9703782-4
IPOS COPY

SERIAL PORT FOR A HOST ADAPTER INTEGRATED CIRCUIT USING A SINGLE TERMINAL

THIS SHEET HAS BEEN REPLACED

BACKGROUND OF THE INVENTION

Reference to Microfiche Appendix

Microfiche Appendix A of 2 sheets and 75 frames and microfiche Appendix B of 1 sheet and 58 frames are part of the present disclosure, and are incorporated herein by reference in their entirety.

Microfiche Appendices A and B include VERILOG code listings for generating the modules for a serial port for a host adapter integrated circuit and a slave serial port input-output integrated circuit respectively.

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

FIELD OF THE INVENTION

The present invention is related generally to a serial port for an integrated circuit and in particular to a serial port for input and output of information to a circuit external to the integrated circuit using a single pin terminal of the integrated circuit.

DESCRIPTION OF RELATED ART

As the number of functions performed by an integrated circuit, hereinafter IC, increases, typically the number of pins of the integrated circuit also increases. However, as a rule of thumb, a

This Sheet Has Been Replaced



packaged integrated circuit with a large number of pins is more expensive to fabricate than an IC with relatively fewer pins. Also, a large number of pins adds to the cost of the board on which the IC is to be mounted. Packaged ICs with a large number of pins at the periphery cannot be used due to lack of real estate on the board. Packaged ICs with multiple rows of pins inside the periphery at the bottom of the package require additional layers in a board and increase complexity of interconnects on the board. The number of pins of an IC can also impose a limit on the number of functions that can be performed in the IC.

When an IC, such as host adapter 112A that interfaces an input-output bus, e.g. SCSI bus, to a host computer's system bus, e.g. PCI bus, (FIG. 1A) is mounted on a plug-in board 110, the number of pins needed by host adapter 112A is not constrained in a majority of cases. Host adapter 112A (FIG. 1A) has a number of pins, such as pins 112-1, 112-2, ... 112-N to support an adapter read-only-memory 111 for basic input-output software, hereinafter BIOS of host adapter 112A.

External logic (not shown) is needed by some host adapters to support adapter read-only-memory 111. For example, "36C70 SCSI IC Technical Reference Manual" by Future Domain Corporation, 2801 McGraw Avenue, Irvine, CA 92714, November 1993, discloses a host adapter in which "[a] minimal amount of external glue logic is required to serialize the parallel ROM data" (page 3-1). During system start-up, the information from the adaptor read-only-memory can be copied into system memory 170 for quick access by host processor 161, sometimes referred to as microprocessor 161.

In contrast, when a host adapter 112B (FIG. 1B) is mounted on a mother board 60 of a personal computer,

the number of pins of host adapter 112B can be limited to, for example, 100 pins due to less real estate available on mother board 160 as compared to plug-in board 110. Host adapter 112B eliminates the need for a connector that is otherwise necessary for a plug-in board. Host adapter 112B (FIG. 1B) does not have pins to access adapter read-only-memory 111, e.g. pins 112-1, 112-2, ... 112-N of host adapter 112A (FIG. 1A). BIOS for host adapter 112B is loaded from processor read-only-memory 162 that also contains the system BIOS for microprocessor 161. Thus host adapter 112B is limited to performing only certain basic functions, such as data transfer between system bus 120 and input-output bus 140. Such a host adapter 112B cannot be used on plug-in board 110 e.g. if host adapter 112B does not support a read-only-memory.

A way is needed for a limited pin integrated circuit, such as host adapter 112B to use resources, such as a read-only-memory, without increasing the number of pins, so that the same host adapter 112B can be used on both a mother board and a plug-in board.

SUMMARY OF THE INVENTION

In accordance with the principles of this invention, a host adapter integrated circuit, henceforth "host adapter", has a novel single pin serial port. The serial port uses a single bidirectional pin, for transfer of information from and to a circuit, such as a support circuit that is external to the host adapter. The support circuit contains resources that support certain functions that are not available in the host adapter.

The serial port allows various modules of host adapter, to communicate with the support circuit through the single serial port pin. The serial port also allows software on a host processor that is

connected to the host adapter by a system bus to communicate with resources in the support circuit. The serial port has no other pins that are connected to the support circuit for information transfer, such as control pins for interrupt signals or other control signals for handshaking or a data clock pin. In one embodiment, the host adapter serial port and the support circuit are operated synchronous with each other by a common clock signal that originates from an oscillator. A sequencer module in the host adapter buffers the common clock signal and passes the buffered clock signal to various modules of the host adapter, including the serial port.

One embodiment of a host adapter includes a master serial port input-output circuit that receives various internal signals from various modules of the host adapter and drives one or more command signals active onto a serial port command bus that is connected to the serial port. In response to an active command signal, the serial port generates a command byte, formats the command byte into a packet and then transmits the packet on the serial port pin.

The serial port forms a packet from any bytes of information to be transferred, such as a command byte, an address byte or a data byte by adding a start bit before the byte, followed by a parity bit after the byte and a stop bit after the parity bit. After transmitting one or more packets to the support circuit, the serial port waits for an acknowledge packet from the support circuit.

In response to active command signals, the serial port generates and transmits a command packet optionally followed by one or more address packets and data packets serially on the serial port pin that is coupled to the support circuit. The serial port

receives all responses from the support circuit on the same serial port pin.

In one embodiment, the serial port executes a command cycle to implement a serial port input-output protocol of a packet sequence specific to the command byte being transferred. For example, in response to a command signal to write one or two bits, such as a command to turn on and off (1) a light emitting diode, or (2) bus termination of the input-output bus or to reset a slave serial port input-output circuit included in the support circuit, the serial port executes a bit write command cycle in which the serial port includes the bits to be transmitted in the command byte, transmits a packet containing the command byte and waits for an acknowledge packet following transmission of the packet.

In response to a command signal to write a byte, for example to a predetermined register, in addition to transmitting a packet containing a command byte, the serial port also transmits a packet containing a data byte and then waits for the acknowledge packet.

In response to a command signal to write a byte to a specific address, for example to an electrically erasable programmable read only memory, the serial port transmits a packet containing command byte, followed by one or more packets containing the address bytes, e.g. two packets for a 16-bit address, followed by a packet containing the data byte and then waits for the acknowledge packet.

Similarly, in response to a command signal to read a byte, for example from a predetermined register, the serial port transmits a packet containing the command byte and then waits for an acknowledge packet that is followed by a packet containing the data in the register.

To read a four-byte word from an address in memory, such as a random-access-memory the serial port transmits a packet containing the command byte followed by two packets containing the address bytes and then waits for an acknowledge packet followed by four packets containing data bytes.

The support circuit determines the number of packets expected from the serial port of the host adapter from the contents of the packet containing the command byte. After receiving all expected packets, the support circuit starts transmission of an acknowledge packet and while transmitting the acknowledge packet, performs the operation indicated by the command byte. On completion of the operation, the support circuit terminates transmission of the acknowledge packet and depending on the operation transmits one or more data packets if necessary.

The use of such a serial port to off-load various functions of a host adapter to a support circuit is a significant improvement over prior art integrated circuits because the serial port reduces the number of pins of a host adapter. Such a host adapter can be used on a plug-in board with a support circuit, such as a slave serial port input-output circuit that provides various functions, such as support for an external read-only-memory. Instead of a slave serial port input-output circuit, a programmable logic circuit or a shift register can pass to the host adapter, for example, a device identification byte and byte of status of various resources accessible through the serial port. The same host adapter can also be used on a personal computer mother board to provide data transfer functions, without a support circuit.

In response to an active bit in a serial port control register, the serial port operates in a test mode in which the serial port passes an internal signal

of the host adapter to the serial port pin. The serial port exits the test mode only when reset.

In one embodiment, in the absence of a support circuit, the serial port pin that is normally used for information transfer is used by the host adapter for a default internal signal, such as turning power for bus termination on and off, which further reduces the total number of pins. Such a host adapter results in lower cost due to a smaller number of pins, smaller die size and volume production for use on a mother board as well as a plug-in board. Such a host adapter also takes less space on a mother board and so reduces the overall system cost for supporting data transfer between a system bus and an input-output bus.

The use of such a predetermined protocol in which one integrated circuit always waits for another integrated circuit eliminates possibility of contention for serial port input-output line avoids collision of packets and so eliminates need for control lines, in addition to serial port input-output line between a serial port of a host adapter and a support circuit.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGs. 1A and 1B illustrate two prior art computer systems that use a host adapter for data transfer between an input-output bus and a system bus.

FIGs. 2A and 2B show illustrative high level block diagrams of two embodiments of a computer system including a host adapter of this invention.

FIG. 3 shows an intermediate level block diagram for one embodiment of host adapter of FIGs. 2A and 2B.

FIGs. 4A-4I illustrate the sequence and timing of various packets generated and received by the serial port of FIG. 3.

FIG. 5 illustrates modules of the serial port of FIG. 3.

FIG. 6 shows an illustrative low level block diagram for the byte generator of FIG. 5.

FIG. 7 shows an illustrative low level block diagram of the converter of FIG. 5.

FIG. 8 shows an illustrative state diagram for the shifter state machine of FIG. 7.

FIGs. 9A-9C illustrate the timing of signals controlled by the shifter state machine of FIG. 8.

FIG. 10 shows an illustrative low level block diagram for the packet input-output controller of FIG. 7.

FIG. 11 illustrates a multiplexer and flip-flop used to implement a shifter stage in the shift register of FIG. 10.

FIGs. 12A and 12B illustrate two alternative embodiments of the line controller of FIG. 5.

FIG. 13 shows an illustrative high level block diagram for the slave serial port input-output circuit of FIG. 3.

FIG. 14 shows an illustrative state diagram for sequencer module 1330 of FIG. 13.

FIG. 15 illustrates an apparatus and method for imposing a single load on the PCI bus in one embodiment.

FIG. 16 shows an illustrative circuit diagram for implementing host adapter plug-in board of FIG. 3.

FIG. 17 illustrates skew acceptable in the clock signals of host adapter and the slave serial port input-output circuit of FIG. 3.

FIG. 18 shows an illustrative high level block diagram of an embodiment of a computer system including a host adapter of this invention connected to multiple slave devices.

FIG. 19 shows an illustrative high level block diagram of an embodiments of a computer system including multiple host adapters of this invention.

DETAILED DESCRIPTION

In accordance with the principles of this invention, a novel single pin serial port 230 (FIG. 2A) is included in an integrated circuit, such a host adapter 240. Serial port 230 uses a single serial port pin 241 of host adapter 240, for serial transfer of information from and to a circuit that is external to host adapter 240, such as support circuit 250. This is in sharp contrast to the normal serial port that typically requires at least three pins. Therefore, either the total number of integrated circuit pins can be reduced or additional pins are available for other functions.

Support circuit 250 includes a slave serial port-input output circuit 254 that interfaces with serial port 230, and that supports a number of functions using various external resources such as resources 251, 252 and 253. For example, serial port 230 can use support circuit 250 (1) to provide device identification data from an initialization resource, for example a programmable logic circuit, (2) basic-input-output software, henceforth "BIOS," from a memory resource, for example read only memory, henceforth "ROM", and (3) turn on and off a hardware resource, for example, a light emitting diode, henceforth "LED."

Serial port 230 interfaces with a master serial port input-output circuit 210 within host adapter 240 which controls use of various resources in support circuit 250 by various modules of data transfer circuit 220. Data transfer circuit 220 transfers data between system bus terminals 244 that are coupled to system bus 283 and input-output bus terminals 243 that are coupled to input-output bus 284. Data transfer circuit 220 has a number of modules, such as input-output bus module 221, FIFO module 222, sequencer module 223, and system bus module 225 that are all

connected to each other by a data transfer bus 226 and all of which are described in, for example, copending and commonly assigned U.S. patent application serial number 07/964,532, entitled "Intelligent SCSI Bus Host Adapter Integrated Circuit", filed on October 15, 1992, by Craig A. Stuber et al. that is hereby incorporated by reference in its entirety. Memory module 224 is used to temporarily store various data values that are used by firmware in sequencer module 223.

When used with support circuit 250, host adapter 240 does not need certain pins, for example sixteen address pins, eight data pins and two control pins for accessing a ROM, other control pins for other resources, and power and ground pins that were otherwise necessary in prior art host adapters. The smaller number of pins allows use of host adapter 240 in applications that do not require ROM support, as described below in reference to FIG. 2B. Also, the smaller number of pins allows use of host adapter 240 in applications where the number of pins is limited for compatibility with other products.

Moreover, the use of only one pin, e.g. serial port pin 241, henceforth "pin 241" for serial communication with support circuit 250 is a significant enhancement over prior art host adapters, because serial port 230 reduces the number of pins for serial communication to the smallest possible number of pins, i.e., one pin. As is known to those skilled in the art, as integration on a chip increases, one of the limiting factors is the numbers of pins on the integrated circuit. The smaller number of pins of host adapter 240 facilitates smaller die size and smaller package size which lowers cost as compared to prior art host adapters.

To utilize a resource in support circuit 250, a module of data transfer circuit 220 instructs master

serial port input-output circuit 210, on master input-output bus 245 that is connected to data transfer circuit 220, to access the resource through serial port 230. In response to instruction signals from a module of host adapter 240, master serial port input-output circuit 210 drives certain command signals active on serial port input-output bus 246 that connects serial port 230 to master serial port input-output circuit 210.

In response to the command signals, serial port 230 generates and serially transmits a command packet optionally followed by one or more address packets and data packets serially on pin 241 that is coupled to support circuit 250 by a serial port input-output line SPIO-, henceforth line SPIO-.

To indicate receipt of the information transmitted on pin 241, support circuit 250 transmits an acknowledge packet on line SPIO-. Support circuit 250 can transmit one or more packets on line SPIO- only in response to one or more packets from serial port 230, except that following a power-on reset, support circuit 250 transmits two packets containing initialization information. Non-receipt of the two initialization packets, within a predetermined time after power-on reset, indicates to serial port 230 to pass a default internal signal on line SPIO-, for example for input-output bus termination power control, instead of transmitting packets.

Serial port 230 can transmit on line SPIO- at any time that support circuit 250 is not allowed to transmit, for example after the two initialization packets. Such a predetermined protocol in which one integrated circuit always waits for another integrated circuit eliminates possibility of contention for line SPIO-, avoids collision of packets and so eliminates

need for control lines, in addition to line SPIO-, between serial port 230 and support circuit 250.

Serial port 230 receives all responses from support circuit 250 on the same pin 241. During reception of the acknowledge packet at pin 241, serial port 230 drives an acknowledge detect signal active on serial port input-output bus 246.

In response to the active acknowledge detect signal, master serial port input-output circuit 210 drives a command acknowledge signal and a command busy signal active on master input-output bus 245 to indicate that the command cycle is in progress and the resource is busy, e.g. not available for another command cycle, until completion of the current command cycle.

In FIG. 2A, for synchronous operation of host adapter 240 and support circuit 250, an oscillator 260 drives a clock signal CLK40 on both (1) host adapter clock line 206 that is connected to clock terminal 242 of host adapter 240 and (2) support circuit clock line 208, shown dotted, that is connected to support circuit 250. Such a common oscillator eliminates the need for a data clock signal from host adapter 240 to support circuit 250. Clock terminal 242 of host adapter 240 provides a clock signal that is buffered by sequencer module 223 to all internal modules of host adapter 240. So clock terminal 242 is not utilized solely with serial port 230 as was done by certain prior art host adapters. Support circuit clock line 208 is not necessary if a bus terminator in support circuit 250 is directly coupled to pin 241.

Execution of a command cycle by (1) generation of packets from command signals, (2) transmission of the generated packets on a single pin 241, and (3) reception of the acknowledge packet on the same pin 241 eliminates the need for additional lines, such as one

or more control lines for transferring interrupt signals or other handshaking signals and a clock line from serial port 230 to support circuit 250 and thereby facilitates a serial port having just one pin 241.

A single pin serial port 230 has the advantages of less pins on the two interconnected integrated circuits, less PCB etch routing, reduced timing constraints of multi-signal interface by elimination of signal-to-signal skew concerns, fewer signals, less noise and lower power requirements, than conventional serial ports.

In another embodiment, the same host adapter 240 described above, is also used in applications in which support for read-only-memory is not needed, for example, on a mother board 290 (FIG. 2B). In FIG. 2B, BIOS for host adapter 240 is loaded from processor ROM 291 that also contains system BIOS for host processor 281. In this embodiment, serial port 230 drives a default internal signal on pin 241. This embodiment is accomplished within the IC package of host adapter 240 by a bond wire 249 that connects a serial port disable terminal 247 on the die of host adapter 240 to a ground pin 248 of the die.

Disable terminal 247 is coupled to serial port command lines 323 and so inhibits serial port 230 from responding to command signals from master serial port input-output circuit 210 except for the default internal signal.

In such an inhibited state, pin 241 is used only by host adapter 240 for bus termination control, as described below in reference to FIG. 12B. When disable terminal 247 is left unconnected, serial port 230 is configured to transfer data between master serial port input-output circuit 210 and support circuit 250 as described above in reference to FIG. 2A.

Use of the same host adapter 240 on either a mother board 290 (FIG. 2B) or on a plug-in board 270 (FIG. 2A) results in lower costs due to volume production of a single die for both uses. Moreover, host adapter 240 also has smaller die-size, smaller number of terminals and therefore lower costs, all of which are important criteria for use on a mother board. Host adapter 240 also reduces the overall system cost of a computer system 200B (FIG. 2B), because host adapter 240 occupies less real estate than a conventional host adapter.

In response to various internal signals of host adapter 240, such as a read instruction signal or a write instruction signal from a hardware module, firmware, or software in data transfer circuit 220, an instruction router 311 (FIG. 3) in master serial port input-output circuit 210 passes the instruction signals to a resource controller, such as hardware resource controller 313, soft resource controller 314, initialization resource controller 315, and memory resource controller 316.

For example, in response to an internal signal on master serial input-output bus 245 to turn on or off a hardware resource, such as light emitting diode 350, hereinafter LED 350, instruction router 311 passes a write instruction signal on hardware bus 311H to a single byte write command controller, in hardware resource controller 313. In response to such a write instruction signal, the single byte write command controller drives one or more command signals active, e.g. a LED request signal LEDREQ (not shown) active to indicate a request for hardware resource LED 350 and a LED state signal LEDSTATE (not shown) active to indicate that LED 350 should be turned on, on serial port command bus 323 that is included in serial port input-output bus 246.

Command signals that are specific to a resource included in support circuit 250 have a reference numeral that is a combination of the command the resource. In general, for resource "X", reference numeral XREQ is a resource request signal for resource X; reference numeral XREAD is a read signal for resource X; reference numeral XSEND is a write signal for resource X; and reference numeral XBSY is a busy signal for resource X. Herein X can be any one of ROM, BRD, LED, SEE and SOFT, where "ROM" represents EEPROM 390; "BRD" represents board control logic 370; "LED" represents LED 350; "SEE" represents serial EEPROM 380; and "SOFT" represents soft resource 341.

In response to active command signals, such as LED request signal LEDREQ and LED state signal LEDSTATE, command executor 320 executes a command cycle, such as a write command cycle. In the command cycle a byte generator in command executor 320 first generates a command which in this embodiment is a command byte 410 (FIG. 4A).

In command byte 410, most significant bit R that is shifted out first is set for a read command cycle and cleared for a write command cycle. Bits C0 to C4 in command byte 410 are command bits that are used to encode various commands that can be transmitted by serial port 230 to support circuit 250. Among bits C0 to C4, bit C0 is the least significant bit. Bits D0 and D1 are data bits and bit D1 is the most significant bit in command byte 410.

A converter in command executor 320 formats the command byte into a command packet, such as command packet 420 (FIG. 4B). In this embodiment, a start bit 421 is inserted at the start of the packet and is followed immediately by command byte 410. Command byte 410 is followed by a parity bit 422 and then a stop bit 423. A line controller included in command

executor 320 serially transmits the command packet on pin 241 (FIG. 3) that is connected to a slave serial input-output circuit 254, also referred to as SSPIOC 254 in support circuit 250, by line SPIO-.

In the example for LED 350 described above, command executor 320 executes a bit write command cycle in which command executor 320 transmits a LED command packet 451 (FIG. 4E) that includes bit R "0", a command code "10000" for LED request signal LEDREQ in command code bits C0-C4 (FIG. 4A), the value of LED state signal LEDSTATE in data bit D0 and a constant e.g. 0 in data bit D1.

In general, command executor 320 forms a packet, such as a command packet, an address packet or a data packet by inserting a start bit 421 (FIG. 4B) having a first value that is followed by a command byte, an address byte or a data byte respectively, followed by a parity bit 422 which in turn is followed by a stop bit 423 having a second value different from the first value. In FIG. 4B, start bit 421 has a first value of a logical zero, hereinafter zero parity bit 422 is selected so that the transferred byte together with the parity bit has an odd parity, stop bit 423 has a second value of a logical one hereinafter one, and packet 420 is transmitted on line SPIO-. Line SPIO- is pulled up to a default value of one when line SPIO- is not being driven either by serial port 230 or by slave circuit 250. The identical value for the stop bit and for the default value of line SPIO- provides active negation that permits transmission of information across line SPIO- at higher speeds than possible without active negation. Active negation also prevents false detection of additional packets that can happen if line SPIO- is left at zero with only the pull-up resistor to take line SPIO- to one.

After a transmission time period Tp (FIG. 4D), that is associated with transmission of a packet, command executor 320, depending on the command cycle being executed, transmits additional address or data packets as described below and waits for an acknowledge packet 430 (FIG. 4C) that is described more completely below, during an acknowledge window time period Tn (FIG. 4D).

If command executor 320 does not receive a first bit of an acknowledge packet within acknowledge window time period Tn, command executor 320 sets a bit NOACK in an error register ERROR (not shown in FIG. 2A) of system bus module 225 and terminates the command cycle which in turn generates a target abort response to host processor 281, if the command packet transmitted on pin 241 originated from a command of host processor 281. In one embodiment, acknowledge window time period Tn is eight clock cycles after command executor 320 drives a stop bit on pin 241.

In general, in response to a command packet, such as LED command packet 451, slave serial port input-output circuit 254 starts transmission of an acknowledge packet 452 and performs the action indicated by command byte 410 retrieved from the command packet 420, for example send a one to LED 350 if the value of bit D0 is one, while continuing to transmit the acknowledge packet. On completion of the action indicated by command byte 410, slave serial port input-output circuit 254 completes transmission of acknowledge packet 430.

In general, slave serial port input-output circuit 254 waits for a first turnaround period of time and after receipt of a stop bit of the last packet expected from serial port 230, before transmission of an acknowledge packet. Serial port 230 also waits for the first turnaround time period to become sensitive to an

active signal on line SPIO- from slave serial port input-output circuit 254. The first turnaround time period can be any predetermined time period e.g. one clock cycle, two clock cycles or any other number of clock cycles.

Serial port 230 waits for a second turnaround time period to send a command packet after receipt of a packet from slave serial port input-output circuit 254. The second turnaround time period can also be any predetermined time period one clock cycle, two clock cycles or any other number of clock cycles. In one embodiment, the first turnaround time period and the second turnaround time period are identical and are equal to two clock cycles.

In general, acknowledge packet 430 includes a first bit 431 and a second bit 433, both having the same first value followed by a third bit 434 having a second value that is different from the first value. Acknowledge packet 430 can include any number of optional bits between first bit 431 and second bit 433 all of which have the same first value, depending on the amount of time needed by slave serial port input-output circuit 254 to complete the action indicated by command byte 410. In the embodiment of FIG. 4C, first bit 431 and second bit 433 both have first value zero, and third bit 434 has second value one and a number of optional bits 432 of first value zero are inserted between first bit 431 and second bit 433.

In response to a start bit 431 of an acknowledge packet 430 received at pin 241, command executor 320 drives an acknowledge detect signal ACKDET active on acknowledge detect line 322 that is connected to each of hardware resource controller 313, software resource controller 314, initialization resource controller 315 and memory resource controller 316. Command executor 320 continues to drive the acknowledge detect

signal active on acknowledge detect line 322 until receipt of third bit 434 on pin 241, at which point command executor 320 drives acknowledge detect signal ACKDET inactive and terminates the command cycle.

In the example for LED 350 described above, in response to an active acknowledge detect signal, the single byte write command controller in hardware resource controller 313 drives a set busy signal SETBSY (not shown) active to indicate that a serial port command cycle is in progress. When acknowledge detect signal goes inactive, hardware resource controller 313 drives set busy signal SETBSY inactive and waits for the next active instruction signal from instruction router 311.

While responding to an instruction signal, hardware resource controller 313 is insensitive to any additional instruction signals, for example for changing the state of LED 350. State change instruction signals for LED 350, that occur at a rate faster than the speed of serial transmission of a command packet on pin 241 and serial receipt of an acknowledge packet on the same pin 241, are lost. The state of LED 350 at the end of a series of state change instruction signals is the last state that was sent to slave serial port input-output circuit 254.

In one embodiment, instruction router 311 receives an internal signal LED- on master input-output bus 245 from a status switching circuit that is described in copending and commonly assigned U.S. Patent Application Serial No. 08/301,458, Attorney Docket No. M-2933, entitled "Status Indicator for a Host Adapter" of Stillman F. Gates and Charles S. Fannin, filed on September 7, 1994, that is incorporated by reference herein in its entirety. In one embodiment, a hardware module, or firmware or software of host adapter 240

turns on and off LED 350 as described for LED request signal LEDREQ in Table 1 below. Table 1 lists examples of conditions under which various command signals are generated by master serial port input-output circuit 210. Tables 2-9 list definitions of the bits of various registers in master serial port input-output circuit 210. Some of the bit definitions and signal values described herein are similar or identical to those described in "A1C-7870 PCI Bus Master Single-Chip SCSI Host Adapter Data Book-Preliminary" published in December 1993 available from Adaptec, Inc. 691 South Milpitas Boulevard, Milpitas CA 95035, that is incorporated by reference herein in its entirety.

TABLE 1

Command Signal	Function
LEDREQ (write)	Turn on or off LED 350
STPWREQ (write)	Turn on or off power to I/O bus terminators 360
CHPRSTREQ (write)	Reset slave serial port input-output circuit 254
BRDREQ (read or write)	Read from or write to board control register BRDCTL in board control logic 370.
SOFTREQ (read or write)	Read from or write to soft resource 340.
EIREAD (Read)	Receive identification byte IDDAT and external resource status byte ESTAT from SSPIOC 254

Command Signal	Function
ROMREQ (read or write)	Read ROM or EEPROM memory location or write to EEPROM memory location
SEEREQ (read or write)	Read from or write to serial EEPROM

Signal LEDREQ (TABLE 1) is a write-only signal that controls the state of LED 350 that is connected to a pin LED# on SSPIOC 254. Signal LEDREQ can originate from three sources. A first source is automatic hardware action generated by changes in use state of input-output bus 284 e.g. from not busy with data transfers to busy or from not busy and available for data transfer to busy with data transfers, if slave present bit SSPIOCPS is active (e.g. one) in external resource status register SPIOCAP in instruction router 311 (TABLE 2) and bit DIAGLEDEN is not active in register SBLKCTL (e.g., bit 7 in reg. 1Fh is zero) in input-output bus module 221. In response to an LED command byte containing a D0 as a one to indicate use state charged from busy to not busy, slave serial port input-output circuit 254 drives a signal on pin LED# high. In response to an LED command byte containing bit D0 as a zero command to indicate use state charged from not busy to busy, slave serial port input-output circuit 254 drives the signal on pin LED# low. Bit D1 for an LED command byte is always zero.

A second source of signal LEDREQ is generated by changing the value of bit DIAGLEDON in register SBLKCTL (e.g. bit 6 in reg. 1Fh) in input-output bus module 221, provided bit SSPIOCPS in external resource status register SPIOCAP is active (e.g. one) and bit DIAGLEDEN in register SBLKCTL (bit 7 in reg 1Fh) is active (e.g. one). Also in this case bit D1 is always

zero and slave serial port input-output circuit 254 asserts the signal on pin LED# low when bit D0 is zero e.g. bits DIAGLEDON and DIAGLEDEN are both one. Slave serial port input-output circuit 254 drives a signal on pin LED high when bit D0 is a one e.g. bit DIAGLEDEN is a zero and bit DIAGLEDEN is a one or input-output bus 284 is not busy and bit DIAGLEDEN is a zero.

A third source of signal LEDREQ is generated by firmware or software using a pre-defined soft command byte to put serial port 230 in a test mode. This test mode is indicated by setting bit D1 one in the LED command cycle, and in this test mode, line SPIO- is used to bring out internal signals of host adapter 240 during system operation for real time debugging purposes. Once SSPIOC 254 receives a LED command packet with bit D1 at a one, the SSPIOC 254 stops responding to all command packets and simply passes the signal on line SPIO- as the signal on pin LED#.

The test mode is exited only by a reset signal RST# or by power-on reset. In the test mode, host adapter 240 multiplexes out internal signals onto line SPIO- depending on the value stored in bits TESTSEL [2:0] of register SPIODATA (1Dh), which bits while in test mode are assigned the function name of TESTSEL, with bits [7:3] reserved. The value of bits TESTSEL can be changed to select a different internal signal to drive line SPIO-. See TABLE 3 below for definition of bits TESTSEL [2:0].

TABLE 3

Serial Port Test Mode Assignment Definitions

Value of bits TESTSEL (SPIODATA [2:0])	Name of internal signal in data transfer circuit 220

7 or 6	Unassigned
5	DFCACHES
4	DFCACHETHLA
3	PAUSE (not equal PAUSEACK)
2	MREQPEND
1	FRAME0-
0	DEVSEL0-

Each of the above internal signals is supplied on a LED test signal bus LTSTSIG1 that is included in master serial input-output bus 245. In response to a write to register SPIODAT, soft resource controller 314 passes a selected internal signal to serial port 230.

Signal DFCACHES indicates that there is data or space of one selected cache line in FIFO module 222 that is used by system bus module 225 to select a predetermined command for accessing system bus 283.

Signal DFCACHETHLA is a look ahead signal for streaming cache lines to improve bus performance from FIFO module 222.

Signal PAUSE requests sequencer module 223 to halt operations as described in the U.S. Patent Application filed by Craig A. Stuber referenced above.

Signal MREQPEND indicates that a memory request is pending for access to system bus 283 by system bus module 225 to transfer data between FIFO module 222 and system memory 282.

Signal FRAME0- is an internal signal that is active when host adapter 240 is acting as a bus master and is the signal supplied to the PCI bus as signal FRAME#.

Signal DEVSEL0- is an internal signal from system bus module 225 that is active when host adapter 240 is

acting as a bus target and is the signal supplied to the PCI bus as signal DEVSEL#.

In addition to the three sources of signal LEDREQ described above, when any one of the status bits is set (an OR condition) in slave serial port status register SPIOSTAT in SSPIOC 254, SSPIOC 254 drives a signal on pin LED# active to visually indicate an error, such as a parity error, provided that enable LED error display bit 7 is set. This feature can be used to aid in system debugging.

Referring back to Table 1, in response to command signal STPWREQ, command byte bit D1 is set to the state of bit STPWLEVEL in register DEVCONFIG of system bus module 225 and bit D0 is set to state of bit STPWEN in register SXFRCTL1 of input-output bus module 221. This command signal is issued after the first write of a one to SCSI termination power write enable bit STPWEN in input-output module 221 and there after whenever bit STPWEN changes state. Changing the state of SCSI termination power level bit STPWLEVEL in system bus module 225 does not cause this command signal to be issued. Signal STPWREQ can only be generated when bit SSPIOCPS in register SPIOCAP is one.

Following assertion of a reset signal RST# on system bus 283, SSPIOC 254 keeps SCSI termination power control pin STPWCTL in a float condition (three state) until the first command signal STPWREQ is received as shown by Table 4.

Thereafter, SSPIOC 254 controls SCSI termination power control pin STPWCTL based on values of bits STPWLEVEL and STPWEN.

TABLE 4

Signal on SCSI Termination Power Control pin STPWCTL

_____	STPWLEVEL	STPWEN	STPWCTL
0	—	—	Z (three state)

_____	STPWLEVEL	STPWEN	STPWCTL
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Bits STPWLEVEL and STPWEN are transmitted to SSPIOC 254 from bit STPWLEVEL in register DEVCONFIG and bit STPWEN in register SXFRCTL1 when command signal STPWREQ is asserted. Bit STPWEN is used to activate bus terminators 360. Bit STPWLEVEL indicates the voltage needed to deactivate bus terminators 360.

