperature	-40 C to 125 C
Ambient temperature under bias	-20 C to 85 C

Stress beyond listed maximum ratings may cause permanent damage to the device. Exposure to maximum rated conditions for extended periods will adversely affect device reliability.

TABLE 10. Input Protection

Electrostatic Discharge (between any 2 pins)	+/- 2.0KV
DC input current (for latchup protection)	+/- 100mA

TABLE 11. Electrical Characteristics Over Operating Range

Parameter			Condition	MAX	TYP
Vdd	Operating Voltage		5.25 V		4.5 V
	Junction Temperature		85 C		0 C
Idd	Supply Current		200 mA	50 mA	20 mA
Voh	High level output voltage	Ioh max		2.4V	2.4V
Vol	Low level output voltage	Iol max	0.4V		
Vih	High level input voltage				2.0V
Vil	Low level input voltage		0.8V		
Iil/Iih	Input leakage current		10μA		-10μA
Iolk	Tristate output leakage		10μA	-10μA	

2.3 DC Electrical Characteristics

The following DC characteristics are over the range: *4.5V <= Vdd <= 5.25V*. Each pads output current is tested to these specifications in addition to one-half of the indicated value at VDD=4.5V.

Group	Ioh (@Voh)	Iol (@ Vol)
Group 1	1.0ma	-4.0ma
Group 2	2.0ma	-4.0ma
Group 3	3.0ma	-7.4ma
Group 4		-7.4ma
Group 5	2.2ma	-4.5ma

HEWLETT PACKARD

Pin	Name	I/O	DC Current Group
1	hsSctrl	O	1
2	scDbP/NhsFast	Tri-state	1
3	NscLoopOe/hsB[0]	Tri-state	2
4	scWrapLe/hsB[1]	Tri-state	2
5	NscReset/hsPon	Output	1
6	CVDD		
7	NhsIoend	Input	-
8	NhsIntr	Input	-
9	NhsDmrq	Input	-
10	NSCSIsel/hsA[1]	Output	5
11	DGND		
12	hsA[2]	Output	1
13	NmedusaCs	Output(PU)	1
14	scHin/hsSctrl	Output	1
15	DVDD		
16	Nscwrite/NhsIogo	Output	1
17	Nscread/hsWrite	Output	1
18	NspcCs	Output(PU)	1
19	scDmrsp/hsA[0]	1	
20	DGND		
	BVDD		
	BVDD		
21	DVDD		
22	DGND		
23	NrsIntr	Input	-
24	rsA0	Output(PU)	1
25	rsA1	Output(PU)	1
26	rsA2	Output(PU)	1
27	rsReset	Output	1
28	NrsRemEn	Output	1
29	NrsIoW	Output	1
30	DGND		
31	NrsIoR	Output	1
32	N8250cs	Output(PU)	1
33	NcnStrobe	Input/Output	1
34	cnDclk	Output	1
35	cnCclk	Output	1
36	CGND		
37	NcnError	Output	1
38	NcnAck	Input	-
39	cnBusy	Input/Output	1
40	cnPe	Input	-

Pin	Name	I/O	DC Current Group
41	cnSelect	I/O(PU)	1
42	cnOut	I/O(PU)	1
43	oe	Input(PU)	-
44	Ntest	Input(PU)	-
45	lsDb0	Tri-state	1
46	lsDb1	Tri-state	1
47	lsDb2	Tri-state	1
48	lsDb3	Tri-state	1
49	lsDb4	Tri-state	1
50	lsDb5	Tri-state	1
51	lsDb6	Tri-state	1
52	lsDb7	Tri-state	1
53	NkbNmi	Input	-
54	NlsCont	Input	-
55	NlsAccRq	Input	-
56	NlsIntr	Input	-
57	N9914cs	Output(PU)	1
58	NlsAccGr	Output	1
59	DVDD		
60	DGND		
61	VDD	(core)	
62	lsDbin	Output	1
63	NlsWe	Output	1
64	lsSctrl	Output	1
65	ba1	Input	-
66	DGND		
67	ba2	Input	-
68	ba3	Input	-
69	ba4	Input	-
70	ba5	Input	-
71	DVDD		
72	ba15	Input	-
73	ba16	Input	-
74	ba17	Input	-
75	ba18	Input	-
76	ba19	Input	-
77	ba20	Input	-
78	ba21	Input	-
79	ba22	Input	-
80	ba23	Input	-
	BVDD		
	BGND		

HEWLETT PACKARD

Pin	Name	I/O	DC Current Group
81	d16	TriState	3
82	d17	TriState	3
83	d18	TriState	3
84	d19	TriState	3
85	DGND		
86	d20	TriState	3
87	d21	TriState	3
88	d22	TriState	3
89	d23	TriState	3
90	DVDD		
91	d24	TriState	3
92	d25	TriState	3
93	d26	TriState	3
94	d27	TriState	3
95	DGND		
96	d28	TriState	3
97	d29	TriState	3
98	d30	TriState	3
99	d31	TriState	3
100	DGND		
101	d0	TriState	2
102	d1	TriState	2
103	d2	TriState	2
104	d3	TriState	2
105	DGND		
106	d4	TriState	2
107	d5	TriState	2
108	d6	TriState	2
109	d7	TriState	2
110	DVDD		
111	d8	TriState	2
112	d9	TriState	2
113	d10	TriState	2
114	d11	TriState	2
115	DGND		
116	d12	TriState	2
117	d13	TriState	2
118	d14	TriState	2
119	d15	TriState	2
	BVDD		
	BGND		
120	NpuReset	Input(PU)	-

Pin	Name	I/O	DC Current Group
121	Nreset	Input(PU)	-
122	Ndone	Input	-
123	Ndmack0	Input	-
124	Ndmack1	Input	-
125	Nlds	Input	-
126	Nbas24	Input	-
127	DGND		
128	Ndtkall	Input	-
129	Ndmardy	Open Drain	4
130	Ndmrq0	Open Drain	4
131	Ndmrq1	Open Drain	4
132	NdmaActive	Output	1
133	DVDD		
134	Nwrite	Input	-
135	Ndtack16	Open Drain	4
136	Nir3	Open Drain(PU)	1
137	Nir4	Open Drain(PU)	1
138	Nir5	Open Drain(PU)	1
139	Nir6	Open Drain(PU)	1
140	Nima	Open Drain	2
141	GND	(core)	
142	clk40M	Input	-
143	DVDD		
	BGND		
	BVDD		
144	clk8M	Output	1
145	DVDD		
146	DGND		
147	clk24576	1	
	BGND		
	BVDD		
148	scDmrq	Input	-
149	scIntr	Input	-
150	scTrmPwr	Input	-
151	DVDD		
152	exDb[0]	Tri-state	1
153	exDb[1]	Tri-state	1
154	exDb[2]	Tri-state	1
155	exDb[3]	Tri-state	1
156	DGND		
157	exDb[4]	Tri-state	1
158	exDb[5]	Tri-state	1
159	exDb[6]	Tri-state	1
160	exDb[7]	Tri-state	1

2.4 AC Electrical Specifications

2.5 Timing Diagrams

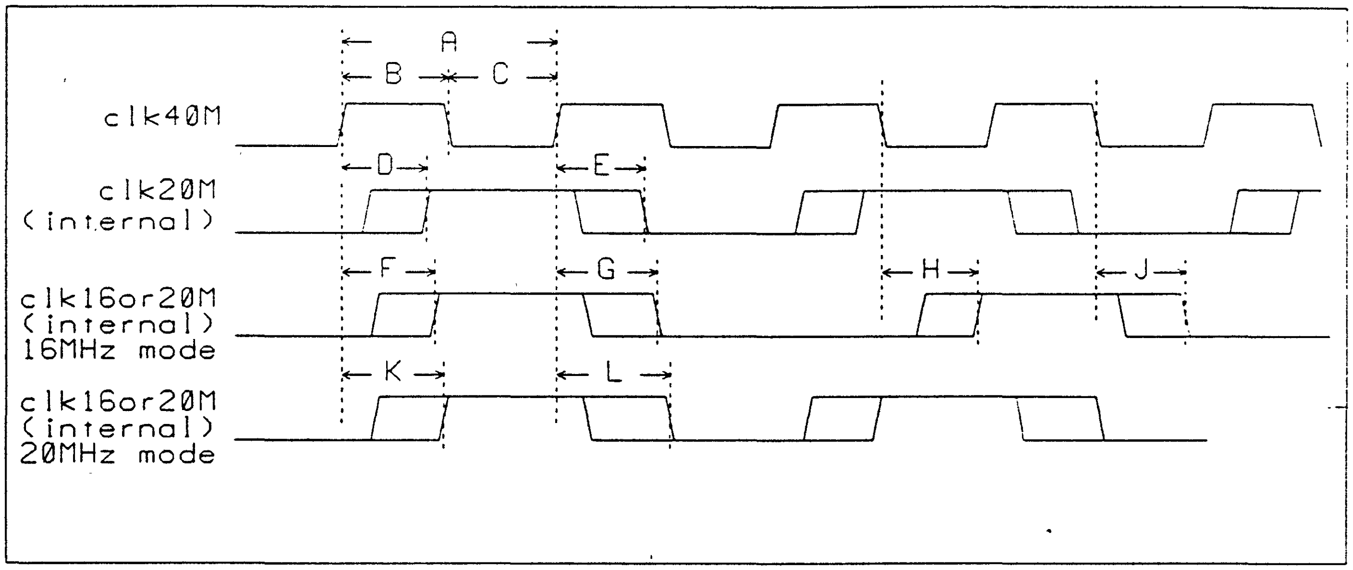
A
B
C
clk40M
D
E
clk20M (internal)
F
G
H
J
clk16or20M (internal) 16MHz mode
K
L
clk16or20M (internal) 20MHz mode

Figure 2. Clock Timing

REF	Parameter	MAX	MIN	NOTE
A	Input clock period	26.25ns	23.75ns	
B	Input clock high time	14.75ns	8.0ns	
C	Input clock low time	14.75ns	8.0ns	
D	Internal clk20M rising delay	9.93ns	2.95ns	
E	Internal clk20M falling delay	9.93ns	2.99ns	
F	Internal clk16or20M rising delay from clk40M rising edge (16MHz mode)	11.37ns	3.73ns	
G	Internal clk16or20M falling delay from clk40M rising edge (16MHz mode)	11.37ns	3.56ns	
H	Internal clk16or20M rising delay from clk40M falling edge (16MHz mode)	9.83ns	3.42ns	
J	Internal clk16or20M falling delay from clk40M falling edge (16MHz mode)	9.84ns	3.27ns	
K	Internal clk16or20M rising delay from clk40M rising edge (20MHz mode)	12.03ns	3.82ns	
L	Internal clk16or20M falling delay from clk40M rising edge (20MHz mode)	11.73ns	3.58ns	

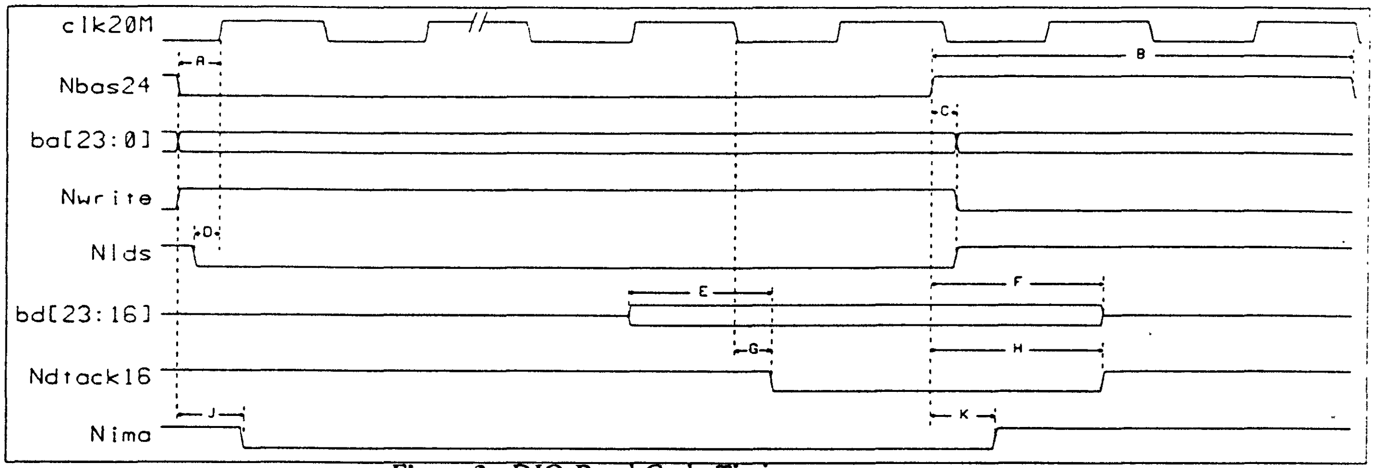

Figure 3. DIO Read Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	Nbas24 setup to clk20M (tester ref)		12.82ns	
A	Nbas24 setup to clk16or20M (tester ref)		16.41ns	
B	Nbas24 high time		125ns	
C	Address and write hold from Nbas24 or Nlds		15ns	
D	Nlds setup (tester ref)		6.4ns	
E	Data setup to Ndtack16		35ns	
F	Data tristate after Nbas deassertion	20ns	0ns	
G	Clock falling to Ndtack16 falling	13ns	0ns	
H	Ndtack16 release after Nbas deassertion	15ns	0ns	
J	Nima assertion after Nbas assertion	17ns	0ns	
K	Nima deassertion after Nbas deassertion	22ns	0ns	

HEWLETT PACKARD

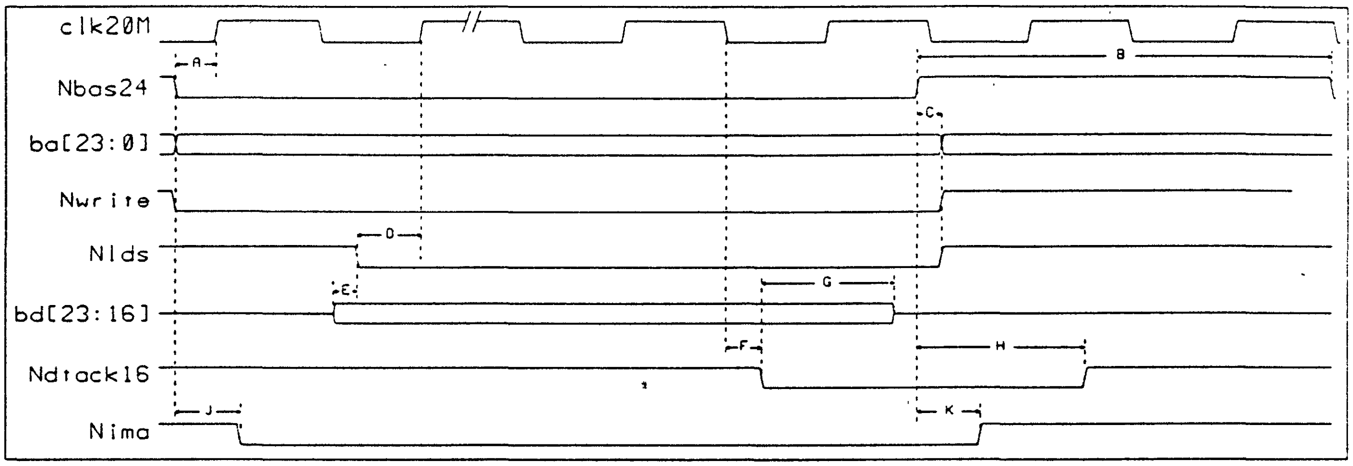

Figure 4. DIO Write Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	Nbas24 setup to clk20M (tester ref)		12.8ns	
A	Nbas24 setup to clk16or20M (tester ref)		16.4ns	
B	Nbas24 high time		125ns	
C	Address and write hold from Nbas24 or Nlds		15ns	
D	Nlds setup (tester ref)		6.4ns	
E	Data setup to Nlds		0ns	
F	Clock falling to Ndtack16 falling	13ns	0ns	
G	Data hold from Ndtack16 falling		50ns	
H	Ndtack16 release after Nbas deassertion	15ns	0ns	
J	Nima assertion after Nbas assertion	17ns	0ns	
K	Nima deassertion after Nbas deassertion	22ns	0ns	

HEWLETT PACKARD

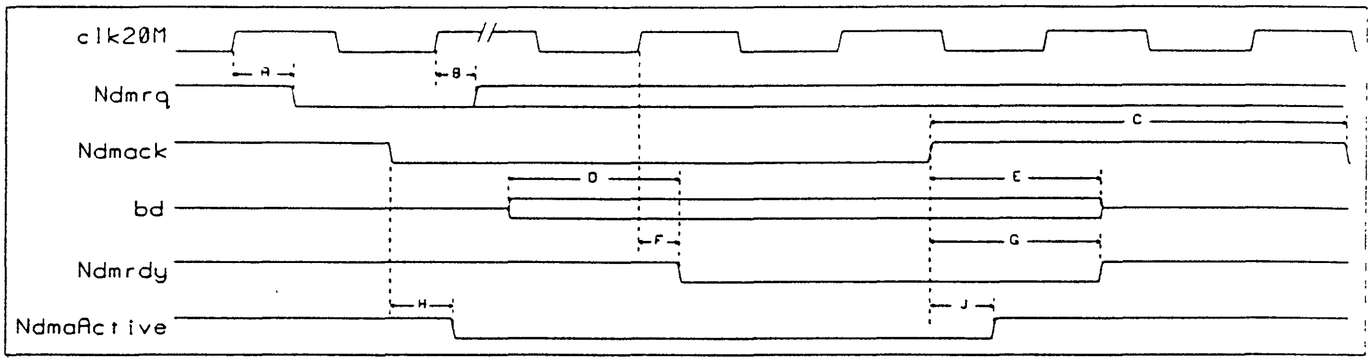

Figure 5. DIO Inbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	Clock to Ndmrq falling (tester ref)	21ns		
B	Clock to Ndmrq rising (tester ref)	21ns		
C	Ndmack high time		100ns	
D	Data set-up to Ndmrdy assertion		75ns	
E	Data tristate after Ndmack deassertion	40ns		
F	Clock falling to Ndmrdy falling (tester ref)	16ns		
G	Ndmrdy release after Ndmack deassertion	12ns		
H	NdmaActive falling from Ndmack assertion	17ns		
J	NdmaActive rising from Ndmack deassertion	15ns		

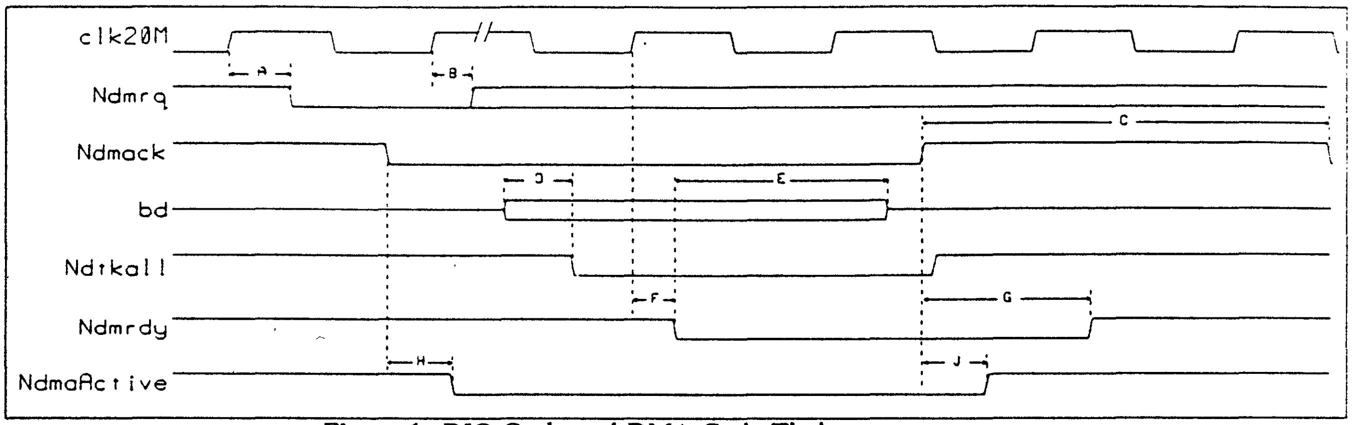

Figure 6. DIO Outbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	Clock to Ndmrq falling (tester ref)	21ns		
B	Clock to Ndmrq rising (tester ref)	21ns		
C	Ndmack high time		100ns	
D	Data set-up to Ndtkall assertion longword mode		-75ns	
D	Data set-up to Ndtkall assertion word/byte mode		0ns	
E	Data hold after Ndmrdy assertion		85ns	
F	Clock falling to Ndmrdy falling (tester ref)	16ns		
G	Ndmrdy release after Ndmack deassertion	12ns		
H	NdmaActive falling from Ndmack assertion	17ns		
J	NdmaActive rising from Ndmack deassertion	15ns		

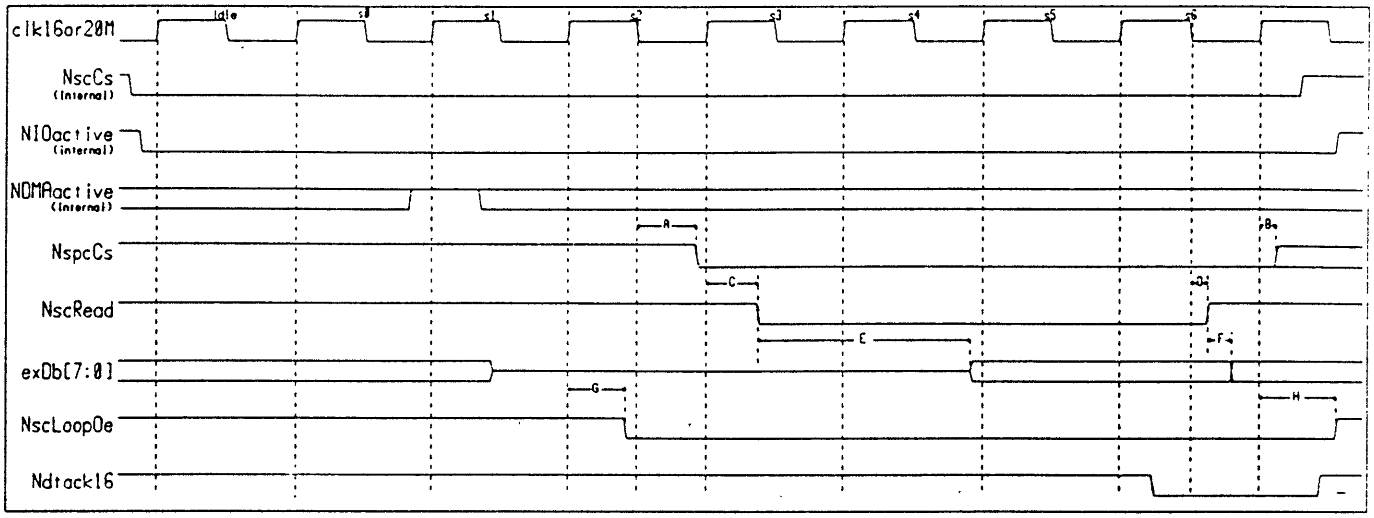

Figure 7. SCSI Read Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	Clock falling to NspcCs falling (tester ref)	19ns		
B	Clock rising to NspcCs rising (tester ref)	19ns		
C	Clock rising to NscRead falling (tester ref)	19ns		
D	Clock falling to NscRead rising (tester ref)	17ns		
E	NscRead to exDb data valid	70ns		
F	exDb data hold from NscRead rising		0ns	
G	Clock to NscLoopOe falling (tester ref)	25ns		
H	Clock to NscLoopOe rising (tester ref)	25ns		

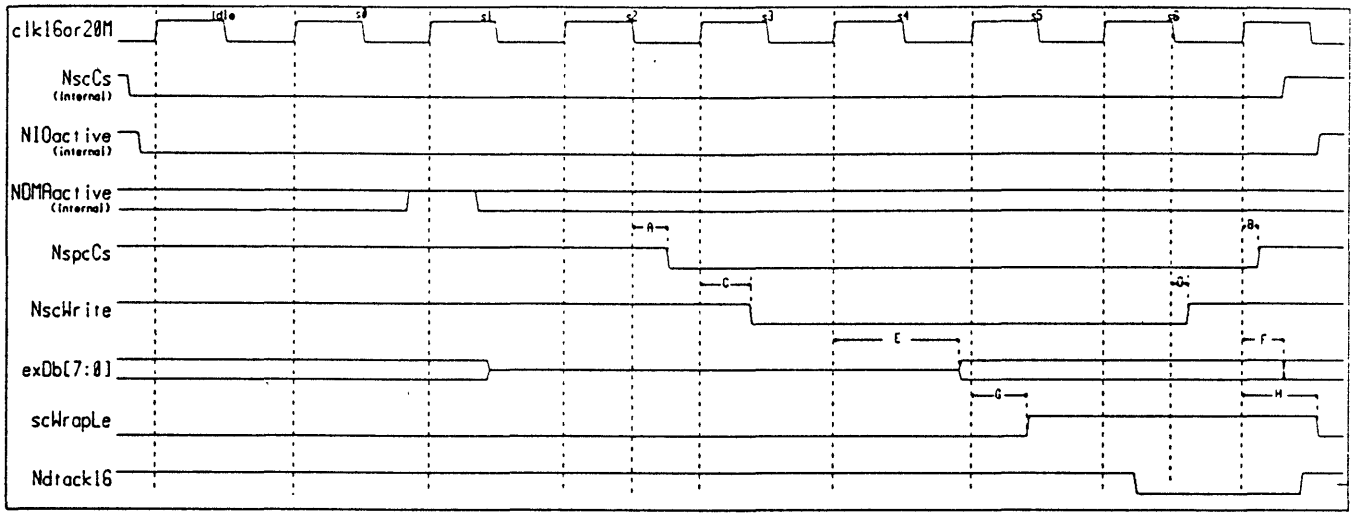

Figure 8. SCSI Write Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	Clock falling to NspcCs falling (tester ref)	19ns		
B	Clock rising to NspcCs rising (tester ref)	19ns		
C	Clock rising to NscWrite falling (tester ref)	19ns		
D	Clock falling to NscWrite rising (tester ref)	17ns		
E	Clock to exDb data valid (tester ref)	0ns		
F	Clock to exDb data change (tester ref)	50ns		
G	Clock to scWrapLe rising (tester ref)	22ns		
H	Clock to scWrapLe falling (tester ref)	22ns		