Termination power supplied by bus terminators 360 to input-output bus 284 is turned on or off as indicated by bit STPWCTL, based on the values of bits STPWLEVEL and STPWEN listed in Table 4.

When SSPIOC 254 is not present or if bond wire 249 (FIG. 2B) is present, pin 241 defaults to SCSI termination power control function and causes the output of pin 241 to float until bit STPWEN is written, as described below in reference to FIG. 12B.

Whenever a module of data transfer circuit 220 writes to a register SFXRCTL1 (not shown) in input-output bus module 221, instruction router 311 passes a signal to a single byte write command controller (not shown) of hardware resource controller 313 to transfer the current state of SCSI bus termination power level signal STPWLEVEL and power enable signal STPWEN to bus terminators 360 in support circuit 250. The command cycle executed by serial port 230 in response to command signal STPWREQ for controlling bus terminators 360 is similar to the sequence of actions described above in reference to LED 350, except that

bits D0 and D1 in the command byte contain the states of signals STPWLEVEL and STPWEN respectively.

Command signal CHPRSTREQ (Table 1) is issued as a result of setting bit CHPRST in register HCNTRL in system bus module 225, provided that slave present bit SSPIOCPS is one. The action of serial port 230 is different for command signal CHPRSTREQ than that described above for signal LEDREQ. Rather than transmit a command packet with start, stop, and parity bits serial port 230 simply drives a signal on pin 241 to a predetermined voltage for a predetermined number of clock cycles. Specifically in response to an active reset command signal CHPSTREQ, serial port 230 resets all internal registers except for system configuration registers located in system bus module 225. Serial port 230 also executes a soft reset command cycle in which serial port 230 drives a signal on pin 241 to logical zero for 12 or more clock cycles to reset slave serial port input-output circuit 254. In response to this reset packet SSPIOC 254 resets certain registers in SSPIOC 254 that are visible to host adapter 240, such as registers SPIOSTAT, BRDCTL and SEEPROM. Certain flip-flops that hold the address for BIOS accesses are not reset. SSPIOC 254 also drives the signal on pin STPWCTL to the selected inactive state (bit STPWLEVEL unchanged and bit STPWEN inactive).

The reset packet is issued as soon as possible after the leading edge of signal CHPRST by master serial port input-output circuit 210. In all commands except a soft command, SSPIOC 254 is reset on completion of the current command, and soft commands are aborted if the last command byte is still not sent.

Therefore, master serial port input-output circuit 210 and serial port 230 provide a seamless and compatible interface between resources in support circuit 250 and various host adapter modules. Host

adapter 240 turns on and off at least three different resources: slave serial port input-output circuit 254, LED 350 and bus terminators 360, using hardware resource controller 313, command executor 320 and serial port 230 on a single pin 241.

In addition, a module in host adapter 240 can write a data byte on data input bus 325, that is connected to a write data bus CDDAT included in data transfer bus 226, for example by putting an address 1Dh of a master board control register BRDCTL in hardware resource controller 313 on master input-output bus 245, that is connected to a write address bus CDADR included in data transfer bus 226. In response to such a write to master board control register BRDCTL, instruction router 311 drives a signal active on a hardware bus 311H coupled to a board controller included in hardware resource controller 311, if serial port 230 is available and if board control logic 370 exists on support circuit 250.

Instruction router 311 also determines the source of an internal signal on master input-output bus 245 from the state of a sequencer source signal PAC2SPIO on master input-output bus 245. Depending on the source of the internal signal, instruction router 311 drives a stretch signal, such as stretch sequencer signal STRETCHSEQ or a stretch host signal SEEBRDRDY, to the source of the instruction, such as sequencer module 223 or system bus module 225, to wait until the command cycle is completed. On completion of the command cycle, instruction router 311 drives the corresponding stretch signal inactive.

In response to an active instruction signal on hardware bus 311H from instruction router 311, the board controller in hardware resource controller 313 drives board request signal BRDREQ (Table 1) and board write signal BRDSEND active on serial port command

bus 323. In response to active command signals BRDREQ and BRDSEND command executor 320 executes a byte write command cycle that is similar to the bit write command cycle described above for LED 350, except that command executor 320 transmits a data packet 462 subsequent to transmission of command packet 461 and then waits for an acknowledge packet 463 from slave serial port input-output circuit 254.

So, board control command signal BRCTLREQ accesses board control register BRDCTL in SSPIOC 254. This command signal is issued when an address for board control logic 370, e.g. 1Dh is accessed provided board control bit BRDCTL in external resource register SPIOCAP is active and soft command enable bit SOFTCMDEN in external resource register SPIOCAP is inactive. When the access is from host processor 281, signal TRDY# on PCI bus 283 is not returned by system bus module 225 until the command is completed. When the access is from sequencer module 223, the sequencer clock is stretched until this command is completed.

Board control register BRDCTL, that is defined in TABLE 5, provides the capability to control reading and writing of external device(s) interconnected to the SSPIOC's memory port which may be shared with EEPROM and SEEPROM.

TABLE 5

Register BRDCTL in SSPIOC 254 and in Hardware Resource Controller 313

Bit #	Definition
7:5	BRDDAT[7:5].
4	BRDSTB.
3	BRDCS.
2	BRDRW.

1:0	Unused bits (BRDCTL[1:0]).

Board data bits BRDDAT[7:5] are read/write data bits that are only connected to pins MD[7:5] of slave serial port input-output circuit 254 when bit SEEMS is active. Bits BRDDAT are write bits when board read write bit BRDRW defined in Table 5 is inactive (e.g. zero) and the written value of bits BRDDAT is asserted on pins MD[7:5].

When bit BRDRW is active e.g. one, bits BRDDAT are read bits. One embodiment of slave serial port input-output circuit 254 connects bits BRDDAT [7:5] to pins MD [7:5] to read the current logical state of the signals on pins MD [7:5]. Another embodiment of slave serial port input-output circuit 254 connects bit BRDDAT[7] to pin MD7 and connects BRDDAT[6:5] to SCSI cable detection input pins CBLDET and XCBLDET (not shown) or SCSI termination detection input pins TRMSH1 and TRMSL0 (not shown) depending on the state of bits BRDSTB and BRDCS.

Board strobe bit BRDSTB is coupled to pin MD4 when bit SEEMS is active. When bit BRDSTB is active (e.g. one) pin MD4 is asserted low. Normal use is to store the desired write data in bits BRDDAT[7:5] and a one in bit BRDCS and then store a one in bit BRDSTB to activate external pin MD4 of SSPIOC 254 at a low level and start the strobe to write the data into the board control logic. A zero in bit BRDSTB, ends the strobe.

Board chip select bit BRDCS is a read/write bit connected to pin MD3 when bit SEEMS is asserted. When bit BRDCS is active (e.g. one), output on pin MD3 of SSPIOC 254 is asserted low.

Board read/write bit BRDRW controls the output state of pin MA15 of SSPIOC 254 when bit SEEMS is active. The state of bit BRDRW also controls the data direction of bits BRDDAT[7:5]. When bit BRDRW is

inactive e.g. zero bit BRDDAT[7:5] are output and pin MA15 is asserted low e.g. zero.

Unused bits are always read as zero and writes to unused bits are ignored.

The timing provided to external resources is a function of a software routine that matches the device's timing. Bits SEECTL, as indicated, may also be used for board logic control if desired.

After receipt of command packet 461, slave serial port input-output circuit 254 waits for the data packet 462 and then starts transmission of acknowledge packet 463. Slave serial port input-output circuit 254 updates a slave board control register BRDCTL with the data byte retrieved from data packet 462 so that both the master board control register and the slave control register have identical values. For convenience, the same reference numeral BRDCTL is used for the master board control register and the slave board control register. Slave serial port input-output circuit 254 then completes transmission of acknowledge packet 463.

The actions of command executor 320 and hardware resource controller 313 subsequent to receipt of acknowledge packet 463 on pin 241 are similar to the actions described above in reference to acknowledge packet 452 for LED 350. When the acknowledge detect signal ACKDET goes inactive on acknowledge detect line 322, the command cycle is completed and hardware resource controller 313 drives a board ready signal BRDCTLRDY active on hardware bus 311H that is connected to instruction router 313. In response to active board ready signal BRDCTLRDY, instruction router 311 drives the corresponding stretch signal to the host or the sequencer inactive to indicate completion of the write to slave board control register BRDCTL.

In response to a read of a master board control register BRDCTL, the actions of instruction router 311, hardware resource controller 313, command executor 320 and slave serial port input-output circuit 254 are similar to the actions described above in reference to the write to master board control register BRDCTL, except that command executor 320 executes a byte read command cycle in which command executor 320 transmits a command packet 471 (FIG. 4G) and after receipt of acknowledge packet 472, waits for a data packet 473. Command executor 320 passes a data byte retrieved from data packet 473 on received data lines 321 to retrieved data router 312, which in turn passes the retrieved data byte on data input bus 326, that in one embodiment is coupled to a destination data bus CSDAT included in data transfer bus 226, to the module that originated the internal signal to read data from master board control register BRDCTL. One set of bit definitions for a master board control register BRDCTL and a slave board control register BRDCTL are listed in "A1C-7870 PCI Bus Master Single-Chip SCSI Host Adapter Data Book-Preliminary" available from Adaptec, Inc. of Milpitas, California that was incorporated by reference above.

Soft command signal SOFTREQ (Table 1) is only issued as a result of access by a host processor 281 or sequencer module 223 to serial port control register SPIOCTL and serial port data register SPIODATA in soft resource controller 314. Before issuing a soft command signal SOFTREQ, the firmware or software examines the external resource status register SPIOCAP to determine whether soft commands are supported. The command byte value and number of bytes to be transferred following the command byte is register based and allows all values to be used. The acknowledge detect signal functions the same as in the other command cycles.

In response to an internal signal from a module of data transfer circuit 220, to write or read a register in soft resource 340, instruction router 311 drives corresponding signals active on soft controller lines 311S that are connected to soft resource controller 314. The operation of soft resource controller 314 is similar to the operation of board controller in hardware resource controller 313 described above except that the actions of soft resource controller 314 are controlled by the value of the byte stored in register SPIOCTL that is listed in Table 6.

Soft resource controller 314 includes a first-in-first-out memory element that buffers up to three bytes of data read from soft resource 340 in response to a read instruction from instruction router 311. Command executor 320 buffers a fourth byte of data from soft resource 340.

The address of register 341 in soft resource 340 is identical to the address of the master board control register BRDCTL and instruction router 311 selects register 341 if a bit SOFTCMDEN is active in a register SPIOCAP that is described in Table 2 above and otherwise selects master board control register BRDCTL.

Prior to issuing a soft command signal SOFTREQ, soft command enable bit SOFTCMDEN in external resource status register SPIOCAP must be set. Soft resource controller 314 is clocked only when soft command enable bit SOFTCMDEN is active. Therefore, to conserve power, soft command enable bit SOFTCMDEN is set only when a soft command signal is to be processed. Care should be taken when the firmware in sequencer module 223 issues soft command internal signals because of possible interaction between host processor software and host adapter firmware in sharing soft resource

controller 314. The following is one scheme using lock bit L as a semaphore.

If the firmware in sequencer module 223 wants to issue a soft command signal, sequencer module 223 should first check lock bit L. If lock bit L is not set, sequencer module 223 locks serial port 230 by setting lock bit L to one in serial port control register SPIOCTL that is defined in TABLE 6A.

TABLE 6A

Serial port control register SPIOCTL in soft resource controller 314

Bit #	Function
7	Send (S).
6	Last Byte (LB).
5	Lock (L).
4	Timer (T).
3:2	Reserved. Always read as zero.
1:0	Read (R[1:0]).

Send bit S is written active (e.g. one) after a byte has been written into register SPIODAT at address 1Dh. After the byte has been shifted out to line SPIO-, soft resource controller 314 clears bit S. When a module in data transfer circuit 220 samples bit S inactive, the next byte in the current command can be loaded into register SPIODAT. After the last byte is shifted out as indicated by bit LB, soft resource controller 314 begins sampling for a response for the current command from line SPIO-.

Last byte bit LB is set when the last byte for the current command is to be sent or received. After the

byte has been shifted out or in through line SPIO-, soft resource controller 314 clears last byte bit LB along with bit S.

Lock bit L is a read/write bit without any hardware function. Lock bit L is used by host processor 281 and sequencer module 223 as a semaphore to prevent overwriting each other if both of them issue soft commands. A semaphore is not needed if sequencer module 223 does not use soft commands.

Timer bit T is a hardware timer bit provided for software usage instead of a software timer loop. Timer bit T when written active (e.g. 1) automatically becomes inactive 800 ns after last byte bit LB is cleared, e.g. last byte of command is completely shifted out. The timer is referenced to a 40 MHz clock signal from sequencer module 223.

In one embodiment, instead of last byte bit LB, read bits R[1:0] define how many bytes are to be expected in response to the current command. In such an embodiment, after a byte has been shifted in from line SPIO- and transferred to register SPIODATA, that is defined in table 6B, soft resource controller 314 decrements the value of read bits R[1:0]. Software can read the value to determine when the shifted in data is available. Read bits R[1:0] can have the following values: 0: No bytes expected; 1: One byte expected; 2: Two bytes expected; 3: Three bytes expected. In one embodiment read bits R[1:0] are encoded as data bits D1 and D0 in the soft command byte formed in response to command signal SOFTREQ.

When set, lock bit L in serial port control register SPIOCTL prevents host processor 281 from interjecting a request during a multiple byte command cycle or when read data is expected. If send bit S is set while a hardware issued command cycle is being executed (e.g. due to automatic hardware action or PCI

ROM/EEPROM access), the soft command signal cannot begin until the current command cycle is terminated.

If host processor 281 wants to gain control of soft resource controller 314, host processor 281 first pauses sequencer module 223 and examines lock bit L in register SPIOCTL. If lock bit L is set, host processor 281 un-pauses sequencer module 223 and retries at a later time, until lock bit L is not set. Following this procedure prevents a deadlock of serial port 230. Host processor 281 does not rely on automatic pause (AAP) action to access serial port registers since sequencer module 223 can issue a soft command and overwrite host processor data.

In a soft command cycle, serial port data register SPIODAT is used to transfer information to SSPIOC 254, as shown in Table 6B.

Table 6B

Serial Port Input-Output Data Register SPIODATA in Soft resource Controller 314

BITS	DATA FUNCTION
7:0	Send command, address, or data byte when bit S is active in register SPIOCTL
7:0	Read data when bit S is inactive in register SPIOCTL

Although certain bit definitions for serial port control register SPIOCTL and serial port data register SPIODATA are described herein, these registers can be used with other definitions. For example, when a soft command cycle is used to put serial port 230 in test mode, serial port data register SPIODAT is used as a control register to indicate which internal signal is to be passed to pin 241.

If the command byte in register SPIODATA has a value of a command code as defined in Table 12, e.g. to access register SPIOSTAT, which register SPIOSTAT is defined in Table 7, a serial port status command cycle is implemented to access the serial port status register SPIOSTAT. Serial port status register SPIOSTAT is read when an acknowledge signal NOACK is returned to determine the exact cause. Reading register SPIOSTAT returns the current status and writing a one to a bit clears that bit.

TABLE 7

Serial port status register SPIOSTAT
in slave serial port input-output circuit 254

Bit#	Definition
7	ERRLEDENAB Enable LED error display
6-3	Reserved.
2	SPIOPARERR received data Parity Error
1	UNSUPPORTED command attempt
0	Reserved.

When bit ERRLEDENAB is set and either bit 2 or 1 is set, the signal on pin LED# is asserted, e.g. zero, to indicate an error condition. Bit ERRLEDENAB defaults to zero after power-on reset.

Bit SPIOPARERR is set when SSPIOC 254 detects a parity error in the shift-in data. Bit SPIOPARERR is cleared when a one is written to bit SPIOPARERR.

Unsupported command bit UNSUPPORTED is set when SSPIOC 254 receives an unrecognized or unsupported command. Unsupported command bit UNSUPPORTED is cleared when a one is written to unsupported command bit UNSUPPORTED.

Similarly to register SPIOSTAT, register SSPIOERRGEN in SSPIOC 254 is accessed by using command signal SOFTREQ (Table 1) with the appropriate command code in register SPIODATA (Table 13). When bit 0 is set to one in register SSPIOERRGEN, slave serial port input-output circuit 254 sets the parity bit so that the packet transmitted on line SPIO- has even parity and so exercises the parity check circuitry in serial port 230. When bit 0 is set to zero in register SSPIOERRGEN, SSPIOC 254 sets the parity bit so that the transmitted packet has odd parity and this is the normal mode. See TABLE 8.

Command signal EIREAD (Table 1) is issued by initialization resource controller 315 following each assertion of a signal on pin RST# (not shown) of host adapter 240. Also, after being reset, SSPIOC 254 automatically sends two initialization packets containing bytes IDDAT and ESTAT to initialization resource controller 315. While waiting for initialization packets, any host processor access causes a target abort response to be returned. In response to the initialization packets, command executor 320 passes byte IDDAT and byte ESTAT to initialization resource controller 315, that in turn writes byte IDDAT into a device identification register DEVICEID1 (PCI configuration register 00h, byte 3) in system bus module 225 that is used for all communications with host processor 281, and saves byte ESTAT in resource status register SPIOCAP (1Bh) that is used by instruction router 311 to determine the existence of various resources on support circuit 250. Byte ESTAT when shifted over line SPIO- is active low. Byte ESTAT is stored active high in register SPIOCAP. An acknowledge packet starting after the leading edge of a reset signal no earlier than 4 clock cycles and no later than 16 clock cycles precedes bytes IDDAT and

ESTAT. If initialization resource controller 315 does not receive an acknowledge packet within 16 clock periods after the rising edge of the signal on pin RST#, initialization resource controller 315 times out and assumes that an external slave serial port input-output circuit 254 does not exist.

Until this initialization command cycle is completed, any PCI target access attempted to host adapter 240 is responded to with signal RETRY to ensure that only the shifted in byte IDDAT is accessed by host processor 281 and existing external features have been identified and enabled.

When a slave serial port input-output circuit 254 is not used and device ID substitution is desired, an external device, such as a shift register device or a programmable logic circuit, provide bytes IDDAT and ESTAT. The rules for minimum time before driving line SPIO- and start, parity and stop bits are followed by the external device. However, if byte ESTAT is FFh and parity and stop bits are also e.g. one, the external device can stop driving line SPIO- after shifting out bit 0 of byte ESTAT, and is implemented in an 8-bit device in one embodiment. Allowing the external device to stop driving the SPIO- early is one reason for choosing an odd parity scheme in one embodiment.

During power-on reset of host adapter 240, instruction router 311 passes instruction signals to initialization resource controller 315 to receive a device identification byte IDDAT and a resource status byte ESTAT from slave serial port input-output circuit 254. No command packet is sent by command executor 320 on pin 241. Following reset, slave serial port input-output circuit automatically places bus terminators 360 in a float condition and sends initialization packets containing bytes IDDAT and ESTAT. Device identification byte IDDAT contains the

device identification code to be provided by host adapter 240 when register DEVICEID1 (not shown) is accessed by host processor 281. When byte IDDAT value is ffh, an internal device identification value is used. Resource status byte ESTAT contains information to be stored in register SPIOCAP.

When slave serial port input-output circuit 254, a shift register, or a programmable device is absent, an acknowledge packet is not received between 4 clock cycles and 16 clock cycles after signal RST# becoming inactive. In such a case, command executor 320 causes initialization resource controller 315 to use a default device identification code, to set byte ESTAT to FFh, and to pass the default internal signal STPWCTL to pin 241.

When present, slave serial port input-output circuit 254, on being reset, starts transmission of an acknowledge packet on line SPIO- that is connected to pin 241 before command executor 320 times out. While transmitting the acknowledge packet, slave serial port input-output circuit 254 can take any amount of time necessary to obtain the device identification code from, for example, a programmable logic circuit 330 and time required to assemble byte ESTAT by polling various resources, such as LED 350, soft resource 340, bus terminators 360, board control logic 370, SEEPROM 380 and EEPROM 390. After obtaining bytes IDDAT and ESTAT, slave serial port input-output circuit 254 terminates the acknowledge packet and transmits two initialization packets that contain the two obtained bytes.

Support for each command cycle by SSPIOC 254 is indicated by the associated bit being a zero in byte ESTAT. The bit definitions of register SPIOCAP are shown in Table 2. If the initialization command cycle times out or if byte ESTAT is FFh, none of register SPIOCAP bits are set to one.

Instruction router 311 generates instruction signals to access a resource only after determining the presence of the resource on support circuit 250. Instruction router 311 determines existence of a resource from the value of the corresponding bit in an external resource status register SPIOCAP, also referred to as "register SPIOCAP" that is defined in Table 2 below, such as bit BRDCTL for board control logic 370. External resource status register SPIOCAP is initialized to byte ESTAT after reset.

TABLE 2

Register SPIOCAP in Instruction Router 311

Bit #	Read/ Write	Definition of function enable bits
7:6	R	SOFT[1:0].
5	R/W	Soft Commands Enable (SOFTCMDEN). (Default "0".)
4	R	BRDCTL.
3	R	SEEPROM.
2	R	EEPROM.
1	R	ROM.
0	R	SSPIOC Present (SSPIOCPS).

Bits SOFT [1:0] of register SPIOCAP (Table 2) have no hardware assigned function, but allow software/firmware to be aware of external conditions on a board to board basis to allow customization flexibility. One or more ones indicate that external support is present for performing "soft" commands other than pre-defined soft commands.

Soft command enable bit 5 is written active

(e.g. one) by host processor 281 or sequencer module 223 to enable "soft" commands. When bit SOFTCMDEN is inactive, the definitions for address 1Dh and 1Eh are registers BRDCTL (Table 5) and SEECTL (Table 8), respectively. When bit SOFTCMDEN is active, the definitions for addresses 1Dh and 1Eh are registers SPIODATA (Table 6B and Table 3) and SPIOCTL (Table 6A), respectively. Bit SOFTCMDEN is not affected by the value of bit 5 of byte ESTAT but is read/write from host processor 281 or sequencer module 223.

When bit BRDCTL is a one, external logic is present to control external board control logic 370. Accesses to board control register BRDCTL in host adapter 240 are automatically redirected to board control register BRDCTL in SSPIOC 254. When board control register BRDCTL is a zero, writes to bit BRDCTL are ignored and reads to board control register BRDCTL return all zeros.

When bit SEEPROM is a one, external logic is present to access an external serial EEPROM. Accesses to serial EEPROM control register SEECTL in host adapter 240 are automatically redirected to serial EEPROM control register SEECTL in the external SSPIOC. When bit SEEPROM is a zero writes to serial EEPROM control register SEECTL are ignored and reads from serial EEPROM control register SEECTL return all zeros.

When bit EEPROM is a one and the ROM bit is also a one the external BIOS ROM support is also writable for in place BIOS updates.

When bit ROM is one, external logic is present to access an external BIOS ROM. Bit ROM when stored as a one allows register EXROMCTL in the configuration space of system module 225 to be read with the value one written to bit ROM indicating PCI external BIOS ROM support. When not a one, reading register EXROMCTL

returns a 0h value indicating no support for any value written to EXROMCTL.

Slave present bit SSPIOCPS being one indicates that SSPIOC 254 is connected to pin 241 and enables the LED, STPW and SPIORST command cycles. Bit SSPIOCPS is not set if a shift register or programmable logic is used to pass in byte IDDAT.

Command signal ROMREQ is only issued as a result of a host processor 281 access when bit ROM in register SPIOCAP is active e.g. one, bit EXPROMEN in register EXROMCTL is active, bit MSPACEEN in register COMMAND is active and the access is within the 64 KByte memory address spaced stored in register EXROMCTL.

In a memory read command cycle, the command packet is followed by two address packets that identify the double word to be accessed in the external memory SSPIOC 254 uses 18 address bits MA17-MA0, to access memory, such as ROM or EEPROM. Bits MA7-MA0 are sent in the first address byte, bits MA15-MA8 are sent in the second address byte, and bits MA17 and MA16 are sent in the command packet in bits D0 and D1. See FIG, 4I. Also, in a memory read command cycle SSPIOC 254 requires that bits MA1 and MA0 always be 0. SSPIOC 254 issues four ROM commands to complete the double word required for host processor 281 ROM access. In one embodiment bits [17:16] are used to access up to 256K ROM.

If serial port 230 is busy, the host processor's ROM access is terminated with the PCI signal RETRY, without generating a command signal to memory resource controller 316. If the memory port on SSPIOC 254 is busy because the memory port has been reconfigured for SEEPROM, the access is terminated with a PCI Target Abort, without generating a command signal to memory resource controller 316. The state of the memory port

can be determined by examining bit SEEMS in register SEECTL.

For memory reads, command signal ROMREQ (Table 1) is issued as a result of a host processor 281 access when bit ROM in register SPIOCAP is active (e.g. 1), bit EXROMEN in register EXROMCTL is active, bit MSPACEEN in register COMMAND is active, and the access is within the 64 KByte memory address space stored in register EXROMCTL. For memory writes, in addition to the above bits for memory reads, the EEPROM bit in register SPIOCAP must also be an active e.g. one. In the memory write command cycle, the command packet is followed by two address packets and a data packet containing a data byte that is to be written in the external EEPROM.

All bits in the address bytes are used to identify the byte being written. SSPIOC 254 writes the data byte into the EEPROM, and after completing the write, terminates acknowledge packet indicating that the write is completed, so that host adapter 240 can assert PCI signals TRDY# and STOP# to host processor 281 to terminate the transaction.

In response to instruction signals on master serial input-output bus 245 to access a memory location in a memory resource, such as EEPROM 390, or SEEPROM 380, instruction router 311 passes an active instruction signal on memory bus 311M that is connected to memory resource controller 316.

In response to an active instruction signal to access EEPROM, a parallel memory controller included in memory resource controller 316 drives a ROM request signal ROMREQ (not shown) active on serial port command bus 323 to indicate the request for memory resource EEPROM 390. The parallel memory controller also drives a ROM read signal ROMREAD (not shown) active or alternatively a ROM write signal ROMSEND (not shown)

active on serial port command bus 323 to indicate a read command cycle or a write command cycle, respectively.

In response to an active signal ROMSEND, command executor 320 executes a memory byte write command cycle that is similar to the memory byte write command cycle described above for the board control register except that command executor 320 transmits two address packets 482 and 483 (FIG. 4H) between command packet 481 and data packet 484.

After receiving all packets 481 to 484, slave serial port input-output circuit 254 starts transmission of acknowledge packet 485, updates EEPROM 390 at the memory location identified by the address contained in address packets 482 and 483 with the data contained in the data packet 484 and then terminates acknowledge packet 485. In one embodiment, slave serial port input-output circuit 254 uses two data bits D0 and D1 of command packet 481 as high order address bits in accessing the memory location. The actions of command executor 320 and memory resource controller 316 subsequent to receipt of acknowledge packet 485 on pin 241 are similar to the actions described above in reference to acknowledge packet 452 for LED 350.

In response to an active ROM read command signal ROMREAD, command executor 320 executes a memory byte read command cycle in which command executor 320 transmits a command packet followed by two address packets 492 and 493. (FIG. 4I)

After receiving all packets 491 to 493, slave serial port input-output circuit 254 starts transmission of an acknowledge packet 494 and while transmitting acknowledge packet 494 retrieves a byte of data from a memory location identified by the address in address packets 492 and 493, completes transmission

of acknowledge packet 494 and transmits the retrieved data byte in a data packet 495.

On receipt of data packet 495, command executor 320 passes the data byte retrieved from data packet 495 on received data lines 321 to retrieve data router 312. In response to the data byte on received data lines 321, retrieved data router 312 writes the data byte into a byte assembly register (not shown) in system bus module 225 via data output lines 326. In the embodiment of FIG. 4I, host processor 281 expects a four byte word for each access to ROM. So, during transmission of data packet 495, slave serial port input-output circuit 340 increments the address received in address packets 492 and 493 and retrieves a second byte of data from EEPROM 390. On completion of transmission of data packet 495, slave serial port input-output circuit 230 transmits the second byte in data packet 496, and similarly third and fourth bytes in data packets 497 and 498.

The actions of command executor 320 and retrieved data router 312 in response to the second, third and fourth bytes are similar to that described above for the first byte. When all four bytes have been assembled in the byte assembly register, retrieved data router 312 drives a ROM ready signal SPROMRDY (not shown) active to system bus module 225 which then indicates that all four bytes are available to host processor 281.

Although the above description refers to a memory resource EEPROM 390, a different memory resource, such as a battery backed SRAM, a flash EEPROM or a EEPROM can also be used. Moreover, a ROM can also be used except that instruction router 311 will not permit writing to a ROM based on bit EEPROM of status register SPIOCAP.

Command signal SEEREQ (Table 1) that is generated by memory resource controller 316 is enabled when bit SEEPROM in register SPIOCAP is active and soft command enable bit SOFTCMDEN in register SPIOCAP is inactive. Command signal SEEREQ is issued when master serial EEPROM control register SEECTL (1Eh) is written or read, with the timer SEERDY expired and no other command in process.

When the request is from host processor 281, PCI signal TRDY# is not returned until the command cycle is completed. When the request is from sequencer module 223, the sequencer clock is stretched until the command cycle is completed.

Command signal SEEREQ results in access to slave serial EEPROM control register SEECTL on SSPIOC 254. SSPIOC 254 immediately provides an acknowledge packet for write or read comments, and in the case of reading also immediately sends back as a real data byte the contents of the slave serial EEPROM control register SEECTL; which includes the SEEPROM control signals and serial data out.

Slave serial EEPROM control register SEECTL on SSPIOC 254 provides the capability to control reading and writing an external serial 1-bit EEPROM device that contains a four pin interface (typical devices are NM93C06/C46/C56/C66 that are available from National Semiconductor). The SEEPROM is interconnected to the SSPIOC's memory port which may be shared with a ROM/EEPROM or board logic devices. The data and clock timing required by the external SEEPROM is provided by a software routine that matches the device's timing. Due to the slow clock rate, typically 1 MHz maximum, a hardware timer is provided in master serial EEPROM control register SEECTL to ease software development and provide portability. Table 8 shows the bit definition of register SEECTL.

TABLE 8

SEEPROM control register SEECTL
in memory resource controller 316 and
slave serial port input-output circuit 254

Bit#	Definition when bit SOFTCMDEN is inactive in register SPIOCAP - Table 2)	SSPIOC 254 I/O pin
7	reserved = 0	
6	reserved = 0	
5	Serial EEPROM Mode Select bit SEEMS.	
4	Serial EEPROM Ready bit timer SEERDY.	
3	Serial EEPROM Chip Select bit SEECS.	SEECS
2	Serial EEPROM Clock bit SEECK.	MD2
1	Serial EEPROM Data Out bit SEEDO.	MD1
0	Serial EEPROM Data In bit SEEDI.	MD6

Serial EEPROM Mode Select bit SEEMS is a read/write bit which when set active, e.g. one, or set inactive, e.g. zero, causes memory resource controller 316 to send a request to SSPIOC254 for access to SEEPROM 380 or to board control logic 370. An active bit SEEMS reconfigures the SSPIOC memory port to allow SEEPROM control bit SEECTL to redefine pins MD[2:0] (see Table 8) and activate SEEPROM chip select bit SEECS for access of SEEPROM 380 or board control logic. Once SEEPROM mode select bit SEEMS is asserted, access to other resources of SSPIOC 254 such as a ROM and EEPROM is inhibited until SEEPROM mode select bit SEEMS is inactive.

Serial EEPROM Ready bit SEERDY is a read only bit that provides a hardware timer that is used instead of a software timer when accessing SEEPROM 380 or board control logic 370. Each time register SEECTL is written, bit SEERDY goes inactive and after a 800 ns or longer delay when access to SEEPROM 380 is completed becomes active, e.g. one. The state of bit SEERDY must be read active before continuing to the next step of addressing SEEPROM 380.

Serial EEPROM Chip Select bit SEECS is a read/write bit that is used to control SEEPROM chip select pin SEECS of SSPIOC 254. When SEEPROM chip select bit SEECS to active, e.g. one, can drive signal active on pin SEECS only if bit SEEMS is active. Pin SEECS can also be used to qualify the SSPIOC memory interface lines as BRDCTL signals when both SEEPROM 380 and board control logic 370 are present. Pin SEECS requires an external pull down resistor if SEEPROM 380 is present.

Serial EEPROM clock bit SEECK is a read/write bit that controls the state of pin MD2 which is connected to the serial data clock input pin of SEEPROM 380.

Serial EEPROM Data Out bit SEEDO is a read/write bit that controls the state of pin MD1. Pin MD1 is connected to the data input pin of SEEPROM 380. When bit SEEDO and SEEMS are active, e.g. one, pin MD1 is set to a high level for writing a bit into SEEPROM 380. Bit SEEDO can be used with other resources in addition to SEEPROM 380. For example bit SEEDO can be used as a fourth bit in addition to three board data bits BRDDAT[7:5] that are listed in Table 5 for example to write a nibble of data into a four bit register in board control lgoci 370.

Serial EEPROM Data In bit SEEDI is a read only bit that is used to access data from SEEPROM 380. The value of bit SEEDI reflects the value of pin MD0 which is connected to the data output terminals of SEEPROM 380 when bit SEEMS is active.

One set of bit definitions for a master serial EEPROM control register SEECTL and a slave serial EEPROM control register SEECTL are listed in "A1C-7870 PCI Bus Master Single-Chip SCSI Host Adapter Data Book-Preliminary" available from Adaptec, Inc. of Milpitas, California.

A module of host adapter 240 issues an internal signal to access SEEPROM 380 by setting a serial EEPROM mode select bit SEEMS in master serial EEPROM control register SEECTL. In response to a read or write to an address of a master serial EEPROM control register SEECTL in memory resource controller 316, instruction router 311 passes corresponding instruction signals to memory resource controller 316.

Instruction router 311 also drives a signal SEEBRDRDY inactive for at least 800 nanoseconds, and then continues to drive signal SEEBRDRDY inactive until a signal from memory resource controller 316 indicates completion of the access to SEEPROM 380. In response to a read or write internal signal for access

to SEEPROM 380, the actions of memory resource controller 316, command executor 320 and slave serial port input-output circuit 254 are similar to the actions described above in reference to read and write for master board control register BRDCTL.

In one embodiment, master serial port input-output circuit 210 and serial port 230 were implemented using the VERILOG code of appendix C and microfiche appendix A in particular, the VERILOG subroutines shown in Table 9. The VERILOG code in appendix C and appendix A microfiche was synthesized using Synopsys compiler version 3.1, that is available from Synopsys of Mountain View, CA to generate a gate net list used in a host adapter integrated circuit implemented in one embodiment as 0.8 micron standard cell CMOS integrated circuit.

The VERILOG subroutines shown in Table 9 utilize and generate signals that are specific to a particular embodiment of host adapter 240. Tables 10 and 11 define signals to and from data transfer circuit 220 and serial port 230 respectively so that those skilled in the art can implement the invention in other embodiments.

TABLE 9

VERILOG Code used in Modules of Host Adapter

Modules of Host Adapter	VERILOG Code of Appendix C and Microfiche Appendix A
Software Resource Controller 314	softctl, softcntrl, softcmdsm, softregs, softtimer, spfifo, spfifoctl, spfiforegs
Hardware Resource Controller 313	brdcntrl, brdctlsm, sbwcsm, sbwcctl, led, stpwr

Modules of Host Adapter	VERILOG Code of Appendix C and Microfiche Appendix A
Memory Resource Controller 316	romctl, spromctl, seecntrl, seectlsm spromsm
Initialization Resource Controller 315	eism, iddatestat, sprstctl
Instruction Router 311	sprw, sprwctl, spstretch, spclkgen and sprwdec
Retrieved Data Router 312	sprdmux
Serial Port, Byte Generator 510	sparbomux
Serial Port, Converter 520	spcnt, spctl, spctlsm, spcntrl and spshft

TABLE 10

Signals to Data Transfer Circuit from Serial Port

Signal Name in VERILOG Code	Function	From	To
berren	generate parity error to exercise parity checking circuit	System bus module 225	Serial port 230
chprstb	reset chip (initialize) (self resetting)	System bus module 225	MSPIOC 210 and serial port 230

Signal Name in VERILOG Code	Function	From	To
clrparerr_	clear parity error that is currently being reported (self resetting)	system bus module 225	serial port 230
crbusy	part of data transfer bus 226 is being read	retrieved data router 312	Sequencer Module 223 or system bus module 225
clk40	40 MHz clock for timing, can stay high to save power in power down mode	Sequencer module 223	MSPIOC 210 and serial port 230
cdadr_ [7:0]	part of data transfer bus 226, destination write address	Sequencer Module 223 or system bus module 225	instruction router 311
cddat [7:0]	part of data transfer bus 226, destination write data	Sequencer Module 223 or system bus module 225	MSPIOC 210 and serial port 230

Signal Name in VERILOG Code	Function	From	To
cdwen_	data transfer bus 226, destination write enable	Sequencer module 223 or system bus module 225	instruction router 311
csadr_ [7:0]	data transfer bus 226, source address	Sequencer module 223 or system bus module 225	retrieved data router 312
csren_	data transfer bus 226 source read enable drives CSDAT	Sequencer module 223 or system bus module 225	retrieved data router 312
ltstsigi [7:0]	LED test bus for test configuration signals	Input output bus module 221	Soft resource controller 314
mparcken	enables parity checking of received packets	System bus module 225	Serial port 230
pac2spio	Sequencer is in control of bus 226	Sequencer module 223	Instruction Router 311

Signal Name in VERILOG Code	Function	From	To
por	power on reset	System bus module 225 - from PCI reset or chprstb	Initial-ization Resource Controller 315, MSPIOC 210, serial port 230
rstib_	PCI reset to initialize various things	System bus module 225	Initial-ization Resource Controller 315 MSPIOC 210 serial port 230
sled_	scsi bus busy signal from SCSI led logic	Input-output bus module 221	Hardware Resource Controller 313
romdec_	ROM decode goes true when bits present, enable are active and address is valid.	System bus module 225	Instruc-tion Router 311
romadr [15:0]	Address of ROM location to be read.	System bus module 225	Instruction Router 311

Signal Name in VERILOG Code	Function	From	To
romwr	ROM write command	System bus module 225	Instruction Router 311
stpwenb	SCSI power down bus termination (control bit 0)	Input-Output bus module 221	Instruction Router 311
stpwlvlb	SCSI power down bus termination level (control bit 1)	Configuration Register in System bus module 225	Instruction Router 311
iddat	new device identification	serial port 230	System bus module 225
csdat [7:0]	part of data transfer bus 226, source read data	retrieved data router 312	Sequencer module, 223 system bus module 225
eepromen	EEPROM enable	serial port 230	register SPIOCAP
idldrdy	Indicates when byte IDDAT has been stored in register DEVICED1 following a reset	Hardware resource controller 313	System bus module 225
noack	Send target abort to host processor	serial port 230	System bus module 225

Signal Name in VERILOG Code	Function	From	To
romrden	ROM present enable	serial port 230	SPIOCAP register and system bus 225
spromrdy	4 bytes of ROM data is available for host	memory resource controller 316	system bus module 225
sspiocps	SSPIOC is present	serial port 230	register SPIOCAP
seems	SEEPROM mode select signal; only SEEPROM and BRDCTL operations allowed (no access to ROM, LED, EEPROM etc.)	Memory resource controller 316	Instruction router 311
spromclk_	Clocks each byte from ROM into a 32 bit holding register	Memory resource controller 316	System bus module 225
spioparerr	Parity error detected in packet received on line SPIO-	Serial port 230	System bus module 225

Signal Name in VERILOG Code	Function	From	To
spbrden	Board enable	Serial port 230	Register SPIOCAP
spseeen	SEEPROM enable	Serial port 230	Register SPIOCAP
spiobsy_	Serial port access in progress	Host adapter modules 313, 314, 315 and 316	Instruction router 311
stretchseq	Stretch clock of cycles, sequencer module 223	Instruction Router 311	Sequencer Module 223
spioackdet	Acknowledge packet detected - continue command cycle till acknowledge packet completed	Instruction router 311	Host adapter modules 313, 314, 315 and 316
seebrdrdy	Board control or SEEPROM related command cycles have completed	Instruction router 311	System bus module 225 and sequencer module 223

TABLE 11

Signals to Serial Port from Data Transfer Circuit

Signal Name in VERILOG code	Function	From	To
ackdet	Start of acknowledge packet detected,	Serial port 230	Master serial port input-output circuit 210
xbsy	X Resource controller is currently active and service for a command signal is currently in progress.	Resource controller	Instruction router 311
xread	Command executor 320 is to read data from resource X.	X Resource controller	Command executor 320
xreq	Service request for resource X.	X Resource controller	Command executor 320
xsend	Command executor 320 is to write data to resource X.	X Resource controller	Command executor 320

Signal Name in VERILOG code	Function	From	To
chprstbsy	Chip reset is currently active	Initialization resource controller 315	Command executor 320
chprstreq	Service request for chip reset	Initialization resource controller 315	Command executor 320
clk40b1	Clock signal for various parts of host adapter 240		Command executor 320
eiread	Command executor 320 is to read initialization packets from line SPIO-; this command signal triggered by signal RST#	Initialization resource controller 315	Command executor 320
ledstate	State of LED	LED controller in hardware resource controller 313	command executor 320

Signal Name in VERILOG code	Function	From	To
noack	Timeout before receipt of first bit of an acknowledge packet	Command executor 320	MSPIOC 210
rstspios_	Power on reset	Initialization resource controller 315	Command executor 320
selroma0	Select first ROM address	Memory resource controller 316	Command executor 320
selroma1	Select second ROM address	Memory resource controller 316	Command executor 320
stpe	SCSI termination power enable	Hardware resource controller 313	Command executor 320
stpl	SCSI termination power level	Hardware resource controller 313	Command executor 320

Signal Name in VERILOG code	Function	From	To
cddat	Data for a destination address	Data transfer bus 226	Command executor 320
spiowdat	Data to be written into soft resource register 341	Data transfer bus 226	Soft resource controller 341
romadr	Address of memory location to be accessed in ROM	Memory resource controller 316	Command executor 320
xack	Resource X command acknowledge has been detected	Command executor 320	Various resource controllers in master serial port input-output circuit 210
spnakabrt	noack signal to system abort logic	Command executor 320	System bus module 225

Signal xbsy remains true until the current command service is completed. Examples of signal xbsy include brdbsy, chprstbsy, rombsy, sbwcbsy, seebsy, softbsy and spiobsy.

Signal ackdet indicates that a command packet and any additional packets were successfully sent to slave serial port input-output circuit 254.

Command signal xread triggers signal readspio. Examples of signal xread include brdread, romread, sbwcread, seeread softread and eiread.

Examples of signal xreq include brdreq, chprstreq, ledreq, romreq, seereq, softreq and stpwreq.

Command signal xsend triggers signal sendbyte. Examples of signal xsend include brdsend, romsend, seesend and softsend.

Examples of command signal xack include: brdack, chprstack, ledack, romack, seeack, softack and stpwack.

To summarize, in response to an instruction signal from a module of host adapter 240, master serial port input-output circuit 210 drives a command signal active and in response to the active command signal, command executor 320 generates a command packet and transmits the command packet serially on a single pin 241 that is connected to a slave serial port input-output circuit 254. Hence, host adapter 240 accesses a resource connected to slave serial port input-output circuit 254 by use of just one pin 241. So, host adapter 240 can perform read and write operations to an external register or memory, in addition to turning on and off various external resources, using a serial port 230 with a serial port input-output bus 246 that interfaces to other portion of host adapter 240 and pin 241 that interfaces to the external registers or memory.

Host adapter 240 does not need pins dedicated to access certain resources that were otherwise necessary in prior art host adapters to access various resources for example EEPROM 390. The use of only one pin 241 for serial communication with slave serial port input-output circuit 254 is the smallest possible number of

pins for serial communication, and so this invention reduces costs as noted above.

In response to an active command signal on one of serial port command lines 323, a byte generator 510 (FIG. 5) in command executor 320 generates a byte to be transmitted, such as a command byte 410 (FIG. 4A). Command byte 410 includes four command bits C0-C4 preceded by a read-write bit R and followed by two data bits D0-D1. The specific order of these bits is not important to practicing this invention. Table 12 lists the command byte format used by one embodiment of byte generator 510. Byte generator 510 is sometimes referred to herein as a command generator.

TABLE 12

Signal on command request line COMMANDREQ_I (FIG. 6) [PRIORITY 1 = Highest]	Command Byte Format (FIG. 4A)				Additional Bytes to be sent	Response Bytes
	Bit R 0 = Write 1 = Read	Bits C0-C4	Bits			
			D1	D0		
RST# [1]	NONE when RST# becomes inactive, command is automatically performed				0	2 (IDDAT/ ESTAT)

CHPRSTREQ [2]	All 11 bits zero				one bit of zero after 11 zero bits	0
ROMREQ [3]	1	03h	A17	A16	2 addr	4
	0	04h	A17	A16	2 addr and 1 data	0
BRDREQ [4]	1	07h	0	0	0	1
	0	07h	0	0	1 data	0
SEEREQ [5]	1	06h	0	0	0	1
	0	06h	0	0	1 data	0
STPWREQ [6]	0	02h	(a)	(b)	0	0
SOFTREQ [7]	(d)	10-1Fh	(d)	(d)	(d)	(d)
SOFTREQ (test mode) [7]	0	01h	1	0	0 (e)	0
SOFTREQ (SPIOSTAT[7]	1	10h	0	1	0	1
	0	10h	0	0	1 data	0
SOFTREQ (SPIO ERREGEN) [7]	1	11h	0	1	0	1
	0	11h	0	0	1 data	0

LEDREQ [8]	0	01h	0	(c)	0	0

(a) state of bit STPWEN;

(b) state of bit STPWLEVEL;

(c) change in status of input-output bus 284 if bit DIAGLEDEN is a zero or state of bit DIAGLEDON if bit DIAGLEDEN is one;

(d) to be defined at time of use;

(e) a real-time internal signal of data transfer circuit 220 is passed to pin 241;

In response to an active write signal on a command send line COMMANDSEND_I in a set of command send lines COMMANDSEND_1, COMMANDSEND_2, ... COMMANDSEND_N, within serial port command bus 323, command send logic element 620 (FIG. 6), which in this embodiment is a logic OR gate, drives a send byte signal SENDBYTE active on one of control terminals 512 that are connected to input control bus 521 of converter 520 (FIG. 5). The signals on command send line COMMANDSEND_I can be, for example one of signals xSEND, such as board send signal BRDSEND or ROM send signal ROMSEND.

When a resource controller, such as a hardware resource controller 313 (FIG. 3), drives a signal xREAD on a command read line COMMANDREAD_I, the same resource controller also drives a signal xREQ active on one of command request lines COMMANDREQ_I in a set of command request lines COMMANDREQ_1 ... COMMANDREQ_N within serial port command bus 323. The set of command request lines are input lines to command prioritizer 630. If only one signal on one of command request lines COMMANDREQ_I is active, command prioritizer 630 drives a signal xACK active on a command acknowledge line COMMANDACK_I, in a set of

command acknowledge lines COMMANDACK_1 ... COMMANDACK_N and also drives a signal active on a command pending line COMMANDPEND_I in a set of command pending lines COMMANDPEND_1, ... COMMANDPEND_N.

If more than one line in the set of command request lines carries an active signal when a signal on a command request line goes active, command prioritizer 630 (FIG. 6) drives active signals on the command acknowledge line and the command pending line corresponding to the highest priority command request line that has an active signal. Assuming that among the command request lines that carry an active signal, the command request line COMMANDREQ_I has the highest priority, command prioritizer 830 drives a signal active on command acknowledge line COMMANDACK_I that is part of status bus 322 and a signal active on a command pending line COMMANDPEND_I. Command prioritizer 630 prioritizes the command request lines in the order of decreasing priority as listed in TABLE 12 above and processes any remaining active command signals sequentially according to the priority of the active command signal.

In response to an active signal on a command pending line COMMANDPEND_I, command encoder 640 supplies an 8-bit command code on command code bus CMDCODE to extension multiplexer 670. Specifically, each active signal on a command pending line COMMANDPEND_I to command encoder 640 addresses a different location in a command look-up table. The byte passed through extension multiplexer 670 is determined by the signal on a port busy line PORTBUSY that is the output line of command busy logic element 650 which in turn is driven by the signals on command busy lines COMMANDBSY_1 ... COMMANDBSY_N. For active signals LEDREQ or STPWREQ, command encoder 640 uses a LED state signal LEDSTATE, to encode bit D0, or

the SCSI termination power level signal STPL and the SCSI termination power enable signal STPE respectively to encode bits D0 and D1 of the 8-bit command code. In the embodiment, command busy logic element 650 is a logic OR gate.

Since initially, all of the signals on the command busy lines are inactive, extension multiplexer 670 passes the signals on command code bus CMDCODE as the signals on hardware command data bus HWCDAT that is a first input bus of write data multiplexer 680. The byte passed through write data multiplexer 680 is determined by the signal on soft cycle line SOFT that is the output of soft data multiplexer 690 that is a logic OR gate in this embodiment. Since initially, signals SOFTACK and SOFTBSY of this embodiment on command acknowledge line COMMANDACK_N and command busy line COMMANDBSY_N are inactive, the signal on soft cycle line SOFT is also inactive and write data multiplexer 680 passes the signals on hardware command data bus HWCDAT as the signals on shift out bus SHFTOUT that is coupled to shift-out byte terminals 514.

In response to an active send byte signal SENDBYTE, on an input control bus 521 (FIG. 5) that is coupled to control terminals 512, a packet input-output controller in converter 520 formats the byte received on parallel bus 523, that is coupled to data terminals 514, into a packet and transmits the packet serially on serial data terminal 527 that is connected to serial data line 532 of line controller 530. Simultaneously, a shifter state machine in converter 520 drives a control signal on drive terminal 525 that is connected to a drive line 531 of line controller 530.

In response to an active signal on drive line 531, line controller 530 passes the signal on serial data line 532 to pin 241.

When the signal on drive line 531 goes inactive, line controller 530 stops driving the signal on pin 241 and couples pin 241 via an input buffer (not shown in FIG. 5) to serial data received terminal 536 that is connected to a serial data received line 522 of converter 520.

In response to an active read signal on a command read line COMMANDREAD_I in a set of command read lines COMMANDREAD_1, COMMANDREAD_2 ... COMMANDREAD_N within serial port command bus 323, command read logic element 610 drives a read serial port signal READSPIO active on one of control terminals 512 coupled to converter 520 (FIG. 5). A read signal can be, for example one of resource read signals xREAD, such as board read signal BRDREAD and ROM read signal ROMREAD. In this embodiment, command read logic element 616 is a logic OR gate.

In response to an active read serial port signal READSPIO, a packet controller in converter 520 retrieves a byte from a packet of data received serially on serial data received line 522 and supplies the retrieved byte on received data terminals 524 that are coupled to received data bus 321 of command executor 320 (FIG. 3).

If the received packet is an acknowledge packet, during reception of the packet converter 520 drives an acknowledge detect signal ACKDET active on transmit status terminal 526 that is coupled to a transmit status line 515 of byte generator 510. In response to an active acknowledge detect signal ACKDET, byte generator 510 disables an acknowledge timer. If the acknowledge timer times out, byte generator 510 drives a no-acknowledge signal NOACK active on status bus 322 to abort the command cycle currently being executed.

If during the current command cycle, additional bytes are to be transferred, such as a data byte for a

byte write command cycle, the respective resource controller, such as hardware resource controller 313 drives a signal SELBYTE active on select byte line 661 to indicate to command cycle extender 660 which of two address bytes on serial port address bus 324 and the data byte on serial port data bus 325 is to be passed to extend command bus EXTCMD (FIG. 6). In such a case, extension multiplexer 670 supplies the signals on extension command bus EXTCMD to hardware command data bus HWCDAT because port busy line PORTBUSY has an active signal.

The active signal on port busy line PORTBUSY is generated due to an active signal on one of command busy lines COMMANDBSY_I related to the command cycle currently in progress. Write data multiplexer 680 passes the data byte or the address byte, that is selected by the signal on select byte line SELBYTE, to shift-out byte terminals 514. Converter 520 formats the selected byte into a packet which is transmitted by line controller 530 in a manner similar to that described above for transmission of the command byte.

Instead of selecting a data byte from serial port data bus 325, byte generator 510 can supply a data byte from serial port write data bus SPIOWDAT when the current command cycle relates to a soft command. For a soft command, when a signal on command acknowledge line COMMANDACK_N and on command busy line COMMANDBSY_N goes active, soft logic element 690 drives a signal active on soft cycle line SOFT that in turn causes write data multiplexer 680 to couple the serial port write data bus SPIOWDAT to the shift-out byte terminals 514.

In one embodiment, in response to an active signal on a parity error check enable line in error control bus 550, converter 520 checks the parity of every packet received from line controller 530 and in case of

an error, drives a signal active on a parity error terminal 529 that is connected to status bus 322. In response to an active signal on a clear error flag line in error control bus 550, converter 520 drives a signal inactive on parity error terminal 529 that is coupled to a line in status bus 322. Moreover, during execution of a command cycle, if converter 520 times out, for example due to non-receipt of an acknowledge packet, converter 520 drives a signal active on a command error terminal 528 active that is connected to status bus 322.

One embodiment of converter 520 includes a shifter state machine 720 (FIG. 7) that creates a number of signals that control operation of a packet controller 740 that is also included in converter 520. In response to an active signal SENDBYTE on input control bus 521 as illustrated in FIG. 9A at time T1, shifter state machine 720 transitions from an idle state 810 (FIG. 8) along a branch 812 to a send state 820 and drives a serial port enable signal SPIOEN active on one of packet controller terminals 726 that are connected to packet controller bus 742 of packet controller 740. In response to an active serial port enable signal SPIOEN, packet controller 740 passes a byte of data received on parallel bus 523 serially out on serial data terminal 527, with a start bit preceding the byte and a parity bit and a stop bit following the byte, which is passed by line controller 530 (FIG. 5) to pin 241 (FIG. 3) as illustrated in FIG. 9A between times T2 and T3.

Shifter state machine 720 also drives a counter-on signal CNTRON (FIG. 9A) active on counter terminal 723 (FIG. 7) that is connected to a counter input line 713 of counter 710. Counter 710 increments a count as long as a counter-on signal CNTRON is active at a rising edge of clock signal CLK40B. Counter 710 supplies

various count signals, CNT0, CNT8, CNT9 and CNTA after 0, 8, 9 and 10 clock cycles respectively for which counter-on signal CNTRON is active.

The signals that trigger or are driven by shifter state machine 720 are listed in TABLE 13.

TABLE 13

SIGNAL	FUNCTION
NOACK	No ack packet received within a predetermined time period for an acknowledge window timeout
EITO	No initialization information received
READSPIO	To read information on serial port pin
CNTRON	Allows counter to increment
ACKDET	Ack packet has been received
STRTDET	Start bit has been received
SETNOACK	Drive signal NOACK active at rising edge
ENEITO	Enable signal EITO at rising edge
CNT0	Counter has not yet incremented
CNT8	Counter has incremented 8 times
CNT9	Counter has incremented 9 times
CNTA	Counter has incremented 10 times
SENDBYTE	To send a byte
BYTESENT	Byte has been sent
SPIOEN	Enable output on serial port pin
TO	((ENEITO) and (CNTO)) or (($\overline{\text{ENEITO}}$)) and ((CNTA))

SIGNAL	FUNCTION
ACKWIN	Wait window for ack packet
SETEITO	Drive signal EITO at rising edge
STRTWIN	Wait window for start bit

In FIG. 8 when a number of signals are required for a transition or alternatively a number of signals are driven active, the signals are listed separated by a comma ",". When signals are connected by a slash "/", the signals preceding the slash are triggers for shifter state machine 720 to drive the signals following the slash. Also when two signal names are connected by an "=", a signal preceding the "=" goes active when the signal following the "=" goes active.

In this embodiment, when counter signal CNT9 from counter 710 goes active a generated output parity bit PARO is shifted out to line SPIOO, by packet controller 740 and shifter state machine 720 transitions from send state 820 to stop state 830 (FIG. 8). During transition, shifter state machine 720 drives counter-on signal CNTRON inactive, continues to drive serial port enable signal SPIOEN active. In response to active serial port enable signal SPIOEN and active byte send signal BYTESENT, packet controller 740 clocks out a stop bit on pin 241 to complete the 11 bits for a packet. In response to inactive counter-on signal CNTRON, counter 710 drives counter signal CNTA inactive and counter signal CNT0 active.

Depending on the command cycle being executed, if send byte signal SENDBYTE is active when shifter state machine 720 is in stop state 830, for example, to write a data packet 462 (FIG. 4F), shifter state machine 720 transitions back to send state 820 (FIG. 8). During the transition, shifter state machine 720 drives serial

port enable signal SPIOEN active and also drives counter-on signal CNTRON active to restart counter 710 (FIG. 7).

In stop state 830, if a read serial port signal READSPIO is active on input control bus 521 (FIG. 5), shifter state machine 720 transitions to the wait-for-ack state 840, at time T3 (FIG. 9B).

Shifter state machine 720 can also transition from idle state 810 directly to wait-for-ack state 840 via branch 811 if the read serial port signal READSPIO is active, for example, immediately after reset, to receive device identification byte IDDAT and external resource status byte ESTAT.

While shifter state machine 720 is in wait-for-ack state 840, shifter state machine 720 drives an acknowledge window signal ACKWIN active on packet controller terminal 726 (FIG. 7). In response to the active acknowledge window signal ACKWIN, packet controller 740 waits for an active acknowledge detect signal ACKDET that indicates receipt of a first bit of an acknowledge packet on serial data received line 522. In response to an active acknowledge detect signal ACKDET, shifter state machine 740 transitions from wait-for-ack state 840 (FIG. 8) to end-of-ack state 850 and drives signal CNTRON inactive.

While in wait-for-ack state 840, if shifter state machine 720 times out, e.g. after ten clock cycles, shifter state machine 720 transitions via branch 841 to idle state 810 and drives set no acknowledge signal SETNOACK, set initialization timeout signal SETEITO active, counter-on signal CNTRON inactive, and acknowledge window signal ACKWIN inactive during the transition. In response to an active set no acknowledge signal SETNOACK, signal synchronizer 730 (FIG. 7) drives a no acknowledge signal active on status terminal 733 that is connected to a status

line 721 of shifter state machine 720, at the next rising edge of a clock signal CLK40B that is a buffered version of clock signal CLK40 driven by oscillator 260 (FIG. 2A).

In end-of-ack state 850, shifter state machine 720 continues to drive acknowledge window signal ACKWIN active until acknowledge detect signal ACKDET goes inactive. When packet controller 740 receives the last bit of the acknowledge packet, also referred to as the third bit, packet controller 740 drives the acknowledge detect signal ACKDET inactive.

In response to an inactive acknowledge detect signal ACKDET, if read serial port signal READSPIO is inactive, shifter state machine 720 drives acknowledge window signal ACKWIN inactive and transitions to idle state 810 via branch 851. Branch 851 is a normal transition for shifter state machine 720 for certain command cycles, in which no further bytes are to be received, such as the bit write command cycles triggered by the command signals STPWREQ and LEDREQ.

In response to an inactive acknowledge detect signal ACKDET, if read serial port signal READSPIO is active, for example, during execution of a byte read command cycle illustrated in FIG. 4G, shifter state machine 720 transitions to start state 860 (FIG. 8) and drives acknowledge window signal ACKWIN inactive, as illustrated in FIG. 9C at time T6.

In start state 860, shifter state machine 720 drives a start window signal STRTWIN active as long as a start detect signal STRTDET remains inactive. In response to active start window signal STRTWIN, packet controller 740 becomes sensitive to a start bit of a packet. In response to a start bit, packet controller 740 drives the start detect signal STRTDET active on receive status terminals 526 that are coupled to receive status bus 722 of shifter state machine 720.

In response to an active start detect signal STRTDET (FIG. 9C), shifter state machine 720 drives the start window signal STRTWIN inactive and also drives a counter-on signal CNTRON active on counter terminal 723 and transitions to read data state 870, for example as illustrated at time T7 of FIG. 9C.

In read data state 870, shifter state machine 720 drives the counter-on signal CNTRON active and also drives a set shift-in valid signal SETSIVAL active on set shift-in terminal 725 (FIG. 7) that is coupled to set shift-in line 741 and waits for counter signal CNT9 to go active. In response to an active set shift-in valid signal SETSIVAL on set shift-in line 741, packet controller 740 clocks in serial port input signal SPIOI serially from serial data received line 522 into a shift register (not shown in FIG. 7).

Nine clock cycles after counter-on signal CNTRON goes active, counter 710 drives counter signal CNT9 active. At this point, packet controller 740 has clocked in a total of ten bits from the serial data received line 522, i.e., a start bit before counter-on signal CNTRON goes active, followed by a byte, and a parity bit after the byte.

In response to an active counter signal CNT8, shifter state machine 720 drives set shift-in valid signal SETSIVAL active to enable parity of the byte in the received packet to be checked. In response to an active counter signal CNT9, shifter state machine 720 transitions to check read state 880 (FIG. 8) and drives counter-on signal CNTRON inactive, as illustrated at time T8 of FIG. 9C. In check read state 880 the parity is checked as described below, and if read serial port signal READSPIO is active, for example, during the execution of a memory read command cycle illustrated in FIG. 4I, shifter state machine 720 returns to start

state 860 (FIG. 8). If the read serial port signal READSPIO is inactive, shifter state machine 720 returns to idle state 810.

Packet controller 740 includes a shifter circuit 1010 and a parity circuit 1020. In response to an active send byte signal SENDBYTE, parity circuit 1020 loads the value of signal SENDBYTE as a start bit in to send output register 1023.

In response to an active chip reset signal CHPRSTBSY, parity circuit 1020 drives a serial port output signal SPIOO active (low) on serial data terminal 527. In response to the active chip reset signal CHPRSTBSY, line controller 530 passes serial port output signal SPIOO to pin 241 as illustrated in FIG. 12B.

In response to the active send byte signal SENDBYTE, on the next rising edge of buffered clock signal CLK40B a parallel input multiplexer 1012 in shifter circuit 1010 passes shift out signals SHFTOUT[7:0] on parallel input terminal D1 that is connected to parallel bus 523 simultaneously, in parallel, one bit to each stage in a shift register 1014 also included in shifter circuit 1010.

Stages 1014_1, 1014_2, ... 1014_I, ... and 1014_N of shift register 1014 are connected in series so that a bit stored in a stage, such as stage 1014_N is transferred to a successive stage, in the direction from 1014_N to 1014_1 with stage 1014_N as the serial shift-in stage and stage 1014_1 as the serial shift-out stage on each rising edge of signal CLK40B. In FIG. 10, there are a total of eight stages in shift register 1014. On each rising edge of buffered clock signal CLK40B, the bit stored in the last stage 1014_1 drives a signal on output terminal 1011 that is coupled to an input line 1021 of parity circuit 1020.

In one embodiment, a stage, such as one of stages 1014_1, ... 1014_N is implemented by a two to one multiplexer 1110 (FIG. 11) and a storage element such as flip-flop 1120. On each rising edge of buffered clock signal CLK40B, a signal from the previous stage drives terminal D0 of multiplexer 1110, except that serial port input signal SPIOI received at pin 241 drives the first stage. A signal from input multiplexer 1012 (FIG. 10) drives terminal D1 of multiplexer 1110. Moreover, signal SENDBYTE drives terminal S0 of multiplexer 1110. In response to active signal SENDBYTE, multiplexer 1110 passes the signal on terminal D1 to flip-flop 1120 and otherwise supplies the signal on terminal D0 to flip-flop 1120.

If read serial port signal READSPIO is inactive, parity circuit 1020 passes a signal on input line 1021 to serial data terminal 527. Parity circuit 1020 computes the parity of the signals on input line 1021. In response to active counter-on signal CNT9, parity circuit 1020 clocks a parity bit on serial data terminal 727 based on the computed parity for the packet out parity bit. In response to an active byte sent signal BYTESENT, parity circuit 1020 clocks a stop bit on serial data terminal 527 to complete transmission of packet.