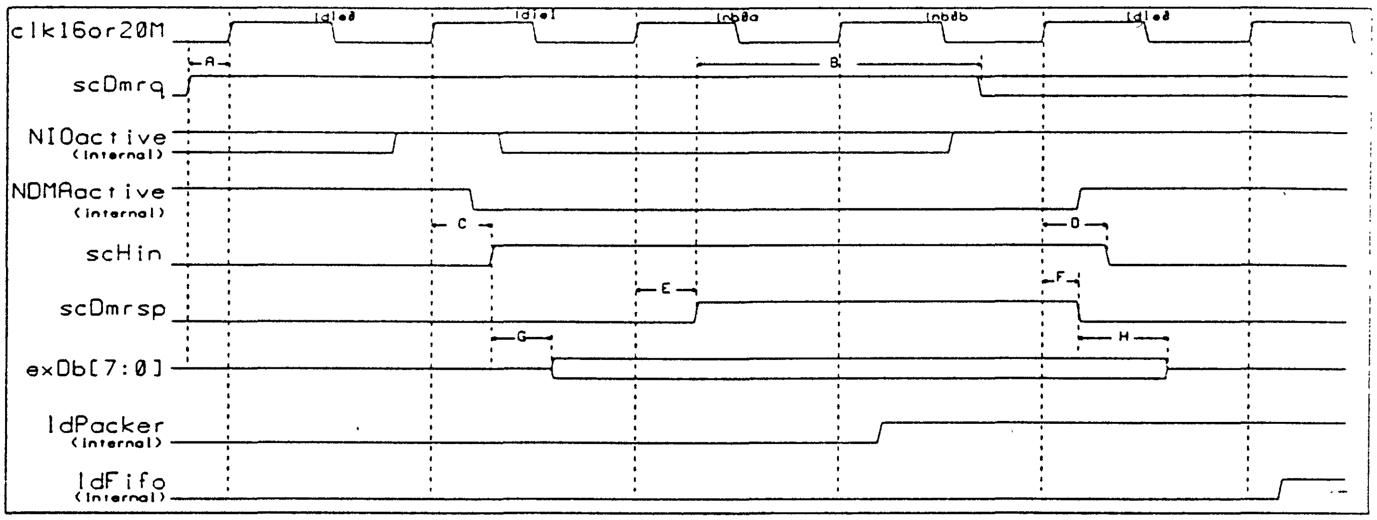

Figure 9. SCSI-FIFO Inbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	scDmrq set-up to clock (tester ref)		9ns	
B	scDmrsp assertion to deassertion scDmrq	70ns	0ns	
C	Clock to scHin rising (tester ref)	25ns	0ns	
D	Clock to scHin falling (tester ref)	22ns	0ns	
E	Clock to scDmrsp rising (tester ref)	20ns		
F	Clock to scDmrsp falling (tester ref)	20ns		
G	scHin rising to data valid	40ns		
H	scDmrsp falling to data change	90ns	10ns	

HEWLETT PACKARD

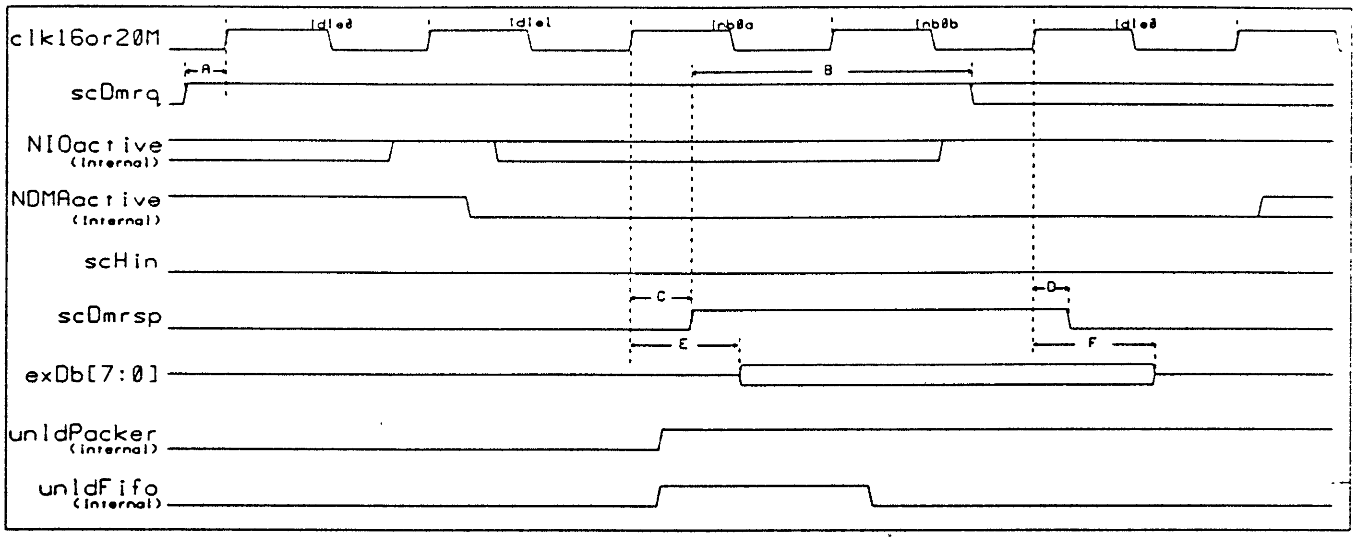

Figure 10. SCSI-FIFO Outbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	scDmrq set-up to clock (tester ref)		9ns	
B	scDmrsp assertion to deassertion scDmrq	70ns	0ns	
C	Clock to scDmrsp rising (tester ref)	20ns		
D	Clock to scDmrsp falling (tester ref)	20ns		
E	Clock to data valid (tester ref)	30ns		
F	Clock to data tristate (tester ref)	27ns		

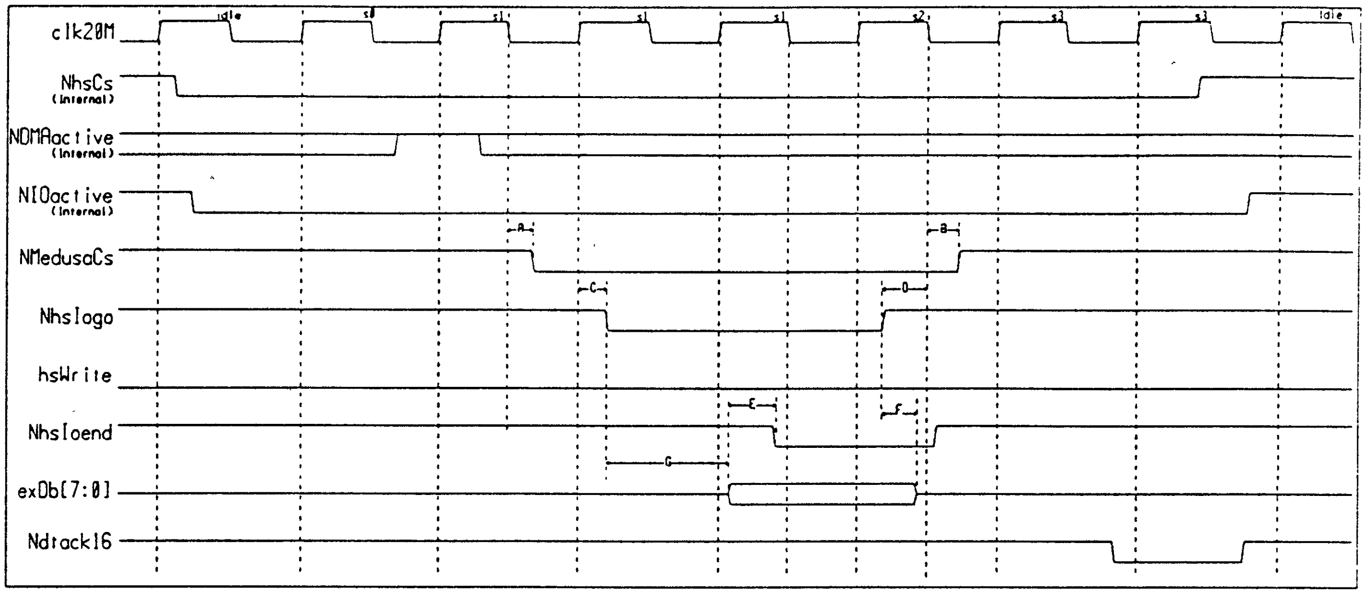

Figure 11. HS-HPIB Read Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M falling to NMedusaCs falling	17ns		
B	clk20M falling to NMedusaCs rising	15ns		
C	clk20M rising to NhsIogo falling	28ns		
D	clk20M rising to NhsIogo rising	27ns		
E	Data setup to NhsIoend falling		-30ns	
F	Data hold after NhsIogo rising		0ns	
G	Data valid from NhsIogo falling	140ns		

HEWLETT PACKARD

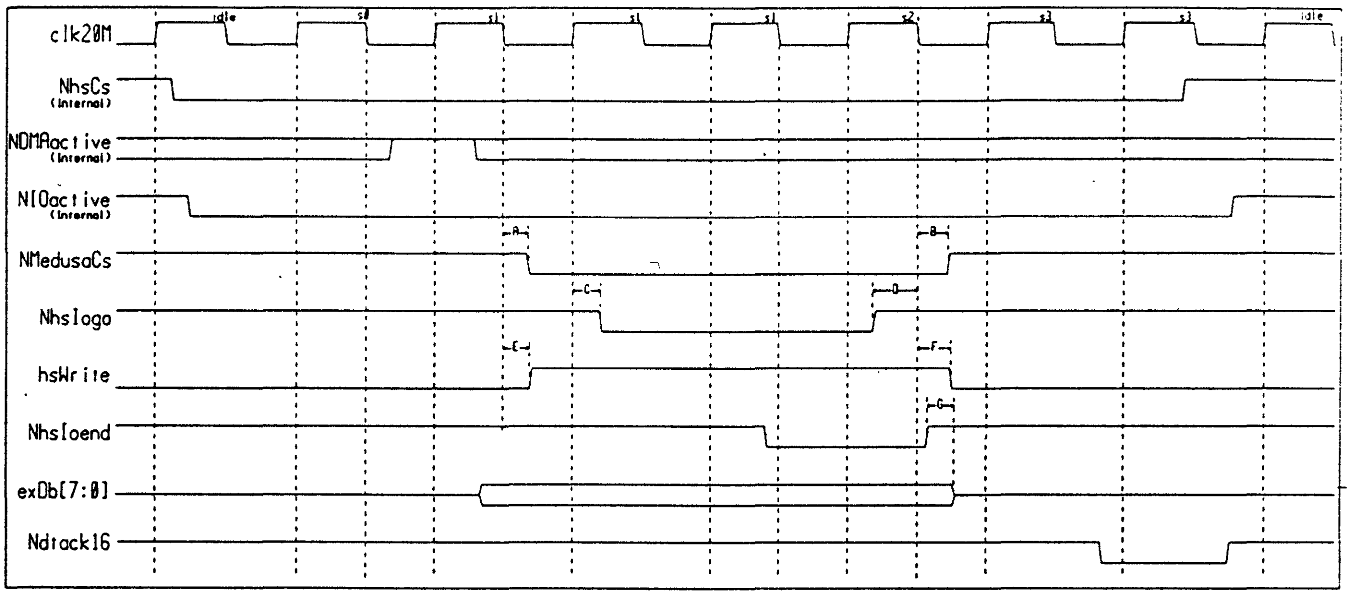

Figure 12. HS-HPIB Write Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M falling to NMedusaCs falling	17ns		
B	clk20M falling to NMedusaCs rising	15ns		
C	clk20M rising to NhsIogo falling	28ns		
D	clk20M rising to NhsIogo rising	27ns		
E	clk20M falling to hsWrite rising	19ns		
F	clk20M falling to hsWrite falling	18ns		
G	exDb[7:0] hold from NhsIoend rising		0ns	

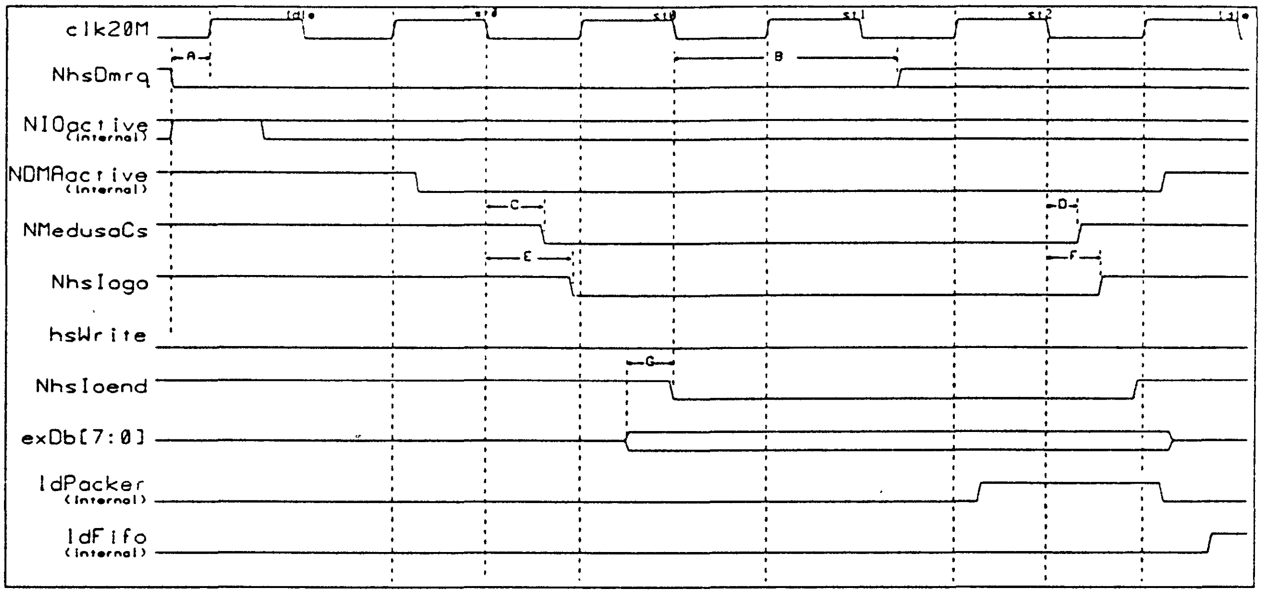

Figure 13. HS-HPIB Inbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	NhsDmrq setup to clk20M (tester ref)		10ns	
B	NhsIoend falling to NhsDmrq rising	150ns		
C	clk20M falling to NMedusaCs falling	18ns		
D	clk20M falling to NMedusaCs rising	17ns		
E	clk20M falling to NhsIogo falling	34ns		
F	clk20M falling to NhsIogo rising	31ns		
G	Data setup to NhsIoend falling		-30ns	

HEWLETT PACKARD

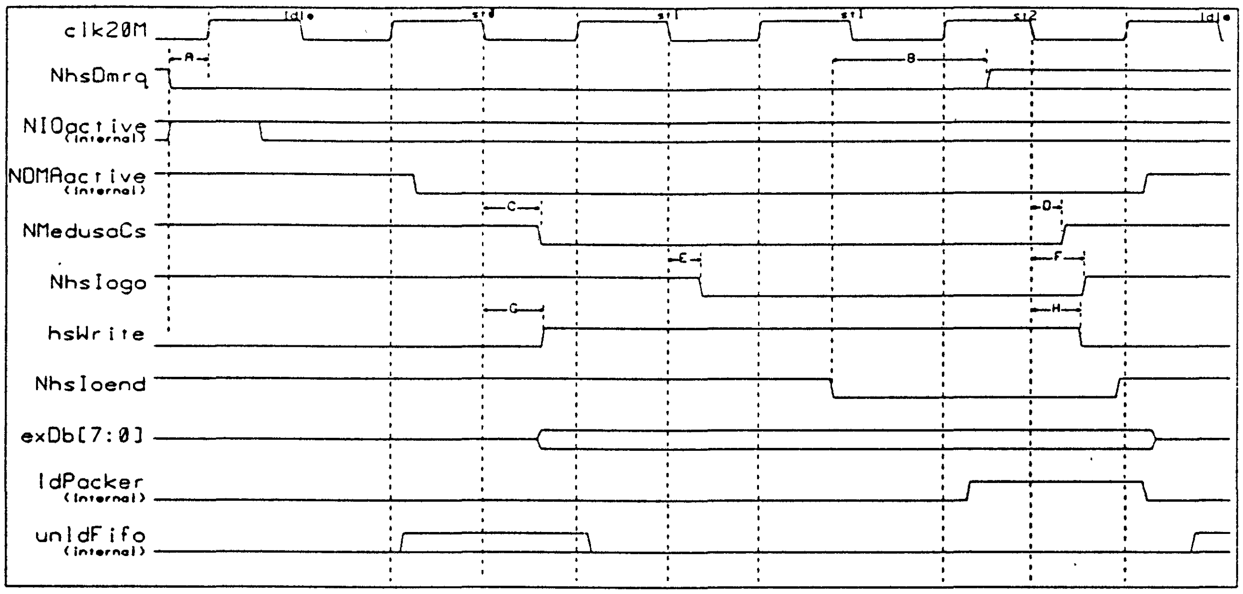

Figure 14. HS-HPIB Outbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	NhsDmrq setup to clk20M (tester ref)		10ns	
B	NhsIoend falling to NhsDmrq rising	150ns		
C	clk20M falling to NMedusaCs falling	18ns		
D	clk20M falling to NMedusaCs rising	17ns		
E	clk20M falling to NhsIogo falling	34ns		
F	clk20M falling to NhsIogo rising	31ns		
G	clk20M falling to hsWrite rising	19ns		
H	clk20M falling to hsWrite falling	18ns		

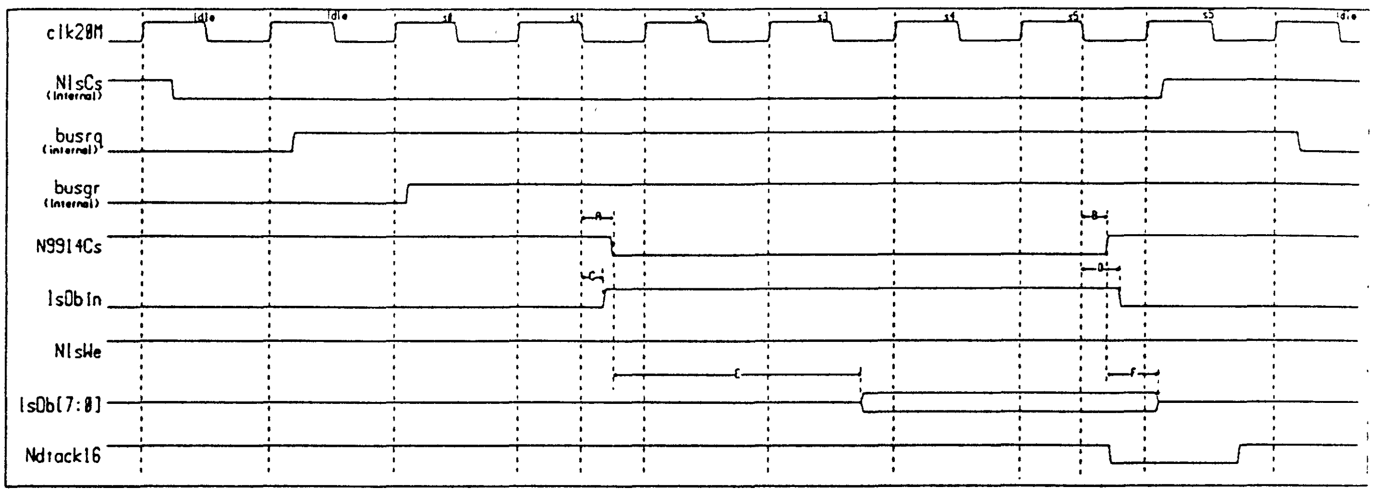

Figure 15. LS-HPIB Read Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M falling to N9914Cs falling	23ns		
B	clk20M falling to N9914Cs rising	19ns		
C	clk20M falling to lsDbin rising	13ns		
D	clk20M falling to lsDbin falling	12ns		
E	data valid from N9914Cs falling	150ns		
F	data tristate from lsDbin rising	100ns		

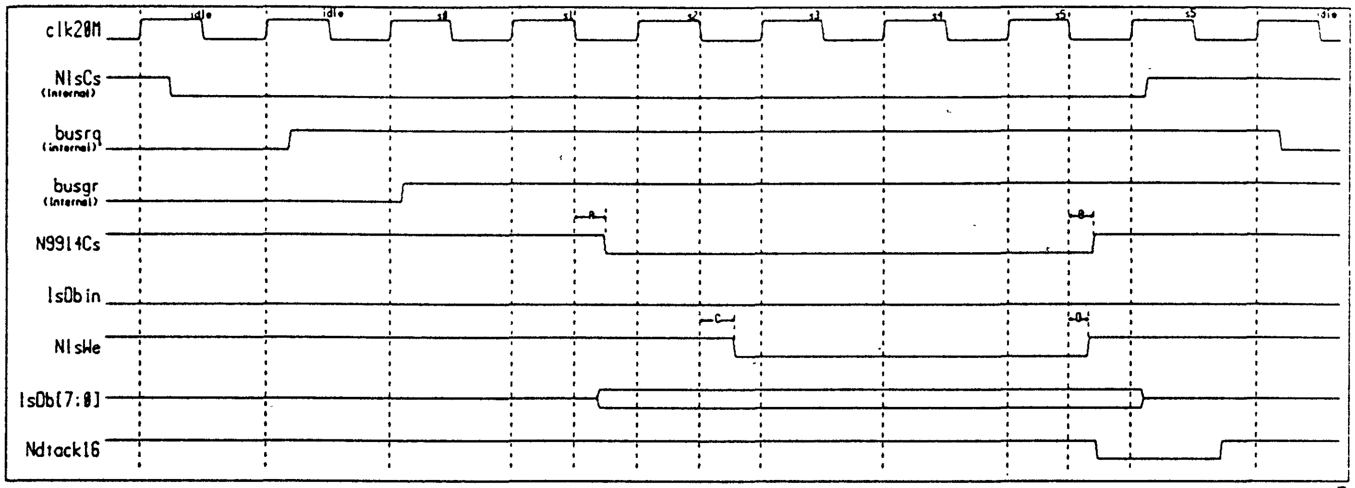

Figure 16. LS-HPIB Write Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M falling to N9914Cs falling	23ns		
B	clk20M falling to N9914Cs rising	19ns		
C	clk20M falling to NlsWe falling	13ns		
D	clk20M falling to NlsWe rising	11ns		

HEWLETT PACKARD

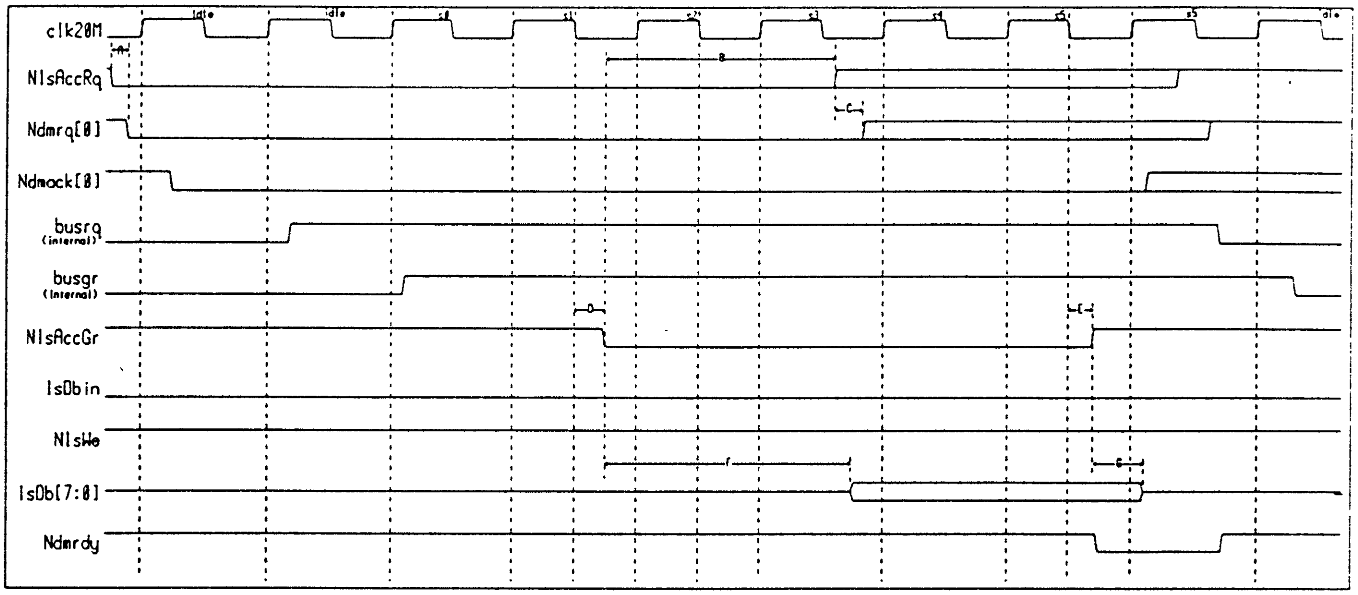

Figure 17. LS-HPIB Inbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	NlsAccRq falling to Ndmrq[0] falling	12ns		
C	NlsAccRq rising to Ndmrq[0] rising	11ns		
D	clk20M falling to NlsAccGr falling	23ns		
E	clk20M falling to NlsAccGr rising	19ns		
F	Data valid from NlsAccGr falling	150ns		
G	Data tristate from NlsAccGr rising	100ns		

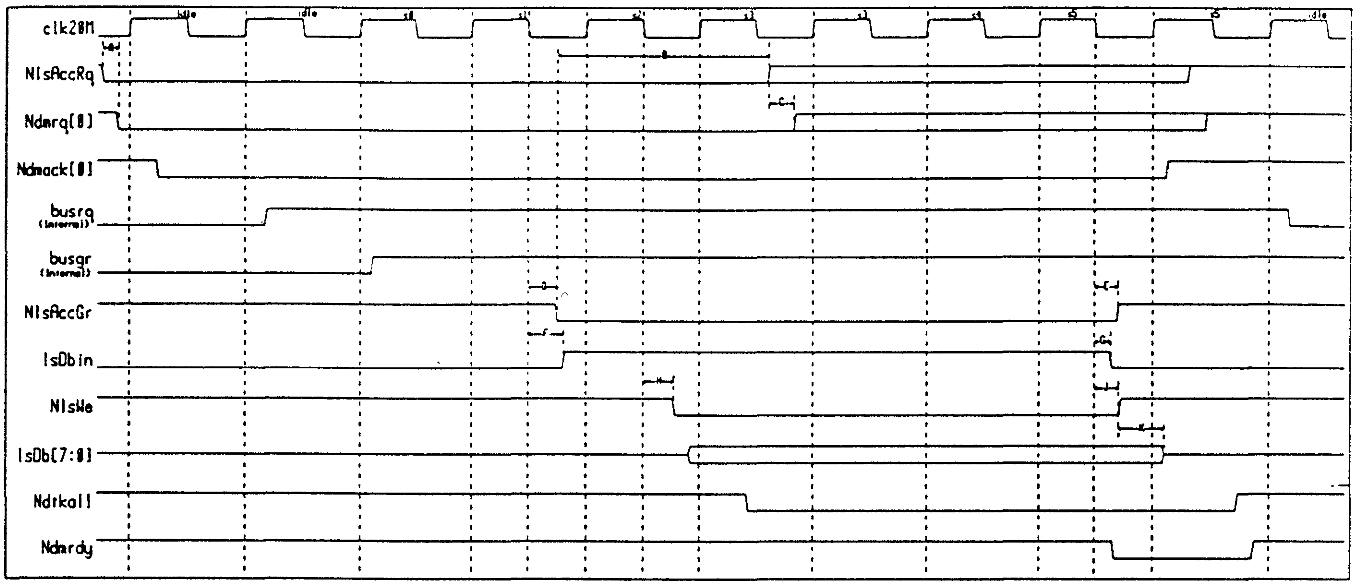

Figure 18. LS-HPIB Outbound DMA Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	NlsAccRq falling to Ndmrq[0] falling	12ns		
B	NlsAccGr fallingto NlsAccRq rising	100ns		
C	NlsAccRq rising to Ndmrq[0] rising	11ns		
D	clk20M falling to NlsAccGr falling	23ns		
E	clk20M falling to NlsAccGr rising	19ns		
F	clk20M falling to lsDbin rising	13ns		
G	clk20M falling to lsDbin falling	12ns		
H	clk20M falling to NlsWe falling	13ns		
J	clk20M falling to NlsWe rising	11ns		
K	Data hold from NlsWe rising		0ns	