When byte error enable signal BERREN goes active on an error control bus 550, parity circuit 1020 passes the incorrect parity bit to serial data terminal 527, and so exercises the parity circuit in a slave serial port input-output circuit 254. In response to an active read serial port signal READSPIO after completing the transmission of the packet, shifter state machine 720 drives serial port enable signal SPIOEN inactive. In response to the inactive serial port enable signal SPIOEN, line controller 530 three states output driver 1220 (FIG. 12B), that is connected

to pin 241. So, output signal from pin 241 is inhibited and pin 241 is usable as an input pin and line controller 530 waits for an acknowledge packet to be returned from SSPIOC 254 to pin 241 and passed as signal SPIOI by input buffer 1223.

When send byte signal SENDBYTE is inactive, parallel input multiplexer 1012 only transfers serial port input signal SPIOI from serial port input line 522 that is coupled to pin 241, to the input pin of first stage 1014-N of shift register 1014.

In response to an active set shift-in valid signal SETSIVALID on set shift-in valid line 741, an output multiplexer 1016 simultaneously, in parallel, passes the bits shifted in from pin 241 and stored in each of the stages of shift register 1014 into an eight-bit hold register 1018. Hold register 1018 in turn passes the received bits as signals SHFTIN[7:0] to received data terminals 524. Shift register 1014 also passes the received bits to output terminal 1011. When shift-in valid signal SIVALID goes active, and if a signal MPARCKEN is active on a parity error control bus 550, parity circuit 1020 compares the computed parity with a received parity bit on input line 1021 and drives a parity error detected signal SPIOPARERR active on packet error terminal 529 in case of errors. When clear parity error signal CLRPARERR- goes active on error control bus 550, parity circuit 1020 drives a serial port parity error signal SPIOPARERR inactive until the next parity error is detected

Although one embodiment of a parity circuit 1020 is illustrated in FIGs. 10 and 11, any other parity circuit can be used in accordance with this invention.

One embodiment of line controller 530 (FIG. 5) includes a control enabled output driver 1210 (FIG. 12A) that is controlled by drive signal SPIODRV on drive line 531. In response to an active drive

signal SPIODRV, output driver 1210 passes a serial port output signal SPIOO from serial data line 532 to pin 241. An inactive drive signal SPIODRV disables output driver 1210 and so provides a high impedance to pin 241 and input buffer 1220 passes the signal received on pin 241 as serial port input signal SPIOI to serial data line 722.

In response to an active LED test signal LEDTST, line controller 530 passes one of eight internal signals TSTSIGNI[7:0] that is selected by the value of bits TESTSEL[2:0] described above, to output driver 1220. In response to the active LED test signal LEDTST, output driver 1220 passes the selected internal signal to pin 241.

In response to an active package input signal PKG1- that is passed by bond wire 249 of FIG. 2B, line controller 530, drives the serial port input signal SPIOI inactive (e.g. one) irrespective of the signal on pin 241.

In response to an inactive status signal SSPIOCPS, line controller 530 uses two multiplexers 1230 and 1240 (FIG. 12B) and a flip-flop 1250 to switch the connection of pin 241 to either (1) drive line 531 and serial data line 532 or (2) default command bus 533 that carries bus termination signals STPWLEVEL and STPWEN, depending on whether or not status signal SSPIOCPS on default command bus 533 is active or inactive respectively, which indicates the presence or absence of a slave serial port input-output circuit. In the absence of a slave serial port input-output circuit, line controller 530 drives a first bus termination signal STPWCTL on pin 241 to control turning on or off power to bus terminators of an input-output bus as described above.

A slave serial port input-output circuit 254 (FIG. 3) is implemented in one embodiment using the

VERILOG code in microfiche appendix B with Synopsys Synthesizer version 3.1 in SMOS gate array SLA 20000 available from SMOS Systems, Inc. of San Jose, CA.

In response to a start bit on slave serial port pin 341, hereinafter slave pin 341, that is connected to line SPIO-, a shift register 1310 in slave serial port in put-output circuit 254 shifts data in serially from slave pin 341 and transfers the shifted in data to sequencer 1330. In one embodiment, shift register 1310 transfers 11 bits of the shifted in data to sequencer 1330. The 11 bits comprise the received packet including a start bit, a command byte, a parity bit and a stop bit as described above. In this embodiment, when all of the 11 bits have the value zero, sequencer 1330 determines that the received packet contains a reset command byte indicative of signal CHIPRSTREQ as described above.

In another embodiment, shift register 1310 transfers 12 bits of the shifted in data to sequencer 1330. In this embodiment, if each of the 12 bits is zero, sequencer 1330 interprets the 12 bits as the reset command byte.

In response to a packet from shift register 1310, sequencer 1330 transitions through a sequence of states that is specific to the command byte in the received packet and generates control signals that drive command executor 1370. In response to a packet that contains a reset command packet, sequencer 1330 that is waiting in wait-for-start state 1420 FIG. 14) initializes all of the state variables and continuous in wait-for-start state 1420.

In one embodiment, sequencer 1330 is encoded as a "one-hot" state machine 1400 (FIG. 14) having flip-flops associated with each of 36 states as listed at page 19 of Microfiche Appendix B.

In response to a reset signal on a reset pin 1501 (FIG. 15) that is coupled to a reset line of system bus 250, such as the PCI bus, the slave serial port input-output circuit 254 is reset and sequencer 1330 starts in a hard reset state 1410, initializes variables, goes to reset acknowledge state 1411 and signals command executor 1370 to receive device identification byte IDDAT and external resource status byte ESTAT from for example, a programmable logic circuit 330.

In reset acknowledge state 1411, sequencer 1330 also causes acknowledge multiplexer 1320 to send an acknowledge packet on pin 341 that is coupled to line SPIO- for as long as sequencer 1330 stays in reset acknowledge state 1411. Sequencer 1330 stays in reset acknowledge state 1411 for a variable number of clock cycles depending on the time needed by ID-ESTAT detector 1375 which depends on, for example presence or absence of programmable logic circuit 330 (FIG. 3).

In response to a reset signal on pin 1501, ID-ESTAT detector 1375 clocks in bytes IDDAT and ESTAT from a programmable logic circuit 330. While waiting for bytes IDDAT and ESTAT, if ID-ESTAT detector 1375 times out, for example, due to absence of programmable logic circuit 330, then ID-ESTAT detector 1375 uses a default byte as byte IDDAT and creates byte ESTAT by sensing the signals on various pins for presence or absence of pull up or pull down resistors on plug-in board 270, as described below in reference to FIG. 16. ID-ESTAT detector 1375 then signals sequencer 1330, when bytes IDDAT and ESTAT are assembled and ready for transmission to host adapter 240.

In response to the signal from ID-ESTAT detector 1375, sequencer 1330 transitions to send identification state 1412 and supplies signals for multiplexer 1360 to pass byte IDDAT to shift

register 1310. In send identification state 1412, sequencer 1330 loops back for ten clock cycles, indicated in FIG. 14 by "X10" for a total of eleven clock cycles in send identification state 1412. In send identification state 1412, shift register 1310 clocks out byte IDDAT in an initialization packet on pin 341. In one embodiment, an exclusive OR gate (not shown) computes and stores a parity bit for a byte stored in shift register 1310 on a clock cycle immediately subsequent to the clock cycle in which the byte is stored.

Then sequencer 1330 transitions from send identification state 1412 to send external status state 1413, waits for twelve clock cycles during which time shift register 1310 clocks out byte ESTAT as a packet on pin 341 and then transitions to wait-for-start state 1420.

In response to a signal from shift register 1310 indicating receipt of a start bit that is denoted in FIG. 14 as "SB", sequencer 1330 transitions from wait-for-start state 1420 to shift-in-command state 1421. Sequencer 1330 loops back in shift-in-command state 1421 for nine clock cycles while shift register 1310 clocks in the rest of the packet from pin 341.

If sequencer 1330 is unable to decode the received command, sequencer 1330 transitions back to wait-for-start state 1420.

If sequencer 1330 receives a command byte originating from a valid command signal such as LED request signal LEDREQ or bus terminator request STPWREQ, sequencer 1330 transitions to write data bits state 1430, and causes board control register and monitoring circuitry 1374 in command executor 1370 to use bits D0 and D1 from the command byte to drive signals to the corresponding resources such as LED 350

or bus terminators 360. Simultaneously, sequencer 1330 also causes acknowledge multiplexer 1320 to transmit an acknowledge packet on pin 341. On completion of the transmission of the acknowledge packet, sequencer 1330 returns to wait-for-start state 1420.

In response to other command packets, sequencer 1330 transitions to the corresponding state, such as write byte wait start state 1440 eewrite-wait-low address state 1460 and eeread-wait-low address state 1470. The actions of sequencer 1330 in such states are similar to the actions described above in respect to write data bits state 1430, except that sequencer 1330 causes shift register 1310 to clock in additional address or data packets as necessary and also causes command executor 1370 to perform actions indicated by the respective command bytes such as writing or reading internal registers BRDCTL or SPIOSTAT or memories ROM or EEROM.

In one embodiment if there are any errors, sequencer 1330 sets a bit in a status register and causes command executor 1370 to turn-on LED 350 (FIG. 3).

In one embodiment, command executor 1370 includes board control register and monitoring circuitry that detects the presence or absence of cables connected and terminators installed for input-output bus 284 (FIG. 4) and so there is no need for board control logic 370 external to slave serial port input-output circuit 254.

Command executor 1370 also includes an ID-ESTAT detector 1375 that (1) receives device identification byte IDDAT from a programmable array logic, henceforth PAL, and (2) senses signals on various pins such as memory address pins MA16 and MA17 (FIG. 16) for ROM, to detect the presence or absence of various predetermined devices on support circuit 250, such as EEPROM 390, SEEPROM 380, board control logic 370, bus

terminators 360 and LED 350 (FIG. 3). Any pin of slave serial port input-output circuit 254 that is normally connected to a predetermined device carries, an active signal when the predetermined device is present on support circuit 250 and an inactive signal to indicate absence of the predetermined device, as described below in reference to FIG. 16.

In addition to predetermined devices, other devices can also be sensed and controlled by slave serial input-output circuit 254. For example, in response to command SOFTREQ (Table 3) serial port 230, can send a command packet that causes slave serial port input-output circuit 254 to merely passes to a soft resource 341 information contained in packets received subsequent to the command packet and so allows future expansion in the numbers and types of soft resources included in support circuit 250 (FIG. 3).

One example of a soft resource 341 is a debugger that is polled by sequencer module 223 during boot-up and if present, sequencer module 223 loads in additional firm ware or data or both firmware and data from debugger 341, executes the loaded firmware and writes status to debugger 341.

In one embodiment, host adapter 240 and slave serial port input-output circuit 254 are coupled to impose a single load on system bus 250, such as a PCI bus as illustrated in FIG. 15. A reset input terminal 1501 of slave serial port input-output circuit 254 is coupled to a system bus reset line of system bus 283 that carries signal PCIRST-.

In response to an active signal PCIRST-, slave serial port input-output circuit 254 buffers signal PCIRST- in input buffer 1502 for internal use by flip-flop 1505, for example to generate reset command signal CHPRSTREQ. Slave serial port input-output circuit 254 buffers the buffered signal PCIRST- in

output buffer 1503 and drives signal PCIRSTB- on output terminal 1504 for external use by host adapter 240.

Host adapter 240 receives signal PCIRSTB- from slave serial port input-output circuit 254 at reset input pin 1521 that is different from serial port pin 241. Host adapter 240 passes signal PCIRSTB- through input buffer 1522 and reset line RSTIB- to serial port 230. Master serial port input-output circuit 210 uses flip-flops 1541 and 1542 to double synchronize the reset signal from input buffer 1522.

In response to an active signal on reset line RSTIB-, serial port 230 resets all state machines to their initial states. In response to command signal CHIPRSTREQ, serial port 230 generates a packet containing a reset command code that resets slave serial port input-output circuit 254.

As noted above, slave serial port input-output circuit 254 senses the presence or absence of various resources from the signals on one or more lines connected to the resource. For example, when ID-ESTAT detector 1375 senses a low signal on memory chip select line MCS# (FIG. 16), ID-ESTAT detector 1375 determines that there is no EEPROM 390 that is coupled to slave serial port input-output circuit 254. Similarly, when ID-ESTAT detector 1375 senses a high signal on SEEPROM chip select line SEECS (FIG. 16), ID-ESTAT detector 1375 determines that serial EEPROM 380 is absent. In the absence of EEPROM 390 or serial EEPROM 380, memory chips select line MCS# or serial EEPROM chip select line SEECS is coupled for example by a resistor to a first voltage e.g. high or a second voltage e.g. low respectively.

In one embodiment, in response to a high signal on memory chip select line MCS#, ID-ESTAT detector 1375 senses signals at a number of high address lines, such as first memory address line MA17 and second memory

address line MA16. If ID-ESTAT detector 1375 senses a low signal on first memory address line MA17 and also a low signal on second memory address line MA 16, ID-ESTAT detector 1375 determines that the size of EEPROM 390 is only 64 kilobytes. If ID-ESTAT detector 1375 senses a high signal on second memory address line MA16, and either (1) a low signal on first memory address line MA17 or (2) a high signal on first memory address line MA17, the size of EEPROM 390 is 128 kilobytes or 256 kilobytes respectively.

In response to a low signal on memory write line MWR#, ID-ESTAT detector 1375 determines that a read only memory ROM is coupled to slave serial port input-output circuit 254, rather than an EEPROM.

FIG. 16 illustrates certain variations in circuitry by an asterisk "*". For example, a programmable logic circuit (PAL) 330 is directly connected to line SPIO- in the absence of slave serial port input-output circuit 254 to provide device identification byte IDDAT and external resource status byte ESTAT, that is FFh in this case. Programmable logic circuit 330 provides bytes IDDAT and ESTAT encoded in packets of the type illustrated in FIG. 4A. In one embodiment, device identification byte IDDAT and external resource status byte ESTAT are visible to host processor 281 as register 00, byte 3, and as register 1Bh respectively of host adapter 240.

In this embodiment, host adapter 240 and slave serial port input-output circuit 254 operate synchronously by using clock signals derived from the same oscillator 215. Sequencer module 223 drives a signal CLK40B, that is a buffered version of the same clock signal CLK40 used for timing by other modules, such as system bus module 225 and input-output bus module 221. When a power-down bit is set in an internal register of sequencer module 223, sequencer

module 223 clamps signal CLK40B high, until the power-down bit is reset, to reduce power consumption in host adapter integrated circuit 240.

When host adapter 240 drives a signal on line SPIO-, a serial port 230 that exhibits slow characteristics can have a setup time Tss and a hold time Ths as illustrated in FIG. 17. A serial port 230 that exhibits fast characteristics can have a set time Tsf and a hold time Thf also illustrated in FIG. 17. So either a fast or a slow slave serial port input-output circuit 254 can sample signal SPIO- at rising edge Tr (FIG. 17) of a clock signal, when there is no skew in the clock signals of serial port 230 and slave serial port input-output circuit 254.

However, a slave serial port input-output circuit 254 can have a positive skew or a negative skew in its clock signal as compared to the clock signal of serial port 230 depending on the layout of plug-in board 270. Positive skew or negative skew can result in EARLY IN CLOCK or LATE IN CLOCK respectively, with the set up and hold times for fast characteristics and slow characteristics centered around the respective rising edges as shown in FIG. 17. So a serial port 230 that exhibits fast characteristics must output no faster than a slave serial port input-output circuit 254 that exhibits slow input characteristics. Examples of timing characteristics are listed in Tables 14A-14C.

TABLE 14A

Minimum timing required by slave port 230 of host adapter 240

Serial Port 230	Slave serial port input-output circuit 254 provides:
set up: 0.66 ns	8.33 ns
hold: 2.48 ns	5.08 ns

The above set-up times are also applicable when a programmable logic circuit is directly connected to pin 241 instead of a slave serial port input-output circuit 254 to provide bytes IDDAT and ESTAT to host adapter 240.

TABLE 14B

Minimum timing provided to slave serial port input-output circuit 254

Serial Port 230	Slave serial port input-output circuit 254 requires:
set up: 3.38 ns	.97 ns
hold: 6.99 ns	2.07 ns

The above setup times in tables 14A and 14B are based on a 40 MHz clock with 25 ns cycle and output timing is measured with a 15 pF load.

In one embodiment, programmable logic circuit 330 passes bytes IDDAT and ESTAT into SSPIOC 254 after reset. SSPIOC 254 then passes these bytes to host adapter 240. Table 14C shows the timing requirements imposed on programmable logic circuit 330 by SSPIOC 254.

TABLE 14C

For a shift register in a programmable logic circuit

hold:	-1.93 ns
setup:	8.86 ns

One embodiment of support circuit 1800 (FIG. 18) includes a plurality of slave serial port input-output circuits, such as SSPIOC 254A, SSPIOC 254B and SSPIOC 254C that are all connected to the same serial port input-output line SPIO- that is coupled to the serial port pin 241 of host adapter 240. Each of the slave serial port input-output circuits 254A, 254B and 254C receive the same clock signals CLK40 from oscillator 260 that is also supplied to host adapter 240. Each slave serial port input-output circuit allows host adapter 240 to access a resource that is connected to that particular slave serial port input-output circuit similar to the above description in reference to SSPIOC 254 and host adapter 240. For example, host adaptor 240 can access resources 1810 and 1820 through SSPIOC 254A, resources 1830 and 1840 through SSPIC 254B and resource 1850 through SSPIOC 254C.

In the embodiment of FIG. 18, SSPIOC 254A has a design identical to the design for slave serial port input-output circuit 254 described above. Therefore, slave serial port input-output circuit 254A sends two initialization packets containing bytes IDDAT and ESTAT to host adapter 240 following reset.

The other two slave serial port input-output circuits, SSPIOC 254B and SSPIOC 254C are similar to slave serial port input-output circuit 254 except that SSPIOC 254B and SSPIOC 254C respond to command packets

containing command bytes of values different from the values listed in Table 12 above, for example command byte values 05 and 08 respectively. The command byte values to which each slave serial port input-output circuit responds is mutually exclusive from the command byte to which another slave serial port input-output circuit responds in order to eliminate the possibility of contention for line SPIO-, avoid collision of packets and eliminate need for control lines.

In another embodiment, a slave serial port input-output circuit such as SSPIOC1910A (FIG. 19) or SSPIOC1910B includes a memory port interface of the type described in "AIC-7870PCI Bus Master Single-Chip SCSI Host Adapter Data Book-Preliminary", at for example, pages 5-18 to 5-21.

In the embodiment of FIG. 19, SSPIOC 1910A and SSPIOC 1910B are coupled to host adapters 1920 and 1930 respectively and allow these two host adapters to share one or more common memory resources, such as shared resources 1951, 1952 and 1953 through a shared bus arbiter 1960. Computer system 1900 (FIG. 19) also includes a host adapter 1940 of the type described in "AIC-7870PCI Bus Master Single-Chip SCSI Host Adapter Data Book-Preliminary" referenced above that can also access shared resources 1951, 1952, and 1953 via shared bus arbiter 1960.

The use of a predetermined protocol as described above in which one or more slave integrated circuits always wait for a master integrated circuit eliminates possibility of contention for the shared serial port input-output line, avoids collision of packets and so eliminates need for control lines in addition to the shared serial port input-output line between a master integrated circuit and the slave integrated circuits.

Although the present invention has been described in connection with the above described illustrative

embodiments, the present invention is not limited thereto. For example, instead of a pin, a surface mount lead can be used. Various modification and adaptations of the above discussed embodiments are encompassed by the appended claims.

SERIAL PORT FOR A HOST ADAPTER INTEGRATED CIRCUIT USING A SINGLE TERMINAL

Stillman F. Gates
Christopher Burns

APPENDIX A

brdcntrl.v

```
/*****************************************************************************

Function:

Board Logic Control.  Brdctl command is issued whenever brdctl (1d) is written
or read.

*****************************************************************************/

module brdcntrl ( ackdet, brdctlen, brdctlrdy, brdack, bytesent, cddat,
        clk40b2, noack, rdbrdctl_, rstspios_, shftin, sivalid, wrbrdctl_,
        brdbsy, brdctl, brddat, brdread, brdreq, brdsend, rstbrdctl_);

parameter FFDLY = 1;

input   ackdet, brdctlen, brdack, bytesent, clk40b2, noack,
        rdbrdctl_, rstspios_, sivalid, wrbrdctl_;
input   [7:0] cddat, shftin;

output  brdbsy;         // brdctl command in progress
output  [2:0] brdctl;   // brdctl control bits
output  [2:0] brddat;   // brdctl data bits
output  brdread;        // read SPIO port
output  brdreq;         // request for SPIO port
output  brdsend;        // shift out write data
output  brdctlrdy;      // no brdctl command active
output  rstbrdctl_;     // reset brdctl logic

//-------------------------------------------------------------------
// registered outputs

reg     brdread, brdreq, brdbsy, brdsend, brdctlrdy;
reg     [2:0] brdctl, brddat;

//-------------------------------------------------------------------
// internal signals

reg     brdacts1;       // sync
reg     brdacts2;       // sync
reg     brdract;        // brdctl read active
reg     brdwact;        // brdctl write active
wire    readreq;        // brdctl read request
wire    rstbrdctl_;     // reset all brdctl logic
wire    rstreqgen_;     // reset request generation bit
wire    writereq;       // brdctl write request

wire    setread, wrdone, rdatval, setbsy, setsend;

//-------------------------------------------------------------------
// reset logic

assign rstbrdctl_ = rstspios_ & brdctlen;
assign rstreqgen_ = rstbrdctl_ & ~brdacts2;

//-------------------------------------------------------------------
// brdctl ready control

always @(posedge clk40b2 or negedge rstbrdctl_)
  if (~rstbrdctl_)
        brdctlrdy <= #FFDLY 1'b0;
  else
```

```
        brdctlrdy <= #FFDLY wrdone | rdatvat;

//-----------------------------------------------------------------
// brdctl register

// bits 7:5
always @(posedge clk40b2 or negedge rstbrdctl_)
  if (~rstbrdctl_)
        brddat <= #FFDLY 3'b000;
  else
        brddat <= #FFDLY rdatval ? shftin[7:5] : brddat;

// bits 4:2
always @(posedge wrbrdctl_ or negedge rstbrdctl_)
  if (~rstbrdctl_)
        brdctl <= #FFDLY 3'b000;
  else
        brdctl <= #FFDLY cddat[4:2];

// bits 1:0 always "0"

//-----------------------------------------------------------------
// request generation

assign writereq = brdacts1 & brdacts2 & rdbrdctl_;
assign readreq = brdacts1 & brdacts2 & ~rdbrdctl_;

always @(negedge wrbrdctl_ or negedge rstreqgen_)
  if (~rstreqgen_)
        brdwact <= #FFDLY 1'b0;
  else
        brdwact <= #FFDLY 1'b1;

always @(negedge rdbrdctl_ or negedge rstreqgen_)
  if (~rstreqgen_)
        brdract <= #FFDLY 1'b0;
  else
        brdract <= #FFDLY 1'b1;

always @(posedge clk40b2 or negedge rstbrdctl_)
  if (~rstbrdctl_) begin
        brdacts1 <= #FFDLY 1'b0;
        brdacts2 <= #FFDLY 1'b0;
        brdreq <= #FFDLY 1'b0;
  end
  else begin
        brdacts1 <= #FFDLY brdwact | brdract;
        brdacts2 <= #FFDLY brdacts1;
        brdreq <= #FFDLY writereq | readreq | brdreq & ~brdack;
  end

//-----------------------------------------------------------------
// registered outputs

always @(posedge clk40b2 or negedge rstbrdctl_)
  if (~rstbrdctl_) begin
        brdbsy <= #FFDLY 1'b0;
        brdsend <= #FFDLY 1'b0;
        brdread <= #FFDLY 1'b0;
  end
  else begin
        brdbsy <= #FFDLY setbsy;
        brdsend <= #FFDLY setsend;
        brdread <= #FFDLY setread;
  end
```

brdcntrl.v

```
//-----------------------------------------------------------------
// Instantiate state-machine

brdctlsm brdctlsm ( .clk40b2(clk40b2), .ackdet(ackdet), .brdack(brdack),
            .bytesent(bytesent), .rstbrdctl_(rstbrdctl_), .rdbrdctl_(rdbrdctl_),
            .noack(noack), .sivalid(sivalid), .setread(setread), .rdatval(rdatval),
            .setbsy(setbsy), .setsend(setsend), .wrdone(wrdone) );

//-----------------------------------------------------------------

endmodule
```

brdctlsm.v

Function:

Board Logic Control state machine.

```
*****/

module brdctlsm ( clk40b2, ackdet, brdack, bytesent, rstbrdctl_, rdbrdctl_, noack,
        sivalid, setread, rdatval, setbsy, setsend, wrdone );

parameter FFDLY = 1;

parameter [2:0]             // synopsys enum state_info
        TST0 = 3'b000,      // step through all states
        IDLE = 3'b001,      // idle, waiting for brdack
        CMDC = 3'b010,      // shifting command code
        RWAK = 3'b011,      // read command, waiting for ACK
        RDAT = 3'b100,      // reading data
        WDAT = 3'b101,      // writing data
        WWAK = 3'b110,      // write command, waiting for ACK
        EACK = 3'b111;      // write command, waiting for end of ACK

input   clk40b2, ackdet, brdack, bytesent, rstbrdctl_, rdbrdctl_, noack, sivalid;

output  setread;
output  rdatval;            // read data valid
output  setbsy;
output  setsend;
output  wrdone;             // write command is completed

//-----------------------------------------------------------------
// register declatation

reg     setread, rdatval, setbsy, setsend, wrdone;

//-----------------------------------------------------------------
// internal signals

                    // synopsys state_vector state
reg     [2:0]       // synopsys enum state_info
        state;      // current state
reg     [2:0]       // synopsys enum state_info
        next;       // next state

//-----------------------------------------------------------------
// combinational logic

always @( state or ackdet or brdack or bytesent or rdbrdctl_ or noack or sivalid )
begin

   setbsy = 1'b1;
   setsend = 1'b0;
   rdatval = 1'b0;
   setread = 1'b0;
   wrdone = 1'b0;

   case (state)             // synopsys parallel_case

        TST0: begin
                next = IDLE;
```

brdctlsm.v

```
            setbsy = 1'b0;
        end

        IDLE: begin
                next = brdack ? CMDC : IDLE;
                setbsy = brdack;
        end

        CMDC: begin
          if (bytesent) begin
                next = rdbrdctl_ ? WDAT : RWAK;
                setsend = rdbrdctl_;
                setread = ~rdbrdctl_;
          end
          else begin
                next = CMDC;
          end
        end

        RWAK: begin
          if (noack) begin
                next = IDLE;
          end
          else begin
                next = ackdet ? RDAT : RWAK;
                setread = 1'b1;
          end
        end

        RDAT: begin
                next = sivalid ? IDLE : RDAT;
                setread = ~sivalid;
                rdatval = sivalid;
        end

        WDAT: begin
                next = bytesent ? WWAK : WDAT;
                setread = bytesent;
        end

        WWAK: begin
          if (noack) begin
                next = IDLE;
          end
          else begin
                next = ackdet ? EACK : WWAK;
          end
        end

        EACK: begin
                next = ackdet ? EACK : IDLE;
                setbsy = ackdet;
                wrdone = ~ackdet;
        end

        default: next = 3'bxxx;

    endcase

end     // always

//-----------------------------------------------------------------
// state vector

always @(posedge clk40b2 or negedge rstbrdctl_)
```

```
if (~rstbrdctl_)
        state <= #FFDLY TST0;
else
        state <= #FFDLY next;

//-----------------------------------------------------------------

endmodule          // brdctlsm
```

```
/*****************************************************************************

Function:

ESTAT/IDDAT state machine.  Waits for IDDAT and ESTAT bytes.  After reading
or time-out, generate idldrdy so that the PHOST can accept PCI commands.



*****************************************************************************/

module eism ( clk40b3, elto, rstls_, sivalid, setckes, setckid, setread, eneito,
        setidrdy );

parameter FFDLY = 1;        // clock-to-q delay

parameter [1:0]             // synopsys enum state_info
        PU = 2'b00,         // power up
        ID = 2'b01,         // waiting for IDDAT valid
        ES = 2'b10,         // waiting for ESTAT valid
        DN = 2'b11;         // done

input   clk40b3;
input   elto;
input   rstls_;
input   sivalid;

output  setckes;            // clock in ESTAT
output  setckid;            // clock in IDDAT
output  setread;            // readport command from eism
output  eneito;             // enable ESTAT/IDDAT time-out
output  setidrdy;           // set idldrdy

//-----------------------------------------------------------------------
// register declarations
reg     setread, eneito, setckid, setckes, setidrdy;

                    // synopsys state_vector state
reg     [1:0]       // synopsys enum state_info
        state;      // current state
reg     [1:0]       // synopsys enum state_info
        next;       // next state

//-----------------------------------------------------------------------
// combinational logic

always @( sivalid or elto or state) begin

   setread = 1'b0;
   eneito = 1'b0;
   setckid = 1'b0;
   setckes = 1'b0;
   setidrdy = 1'b0;

   case (state)             // synopsys parallel_case

        PU: begin
          next = ID;
          setread = 1'b1;
        end

        ID: begin
          eneito = 1'b1;
```

```
            setread = ~elto;
            setckid = sivalid;
            if (elto)
                    next = DN;
            else
                    next = sivalid ? ES : ID;
          end

          ES: begin
            next = sivalid ? DN : ES;
            setread = ~sivalid;
            setckes = sivalid;
          end

          DN: begin
            next = DN;
            setidrdy = 1'b1;
          end

          default: next = DN;

    endcase

end     // always

//-----------------------------------------------------------------
// state latches

always @(posedge clk40b3 or negedge rstls_)
    if (~rstls_)
          state <= #FFDLY PU;
    else
          state <= #FFDLY next;

//-----------------------------------------------------------------

endmodule           // eism
```

```
/*****************************************************************************

Function:

ESTAT/IDDAT controller.  Contains the ESTAT/IDDAT state machine, IDDAT register
and ESTAT register.



*****************************************************************************/

module iddatestat ( cddat5,clk40b3, eito, parerrdet, rstis_, shftin, sivalid,
        swrstrdy, wrspiocap_, brdctlen, eepromen, eiread, eneito, iddat,
        iddrdy, romrden, seectlen, soft, softcmden, spbrden, spseeen,
        sspiocps );

parameter FFDLY = 1;
parameter [7:0] DEFAULTID = 8'h55;

input   cddat5, clk40b3, eito, parerrdet, rstis_, sivalid, swrstrdy, wrspiocap_;
input   [7:0] shftin;

output  brdctlen;           // board control enable
output  eepromen;           // eeprom writable
output  eiread;
output  eneito;
output  [7:0] iddat;        // IDDAT registger
output  iddrdy;             // finished loading IDDAT and ESTAT
output  romrden;            // external rom readable
output  seectlen;           // seeprom control enable
output  [1:0] soft;         // soft command capability bits
output  softcmden;          // soft command enable
output  spbrden;            // brdctl register enabled
output  spseeen;            // seectl register enabled
output  sspiocps;           // external SSPIOC present

//-----------------------------------------------------------------------
// declare registers
reg     eiread, softcmden, brdctlen, seectlen, eepromen, romrden, sspiocps, idrdy;
reg     [7:0] iddat;
reg     [1:0] soft;

// internal signals
wire    ckiddat;            // clock iddat
wire    ckestat;            // clock estat
wire    setckid, setckes, setidrdy;
reg     ckes, ckid;
wire    rstsoften_;         // reset softcmden
wire    setread;

//-----------------------------------------------------------------------
// enable seectl and brdctl registers

assign spseeen = seectlen & ~softcmden;
assign spbrden = brdctlen & ~softcmden;

//-----------------------------------------------------------------------
// IDDAT register

// gated clock
assign ckiddat = ckid & ~parerrdet;

// register write
```

iddatestat.v

```
always @(posedge clk40b3 or negedge rstis_) begin
  if (~rstis_)
        iddat <= #FFDLY DEFAULTID;
  else
        iddat <= #FFDLY clkiddat ? shftin : iddat;
end     // register

//-----------------------------------------------------------------
// SPIOCAP register

assign rstsoften_ = rstis_ & ssplocps;

// gated clock
assign clkestat = ckes & ~parerrdet;

// register write for bits 7:6, 4:0
always @(posedge clk40b3 or negedge rstis_) begin
  if (~rstis_)
        {soft,brdctlen,seectlen,eepromen,romrden,ssplocps}
              <= #FFDLY 7'b0000000;
  else
        {soft,brdctlen,seectlen,eepromen,romrden,ssplocps}
              <= #FFDLY clkestat ? ~{shftin[7:6], shftin[4:0]}
                    : {soft,brdctlen,seectlen,eepromen,romrden,ssplocps};
end     // bits 7:6, 4:0

// register write for bit 5
always @(posedge wrsplocap_ or negedge rstsoften_) begin
  if (~rstsoften_)
        softcmden <= #FFDLY 1'b0;
  else
        softcmden <= #FFDLY cddat5;
end     // bit 5

//-----------------------------------------------------------------
// register state machine outputs

always @(posedge clk40b3 or negedge rstis_)
  if (~rstis_) begin
        idrdy <= #FFDLY 1'b0;
        ckid <= #FFDLY 1'b0;
        ckes <= #FFDLY 1'b0;
        eiread <= #FFDLY 1'b0;
  end
  else begin
        idrdy <= #FFDLY setidrdy;
        ckid <= #FFDLY setckid & (shftin!=8'hff);
        ckes <= #FFDLY setckes;
        eiread <= #FFDLY setread;
  end

assign ididrdy = idrdy & swrstrdy;

//-----------------------------------------------------------------
// instantiate state-machine

elsm elsm ( .clk40b3(clk40b3), .eito(eito), .rstis_(rstis_), .sivalid(sivalid),
        .setckes(setckes), .setckid(setckid), .setread(setread), .eneito(eneito),
        .setidrdy(setidrdy) );

//-----------------------------------------------------------------

endmodule
```

romctl.v

```
/*****************************************************************************

Function:

ROM control logic inside PHOST.  Interfaces to SPIO block.




*****************************************************************************/

module romctl ( beinadrdec0, beinadrdec1, exadrdecvdd_, mwrc, pclk, hpors_, reg8do,
        adrinreg, romdeci_, spioackdet, spromclk_, spromrdy, romadr, disextmr_,
        romdec_, romdo, romrdy, romwr );

parameter FFDLY = 1;

input   beinadrdec0, beinadrdec1, exadrdecvdd_, mwrc, pclk, hpors_, spromrdy,
        romdeci_, spioackdet, spromclk_;
input   [7:0] reg8do;
input   [15:2] adrinreg;

output  [15:0] romadr;      // rom address to SPIO
output  disextmr_;          // got ACK, disable exceedtimer
output  romdec_;            // ROM access decoded to SPIO
output  [31:0] romdo;       // ROM read data to path_out
output  romrdy;             // ROM ready to target cycle control
output  romwr;              // EEPROM write to SPIO

//-------------------------------------------------------------
// internal signals

reg     [31:0] romdo;
reg     [1:0] bytecnt;
reg     romdec_, disextmr_, romrdy, waitdec, done, done1, done2;
reg     disextmrs1, disextmrs2;
wire    rstdisextmr_, rstdone_;
wire    bciken0_, bciken1_, bciken2_, bciken3_;

//-------------------------------------------------------------
// rom addressing

always @(posedge pclk or negedge hpors_)
  if (~hpors_)
        romdec_ <= #FFDLY 1'b1;
  else
        romdec_ <= #FFDLY  romdeci_ | exadrdecvdd_;

buf0104 ad_drvr_20 ( .Q(romwr),  .A(mwrc) );

buf0104 ad_drvr_02 ( .Q(romadr[2]),  .A(adrinreg[2]) );
buf0104 ad_drvr_03 ( .Q(romadr[3]),  .A(adrinreg[3]) );
buf0104 ad_drvr_04 ( .Q(romadr[4]),  .A(adrinreg[4]) );
buf0104 ad_drvr_05 ( .Q(romadr[5]),  .A(adrinreg[5]) );
buf0104 ad_drvr_06 ( .Q(romadr[6]),  .A(adrinreg[6]) );
buf0104 ad_drvr_07 ( .Q(romadr[7]),  .A(adrinreg[7]) );
buf0104 ad_drvr_08 ( .Q(romadr[8]),  .A(adrinreg[8]) );
buf0104 ad_drvr_09 ( .Q(romadr[9]),  .A(adrinreg[9]) );
buf0104 ad_drvr_10 ( .Q(romadr[10]), .A(adrinreg[10]) );
buf0104 ad_drvr_11 ( .Q(romadr[11]), .A(adrinreg[11]) );
buf0104 ad_drvr_12 ( .Q(romadr[12]), .A(adrinreg[12]) );
buf0104 ad_drvr_13 ( .Q(romadr[13]), .A(adrinreg[13]) );
buf0104 ad_drvr_14 ( .Q(romadr[14]), .A(adrinreg[14]) );
buf0104 ad_drvr_15 ( .Q(romadr[15]), .A(adrinreg[15]) );
```

```
assign romadr[1:0] = romwr ? {beinadrdec1,beinadrdec0} : {2'b00};

//-----------------------------------------------------------------
// disable exceed timer

assign rstdisextmr_ = hpors_ & disextmr_;

always @(posedge sploackdet or negedge rstdisextmr_)
  if (~rstdisextmr_)
        disextmrs1 <= #FFDLY 1'b0;
  else
        disextmrs1 <= #FFDLY 1'b1;

always @(posedge pclk or negedge hpors_)
  if (~hpors_) begin
        disextmrs2 <= 1'b0;
        disextmr_ <= 1'b1;
  end
  else begin
        disextmrs2 <= #FFDLY disextmrs1;
        disextmr_ <= #FFDLY ~disextmrs2;
  end

//-----------------------------------------------------------------
// byte counter

always @(posedge spromclk_ or negedge hpors_)
  if (~hpors_)
        bytecnt <= #FFDLY 2'b00;
  else
        bytecnt <= #FFDLY bytecnt + 2'b01;

//-----------------------------------------------------------------
// generate byte clocks

assign bclken0_ = (bytecnt!=2'b00);
assign bclken1_ = (bytecnt!=2'b01);
assign bclken2_ = (bytecnt!=2'b10);
assign bclken3_ = (bytecnt!=2'b11);

nor02 gclk_0 ( .NQ(bclk0_), .A(spromclk_), .B(bclken0_) );
nor02 gclk_1 ( .NQ(bclk1_), .A(spromclk_), .B(bclken1_) );
nor02 gclk_2 ( .NQ(bclk2_), .A(spromclk_), .B(bclken2_) );
nor02 gclk_3 ( .NQ(bclk3_), .A(spromclk_), .B(bclken3_) );

nbuf03 gclk_drv_0 ( .NQ(bclk0), .A(bclk0_) );
nbuf03 gclk_drv_1 ( .NQ(bclk1), .A(bclk1_) );
nbuf03 gclk_drv_2 ( .NQ(bclk2), .A(bclk2_) );
nbuf03 gclk_drv_3 ( .NQ(bclk3), .A(bclk3_) );

//-----------------------------------------------------------------
// rom data registers

always @(posedge bclk0 or negedge hpors_)
  if (~hpors_)
        romdo[7:0] <= #FFDLY 8'h00;
  else
        romdo[7:0] <= #FFDLY reg8do;

always @(posedge bclk1 or negedge hpors_)
  if (~hpors_)
        romdo[15:8] <= #FFDLY 8'h00;
  else
        romdo[15:8] <= #FFDLY reg8do;
```

romctl.v

```
always @(posedge bclk2 or negedge hpors_)
  if (~hpors_)
        romdo[23:16] <= #FFDLY 8'h00;
  else
        romdo[23:16] <= #FFDLY reg8do;

always @(posedge bclk3 or negedge hpors_)
  if (~hpors_)
        romdo[31:24] <= #FFDLY 8'h00;
  else
        romdo[31:24] <= #FFDLY reg8do;

//-----------------------------------------------------------------
// rom ready control

assign rstdone_ = hpors_ & ~done2;

// 4 bytes transferred - aysnc
always @(posedge epromrdy or negedge rstdone_)
  if (~rstdone_)
        done <= #FFDLY 1'b0;
  else
        done <= #FFDLY 1'b1;

always @(posedge pclk or negedge hpors_)
  if (~hpors_) begin
        done1 <= #FFDLY 1'b0;
        done2 <= #FFDLY 1'b0;
        waitdec <= #FFDLY 1'b0;
        romrdy <= #FFDLY 1'b1;
  end else begin
        done1 <= #FFDLY done;
        done2 <= #FFDLY done1;
        waitdec <= #FFDLY waitdec ? ~romdec_ : ~romrdy & done2;
        romrdy <= #FFDLY romdecl_ | waitdec | ~romrdy & done2;
  end

endmodule
```

rtl_lib.v

```
module nbuf02 ( NQ, A);
input     A;
output    NQ;
not (NQ, A);
endmodule

//-----------------------------------------------------------

module and05 ( Q, A, B, C, D, E);
input     A, B, C, D, E;
output    Q;
and (Q, A, B, C, D, E);
endmodule

//-----------------------------------------------------------

module nand03 ( NQ, A, B, C);
input     A, B, C;
output    NQ;
nand (NQ, A, B, C);
endmodule

//-----------------------------------------------------------

module or02 (Q, A, B);
input     A, B;
output    Q;
or (Q, A, B);
endmodule

//-----------------------------------------------------------

module and03 (Q, A, B, C);
input     A, B, C;
output    Q;
and (Q, A, B, C);
endmodule

//-----------------------------------------------------------

module nor02 (NQ, A, B);
input     A, B;
output    NQ;
nor (NQ, A, B);
endmodule

//-----------------------------------------------------------

module buf0102 ( Q, A );
input     A;
output    Q;
buf (Q, A);
endmodule

//-----------------------------------------------------------

module inv01 ( NQ, A);
input     A;
output    NQ;
not (NQ, A);
endmodule

//-----------------------------------------------------------
```

```
module bts9 (Q, A, NE);
input     A, NE;
output    Q;
bufif0 (Q, A, NE);
endmodule

//-----------------------------------------------------------

module   bushold (DUMMY, Z);
input     DUMMY;
inout     Z;
not (weak0, weak1) (Z, DUMMY);
not (DUMMY, Z);
endmodule

//-----------------------------------------------------------

module bts1 (Q, A, NE);
input     A, NE;
output    Q;
bufif0 (Q, A, NE);
endmodule

//-----------------------------------------------------------

module bts3 (Q, A, NE);
input     A, NE;
output    Q;
bufif0 (Q, A, NE);
endmodule

//-----------------------------------------------------------

module and02 (Q, A, B);
input     A, B;
output    Q;
and (Q, A, B);
endmodule

//-----------------------------------------------------------

module nrslt (Q,NQ,NS,NR);
 input  NS,NR;
 output Q,NQ;

U_NRSLT inst1 (Q, NS, NR);
not inst2 (NQ,Q);

 specify
     (NR +=> NQ) = (1,1);
     (NS +=> Q) = (1,1);
 endspecify

endmodule

//-----------------------------------------------------------

primitive U_NRSLT ( Q, NS, NR);
output Q;     reg Q;
input NS, NR;

table
 // NS   NR  :  Qt  :  Qt+1
```

```
//-------------------------------------------------------------------
    1  1 : ? : -; // Latch state.
    1  0 : ? :  0; // Reset state.
    0  1 : ? :  1; // set state.
    0  0 : ? :  1; // Set overrides reset.

    0  x : ? :  1;

    1  x : 0 :  0; // Indeterminate set and reset.
    x  1 : 1 :  1;

endtable
endprimitive

//-------------------------------------------------------------------

module mux41l (NQ,D0,D1,D2,D3,S0,S1);

input  D0,D1,D2,D3,S0,S1;
output NQ;

U_MUX41L inst (NQ,D0,D1,D2,D3,S0,S1);

endmodule

//-------------------------------------------------------------------

primitive U_MUX41L (NQ, D0, D1, D2, D3, S0, S1);

  input D0, D1, D2, D3, S0, S1;
  output NQ;

// FUNCTION : FOUR TO ONE MULTIPLEXER WITH 2 SELECT CONTROLS
//          AND INVERTING OUTPUT

  table

// D0 D1 D2 D3  S0 S1 : NQ

   0  ?  ?  ?   0  0 : 1;
   1  ?  ?  ?   0  0 : 0;

   ?  0  ?  ?   1  0 : 1;
   ?  1  ?  ?   1  0 : 0;

   ?  ?  0  ?   0  1 : 1;
   ?  ?  1  ?   0  1 : 0;

   ?  ?  ?  0   1  1 : 1;
   ?  ?  ?  1   1  1 : 0;


   0  0  0  0   ?  ? : 1;   //pessimism

   1  1  1  1   ?  ? : 0;

   0  0  ?  ?   ?  0 : 1;   //pessimism
   1  1  ?  ?   ?  0 : 0;   //pessimism

   ?  ?  0  0   ?  1 : 1;   //pessimism
   ?  ?  1  1   ?  1 : 0;   //pessimism

   0  ?  0  ?   0  ? : 1;
   1  ?  1  ?   0  ? : 0;

   ?  0  ?  0   1  ? : 1;
```

```
  ? 1 ? 1 1 ? :0;

  endtable
endprimitive
// @(#) buf0103.v REV 1.1 6/7/93

//-----------------------------------------------------------

module buf0103 (Q,A);

input  A;
output Q;

buf b0 (Q,A);

endmodule

//-----------------------------------------------------------

module ndc0r (Q, NQ, D, C, NR);

  input  D, C, NR;
  inout  NQ;
  output Q;

  reg NOTIFY_REG ;

      U_NDC0R_Q
        inst1 (Q, D, C, NR, NOTIFY_REG);
      not
                inst2 (NQ, Q);

  specify
    (posedge C => (NQ -: D))   = (1.4,1.3);
    (negedge NR => (NQ +: 1'b1)) = (0.4,0);
                (NQ => Q) = (0.4,0.4);
  endspecify

endmodule

//-----------------------------------------------------------

primitive U_NDC0R_Q (Q, D, CP, RB, NOTIFIER_REG);

  output Q;
  input  NOTIFIER_REG,
      D, CP, RB;
  reg  Q;

// FUNCTION : POSITIVE EDGE TRIGGERED D FLIP-FLOP WITH ACTIVE LOW
//         ASYNCHRONOUS CLEAR ( Q OUTPUT UDP ).


  table

  // D  CP    RB   NOTIFIER_REG :  Qt :  Qt+1

    1  (01)  1     ?       : ? : 1; // clocked data
    0  (01)  1     ?       : ? : 0;

    0  (01)  x     ?       : ? : 0; // pessimism
    0  ?    x     ?       : 0 : 0; // pessimism

    1  0     x     ?       : 0 : 0; // pessimism
    1  x    (?x)    ?       : 0 : 0; // pessimism
```

rtl_lib.v

```
1  1  (?x)   ?     : 0 : 0; // pessimism

x  0   x     ?     : 0 : 0; // pessimism
x  x  (?x)   ?     : 0 : 0; // pessimism
x  1  (?x)   ?     : 0 : 0; // pessimism

1 (x1)  1    ?     : 1 : 1; // reducing pessimism
0 (x1)  1    ?     : 0 : 0;
1 (0x)  1    ?     : 1 : 1;
0 (0x)  1    ?     : 0 : 0;


? ?   0      ?     : ? : 0; // asynchronous clear

? (?0)  ?    ?     : ? : -; // ignore falling clock
? (1x)  ?    ?     : ? : -; // ignore falling clock
*  ?    ?    ?     : ? : -; // ignore the edges on data

? ?  (?1)    ?     : ? : -; // ignore the edges on clear

? ?   ?      *     : ? : x;

endtable
endprimitive
```

sbwcctl.v

Function:

Single byte write command controller. Handles the following harware gernated commands: LED and STPW. These commands will not be issued unless ssplocps is "1".

LED Issues LED command whenever led level changes. Earlier commands can be lost if LED is changing too fast and the port is busy. The last value will always be issued, though.

STPW SCSI term power control state machine. Command is issued whenever SXFRCTL1 (02h) is written. The associated term power level and term power enable values are latched for later shifting.

```
*****************************************************************************/

module sbwcctl ( ackdet, bytesent, clk40b1, ledack, noack,
        rstsplos_, sled_, ssplocps, stpwack, stpwenb, stpwlvlb,
        ledreq, ledstate, sbwcbsy, sbwcread, stpe, stpl, stpwreq );

parameter FFDLY = 1;

input   ackdet, bytesent, clk40b1, ledack, noack, rstsplos_,
        sled_, ssplocps, stpwack, stpwenb, stpwlvlb;

output  ledreq;         // request port for LED command
output  ledstate;       // ledstate to be shifted out
output  sbwcbsy;        // LED, STPW or SPIORST command in progress
output  sbwcread;       // read SPIO port for ACK
output  stpe;           // STPWEN value to be shifted out
output  stpl;           // STPWLEVEL value to be shifted out
output  stpwreq;        // request port of STPW command

//-------------------------------------------------------------
// registered outputs

reg     sbwcread, ledstate, sbwcbsy, stpe, stpl, ledreq, stpwreq;

//-------------------------------------------------------------
// internal signals

// common
wire    rstsbwc_;               // reset sbwcctl
wire    setbsy;                 // generate sbwcbsy
wire    sbwcack;                // combination of ledack or stpwack
wire    setread;

// led
reg     sleds;                  // synced sled_
wire    ledchg;                 // led level change

// stpw
reg     stpwens;                // stpwenb synced
wire    stpwenchg;              // STPWEN bit is changed

//-------------------------------------------------------------
// reset logic
```

sbwcctl.v

```
assign rstsbwc_ = rstspios_ & sspiocps;
assign sbwcack = ledack | stpwack;

//-----------------------------------------------------------------
// led request logic

assign ledchg = sleds ^ ledstate;

always @(posedge clk40b1 or negedge rstsbwc_) begin
   if (~rstsbwc_) begin
          sleds <= #FFDLY 1'b0;
          ledstate <= #FFDLY 1'b0;
          ledreq <= #FFDLY 1'b0;
   end
   else begin
          sleds <= #FFDLY sled_;
          ledstate <= #FFDLY sleds;
          ledreq <= #FFDLY ledchg | ledreq & ~ledack;
   end
end       // always

//-----------------------------------------------------------------
// stpw request logic

assign    stpwenchg = stpwens ^ stpe;

always @(posedge clk40b1 or negedge rstsbwc_)
   if (~rstsbwc_) begin
          stpwens <= #FFDLY 1'b0;
          stpe    <= #FFDLY 1'b0;
          stpwreq <= #FFDLY 1'b0;
          stpl    <= #FFDLY 1'b0;
   end
   else begin
          stpwens <= #FFDLY stpwenb;
          stpe    <= #FFDLY stpwens;
          stpwreq <= #FFDLY stpwenchg | stpwreq & ~stpwack;
          stpl    <= #FFDLY stpwenchg ? stpwlvlb : stpl;
   end

//-----------------------------------------------------------------
// registered outputs

always @(posedge clk40b1 or negedge rstsbwc_)
   if (~rstsbwc_) begin
          sbwcbsy <= #FFDLY 1'b0;
          sbwcread <= #FFDLY 1'b0;
   end
   else begin
          sbwcbsy <= #FFDLY setbsy;
          sbwcread <= #FFDLY setread;
   end

//-----------------------------------------------------------------
// instantiate state-machine

sbwcsm sbwcsm ( .ackdet(ackdet), .bytesent(bytesent), .clk40b1(clk40b1),
          .noack(noack), .rstsbwc_(rstsbwc_), .sbwcack(sbwcack),
          .setread(setread), .setbsy(setbsy) );

//-----------------------------------------------------------------

endmodule
```

sbwcsm.v

```
/*****************************************************************

Function:

Single Byte Write Command control state machine.  Monitors the progress of a
single byte write command.

History:

01/15/94 JYL        First rev.

*****************************************************************/

module sbwcsm ( ackdet, bytesent, clk40b1, noack, rstsbwc_, sbwcack, setread,
        setbsy );

parameter FFDLY = 1;

parameter [1:0]         // synopsys enum state_info
        ID = 2'b00,     // IDLE
        CC = 2'b01,     // Command code
        WA = 2'b10,     // wait for ACK begin
        EA = 2'b11;     // wait for ACK end

input   ackdet, bytesent, clk40b1, noack, rstsbwc_, sbwcack;

output  setread;        // read spio for ACK
output  setbsy;         // LED command in progress

//-----------------------------------------------------------------
// internal signals

                // synopsys state_vector state
reg     [1:0]   // synopsys enum state_info
        state;  // current state
reg     [1:0]   // synopsys enum state_info
        next;   // next state

reg     setread, setbsy;

//-----------------------------------------------------------------
// combinational logic

always @( state or sbwcack or noack or bytesent or ackdet ) begin

  setbsy = 1'b0;
  setread = 1'b0;

  case (state)              // synopsys parallel_case

        ID: begin
          next = sbwcack ? CC : ID;
          setbsy = sbwcack;
        end

        CC: begin
          next = bytesent ? WA : CC;
          setbsy = 1'b1;
          setread = bytesent;
        end

        WA: begin
          if (noack) begin
                next = ID;
          end
```

```
            else begin
                    next = ackdet ? EA : WA;
                    setbsy = 1'b1;
              end
            end

            EA: begin
              next = ackdet ? EA : ID;
              setbsy = ackdet;
            end

            default: next = 2'bxx;

      endcase

  end       // always

  //-----------------------------------------------------------------
  // state vector

  always @(posedge clk40b1 or negedge rstsbwc_) begin
    if (~rstsbwc_)
            state <= #FFDLY ID;
    else
            state <= #FFDLY next;
  end       // always

  //-----------------------------------------------------------------

  endmodule
```

Function:

SEEPROM controller. Issues SEEPROM command if SEECTL is written or SEECTL is read while SEERDY is not set but SEERDY counter expires. The 800ns SEERDY delay has two components: SEERDY timer and command shift time. SEERDY timer is 16 clocks. Command shift time = start bit (1) + command code (8) + parity bit (1) + stop bit (1) + turn around (2) + min ack (3) = 16 clocks.
32*25ns = 800ns.

```
*****************************************************************************/

module seecntrl ( ackdet, bytesent, clk40b2, noack, rdseectl_, rstspios_, seeack,
        seectlen, shftin0, sivalid, wrseectl_, cddat, rstseectl_, seectl,
        seectlrdy, seedi, seems, seerdy, seeread, seereq, seebsy, seesend );

parameter FFDLY = 1;

input   ackdet, bytesent, clk40b2, noack, rdseectl_, rstspios_,
        seeack, seectlen, shftin0, sivalid, wrseectl_;
input   [7:0] cddat;

output  rstseectl_;         // reset seectl logic
output  seebsy;
output  [2:0] seectl;       // seectl (1Eh) register
output  seedi;              // SEEPROM input data
output  seems;              ///SEEPROM mode select
output  seerdy;             // SEECTL read ready
output  seeread;
output  seereq;
output  seesend;
output  seectlrdy;          // SEEPROM command ready, used to stretch PHOST or seq

//-------------------------------------------------------------
// registered outputs

reg     [2:0] seectl;
reg     seeread, seems, seerdy, seesend, seedi, seereq, seebsy, seectlrdy;

//-------------------------------------------------------------
// internal signals

reg     [3:0] rdytimer;     // SEERDY timer
reg     seewact;            // SEECTL write active
reg     seeract;            // SEECTL read active
reg     seeacts1;           // sync
reg     seeacts2;           // double sync
wire    cnt0;               // seerdy timer expired
wire    rstseectl_;         // reset SEECTL logic
wire    writereq;           // CIO write to SEECTL
wire    readreq;            // SPIO read command is needed
wire    cioreadact;         // CIO read active
wire    noreadreq;          // no SPIO request needed, cancel seectlrdy_

wire    setread, setbsy, setsend, strttmr, rdatval, rstreqgen_;

//-------------------------------------------------------------
```

```
// reset logic

assign    rstseectl_ = rstsplos_ & seectlen;
assign    rstreqgen_ = rstseectl_ & ~seeacts2;

//-----------------------------------------------------------------
// SEEPROM ready signal

always @(posedge clk40b2 or negedge rstseectl_)
  if (~rstseectl_)
        seectlrdy <= #FFDLY 1'b0;
  else
        seectlrdy <= #FFDLY noreadreq | rdatval | strttmr;

//-----------------------------------------------------------------
// SEECTL register

always @(posedge wrseectl_ or negedge rstseectl_)
  if (~rstseectl_) begin
        seems <= #FFDLY 1'b0;
        seectl <= #FFDLY 3'b000;
  end
  else begin
        seems <= #FFDLY cddat[5];
        seectl <= #FFDLY cddat[3:1];
  end

always @(posedge clk40b2 or negedge rstseectl_)
  if (~rstseectl_)
        seedi <= #FFDLY 1'b0;
  else
        seedi <= #FFDLY rdatval ? shftin0 : seedi;

//-----------------------------------------------------------------
// request generation

assign writereq = seeacts1 & seeacts2 & rdseectl_;
assign cloreadact = seeacts1 & seeacts2 & ~rdseectl_;
assign readreq = cloreadact & cnt0;
assign noreadreq = cloreadact & ~cnt0;

always @(negedge wrseectl_ or negedge rstreqgen_)
  if (~rstreqgen_)
        seewact <= #FFDLY 1'b0;
  else
        seewact <= #FFDLY 1'b1;

always @(negedge rdseectl_ or negedge rstreqgen_)
  if (~rstreqgen_)
        seeract <= #FFDLY 1'b0;
  else
        seeract <= #FFDLY 1'b1;

always @(posedge clk40b2 or negedge rstseectl_)
  if (~rstseectl_) begin
        seeacts1 <= #FFDLY 1'b0;
        seeacts2 <= #FFDLY 1'b0;
        seereq  <= #FFDLY 1'b0;
  end
  else begin
        seeacts1 <= #FFDLY seeract | seewact;
        seeacts2 <= #FFDLY seeacts1;
        seereq  <= #FFDLY writereq | readreq | (seereq & ~seeack);
  end
```

```
//------------------------ ----------------------------------------
// seerdy generation

always @(posedge clk40b2 or negedge rstseectl_)
  if (~rstseectl_)
        seerdy <= #FFDLY 1'b0;
  else
        seerdy <= #FFDLY writereq ? 1'b0 : seerdy | rdatval;

//-----------------------------------------------------------------
// seerdy timer

assign cnt0 = (rdytimer==4'h0);

always @(posedge clk40b2 or negedge rstseectl_) begin
  if (~rstseectl_)
        rdytimer <= #FFDLY 4'h0;
  else begin
        if (strttmr)
          rdytimer <= #FFDLY 4'h1;
        else
          rdytimer <= #FFDLY cnt0 ? 4'h0 : rdytimer + 4'h1;
  end
end     // always

//-----------------------------------------------------------------
// registered output

always @(posedge clk40b2 or negedge rstseectl_)
  if (~rstseectl_) begin
        seebsy <= #FFDLY 1'b0;
        seesend <= #FFDLY 1'b0;
        seeread <= #FFDLY 1'b0;
  end
  else begin
        seebsy <= #FFDLY setbsy;
        seesend <= #FFDLY setsend;
        seeread <= #FFDLY setread;
  end

//-----------------------------------------------------------------
// instantiate state-machine

seectlsm seectlsm ( .ackdet(ackdet), .bytesent(bytesent), .clk40b2(clk40b2),
        .rdseectl_(rdseectl_), .noack(noack), .seeack(seeack),
        .rstseectl_(rstseectl_), .sivalid(sivalid), .rdatval(rdatval),
        .setread(setread), .setbsy(setbsy),
        .setsend(setsend), .strttmr(strttmr) );

//-----------------------------------------------------------------

endmodule
```

seectlsm.v

```
/*****************************************************************************

Function:

SEEPROM control state-machine.



*****************************************************************************/

module seectlsm ( ackdet, bytesent, clk40b2, rdseectl_, noack, seeack, rstseectl_,
        sivalid, rdatval, setread, setbsy, setsend, strttmr );

parameter FFDLY = 1;

parameter [2:0]             // synopsys enum state_info
        TST0 = 3'b000,      // step all states for testability
        IDLE = 3'b001,      // idle
        CMDC = 3'b010,      // shifting command code
        RWAK = 3'b011,      // read command, waiting for ACK
        RDAT = 3'b100,      // shifting in data
        WDAT = 3'b101,      // shifting out data
        WWAK = 3'b110,      // write command, waiting for ACK
        EACK = 3'b111;      // write command, waiting for end of ACK

input   ackdet, bytesent, clk40b2, rdseectl_, noack, rstseectl_, seeack, sivalid;

output  rdatval;            // read data is valid
output  setread;            // read SPIO port
output  setbsy;             // SEECTL command in progress
output  setsend;            // shift write data
output  strttmr;            // start timer

//-----------------------------------------------------------------------
// internal signals

                    // synopsys state_vector state
reg     [2:0]       // synopsys enum state_info
        state;      // current state
reg     [2:0]       // synopsys enum state_info
        next;       // next state

reg     rdatval, setsend, setbsy, setread, strttmr;

//-----------------------------------------------------------------------
// combinational logic

always @( state or ackdet or noack or sivalid or bytesent or rdseectl_
        or seeack )
begin

   strttmr = 1'b0;
   setread = 1'b0;
   setsend = 1'b0;
   setbsy = 1'b0;
   rdatval = 1'b0;

   case (state)             // synopsys parallel_case

        TST0: begin
          next = IDLE;
        end
```

```
        IDLE: begin
          next = seeack ? CMDC : IDLE;
          setbsy = seeack;
        end

        CMDC: begin
          setbsy = 1'b1;
          if (bytesent) begin
                  next = rdseectl_ ? WDAT : RWAK;
                  setsend = rdseectl_;
                  setread = ~rdseectl_;
          end
          else begin
                  next = CMDC;
          end
        end

        RWAK: begin
          if (noack) begin
                  next = IDLE;
          end
          else begin
                  setread = 1'b1;
                  next = ackdet ? RDAT : RWAK;
                  setbsy = 1'b1;
          end
        end

        RDAT: begin
          next = sivalid ? IDLE : RDAT;
          setread = ~sivalid;
          setbsy = 1'b1;
          rdatval = sivalid;
        end

        WDAT: begin
          next = bytesent ? WWAK : WDAT;
          setbsy = 1'b1;
          setread = bytesent;
        end

        WWAK: begin
          if (noack) begin
                  next = IDLE;
          end
          else begin
                  next = ackdet ? EACK : WWAK;
                  setbsy = 1'b1;
          end
        end

        EACK: begin
          next = ackdet ? EACK : IDLE;
          setbsy = ackdet;
          strttmr = ~ackdet;
        end

        default: next = 3'bxxx;

    endcase

end     // always

//-----------------------------------------------------------------
// state vector
```