HEWLETT PACKARD

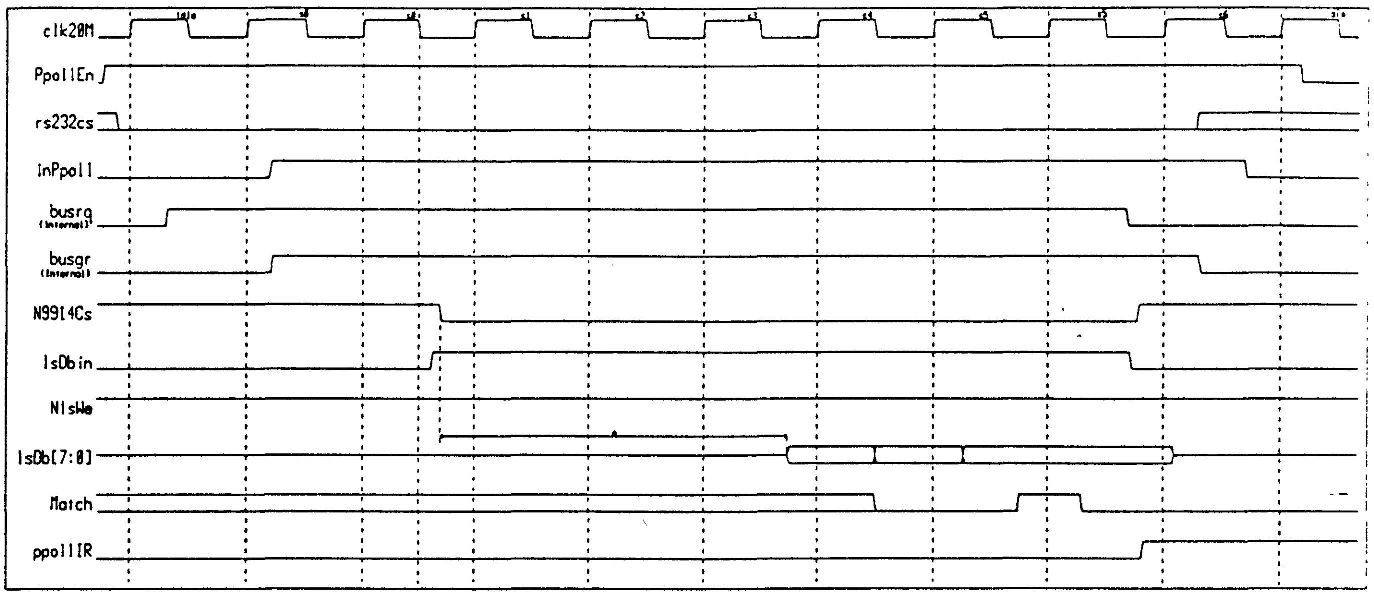

Figure 19. LS-HPIB Parallel Poll Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	N9914Cs falling to data valid	150ns		

HEWLETT PACKARD

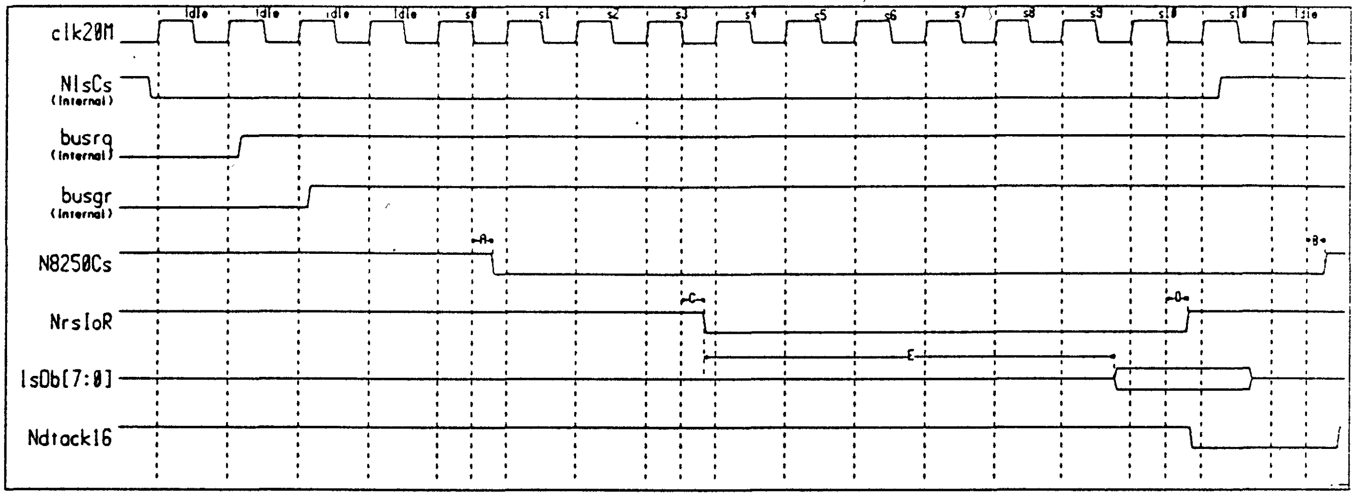

Figure 20. RS-232 Read Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M falling to N8250Cs falling	13ns		
B	clk20M falling to N8250Cs rising	12ns		
C	clk20M falling to NrsIoR falling	14ns		
D	clk20M falling to NrsIoR rising	13ns		
E	NrsIoR falling to data valid	300ns		

HEWLETT PACKARD

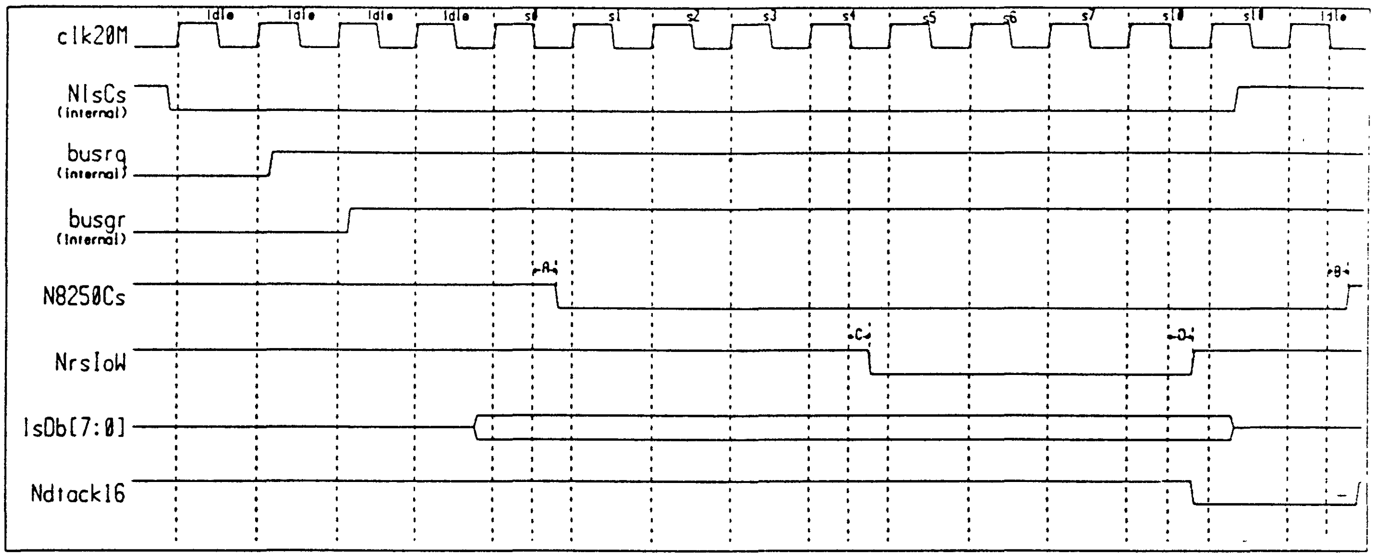

Figure 21. RS-232 Read Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M falling to N8250Cs falling	13ns		
B	clk20M falling to N8250Cs rising	12ns		
C	clk20M falling to NrsIoW falling	14ns		
D	clk20M falling to NrsIoW rising	13ns		

HEWLETT PACKARD

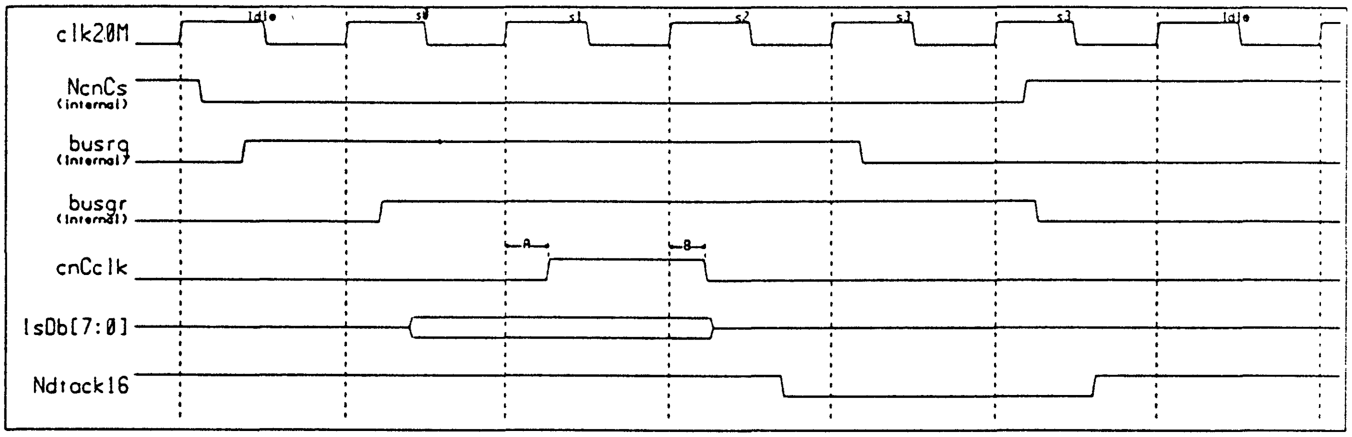

Figure 22. Centronics Write Register 7 Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M rising to cnCclk rising	22ns		
B	clk20M rising to cnCclk falling	23ns		

HEWLETT PACKARD

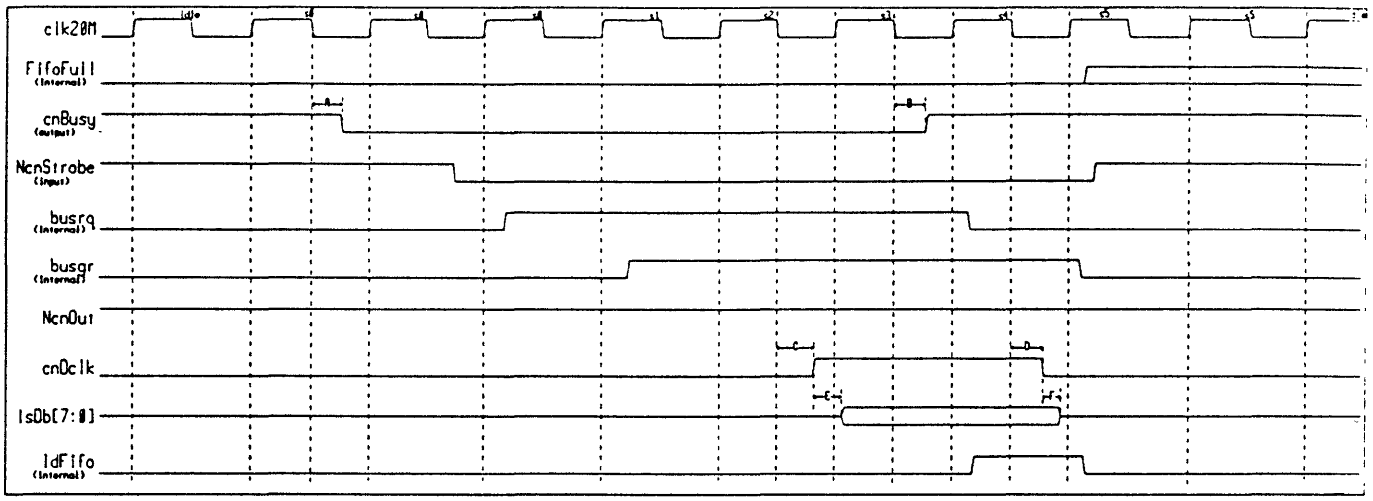

Figure 23. Centronics Back-end Inbound Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	clk20M falling to cnBusy falling	15ns		
B	clk20M falling to cnBusy rising	16ns		
C	clk20M falling to cnDclk rising	13ns		
D	clk20M falling to cnDclk falling	12ns		
E	cnDclk rising to data valid	30ns		
F	cnDclk falling to data tri-state	30ns		

HEWLETT PACKARD

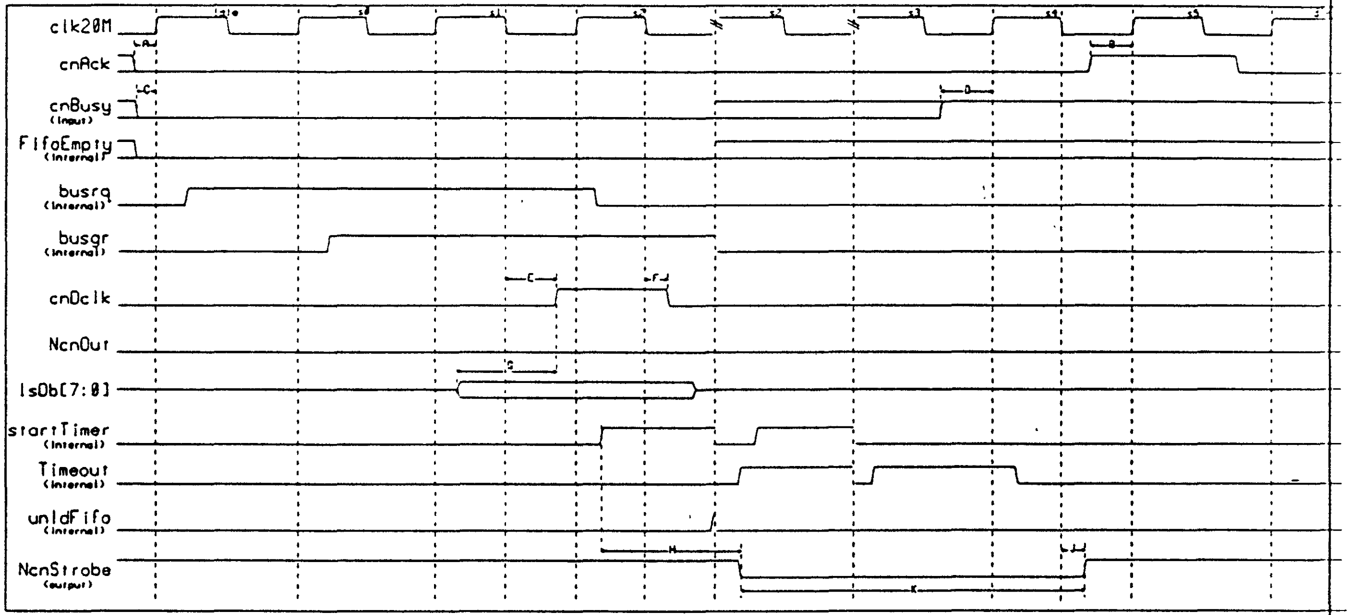

Figure 24. Centronics Back-end Outbound Cycle Timing

REF	Parameter	MAX	MIN	NOTE
A	cnAck falling to clk20M falling (tester ref)	7ns		
B	cnAck rising to clk20M falling (tester ref)	7ns		
C	cnBusy falling to clk20M falling (tester ref)	7ns		
D	cnBusy rising to clk20M falling (tester ref)	7ns		
E	clk20M falling to cnDclk rising	13ns		
F	clk20M falling to cnDclk falling	12ns		
G	data setup to cnDclk rising		20ns	
H	Centronics data Setup to NcnStrobe		700ns	
J	clk20M falling to NcnStrobe falling	13ns		
K	NcnStrobe low time		800ns	

2.6 *Packaging*

The -0302 part is in a EIAJ 160 pin plastic quad flat pack, for more information contact NID.

2.7 *Mechanical Characteristics*

The packaged part will meet or exceed all HP class B environmental specifications.

2.8 *Test Considerations*

Contact NID for tester configuration, specification and operation. Nikki is currently tested on a Sentry 15 IC tester. All of the tests were developed with Guide and exist on the same tapes as the schematics.

3. Software ERS and Theory of Operation

3.1 Software Interface

Nikki is functionally 4 independent devices that share the DIO-II bus. Each has a standard register set, a hidden register set and a select code register set. The hidden register set contains configuration information that the system or user can set which previous designs set using switches. On the first product using Nikki (375, 345), an EEPROM contains the "switch" settings for each interface, among other things. Thus, a user can configure his system uniquely without disassembling his machine, and the configuration is non-volatile. It is the bootrom's responsibility to configure Nikki if needed.

TABLE 12. Select code register map

Address (HEX)	Interface	Nikki	Default Select Code (Decimal)	Value (HEX)
0x400013	LS-HPIB	1	internal 7	0x47
0x400015	SCSI/HS-HPIB	1	14	0x6E
0x400017	Centronics	1	23	0x77
0x40001B	RS-232	1	9	0x69
0x40001D	SCSI/HS-HPIB	2	24	0x78
0x40001F	Centronics	2	17	0x71
0x400023	RS-232	2	10	0x6A
0x400025	SCSI/HS-HPIB	3	26	0x7A
0x400027	Centronics	3	19	0x73
0x40002B	RS-232	3	15	0x6F
0x40002D	SCSI/HS-HPIB	4	30	0x7E
0x40002F	Centronics	4	31	0x7F
0x400033	RS-232	4	16	0x70

Each select code except for LS-HPIB can be modified by writing the current value into the select code register. This enables hidden registers for that device and enables the select code register to be modified. The value to be written is the upper 8 bits of a 24 bit address, which is the value 0x60 logically OR'd with a 5 bit select code value from 0 to 32. The address space of the interface is 0xSC0000 through 0xSCFFFF (Where SC is the upper 8 bits of address). LS-HPIB's hidden mode can be enabled by writing a 0x47 to address 0x400013. This is internal select code 7 and its value can not be modified.

When hidden mode is enabled (writing the current value back into the select code register), the normal register set is replaced by a hidden register set. Any time the select code register is read, hidden mode is disabled.

If an interface does not exist (or is disabled), Nikki will not assert *NDTACK16* which will cause a bus error.

3.1.1 SCSI

TABLE 13. SCSI Standard Register Set

Address (HEX)	Name		Contents 7	6	5	4	3	2	1	0
0x1	ID	read	R/L	E1	E0	0	0	1	1	1
0x1	Reset	write	X	X	X	X	X	X	X	X
0x3	Status	read	IE	IR	IL1	IL0	L/W	I/O	DE1	DE0
0x3	Control	write	IE	X	X	X	L/W	I/O	DE1	DE0
0x5	Loopback	read	SD7	SD6	SD5	SD4	SD3	SD2	SD1	SD0
0x5	Wraparound	write	REQ	ACK	X	X	BSY	MSG	C/D	I/O
0x7	Configuration	read	TP	R1	R0	SD	P	SA2	SA1	SA0
0x7	Flush	write	X	X	X	X	X	X	X	X
0x9	Packer Status	read	0	8/10	DREQ	EID	0	PV2	PV1	PV0
0x9	Packer Status	write	X	X	X	EID	X	X	X	X
0xB	Packer Byte 0	read	D7	D6	D5	D4	D3	D2	D1	D0
0xB	Packer Byte 0	write	X	X	X	X	X	X	X	X
0xD	Packer Byte 1	read	D7	D6	D5	D4	D3	D2	D1	D0
0xD	Packer Byte 1	write	X	X	X	X	X	X	X	X
0xF	Packer Byte 2	read	D7	D6	D5	D4	D3	D2	D1	D0
0xF	Packer Byte 2	write	X	X	X	X	X	X	X	X
0x10	Byte DMA	read	D7	D6	D5	D4	D3	D2	D1	D0
0x10	Byte DMA	write	X	X	X	X	X	X	X	X
0x21	SPC Reg 0	BDID								
0x23	SPC Reg 1	SCTL								
0x25	SPC Reg 2	SCMD								
0x27	SPC Reg 3	TMOD								
0x29	SPC Reg 4	INTS								
0x2B	SPC Reg 5	PSNS								
0x2D	SPC Reg 6	SSTS								
0x2F	SPC Reg 7	SERR								
0x31	SPC Reg 8	PCTL								
0x33	SPC Reg 9	MBC								
0x35	SPC Reg A	DREG								
0x37	SPC Reg B	TEMP								
0x39	SPC Reg C	TCH								
0x3B	SPC Reg D	TCM								
0x3D	SPC Reg E	TCL								

3.1.1.1 SCSI ID and Reset Register (1)

R/L	Remote/Local: This is always 0 meaning LOCAL.
E1	Set if SCSI is differential drive; clear if single-ended
E0	Set if 16-bit DMA only; clear if 16 and 32 bit DMA capability
00111	Primary ID

†	A write of any value to register 1 will generate a hardware reset.

3.1.1.2 SCSI Status and Control Register (3)

IE	Interrupt Enable: set to enable interrupts.
IR	Set if SCSI is currently generating an interrupt (or would be if the *IE* bit were set).
IL1,IL0	Indicate the interrupt level.

IL1	IL0	Level
0	0	3
0	1	4
1	0	5
1	1	6

L/W	Longword/Word: If set, select 32 bit DMA operations, otherwise 16 bit transfers. Should be set only if 32-bit capability was indicated in ID register (1).
I/O	DMA direction: Set indicates an inbound transfer, otherwise, an outbound transfer. This bit must be properly set by controlling software prior to initiating a DMA transfer.
DE1,DE0	DMA enables for channel 1 and 0. Only one may be set at a time. Once these are set, *Ndmrq* will be asserted, and DMA to/from the 87033 will begin as soon as possible. Unlike HS-HPIB, there is no loss of data if the DE0/1 bits are toggled during DMA. i.e. switched from channel 1 to channel 0.
†	All bits except *IL1,IL0* are cleared by a software or hardware reset.

3.1.1.3 SCSI Wraparound and Loopback Data (5)

SD[7:0]	SCSI data bus bit 7-0. Used to sample the SCSI data bus during testing/troubleshooting.
REQ	SCSI control signal *REQ*.
ACK	SCSI control signal *ACK*.
BSY	SCSI control signal *BSY*.
MSG	SCSI control signal *MSG*.
C/D	SCSI control signal *C/D*.
I/O	SCSI control signal *I/O*.
†	Writing to this register can aid in testing/troubleshooting. Settings within this register will override the other SCSI drivers. During normal operation this register should have had a 0 written to it.

3.1.1.4 SCSI Flush and Hardware Configuration register (7)

TP	Terminator Power: If set, termination resistors are properly powered.
R1,R0	Information on maximum synchronous transfer rate that should be used in synchronous negotiations

		8MHz mode	
R1	R0	32-bit mode	16-bit mode
0	0	4.00 mbps	2.00 mbps
0	1	2.67 mbps	1.60 mbps
1	0	2.00 mbps	Sync not recommended
1	1	sync not available	
		10MHz mode	
0	0	5.00 mbps	2.50 mbps
0	1	3.33 mbps	2.00 mbps
1	0	2.50 mbps	Sync not recommended
1	1	sync not available	

SD	ShutDown: Indicates that Bus drivers have shut down if set. Always returns a zero value.
P	Parity selected: This bit should be read and then written out to the SPC *SCTL* register bit 3. It is set via a hidden register.
SA2,SA1,SA0	Inverted SCSI bus address 0-7. These three bits must be read, inverted, and written out to the SPC *BDID* register. These bits are set via a hidden register.
†	Writing to this register will flush any DMA data within the ASIC, reset the PACKER and all state machines. This may be used to recover after a peripheral disconnects from the bus in the middle of an outbound transfer.

3.1.1.5 SCSI Packer and Misc Status Register (9)

8/10	8MHz/10MHz mode. If clear, Fujitsu is running at 8MHz, otherwise 10MHz. This bit can be set only by hidden register 3.
DREQ	Fujitsu DMA request line. This bit *must* be set before the *Byte DMA* register is read.
EID	Enhanced ID. This bit is read/writable only when SCSI registers 0x9-0x10 exist. Previous SCSI implementations do not include these enhanced capabilities and will not have this bit read/writable.
PV2	Packer valid bit 2. If set, Packer byte 2, 1 and 0 are valid, and subsequent reads from registers 0xF, 0xD, and 0xB will return their data. This bit is only valid at the end of an inbound DMA transfer. It will be set if the Fujitsu transferred an odd number of bytes into Nikki.

PV1	Packer valid bit 1. If set, Packer byte 1 and 0 are valid, and subsequent reads from register 0xD, and 0xB will return their data. This bit is only valid at the end of an inbound DMA transfer. It will be set if the Fujitsu didn't transfer an aligned (DMA width) amount of data into Nikki.
PV0	Packer valid bit 0. If set, Packer byte 0 is valid, and a subsequent read from register 0xB will return its data. This bit is only valid at the end of an inbound DMA transfer. It will be set if the Fujitsu didn't transfer an aligned (DMA width) amount of data into Nikki.
†	A software or hardware reset will clear EID, PV2, PV1, and PV0.