seectlsm.v

```
always @(posedge clk40b2 or negedge rstseectl_)
  if (~rstseectl_)
        state <= #FFDLY IDLE;
  else
        state <= #FFDLY next;

//-----------------------------------------------------------------

endmodule
```

```
/*****************************************************************

Function:

Soft command state machine.

*****************************************************************/
module softcmdsm ( ackdet, bytesent, clk40b5, cnt0, cnt1, lastb, ledtstact, noack,
        rstsoftcmd_, sivalid, softack, softsend, loadcnt, softabrtok,
        setread, setbsy, setclrlast, setclrsend, setledtst, wrfifo );

parameter FFDLY = 1;

parameter [2:0]         // synopsys enum state_info
        IDLE = 3'b000,  // waiting for softack
        SEND = 3'b001,  // sending a byte
        WNXT = 3'b010,  // waiting to send the next byte
        LSTB = 3'b011,  // sendint the last byte
        WACK = 3'b100,  // waiting for ACK
        EACK = 3'b101,  // waiting for the end of ACK
        RDAT = 3'b110,  // reading SPIO
        LTST = 3'b111;  // led test mode

input   ackdet, bytesent, clk40b5, cnt0, cnt1, lastb, ledtstact, noack,
        rstsoftcmd_, sivalid, softack, softsend;

output  loadcnt;        // load read counter
output  setclrlast;     // clear last bit, clear fifo and set pointers
output  setbsy;         // set soft busy
output  setclrsend;     // clear send bit
output  setledtst;      // go into led test mode
output  softabrtok;     // ok to abort (SW reset) this multi-byte soft command
output  setread;        // read SPIO port
output  wrfifo;         // write soft fifo

//-----------------------------------------------------------------
// register declaration for outputs

reg     softabrtok, setread, setclrsend, wrfifo, loadcnt, setclrlast, setbsy,
        setledtst;

//-----------------------------------------------------------------
// internal signals

                // synopsys state_vector state
reg     [2:0]   // synopsys enum state_info
        state;  // current state
reg     [2:0]   // synopsys enum state_info
        next;   // next state

//-----------------------------------------------------------------
// combinatinal logic

always @( state or ackdet or bytesent or lastb or softack or noack or cnt0
        or ledtstact or cnt1 or softsend or sivalid )
begin

   setbsy = 1'b1;
   setclrlast = 1'b0;
   setclrsend = 1'b0;
   wrfifo = 1'b0;
```

```
loadcnt = 1'b0;
setread = 1'b0;
softabrtok = 1'b0;
setledtst = 1'b0;

case (state)                    // synopsys parallel_case

        IDLE: begin
          if (softack) begin
                  next = lastb ? LSTB : SEND;
                  loadcnt = lastb;
          end
          else begin
                  next = IDLE;
                  setbsy = 1'b0;
          end
        end

        SEND: begin
          next = bytesent ? WNXT : SEND;
          setclrsend = bytesent;
          softabrtok = 1'b1;
        end

        WNXT: begin
          softabrtok = 1'b1;
          if (softsend) begin
                  next = lastb ? LSTB : SEND;
                  loadcnt = lastb;
          end
          else begin
                  next = WNXT;
          end
        end

        LSTB: begin
          next = bytesent ? WACK : LSTB;
          setread = bytesent;
        end

        WACK: begin
          if (noack) begin
                  next = IDLE;
                  setbsy = 1'b0;
                  setclrlast = 1'b1;
          end
          else if (ackdet) begin
                  next = cnt0 ? EACK : RDAT;
                  setclrlast = 1'b1;
                  setread = ~cnt0;
          end
          else begin
                  next = WACK;
                  setread = ~cnt0;
          end
        end

        EACK: begin
          if (ackdet) begin
                  next = EACK;
                  setbsy = 1'b1;
          end
          else begin
                  next = ledtstact ? LTST : IDLE;
                  setbsy = ledtstact;
```

```
            end
          end

          RDAT: begin
            next = (sivalid & cnt1) ? IDLE : RDAT;
            setbsy = 1'b1;
            setread = ~(sivalid & cnt1);
            wrfifo = sivalid;
          end

          LTST: begin
            next = LTST;
            setbsy = 1'b1;
            setledtst = 1'b1;
          end

          default: next = 3'bxxx;

    endcase

end         // always

//-----------------------------------------------------------------
// state vector

always @(posedge clk40b5 or negedge rstsoftcmd_)
    if (~rstsoftcmd_)
          state <= #FFDLY IDLE;
    else
          state <= #FFDLY next;

//-----------------------------------------------------------------

endmodule           // softcmdsm
```




```
/*****************************************************************

Function:

Top level of soft command control logic.

*****************************************************************/

module softcntrl ( ackdet, bytesent, cddat, clk40b5, clk40b6, ensoftcdat, ltstsigl, noack,
        rdsplodat_, rstsplos_, shftin, sivalid, softack, softcmden,
        wrsplodat_, wrsplodat_, ledtst, ledtststg, softabrtok, softbsy,
        softread, softreq, softsend, splocdr10, splocdr74, splordat, splowdat );

input   ackdet, bytesent, clk40b5, clk40b6, ensoftcdat, noack, rdsplodat_,
        rstsplos_, sivalid, softack, softcmden, wrsplocdl_,
        wrsplodat_;
input   [7:0] cddat, ltstsigl, shftin;

output  ledtst, ledtststg, softabrtok, softbsy, softread, softreq, softsend;
output  [1:0] splocdr10;
output  [3:0] splocdr74;
output  [7:0] splordat, splowdat;

//-----------------------------------------------------------------
// bus declaration

wire    [1:0] readcnt;
wire    [7:0] ltstsigl;

//-----------------------------------------------------------------

softctl softctl ( .clk40b5(clk40b5), .noack(noack), .rstsplos_(rstsplos_),
        .sendb(sendb), .setbsy(setbsy), .setclrlast(setclrlast),
        .setclrsend(setclrsend), .setledtst(setledtst), .softack(softack),
        .splowdat(splowdat), .softcmden(softcmden), .ledtst(ledtst),
        .ledtstact(ledtstact), .clrlastb_(clrlastb_), .rstfifo_(rstfifo_),
        .rstsend_(rstsend_), .rstsoftcmd_(rstsoftcmd_), .softbsy(softbsy),
        .setread(setread), .softread(softread), .softreq(softreq),
        .softsend(softsend) );

softcmdsm softcmdsm ( .ackdet(ackdet), .bytesent(bytesent), .clk40b5(clk40b5),
        .cnt0(cnt0), .cnt1(cnt1), .lastb(lastb), .ledtstact(ledtstact),
        .noack(noack), .rstsoftcmd_(rstsoftcmd_), .sivalid(sivalid),
        .softack(softack), .softsend(softsend), .loadcnt(loadcnt),
        .softabrtok(softabrtok), .setread(setread), .setbsy(setbsy),
        .setclrlast(setclrlast), .setclrsend(setclrsend),
        .setledtst(setledtst), .wrfifo(wrfifo) );

spfifo spfifo ( .clk40b5(clk40b5), .clk40b6(clk40b6), .loadcnt(loadcnt),
        .ensoftcdat(ensoftcdat), .rdsplodat_(rdsplodat_), .readcnt(readcnt),
        .rstfifo_(rstfifo_), .rstsoftcmd_(rstsoftcmd_), .shftin(shftin),
        .wrfifo(wrfifo), .cnt0(cnt0), .cnt1(cnt1), .splocdr10(splocdr10),
        .splordat(splordat) );

softtimer softtimer ( .clk40b5(clk40b5), .clrlastb_(clrlastb_),
        .rstsoftcmd_(rstsoftcmd_), .timer(timer), .rsttimer_(rsttimer_) );

softregs softregs ( .cddat(cddat), .ledtststg(ledtststg), .ltstsigl(ltstsigl),
        .ensoftcdat(ensoftcdat), .rstsend_(rstsend_),
        .rstsoftcmd_(rstsoftcmd_), .rsttimer_(rsttimer_),
        .wrsplocdl_(wrsplocdl_), .wrsplodat_(wrsplodat_), .lastb(lastb),
        .readcnt(readcnt), .sendb(sendb), .splocdr74(splocdr74),
        .splowdat(splowdat), .timer(timer) );

endmodule
```



```
/*****************************************************************

Filename:       softctl.v

Function:

Soft Command Control logic.



*****************************************************************/

module softctl ( clk40b5, noack, rstspios_, sendb, setbsy, setledtst, setclrlast,
        setread, ledtst, ledtstact, setclrsend, spiowdat, softack, softcmden,
        clrlastb_, rstfifo_, rstsend_, rstsoftcmd_, softbsy, softread, softreq,
        softsend );

parameter FFDLY = 1;

input   clk40b5, noack, rstspios_, sendb, setbsy, setclrlast,
        setledtst, setclrsend, setread, softack, softcmden;
input   [7:0] spiowdat;

output  clrlastb_;      // last LB bit in softctl register
output  ledtst;         // led test command mode
output  ledtstact;      // led test command pending
output  rstfifo_;       // reset fifo registers
output  rstsend_;       // reset send and last_byte bits
output  rstsoftcmd_;    // reset soft command logic
output  softbsy;        // soft command in progress
output  softread;       // read SPIO port
output  softreq;        // request for SPIO port
output  softsend;       // send a byte

//-----------------------------------------------------------------
// register declaration for outputs

reg     ledtst, ledtstact, softreq, softbsy, softread, softsend, clrlastb_;

//-----------------------------------------------------------------
// internal signals

reg     clrsend_, softacts1, softacts2;
wire    rstledtst_;

//-----------------------------------------------------------------
// reset logic

assign rstsoftcmd_ = rstspios_ & softcmden;
assign rstfifo_ = rstsoftcmd_ & clrlastb_;
assign rstsend_ = rstsoftcmd_ & clrlastb_ & clrsend_;
assign rstledtst_ = rstsoftcmd_ & ~noack;

//-----------------------------------------------------------------
// request logic

always @(posedge clk40b5 or negedge rstsoftcmd_)
  if (~rstsoftcmd_) begin
        softacts1 <= #FFDLY 1'b0;
        softacts2 <= #FFDLY 1'b0;
        softreq <= #FFDLY 1'b0;
        softsend <= #FFDLY 1'b0;
        softread <= #FFDLY 1'b0;
```



```
    end
    else begin
            softacts1 <= #FFDLY sendb;
            softacts2 <= #FFDLY softacts1;
            softreq <= #FFDLY softacts1 & ~softacts2 & ~softbsy;
            softsend <= #FFDLY softacts1 & ~softacts2 & softbsy;
            softread <= #FFDLY setread;
    end

//-----------------------------------------------------------------
// registered outputs

always @(posedge clk40b5 or negedge rstsoftcmd_)
  if (~rstsoftcmd_) begin
            clrlastb_ <= #FFDLY 1'b1;
            clrsend_ <= #FFDLY 1'b1;
            softbsy <= #FFDLY 1'b0;
            ledtst <= #FFDLY 1'b0;
  end
  else begin
            clrlastb_ <= #FFDLY ~setclrlast;
            clrsend_ <= #FFDLY ~setclrsend;
            softbsy <= #FFDLY setbsy;
            ledtst <= #FFDLY setledtst;
  end

//-----------------------------------------------------------------
// test mode

always @(posedge clk40b5 or negedge rstledtst_)
  if (~rstledtst_)
            ledtstact <= #FFDLY 1'b0;
  else
            ledtstact <= #FFDLY ledtstact | softack & (splowdat==8'h82);

//-----------------------------------------------------------------

endmodule
```