3.1.1.6 SCSI Packer Byte 0 Register (B)

D7-0	Packer byte 0 data. This data is valid only if PV2-0 indicates so. It will be valid if the Fujitsu transferred less than the DMA width on the last inbound DMA transfer. This byte will be "stuck" in the packer waiting for more Fujitsu to FIFO dma transfers to fill the packer. Use register 7 to clear the packer contents.

3.1.1.7 SCSI Packer Byte 1 Register (D)

D7-0	Packer byte 1 data. This data is valid only if PV2-0 indicates so. It will be valid if the Fujitsu transferred only two or three bytes instead of four on the last inbound DMA transfer. Use register 7 to clear the packer contents.

3.1.1.8 SCSI Packer Byte 2 Register (F)

D7-0	Packer byte 2 data. This data is valid only if PV2-0 indicates so. It will be valid if the Fujitsu transferred only three bytes instead of four on the last inbound DMA transfer. All three bytes will be "stuck" in the packer waiting for a forth. Use register 7 to clear the packer contents.

3.1.1.9 SCSI Byte DMA Register (0x10)

D7-0	A read from this register will perform a single byte DMA xfer from the Fujitsu and return the data. A read from this register can be performed *only* when the DE1,DE0 bits are zero, the I/O bit is set and the DREQ bit is set. This register is used for aligning DMA xfers if a peripheral disconnected on an unaligned boundary.

TABLE 14. SCSI Hidden Register Set

Address (HEX)		Contents 7	6	5	4	3	2	1	0
0x1	read	0	0	IL1	IL0	BD2	BD1	BD0	P
0x1	write	0	0	IL1	IL0	BD2	BD1	BD0	P
0x3	read	0	0	0	8/10	0	0	R1	R0
0x3	write	0	0	0	8/10	0	0	R1	R0

3.1.1.10 SCSI Hidden Register 1

IL1,IL0	Interrupt Level. Same as standard register 1.
BD2,BD1,BD0	Inverted bus address. Same as standard register 7.
P	Enable, disable parity checking. Same as standard register 7.
†	The default value is 0x11. This register is only reset when *NPuReset* is asserted.

3.1.1.11 SCSI Hidden Register 3

8/10	Sets clock frequency that drives the 87033 to 8MHz when clear or 10MHz when set.
R1,R0	Max synchronous transfer rate. Same as standard register 7.
†	The default value is 0. This register is only reset when *NPuReset* is asserted.

3.1.2 HS-HPIB

There are actually two separate HS-HPIB standard register sets. One is compatible with the 98625B. The other adds 32-bit DMA transfer capability. The selection of one of the two sets is through a hidden register. The ID register uniquely identifies the two interfaces.

TABLE 15. HS-HPIB Register Set (98625B compatible)

Address (HEX)	Name		Contents 7	6	5	4	3	2	1	0
1	ID	read	R/L	0	0	0	1	0	0	0
1	Reset	write	X	X	X	X	X	X	X	X
3	Status	read	IE	IR	IL1	IL0	O/I	W/B	DE1	DE0
3	Control	write	IE	X	X	X	O/I	W/B	DE1	DE0
5	EOI	write	X	X	X	X	X	X	X	EOI
7	DMA Status	write	X	X	X	X	X	X	X	BYTE
11	Medusa Reg 0	INTR								
13	Medusa Reg 1	INTM								
15	Medusa Reg 2	DMA								
17	Medusa Reg 3	STS								
19	Medusa Reg 4	CTL								
1B	Medusa Reg 5	ADR								
1D	Medusa Reg 6	PPM								
1F	Medusa Reg 7	PPS								

TABLE 16. HS-HPIB Register Set (98625C)

Address (HEX)	Name		Contents 7	6	5	4	3	2	1	0
1	ID	read	R/L	1	0	0	1	0	0	0
1	Reset	write	X	X	X	X	X	X	X	X
3	Status	read	IE	IR	IL1	IL0	O/I	L/W	DE1	DE0
3	Control	write	IE	X	X	X	O/I	L/W	DE1	DE0
5	EOI	write	X	X	X	X	X	X	X	EOI
7	DMA Status	read	X	X	X	X	X	LC	VB1	VB0
7	DMA Status	write	X	X	X	X	X	X	VB1	VB0
11	Medusa Reg 0	INTR								
13	Medusa Reg 1	INTM								
15	Medusa Reg 2	DMA								
17	Medusa Reg 3	STS								
19	Medusa Reg 4	CTL								
1B	Medusa Reg 5	ADR								
1D	Medusa Reg 6	PPM								
1F	Medusa Reg 7	PPS								

3.1.2.1 HS-HPIB ID and Reset register (1)

R/L	Remote/Local: This is always 0 meaning LOCAL.
0001000 (1001000)	Device ID: Value is 8 for **98625B** mode (8/16 bit DMA) or 0x48 for 98625C mode (16/32 bit DMA).
†	Writing any value to this register will reset all of the state machines, the FIFO and *hsPon* will be deasserted for 500ns.

3.1.2.2 HS-HPIB Status and Control register (3)

IE	Interrupt Enable: set to enable interrupts.
IR	Set if HPIB is currently generating an interrupt (or would be if the *IE* bit were set).
IL1,IL0	Indicate the interrupt level read in from the configuration port.

IL1	IL0	Level
0	0	3
0	1	4
1	0	5
1	1	6

O/I	Outbound/Inbound: If set, DMA transfers are outbound, else they are inbound. This bit must be properly set by controlling software prior to initiating a DMA transfer.
W/B (L/W)	Word/Byte: If set, DMA transfers are 16-bits wide (word), else they are 8-bit wide (byte). **98625B** mode. Longword/Word: If set, DMA transfers are 32-bits wide (longword), else the are 16-bits wide. **98625C** mode.
DE1,DE0	DMA enables for channel 1 and 0. Only one may be set at a time, and it must be different than the DMA enables for the SCSI interface.
†	All of these bits except for IL1/IL0 are cleared during a soft/hard reset

3.1.2.3 HS-HPIB EOI Control (5)

EOI	End-Of-transfer Interrupt: If set, the *Ndone* signal will not cause an end-of-transfer interrupt. Also, no EOI is included with the last byte of an outbound transfer (to Medusa on *Hb[1:0]*).
†	This bit is cleared during a soft/hard reset.

3.1.2.4 HS-HPIB DMA status (7)

BYTE	This bit indicates that the last word of data in a 16-bit inbound transfer contained only one byte of valid data. **98625B** mode.
VB1,VB0	These two bits indicate how many bytes were valid in the last DMA transfer. **98625C** mode.

VB1	VB0	Valid Bytes
0	0	All
0	1	1
1	0	2
1	1	3

L/C	This bit, if set, indicates that either 16 or 32 bit DMA can be selected. If clear, only 16 bit DMA can be performed. **98625C** mode.
†	All bits except L/C are cleared during a hard/soft reset.

TABLE 17. HS-HPIB Hidden Register Set

Address (HEX)		7	6	5	4	3	2	1	0
0x1	read	0	0	IL1	IL0	Sys	F/S	E1	E0
0x1	write	0	0	IL1	IL0	Sys	F/S	E1	E0

3.1.2.5 HS-HPIB Hidden Register 1

IL1,IL0	Interrupt Level. Same as standard register 1.
Sys	System controller "switch". If set, Medusa is system controller.
F/S	Fast/slow "switch". If set, a tuned resistor is used by Medusa to optimize HPIB transfers, else a standard value is used.
E1	Select HS-HPIB interface type. If set, 16/32 bit DMA interface, otherwise a 8/16 bit DMA interface.
E0	DIO-I/II select. If set, the **98625C** interface can be used in either 16 or 32 bit DMA mode, othewise only 16 bit DMA mode is possible.
†	The default value is 0x19. This register is only reset when *NPuReset* is asserted.

3.1.3 LS-HPIB

TABLE 18. LS-HPIB Standard Register Set

Address (HEX)			Contents 7	6	5	4	3	2	1	0
0x3	Status	read	1	INT	0	0	0	0	0	DE
0x3	Control	write	X	X	X	X	X	X	X	DE
0x5	Aux Status	read	SYS	NAC	0	0	0	KB NMI	0	1
0x7	Par. Poll Status	read	IE	IR	0	0	0	0	0	0
0x7	Par. Poll Control	write	IE	IR	X	X	X	X	X	X
0x9	Par. Poll Mask	read	d7	d6	d5	d4	d3	d2	d1	d0
0x9	Par. Poll Mask	write	d7	d6	d5	d4	d3	d2	d1	d0
0x11	TMS9914 Reg	0								
0x13	TMS9914 Reg	1								
0x15	TMS9914 Reg	2								
0x17	TMS9914 Reg	3								
0x19	TMS9914 Reg	4								
0x1B	TMS9914 Reg	5								
0x1D	TMS9914 Reg	6								
0x1F	TMS9914 Reg	7								

3.1.3.1 LS-HPIB status/DMA control (3)

INT	Interrupt from TMS9914. If set the TMS9914 is interrupting.
DE	Writing a one to this bit will enable DMA on channel 0. Clearing it will disable DMA.

3.1.3.2 LS-HPIB auxillary status (5)

SYS	System controller line on HPIB. If set, the the TMS9914 is the system controller.
NAC	Acitve controller status. If clear, the TMS9914 is currently the Controller in Charge.
KB NMI	Keyboard non-maskable interrupt. If set, signal *NkbNmi* is asserted

3.1.3.3 LS-HPIB Parallel Poll Status and Control (7)

IE Interrupt enable. If set, Nikki will start performing a Parallel poll. If the poll response AND'd with the poll mask is not zero, the parallel poll will terminate and the IR bit will be set. This will cause *Nir[3]* to be asserted until IR is cleared.

IR Interrupt request. If set, the parallel poll has terminated with a match, and *Nir[3]* will be asserted. Writing a zero to this bit will clear the interrupt request.

† Both of these bits will be cleared by a hard/soft reset.

3.1.3.4 LS-HPIB Parallel Poll Mask (9)

d7-d0 These bits form a mask for enabling parallel poll responses. If a bit is clear, then a parallel poll response which has the corresponding bit set (DIO8-DIO1) will be ignored. Any time a response bit and its corresponding mask bit is set, the parallel poll will be terminated and the IR bit set.

† All bits will be cleared during a hard/soft reset.

TABLE 19. LS-HPIB Hidden Register Set

Address (HEX)		Contents							
		7	6	5	4	3	2	1	0
0x1	read	0	0	0	0	0	0	0	SYS
0x1	write	X	X	X	X	X	X	X	SYS

3.1.3.5 LS-HPIB Hidden Register 1

SYS This bit replaces the System Controller switch. It is reflected in standard register 5.

† The power-up value of this register is 0x1. This bit is set whenever *NPuReset* is asserted.

3.1.4 RS-232

TABLE 20. RS-232 Standard Register Set

Address (HEX)			Contents 7	6	5	4	3	2	1	0
0x1	read	R/L	1	0	0	0	0	1	0	
0x1	write	X	X	X	X	X	X	X	X	
0x3	read	IE	IR	IL1	IL0	0	0	HH	HS	
0x3	write	IE	X	X	X	X	X	X	X	
0x11	NS16550 Reg	0								
0x13	NS16550 Reg	1								
0x15	NS16550 Reg	2								
0x17	NS16550 Reg	3								
0x19	NS16550 Reg	4								
0x1B	NS16550 Reg	5								
0x1D	NS16550 Reg	6								
0x1F	NS16550 Reg	7								

3.1.4.1 RS-232 ID/Reset Register (1)

R/L	Remote/Local. If set, the RS-232 port should be used as the system console.
†	Writing any value to this register will reset the interface and assert *NrsReset* for 26 ×seconds.

3.1.4.2 RS-232 Status/Control Register (3)

IE	Interrupt Enable. If set, the interface is enabled to interrupt on the level specified by IL1,IL0.
IR	Interrupt Request. If set, the NS16550 is currently interrupting. This bit does not depend on IE.
IL1,IL0	Indicate the interrupting level.

IL1	IL0	Level
0	0	3
0	1	4
1	0	5
1	1	6

HH	Hardware Handshake. If set, hardware handshake circuitry is available on this interface.

HS	High Speed mode. If set, the higher baud rate generator clock is active. This frequency is 7.2727MHz. If clear the baud rate generator clock is 2.4614MHz
†	The IE bit is cleared during a soft/hard reset.

TABLE 21. RS-232 Hidden Register Set

Address (HEX)		Contents							
		7	6	5	4	3	2	1	0
0x1	read	R/L	0	IL1	IL1	0	HH	HS	REN
0x1	write	R/L	X	IL1	IL1	X	HH	HS	REN

R/L	This bit replaces the Remote/Local switch. It is reflected in standard register 1.
IL1/IL0	These bits replace the Interrupt level switches. They are reflected in standard register 3.
HH	This bit should be set if "hardware handshake" circuitry is implemented on this particular RS-232 interface. It is reflected in standard register 3.
HS	This bit when set will enable the 7.2727MHz baud rate generator clock, otherwise the standard 2.4576MHz frequency is used. It is reflected in standard register 3.
REN	This bit replaces the Remote Enable switch. When set, the modem handshake lines on the NS16550 are permanently driven true. When clear, the levels are those driven in from the RS-232 connector. This bit is reflected in standard register 3.
†	The default value for this register is 0x21. It is set to this value when *NPuReset* is asserted.

3.1.5 Centronics Software ERS

TABLE 22. Centronics Standard Register Set

Address			Contents							
(HEX)			7	6	5	4	3	2	1	0
0x1	ID	read	0	0	0	0	0	1	1	0
0x1	Reset	write	X	X	X	X	X	X	X	X
0x3	Status	read	IE	IR	IL1	IL0	IOM	I/O	DE1	DE0
0x3	Control	write	IE	X	X	X	IOM	I/O	DE1	DE0
0x5	Device Status	read	FULL	EMPTY	STROBE	BUSY	ACK	ERROR	SELECT	PE
0x5	write	X	X	X	X	X	X	X	X	
0x7	Device Control	read	0	0	0	0	0	INIT	SLCTIN	WRnRD
0x7	Device Control	write	X	X	X	X	X	INIT	SLCTIN	WRnRD
0x9	Intr Status	read	FULL	EMPTY	0	nBUSY	ACK	ERROR	SELECT	PE
0x9	Intr Control	write	FULL	EMPTY	0	nBUSY	ACK	ERROR	SELECT	PE
0xB	FIFO data	read	d7	d6	d5	d4	d3	d2	d1	d0
0xB	FIFO data	write	d7	d6	d5	d4	d3	d2	d1	d0

3.1.5.1 Centronics ID/Reset Register (1)

0x6	This is the ID value of the Centronics interface.
†	Writing any value to this register will reset the interface and clear the FIFO.

3.1.5.2 Centronics Interface Control/Status Register (3)

IE	Interrupt Enable. Setting this bit enables interrupting conditions to assert the *Nir* signal indicated by IL1,IL0.
IR	Interrupt Request. If this bit is set, the interface is currently generating an interrupt, or would be if IE was set.
IL1,IL0	Interrupt Level. Specifies interrupting level 3,4,5, or 6 (see SCSI reg 3).
IOM	IO modifier. If cleared, outbound transfers handshake with both BUSY and NACK and inbound transfers will use the FIFO. If set, outbound transfers will handshake with BUSY only and inbound transfers will only use 1 location in the FIFO (FIFO disable).
I/O	I/O direction. If cleared, the interface will be performing outbound transfers. If set, the interface will be performing inbound transfers.
DE1,DE0	DMA enable channel 1 and 0. When one (and only one) of these bits is set, the DMA machines will be enabled to start transferring. Once inbound DMA has been enabled it (DMA) must be used for the entire data transfer (until NACK for a ScanJet).

†	A hard/soft reset will clear all except the IL1,IL0 bits.

3.1.5.3 *Centronics Device Status Register (5)*

FULL	FIFO full. If set, the FIFO is currently full. This is valid for FIFO disable mode also.
EMPTY	FIFO empty. If set, the FIFO is currently empty. This is valid for FIFO disable mode also.
STROBE	Centronics STROBE. If set, the NSTROBE signal is asserted (low); otherwise NSTROBE is not asserted.
BUSY	Centronics BUSY. If set, the BUSY signal is asserted (high); otherwise BUSY is not asserted.
ACK	Centronics ACK. If set, the NACK signal is asserted (low); otherwise NACK is not asserted.
ERROR	Centronics NERROR. If set, the NERROR signal is asserted (low); otherwise NERROR is not asserted.
SELECT	Centronics SELECT. If set, the SELECT signal is asserted (high); otherwise SELECT is not asserted.
PE	Centronics PE (paper error). If set, the PE signal is asserted (high); otherwise PE is not asserted.
†	A hard/soft reset will clear FULL and STROBE, and set EMPTY.

3.1.5.4 *Centronics Device Control Register (7)*

INIT	Centronics NINIT. If set the NINIT signal will be asserted (low); otherwise NINIT will be deasserted.
SLCTIN	Centronics NSLCTIN (select in). If set the NSLCTIN will be asserted (low); otherwise NSLCTIN will be deasserted.
WRnRD	Centronics NAUTOFDXT, ScanJet WRnRD signal. If set, WRnRD will be asserted (high); otherwise WRnRD will be deasserted.
†	A hard/soft reset will affect this register but *will not* affect the actual lines. Therefore, to keep this register consistent with the signals, a write must be performed after a reset.

3.1.5.5 *Centronics Interrupt Control/Status Register (9)*

FULL IE/IR	FIFO full Interrupt Enable and Request. If a one is written to this bit then a FIFO full condition will be reflected in this register and IR of register 3.
EMPTY IE/IR	FIFO empty Interrupt Enable and Request. If a one is written to this bit then a FIFO empty condition will be reflected in this register and IR of register 3.

nBUSY IE/IR	NOT BUSY Interrupt Enable and Request. If a one is written to this bit then a NOT BUSY (BUSY LOW) condition will be reflected in this register and IR of register 3.
ACK IE/IR	ACK Interrupt Enable and Request. If a one is written to this bit then a NACK falling edge will set the read version of this bit and set IR of register 3. The interupting condition is cleared by writing a zero to this bit.
ERROR IE/IR	ERROR Interrupt Enable and Request. If a one is written to this bit then a NERROR falling *or* rising edge will set the read version of this bit and set IR of register 3. The interupting condition is cleared by writing a zero to this bit.
SELECT IE/IR	SELECT Interrupt Enable and Request. If a one is written to this bit then a NSELECT falling *or* rising edge will set the read version of this bit and set IR of register 3. The interupting condition is cleared by writing a zero to this bit.
PE IE/IR	PE (paper error) Interrupt Enable and Request. If a one is written to this bit then a PE falling *or* rising edge will set the read version of this bit and set IR of register 3. The interupting condition is cleared by writing a zero to this bit.
†	A soft/hard reset will clear all bits in this register.

3.1.5.6 Centronics FIFO Data Register (11)

D7-0 If the I/O bit of register 3 is cleared, a write to this register will write data into the FIFO. Software must guarantee that FULL is not set. The back end machine upon seeing the FIFO is not empty will perform an output transaction (wait for NOT BUSY, assert data, then NSTROBE, etc).
If the I/O bit of register 3 is set, a read from this register will return the next data in the FIFO. Software must guarantee that EMPTY is not set. The back end machine upon seeing the FIFO is not full will perform an input transaction (deassert BUSY, wait for NSTROBE, latch data into FIFO, etc).

TABLE 23. Centronics Hidden Register Set

Address (HEX)		Contents							
		7	6	5	4	3	2	1	0
0x1	read	0	0	IL1	IL0	0	0	0	0
0x1	write	X	X	IL1	IL0	X	X	X	X

IL1,IL0 Interrupting level. The values 0,1,2 and 3 correspond to interrupt levels 3,4,5 and 6. The value of these bits are reflected in standard register 3.

3.2 Software Performance Specifications and Considerations

Read the Errata section in conjunction with the following sections.

3.2.1 SCSI

Software must be careful to set the direction and size bits in register 3 prior to setting either of the DMA enable bits.

During inbound DMA when the Fujitsu interrupts, Nikki immediately requests a DMA transfer, if there is data in the FIFO. If the DMA channel is in priority mode, all of the DATA should be transferred to main memory. But there is a chance given other DMA hardware designs that the FIFO may not be empty even if the 87033 interrupt indicating an end of transfer. One should make sure that the DMA chip reports all bytes transferred prior to disabling DMA.

3.2.2 HS-HPIB

To work around a Nikki bug, the Medusa interrupt signal should be delayed 30 microseconds (time span of the internal timer). If this is not done, it is possible for software to respond to the Medusa interrupt and disable the DMA controller before Nikki has transferred the rest of its data.

3.2.3 LS-HPIB

Do not enable DMA if parallel polling is enabled! (This is not a new requirement)

3.2.4 RS-232

Hardware Handshake is completely independent from the presence of this ASIC.

3.2.5 Centronics

There is a state machine bug that requires the direction bit in register 3 to be toggled just before enabling INBOUND DMA. Basically, the machine was hand kludged incorrectly in a previous mask turn and this was never corrected. Thus this is a strange bug. The I/O bit should be cleared, and then BOTH the I/O bit AND the proper DMA enable bit should be set at the SAME time. This is for INBOUND DMA ONLY! If this is not done, the transfer will lock up because Nikki will never request a DMA transfer.

There is a design bug which inhibits the use of mixed DMA and programmed transfers for INBOUND ONLY! Once an INBOUND transfer uses DMA, all future inbound transfers, UNTIL THE NACK INTERRUPT (end of transfer) or an output transfer occurs, should use DMA. The hardware unloads a byte from the FIFO into a holding register and waits for the next DMA transfer to take that byte. If the DMA enable bit is cleared, this byte will remain in the holding register. Thus if the FIFO register is read, the next byte will be read instead of the one in the holding register. Once DMA is enabled this byte (the one in the holding register) will be the next one out. There is no way to tell if there is a byte in the holding register. An acceptable sequence follows: outbound DMA/non-DMA, inbound NON-DMA, inbound NON-DMA, inbound DMA, inbound DMA, NACK interrupt indicating end of transfer, outbound DMA/non-DMA, outbound DMA/non-DMA, inbound non-DMA, etc.

4. Errata

Below is a list of things I would change/fix given a chance to reroute/mask-turn Nikki.

- The Centronics FIFO counters do not count in a Grey code manner. There is currently a screen at NID for chips that might be flakey due to this. My test vectors do not pass HILO MIN timing with NO capacitance simulations, but even +3sigma parts at -20 C aren't that fast.
- The Centronics DMA state machine (cnDMAsm) was kludged incorrectly at one time and this bad fix has propagated through the last turn. The PD source should generate proper logic to fix this problem.
- The Centronics DMA state machine should be modified to not unload the FIFO into the holding register until *Ndmack* is asserted during inbound DMA. This would enable mixed DMA and programmed inbound transfers.
- The HS-HPIB DMA machine should generate a FIFO timeout when a Medusa EOT occurs during inbound DMA. This would eliminate the need for the 30microsecond delay of the Medusa interrupt signal.
- It would be nice for software to have a Centronics "IDLE" interrupting condition. Centronics would be IDLE during outbound transfers when the FIFO is empty and the last transfer has completed (including the detection of the NACK pulse). This would be used by software to guarantee that the last byte was accepted by the peripheral so that it could turn the bus around.

5. State Machine Algorithms

The algorithms for most of the state machines in the chip are included in this section. The equations are expressed in a Pascal-like format which is required for the PAL equation generation aid: PD.

5.1 SCSI/HS-HPIB DIO-FIFO DMA Machine

```
pal DIOdmaSM;

{
        PAL Designer DIO DMA State Machine program.

        This file generates the Next State Equations and Output
        equations for the DIO-DMA half of the SCSI portion of the interface.

        Don Soltis      v0.1    9/27/88
                Initial equations
        Don Soltis      v0.2    1/8/89
                modified for active low outputs

}

input .                 DMAen,IOactive,FifoFull,FifoEmpty,inbound,
                        rst,word,almostFull,
                        dmack,dmrq,dtacki,
                        timeout,
                        EOT,                    {End of transfer}
                        SCSI,                   {SCSI/~HPIB select}
                        state[2..0];

output                  state[2..0],
                        SldFifo,                { set ldFifo }
                        RldFifo,                { reset ldFifo }
                        SunldFifo,              { set unldFifo }
                        RunldFifo,              { reset unldFifo }
                        Sdmrq,          { set DMA request }
                        Rdmrq,          { reset DMA request }
                        oDataEn,        { Output data Enable }
                        intr,           { End of xfer interrupt }
                        dmrdy;

pal_type                'ASIC';
pal_pin_order           ~SldFifo,~SunldFifo,~Sdmrq;

state_vars              state[2..0];

macro idle;             begin  0  end;

macro inb0;             begin  1  end; {Inbound state 0}
macro inb1;             begin  3  end; {Inbound state 1}
macro inb2;             begin  2  end; {Inbound state 2}

macro outb0;            begin  4  end; {outbound state 0}
macro outb1;            begin  5  end; {outbound state 1}
macro outb2;            begin  6  end; {outbound state 2}

macro next_state;       begin  state[2..0]  end;

begin

if rst then begin       { Reset }

   state[2..0]  := idle;
   Rdmrq        := 1;
   RldFifo      := 1;
   RunldFifo    := 1;

   end;

```

HEWLETT PACKARD

```
case state[2..0] of

   idle:        {idle states}
     begin

        if DMAen then begin  { Make sure DMA is enabled }

           {*****}
           { start inbound if Fifofull or timeout }
           {*****}
          if( inbound AND (FifoFull or (timeout*~FifoEmpty)) ) then begin
           SunldFifo := 1;
           Sdmrq     := 1;
           next_state := inb0;
           end

           {*****}
           { start outbound if FifoEmpty }
           {*****}
          else if( ~inbound AND FifoEmpty ) then begin
            Sdmrq   := 1;
            next_state := outb0;
            end
         else
            next_state := idle;

          end {if DMAen}
        else
           next_state := idle;

        end; {in idle0}

   inb0: begin
      RunldFifo := 1;
        if dmack then
           next_state := inb1
        else
           next_state := inb0; { wait for dmack }
      end; {case inb0}

   inb1: begin
      if FifoEmpty then
         Rdmrq := 1;    {no more DMA requests}

      oDataEn := 1;
      RunldFifo := 1;
      next_state := inb2;
      end; {case inb1}

   inb2: begin
      dmrdy := 1;
      oDataEn := 1;
      if dmack then begin
        if ( ~SCSI and EOT ) then begin
           intr = 1;           { interrupt if we are done and in HS-HPIB mode}
           Rdmrq := 1;         {no more DMA requests}
           end;
        next_state := inb2;
        end
      else begin
        if ~dmrq then begin
           next_state := idle;
           end
        else begin             { Another transfer }
           SunldFifo  := 1;
           next_state := inb0;
           end;
        end;
      end; {case inb2}

   outb0: begin

      if DMAen then begin
        if dmack then begin
           if almostFull then          {Only have one slot left}
              Rdmrq := 1               {no more requests}
           else
```

```
              Sdmrq := 1;               {continue Request}
           if dtacki then begin
              next_state := outb1
              end
           else
              next_state := outb0;
           end
        else
           next_state := outb0; { wait for dmack }
        end

      else begin
        next_state := idle;
        Rdmrq := 1;         {don't dmrq}
        end;

      end; {case outb0}

   outb1: begin
      dmrdy := 1;               {data hold 85 ns after this}
      SldFifo := 1;             {Actually occurs in next state}
      next_state := outb2;
      end; {case outb1}

   outb2: begin
      dmrdy := 1;
      RldFifo := 1;
      if dmack then
         next_state := outb2
      else begin
         if ~dmrq then
            next_state := idle
         else begin
            next_state := outb0;
            end;
         end;

      end; {case outb2}

   end; { case }
end. {END}
```

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Mon Sep 18 21:16:52 1989
PALASM Description for DIODMASM
/SLDFIFO
/SUNLDFIFO /SDMRQ

;; PAL Attributes File is
;; Using structure info for "ASIC"

DMRDY := STATE[1] * /STATE[0]
  + STATE[2] * /STATE[1] * STATE[0]

INTR = DMACK * EOT * /SCSI * /STATE[2] * STATE[1] * /STATE[0]

ODATAEN := /STATE[2] * STATE[1]

RDMRQ := DMACK * STATE[2] * /STATE[1] * /STATE[0] * ALMOSTFULL
  + /DMAEN * STATE[2] * /STATE[1] * /STATE[0]
  + DMACK * EOT * /SCSI * /STATE[2] * STATE[1] * /STATE[0]
  + /STATE[2] * STATE[1] * STATE[0] * FIFOEMPTY
  + RST

RLDFIFO := STATE[2] * STATE[1] * /STATE[0]
  + RST

RUNLDFIFO := /STATE[2] * STATE[0]
  + RST

/SDMRQ := /DMAEN
  + /DMACK * STATE[2]
  + STATE[2] * ALMOSTFULL
  + STATE[1]
  + STATE[0]
  + /STATE[2] * /FIFOFULL * FIFOEMPTY * INBOUND
  + /STATE[2] * /FIFOEMPTY * /INBOUND
  + /STATE[2] * /FIFOFULL * INBOUND * /TIMEOUT

/SLDFIFO := /STATE[2]
  + STATE[1]
  + /STATE[0]

STATE[0] := DMAEN * DMACK * STATE[2] * /STATE[1] * /STATE[0] * DTACKI
  + /DMACK * DMRQ * /STATE[2] * STATE[1] * /STATE[0]
  + DMAEN * /STATE[2] * /STATE[1] * FIFOFULL * INBOUND
  + DMAEN * /STATE[2] * /STATE[1] * /FIFOEMPTY * INBOUND * TIMEOUT
  + /STATE[2] * /STATE[1] * STATE[0]

STATE[1] := DMACK * STATE[1] * /STATE[0]
  + STATE[2] * /STATE[1] * STATE[0]
  + DMACK * /STATE[2] * STATE[0]
  + /STATE[2] * STATE[1] * STATE[0]

STATE[2] := DMRQ * STATE[2] * STATE[1] * /STATE[0]
  + DMACK * STATE[2] * STATE[1] * /STATE[0]
  + DMAEN * STATE[2] * /STATE[1]
  + STATE[2] * /STATE[1] * STATE[0]
  + DMAEN * /STATE[1] * /STATE[0] * FIFOEMPTY * /INBOUND

/SUNLDFIFO := /DMAEN * /STATE[1]
  + STATE[2]
  + DMACK * STATE[1]
  + /DMRQ * STATE[1]
  + STATE[0]
  + /STATE[1] * /FIFOFULL * FIFOEMPTY
  + /STATE[1] * /INBOUND
  + /STATE[1] * /FIFOFULL * /TIMEOUT


```

5.2 SCSI Register Access Machine

```
pal SCSIdioSM;

```

```
{
        PAL Designer SCSI DIO State Machine program.

        This file generates the Next State Equations and Output
        equations for DIO register accesses of the SCSI portion
        of Nikki.

        Don Soltis      v0.1    1/11/89
                Initial equations
        Don Soltis      v0.2    1/14/89
                Stretched out SPC (87033) signals

}

input           CS, write, DMAactive, rst, a[5..0],
                hiddenEn, state[2..0];

output          state[2..0],
                IOactive,
                writeReg1, writeReg3, wrapLe, writeReg7,  { DIO reg writes }
                writeReg9,
                readReg1,  readReg3, loopOe,  readReg7,   { DIO reg reads  }
                readReg9, readRegB, readRegD, readRegF,
                schin, dmrsp,                              { byte DMA reg }
                dtack, OdioEn1,
                OscsiEn,
                swrite, sread;          { SPC control sigs }
                                        { spcCs, Nswrite and Nsread }
                                        { are derived from these }

pal_type                'ASIC';

{ ONLY the active low outputs need to be specified here for logic reduction }
{ use the psm version of pd2ddl for proper generation }

state_vars              state[2..0];

macro idle;             begin  0  end;
macro s0;               begin  1  end;
macro s1;               begin  3  end;
macro s2;               begin  2  end;
macro s3;               begin  6  end;
macro s4;               begin  7  end;
macro s5;               begin  5  end;
macro s6;               begin  4  end;

macro address;          begin a[5..0] end;

macro dio_reg;          begin
        ( a[5] == 0 )
        end;

macro spc_reg;          begin
        ( a[5] == 1 )
        end;

macro next_state;       begin  state[2..0]  end;

Procedure enable_dio_reg;
   begin
        case a[4..1] of
           0: {reg 1}
              readReg1 = 1;
           1: {reg 3}
              readReg3 = 1;
           2: {reg 5}
              loopOe = 1;
           3: {reg 7}
              readReg7  = 1;
           4: {reg 9}
              readReg9  = 1;
           5: {reg B}
              readRegB  = 1;
           6: {reg D}
              readRegD  = 1;
           7: {reg F}
              readRegF  = 1;
           end;
        end;

Procedure write_dio_reg;
   begin
        case a[4..1] of
           0: {reg 1}
              writeReg1 = 1;
           1: {reg 3}
              writeReg3 = 1;
           2: {reg 5}
              begin
                wrapLe = 1;
                OscsiEn = 1;
              end;
           3: {reg 7}
              writeReg7  = 1;
           4: {reg 9}
              writeReg9  = 1;
           end;
        end;

Procedure SCSI_state_hold;
{ This procedure is used in states s2,s3,s4,s5 }
   begin
      if CS then begin
         IOactive = 1;  {continue to assert IOactive the entire time}
         if ~write then begin
            OdioEn1 = 1;
            if a[5..1] = 8 then begin
               schin = 1;       {scHin for read DMA reg}
               dmrsp = 1;
               end;
            end;
         if spc_reg then begin {spc reg access}
            if write then begin
               OscsiEn = 1;
               swrite  = 1;
               end
            else sread = 1
            end
         else {dio_reg} begin
            if ~write then begin
               enable_dio_reg;
               end
            else begin
               if a[5..1] = 0 then writeReg1 = 1;       {Soft reset signal}
               if a[5..1] = 2 then OscsiEn = 1;         {drive dio onto scsi}
               if a[5..1] = 3 then writeReg7 = 1;       {Fifo reset signal}
               end;
            end; {else dio_reg}
         case state[2..0] of
            s2: next_state := s3;
            s3: next_state := s4;
            s4: next_state := s5;
            s5: begin
               if write*~spc_reg then write_dio_reg;
               next_state := s6;
               end;
            end;
         end {if CS}
      else {~CS}
        next_state := idle;
     end; {procedure SCSI_state_hold}

{*************************************
 * Main
 *************************************}
begin

if rst then begin       { Reset }

   state[2..0]       := idle;

   end;

case state[2..0] of
```

```

idle:       {idle states}
  begin
   if CS then begin
     IOactive = 1;
     next_state := s0;
     end
   else
     next_state := idle;
   end; {in idle}

s0:         { continue to assert IOactive, DMA has priority }
  begin
   if CS then begin
     IOactive = 1;
     next_state := s1
     end
   else
     next_state := idle;
   end; {case s0}

s1:
  begin
   if CS then begin
      IOactive = 1;
      if DMAactive then
         next_state := s1
      else begin  {~DMAactive}

         if ~write then begin
            OdioEn1 = 1;
            if a[5..1] = 8 then schin = 1;   {scHin for read DMA reg}
            end;

         if spc_reg then begin   {spc reg access}
            if write then OscsiEn = 1;
            end
         else begin {dio_reg}
            if ~write then begin { read }
               enable_dio_reg;
               end;
            end; {else dio_reg}
         next_state := s2;
         end; {else ~DMAactive}
      end {if CS}
   else {~CS}
     next_state := idle;

   end; {case s1}

s2:
  begin
     SCSI_state_hold;
  end; {case s2}

s3:
  begin
     SCSI_state_hold;
  end; {case s3}

s4:
  begin
     SCSI_state_hold;
  end; {case s4}

s5:
  begin
     SCSI_state_hold;
  end; {case s5}

s6:
  begin
   if CS then begin
      IOactive = 1;  {continue to assert IOactive the entire time}
      dtack = 1;
      if ~write then begin
          OdioEn1 = 1;
```

```
             if a[5..1] = 8 then schin = 1;     {scHin for read DMA reg}
             end;
         if spc_reg then begin {spc reg access}
            if write then begin
               OscsiEn = 1;
               swrite  = 0; {end Transaction}
               end;
            end
         else {dio_reg} begin
            if ~write then begin
               enable_dio_reg;
               end;
            {else {dio reg write} write_dio_reg;}
            end; {else dio_reg}
         next_state := s6;
         end {if CS}
      else {~CS}
        next_state := idle;
     end; {case s3}

   end;
end.
```

HEWLETT
PACKARD

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Tue Oct 17 18:52:13 1989
PALASM Description for SCSIDIOSM


;; PAL Attributes File is
;; Using structure info for "ASIC"

DMRSP = CS * /WRITE * /A[5] * A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * STATE[0]
  + CS * /WRITE * /A[5] * A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]

DTACK = CS * STATE[2] * /STATE[1] * /STATE[0]

IOACTIVE = CS

LOOPOE = CS * /WRITE * /A[5] * /A[4] * /A[3] * A[2] * /A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * A[2] * /A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * A[2] * /A[1] * STATE[1] * /STATE[0]

ODIOEN1 = CS * /WRITE * STATE[2]
  + CS * /WRITE * STATE[1] * /DMAACTIVE
  + CS * /WRITE * STATE[1] * /STATE[0]

OSCSIEN = CS * WRITE * A[5] * STATE[2]
  + CS * WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[2] * STATE[0]
  + CS * WRITE * A[5] * STATE[1] * /DMAACTIVE
  + CS * WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[1] * /STATE[0]
  + CS * WRITE * A[5] * STATE[1] * /STATE[0]

READREG1 = CS * /WRITE * /A[5] * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]

READREG3 = CS * /WRITE * /A[5] * /A[4] * /A[3] * /A[2] * A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * /A[2] * A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * /A[2] * A[1] * STATE[1] * /STATE[0]

READREG7 = CS * /WRITE * /A[5] * /A[4] * /A[3] * A[2] * A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * A[2] * A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * /A[3] * A[2] * A[1] * STATE[1] * /STATE[0]

READREG9 = CS * /WRITE * /A[5] * /A[4] * A[3] * /A[2] * /A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * A[3] * /A[2] * /A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]

READREGB = CS * /WRITE * /A[5] * /A[4] * A[3] * /A[2] * A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * A[3] * /A[2] * A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * A[3] * /A[2] * A[1] * STATE[1] * /STATE[0]

READREGD = CS * /WRITE * /A[5] * /A[4] * A[3] * A[2] * /A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * A[3] * A[2] * /A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * A[3] * A[2] * /A[1] * STATE[1] * /STATE[0]

READREGF = CS * /WRITE * /A[5] * /A[4] * A[3] * A[2] * A[1] * STATE[2]
  + CS * /WRITE * /A[5] * /A[4] * A[3] * A[2] * A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * /A[4] * A[3] * A[2] * A[1] * STATE[1] * /STATE[0]

SCHIN = CS * /WRITE * /A[5] * A[4] * /A[3] * /A[2] * /A[1] * STATE[2]
  + CS * /WRITE * /A[5] * A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /DMAACTIVE
  + CS * /WRITE * /A[5] * A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]

SREAD = CS * /WRITE * A[5] * STATE[2] * STATE[0]
  + CS * /WRITE * A[5] * STATE[1] * /STATE[0]

STATE[0] := CS * STATE[2] * STATE[1]
  + CS * /STATE[2] * STATE[0] * DMAACTIVE
  + CS * /STATE[2] * /STATE[1]

STATE[1] := CS * STATE[1] * /STATE[0]
  + CS * /STATE[2] * STATE[0]

STATE[2] := CS * STATE[2]
  + CS * STATE[1] * /STATE[0]

SWRITE = CS * WRITE * A[5] * STATE[2] * STATE[0]
  + CS * WRITE * A[5] * STATE[1] * /STATE[0]
```

```

WRAPLE = CS * WRITE * /A[5] * /A[4] * /A[3] * A[2] * /A[1] * STATE[2] * /STATE[1] * STATE[0]

WRITEREG1 = CS * WRITE * /A[5] * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * STATE[0]
  + CS * WRITE * /A[5] * /A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]

WRITEREG3 = CS * WRITE * /A[5] * /A[4] * /A[3] * /A[2] * A[1] * STATE[2] * /STATE[1] * STATE[0]

WRITEREG7 = CS * WRITE * /A[5] * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * STATE[0]
  + CS * WRITE * /A[5] * /A[4] * /A[3] * A[2] * A[1] * STATE[1] * /STATE[0]

WRITEREG9 = CS * WRITE * /A[5] * /A[4] * A[3] * /A[2] * /A[1] * STATE[2] * /STATE[1] * STATE[0]
```

5.3 SCSI FIFO-87033 DMA Machine

```
pal SCSIdmaSM;

{
        PAL Designer SCSI DMA State Machine program.

        This file generates the Next State Equations and Output
        equations for the DMA half of the SCSI portion of the interface.

        Don Soltis      v0.1    9/8/88
                Initial equations
        Don Soltis      v0.2    9/12/88
                Fixed Reseting of unld- and ld-Packer[1] for
                ldFifo
        Don Soltis      v0.3    9/13/88
                Moved reset of unld- and ld-Packer[1] to during
                clk8 low period (removed spikes)
        Don Soltis      v0.4    9/15/88
                changed from 8MHz to 16MHz clock to get rid of
                spikes on rld- and runld-Packer.
        Don Soltis      v0.5    9/19/88
                modified state transitions to increase performance
                and fixed outbound equations
        Don Soltis      v0.6    10/6/88
                Modified ldFifo and unldFifo as RS flip flops
                to fix race condition
        Don Soltis      v0.7    12/19/88
                Added pal_pin_order to specify active low outputs and
                used pd2ddl with '-p' switch.
        Don Soltis      v0.8    3/1/89
                Added assertion of DMAactive during idle1
        Don Soltis      v1.1    9/18/89
                Removed usage of DONE, and fixed DMA enable signal
                so that no data is lost.

}

input                   DMAen,IOactive,FifoFull,FifoEmpty,inbound,
                        rst,
                        dmrq,           {This should be delayed one}
                                        {extra clock cycle}
                        word,
                        ldPacker[3..0],unldPacker[3..0],
                        state[1..0],clk16,
                        delayedDmrsp;

output                  state[1..0],
                        SldFifo,                { set ldFifo }
                        RldFifo,                { reset ldFifo }
                        SunldFifo,              { set unldFifo }
                        RunldFifo,              { reset unldFifo }
                        SldPacker[3..0],        { set ldPacker }
                        RldPacker[3..0],        { reset ldPacker }
                        SunldPacker[3..0],      { set unldPacker }
                        RunldPacker[3..0],      { reset unldPacker }
                        DMAactive,
                        OdataEn,                { SCSI data output enable }
                        dmrsp;

pal_type                'ASIC';
```

```

{ ONLY the active low outputs need to be specified here for logic reduction }
{ use the psm version of pd2ddl for proper generation }

pal_pin_order           ~SldFifo,~SunldFifo,
                        ~SldPacker[3..0],
                        ~SunldPacker[3..0],
                        ~rst;

state_vars              state[1..0];

macro idle0;            begin  0  end;
macro idle1;            begin  1  end;
macro inb0a;            begin  3  end; {Inbound/outbound, byte 0}
macro inb0b;            begin  2  end; {Inbound/outbound, byte 0}

macro next_state;       begin  state[1..0]  end;

macro iBYTE0;   begin  0 end;
macro iBYTE1;   begin  1 end;
macro iBYTE2;   begin  2 end;
macro iBYTE3;   begin  4 end;
macro iBYTE4;   begin  8 end;

macro packer_full; begin
        (ldPacker[3]*~word or ldPacker[1]*word)
        end;


macro oBYTE0;   begin  0 end;
macro oBYTE1;   begin  1 end;
macro oBYTE2;   begin  2 end;
macro oBYTE3;   begin  4 end;
macro oBYTE4;   begin  8 end;

macro unpacker_empty; begin
        (~unldPacker[3]*~unldPacker[2]*~unldPacker[1]*~unldPacker[0])
        end;

macro unpacker_emptied; begin
        (unldPacker[3]*~word or unldPacker[1]*word)
        end;

begin

if rst then begin       { Reset }

   state[1..0]       := idle0;
   RldPacker[3..0]   := 15;
   RunldPacker[3..0] := 15;
   RldFifo           := 1;
   RunldFifo         := 1;
   DMAactive          = 0;
   dmrsp             := 0;

   end;

case state[1..0] of

   idle0:       {idle states}
     begin

        dmrsp := 0;     {dma respond is always deasserted }
        DMAactive =0;   {dma active is always deasserted }

          {*****}
          { load fifo if the packer is full and the Fifo is not full }
          {*****}
         if inbound*packer_full*~FifoFull then begin
          SldFifo := 1;
          RldPacker[1]  := 1;          {if word mode}
          RldPacker[3]  := 1;          {if longword mode}
          end;

          {*****}
          { unload fifo if unpacker is empty and Fifo is not empty }
          {*****}
```

```
      if ~inbound*unpacker_emptied  then begin
         RunldPacker[1]  := 1;        {if word mode}
         RunldPacker[3]  := 1;        {if longword mode}
         end;

      if ~inbound*delayedDmrsp  then begin
         DMAactive = 1;
         OdataEn   = 1;
         end;

      next_state := idle1;
     end; {case idle0}

idle1:

     begin

     RldFifo     := 1;

     if DMAen then begin

       {*****}
       { inbound transfers }
       {*****}

      { load bytes for inbound transfers }
      if ~IOactive*inbound*dmrq*~packer_full  then begin
         DMAactive = 1;
         next_state := inb0a;
       . end

      else begin
      {*****}
      { outbound transfers }
      {*****}

      { unload bytes for outbound transfers }
      if ~IOactive*~inbound*dmrq  then begin
         if unpacker_empty*~FifoEmpty  then begin
            SunldFifo := 1;          {load packer}
            SunldPacker[0]  := 1;    {enable unpacker byte 0 onto data bus}
            DMAactive = 1;
            next_state := inb0a;
            end;
         if unpacker_empty*FifoEmpty  then begin
            next_state := idle0;
            end;
         if unldPacker[0]  then begin
             RunldPacker[0]  := 1;   {disable unpacker byte 0 onto data bus}
             SunldPacker[1]  := 1;   {enable unpacker byte 1 onto data bus}
             DMAactive = 1;
             next_state := inb0a;
             end;
         if unldPacker[1]  then begin
             RunldPacker[1]  := 1;   {disable unpacker byte 0 onto data bus}
             SunldPacker[2]  := 1;   {enable unpacker byte 1 onto data bus}
             DMAactive = 1;
             next_state := inb0a;
             end;
         if unldPacker[2]  then begin
             RunldPacker[2]  := 1;   {disable unpacker byte 0 onto data bus}
             SunldPacker[3]  := 1;   {enable unpacker byte 1 onto data bus}
             DMAactive = 1;
             next_state := inb0a;
             end;
         end {if ~IOactive, etc.}

         else {if not above}
            next_state := idle0;

       end; {else begin}

      end {if DMAen}

     else {DMAen is not true}
       next_state := idle0;

      end; {case idle1}

  inb0a:

     begin

     RunldFifo    := 1;

        DMAactive = 1;
        dmrsp     := 1;
        if inbound then begin
           if ldPacker[3..0] == 0 then
             SldPacker[0]  := 1;                { load packer byte 0 }
           if ldPacker[0] then begin
             RldPacker[0]  := 1;        { complete load packer byte 0 clock }
             SldPacker[1]  := 1;        { load packer byte 1 }
             end;
           if ldPacker[1] then begin
             RldPacker[1]  := 1;        { complete load packer byte 0 clock }
             SldPacker[2]  := 1;        { load packer byte 1 }
             end;
           if ldPacker[2] then begin
             RldPacker[2]  := 1;        { complete load packer byte 0 clock }
             SldPacker[3]  := 1;        { load packer byte 1 }
             end;
           next_state := inb0b;
        end {if inbound}
        else begin {outbound}
           OdataEn    = 1;
           next_state := inb0b;
           end;
     end; {case inb0a}

  inb0b:
       begin                            { Finish transfer regardless of DMAen}
        DMAactive = 1;
        if ~inbound then
           OdataEn    = 1;
        dmrsp     := 1;
        next_state := idle0;
        end; {case inb0b}

  end; { case state }

end.
```

HEWLETT PACKARD

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Mon Sep 18 17:19:17 1989
PALASM Description for SCSIDMASM
/SLDFIFO /SUNLDFIFO /SLDPACKER[3] /SLDPACKER[2] /SLDPACKER[1]
/SLDPACKER[0] /SUNLDPACKER[3] /SUNLDPACKER[2] /SUNLDPACKER[1] /SUNLDPACKER[0] /RST

;; PAL Attributes File is
;; Using structure info for "ASIC"

DMAACTIVE = /STATE[0] * /INBOUND * DELAYEDDMRSP
  + STATE[1]
  + DMAEN * DMRQ * STATE[0] * /IOACTIVE * /FIFOEMPTY * /INBOUND * /UNLDPACKER[3]
  + DMAEN * DMRQ * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[2]
  + DMAEN * DMRQ * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[1]
  + DMAEN * DMRQ * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[0]
  + DMAEN * DMRQ * /WORD * STATE[0] * /IOACTIVE * INBOUND * /LDPACKER[3]
  + DMAEN * DMRQ * WORD * STATE[0] * /IOACTIVE * INBOUND * /LDPACKER[1]

DMRSP := STATE[1]

ODATAEN = /STATE[0] * /INBOUND * DELAYEDDMRSP
  + STATE[1] * /INBOUND

RLDFIFO := /STATE[1] * STATE[0]
  + RST

RLDPACKER[0]  := STATE[1] * STATE[0] * INBOUND * LDPACKER[0]
  + RST

RLDPACKER[1]  := STATE[1] * STATE[0] * INBOUND * LDPACKER[1]
  + /WORD * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[3]
  + WORD * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + RST

RLDPACKER[2]  := STATE[1] * STATE[0] * INBOUND * LDPACKER[2]
  + RST

RLDPACKER[3]  := /WORD * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[3]
  + WORD * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + RST

RUNLDFIFO := STATE[1] * STATE[0]
  + RST

RUNLDPACKER[0]  := DMAEN * DMRQ * /STATE[1] * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[0]
  + RST

RUNLDPACKER[1]  := DMAEN * DMRQ * /STATE[1] * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[1]
  + /WORD * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[3]
  + WORD * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + RST

RUNLDPACKER[2]  := DMAEN * DMRQ * /STATE[1] * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[2]
  + RST

RUNLDPACKER[3]  := /WORD * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[3]
  + WORD * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + RST

/SLDFIFO := /INBOUND
  + FIFOFULL
  + STATE[0]
  + STATE[1]
  + /WORD * /LDPACKER[3]
  + WORD * /LDPACKER[1]

/SLDPACKER[0]  := /STATE[1]
  + /STATE[0]
  + /INBOUND
  + LDPACKER[3]
  + LDPACKER[2]
  + LDPACKER[1]
  + LDPACKER[0]

/SLDPACKER[1]  := /STATE[1]
  + /STATE[0]
  + /INBOUND
  + /LDPACKER[0]

/SLDPACKER[2]  := /STATE[1]
  + /STATE[0]
  + /INBOUND
  + /LDPACKER[1]

/SLDPACKER[3]  := /STATE[1]
  + /STATE[0]
  + /INBOUND
  + /LDPACKER[2]

STATE[0]  := DMAEN * DMRQ * /STATE[1] * /IOACTIVE * /FIFOEMPTY * /INBOUND * /UNLDPACKER[3]
  + DMAEN * DMRQ * /STATE[1] * /IOACTIVE * /INBOUND * UNLDPACKER[2]
  + DMAEN * DMRQ * /STATE[1] * /IOACTIVE * /INBOUND * UNLDPACKER[1]
  + DMAEN * DMRQ * /STATE[1] * /IOACTIVE * /INBOUND * UNLDPACKER[0]
  + DMAEN * DMRQ * /WORD * /STATE[1] * /IOACTIVE * INBOUND * /LDPACKER[3]
  + DMAEN * DMRQ * WORD * /STATE[1] * /IOACTIVE * INBOUND * /LDPACKER[1]
  + /STATE[1] * /STATE[0]

STATE[1]  := DMAEN * DMRQ * STATE[0] * /IOACTIVE * /FIFOEMPTY * /INBOUND * /UNLDPACKER[3]
  + DMAEN * DMRQ * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[2]
  + DMAEN * DMRQ * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[1]
  + DMAEN * DMRQ * STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[0]
  + STATE[1] * STATE[0]
  + DMAEN * DMRQ * /WORD * STATE[0] * /IOACTIVE * INBOUND * /LDPACKER[3]
  + DMAEN * DMRQ * WORD * STATE[0] * /IOACTIVE * INBOUND * /LDPACKER[1]

/SUNLDFIFO  := /DMAEN
  + /DMRQ
  + STATE[1]
  + /STATE[0]
  + IOACTIVE
  + FIFOEMPTY
  + INBOUND
  + UNLDPACKER[3]
  + UNLDPACKER[2]
  + UNLDPACKER[1]
  + UNLDPACKER[0]

/SUNLDPACKER[0]  := /DMAEN
  + /DMRQ
  + STATE[1]
  + /STATE[0]
  + IOACTIVE
  + FIFOEMPTY
  + INBOUND
  + UNLDPACKER[3]
  + UNLDPACKER[2]
  + UNLDPACKER[1]
  + UNLDPACKER[0]

/SUNLDPACKER[1]  := /DMAEN
  + /DMRQ
  + STATE[1]
  + /STATE[0]
  + IOACTIVE
  + INBOUND
  + /UNLDPACKER[0]

/SUNLDPACKER[2]  := /DMAEN
  + /DMRQ
  + STATE[1]
  + /STATE[0]
  + IOACTIVE
  + INBOUND
  + /UNLDPACKER[1]

/SUNLDPACKER[3]  := /DMAEN
  + /DMRQ
  + STATE[1]
  + /STATE[0]
  + IOACTIVE
  + INBOUND
  + /UNLDPACKER[2]
```

5.4 HS-HPIB Register Access Machine

```
pal HPIBdioSM;

{
        PAL Designer HPIB DIO State Machine program.