Soft

```
/*****************************************************************************

Function:

SPIO Soft Command Control registers (1D, 1E when softcmden==1).



*****************************************************************************/

module softregs ( cddat, ltstsigi, ensoftcdat, rstsend_, rstsoftcmd_, rsttimer_,
        wrspioctl_, wrspiodat_, ledtstsig, lastb, readcnt, sendb, spioctlr74,
        spiowdat, timer );

parameter FFDLY = 1;

input   ensoftcdat, rstsend_, rstsoftcmd_, rsttimer_, wrspioctl_, wrspiodat_;
input   [7:0] ltstsigi, cddat;

output  lastb;                  // SPIOCTL bit 6
output  ledtstsig;              // led test mode output signal
output  [1:0] readcnt;          // SPIOCTL bits 1:0
output  sendb;                  // SPIOCTL bit 7
output  [3:0] spioctlr74;       // read data for SPIOCTL bits 7:4
output  [7:0] spiowdat;         // contents of spiodat register
output  timer;                  // SPIOCTL bit 4

//-----------------------------------------------------------------
// register declaration for outputs

reg     [7:0] spiowdat;
reg     ledtstsig, lock, sendb, lastb, timer;
reg     [1:0] readcnt;
reg     [2:0] spioctl_i;
wire    [2:0] wdat;

//-----------------------------------------------------------------
// led test mode signal select

assign wdat = spiowdat[2:0];

always @(ltstsigi or wdat )
  case (wdat)                   // synopsys parallel_case full_case
        3'b000: ledtstsig = ltstsigi[0];
        3'b001: ledtstsig = ltstsigi[1];
        3'b010: ledtstsig = ltstsigi[2];
        3'b011: ledtstsig = ltstsigi[3];
        3'b100: ledtstsig = ltstsigi[4];
        3'b101: ledtstsig = ltstsigi[5];
        3'b110: ledtstsig = ltstsigi[6];
        3'b111: ledtstsig = ltstsigi[7];
        default: ledtstsig = 1'bx;
  endcase

//-----------------------------------------------------------------
// SPIODAT (1Dh)

always @(posedge wrspiodat_ or negedge rstsoftcmd_)
  if (~rstsoftcmd_)
        spiowdat <= #FFDLY 8'h00;
  else
        spiowdat <= #FFDLY cddat;
```

```
//-----------------------------------------------------------------
// SPIOCTL (1Eh)

// bits 7:6
always @(posedge wrspioctl_ or negedge rstsend_)
  if (~rstsend_) begin
        sendb <= #FFDLY 1'b0;
        lastb <= #FFDLY 1'b0;
  end
  else begin
        sendb <= #FFDLY cddat[7];
        lastb <= #FFDLY cddat[6];
  end

// bit 5, 1:0
always @(posedge wrspioctl_ or negedge rstsoftcmd_)
  if (~rstsoftcmd_) begin
        lock <= #FFDLY 1'b0;
        readcnt <= #FFDLY 2'b00;
  end
  else begin
        lock <= #FFDLY cddat[5];
        readcnt <= #FFDLY cddat[1:0];
  end

// bit 4
always @(posedge wrspioctl_ or negedge rsttimer_)
  if (~rsttimer_)
        timer <= #FFDLY 1'b0;
  else
        timer <= #FFDLY cddat[4];

always @( ensoftcdat or sendb or lastb or timer )
  if (ensoftcdat)
        spioctl_i <= #FFDLY {sendb, lastb, timer};

assign spioctlr74 = {spioctl_i[2:1], lock, spioctl_i[0]};

//-----------------------------------------------------------------

endmodule          // softregs
```

```
/*****************************************************************************

Function:

Soft command timer.



*****************************************************************************/

module softtimer ( clk40b5, clrlastb_, rstsoftcmd_, timer, rsttimer_ );

parameter FFDLY = 1;

input     clk40b5, clrlastb_, rstsoftcmd_, timer;

output    rsttimer_;          // reset timer bit

//---------------------------------------------------------------------
// Internal signals

reg       [3:0] cnt;          // softcmd counter
reg       [3:0] new_cnt;      // new count
reg       rsttmrb_;           // reset timer bit

//---------------------------------------------------------------------
// counter logic

always @( cnt or clrlastb_ or timer )
  if (~clrlastb_ & timer)
        new_cnt = 4'h1;
  else if ((cnt==4'h0))
        new_cnt = 4'h0;
  else
        new_cnt = cnt + 4'h1;

always @(posedge clk40b5 or negedge rstsoftcmd_)
  if (~rstsoftcmd_)
        cnt <= #FFDLY 4'h0;
  else
        cnt <= #FFDLY new_cnt;

//---------------------------------------------------------------------
// reset timer bit

always @(posedge clk40b5 or negedge rstsoftcmd_)
  if (~rstsoftcmd_)
        rsttmrb_ <= #FFDLY 1'b1;
  else
        rsttmrb_ <= #FFDLY (cnt!=4'hf);

assign rsttimer_ = rstsoftcmd_ & rsttmrb_;

//---------------------------------------------------------------------

endmodule         // softtimer
```

```
/*****************************************************************************

Combinted sparb.v and spomux.v to alleviate time budgeting problems during
logic synthesis.

*****************************************************************************/

/*****************************************************************************

Function:

Serial port arbiter.  Prioritizes requests for SPIO usage and generates readspio
and sendbyte.

History:

00/00/00 JYL        First rev.

*****************************************************************************/

module sparbomux ( ackdet, brdbsy, brdread, brdreq, brdsend, cddat, chprstbsy,
        chprstreq, clk40b1, eiread, ledstate, ledreq, noack, romadr, rombsy,
        romread, romreq, romsend, romwr, rstspios_, sbwcbsy, sbwcread, seebsy,
        seeread, seereq, seesend, selroma1, selroma0, softbsy, softread, softreq,
        softsend, spiobsy_, spiowdat, stpe, stpi, stpwreq, brdack, chprstack,
        ledack, readspio, romack, seeack, shftout, sendbyte, softack, spioackdet,
        spnakabrt, stpwack );

parameter FFDLY = 1;

input   ackdet, brdbsy, brdread, brdreq, brdsend, chprstbsy, chprstreq, clk40b1,
        eiread, ledstate, ledreq, noack, rombsy, romread, romreq, romsend, romwr,
        rstspios_, sbwcbsy, sbwcread, seebsy, seeread, seereq, seesend, selroma0,
        selroma1, softbsy, softread, softreq, softsend, spiobsy_, stpe, stpi,
        stpwreq;
input   [7:0] cddat, spiowdat;
input   [15:0] romadr;

output  brdack;         // issue BRDCTL command
output  chprstack;      // issue SPIORST command
output  ledack;         // issue LED command
output  readspio;       // read from spio port
output  romack;         // issue ROM read/write command
output  seeack;         // issue SEECTL command
output  sendbyte;       // shift a byte out
output  [7:0] shftout;  // data to shift out
output  softack;        // issue SOFT command
output  spioackdet;     // SPIO ACK detected, disable exceed timer
output  spnakabrt;      // NOACK signal to abort logic
output  stpwack;        // issue STPW command

//-------------------------------------------------------------------
// register declaration for outputs

reg     spioackdet;

//-------------------------------------------------------------------
// internal signals

wire    [7:0] hwcdat;   // hardware command data
reg     [7:0] cmdcode;  // spio command code
reg     [7:0] extcmd;   // extended command data
wire    soft;           // soft command
wire    sendcmd;        // send spio command
wire    req;            // one or more request lines active
```

```
wire        portbusy;           // SPIO port is busy

// pending command
wire        rompend, brdpend, seepend, stpwpend, softpend, ledpend;

//-------------------------------------------------------------------
// NOACK to abort PCI cycle, ACKDET to disable exceeed timer

assign spnakabrt = noack & ( rombsy | seebsy | brdbsy );

always @(posedge clk40b1 or negedge rstspios_)
  if (~rstspios_)
        spioackdet <= #FFDLY 1'b0;
  else
        spioackdet <= #FFDLY ackdet & ( rombsy | seebsy | brdbsy);

//-------------------------------------------------------------------

assign portbusy = chprstbsy | sbwcbsy | rombsy | seebsy | brdbsy | softbsy;
assign req = ledreq | stpwreq | romreq | chprstreq | seereq | brdreq | softreq;
assign sendcmd = ~portbusy & req;
assign readspio = sbwcread | eiread | romread | softread | seeread | brdread;
assign sendbyte = romsend | seesend | brdsend | softsend | sendcmd;

//-------------------------------------------------------------------
// prioritizer

assign rompend  = ({chprstreq,romreq}==2'b01);
assign brdpend  = ({chprstreq,romreq,brdreq}==3'b001);
assign seepend  = ({chprstreq,romreq,brdreq,seereq}==4'b0001);
assign stpwpend = ({chprstreq,romreq,brdreq,seereq,stpwreq}==5'b00001);
assign softpend = ({chprstreq,romreq,brdreq,seereq,stpwreq,softreq}==6'b000001);
assign ledpend  = ({chprstreq,romreq,brdreq,seereq,stpwreq,softreq,ledreq}
                                  ==7'b0000001);

assign chprstack = ~portbusy & chprstreq;
assign romack  = ~portbusy & rompend;
assign brdack  = ~portbusy & brdpend;
assign seeack  = ~portbusy & seepend;
assign stpwack = ~portbusy & stpwpend;
assign softack = ~portbusy & softpend;
assign ledack  = ~portbusy & ledpend;

//-------------------------------------------------------------------
// output data mux

assign soft = softack | softbsy;
assign shftout = soft ? spiowdat : hwcdat;
assign hwcdat = portbusy ? extcmd : cmdcode;

//-------------------------------------------------------------------
// generate spio command code

always @( ledpend or stpwpend or rompend or seepend or brdpend
        or romwr or spiobsy_ or ledstate or stpe or stpl)

   casex ({ ledpend,stpwpend,rompend,seepend,brdpend,romwr,spiobsy_})
                                        // synopsys parallel_case

        7'b1xxxxxx: cmdcode = {1'b0,ledstate,5'b00001,1'b0};
        7'bx1xxxxx: cmdcode = {stpe,stpl,5'b00010,1'b0};
        7'bxx1xx0x: cmdcode = {2'b00,5'b00011,1'b1};
        7'bxx1xx1x: cmdcode = {2'b00,5'b00100,1'b0};
        7'bxxx1xx0: cmdcode = {2'b00,5'b00110,1'b1};
        7'bxxx1xx1: cmdcode = {2'b00,5'b00110,1'b0};
```

```
            7'bxxxx1x0: cmdcode = {2'b00,5'b00111,1'b1};
            7'bxxxx1x1: cmdcode = {2'b00,5'b00111,1'b0};
            default:    cmdcode = 8'hxx;

        endcase

    //-------------------------------------------------------------------
    // select extend command data

    always @( selroma1 or selroma0 or romwr or romadr or cddat )

        casex ({selroma1,selroma0})          // synopsys parallel_case

            2'b10: extcmd = romadr[15:8];
            2'b01: extcmd = romadr[7:0];
            2'b00,
            2'b11: extcmd = cddat;
            default: extcmd = 8'hxx;

        endcase

    //-------------------------------------------------------------------

    endmodule
```

```
/*****************************************************************************

Clock tree

*****************************************************************************/

module spclkgen ( clk40, ldldrdy, romdecs2_, rstbrdctl_, rstis_, rstseectl_,
        softbsy, softcmden, sspiocps, clk40b0, clk40b1, clk40b2, clk40b3,
        clk40b4, clk40b5, clk40b6, clk40b7 );

input   clk40;          // 40 MHz into SPIO block
input   ldldrdy;        // finished loading replacement ID
input   romdecs2_;      // rom/eeprom command active
input   rstbrdctl_;     // reset brdctl logic
input   rstis_;         // external reset
input   rstseectl_;     // reset seectl logic
input   softbsy;        // soft command in progress, enable fifo
input   softcmden;      // soft command enabled
input   sspiocps;       // slave SPIO controller present

output  clk40b0;        // buffered clk40: rstl_, softcmdens
output  clk40b1;        // gated by SSPIOCPS: sprstctl, arbomux, sbwcctl
output  clk40b2;        // gated by rstbrdctl & rstseectl: brdctl, seectl
output  clk40b3;        // gated by ldldrdy: lddatestat
output  clk40b4;        // gated by romact: romctl
output  clk40b5;        // gated by softcmden: softcntrl
output  clk40b6;        // gated by softbsy: spfifo
output  clk40b7;        // gated by ldldrdy & sspiocps: spcntrl

//-------------------------------------------------------------

supply1 vdd;

buf0101 drv_ini0 ( .Q(rstis1_), .A(rstis_) );
buf0101 drv_ini1 ( .Q(rstis2_), .A(rstis1_) );
buf0101 drv_ini2 ( .Q(rstis3_), .A(rstis2_) );
buf0102 drv_init ( .Q(init_clks), .A(rstis3_) );

//-------------------------------------------------------------
// clk40b0

buf0102  drv_b0 ( .Q(clk40b0), .A(clk40) );

//-------------------------------------------------------------
// clk40b1

inv01    drv_e1 ( .NQ(sspiocps_), .A(sspiocps) );
bts9     drv_b1 ( .Q(clk40b1), .A(clk40), .NE(sspiocps_) );
bushold  drv_h1 ( .DUMMY(dummy1), .Z(clk40b1) );
bts1     drv_l1 ( .Q(clk40b1), .A(vdd), .NE(init_clks) );

//-------------------------------------------------------------
// clk40b2

nor02    drv_e2 ( .NQ(enseebrd_), .A(rstbrdctl_), .B(rstseectl_) );
bts9     drv_b2 ( .Q(clk40b2), .A(clk40), .NE(enseebrd_) );
bushold  drv_h2 ( .DUMMY(dummy2), .Z(clk40b2) );
bts1     drv_l2 ( .Q(clk40b2), .A(vdd), .NE(init_clks) );

//-------------------------------------------------------------
// clk40b3

bts9     drv_b3 ( .Q(clk40b3), .A(clk40), .NE(ldldrdy) );
bushold  drv_h3 ( .DUMMY(dummy3), .Z(clk40b3) );
```

```
//-------------------------------------------------------------
// clk40b4

bts3      drv_b4 ( .Q(clk40b4), .A(clk40), .NE(romdecs2_) );
bushold   drv_h4 ( .DUMMY(dummy4), .Z(clk40b4) );
bts1      drv_l4 ( .Q(clk40b4), .A(vdd), .NE(init_clks) );

//-------------------------------------------------------------
// clk40b5

ndc0r  dff_e1 ( .Q(softcmdens1), .D(softcmden), .C(clk40b0), .NR(rstls_) );
ndc0r     dff_e2 ( .NQ(softcmdens2_), .D(softcmdens1), .C(clk40b0), .NR(rstls_) );
bts9      drv_b5 ( .Q(clk40b5), .A(clk40), .NE(softcmdens2_) );
bushold   drv_h5 ( .DUMMY(dummy5), .Z(clk40b5) );
bts1      drv_l5 ( .Q(clk40b5), .A(vdd), .NE(init_clks) );

//-------------------------------------------------------------
// clk40b6

inv01     drv_e6 ( .NQ(softbsy_), .A(softbsy) );
bts9      drv_b6 ( .Q(clk40b6), .A(clk40), .NE(softbsy_) );
bushold   drv_h6 ( .DUMMY(dummy6), .Z(clk40b6) );
bts1      drv_l6 ( .Q(clk40b6), .A(vdd), .NE(init_clks) );

//-------------------------------------------------------------
// clk40b7

and02     drv_e7 ( .Q(enshft_), .A(ldldrdy), .B(sspiocps_) );
bts9      drv_b7 ( .Q(clk40b7), .A(clk40), .NE(enshft_) );
bushold   drv_h7 ( .DUMMY(dummy7), .Z(clk40b7) );

endmodule
```

```
/*****************************************************************************

Function:

General purpose counter to support the SPIO shifter.  When cntron is inactive,
the count is synchronously reset.  When contron is active, the counter
increments every clock.




*****************************************************************************/

module spcnt ( clk40b7, cntron, rstspios_, cnt0, cnt8, cnt9, cnta );

parameter FFDLY = 1;        // clock-to-q delay

input     clk40b7;                  // input clock
input     cntron;           // counter on
input     rstspios_;        // power on reset

output    cnt0;             // count value == 4'h0
output    cnt8;             // count value == 4'h8
output    cnt9;             // count value == 4'h9
output    cnta;             // count value == 4'hA

// internal signals
reg       [3:0] cnt;        // counter value

//-----------------------------------------------------------------------
// decoded counter values

assign cnt0 = (cnt==4'h0);
assign cnt8 = (cnt==4'h8);
assign cnt9 = (cnt==4'h9);
assign cnta = (cnt==4'ha);

//-----------------------------------------------------------------------

always @(posedge clk40b7 or negedge rstspios_)          begin
  if (~rstspios_)
        cnt <= #FFDLY 4'h0;
  else
        cnt <= #FFDLY cntron ? cnt + 4'h1 : 4'h0;
end     // counter

//-----------------------------------------------------------------------

endmodule
```

```
//-----------------------------------------------------------------------

Function:

SPIO controller.  This module performs two operations:

        1) serialize a byte and shift out and
        2) receive serial data from the SPIO port.

(1) is performed when "sendbyte" becomes active.
(2) is performed when "readspio" becomes active.



*************************************************************************/

module spcntrl ( berren, chprstbsy, clk40b7, clrparerr_, eneito, ledtst,
        ledtstsig, mparcken, sendbyte, readspio, spiol, spioparerr, rstls_,
        rstspios_, shftout, ackdet, eito, bytesent, noack, sivalid, spioo,
        spiodrv_, parerrdet, shftin );

input   clk40b7, berren, clrparerr_, eneito, mparcken, sendbyte, readspio,
        ledtst, ledtstsig, chprstbsy, spiol, rstls_, rstspios_;
input   [7:0] shftout;

output  ackdet, bytesent, eito, noack, sivalid, spioo, spiodrv_, spioparerr,
        parerrdet;
output  [7:0] shftin;

// internal signals
wire    cntron;                         // counter on
wire    cnt0, cnt8, cnt9, cnta;         // counter values


//-----------------------------------------------------------------
// counter

spcnt spcnt ( .clk40b7(clk40b7), .cntron(cntron), .rstspios_(rstspios_),
        .cnt0(cnt0), .cnt8(cnt8), .cnt9(cnt9), .cnta(cnta) );

spshft spshft ( .ackwin(ackwin), .berren(berren), .bytesent(bytesent),
        .chprstbsy(chprstbsy), .clk40b7(clk40b7), .cnt9(cnt9),
        .clrparerr_(clrparerr_), .ledtst(ledtst), .ledtstsig(ledtstsig),
        .mparcken(mparcken), .rstls_(rstls_), .setsival(setsival),
        .readspio(readspio), .sendbyte(sendbyte), .shftout(shftout),
        .spiol(spiol), .sivalid(sivalid), .strtwin(strtwin), .ackdet(ackdet),
        .parerrdet(parerrdet), .shftin(shftin), .spioo(spioo),
        .spioparerr(spioparerr), .strtdet(strtdet) );

spctl spctl ( .chprstbsy(chprstbsy), .clk40b7(clk40b7), .ledtst(ledtst),
        .rstls_(rstls_), .rstspios_(rstspios_), .seteito(seteito),
        .setnoack(setnoack), .setsival(setsival), .spioen(spioen),
        .eito(eito), .noack(noack), .sivalid(sivalid), .spiodrv_(spiodrv_) );

spctlsm spctlsm ( .ackdet(ackdet), .clk40b7(clk40b7), .cnt0(cnt0),
        .cnt8(cnt8), .cnt9(cnt9), .cnta(cnta), .eito(eito), .eneito(eneito),
        .noack(noack), .rstspios_(rstspios_), .readspio(readspio),
        .strtdet(strtdet), .sendbyte(sendbyte), .ackwin(ackwin),
        .bytesent(bytesent), .seteito(seteito), .setnoack(setnoack),
        .setsival(setsival), .cntron(cntron), .strtwin(strtwin), .spioen(spioen)
        );

endmodule
```

```
/*****************************************************************************

Function:

Registers output signals from state machine.



*****************************************************************************/

module spctl ( chprstbsy, clk40b7, ledtst, rstis_, rstspios_, setelto, setnoack,
        setsival, spioen, elto, noack, sivalid, spiodrv_ );

parameter FFDLY = 1;

input   chprstbsy, clk40b7, ledtst, rstis_, rstspios_, setelto, setnoack,
        setsival, spioen;

output  elto;           // ESTAT/IDDAT time-out
output  noack;          // NOACK
output  sivalid;        // shift-in data valid
output  spiodrv_;       // SPIO output enable

//-----------------------------------------------------------------------
// Internal signals

reg     spiodrv_, sivalid, elto, noack;

//-----------------------------------------------------------------------
// output FFs

always @(posedge clk40b7 or negedge rstspios_)
  if (~rstspios_) begin
        noack <= #FFDLY 1'b0;
        elto <= #FFDLY 1'b0;
        sivalid <= #FFDLY 1'b0;
  end
  else begin
        noack <= #FFDLY setnoack;
        elto <= #FFDLY setelto;
        sivalid <= #FFDLY setsival;
  end

always @(posedge clk40b7 or negedge rstis_)
  if (~rstis_)
        spiodrv_ <= #FFDLY 1'b1;
  else
        spiodrv_ <= #FFDLY ~(spioen | ledtst | chprstbsy);

//-----------------------------------------------------------------------

endmodule       // spctl
```

```
/*****************************************************************

Function:

Serial Port Control State Machine.
This module keeps track of the "sendbyte" and "readspio" commands.  It uses
an external counter for timing.



*****************************************************************/

module spctlsm ( ackdet, clk40b7, cnt0, cnt8, cnt9, cnta, eito, eneito,
        noack, rstspios_, readspio, strtdet, sendbyte, ackwin, bytesent,
        setnoack, setsival, seteito, cntron, strtwin, spioen );

parameter EFDLY = 1;

parameter [2:0]             // synopsys enum state_info
        IDLE = 3'b000,      // idle
        SEND = 3'b001,      // send a byte
        STOP = 3'b010,      // stop bit
        WACK = 3'b011,      // wait for ACK
        EACK = 3'b100,      // wait for ACK termination
        STRT = 3'b101,      // wait for start bit
        RDAT = 3'b110,      // reading a byte of data
        CHKR = 3'b111;      // check whether to read again

input   ackdet, clk40b7, cnt0, cnt8, cnt9, cnta, eito, eneito, noack,
        rstspios_, readspio, sendbyte, strtdet;

output  ackwin;             // ACK detection window
output  bytesent;           // finished shifting a byte
output  cntron;             // turn counter on
output  seteito;            // set iddat/estat time-out
output  setsival;           // early sivalid
output  setnoack;           // generate NOACK signal
output  spioen;             // enable serial port output
output  strtwin;            // window for start-bit detection

//-----------------------------------------------------------------
// internal signals

                // synopsys state_vector state
reg     [2:0]   // synopsys enum state_info
        state;  // state vector
reg     [2:0]   // synopsys enum state_info
        next;   // next state

reg     spioen;             // enable SPIO output
reg     setnoack;           // set NOACK condition
reg     seteito;            // set ESTAT/IDDAT time-out
reg     setsival;

// registered outputs
reg     ackwin, strtwin, cntron, bytesent;

//-----------------------------------------------------------------
// combinational logic

always @( ackdet or state or sendbyte or readspio or cnta or cnt0
        or noack or eito or eneito or strtdet or cnt8 or cnt9 )
begin
```

```
ackwin = 1'b0;
strtwin = 1'b0;
cntron = 1'b0;
spioen = 1'b0;
bytesent = 1'b0;
setnoack = 1'b0;
setelto = 1'b0;
setsival = 1'b0;

case (state)

        IDLE: begin

          if (noack | elto) begin
                next = IDLE;
          end

          else begin

            case ({sendbyte, readspio})                    // synopsys parallel_case
                  2'b10: next = SEND;
                  2'b01: next = WACK;
                  2'b00,
                  2'b11: next = IDLE;
                  default: next = 3'bxxx;
            endcase

            cntron = sendbyte | readspio;
            spioen = sendbyte;

          end

        end

        SEND: begin
          next = cnta ? STOP : SEND;
          spioen = 1'b1;
          cntron = ~cnta;
          bytesent = cnta;
        end

        STOP: begin

          case ({sendbyte, readspio})              // synopsys parallel_case
                  2'b10:    next = SEND;
                  2'b01:    next = WACK;
                  2'b00,
                  2'b11:    next = IDLE;
                  default: next = 3'bxxx;
          endcase

          cntron = sendbyte;
          spioen = sendbyte;
        end

        WACK: begin
          ackwin = 1'b1;
          if (ackdet) begin
                next = EACK;
          end
          else if (enelto & cnt0 | ~enelto & cnta) begin                    // time-out
                next = IDLE;
                setnoack = ~enelto;
                setelto = enelto;
```

```
            end
            else begin
                    next = WACK;
                    cntron = 1'b1;
            end
          end

          EACK: begin
            ackwin = 1'b1;
            if (readspio) begin
                    next = ackdet ? EACK : STRT;
            end
            else begin
                    next = ackdet ? EACK : IDLE;
            end
          end

          STRT: begin
            next = strtdet ? RDAT : STRT;
            strtwin = 1'b1;
            cntron = strtdet;
          end

          RDAT: begin
            next = cnt9 ? CHKR : RDAT;
            cntron = ~cnt9;
            setslval = cnt8;
          end

          CHKR: begin
            next = readspio ? STRT : IDLE;
          end

          default: next = 3'bxxx;

    endcase

end

//-----------------------------------------------------------------
// state and output FFs

always @(posedge clk40b7 or negedge rstspios_)
    if (~rstspios_)
            state <= #FFDLY IDLE;
    else
            state <= #FFDLY next;

//-----------------------------------------------------------------

endmodule           // spctism
```

```
/*****************************************************************************

File:       spfifo.v

Function:

Soft Command read fifo.




*****************************************************************************/

module spfifo ( clk40b5, clk40b6, loadcnt, ensoftcdat, rdspiodat_, readcnt,
                rstfifo_, rstsoftcmd_, shftin, wrfifo, cnt0, cnt1, spioctlr10, spiordat
                );

input       clk40b5, clk40b6, loadcnt, ensoftcdat, rdspiodat_, rstfifo_,
            rstsoftcmd_, wrfifo;
input       [1:0] readcnt;
input       [7:0] shftin;

output      cnt0, cnt1;
output      [1:0] spioctlr10;
output      [7:0] spiordat;

//-----------------------------------------------------------------------
// bus declaration

wire        [1:0] rdptr, spioctlr10;

//-----------------------------------------------------------------------
// control

spfifoctl spfifoctl ( .clk40b5(clk40b5), .loadcnt(loadcnt), .readcnt(readcnt),
                .ensoftcdat(ensoftcdat), .rdspiodat_(rdspiodat_), .rstfifo_(rstfifo_),
                .rstsoftcmd_(rstsoftcmd_), .wrfifo(wrfifo), .cnt0(cnt0), .cnt1(cnt1),
                .rdptr(rdptr), .spioctlr10(spioctlr10), .wrfifo0(wrfifo0),
                .wrfifo1(wrfifo1), .wrfifo2(wrfifo2) );

//-----------------------------------------------------------------------
// registers

spfiforegs spfiforegs ( .clk40b6(clk40b6), .rdptr(rdptr), .rstfifo_(rstfifo_),
                .shftin(shftin), .wrfifo0(wrfifo0), .wrfifo1(wrfifo1),
                .wrfifo2(wrfifo2), .spiordat(spiordat) );

//-----------------------------------------------------------------------

endmodule
```

```
/*****************************************************************************

Function:

Soft Command fifo control logic.

*****************************************************************************/

module spflfoctl ( clk40b5, loadcnt, readcnt, ensoftcdat, rdsplodat_, rstfifo_,
        rstsoftcmd_, wrfifo, cnt0, cnt1, rdptr, sploctlr10, wrfifo0, wrfifo1,
        wrfifo2 );

parameter FFDLY = 1;

input   clk40b5, loadcnt, ensoftcdat, rdsplodat_, rstfifo_, rstsoftcmd_, wrfifo;
input   [1:0] readcnt;

output  cnt0;                   // no more byte to read
output  cnt1;                   // one more byte to read
output  [1:0] rdptr;            // fifo read pointer
output  [1:0] sploctlr10;       // sploctl read bits 1:0
output  wrfifo0;                // write first byte
output  wrfifo1;                // write second byte
output  wrfifo2;                // write third byte

//-----------------------------------------------------------------
// register declaration for outputs

reg     [1:0] rdptr, sploctlr10;
wire    [1:0] readcnt;

//-----------------------------------------------------------------
// internal signals

reg     [1:0] new_cnt;          // new read count
reg     [1:0] new_wrptr;        // new write pointer
reg     [1:0] rcnt;             // read decrement counter
reg     [1:0] wrptr;            // write pointer

//-----------------------------------------------------------------
// read counter

// decrementing logic
always @( rcnt or wrfifo or readcnt or loadcnt ) begin
  if (loadcnt)
        new_cnt = readcnt;
  else if (wrfifo)
        new_cnt = rcnt - 2'b01;
  else
        new_cnt = rcnt;
end

// count registers
always @(posedge clk40b5 or negedge rstsoftcmd_)
  if (~rstsoftcmd_)
        rcnt <= #FFDLY 2'b00;
  else
        rcnt <= #FFDLY new_cnt;

// latch for CIO data bus
always @(ensoftcdat or rcnt)
```

```
  if (ensoftcdat)
        spioctlr10 <= #FFDLY rcnt;

assign cnt1 = (rcnt==2'b01);
assign cnt0 = (rcnt==2'b00);

//-----------------------------------------------------------------
// write pointer

// incrementing logic
always @( wrptr or loadcnt or wrfifo )
  if (loadcnt)
        new_wrptr = 2'b00;
  else if (wrfifo)
        new_wrptr = wrptr + 2'b01;
  else
        new_wrptr = wrptr;

// pointer registers
always @(posedge clk40b5 or negedge rstsoftcmd_)
  if (~rstsoftcmd_)
        wrptr <= #FFDLY 2'b00;
  else
        wrptr <= #FFDLY new_wrptr;

// decoded write enables
assign wrfifo2 = wrfifo & (wrptr==2'b10);
assign wrfifo1 = wrfifo & (wrptr==2'b01);
assign wrfifo0 = wrfifo & (wrptr==2'b00);

//-----------------------------------------------------------------
// read pointer

always @(posedge rdspiodat_ or negedge rstfifo_)
  if (~rstfifo_)
        rdptr <= #FFDLY 2'b00;
  else
        rdptr <= #FFDLY rdptr + 2'b01;

//-----------------------------------------------------------------

endmodule          // spiifoctl
```

```
/*****************************************************************************

Function:

Soft Command Read FIFO.

History:

04/22/94 JYL        First rev.

*****************************************************************************/

module spfiforegs (clk40b6, rdptr, rstfifo_, shftin, wrfifo0, wrfifo1, wrfifo2,
        spiordat );

parameter FFDLY = 1;

input   clk40b6, rstfifo_, wrfifo0, wrfifo1, wrfifo2;
input   [1:0] rdptr;
input   [7:0] shftin;

output  [7:0] spiordat;

//-----------------------------------------------------------------------------
// register declaration for outputs

reg     [7:0] spiordat;

//-----------------------------------------------------------------------------
// internal registers

reg     [7:0] fifo0, fifo1, fifo2;

//-----------------------------------------------------------------------------
// fifo registers

always @(posedge clk40b6 or negedge rstfifo_)
  if (~rstfifo_) begin
        fifo2 <= #FFDLY 8'h00;
        fifo1 <= #FFDLY 8'h00;
        fifo0 <= #FFDLY 8'h00;
  end
  else begin
        fifo2 <= #FFDLY wrfifo2 ? shftin : fifo2;
        fifo1 <= #FFDLY wrfifo1 ? shftin : fifo1;
        fifo0 <= #FFDLY wrfifo0 ? shftin : fifo0;
  end

//-----------------------------------------------------------------------------
// output mux

always @( rdptr or fifo0 or fifo1 or fifo2 )
  case (rdptr)                  // synopsys parallel_case
        2'b00: spiordat = fifo0;
        2'b01: spiordat = fifo1;
        2'b10: spiordat = fifo2;
        2'b11: spiordat = 8'h00;
        default: spiordat = 8'hxx;
  endcase

//-----------------------------------------------------------------------------

endmodule       // spfiforegs
```

```
/*****************************************************************************

Function:

Top-level of SPIO block.

*****************************************************************************/

module spio ( berren, chprstb, clrparerr_, crbusy, clk40, cdadr_, cddat,
        cdwen_, csadr_, csren_, ltstsigl, mparcken, pac2spio, por,
        rstib_, sled_, spiol, romdec_, romadr, romwr, stpwenb, stpwlvlb,
        csdat, eepromen, lddat, lddrdy, noack, romrden, seems,
        spromrdy, spiodrv_, spioo, spnakabrt, spioackdet, spromclk_,
        spbrden, spseeen, spioparerr, spiobsy_, sspiocps, seebrdrdy,
        stretchseq );

input   berren;         // generate even parity
input   chprstb;        // chprst - HCNTRL (87h), bit 0
input   clrparerr_;     // clear parity error
input   crbusy;         // CIO bus is being read
input   clk40;          // 40 MHz clock (clkin)
input   [7:0] cdadr_;   // CIO destination address
input   [7:0] cddat;    // CIO destination data
input   cdwen_;         // CIO destination write enable
input   [7:0] csadr_;   // CIO source address
input   csren_;         // CIO source read enable
input   [7:0] ltstsigl; // led test mode signals
input   mparcken;       // enable parity checking
input   pac2spio;       // pauseack to SPIO
input   por;            // power on reset
input   rstib_;         // external reset
input   sled_;          // ecsi led level
input   spiol;          // input from SPIO pin
input   romdec_;        // ROM request from phost
input   [15:0] romadr;  // ROM address
input   romwr;          // ROM write command
input   stpwenb;        // SXFRCTL1 bit 0
input   stpwlvlb;       // DEVCONFIG bit 1

output  [7:0] csdat, lddat;
output  eepromen, lddrdy, noack, romrden, spromrdy, spiodrv_,
        sspiocps, seems, spioo, spnakabrt, spromclk_, spioparerr,
        spbrden, spseeen, spiobsy_, stretchseq, spioackdet, seebrdrdy;

//-----------------------------------------------------------------

wire    eito, sivalid, eiread, eneito, sendbyte, readspio, spiol,
        ackdet, bytesent;

wire    [15:0] romadr;
wire    [7:0] ltstsigl, spiordat, spiowdat, shftin, shftout;
wire    [3:0] spioctlr74;
wire    [2:0] seectl, brdctl, bnddat;
wire    [1:0] spioctlr10, soft;

//-----------------------------------------------------------------
// Instantiate sub-modules

spclkgen spclkgen ( .clk40(clk40), .lddrdy(lddrdy), .romdecs2_(romdecs2_),
        .rstbrdctl_(rstbrdctl_), .rstls_(rstls_), .rstseectl_(rstseectl_),
        .softbsy(softbsy), .softcmden(softcmden), .sspiocps(sspiocps),
```

```
        .clk40b0(clk40b0), .clk40b1(clk40b1), .clk40b2(clk40b2),
        .clk40b3(clk40b3), .clk40b4(clk40b4), .clk40b5(clk40b5),
        .clk40b6(clk40b6), .clk40b7(clk40b7) );

sprstctl sprstctl ( .chprstack(chprstack), .chprstb(chprstb), .clk40b0(clk40b0),
        .clk40b1(clk40b1),
        .rstib_(rstib_), .softabrtok(softabrtok), .sspiocps(sspiocps),
        .chprstbsy(chprstbsy), .chprstreq(chprstreq), .swrstrdy(swrstrdy),
        .rstis_(rstis_), .rstspios_(rstspios_) );

softcntrl softcntrl ( .ackdet(ackdet), .bytesent(bytesent), .cddat(cddat),
        .clk40b5(clk40b5), .clk40b6(clk40b6), .ensoftcdat(ensoftcdat),
        .ltstsigl(ltstsigl), .noack(noack), .rdspiodat_(rdspiodat_),
        .rstspios_(rstspios_), .shftin(shftin), .sivalid(sivalid),
        .softack(softack), .softcmden(softcmden), .wrspioctl_(wrspioctl_),
        .wrspiodat_(wrspiodat_), .ledtst(ledtst), .ledtstsig(ledtstsig),
        .softabrtok(softabrtok), .softbsy(softbsy), .softread(softread),
        .softreq(softreq), .softsend(softsend), .spioctlr10(spioctlr10),
        .spioctlr74(spioctlr74), .spiordat(spiordat), .spiowdat(spiowdat) );

brdcntrl brdcntrl ( .ackdet(ackdet), .brdctlen(brdctlen), .brdack(brdack),
        .bytesent(bytesent), .cddat(cddat), .clk40b2(clk40b2), .noack(noack),
        .rdbrdctl_(rdbrdctl_), .rstspios_(rstspios_), .shftin(shftin),
        .sivalid(sivalid), .wrbrdctl_(wrbrdctl_), .brdbsy(brdbsy),
        .brdctl(brdctl), .brdctlrdy(brdctlrdy), .brddat(brddat),
        .brdread(brdread), .brdreq(brdreq), .brdsend(brdsend),
        .rstbrdctl_(rstbrdctl_) );

seecntrl seecntrl ( .ackdet(ackdet), .bytesent(bytesent), .clk40b2(clk40b2),
        .noack(noack), .rdseectl_(rdseectl_), .rstspios_(rstspios_),
        .seeack(seeack), .seectlen(seectlen), .shftin0(shftin[0]),
        .sivalid(sivalid), .wrseectl_(wrseectl_), .cddat(cddat),
        .rstseectl_(rstseectl_), .seectl(seectl), .seectlrdy(seectlrdy),
        .seedi(seedi), .seems(seems), .seerdy(seerdy), .seeread(seeread),
        .seereq(seereq), .seebsy(seebsy), .seesend(seesend) );

sbwcctl sbwcctl ( .ackdet(ackdet), .bytesent(bytesent), .clk40b1(clk40b1),
        .ledack(ledack), .noack(noack), .rstspios_(rstspios_), .sled_(sled_),
        .sspiocps(sspiocps), .stpwack(stpwack), .stpwenb(stpwenb),
        .stpwivlb(stpwivlb), .ledreq(ledreq), .ledstate(ledstate),
        .sbwcbsy(sbwcbsy), .sbwcread(sbwcread), .stpe(stpe), .stpl(stpl),
        .stpwreq(stpwreq) );

sprw sprw ( .brdctlen(brdctlen), .brdctlrdy(brdctlrdy), .csadr_(csadr_),
        .csren_(csren_), .cdadr_(cdadr_), .cdwen_(cdwen_), .clk40b0(clk40b0),
        .crbusy(crbusy), .pac2spio(pac2spio), .por(por), .seectlen(seectlen),
        .seectlrdy(seectlrdy), .softcmden(softcmden), .rd1b(rd1b),
        .ensoftcdat(ensoftcdat), .rd1d(rd1d), .rd1e(rd1e), .rdbrdctl_(rdbrdctl_),
        .rdseectl_(rdseectl_), .rdspiodat_(rdspiodat_), .spiobsy_(spiobsy_),
        .seebrdrdy(seebrdrdy), .stretchseq(stretchseq), .wrbrdctl_(wrbrdctl_),
        .wrspiocap_(wrspiocap_), .wrseectl_(wrseectl_), .wrspioctl_(wrspioctl_),
        .wrspiodat_(wrspiodat_) );

sprdmux sprdmux ( .spioctlr10(spioctlr10), .spioctlr74(spioctlr74),
        .spiobsy_(spiobsy_), .spromclk_(spromclk_), .rd1b(rd1b),
        .rd1d(rd1d), .rd1e(rd1e), .brdctl(brdctl), .brddat(brddat),
        .seectl(seectl), .seedi(seedi), .seems(seems), .seerdy(seerdy),
        .shftin(shftin), .spiordat(spiordat), .soft(soft), .rombsy(rombsy),
        .softcmden(softcmden), .brdctlen(brdctlen), .seectlen(seectlen),
        .eepromen(eepromen), .romrden(romrden), .sspiocps(sspiocps),
        .csdat(csdat) );

spromctl spromctl ( .ackdet(ackdet), .bytesent(bytesent), .noack(noack),
        .sivalid(sivalid), .clk40b1(clk40b1), .clk40b4(clk40b4),
        .romack(romack), .romwr(romwr), .por(por), .romdec_(romdec_),
```

```
        .rombsy(rombsy), .romdecs2_(romdecs2_), .spromrdy(spromrdy),
        .romread(romread), .romreq(romreq), .romsend(romsend),
        .selroma0(selroma0), .selroma1(selroma1), .spromclk_(spromclk_) );

iddatestat iddatestat ( .cddat5(cddat[5]), .clk40b3(clk40b3), .eito(eito),
        .parerrdet(parerrdet), .rstis_(rstis_), .shftin(shftin),
        .sivalid(sivalid), .swrstrdy(swrstrdy), .wrspiocap_(wrspiocap_),
        .brdctlen(brdctlen), .eepromen(eepromen), .eiread(eiread),
        .eneito(eneito), .iddat(iddat), .idldrdy(idldrdy), .romrden(romrden),
        .seectlen(seectlen), .soft(soft), .softcmden(softcmden),
        .spbrden(spbrden), .spseeen(spseeen), .sspiocps(sspiocps) );

spcntrl spcntrl ( .berren(berren), .chprstbsy(chprstbsy), .clk40b7(clk40b7),
        .clrparerr_(clrparerr_), .eneito(eneito), .ledtst(ledtst),
        .ledtstsig(ledtstsig), .mparcken(mparcken),
        .sendbyte(sendbyte), .readspio(readspio), .spiol(spiol),
        .spioparerr(spioparerr), .rstis_(rstis_), .rstspios_(rstspios_),
        .shftout(shftout), .ackdet(ackdet), .eito(eito), .bytesent(bytesent),
        .noack(noack), .sivalid(sivalid), .spioo(spioo), .spiodrv_(spiodrv_),
        .parerrdet(parerrdet), .shftin(shftin) );

sparbomux sparbomux ( .ackdet(ackdet), .brdbsy(brdbsy), .brdread(brdread),
        .brdreq(brdreq), .brdsend(brdsend), .cddat(cddat),
        .chprstbsy(chprstbsy), .chprstreq(chprstreq), .clk40b1(clk40b1),
        .eiread(eiread), .ledstate(ledstate), .ledreq(ledreq), .noack(noack),
        .romadr(romadr), .rombsy(rombsy), .romread(romread), .romreq(romreq),
        .romsend(romsend), .romwr(romwr), .rstspios_(rstspios_),
        .sbwcbsy(sbwcbsy), .sbwcread(sbwcread), .seebsy(seebsy),
        .seeread(seeread), .seereq(seereq), .seesend(seesend),
        .selroma1(selroma1), .selroma0(selroma0), .softbsy(softbsy),
        .softread(softread), .softreq(softreq), .softsend(softsend),
        .spiobsy_(spiobsy_), .spiowdat(spiowdat), .stpe(stpe), .stpl(stpl),
        .stpwreq(stpwreq), .brdack(brdack), .chprstack(chprstack),
        .ledack(ledack), .readspio(readspio), .romack(romack), .seeack(seeack),
        .shftout(shftout), .sendbyte(sendbyte), .softack(softack),
        .spioackdet(spioackdet), .spnakabrt(spnakabrt), .stpwack(stpwack) );

endmodule
```

```
/*****************************************************************************

Function:

Serial port read mux to csdat.

*****************************************************************************/

module sprdmux ( sploctlr10, sploctlr74, splobsy_, spromclk_, rd1b, rd1d, rd1e,
        brdctl, brddat, seectl, seedi, seems, seerdy, shftin, splordat, soft,
        rombsy, softcmden, brdctlen, seectlen, eepromen, romrden, sspiocps,
        csdat );

input   splobsy_, spromclk_;
input   rd1b, rd1d, rd1e;
input   [7:0] splordat, shftin;
input   [2:0] seectl, brdctl, brddat;
input   seems, seedi, seerdy;
input   [1:0] soft, sploctlr10;
input   [3:0] sploctlr74;
input   rombsy, softcmden, brdctlen, seectlen, eepromen, romrden, sspiocps;

output  [7:0] csdat;        // read data to CIO bus

//-------------------------------------------------------------------
// internal signals

reg     [7:0] ciooregs;
wire    [7:0] sploctl, sprdbus, shftin, spiocap, seectio, brdctio;
wire    drvcsdat_;

//-------------------------------------------------------------------
// csdat output enable

assign drvcsdat_ = splobsy_ & spromclk_;
assign sprdbus = rombsy ? shftin : ciooregs;
assign #1 csdat = drvcsdat_ ? 8'hzz : sprdbus;

//-------------------------------------------------------------------

assign seectio = {2'b00, seems, seerdy, seectl[2:0], seedi};
assign brdctio = {brddat, brdctl, 2'b00};
assign spiocap = {soft,softcmden,brdctlen,seectlen,eepromen,romrden,sspiocps};
assign sploctl = {sploctlr74, 2'b00, sploctlr10};

always @( rd1b or rd1d or rd1e or softcmden or brdctio or spiocap or seectio
        or sploctl or splordat )

  casex ({rd1b, rd1d, rd1e, softcmden})          // synopsys parallel_case
        4'b1xxx: ciooregs = spiocap;
        4'bx1x0: ciooregs = brdctio;
        4'bx1x1: ciooregs = splordat;
        4'bxx10: ciooregs = seectio;
        4'bxx11: ciooregs = sploctl;
        default: ciooregs = 8'hxx;

  endcase

//-------------------------------------------------------------------

endmodule
```

```
/*****************************************************************************

Function:

Serial port rom controller.

*****************************************************************************/

module spromctl ( ackdet, bytesent, noack, sivalid, clk40b1, clk40b4, romack,
        romwr, por, romdec_, rombsy, romdecs2_, romread, romreq, romsend,
        selroma0, selroma1, spromrdy, spromclk_ );

parameter FFDLY = 1;

input   ackdet, bytesent, noack, sivalid, clk40b1, clk40b4, romack, romwr, por, romdec_;

output  rombsy;         // ROM command in progress
output  romdecs2_;      // rom command decoded
output  romsend;        // send a ROM command byte
output  selroma0;
output  selroma1;
output  spromclk_;      // clock data in romdo
output  romread;
output  romreq;         // ROM request to arbiter
output  spromrdy;       // ROM command finished

//-----------------------------------------------------------------
// variable declaration

reg     rombsy, romsend;
reg     romdecs1;       // synced romdec_
reg     romdecs2;
reg     romread, spromrdy, romreq;
wire    setread, setrdy, setbsy, setsend;

//-----------------------------------------------------------------
// generate request

always @(posedge clk40b1 or posedge por)
  if (por) begin
        romdecs1 <= #FFDLY 1'b0;
        romdecs2 <= #FFDLY 1'b0;
        romreq <= #FFDLY 1'b0;
  end
  else begin
        romdecs1 <= #FFDLY ~romdec_;
        romdecs2 <= #FFDLY romdecs1;
        romreq <= #FFDLY romreq ? ~romack : romdecs1 & ~romdecs2;
  end

//-----------------------------------------------------------------
// combinational outputs

assign spromclk_ = ~( rombsy & sivalid );
assign romdecs2_ = ~romdecs2;

//-----------------------------------------------------------------
// registered outputs

always @(posedge clk40b4 or posedge por)
  if (por) begin
```

```
            rombsy <= #FFDLY 1'b0;
            romsend <= #FFDLY 1'b0;
            romread <= #FFDLY 1'b0;
            spromrdy <= #FFDLY 1'b0;
    end
    else begin
            rombsy <= #FFDLY setbsy;
            romsend <= #FFDLY setsend;
            romread <= #FFDLY setread;
            spromrdy <= #FFDLY setrdy;
    end

//-----------------------------------------------------------------------
// instantiate state machine

spromsm spromsm ( .ackdet(ackdet), .bytesent(bytesent), .noack(noack),
            .sivalid(sivalid), .clk40b4(clk40b4), .romack(romack), .romwr(romwr),
            .por(por), .setread(setread), .setbsy(setbsy), .setrdy(setrdy),
            .setsend(setsend), .selroma0(selroma0), .selroma1(selroma1) );

//-----------------------------------------------------------------------

endmodule
```

```
/*****************************************************************************

Function:

Serial port rom control state-machine.

*****************************************************************************/

module spromsm ( ackdet, bytesent, noack, sivalid, clk40b4, romack, romwr, por,
        setread, setbsy, setrdy, setsend, selroma0, selroma1 );

parameter FFDLY = 1;

parameter [2:0]         // synopsys enum state_info
        ID = 3'b000,    // Idle
        CC = 3'b001,    // command code
        A0 = 3'b010,    // address low
        A1 = 3'b011,    // address high
        D0 = 3'b100,    // data 0
        D1 = 3'b101,    // data 1
        D2 = 3'b110,    // data 2
        D3 = 3'b111;    // data 3

input   ackdet, bytesent, noack, sivalid, clk40b4, romack, romwr, por;

output  setread;        // read SPIO port
output  setbsy;  // rombusy true next clock
output  setrdy;         // set romrdy next clock
output  setsend;        // send byte next clock
output  selroma0;       // select address low
output  selroma1;       // select address high

//-----------------------------------------------------------------
// declare registers

                // synopsys state_vector state
reg     [2:0]   // synopsys enum state_info
        state;  // current state
reg     [2:0]   // synopsys enum state_info
        next;   // next state

reg     setread, setbsy, setrdy, setsend, selroma0, selroma1;

//-----------------------------------------------------------------
// combinational logic

always @( state or romack or romwr or bytesent or ackdet or sivalid or noack )
begin

   setread = 1'b0;
   setbsy = 1'b1;
   setrdy = 1'b0;
   setsend = 1'b0;
   selroma0 = 1'b0;
   selroma1 = 1'b0;

   case (state)         // synopsys parallel_case

        ID: begin
           next = romack ? CC : ID;
           setbsy = romack;
```

```
end

CC: begin
   next = bytesent ? A0 : CC;
   setsend = bytesent;
end

A0: begin
   next = bytesent ? A1 : A0;
   selroma0 = 1'b1;
   setsend = bytesent;
end

A1: begin
   next = bytesent ? D0 : A1;
   selroma1 = 1'b1;
   setsend = romwr & bytesent;
   setread = ~romwr & bytesent;
end

D0: begin
   if (romwr) begin
         next = bytesent ? D1 : D0;
         setread = bytesent;
   end
   else begin
         if (noack) begin
            next = ID;
            setbsy = 1'b0;
         end
         else begin
            next = sivalid ? D1 : D0;
            setread = 1'b1;
         end
   end
end

D1: begin
   if (romwr) begin
         if (noack) begin
            next = ID;
            setbsy = 1'b0;
         end
         else begin
            next = ackdet ? D2 : D1;
         end
   end else begin
         next = sivalid ? D2 : D1;
         setread = 1'b1;
   end
end

D2: begin
   if (romwr) begin
         next = ackdet ? D2 : ID;
         setbsy = ackdet;
         setrdy = ~ackdet;
   end
   else begin
         next = sivalid ? D3 : D2;
         setread = 1'b1;
   end
end

D3 : begin
```

```
            next = sivalid ? ID : D3;
            setbsy = 1'b1;
            setrdy = sivalid;
            setread = ~sivalid;
          end

          default: next = 3'bxxx;

  endcase

end       // always

//-----------------------------------------------------------------------
// state vector

always @(posedge clk40b4 or posedge por)
  if (por)
          state <= #FFDLY ID;
  else
          state <= #FFDLY next;

//-----------------------------------------------------------------------

endmodule
```

```
/*****************************************************************

File:       sprstctl.v

Function:

Serial port reset control.


*****************************************************************/

module sprstctl ( chprstack, chprstb,  clk40b0, clk40b1, rstib_, softabrtok, sspiocps,
        chprstbsy, chprstreq, swrstrdy, rstis_, rstspios_ );

parameter FFDLY = 1;

input   chprstack, chprstb,  clk40b0, clk40b1, rstib_, softabrtok, sspiocps;

output  chprstbsy;          // chprst in progress
output  chprstreq;          // request for SPIO port
output  swrstrdy;           // software reset ready
output  rstis_;             // HW reset
output  rstspios_;          // HW/SW reset spio block

//-----------------------------------------------------------------
// register declaration for outputs

reg     swrstrdy, chprstbsy, chprstreq;

//-----------------------------------------------------------------
// internal signals

reg     [3:0] cnt, new_cnt;                                 // reset counter
reg     rstis1_, rstis2_, spchprst_;            // synced reset signals
reg     chprstpend, chprstpends1, chprstpends2;         // SW reset pending
wire    rstswrst_, rstchprst_, cnt0, strt_swrst;
reg     clrswrst_;

//-----------------------------------------------------------------
// reset logic

always @(posedge clk40b0) begin
  rstis1_ <= #FFDLY rstib_;
  rstis2_ <= #FFDLY rstis1_;
end

assign rstis_    = rstib_ & rstis1_ & rstis2_;
assign rstchprst_ = rstis_ & sspiocps;
assign rstspios_ = rstis_ & spchprst_;
assign rstswrst_ = rstchprst_ & clrswrst_;

always @(posedge clk40b1 or negedge rstchprst_)
  if (~rstchprst_)
        spchprst_ <= #FFDLY 1'b1;
  else
        spchprst_ <= #FFDLY ~strt_swrst;

//-----------------------------------------------------------------
// request logic

always @(posedge chprstb or negedge rstspios_)
  if (~rstspios_)
        chprstpend <= #FFDLY 1'b0;
  else
```

```
        chprstpend <= #FFDLY 1'b1;

always @(posedge clk40b1 or negedge rstchprst_)
  if (~rstchprst_) begin
        chprstpends1 <= #FFDLY 1'b0;
        chprstpends2 <= #FFDLY 1'b0;
        chprstreq <= #FFDLY 1'b0;
  end
  else begin
        chprstpends1 <= #FFDLY chprstpend;
        chprstpends2 <= #FFDLY chprstpends1;
        chprstreq <= #FFDLY chprstpends1 & ~chprstpends2
                | chprstreq & ~chprstack & ~softabrtok;
  end

assign strt_swrst = chprstack | chprstreq & softabrtok;

//-----------------------------------------------------------------
// software reset ready logic

always @(posedge clk40b1 or negedge rstchprst_)
  if (~rstchprst_)
        clrswrst_ <= #FFDLY 1'b1;
  else
        clrswrst_ <= (cnt!=4'hf);

always @(posedge chprstb or negedge rstswrst_)
  if (~rstswrst_)
        swrstrdy <= #FFDLY 1'b1;
  else
        swrstrdy <= #FFDLY 1'b0;

//-----------------------------------------------------------------
// busy logic

always @(posedge clk40b1 or negedge rstchprst_)
  if (~rstchprst_)
        chprstbsy <= #FFDLY 1'b0;
  else
        chprstbsy <= #FFDLY strt_swrst | ~cnt0;

//-----------------------------------------------------------------
// counter

assign cnt0 = (cnt==4'h0);

// counting logic
always @( cnt or cnt0 or strt_swrst )
  if (strt_swrst)
        new_cnt = 4'h1;
  else
        new_cnt = cnt0 ? 4'h0 : cnt + 4'h1;

// counter registers
always @(posedge clk40b1 or negedge rstchprst_)
  if (~rstchprst_)
        cnt <= #FFDLY 4'h0;
  else
        cnt <= #FFDLY new_cnt;

//-----------------------------------------------------------------

endmodule
```

```
/*****************************************************************************

Function:

Read/write decoder

*****************************************************************************/
module sprw ( brdctlen, brdctlrdy, csadr_, csren_, cdadr_, cdwen_, clk40b0,
        crbusy, pac2spio, por, seectlrdy, softcmden, ensoftcdat, rd1b, rd1d,
        rd1e, rdbrdctl_, rdseectl_, rdspiodat_, spiobsy_, seectlen, seebrdrdy,
        stretchseq, wrbrdctl_, wrspiocap_, wrseectl_, wrspioctl_, wrspiodat_ );

input   [7:0] csadr_, cdadr_;
input   brdctlen, brdctlrdy, csren_, cdwen_, clk40b0, crbusy, pac2spio, por,
        seectlen, seectlrdy, softcmden;

output  ensoftcdat, rd1b, rd1d, rd1e, rdbrdctl_, rdseectl_, rdspiodat_,
        seebrdrdy, spiobsy_, stretchseq, wrbrdctl_, wrspiocap_, wrseectl_,
        wrspioctl_, wrspiodat_;

sprwdec sprwdec ( .csadr_(csadr_), .csren_(csren_), .cdadr_(cdadr_),
        .cdwen_(cdwen_), .crbusy(crbusy), .por(por), .softcmden(softcmden),
        .rd1b(rd1b), .rd1d(rd1d), .rd1e(rd1e), .rdbrdctl_(rdbrdctl_),
        .rdseectl_(rdseectl_), .rdspiodat_(rdspiodat_), .spiobsy(spiobsy),
        .spiobsy_(spiobsy_), .wrbrdctl_(wrbrdctl_), .wrspiocap_(wrspiocap_),
        .wrseectl_(wrseectl_), .wrspioctl_(wrspioctl_), .wrspiodat_(wrspiodat_) );

spstretch spstretch ( .brdctlen(brdctlen), .brdctlrdy(brdctlrdy),
        .hostrden(hostrden), .pac2spio(pac2spio), .por(por),
        .rdseebrds_(rdseebrds_), .seectlen(seectlen), .seectlrdy(seectlrdy),
        .seqrden(seqrden), .wrbrdctl_(wrbrdctl_), .wrseectl_(wrseectl_),
        .ensoftcdat(ensoftcdat), .seebrdrdy(seebrdrdy), .stretchseq(stretchseq) );

sprwctl sprwctl ( .brdctlen(brdctlen), .clk40b0(clk40b0), .csren_(csren_), .por(por),
        .rdbrdctl_(rdbrdctl_), .rdseectl_(rdseectl_), .seectlen(seectlen),
        .spiobsy(spiobsy), .spiobsy_(spiobsy_), .hostrden(hostrden),
        .rdseebrds_(rdseebrds_), .seqrden(seqrden) );

endmodule
```

```
/*****************************************************************************

Function:

Read/write decoder



*****************************************************************************/
module sprwctl (brdctlen, clk40b0, csren_, por, rdbrdctl_, rdseectl_, seectlen,
        spiobsy, spiobsy_, hostrden, rdseebrds_, seqrden);

parameter FFDLY = 1;

input   brdctlen, clk40b0, csren_, por, rdbrdctl_, rdseectl_, seectlen, spiobsy,
        spiobsy_;

output  hostrden;                       // enable soft control read data for host
output  rdseebrds_;
output  seqrden;                        // enable soft control read data for sequencer

//-----------------------------------------------------------------------
// internal signals
wire    rdbrdctl_, rdseectl_, spiobsy;
reg     hostrden, seqrden, rdseebrds_;

//-----------------------------------------------------------------------
// stretch sequencer

always @(negedge clk40b0 or posedge por)
  if (por)
        rdseebrds_ <= #FFDLY 1'b1;
  else
        rdseebrds_ <= #FFDLY csren_
                | (~seectlen | rdseectl_) & (~brdctlen | rdbrdctl_);

//-----------------------------------------------------------------------
// hold softctl data for reads

// phost
always @(posedge clk40b0 or negedge spiobsy_)
  if (~spiobsy_)
        hostrden <= #FFDLY 1'b0;
  else
        hostrden <= #FFDLY 1'b1;

// SEQ
always @(negedge clk40b0 or negedge spiobsy)
  if (~spiobsy)
        seqrden <= #FFDLY 1'b1;
  else
        seqrden <= #FFDLY 1'b0;

endmodule
```

```
/*****************************************************************************

Function:

Serial-to-parallel and parallel-to-serial converter.  Automatically generate
start, parity and stop bits when shifting out.  Detects ACK and start bits
and checks parity when reading.



*****************************************************************************/

module spshft ( ackwin, berren, bytesent, chprstbsy, clk40b7, cnt9, clrparerr_,
        ledtst, ledtstsig, mparcken, rstls_, readspio, sendbyte, setsival,
        shftout, spiol, sivalid, strtwin, ackdet, parerrdet, shftin, spioo,
        spioparerr, strtdet );

parameter FFDLY = 1;

input   ackwin, berren, bytesent, clk40b7, cnt9, clrparerr_, mparcken, rstls_,
        chprstbsy, ledtst, ledtstsig, readspio, sendbyte, sivalid, spiol,
        setsival, strtwin;
input   [7:0] shftout;

output  ackdet;         // ACK detected
output  parerrdet;      // parity error detected
output  [7:0] shftin;   // shifted in data
output  spioo;          // SPIO serial output
output  spioparerr;     // SPIO parity error
output  strtdet;        // start bit detected

//-----------------------------------------------------------------
// declare registers
reg     spioparerr, parerrdet;
reg     [7:0] shftin;

//-----------------------------------------------------------------
// internal signals
reg     [7:0] shft;     // shifter register
wire    pardi;          // parity data in
wire    paro;           // parity out
reg     parod;          // parity out delayed
wire    rstparerr_;     // reset parity error
reg     berrens1, berrens2;
reg     sp_shfto;       // serialized output

//-----------------------------------------------------------------
// parity generator / checker

assign pardi = (readspio | sivalid) ? shft[7] : sp_shfto;
assign paro = strtdet | sendbyte | (~pardi ^ parod);

always @(posedge clk40b7 or negedge rstls_) begin
  if (~rstls_) begin
        berrens1 <= #FFDLY 1'b0;
        berrens2 <= #FFDLY 1'b0;
        parod <= #FFDLY 1'b0;
        parerrdet <= #FFDLY 1'b0;
  end
  else begin
        berrens1 <= #FFDLY berren;
        berrens2 <= #FFDLY berrens1;
        parod <= #FFDLY paro;
```

```
/*****************************************************************************

Function:

Serial-to-parallel and parallel-to-serial converter.  Automatically generate
start, parity and stop bits when shifting out.  Detects ACK and start bits
and checks parity when reading.



*****************************************************************************/

module spshft ( ackwin, berren, bytesent, chprstbsy, clk40b7, cnt9, clrparerr_,
        ledtst, ledtstsig, mparcken, rstis_, readspio, sendbyte, setsival,
        shftout, spiol, sivalid, strtwin, ackdet, parerrdet, shftin, spioo,
        spioparerr, strtdet );

parameter FFDLY = 1;

input   ackwin, berren, bytesent, clk40b7, cnt9, clrparerr_, mparcken, rstis_,
        chprstbsy, ledtst, ledtstsig, readspio, sendbyte, sivalid, spiol,
        setsival, strtwin;
input   [7:0] shftout;

output  ackdet;         // ACK detected
output  parerrdet;      // parity error detected
output  [7:0] shftin;   // shifted in data
output  spioo;          // SPIO serial output
output  spioparerr;     // SPIO parity error
output  strtdet;        // start bit detected

//-----------------------------------------------------------------
// declare registers
reg     spioparerr, parerrdet;
reg     [7:0] shftin;

//-----------------------------------------------------------------
// internal signals
reg     [7:0] shft;     // shifter register
wire    pardi;          // parity data in
wire    paro;           // parity out
reg     parod;          // parity out delayed
wire    rstparerr_;     // reset parity error
reg     berrens1, berrens2;
reg     sp_shfto;       // serialized output

//-----------------------------------------------------------------
// parity generator / checker

assign pardi = (readspio | sivalid) ? shft[7] : sp_shfto;
assign paro = strtdet | sendbyte | (~pardi ^ parod);

always @(posedge clk40b7 or negedge rstis_) begin
  if (~rstis_) begin
        berrens1 <= #FFDLY 1'b0;
        berrens2 <= #FFDLY 1'b0;
        parod <= #FFDLY 1'b0;
        parerrdet <= #FFDLY 1'b0;
  end
  else begin
        berrens1 <= #FFDLY berren;
        berrens2 <= #FFDLY berrens1;
        parod <= #FFDLY paro;
```

```
            parerrdet <= #FFDLY sivalid & ~paro;
    end
end

assign rstparerr_ = mparcken & rstis_ & clrparerr_;

always @(posedge clk40b7 or negedge rstparerr_)
    if (~rstparerr_)
            spioparerr <= #FFDLY 1'b0;
    else
            spioparerr <= #FFDLY spioparerr | parerrdet;

//-----------------------------------------------------------------------
// shifter

assign spioo = (~ledtst & sp_shfto | ledtst & ledtstsig);

always @(posedge clk40b7 or negedge rstis_)
    if (~rstis_) begin
            sp_shfto <= #FFDLY 1'b1;
            shft[7] <= #FFDLY 1'b1;
    end
    else begin
            if (sendbyte) begin
               sp_shfto <= #FFDLY 1'b0;
               shft[7] <= #FFDLY shftout[7];
            end
            else begin
               sp_shfto <= #FFDLY ~chprstbsy
                        & (cnt9 ? ~paro ^ berrens2 : shft[0] | bytesent);
               shft[7] <= #FFDLY spioi;
            end
    end

always @(posedge clk40b7 or negedge rstis_)
    if (~rstis_)
            shft[6:0] <= #FFDLY 7'b1111111;
    else
            shft[6:0] <= #FFDLY sendbyte ? shftout[6:0] : shft[7:1];

//-----------------------------------------------------------------------
// ACK and start bit detector

always @(posedge clk40b7 or negedge rstis_)
    if (~rstis_)
            shftin <= #FFDLY 8'h00;
    else
            shftin <= #FFDLY setsival ? shft : shftin;

assign ackdet = ackwin & ~shft[7] & ~shft[6];
assign strtdet = strtwin & ~shft[7] & shft[6];

//-----------------------------------------------------------------------

endmodule // spshft
```

```
/*****************************************************************************

Function:

Read/write timing control. Ajust sequncer timing and hold data for sploctl
reads.

History:

00/00/00 JYL        First rev.

*****************************************************************************/
module spstretch ( brdctlen, brdctlrdy, hostrden, pac2spio, por, rdseebrds_,
        seectlen, seectlrdy, seqrden, wrbrdctl_, wrseectl_, ensoftcdat,
        seebrdrdy, stretchseq);

parameter FFDLY = 1;

input   brdctlen, brdctlrdy, hostrden, pac2spio, por, rdseebrds_,
        seectlen, seectlrdy, seqrden, wrbrdctl_, wrseectl_;

output  ensoftcdat;                 // enable soft control read data
output  seebrdrdy;
output  stretchseq;                 // strech sequencer cycle

//-------------------------------------------------------------------
reg     stretchseq;
wire    act_seq_stretch, rst_stretch_;

//-------------------------------------------------------------------
// stretch sequencer

assign act_seq_stretch =
        ~(
          rdseebrds_
          & (~seectlen | wrseectl_)
          & (~brdctlen | wrbrdctl_)
        )
        & ~pac2spio;

always @(posedge act_seq_stretch or negedge rst_stretch_)
  if (~rst_stretch_)
        stretchseq <= #FFDLY 1'b0;
  else
        stretchseq <= #FFDLY 1'b1;

assign seebrdrdy = brdctlrdy | seectlrdy;
assign rst_stretch_ = ~(por | seebrdrdy);

//-------------------------------------------------------------------
// hold softctl data for reads

assign ensoftcdat = pac2spio ? hostrden : seqrden;

endmodule // spstretch
```

```
`timescale 1ns/1ns
```