        This file generates the Next State Equations and Output
        equations for DIO register accesses of the HS-HPIB portion
        of Nikki.

        Don Soltis      v0.1    2/6/89
                Initial equations
        Don Soltis      v0.2    2/18/89
                Added extra state between ioend and dtack

}

input           CS, write, DMAactive, rst, a[5..0],
                hiddenEn, state[2..0],
                rstWait,        {500ns POW delay}
                ioend;

output          state[2..0],
                IOactive,
                writeReg1, writeReg3, writeReg5, writeReg7,  { DIO reg writes }
                readReg1,  readReg3, readReg5,  readReg7,   { DIO reg reads  }
                dtack, OdioEn1,
                OhpibEn,
                MedusaCs,
                iogo;


pal_type                'ASIC';

{ ONLY the active low outputs need to be specified here for logic reduction }
{ use the psm version of pd2ddl for proper generation }

state_vars              state[2..0];

macro idle;             begin  0  end;
macro s0;               begin  1  end;
macro s1;               begin  4  end;
macro s2;               begin  6  end;
macro s3;               begin  2  end;

macro address;          begin a[5..0] end;

macro dio_reg;          begin
        ( a[4] == 0 )
        end;

macro MedusaReg;        begin
        ( a[4] == 1 )
        end;

macro next_state;       begin  state[2..0]  end;

Procedure enable_dio_reg;
   begin
        case a[3..1] of
           0: {reg 1}
              readReg1  = 1;
           1: {reg 3}
              readReg3  = 1;
           2: {reg 5}
              readReg5  = 1;
           3: {reg 7}
              readReg7  = 1;
           end;
        end;

Procedure write_dio_reg;
```

```
   begin
        case a[3..1] of
           0: {reg 1}
              writeReg1 = 1;
           1: {reg 3}
              writeReg3 = 1;
           2: {reg 5}
              writeReg5 = 1;
           3: {reg 7}
              writeReg7 = 1;
           end;
        end;

{*************************************
 * Main
 *************************************}
begin

if rst then begin       { Reset }

   state[2..0]      := idle;

   end;

case state[2..0] of

   idle:         {idle states}
     begin
      if CS then begin
        IOactive = 1;
        next_state := s0;
        end
      else
        next_state := idle;
      end; {in idle}

   s0:           { continue to assert IOactive, DMA has priority }
     begin
      if CS then begin
        IOactive = 1;
        next_state := s1
        end
      else
        next_state := idle;
      end; {case s0}

   s1:
     begin
      if CS then begin
         IOactive = 1;
         if DMAactive then
            next_state := s1
         else begin  {DMA is not active}

            if ~write then OdioEn1 = 1;

            if MedusaReg then begin   {Medusa reg access}
               if write then OhpibEn = 1;
               MedusaCs = 1;
               iogo = 1;                {IOGO should be delayed 1 clock}
               end
            else begin {dio_reg}
               if ~write then begin { read }
                  enable_dio_reg;
                  end
               else {write}
                  if a[3..1] = 0 then writeReg1 = 1; {Reset}
            end; {else dio_reg}
            if MedusaReg and ~ioend then
               next_state := s1         { wait for ioend }
            else
               next_state := s2;
            end {else DMA is not active}
         end {if CS}
      else {~CS}
        next_state := idle;

      end; {case s1}

   s2:                                  { Give data time to settle }
     begin
      IOactive = 1;
      if dio_reg then
         if write then begin
            write_dio_reg;              { latch data }
            if a[3..1] = 0 then         {reset, extend}
               if rstWait then
                  next_state := s2      {wait for 500ns}
               else
                  next_state := s3
            else                        {not reset register}
               next_state := s3;
            end
         else begin
            OdioEn1 = 1;
            enable_dio_reg;
            next_state := s3;
            end
      else {MedusaReg} begin
         if ~write then OdioEn1 = 1;
         next_state := s3;
         end;
      end; {case s2}

   s3:
     begin
      dtack = 1;
      IOactive = 1;
      if dio_reg then begin
         {if write then}
            {write_dio_reg}             { latch data }
         {else begin}
         if ~write then begin
            OdioEn1 = 1;
            enable_dio_reg;
            end;
         end
      else {MedusaReg}
         if ~write then OdioEn1 = 1;

      if CS or ioend then
         next_state := s3
      else
         next_state := idle;
      end; {case s3}

   end;
end.
```

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Fri Apr 28 13:24:33 1989
PALASM Description for HPIBDIOSM


;; PAL Attributes File is
;; Using structure info for "ASIC"

DTACK = /STATE[2] * STATE[1] * /STATE[0]

IOACTIVE = CS * /STATE[0]
  + STATE[1] * /STATE[0]
  + CS * /STATE[2] * /STATE[1]

IOGO = CS * A[4] * STATE[2] * /STATE[1] * /STATE[0] * /DMAACTIVE

MEDUSACS = CS * A[4] * STATE[2] * /STATE[1] * /STATE[0] * /DMAACTIVE

ODIOEN1 = /WRITE * STATE[1] * /STATE[0]
  + CS * /WRITE * STATE[2] * /STATE[0] * /DMAACTIVE

OHPIBEN = CS * WRITE * A[4] * STATE[2] * /STATE[1] * /STATE[0] * /DMAACTIVE

READREG1 = /WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * /STATE[0] * /DMAACTIVE

READREG3 = /WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[2] * /STATE[0] * /DMAACTIVE

READREG5 = /WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[2] * /STATE[0] * /DMAACTIVE

READREG7 = /WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * /STATE[0] * /DMAACTIVE

STATE[0] := CS * /STATE[2] * /STATE[1] * /STATE[0]

STATE[1] := CS * STATE[1] * /STATE[0]
  + STATE[1] * /STATE[0] * IOEND
  + CS * STATE[2] * /STATE[0] * IOEND * /DMAACTIVE
  + STATE[2] * STATE[1] * /STATE[0]
  + CS * /A[4] * STATE[2] * /STATE[0] * /DMAACTIVE

STATE[2] := WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * STATE[1] * /STATE[0] * RSTWAIT
  + CS * STATE[2] * /STATE[1] * /STATE[0]
  + CS * /STATE[2] * /STATE[1] * STATE[0]

WRITEREG1 = CS * WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * /STATE[0] * /DMAACTIVE
  + WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * STATE[1] * /STATE[0]

WRITEREG3 = WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[2] * STATE[1] * /STATE[0]

WRITEREG5 = WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[2] * STATE[1] * /STATE[0]

WRITEREG7 = WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * STATE[1] * /STATE[0]
```

5.5 HS-HPIB FIFO-Medusa DMA Machine

```
pal HPIBdmaSM;

{
        PAL Designer HPIB DMA State Machine program.

        This file generates the Next State Equations and Output
        equations for the DMA half of the HPIB portion of the interface.

        Don Soltis      v0.1    1/31/89
                Initial equations

}

input                   DMAen,IOactive,FifoFull,FifoEmpty,inbound,
                        done,rst,dmrq,word,byte,
                         ldPacker[3..0],unldPacker[3..0],
                         state[1..0],
                         ioend,b[1..0],EOT;

output                   state[1..0],
                         SldFifo,                 { set ldFifo }
                         RldFifo,                 { reset ldFifo }
                         SunldFifo,               { set unldFifo }
                         RunldFifo,               { reset unldFifo }
                         SldPacker[3..0],         { set ldPacker }
                         RldPacker[3..0],         { reset ldPacker }
                         SunldPacker[3..0],       { set unldPacker }
                         RunldPacker[3..0],       { reset unldPacker }
                         DMAactive,
                         OdataEn,                 { HPIB data output enable }
                         hpibCs,iogo,
                                          {I can only set the odd bits,  }
                                          {a reset or write to reg 7(?)  }
                                          {is the only way to clear      }
                         Sodd0,                   { True if last inbound xfer }
                         Sodd1,                   { was not full width }
                                          { odd1 odd0  residual }
                                          {   0    0    none, full xfer }
                                          {   0    1    one valid byte in Fifo}
                                          {   1    0    two valid bytes in Fifo}
                                          {   1    1    three valid bytes in Fifo}
                                                  {******************}
                         intr;                    { interrupt output }
                                                  { when asserted, beginning }
                                                  { of interrupt.    }

pal_type                 'ASIC';

{ ONLY the active low outputs need to be specified here for logic reduction }
{ use the psm version of pd2ddl for proper generation }

pal_pin_order            ~SldFifo,~SunldFifo,
                         ~SldPacker[3..0],
                         ~SunldPacker[3..0],
                         ~Sodd1,~Sodd0;

state_vars               state[1..0];

macro idle;              begin  0  end;
macro st0;               begin  1  end;
macro st1;               begin  3  end; {Inbound/outbound, byte 0}
macro st2;               begin  2  end; {Inbound/outbound, byte 0}

macro next_state;        begin  state[1..0]  end;

macro packer_full;
        begin
        (      (ldPacker[3]*~word*~byte)  {longword}
            or (ldPacker[1]*word*~byte)   {word}
            or (ldPacker[1]*byte)         {byte}
            )
        end;

macro packer_empty;
        begin
        (
          ~ldPacker[3]*~ldPacker[2]*~ldPacker[1]*~ldPacker[0]
         )
        end;

macro unpacker_empty; begin
        (
          ~unldPacker[3]*~unldPacker[2]*~unldPacker[1]*~unldPacker[0]
         )
        end;

macro unpacker_emptied; begin
        (
              (unldPacker[3]*~word*~byte)
           or (unldPacker[1]*word*~byte)
           or (unldPacker[1]*byte)
           )
        end;

begin

if rst then begin       { Reset }

   state[1..0]      := idle;
   RldPacker[3..0]   := 15;
   RunldPacker[3..0] := 15;
   RldFifo           := 1;
   RunldFifo         := 1;
   DMAactive          = 0;
   hpibCs            := 0;
   iogo              := 0;

   end;

case state[1..0] of

   idle:        {idle states}
     begin

        hpibCs := 0;
        iogo   := 0;
        DMAactive =0;   {dma active is always deasserted }

        if DMAen then begin  { Make sure DMA is enabled }


           {*****}
           { load fifo if the packer is full and the Fifo is not full }
           {*****}
         if inbound*packer_full*~FifoFull  then begin
           SldFifo := 1;
           RldPacker[0] := 1;           {if byte mode}
           RldPacker[1] := 1;           {if word mode}
           RldPacker[3] := 1;           {if longword mode}
           next_state := idle;
           end;

           {*****}
           { reset the ldFifo pulse when done }
           {*****}
         if inbound*packer_empty  then
           RldFifo = 1;

           {*****}
           { unload fifo if unpacker is empty and Fifo is not empty }
           {*****}
         if ~inbound*unpacker_emptied  then begin
           RunldPacker[0] := 1; {if word mode}
           RunldPacker[1] := 1; {if byte mode}
           RunldPacker[3] := 1; {if longword mode}
           next_state := idle;
           end;

         {*****}
         { inbound transfers }
         {*****}

         { load bytes for inbound transfers }
         {********** don't need done in below **********}
         {if ~IOactive*inbound*dmrq*~packer_full*~done  then begin}
         {*********************************************}
         if ~IOactive*inbound*dmrq*~packer_full  then begin
            next_state := st0;
            end

         else begin
         {*****}
         { outbound transfers }
         {*****}

         { unload bytes for outbound transfers }
         if ~IOactive*~inbound*dmrq  then begin
            if unpacker_empty*~FifoEmpty  then begin
               SunldFifo := 1;          {load packer}
```

```
              if ~byte then
                 SunldPacker[0]  := 1   {enable unpacker byte 0 onto data bus}
              else
                 SunldPacker[1]  := 1;  {enable unpacker byte 1 onto data bus}
              next_state := st0;
              end;
           if unpacker_empty*FifoEmpty  then begin
              next_state := idle;
              end;
           if unldPacker[0] then begin
              RunldPacker[0]  := 1;     {disable unpacker byte 0 onto data bus}
              SunldPacker[1]  := 1;     {enable unpacker byte 1 onto data bus}
              next_state := st0;
              end;
           if unldPacker[1]*~word*~byte  then begin
              RunldPacker[1]  := 1;     {disable unpacker byte 0 onto data bus}
              SunldPacker[2]  := 1;     {enable unpacker byte 1 onto data bus}
              next_state := st0;
              end;
           if unldPacker[2] then begin
              RunldPacker[2]  := 1;     {disable unpacker byte 0 onto data bus}
              SunldPacker[3]  := 1;     {enable unpacker byte 1 onto data bus}
              next_state := st0;
              end;
           end {if ~IOactive, etc.}

           else {if not above}
              next_state := idle;

         end; {else begin}

        end {if DMAen}

       else {DMAen is not true}
         next_state := idle;

       end; {case idle}

 st0:

    begin

    RunldFifo    := 1;        {finish FIFO pulse}

    if DMAen then begin

       DMAactive = 1;
       hpibCs = 1;
       if inbound then begin
          iogo = 1;
          if ioend then              { wait for ioend }
             next_state := st1
          else
             next_state := st0;
          end
       else begin {outbound}
          OdataEn = 1;
          next_state := st1;
          end;

    end {if DMAen}
    else begin
       next_state := idle;
       end;

    end; {case inb0a}

 st1:
      begin                          { Finish transfer regardless of DMAen}
       DMAactive = 1;
       hpibCs = 1;
       iogo = 1;
       if inbound then begin
          if ldPacker[3..0] == 0 then
            if ~byte then
               SldPacker[0]  := 1            { load packer byte 0 }
            else
```

```
              SldPacker[1] := 1;            { load packer byte 0 }
         if ldPacker[0] then begin
           RldPacker[0] := 1;        { complete load packer byte 0 clock }
           SldPacker[1] := 1;        { load packer byte 1 }
           end;
         if ldPacker[1] then begin
           RldPacker[1] := 1;        { complete load packer byte 0 clock }
           SldPacker[2] := 1;        { load packer byte 1 }
           end;
         if ldPacker[2] then begin
           RldPacker[2] := 1;        { complete load packer byte 0 clock }
           SldPacker[3] := 1;        { load packer byte 1 }
           end;
         next_state := st2;
         end  {if inbound}
      else begin{ outbound }
         iogo = 1;
         OdataEn = 1;
         if ioend then               { wait for ioend }
            next_state := st2
         else
            next_state := st1;
         end
      end; {case inb0b}

st2:
     begin                           { Finish transfer regardless of DMAen}
      DMAactive = 1;
      if ~inbound then begin
         OdataEn = 1;
         end; {if outbound}
      if ioend then
         next_state := st2
      else
         if ~inbound*unpacker_emptied*done  then
            intr = 1
         else
            {*****}
            { Medusa gave an EOT before filling packer, }
            { set odd0,1 depending on how many bytes are valid }
            { in last fifo entry }
            {*****}
         if inbound*~packer_full*EOT  then begin      {odd transfer}
           if ldPacker[0] = 1 then
              Sodd0 = 1;
           if ldPacker[1] = 1 then
              Sodd1 = 1;
           if ldPacker[2] = 1 then begin
              Sodd0 = 1;
              Sodd1 = 1;
              end;
           {odd = 1;}                {Set Odd}
           SldFifo := 1;             {Load Fifo anyhow}
           RldPacker[0] := 1;                {Clear Packer}
           RldPacker[1] := 1;
           RldPacker[2] := 1;
           RldPacker[3] := 1;
           end;
         next_state := idle;
      end;


end; { case state }

end.
```

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Thu Apr 27 20:58:56 1989
PALASM Description for HPIBDMASM
/SLDFIFO /SUNLDFIFO /SLDPACKER[3] /SLDPACKER[2] /SLDPACKER[1] /SLDPACKER[0]
/SUNLDPACKER[3] /SUNLDPACKER[2] /SUNLDPACKER[1] /SUNLDPACKER[0] /SODD1 /SODD0

;; PAL Attributes File is
;; Using structure info for "ASIC"

DMAACTIVE = STATE[1]
  + DMAEN * STATE[0]

HPIBCS = DMAEN * STATE[0]
  + STATE[1] * STATE[0]

INTR = DONE * WORD * STATE[1] * /STATE[0] * /IOEND * /INBOUND * UNLDPACKER[1]
  + DONE * /WORD * /BYTE * STATE[1] * /STATE[0] * /IOEND * /INBOUND * UNLDPACKER[3]
  + DONE * BYTE * STATE[1] * /STATE[0] * /IOEND * /INBOUND * UNLDPACKER[1]

IOGO = DMAEN * STATE[0] * INBOUND
  + STATE[1] * STATE[0]

ODATAEN = STATE[1] * /INBOUND
  + DMAEN * STATE[0] * /INBOUND

RLDFIFO = DMAEN * /STATE[1] * /STATE[0] * INBOUND * /LDPACKER[3] * /LDPACKER[2] * /LDPACKER[1] * /LDPACKER[0]
  + RST

RLDPACKER[0] := BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + /WORD * /BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[3]
  + WORD * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + STATE[1] * STATE[0] * INBOUND * LDPACKER[0]
  + DMAEN * WORD * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + DMAEN * /WORD * /BYTE * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[3]
  + DMAEN * BYTE * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + RST

RLDPACKER[1] := BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + /WORD * /BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[3]
  + WORD * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + STATE[1] * STATE[0] * INBOUND * LDPACKER[1]
  + DMAEN * WORD * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + DMAEN * /WORD * /BYTE * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[3]
  + DMAEN * BYTE * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + RST

RLDPACKER[2] := BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + /WORD * /BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[3]
  + WORD * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + STATE[1] * STATE[0] * INBOUND * LDPACKER[2]
  + RST

RLDPACKER[3] := BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + /WORD * /BYTE * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[3]
  + WORD * STATE[1] * /STATE[0] * /IOEND * EOT * INBOUND * /LDPACKER[1]
  + DMAEN * WORD * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + DMAEN * /WORD * /BYTE * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[3]
  + DMAEN * BYTE * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND * LDPACKER[1]
  + RST

RUNLDFIFO := /STATE[1] * STATE[0]
  + RST

RUNLDPACKER[0] := DMAEN * DMRQ * /STATE[1] * /STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[0]
  + DMAEN * WORD * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + DMAEN * /WORD * /BYTE * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[3]
  + DMAEN * BYTE * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + RST

RUNLDPACKER[1] := DMAEN * DMRQ * /STATE[1] * /STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[1]
  + DMAEN * WORD * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + DMAEN * /WORD * /BYTE * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[3]
  + DMAEN * BYTE * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + RST

RUNLDPACKER[2] := DMAEN * DMRQ * /STATE[1] * /STATE[0] * /IOACTIVE * /INBOUND * UNLDPACKER[2]
```

```
  + RST

RUNLDPACKER[3]  := DMAEN * WORD * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + DMAEN * /WORD * /BYTE * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[3]
  + DMAEN * BYTE * /STATE[1] * /STATE[0] * /INBOUND * UNLDPACKER[1]
  + RST

/SLDFIFO := /WORD * /BYTE * STATE[1] * LDPACKER[3]
  + BYTE * /STATE[1] * /LDPACKER[1]
  + BYTE * STATE[1] * LDPACKER[1]
  + /DMAEN * /STATE[1]
  + /WORD * /BYTE * /STATE[1] * /LDPACKER[3]
  + WORD * STATE[1] * LDPACKER[1]
  + STATE[1] * /EOT
  + STATE[1] * IOEND
  + STATE[0]
  + /STATE[1] * FIFOFULL
  + /INBOUND
  + WORD * /STATE[1] * /LDPACKER[1]

/SLDPACKER[0]  := BYTE
  + /STATE[1]
  + /STATE[0]
  + /INBOUND
  + LDPACKER[3]
  + LDPACKER[2]
  + LDPACKER[1]
  + LDPACKER[0]

/SLDPACKER[1]  := /STATE[1]
  + /STATE[0]
  + /INBOUND
  + /BYTE * /LDPACKER[0]
  + LDPACKER[3] * /LDPACKER[0]
  + LDPACKER[2] * /LDPACKER[0]
  + LDPACKER[1] * /LDPACKER[0]

/SLDPACKER[2]  := /STATE[1]
  + /STATE[0]
  + /INBOUND
  + /LDPACKER[1]

/SLDPACKER[3]  := /STATE[1]
  + /STATE[0]
  + /INBOUND
  + /LDPACKER[2]

/SODD0 = BYTE * LDPACKER[1]
  + /WORD * /BYTE * LDPACKER[3]
  + /STATE[1]
  + STATE[0]
  + IOEND
  + /EOT
  + /INBOUND
  + WORD * LDPACKER[1]
  + /LDPACKER[2] * /LDPACKER[0]

/SODD1 = /LDPACKER[2] * /LDPACKER[1]
  + /STATE[1]
  + STATE[0]
  + IOEND
  + /EOT
  + /INBOUND
  + /WORD * /BYTE * LDPACKER[3]
  + WORD * LDPACKER[1]
  + BYTE * LDPACKER[1]

STATE[0]  := STATE[1] * STATE[0] * /IOEND * /INBOUND
  + DMAEN * DMRQ * /STATE[1] * /IOACTIVE * /FIFOEMPTY * /INBOUND * /UNLDPACKER[3] * /UNLDPACKER[1]
  + DMAEN * DMRQ * /STATE[1] * /IOACTIVE * /INBOUND * UNLDPACKER[2]
  + DMAEN * DMRQ * /WORD * /BYTE * /STATE[1] * /IOACTIVE * /INBOUND * UNLDPACKER[1]
  + DMAEN * DMRQ * /STATE[1] * /IOACTIVE * /INBOUND * UNLDPACKER[0]
  + DMAEN * /STATE[1] * STATE[0]
  + DMAEN * DMRQ * BYTE * /STATE[1] * /IOACTIVE * INBOUND * /LDPACKER[1]
  + DMAEN * DMRQ * /WORD * /BYTE * /STATE[1] * /IOACTIVE * INBOUND * /LDPACKER[3]
  + DMAEN * DMRQ * WORD * /STATE[1] * /IOACTIVE * INBOUND * /LDPACKER[1]

```

```
STATE[1]  := STATE[1] * IOEND
  + DMAEN * STATE[0] * /INBOUND
  + STATE[1] * STATE[0]
  + DMAEN * STATE[0] * IOEND

/SUNLDFIFO  := /DMAEN
  + /DMRQ
  + STATE[1]
  + STATE[0]
  + IOACTIVE
  + FIFOEMPTY
  + INBOUND
  + UNLDPACKER[3]
  + UNLDPACKER[2]
  + UNLDPACKER[1]
  + UNLDPACKER[0]

/SUNLDPACKER[0]  := /DMAEN
  + /DMRQ
  + BYTE
  + STATE[1]
  + STATE[0]
  + IOACTIVE
  + FIFOEMPTY
  + INBOUND
  + UNLDPACKER[3]
  + UNLDPACKER[2]
  + UNLDPACKER[1]
  + UNLDPACKER[0]

/SUNLDPACKER[1]  := /DMAEN
  + /DMRQ
  + STATE[1]
  + STATE[0]
  + IOACTIVE
  + INBOUND
  + /BYTE * /UNLDPACKER[0]
  + FIFOEMPTY * /UNLDPACKER[0]
  + UNLDPACKER[3]  * /UNLDPACKER[0]
  + UNLDPACKER[2]  * /UNLDPACKER[0]
  + UNLDPACKER[1]  * /UNLDPACKER[0]

/SUNLDPACKER[2]  := /DMAEN
  + /DMRQ
  + WORD
  + BYTE
  + STATE[1]
  + STATE[0]
  + IOACTIVE
  + INBOUND
  + /UNLDPACKER[1]

/SUNLDPACKER[3]  := /DMAEN
  + /DMRQ
  + STATE[1]
  + STATE[0]
  + IOACTIVE
  + INBOUND
  + /UNLDPACKER[2]
```

5.6 LS-HPIB Register Access, DMA, and Parallel poll Machine

```
pal LShpibSM;

{
        PAL Designer low speed HPIB State Machine program.

        This file generates the Next State Equations and Output
        equations for DIO register accesses, parallel polling and
        DMA sequence of the LS-HPIB portion of Nikki.

        Don Soltis      v0.1    2/19/89
                Initial equations

}

input           CS, write, rst, a[5..0],
                state[2..0],
                bg,             {Bus grant from arbiter}
                CncsBr,         {Bus request from Centronics}
                RS232cs,        {RS-232 Chip select}
                inppoll,        {in parallel poll}
                dtackin,        {for DMA use}
                DMAen,          {DMA enabled}
                dmack,          {DMA acknowledge}
                accrq,          {9914 DMA request}
                ppollen,        {parallel poll enable}
                match           {ppoll match}
                ;

output          state[2..0],
                writeReg1, writeReg3, writeReg5,
                writeReg7, writeReg9,           { DIO reg writes }

                readReg1,  readReg3, readReg5,
                readReg7, readReg9,             { DIO reg reads  }

                dtack, OdioEn1,
                dmrdy,
                a9914cs, we, dbin, accgr,       {9914 signals}
                busReq,         {bus request to arbiter}
                Sinppoll,       {Set 'in ppoll' flip-flop}
                Rinppoll,       {Reset 'in ppoll' flip-flop}
                ppollir,
                OlsHpibEn,
                ls2dio
                ;


pal_type                    'ASIC';

{ ONLY the active low outputs need to be specified here for logic reduction }
{ use the psm version of pd2ddl for proper generation }

state_vars                  state[2..0];

macro idle;                 begin  0  end;
macro s0;                   begin  1  end;
macro s1;                   begin  3  end;
macro s2;                   begin  2  end;
macro s3;                   begin  6  end;
macro s4;                   begin  7  end;
macro s5;                   begin  5  end;
macro s6;                   begin  4  end;

{macro idle;                begin  0  end;
macro s0;                   begin  4  end;
macro s1;                   begin  7  end;
macro s2;                   begin  3  end;
macro s3;                   begin  2  end;
macro s4;                   begin  6  end;
macro s5;                   begin  1  end;
macro s6;                   begin  5  end;}

macro address;              begin a[5..0] end;

macro dio_reg;              begin
        ( a[4] == 0 )
        end;

macro a9914Reg; begin
        ( a[4] == 1 )
        end;

macro next_state;           begin  state[2..0]  end;

Procedure enable_dio_reg;
   begin
        case a[3..1] of
           0: {reg 1}
               readReg1 = 1;
           1: {reg 3}
              readReg3 = 1;
           2: {reg 5}
              readReg5 = 1;
           3: {reg 7}
              readReg7  = 1;
           end;
        if a[3] = 1 then readReg9  = 1;
        end;

Procedure write_dio_reg;
   begin
        case a[3..1] of
           0: {reg 1}
              writeReg1 = 1;
           1: {reg 3}
              writeReg3 = 1;
           2: {reg 5}
              writeReg5 = 1;
           3: {reg 7}
              writeReg7 = 1;
           end;
        if a[3] = 1 then writeReg9  = 1;
        end;

        { ******** }
        { hold signals for DIO register transaction }
        { ******** }
Procedure do_reg_io;
   begin
      if (~write and ~inppoll) then OdioEn1 = 1;

      if inppoll then begin
         a9914cs = 1;
         dbin = 1;
         end
      else begin {~inppoll}
         if a9914Reg then begin   {a 9914 reg access, or p. poll}
            if write then begin
               OlsHpibEn = 1;
               a9914cs = 1;
               dbin = 0;
               if state[2..0] = s1 then
                  we = 0
               else
                  we = 1;
               end {if write}
            else   {read 9914 reg} begin
               ls2dio = 1;
               a9914cs = 1;
               dbin = 1;
               end;
            end
         else begin {dio_reg}
            if ~write then begin { read }
               enable_dio_reg;
               end
            end; {else dio_reg}
         end; {else ~inppoll}
      end; {Procedure do_reg_io}

Procedure do_dma_io;
   begin
      if write then OdioEn1 = 1;        {inbound DMA}

      if ~write then begin              {outbound xfer}
         OlsHpibEn = 1;
         accGr = 1;
         dbin = 1;
         if state[2..0] = s1 then
            we = 0
         else
            we = 1;
         end {if write}
      else  {inbound} begin
         ls2dio = 1;
         accGr = 1;
         dbin = 0;
         end;
      end; {Procedure do_dma_io}

        { ********* }
        { Wait in ppoll s5 until we get a match, or we need to do a }
        { dio/dma request, or Centronics/RS-232 wants the bus }
        { ********* }
Procedure ppoll_wait;
   begin
      busReq = 1;
      if match then begin
         ppollir = 1;           {assert interrupt (set enable)}
         Rinppoll = 1;          {not in ppoll, won't come here again}
         next_state := s6;      {one more state before going to idle}
         end {match}
      else {~match} begin
         if CS or (dmack and DMAen) or RS232cs or CncsBr then begin
            Rinppoll = 1;       {Get out of ppoll}
            next_state := s6;
            end
         else {~CS, etc} begin
            a9914cs = 1;
            dbin = 1;
            next_state := s5;
            end;
         end; {else ~match}
      end; {Procedure ppoll_wait}
{***************************************
 * Main
 ***************************************}
begin

if rst then begin      { Reset }

   state[2..0]      := idle;

   end;

case state[2..0] of

                { ********* }
   idle:        {idle states}
     begin
      if CS or (dmack and DMAen) then begin
         busReq = 1;
         next_state := s0;      {wait for bus}
         end
      else {nothing to do} begin
         if ppollEn and ~RS232cs then begin     {Start a ppoll}
            Sinppoll = 1;       { I'm in ppoll}
            busReq = 1;
            next_state := s0;   {wait for bus}
            end
         else
            next_state := idle;
         end; {else nothing to do}
      end; {in idle}

                { ********* }
   s0:          { wait for bg and start a cycle }
     begin
      busReq = 1;
      if bg = 0 then begin
                { ******** }
                { if we get a CS or dmack, cancel ppoll }
                { ******** }
         if inppoll and (CS or (dmack and DMAen)) then begin
            Rinppoll = 1;
            next_state := s0;
            end
         else
            next_state := s0;
         end
      else {bg = 1} begin
               {***********}
               {* Kludge for deglitching output Enable *}
```

```
              {***********}
        if( (CS*~write) or (dmack*write) ) then begin
           OdioEn1 = 1;
           if( ~a9914reg and CS ) then
              ls2dio = 0
           else
              ls2dio = 1;
           end;
        if inppoll then begin
           a9914cs = 1;
           dbin = 1;
           end;
        next_state := s1;      {go for it}
        end;
     end; {case s0}

              { ********* }
  s1:         { Set up 9914 signals }
    begin
     busReq = 1;
     if CS or inppoll then begin
        do_reg_io;
        next_state := s2;
        end
     else {~CS} begin
       if (dmack and DMAen and ~inppoll) then begin
         do_dma_io;
         next_state := s2;
         end
       else
          next_state := idle;          { Shouldn't be here }
       end; {else ~CS}
     end; {case s1}

              { ********* }
  s2:         { Give data time to settle }
    begin
     busReq = 1;
     if CS or inppoll then begin
        do_reg_io;
        next_state := s3;
        end
     else {~CS} begin
       if (dmack and DMAen and ~inppoll) then begin
         do_dma_io;
         next_state := s3;
         end
       else
          next_state := idle;          { Shouldn't be here }
       end; {else ~CS}
     end; {case s2}

              { ********* }
  s3:         { Continue to wait }
    begin
     busReq = 1;
     if CS or inppoll then begin
        do_reg_io;
        next_state := s4;
        end
     else {~CS} begin
       if (dmack and DMAen and ~inppoll) then begin
         do_dma_io;
         if ~write*~dtackin then
            next_state := s3           {Wait for DTACKIN on outbound xfers}
         else
            next_state := s4;
         end
       else
          next_state := idle;          { Shouldn't be here }
       end; {else ~CS}
     end; {case s3}

              { ********* }
  s4:         { Last state }
    begin
     busReq = 1;
```

```
      if CS or inppoll then begin
         if ~inppoll and ~a9914Reg and write then
            write_dio_reg;
         do_reg_io;
         next_state := s5;
         end
      else {~CS} begin
        if (dmack and DMAen and ~inppoll) then begin
         do_dma_io;
          next_state := s5;
          end
        else
           next_state := idle;          { Shouldn't be here }
        end; {else ~CS}
      end; {case s4}

                 { ********* }
   s5:           { finish cycle }
     begin
      busReq = 1;
      if CS and  ~inppoll then begin
         if ~write then begin
            OdioEn1 = 1;
            if a9914Reg then ls2dio = 1
            else             enable_dio_reg;
            end
         else
            if a9914Reg then OlsHpibEn = 1;
         dtack = 1;
         next_state := s5;              { wait for CS to go away }
         end
      else {~CS or inppoll} begin
        if (dmack and DMAen and ~inppoll) then begin
          dmrdy = 1;
          if write then begin {inbound}
             OdioEn1 = 1;
             ls2dio = 1;
             end
          else  {outbound}
             OlsHpibEn = 1;
          next_state := s5;             { wait for dmack to go away }
          end
        else {inppoll or we're done} begin
           if inppoll then begin
              ppoll_wait;
              end
           else {we're done}
              next_state := idle;       { Done }
           end; {inppoll or done}
        end; {else ~CS}
      end; {case s5}

   s6:
      begin
       next_state := idle;
       end; {case s6}

   end;
end.
```

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Mon May  8 15:02:42 1989
PALASM Description for LSHPIBSM


;; PAL Attributes File is
;; Using structure info for "ASIC"

A9914CS = /CS * STATE[2] * STATE[0] * /DMAEN * /MATCH * /CNCSBR * /RS232CS * INPPOLL
  + /CS * STATE[2] * STATE[0] * /DMACK * /MATCH * /CNCSBR * /RS232CS * INPPOLL
  + CS * A[4] * STATE[1]
  + STATE[1] * INPPOLL
  + /STATE[2] * STATE[0] * BG * INPPOLL

ACCGR = /CS * STATE[1] * DMAEN * DMACK * /INPPOLL

BUSREQ = STATE[0]
  + STATE[1]
  + /STATE[2] * /RS232CS * PPOLLEN
  + CS * /STATE[2]
  + /STATE[2] * DMAEN * DMACK

DBIN = /CS * STATE[2] * STATE[0] * /DMAEN * /MATCH * /CNCSBR * /RS232CS * INPPOLL
  + /CS * STATE[2] * STATE[0] * /DMACK * /MATCH * /CNCSBR * /RS232CS * INPPOLL
  + /CS * /WRITE * STATE[1] * DMAEN * DMACK
  + CS * /WRITE * A[4] * STATE[1]
  + STATE[1] * INPPOLL
  + /STATE[2] * STATE[0] * BG * INPPOLL

DMRDY = /CS * STATE[2] * /STATE[1] * STATE[0] * DMAEN * DMACK * /INPPOLL

DTACK = CS * STATE[2] * /STATE[1] * STATE[0] * /INPPOLL

LS2DIO = /CS * WRITE * STATE[2] * STATE[0] * DMAEN * DMACK * /INPPOLL
  + CS * /WRITE * A[4] * STATE[2] * STATE[0] * /INPPOLL
  + /CS * WRITE * STATE[1] * DMAEN * DMACK * /INPPOLL
  + CS * /WRITE * A[4] * STATE[1] * /INPPOLL
  + /CS * WRITE * /STATE[2] * /STATE[1] * STATE[0] * BG * DMACK
  + CS * /WRITE * A[4] * /STATE[2] * /STATE[1] * STATE[0] * BG
  + WRITE * A[4] * /STATE[2] * /STATE[1] * STATE[0] * BG * DMACK

ODIOEN1 = /CS * WRITE * STATE[2] * STATE[0] * DMAEN * DMACK * /INPPOLL
  + CS * /WRITE * STATE[2] * STATE[0] * /INPPOLL
  + /CS * WRITE * STATE[1] * DMAEN * DMACK * /INPPOLL
  + CS * /WRITE * STATE[1] * /INPPOLL
  + CS * /WRITE * /STATE[2] * /STATE[1] * STATE[0] * BG
  + WRITE * /STATE[2] * /STATE[1] * STATE[0] * BG * DMACK

OLSHPIBEN = /CS * /WRITE * STATE[2] * STATE[0] * DMAEN * DMACK * /INPPOLL
  + CS * WRITE * A[4] * STATE[2] * STATE[0] * /INPPOLL
  + /CS * /WRITE * STATE[1] * DMAEN * DMACK * /INPPOLL
  + CS * WRITE * A[4] * STATE[1] * /INPPOLL

PPOLLIR = STATE[2] * /STATE[1] * STATE[0] * MATCH * INPPOLL

READREG1 = CS * /WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * STATE[0] * /INPPOLL
  + CS * /WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /INPPOLL

READREG3 = CS * /WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[2] * STATE[0] * /INPPOLL
  + CS * /WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[1] * /INPPOLL

READREG5 = CS * /WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[2] * STATE[0] * /INPPOLL
  + CS * /WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[1] * /INPPOLL

READREG7 = CS * /WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * STATE[0] * /INPPOLL
  + CS * /WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[1] * /INPPOLL

READREG9 = CS * /WRITE * /A[4] * A[3] * STATE[2] * STATE[0] * /INPPOLL
  + CS * /WRITE * /A[4] * A[3] * STATE[1] * /INPPOLL

RINPPOLL = CS * /STATE[1] * STATE[0] * /BG * INPPOLL
  + /STATE[1] * STATE[0] * /BG * DMAEN * DMACK * INPPOLL
  + CS * STATE[2] * /STATE[1] * STATE[0] * INPPOLL
  + STATE[2] * /STATE[1] * STATE[0] * DMAEN * DMACK * INPPOLL
  + STATE[2] * /STATE[1] * STATE[0] * RS232CS * INPPOLL
  + STATE[2] * /STATE[1] * STATE[0] * CNCSBR * INPPOLL
  + STATE[2] * /STATE[1] * STATE[0] * MATCH * INPPOLL

SINPPOLL = /CS * /STATE[2] * /STATE[1] * /STATE[0] * /DMAEN * /RS232CS * PPOLLEN
  + /CS * /STATE[2] * /STATE[1] * /STATE[0] * /DMACK * /RS232CS * PPOLLEN

STATE[0] := /CS * STATE[2] * STATE[0] * /DMAEN * /MATCH * /CNCSBR * /RS232CS * INPPOLL
  + /CS * STATE[2] * STATE[0] * /DMACK * /MATCH * /CNCSBR * /RS232CS * INPPOLL
  + STATE[2] * STATE[0] * DMAEN * DMACK * /INPPOLL
  + CS * STATE[2] * STATE[0] * /INPPOLL
  + WRITE * STATE[2] * STATE[1] * DMAEN * DMACK
  + STATE[2] * STATE[1] * DMAEN * DMACK * DTACKIN
  + CS * STATE[2] * STATE[1]
  + STATE[2] * STATE[1] * INPPOLL
  + /STATE[2] * /STATE[1] * STATE[0]
  + /STATE[2] * /STATE[1] * /RS232CS * PPOLLEN
  + CS * /STATE[2] * /STATE[1]
  + /STATE[2] * /STATE[1] * DMAEN * DMACK

STATE[1] := STATE[1] * /STATE[0] * DMAEN * DMACK
  + CS * STATE[1] * /STATE[0]
  + STATE[1] * /STATE[0] * INPPOLL
  + /STATE[2] * STATE[1] * DMAEN * DMACK
  + CS * /STATE[2] * STATE[1]
  + /STATE[2] * STATE[1] * INPPOLL
  + /STATE[2] * /STATE[1] * STATE[0] * BG

STATE[2] := STATE[2] * STATE[0] * DMAEN * DMACK
  + CS * STATE[2] * STATE[0]
  + STATE[2] * STATE[0] * INPPOLL
  + STATE[1] * /STATE[0] * DMAEN * DMACK
  + CS * STATE[1] * /STATE[0]
  + STATE[1] * /STATE[0] * INPPOLL

WE = /CS * /WRITE * STATE[2] * STATE[1] * DMAEN * DMACK * /INPPOLL
  + CS * WRITE * A[4] * STATE[2] * STATE[1] * /INPPOLL
  + /CS * /WRITE * STATE[1] * /STATE[0] * DMAEN * DMACK * /INPPOLL
  + CS * WRITE * A[4] * STATE[1] * /STATE[0] * /INPPOLL

WRITEREG1 = CS * WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * STATE[1] * STATE[0] * /INPPOLL

WRITEREG3 = CS * WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[2] * STATE[1] * STATE[0] * /INPPOLL

WRITEREG5 = CS * WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[2] * STATE[1] * STATE[0] * /INPPOLL

WRITEREG7 = CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * STATE[1] * STATE[0] * /INPPOLL

WRITEREG9 = CS * WRITE * /A[4] * A[3] * STATE[2] * STATE[1] * STATE[0] * /INPPOLL
```

5.7 RS-232 Register Access Machine

```
pal RS232SM;
{
        PAL Designer RS232 control State Machine program.

        This file generates the Next State Equations and Output
        equations for the RS232 interface.

        Don Soltis      v0.1    2/27/89
                Initial equations

}
input   cs,             {rs232 CS}
        Nwrite,         {High true read/Low true write}
        reset,          {High True reset signal}
        st[3..0],       {Output Terms Fed Back}
        ior,
        iow,
        a[4],
        a[1],
        busgr;

output  ior,            {Low true IO Read}
        iow,            {Low true IO Write}
        st[3..0],       {State Variables}
        dtack,          {Low True Data Transfer Acknowledge}
        ReadReg1,ReadReg3,      {read strobes}
        WriteReg1,WriteReg3,    {write strobes}
        Sbusrq,Rbusrq,
        oEXTdataEn,
        OdioData1,
        ls2dio,
        uartcs;

PAL_TYPE 'ASIC';

STATE_VARS st[3], st[2], st[1], st[0];

Macro goto;
begin
  st[3..0] :=
end;

Macro state;
begin
  st[3..0]
end;

Macro startstate;       begin 0 end;
Macro s0;               begin 8 end;
Macro s1;               begin 4 end;
Macro s2;               begin 1 end;
Macro s3;               begin 9 end;
Macro s4;               begin 13 end;
Macro s5;               begin 5 end;
Macro s6;               begin 3 end;
Macro s7;               begin 11 end;
Macro s8;               begin 15 end;
Macro s9;               begin 7 end;
Macro s10;              begin 12 end;
Macro s11;              begin 10 end;
Macro s12;              begin 2 end;
Macro s13;              begin 14 end;
Macro s14;              begin 6 end;

macro an8250reg;        begin a[4] end;
macro aDio_reg;         begin ~a[4] end;

BEGIN
case state of

startstate:  {Async. reset put the circuit here}
        begin
          if cs and an8250reg then begin
             Sbusrq = 1;                {ask for bus if cs}
             goto startstate;
             end;
          if cs and aDio_reg then       {register xfer}
             goto s0;
          if cs and an8250reg and busgr then    {start 8250 xfer}
             goto s0;                   { have bus, will travel }
          if ~cs then begin
             Rbusrq = 1;                { no longer request }
             goto startstate;
             end;
          end;

s0:  begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                 {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
            end
          else
           begin
             if an8250reg then begin
                UartCs = 1;
                if Nwrite = 0 then {write}
                   oEXTdataEn;
```

HEWLETT PACKARD

```
                goto s1;
                end
             else begin
                if Nwrite = 1 then begin
                   if a[1] = 0 then
                      readReg1 = 1
                   else
                      readReg3 = 1;
                   {OdioData1 = 1;}
                   end;
                goto s1;
                end;
             end;
        end;

s1: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                   {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
            end
          else begin
             if an8250reg then begin
                UartCs = 1;
                if Nwrite = 0 then {write}
                   oEXTdataEn
                else {READ} begin
                   OdioData1 = 1;
                   ls2dio = 1;
                   end;
                goto s2;
                end
             else begin
                if Nwrite = 1 then begin
                   if a[1] = 0 then
                      readReg1 = 1
                   else
                      readReg3 = 1;
                   OdioData1 = 1;
                   end;
                goto s2;
                end;
             end;
           end;

s2: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                   {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
          end
          else
          begin
             if an8250reg then begin
                UartCs = 1;
                if Nwrite = 0 then
                   oEXTdataEn
                else {READ} begin
                   OdioData1 = 1;
                   ls2dio = 1;
                   end;
                goto s3;
                end
             else begin
                if Nwrite = 1 then begin
                   if a[1] = 0 then
                      readReg1 = 1
                   else
                      readReg3 = 1;
                   OdioData1 = 1;
                   end;
```

HEWLETT PACKARD

```
                goto s3;
                end;
             end;
           end;

s3: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
          end
          else
          begin
             if an8250reg then begin
                if Nwrite = 0 then {write}
                   oEXTdataEn
                else {READ} begin
                   OdioData1 = 1;
                   ls2dio = 1;
                   ior = 1;
                   end;
                UartCs = 1;
                goto s4;
                end
             else begin
                if Nwrite = 1 then begin
                   if a[1] = 0 then
                      readReg1 = 1
                   else
                      readReg3 = 1;
                   OdioData1 = 1;
                   end
                else begin             {write the DIO register}
                   if a[1] = 0 then
                      WriteReg1 = 1
                   else
                      WriteReg3 = 1;
                   end;
                goto s10;
                end;
             end;
           end;

s4: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
          end
          else
          begin
             if an8250reg then begin
                if Nwrite = 0 then begin {write}
                   iow = 1;
                   oEXTdataEn;
                   end
                else {READ} begin
                   OdioData1 = 1;
                   ls2dio = 1;
                   ior = 1;
                   end;
                UartCs = 1;
                goto s5;
                end
             else             {shouldn't be here}
                goto startstate;
             end;
           end;

s5: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                    {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
          end
          else
          begin
             if an8250reg then begin
                if Nwrite = 0 then begin {write}
                   iow = 1;
                   oEXTdataEn;
                   end
                else {READ} begin
                   OdioData1 = 1;
                   ls2dio = 1;
                   ior = 1;
                   end;
                UartCs = 1;
                goto s6;
                end
             else
                goto startstate;
             end;
         end;

s6: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                    {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
            end
          else begin
             if an8250reg then begin
                if Nwrite = 0 then begin {write}
                   iow = 1;
                   oEXTdataEn;
                   end
                else {READ} begin
                   OdioData1 = 1;
                   ls2dio = 1;
                   ior = 1;
                   end;
                UartCs = 1;
                goto s7;
                end
             else
                goto startstate;
             end;
          end;

s7: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                    {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
          end
          else begin
             if an8250reg then begin
                UartCs = 1;
                if Nwrite = 0 then begin {write}
                   iow = 1;
                   oEXTdataEn;
                   goto s10;             {finished}
                   end
                else begin                  {read}
                   ior = 1;
                   OdioData1 = 1;
                   ls2dio = 1;
                   goto s8;
                   end;
                end
             else
                goto startstate;
             end;
           end;

s8: begin
          if ~cs then

          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                 {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
          end
          else begin
             if an8250reg then begin
                UartCs = 1;
                if Nwrite = 0 then begin {write, shouldn't be here}
                   goto startstate;
                   end
                else begin                 {read}
                   ior = 1;
                   OdioData1 = 1;
                   ls2dio = 1;
                   goto s9;
                   end;
                end
             else
              . goto startstate;
             end;
           end;

s9: begin
          if ~cs then
          begin  {Error exit, two cards at same address}
            Rbusrq = 1;                 {let go of bus}
            goto startstate;
            ior := 0;
            iow := 0;
            dtack := 0;
          end
          else
          begin
             if an8250reg then begin
                UartCs = 1;
                if Nwrite = 0 then begin {write, shouldn't be here}
                   goto startstate;
                   end
                else begin                 {read}
                   ior = 1;
                   OdioData1 = 1;
                   ls2dio = 1;
                   goto s10;
                   end;
                end
             else
                goto startstate;
             end;
         end; {case s9}

s10: begin
        if ~cs then begin
           Rbusrq = 1;          {no longer need bus}
           goto startstate;
           end
        else begin {cs still asserted}
           dtack = 1;
             if an8250reg then begin
                UartCs = 1;
                if Nwrite = 0 then {write}
                   oEXTdataEn
                else begin
                   OdioData1 = 1;
                   ls2dio = 1;
```

```
                    end;
                 goto s1;
                 end
              else begin          {internal register}
                 if Nwrite = 1 then begin
                    if a[1] = 0 then
                       readReg1 = 1
                    else
                       readReg3 = 1;
                    OdioData1 = 1;
                    end;
                end; {else not an 8250 reg}
           goto s10;
           end; {else cs}
        end; {case s10}

s11:                             {Unused State}
        begin
          goto startstate;
        end;
s12:                             {Unused State}
        begin
          goto startstate;
        end;
s13:                             {Unused State}
        begin
          goto startstate;
        end;
s14:                             {Unused State}
        begin
          goto startstate;
        end;
end;  {end of case statement}
END.
```

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Tue May  2 07:57:01 1989
PALASM Description for RS232SM


;; PAL Attributes File is
;; Using structure info for "ASIC"

DTACK = CS * ST[3] * ST[2] * /ST[1] * /ST[0]

IOR = CS * ST[2] * ST[0] * A[4] * NWRITE
  + CS * ST[1] * ST[0] * A[4] * NWRITE
  + CS * ST[3] * ST[0] * A[4] * NWRITE

IOW := CS * /ST[2] * ST[1] * ST[0] * A[4] * /NWRITE
  + CS * ST[2] * /ST[1] * ST[0] * A[4] * /NWRITE

LS2DIO = CS * ST[2] * /ST[1] * A[4] * NWRITE
  + CS * ST[0] * A[4] * NWRITE

ODIODATA1 = CS * ST[2] * /ST[1] * /ST[0] * NWRITE
  + CS * ST[0] * A[4] * NWRITE
  + CS * /ST[2] * /ST[1] * ST[0] * NWRITE

OEXTDATAEN = CS * ST[3] * /ST[1] * A[4] * /NWRITE
  + CS * ST[2] * /ST[1] * A[4] * /NWRITE
  + CS * /ST[2] * ST[0] * A[4] * /NWRITE

RBUSRQ = /CS * /ST[1]
  + /CS * ST[0]

READREG1 = CS * ST[2] * /ST[1] * /ST[0] * /A[4] * /A[1] * NWRITE
  + CS * ST[3] * /ST[2] * /ST[1] * /A[4] * /A[1] * NWRITE
  + CS * /ST[2] * /ST[1] * ST[0] * /A[4] * /A[1] * NWRITE

READREG3 = CS * ST[2] * /ST[1] * /ST[0] * /A[4] * A[1] * NWRITE
  + CS * ST[3] * /ST[2] * /ST[1] * /A[4] * A[1] * NWRITE
  + CS * /ST[2] * /ST[1] * ST[0] * /A[4] * A[1] * NWRITE

SBUSRQ = CS * /ST[3] * /ST[2] * /ST[1] * /ST[0] * A[4]

ST[0]  := CS * ST[3] * ST[0] * A[4] * NWRITE
  + CS * /ST[3] * /ST[2] * ST[0] * A[4]
  + CS * /ST[1] * ST[0] * A[4]
  + CS * /ST[3] * /ST[2] * /ST[1] * ST[0]
  + CS * /ST[3] * ST[2] * /ST[1] * /ST[0]

ST[1]  := CS * ST[3] * ST[1] * ST[0] * A[4] * NWRITE
  + CS * /ST[3] * /ST[2] * ST[1] * ST[0] * A[4]
  + CS * /ST[3] * ST[2] * /ST[1] * ST[0] * A[4]

ST[2]  := CS * ST[3] * /ST[1] * /ST[0]
  + CS * ST[3] * /ST[1] * A[4]
  + CS * ST[2] * ST[1] * ST[0] * A[4] * NWRITE
  + CS * ST[3] * /ST[2] * ST[0] * A[4]
  + CS * ST[3] * /ST[2] * /ST[1]

ST[3]  := CS * ST[3] * ST[2] * /ST[1] * /ST[0]
  + CS * /ST[3] * ST[1] * ST[0] * A[4] * NWRITE
  + CS * /ST[2] * ST[0] * A[4]
  + CS * /ST[2] * /ST[1] * ST[0]
  + CS * /ST[3] * /ST[2] * /ST[1] * BUSGR
  + CS * /ST[3] * /ST[2] * /ST[1] * /A[4]

UARTCS = CS * ST[3] * /ST[1] * A[4]
  + CS * ST[2] * /ST[1] * A[4]
  + CS * ST[0] * A[4]

WRITEREG1 = CS * ST[3] * /ST[2] * /ST[1] * ST[0] * /A[4] * /A[1] * /NWRITE

WRITEREG3 = CS * ST[3] * /ST[2] * /ST[1] * ST[0] * /A[4] * A[1] * /NWRITE
```

5.8 Centronics Register Access Machine

```
pal cnDIOsm;

{
        PAL Designer Centronics DIO register state machine.

        This file generates the Next State Equations and Output
        equations for the DIO register Centronics state machine.

        Don Soltis      v0.1    3/7/89
                Initial equations
        Don Soltis      v1.1    10/15/89
                Increased length of Write to register 1 (soft reset)

}

input           Cs, write, a[5..0],
                busgr, FifoFull, FifoEmpty,
                inbound,
                state[2..0];

output          SldFifo,RldFifo,SunldFifo,RunldFifo,
                busrq,
                olsDataEn,
                oDioDataEn1,
                cnCclk,
                dtack,
                ReadReg1,ReadReg3,ReadReg5,ReadReg7,ReadReg9,ReadReg11,
                WriteReg1,WriteReg3,WriteReg7,WriteReg9,
                state[2..0];


pal_type                'ASIC';

{ ONLY the active low outputs need to be specified here for logic reduction }
{ use the psm version of pd2ddl for proper generation }
pal_pin_order           ~SldFifo,~SunldFifo;

state_vars              state[2..0];

macro idle;             begin  0  end;
macro s0;               begin  1  end;
macro s1;               begin  5  end;
macro s2;               begin  6  end;
macro s3;               begin  2  end;

macro next_state;       begin  state[2..0]  end;

macro ExternalReg;      begin  (a[4..1] = 3) end; {Register 7}

macro FifoReg;          begin (a[4..1] = 5)  end; {Register 11}

procedure ReadDioReg; begin
   OdioDataEn1 = 1;
   case a[4..1] of
      0: begin  {Reg 1}
        ReadReg1 = 1;
        end;
      1: begin  {Reg 3}
        ReadReg3 = 1;
        end;
      2: begin  {Reg 5}
        ReadReg5 = 1;
        end;
      3: begin  {Reg 7}
        ReadReg7 = 1;
        end;
      4: begin  {Reg 9}
        ReadReg9 = 1;
        end;
      5: begin  {Reg 11}
        ReadReg11 = 1;
        RunldFifo = 1;
        end;
    end; {case}
end; {Procedure ReadDioReg}

procedure WriteDioReg; begin
   case a[4..1] of
      0: begin  {Reg 1}
        WriteReg1 = 1;
        end;
      1: begin  {Reg 3}
        WriteReg3 = 1;
        end;
      3: begin  {Reg 7}
        WriteReg7 = 1;
        end;
      4: begin  {Reg 9}
        WriteReg9 = 1;
        end;
      5: begin  {Reg 11}
        if ~inbound then SldFifo = 1;
        end;
    end; {case}
end; {Procedure WriteDioReg}

{**************************************
 * Main
 **************************************}
begin

case state[2..0] of

   idle: begin
      if Cs then begin
            {*********}
            {  Write  }
            {*********}
         if write then begin
            if ~ExternalReg then begin
               next_state := s0;
               end
            else begin
               busrq = 1;
               next_state := s0;
               end;
            end {if write}
            {********}
            {  Read  }
            {********}
         else begin {read}
            if inbound*(FifoReg) then SunldFifo = 1;
            next_state := s0;
            end; {else read}
         end {if cs}
      else begin
         next_state := idle;
         end; {else not cs}
      end; {state idle}

   s0: begin
      if Cs then begin
            {*********}
            {  Write  }
            {*********}
         if write then begin
            if ~ExternalReg then begin
               next_state := s1;
               end
            else begin
               busrq = 1;
               if busgr = 0 then next_state := s0
               else begin
                  OlsDataEn = 1;
                  next_state := s1;
                  end;
               end;
            end {if write}
         {********}
         {  Read  }
         {********}
      else begin {read}
         ReadDioReg;
         next_state := s1;
         end; {else read}
      end {if cs}
   else begin
      RldFifo = 1;
      RunldFifo = 1;
      next_state := idle;
      end; {else not cs}
   end; {state s0}

s1: begin
   if Cs then begin
         {*********}
         {  Write  }
         {*********}
      if write then begin
         if ~ExternalReg then begin
            if( a[4..1] = 0 ) then WriteReg1 = 1;
            next_state := s2;
            end
         else begin
            busrq = 1;
            cnCclk = 1;      {Load control into external latch}
            OlsDataEn = 1;
            next_state := s2;
            end;
         end {if write}
         {********}
         {  Read  }
         {********}
      else begin {read}
         ReadDioReg;
         next_state := s2;
         end; {else read}
      end {if cs}
   else begin
      RldFifo = 1;
      RunldFifo = 1;
      next_state := idle;
      end; {else not cs}
   end; {state s1}

s2: begin
   if Cs then begin
         {*********}
         {  Write  }
         {*********}
      if write then begin
         WriteDioReg;
         if ~ExternalReg then begin
            dtack = 1;
            next_state := s3;
            end
         else begin
            busrq = 1;
            dtack = 1;
            OlsDataEn = 1;
            next_state := s3;
            end;
         end {if write}
         {********}
         {  Read  }
         {********}
      else begin {read}
         ReadDioReg;
         dtack = 1;
         next_state := s3;
         end; {else read}
      end {if cs}
   else begin                {Shouldn't happen}
      RldFifo = 1;
      RunldFifo = 1;
```

```
         next_state := idle;
         end; {else not cs}
      end; {state s2}

   s3: begin
      if Cs then begin
            {*********}
            {  Write  }
            {*********}
         if write then begin
            if ~ExternalReg then begin
               RldFifo = 1;
               dtack = 1;
               next_state := s3;       {goto idle when cs goes away}
               end
            else begin
               busrq = 0;
               dtack = 1;
               next_state := s3;       {goto idle when cs goes away}
               end;
            end {if write}
            {********}
            {  Read  }
            {********}
         else begin {read}
            ReadDioReg;
            dtack = 1;
            next_state := s3;
            end; {else read}
         end {if cs}
      else begin
         next_state := idle;
         end; {else not cs}
      end; {state s3}

   end; {Case state of}
end.
```

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Sun Oct 15 14:57:29 1989
PALASM Description for CNDIOSM
/SLDFIFO
/SUNLDFIFO

;; PAL Attributes File is
;; Using structure info for "ASIC"

BUSRQ = CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * STATE[1] * /STATE[0]
  + CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * /STATE[1] * STATE[0]
  + CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * /STATE[2] * /STATE[1]

CNCCLK = CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * /STATE[1] * STATE[0]

DTACK = CS * STATE[1] * /STATE[0]

ODIODATAEN1 = CS * /WRITE * STATE[1] * /STATE[0]
  + CS * /WRITE * /STATE[1] * STATE[0]

OLSDATAEN = CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * STATE[1] * /STATE[0]
  + CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * BUSGR * /STATE[1] * STATE[0]
  + CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * /STATE[1] * STATE[0]

READREG1 = CS * /WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * /A[2] * /A[1] * /STATE[1] * STATE[0]

READREG11 = CS * /WRITE * /A[4] * A[3] * /A[2] * A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * A[3] * /A[2] * A[1] * /STATE[1] * STATE[0]

READREG3 = CS * /WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * /A[2] * A[1] * /STATE[1] * STATE[0]

READREG5 = CS * /WRITE * /A[4] * /A[3] * A[2] * /A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * A[2] * /A[1] * /STATE[1] * STATE[0]

READREG7 = CS * /WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * /A[3] * A[2] * A[1] * /STATE[1] * STATE[0]

READREG9 = CS * /WRITE * /A[4] * A[3] * /A[2] * /A[1] * STATE[1] * /STATE[0]
  + CS * /WRITE * /A[4] * A[3] * /A[2] * /A[1] * /STATE[1] * STATE[0]

RLDFIFO = CS * WRITE * A[4] * /STATE[2] * STATE[1] * /STATE[0]
  + CS * WRITE * A[3] * /STATE[2] * STATE[1] * /STATE[0]
  + CS * WRITE * /A[2] * /STATE[2] * STATE[1] * /STATE[0]
  + CS * WRITE * /A[1] * /STATE[2] * STATE[1] * /STATE[0]
  + /CS * STATE[2] * STATE[1] * /STATE[0]
  + /CS * /STATE[1] * STATE[0]

RUNLDFIFO = CS * /WRITE * /A[4] * A[3] * /A[2] * A[1] * STATE[1] * /STATE[0]
  + /CS * STATE[2] * STATE[1] * /STATE[0]
  + /WRITE * /A[4] * A[3] * /A[2] * A[1] * /STATE[1] * STATE[0]
  + /CS * /STATE[1] * STATE[0]

/SLDFIFO = /CS
  + /WRITE
  + A[4]
  + /A[3]
  + A[2]
  + /A[1]
  + /STATE[2]
  + /STATE[1]
  + STATE[0]
  + INBOUND

STATE[0] := CS * /STATE[2] * /STATE[1]

STATE[1] := CS * STATE[1] * /STATE[0]
  + CS * STATE[2] * /STATE[1] * STATE[0]

STATE[2] := CS * BUSGR * /STATE[1] * STATE[0]
  + CS * A[4] * /STATE[1] * STATE[0]
  + CS * A[3] * /STATE[1] * STATE[0]
  + CS * /A[2] * /STATE[1] * STATE[0]
  + CS * /A[1] * /STATE[1] * STATE[0]
  + CS * /WRITE * /STATE[1] * STATE[0]
```

```
 + CS * STATE[2] * /STATE[1] * STATE[0]

/SUNLDFIFO = /CS
  + WRITE
  + A[4]
  + /A[3]
  + A[2]
  + /A[1]
  + STATE[2]
  + STATE[1]
  + STATE[0]
  + /INBOUND

WRITEREG1 = CS * WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * /STATE[1] * STATE[0]
  + CS * WRITE * /A[4] * /A[3] * /A[2] * /A[1] * STATE[2] * STATE[1] * /STATE[0]

WRITEREG3 = CS * WRITE * /A[4] * /A[3] * /A[2] * A[1] * STATE[2] * STATE[1] * /STATE[0]

WRITEREG7 = CS * WRITE * /A[4] * /A[3] * A[2] * A[1] * STATE[2] * STATE[1] * /STATE[0]

WRITEREG9 = CS * WRITE * /A[4] * A[3] * /A[2] * /A[1] * STATE[2] * STATE[1] * /STATE[0]
```

5.9 Centronics FIFO-CENTRONICS (back end) Machine

```
pal cnBEsm;

{
        PAL Designer back end (Centronics-FIFO) Centronics state machine.

        This file generates the Next State Equations and Output
        equations for the BE Centronics state machine.

        Don Soltis      v0.1    3/7/89
                Initial equations

}


input           FifoEmpty,
                FifoFull,
                inbound,
                iStrobe,
                iBusy,
                iAck,
                busgr,
                slow,
                timeout,
                outbound,
                state[2..0];

output          SldFifo,RldFifo,SunldFifo,RunldFifo,
                oStrobe,oBusy,
                busrq,
                olsDataEn,
                startTimer,
                cnDclk,
                Soutbound,
                Routbound,
                state[2..0];


pal_type                'ASIC';

{ ONLY the active low outputs need to be specified here for logic reduction }
{ use the psm version of pd2ddl for proper generation }
pal_pin_order           ~SldFifo,~SunldFifo,~Soutbound;

state_vars              state[2..0];

macro idle;             begin  0  end;
macro s0;               begin  4  end;
macro s1;               begin  6  end;
macro s2;               begin  5  end;
macro s3;               begin  1  end;
macro s4;               begin  3  end;
macro s5;               begin  2  end;

macro next_state;       begin  state[2..0]  end;

{*************************************
 * Main
 *************************************}
begin

case state[2..0] of

   idle: begin
        {**********}
        { OUTBOUND }
        {**********}
      if ~inbound then begin
         if ~FifoEmpty*~iBusy then begin
            busrq = 1;
            Soutbound;
            next_state := s0;   {wait to get bus}
            end {if fifo not empty}
         else {fifo is empty, or Centronics is Busy} begin
            next_state := idle;
            end;
         end  {if outbound}
        {*********}
        { INBOUND }
        {*********}
      else begin {inbound}
         oBusy = 1;
         if ~FifoFull*~Slow then   begin
            Routbound;
            next_state := s0;
            end
         else begin
                {*** SLOW MODE, wait for fifo empty ***}
            if FifoEmpty*Slow then begin
               Routbound;
               next_state := s0;
               end
            else
               next_state := idle;
            end;
         end; {else inbound}
      end; {state idle}

   s0: begin
        {**********}
        { OUTBOUND }
        {**********}
      if ~inbound*outbound then begin
         busrq = 1;
         if ~busgr then begin
            next_state := s0;   {Wait for bus}
            end
         else begin {have bus}
            OlsDataEn = 1;
            next_state := s1;
            end;
         end  {if outbound}
        {*********}
        { INBOUND }
        {*********}
      else if inbound*~outbound then begin
         oBusy = 0;
         if iStrobe = 0 then next_state := s0
         else begin
            busrq = 1;
            next_state := s1;
            end; {else iStrobe = 1}
         end {else inbound}
      else
         next_state := idle;
      end; {state s0}

   s1: begin
```

```
     {**********}
     { OUTBOUND }
     {**********}
   if ~inbound*outbound then begin
      busrq = 1;
      if ~busgr then begin   {SHOULDN'T be here}
         next_state := s0;
         end
      else begin {have bus}
         OlsDataEn = 1;
         cnDclk = 1;         {Externally latch data}
         startTimer = 1;     {start up 1microsecond timer}
         SunldFifo  = 1;     {Clock fifo begining of s2}
         next_state := s2;
         end;
      end  {if outbound}
     {*********}
     { INBOUND }
     {*********}
   else if inbound*~outbound then begin
      busrq = 1;
      if ~busgr then next_state := s1
      else           next_state := s2;
      end {else inbound}
   else
      next_state := idle;
   end; {state s1}

s2: begin
     {**********}
     { OUTBOUND }
     {**********}
   if ~inbound*outbound then begin
      busrq = 0;     {all done}
      if ~timeout then begin
         next_state := s2;          {Wait for 1microsec data setup}
         end
      else begin {Got timeout, advance}
         RunldFifo = 1;             {advance FIFO}
         startTimer = 1;            {Start up timer again, min strobe width}
         oStrobe  = 1;
         next_state := s3;
         end;
      end  {if outbound}
     {*********}
     { INBOUND }
     {*********}
   else if inbound*~outbound then begin
      busrq = 1;
      cnDclk = 1;            { enable data in }
      {SldFifo = 1;}         {ldFifo start of s3 }
      next_state := s3;
      end {else inbound}
   else
      next_state := idle;
   end; {state s2}

s3: begin
     {**********}
     { OUTBOUND }
     {**********}
   if ~inbound*outbound then begin
      if timeout and iBusy then begin        {done with strobe}
         next_state := s4;
         end
      else begin
         oStrobe = 1;
         next_state := s3;          {Wait for busy}
         end;
      end  {if outbound}
     {*********}
     { INBOUND }
     {*********}
   else if inbound*~outbound then begin
      busrq = 1;
      cnDclk = 1;
      SldFifo = 1;
```

```
         oBusy = 1;                     { The sooner the better ? }
         next_state := s4;
         end {else inbound}
      else
         next_state := idle;
      end; {state s3}

   s4: begin
        {**********}
        { OUTBOUND }
        {**********}
      if ~inbound*outbound then begin
         if ~Slow then begin
            if iAck then next_state := s5        {Wait for Ack}
            else         next_state := s4;
            end
         else begin
            next_state := idle;                 {ignore Ack}
            end; {else Slow}
         end  {if outbound}
        {*********}
        { INBOUND }
        {*********}
      else if inbound*~outbound then begin
         RldFifo = 1;
         busrq = 0;     {Don't need it anymore}
         oBusy = 1;     {Keep busy asserted until iStrobe goes away}
         next_state = s5;
         end {else inbound}
      else
         next_state := idle;
      end; {state s4}

   s5: begin
        {**********}
        { OUTBOUND }
        {**********}
      if ~inbound*outbound then begin
         if ~iAck then next_state := idle       {Wait for Ack to go away}
         else          next_state := s5;
         end  {if outbound}
        {*********}
        { INBOUND }
        {*********}
      else if inbound*~outbound then begin
         oBusy = 1;
         if iStrobe = 1 then next_state := s5
         else                next_state := idle;
         end {else inbound}
      else
         next_state := idle;
      end; {state s5}

   end; {Case state}
end.
```

HEWLETT PACKARD

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Fri May 12 22:01:05 1989
PALASM Description for CNBESM
/SLDFIFO
/SUNLDFIFO /SOUTBOUND

;; PAL Attributes File is
;; Using structure info for "ASIC"

BUSRQ = /STATE[1] * STATE[0] * INBOUND * /OUTBOUND
  + STATE[2] * /STATE[0] * INBOUND * ISTROBE * /OUTBOUND
  + STATE[2] * STATE[1] * /STATE[0] * INBOUND * /OUTBOUND
  + STATE[2] * /STATE[0] * /INBOUND * OUTBOUND
  + /IBUSY * /STATE[2] * /STATE[1] * /STATE[0] * /FIFOEMPTY * /INBOUND

CNDCLK = /STATE[1] * STATE[0] * INBOUND * /OUTBOUND
  + BUSGR * STATE[2] * STATE[1] * /STATE[0] * /INBOUND * OUTBOUND

OBUSY = /STATE[2] * INBOUND * /OUTBOUND
  + /STATE[2] * /STATE[1] * /STATE[0] * INBOUND

OLSDATAEN = BUSGR * STATE[2] * /STATE[0] * /INBOUND * OUTBOUND

OSTROBE = /IBUSY * /STATE[2] * /STATE[1] * STATE[0] * /INBOUND * OUTBOUND
  + /STATE[2] * /STATE[1] * STATE[0] * /INBOUND * /TIMEOUT * OUTBOUND
  + STATE[2] * /STATE[1] * STATE[0] * /INBOUND * TIMEOUT * OUTBOUND

RLDFIFO = /STATE[2] * STATE[1] * STATE[0] * INBOUND * /OUTBOUND

ROUTBOUND = SLOW * /STATE[2] * /STATE[1] * /STATE[0] * FIFOEMPTY * INBOUND
  + /SLOW * /STATE[2] * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND

RUNLDFIFO = STATE[2] * /STATE[1] * STATE[0] * /INBOUND * TIMEOUT * OUTBOUND

/SLDFIFO = STATE[2]
  + STATE[1]
  + /STATE[0]
  + /INBOUND
  + OUTBOUND

/SOUTBOUND = IBUSY
  + STATE[2]
  + STATE[1]
  + STATE[0]
  + FIFOEMPTY
  + INBOUND

STARTTIMER = STATE[2] * /STATE[1] * STATE[0] * /INBOUND * TIMEOUT * OUTBOUND
  + BUSGR * STATE[2] * STATE[1] * /STATE[0] * /INBOUND * OUTBOUND

STATE[0] := /IACK * /SLOW * /STATE[2] * STATE[0] * /INBOUND * OUTBOUND
  + /STATE[1] * STATE[0] * INBOUND * /OUTBOUND
  + /STATE[1] * STATE[0] * /INBOUND * OUTBOUND
  + BUSGR * STATE[2] * STATE[1] * /STATE[0] * INBOUND * /OUTBOUND
  + BUSGR * STATE[2] * STATE[1] * /STATE[0] * /INBOUND * OUTBOUND

STATE[1] := /STATE[2] * STATE[1] * INBOUND * ISTROBE * /OUTBOUND
  + IACK * /STATE[2] * STATE[1] * /STATE[0] * /INBOUND * OUTBOUND
  + /STATE[2] * STATE[0] * INBOUND * /OUTBOUND
  + /SLOW * /STATE[2] * STATE[1] * STATE[0] * /INBOUND * OUTBOUND
  + IBUSY * /STATE[2] * /STATE[1] * STATE[0] * /INBOUND * TIMEOUT * OUTBOUND
  + /BUSGR * STATE[2] * STATE[1] * /STATE[0] * INBOUND * /OUTBOUND
  + STATE[2] * /STATE[1] * /STATE[0] * INBOUND * ISTROBE * /OUTBOUND
  + BUSGR * STATE[2] * /STATE[1] * /STATE[0] * /INBOUND * OUTBOUND

STATE[2] := STATE[2] * /STATE[1] * /INBOUND * /TIMEOUT * OUTBOUND
  + STATE[2] * /STATE[0] * INBOUND * /OUTBOUND
  + STATE[2] * /STATE[0] * /INBOUND * OUTBOUND
  + SLOW * /STATE[2] * /STATE[1] * /STATE[0] * FIFOEMPTY * INBOUND
  + /SLOW * /STATE[2] * /STATE[1] * /STATE[0] * /FIFOFULL * INBOUND
  + /IBUSY * /STATE[2] * /STATE[1] * /STATE[0] * /FIFOEMPTY * /INBOUND

/SUNLDFIFO = /BUSGR
  + /STATE[2]
  + /STATE[1]
  + STATE[0]
+ INBOUND
+ /OUTBOUND
```

5.10 Centronics DIO-FIFO DMA Machine

```
pal cnDMAsm;

{
        PAL Designer Centronics DIO-DMA State Machine program.

        This file generates the Next State Equations and Output
        equations for the DIO-DMA half of the Centronics interface.

        Don Soltis      v0.1    3/8/89
                Initial equations


}

input                   DMAen,FifoFull,FifoEmpty,inbound,
                        done,rst,
                        Udmack,         {Unsynced dmack}
                        dmack,dmrq,dtacki,
                        state[2..0];

output                  state[2..0],
                        SldFifo,                { set ldFifo }
                        RldFifo,                { reset ldFifo }
                        SunldFifo,              { set unldFifo }
                        RunldFifo,              { reset unldFifo }
                        Sdmrq,          { set DMA request }
                        Rdmrq,          { reset DMA request }
                        oDataEn,        { Output data Enable }
                        dmrdy;

pal_type                'ASIC';
pal_pin_order           ~SldFifo,~SunldFifo,~Sdmrq;

state_vars              state[2..0];

macro idle;             begin  0  end;

macro inb0;             begin  1  end; {Inbound state 0}
macro inb1;             begin  3  end; {Inbound state 1}
macro inb2;             begin  2  end; {Inbound state 2}

macro outb0;            begin  4  end; {outbound state 0}
macro outb1;            begin  5  end; {outbound state 1}
macro outb2;            begin  6  end; {outbound state 2}

macro next_state;       begin  state[2..0]  end;

begin

if rst then begin       { Reset }

   state[2..0]   := idle;
   Rdmrq         := 1;
   RldFifo       := 1;
   RunldFifo     := 1;

   end;

case state[2..0] of

   idle:        {idle states}
     begin

        if DMAen then begin  { Make sure DMA is enabled }

           {*****}
           { start inbound if Fifo not empty }
           {*****}
         if( inbound AND ~FifoEmpty ) then begin
           SunldFifo  := 1;
        Sdmrq     := 1;
        next_state := inb0;
        end

        {*****}
        { start outbound if Fifo is not Full }
        {*****}
      else if( ~inbound AND ~FifoFull ) then begin
         Sdmrq   := 1;
         next_state := outb0;
         end
       else
         next_state := idle;

      end {if DMAen}
     else
        next_state := idle;

     end; {in idle0}

inb0: begin
   if DMAen then begin
     if dmack then
        next_state := inb1
     else
        next_state := inb0; { wait for dmack }
     end

   else begin
     next_state := idle;
     Rdmrq := 1;         {don't dmrq}
     end;

   end; {case inb0}

inb1: begin
   if FifoEmpty then
      Rdmrq := 1;    {no more DMA requests}

   oDataEn := 1;
   RunldFifo := 1;
   next_state := inb2;
   end; {case inb1}

inb2: begin
   oDataEn := 1;
   if Udmack then dmrdy := 1; {must release dmrdy 0-50ns after dmack}
   if dmack then begin
     next_state := inb2
     end
   else begin
     if ~dmrq then begin
        next_state := idle
        end
     else begin             { Another transfer }
        SunldFifo  := 1;
        next_state := inb0;
        end;
     end;
   end; {case inb2}

outb0: begin

   if DMAen then begin
     if dmack then begin
        Rdmrq := 1;  {unrequest until later}
        if dtacki then begin
           next_state := outb1
           end
        else
           next_state := outb0;
        end
     else
        next_state := outb0; { wait for dmack }
     end

   else begin
```

```
        next_state := idle;
        Rdmrq := 1;         {don't dmrq}
        end;

      end; {case outb0}

   outb1: begin
      dmrdy := 1;              {data hold 85 ns after this}
      SldFifo := 1;            {Actually occurs in next state}
      next_state := outb2;
      end; {case outb1}

   outb2: begin
      if Udmack then dmrdy := 1; {must release dmrdy 0-50ns after dmack}
      RldFifo := 1;
      if dmack then
         next_state := outb2
      else begin
         if FifoFull then
            next_state := idle
         else begin
            Sdmrq := 1;
            next_state := outb0;
            end;
         end;

      end; {case outb2}

   end; { case }
end. {END}
```

HEWLETT PACKARD

```
<no-name>
PAL Designer 2.2 88/05/05 14:27
Wed Mar  8 09:48:13 1989
PALASM Description for CNDMASM
/SLDFIFO
/SUNLDFIFO /SDMRQ

;; PAL Attributes File is
;; Using structure info for "ASIC"

DMRDY := STATE[1] * /STATE[0] * UDMACK
  + STATE[2] * /STATE[1] * STATE[0]

ODATAEN := /STATE[2] * STATE[1]

RDMRQ := DMACK * STATE[2] * /STATE[1] * /STATE[0]
  + /DMAEN * STATE[2] * /STATE[1] * /STATE[0]
  + /STATE[2] * STATE[1] * STATE[0] * FIFOEMPTY
  + /DMAEN * /STATE[2] * /STATE[1] * STATE[0]
  + RST

RLDFIFO := STATE[2] * STATE[1] * /STATE[0]
  + RST

RUNLDFIFO := /STATE[2] * STATE[1] * STATE[0]
  + RST

/SDMRQ := /DMAEN * /STATE[1]
  + STATE[2] * /STATE[1]
  + STATE[1] * FIFOFULL
  + DMACK * STATE[1]
  + /STATE[2] * STATE[1]
  + STATE[0]
  + /STATE[1] * FIFOEMPTY * INBOUND
  + FIFOFULL * /INBOUND

/SLDFIFO := /STATE[2]
  + STATE[1]
  + /STATE[0]

STATE[0] := DMAEN * DMACK * STATE[2] * /STATE[1] * /STATE[0] * DTACKI
  + /DMACK * DMRQ * /STATE[2] * STATE[1] * /STATE[0]
  + DMAEN * /STATE[2] * /STATE[1] * /FIFOEMPTY * INBOUND
  + DMAEN * /STATE[2] * /STATE[1] * STATE[0]

STATE[1] := DMACK * STATE[1] * /STATE[0]
  + STATE[2] * /STATE[1] * STATE[0]
  + DMAEN * DMACK * /STATE[2] * STATE[0]
  + /STATE[2] * STATE[1] * STATE[0]

STATE[2] := STATE[2] * STATE[1] * /STATE[0] * /FIFOFULL
  + DMACK * STATE[2] * STATE[1] * /STATE[0]
  + DMAEN * STATE[2] * /STATE[1]
  + STATE[2] * /STATE[1] * STATE[0]
  + DMAEN * /STATE[1] * /STATE[0] * /FIFOFULL * /INBOUND

/SUNLDFIFO := STATE[0]
  + /DMAEN * /STATE[1]
  + STATE[2]
  + DMACK * STATE[1]
  + /DMRQ * STATE[1]
  + /STATE[1] * FIFOEMPTY
  + /STATE[1] * /INBOUND
```