SERIAL PORT FOR A HOST ADAPTER INTEGRATED CIRCUIT USING A SINGLE TERMINAL

Stillman F. Gates
Christopher Burns

M-2934-US

APPENDIX B

```
                // Inputs:

                        sync_rst_,
                        seectlen_,
                        clkin,
                        pshreg[6],
                        pshreg[4],
                        pshreg[3],
                        pshreg[2],
                        spioreset       ) ;


sp_brdctl brdctl_1 (

        // Outputs:

                brddat7,
                brddat6,
                brddat5,
                brdstb,
                brdcs,
                brdrw,

        // Inputs:

                sync_rst_,
                brdctlen_,
                clkin,
                pshreg[8],
                pshreg[7],
                pshreg[6],
                pshreg[5],
                pshreg[4],
                pshreg[3],
                spioreset       ) ;


bobsense  bobsense_1 (  clkin,
                        sync_rst_,
                        spioreset,

                        scsi_rst_in_,
                        termsense_b_in,
                        termsense_a_in,
                        x_connsense_in,
                        connsense_in,

                        seems,
                        brdcs,
                        brdrw,

                        sense_brddat    );


iddat iddat_1 (

        // Inputs:

                clkin,
                sync_rst_,
                reset_,

                idpal_in,
                iddat_in,
```

```
                esbrd_in,

                ma_in,
                romcs_in_,
                mrw_in,

                seecs_in,

        // Outputs:

                id_done,
                iddatbyte,
                estatbyte       );


sequencer sequencer_1 (

        // Inputs:

                .clkin          (clkin),
                .sync_rst_      (sync_rst_),
                .pshreg         (pshreg),

                .id_done        (id_done),
                .ledtest        (ledtest),
                .seems          (seems),
                .brdrw          (brdrw),

        // Outputs:

                .romcs_         (romcs_),
                .romoe_         (romoe_),
                .mrw            (mrw),

                .romcsoe        (romcsoe),
                .romoeoe        (romoeoe),
                .mrwoe          (mrwoe),

                .maoe           (maoe),
                .mdoe           (mdoe),         // 7:0

                .seecsoe        (seecsoe),
                .ledoe          (ledoe),
                .stpwctloe      (stpwctloe),

                .readmuxsel     (readmuxsel),   // 2:0

                .pload_shift    (pload_shift),

                .ackdata        (ackdata),
                .ackload_       (ackload_),
                .ackshift_      (ackshift_),

                .spiooe         (spiooe),

                .addrdben_      (addrdben_),
                .addrhen_       (addrhen_),
                .addrlen_       (addrlen_),
                .addrinc        (addrinc),
                .dataen_        (dataen_),

                .spioreset      (spioreset),

                .set_parerr     (set_parerr),
                .set_unsup      (set_unsup),
```

```
                .set_mbusy      (set_mbusy),

                .ledregen_      (ledregen_),
                .stpen_         (stpen_),
                .seectlen_      (seectlen_),
                .brdctlen_      (brdctlen_),
                .statregen_     (statregen_),
                .bdprregen_     (bdprregen_)    );

endmodule
```

```
`timescale      1ns/10ps

// SPIOC_CORE.V
//
// This holds all the internal logic for SPIOC. Everything except the IO cells
// is here


module spioc_core (     clkin,
                        reset_,

                        spio,
                        spiooe,
                        spio_in,

                        led_,
                        ledoe,

                        stpwctl,
                        stpwctloe,

                        ma,
                        maoe,
                        ma_in,

                        md,
                        mdoe,
                        md_in,

                        romcs_,
                        romcsoe,
                        romcs_in_,

                        romoe_,
                        romoeoe,

                        mrw,
                        mrwoe,
                        mrw_in,

                        seecs,
                        seecsoe,
                        seecs_in,

                        idpal_in,
                        iddat_in,
                        esbrd_in,

                        scsi_rst_in_,
                        termsense_b_in,
                        termsense_a_in,
                        x_connsense_in,
                        connsense_in    );

  input         clkin;
  input         reset_;

  output        spio;
  output        spiooe;
  input         spio_in;

  output        led_;
```

```
output          ledoe;

output          stpwctl;
output          stpwctloe;

output  [17:0]  ma;
output          maoe;
input   [17:16] ma_in;

output  [7:0]   md;
output  [7:0]   mdoe;
input   [7:0]   md_in;

output          romcs_;
output          romcsoe;
input           romcs_in_;

output          romoe_;
output          romoeoe;

output          mrw;
output          mrwoe;
input           mrw_in;

output          seecs;
output          seecsoe;
input           seecs_in;

input           idpal_in;
input   [7:0]   iddat_in;
input           esbrd_in;

input           scsi_rst_in_;
input           termsense_b_in;
input           termsense_a_in;
input           x_connsense_in;
input           connsense_in;


////////////////////////////////////////////////////////////
// INTERNAL WIRES


wire    [7:0]   iddatbyte,
                estatbyte,
                readmuxout,
                rom_md;


wire [11:0] pshreg ;

wire [2:0] readmuxsel ;

wire    seecsbit, seems, stpwctloe,
        brddat7, brddat6, brddat5, brdrd7, brdrd6, brdrd5, seedibit,
        brdrw;

wire    [6:5]   sense_brddat;



reg spio ;
```

```
////////////////////////////////////////////////////////////////
//
// Generate "sync_rst_"... it can assert at any time, but
// will always deassert right AFTER a clock

reg     reset_ff_1,
        reset_ff_2;

always @(posedge clkin)

        begin
        reset_ff_1      <= #1   reset_;
        reset_ff_2      <= #1   (reset_ff_1 & reset_);
        end

wire    sync_rst_       = reset_ff_2 & reset_;


//
////////////////////////////////////////////////////////////////


assign seecs = seecsbit & seems ;


assign brdrd7 = seems ? md_in[7] : brddat7 ;
assign brdrd6 = brdrw ? sense_brddat[6] : brddat6 ;
assign brdrd5 = brdrw ? sense_brddat[5] : brddat5 ;


assign seedibit = md_in[0] & seems;


// input select logic for shift register

readmux readmux_1 (     .out    (readmuxout),

                        .in0    (md_in[7:0]),

                        .in1    ({ 2'b01, seems, 1'b1, seecsbit, seeck, seedo, seedibit }),

                        .in2    ({ brdrd7, brdrd6, brdrd5, brdstb, brdcs, brdrw, 2'b00}),

                        .in3    ({ errledenab, 4'b0, spioparerr, unsupported, membusy }),

                        .in4    (iddatbyte),

                        .in5    (estatbyte),

                        .in6    ({ 7'b0, badpar }),

                        .sel    (readmuxsel)    ) ;


  shreg12 shreg12_1 (   .pshregout      (pshreg),
                        .reset_         (sync_rst_),
                        .clkin          (clkin),
                        .spio_in        (spio_in),
                        .spiooe         (spiooe),
                        .pload_shift    (pload_shift),
                        .badpar         (badpar),
                        .pdatain        (readmuxout)    );
```

```
// input mux to reg containing acknowledge value when not shifting data to
// dagger

GTECH_MUX4 gtech_mux4_1 (       .d0     (1'b0),
                                .d1     (pshreg[0]),
                                .d2     (ackdata),
                                .d3     (spio),
                                .sel0   (ackload_),
                                .sel1   (ackshift_),
                                .out    (ackin)         );


always @ (  posedge clkin or negedge sync_rst_ )

        begin
        if (!sync_rst_) spio <= #8 1'b0 ;
        else            spio <= #8 ackin ;
        end



rom_addr rom_addr_1 (   .out            (ma),
                        .reset_         (sync_rst_),
                        .addrdben_      (addrdben_),
                        .addrhen_       (addrhen_),
                        .addrlen_       (addrlen_),
                        .addrinc        (addrinc),
                        .clkin          (clkin),
                        .in             (pshreg[9:2]),

                        .seems          (seems),
                        .brdrw          (brdrw)         ) ;


regen8 rom_data_1 (     .out            (rom_md),
                        .reset_         (sync_rst_),
                        .enable_        (dataen_),
                        .clkin          (clkin),
                        .in             (pshreg[9:2])   ) ;


// Choose EEPROM WRITE DATA or SPECIAL JUNK SIGNALS
// to be sent out to "pad_md",
// still subject to the whims of md_en_[7:0], of course

assign  md[7]   = seems ? brddat7       : rom_md[7];
assign  md[6]   = seems ? brddat6       : rom_md[6];
assign  md[5]   = seems ? brddat5       : rom_md[5];
assign  md[4]   = seems ? ~brdstb       : rom_md[4];
assign  md[3]   = seems ? ~brdcs        : rom_md[3];
assign  md[2]   = seems ? seeck         : rom_md[2];
assign  md[1]   = seems ? seedo         : rom_md[1];
assign  md[0]   =                         rom_md[0];


stp stp_1 (
                // Output:

                        .stpwctl        (stpwctl),

                // Inputs:

                        .reset_         (sync_rst_),
```

```
                    .stpwctlen_     (stpen_),
                    .clkin          (clkin),
                    .delayreg       (pshreg[8:7]),
                    .spioreset      (spioreset)      ) ;

spiostatreg spiostatreg_1 (

                // Outputs:

                        errledenab,
                        spioparerr,
                        unsupported,
                        membusy,
                        errled_,

                // Inputs:

                        clkin,
                        sync_rst_,
                        set_parerr,
                        set_unsup,
                        set_mbusy,
                        spioreset,
                        statregen_,
                        pshreg[8],
                        pshreg[3:1]     );


badparreg badparreg_1 ( clkin,
                        sync_rst_,
                        spioreset,
                        bdprregen_,
                        pshreg[1],

                        badpar  );


ledreg ledreg_1 (

                // Inputs:

                        clkin,
                        sync_rst_,
                        spioreset,
                        ledregen_,
                        pshreg[8:7],
                        errled_,
                        spio_in,

                // Outputs:

                        led_,
                        ledtest );

sp_seectl seectl_1 (

                // Outputs:

                        seems,
                        seecsbit,
                        seeck,
                        seedo,
```

```
`timescale      1ns/10ps

module readmux ( out , in0 , in1 , in2 , in3 , in4 , in5 , in6 , sel ) ;

output [7:0] out ;

input [7:0] in0 , in1 , in2 , in3 , in4 , in5 , in6 ;

input [2:0] sel ;

wire [2:0] sel ;

wire [7:0] out ,  in0 , in1 , in2 , in3 , in4 , in5 , input0 , input1 ,
        input2 , input3 , input4 , input5 , input6 , input7 ;


assign input7 = { 1'b1 , in6[7] , in5[7] , in4[7] , in3[7] , in2[7] , in1[7] ,
        in0[7] } ;

assign input6 = { 1'b1 , in6[6] , in5[6] , in4[6] , in3[6] , in2[6] , in1[6] ,
        in0[6] } ;

assign input5 = { 1'b1 , in6[5] , in5[5] , in4[5] , in3[5] , in2[5] , in1[5] ,
        in0[5] } ;

assign input4 = { 1'b1 , in6[4] , in5[4] , in4[4] , in3[4] , in2[4] , in1[4] ,
        in0[4] } ;

assign input3 = { 1'b1 , in6[3] , in5[3] , in4[3] , in3[3] , in2[3] , in1[3] ,
        in0[3] } ;

assign input2 = { 1'b1 , in6[2] , in5[2] , in4[2] , in3[2] , in2[2] , in1[2] ,
        in0[2] } ;

assign input1 = { 1'b1 , in6[1] , in5[1] , in4[1] , in3[1] , in2[1] , in1[1] ,
        in0[1] } ;

assign input0 = { 1'b1 , in6[0] , in5[0] , in4[0] , in3[0] , in2[0] , in1[0] ,
        in0[0] } ;

assign out[7] = input7[sel] ;

assign out[6] = input6[sel] ;

assign out[5] = input5[sel] ;

assign out[4] = input4[sel] ;

assign out[3] = input3[sel] ;

assign out[2] = input2[sel] ;

assign out[1] = input1[sel] ;

assign out[0] = input0[sel] ;

endmodule
```

```
input           seecsoe;

input           pad_idpal;
output          idpal_in;

input   [7:0]   pad_iddat;
output  [7:0]   iddat_in;

input           pad_esbrd;
output          esbrd_in;

input           pad_scsi_rst_;
output          scsi_rst_in_;

input           pad_termsense_b;
output          termsense_b_in;

input           pad_termsense_a;
output          termsense_a_in;

input           pad_x_connsense;
output          x_connsense_in;

input           pad_connsense;
output          connsense_in;


buf ( clkin , pad_clkin ) ;

buf ( reset_ , pad_pci_reset_ ) ;

buf ( pad_brd_reset_ , reset_ ) ;

buf ( spio_in , pad_spio ) ;
bufif1 ( pad_spio , spio , spiooe ) ;

bufif1 ( pad_led_ , led_ , ledoe ) ;

bufif1 ( pad_stpwctl , stpwctl , stpwctloe ) ;


buf ( ma_in[17], pad_hi_ma[17] );
buf ( ma_in[16], pad_hi_ma[16] );

bufif1 ( pad_hi_ma[17] , ma[17] , maoe ) ;
bufif1 ( pad_hi_ma[16] , ma[16] , maoe ) ;

bufif1 ( pad_ma[15] , ma[15] , maoe ) ;
bufif1 ( pad_ma[14] , ma[14] , maoe ) ;
bufif1 ( pad_ma[13] , ma[13] , maoe ) ;
bufif1 ( pad_ma[12] , ma[12] , maoe ) ;
bufif1 ( pad_ma[11] , ma[11] , maoe ) ;
bufif1 ( pad_ma[10] , ma[10] , maoe ) ;
bufif1 ( pad_ma[9] , ma[9] , maoe ) ;
bufif1 ( pad_ma[8] , ma[8] , maoe ) ;
bufif1 ( pad_ma[7] , ma[7] , maoe ) ;
bufif1 ( pad_ma[6] , ma[6] , maoe ) ;
bufif1 ( pad_ma[5] , ma[5] , maoe ) ;
bufif1 ( pad_ma[4] , ma[4] , maoe ) ;
bufif1 ( pad_ma[3] , ma[3] , maoe ) ;
bufif1 ( pad_ma[2] , ma[2] , maoe ) ;
bufif1 ( pad_ma[1] , ma[1] , maoe ) ;
bufif1 ( pad_ma[0] , ma[0] , maoe ) ;

buf ( md_in[7] , pad_md[7] ) ;
```

```
buf ( md_in[6] , pad_md[6] ) ;
buf ( md_in[5] , pad_md[5] ) ;
buf ( md_in[4] , pad_md[4] ) ;
buf ( md_in[3] , pad_md[3] ) ;
buf ( md_in[2] , pad_md[2] ) ;
buf ( md_in[1] , pad_md[1] ) ;
buf ( md_in[0] , pad_md[0] ) ;

bufif1 ( pad_md[7] , md[7] , mdoe[7] ) ;
bufif1 ( pad_md[6] , md[6] , mdoe[6] ) ;
bufif1 ( pad_md[5] , md[5] , mdoe[5] ) ;
bufif1 ( pad_md[4] , md[4] , mdoe[4] ) ;
bufif1 ( pad_md[3] , md[3] , mdoe[3] ) ;
bufif1 ( pad_md[2] , md[2] , mdoe[2] ) ;
bufif1 ( pad_md[1] , md[1] , mdoe[1] ) ;
bufif1 ( pad_md[0] , md[0] , mdoe[0] ) ;

buf ( romcs_in_, pad_romcs_ );
bufif1 ( pad_romcs_ , romcs_ , romcsoe ) ;

bufif1 ( pad_romoe_ , romoe_ , romoeoe ) ;

buf ( mrw_in, pad_mrw );
bufif1 ( pad_mrw , mrw , mrwoe ) ;

buf ( seecs_in, pad_seecs );
bufif1 ( pad_seecs , seecs, seecsoe ) ;


buf ( idpal_in , pad_idpal ) ;

buf ( iddat_in[7] , pad_iddat[7] ) ;
buf ( iddat_in[6] , pad_iddat[6] ) ;
buf ( iddat_in[5] , pad_iddat[5] ) ;
buf ( iddat_in[4] , pad_iddat[4] ) ;
buf ( iddat_in[3] , pad_iddat[3] ) ;
buf ( iddat_in[2] , pad_iddat[2] ) ;
buf ( iddat_in[1] , pad_iddat[1] ) ;
buf ( iddat_in[0] , pad_iddat[0] ) ;

buf ( esbrd_in , pad_esbrd ) ;

buf ( scsi_rst_in_, pad_scsi_rst_ );
buf ( termsense_b_in, pad_termsense_b );
buf ( termsense_a_in, pad_termsense_a );
buf ( x_connsense_in, pad_x_connsense );
buf ( connsense_in, pad_connsense );

endmodule
```

```
                    pad_scsi_rst_,
                    scsi_rst_in_,

                    pad_termsense_b,
                    termsense_b_in,

                    pad_termsense_a,
                    termsense_a_in,

                    pad_x_connsense,
                    x_connsense_in,

                    pad_connsense,
                    connsense_in    );


input               pad_clkin;
output              clkin;

input               pad_pci_reset_;
output              reset_;
output              pad_brd_reset_;

inout               pad_spio;
output              spio_in;
input               spio;
input               spiooe;

output              pad_led_;
input               led_;
input               ledoe;

output              pad_stpwctl;
input               stpwctl ;
input               stpwctloe;

inout    [17:16]    pad_hi_ma;
output   [15:0]     pad_ma;
output   [17:16]    ma_in;
input    [17:0]     ma;
input               maoe;

inout    [7:0]      pad_md;
output   [7:0]      md_in;
input    [7:0]      md;
input    [7:0]      mdoe;

inout               pad_romcs_;
output              romcs_in_;
input               romcs_;
input               romcsoe;

inout               pad_romoe_;
input               romoe_;
input               romoeoe;

inout               pad_mrw;
output              mrw_in;
input               mrw;
input               mrwoe;

inout               pad_seecs;
output              seecs_in;
input               seecs;
```

```
`timescale      1ns/10ps

// BEHAVIORAL version SPIOC iobuffers

module iobuffers (      pad_clkin,
                        clkin,

                        pad_pci_reset_,
                        reset_,
                        pad_brd_reset_,

                        pad_spio,
                        spio_in,
                        spio,
                        spiooe,

                        pad_led_,
                        led_,
                        ledoe,

                        pad_stpwctl,
                        stpwctl ,
                        stpwctloe,

                        pad_hi_ma,
                        pad_ma,
                        ma_in,
                        ma,
                        maoe,

                        pad_md,
                        md_in,
                        md,
                        mdoe,

                        pad_romcs_,
                        romcs_in_,
                        romcs_,
                        romcsoe,

                        pad_romoe_,
                        romoe_,
                        romoeoe,

                        pad_mrw,
                        mrw_in,
                        mrw,
                        mrwoe,

                        pad_seecs,
                        seecs_in,
                        seecs,
                        seecsoe,

                        pad_idpal,
                        idpal_in,

                        pad_iddat,
                        iddat_in,

                        pad_esbrd,
                        esbrd_in,
```

../verilog/OH3_sequencer.v 1

```
`timescale      1ns/10ps

module sequencer (

                // Inputs:

                        clkin, sync_rst_,
                        pshreq,
                        id_done, ledtest, seems, brdrw,

                // Outputs...

                        romcs_, romoe_, mrw,
                        romcsoe, romoeoe, mrwoe,
                        maoe,
                        mdoe,
                        seecsoe, ledoe, stpwctloe,
                        readmuxsel,
                        pload_shift,
                        ackdata, ackload_, ackshift_,
                        spiooe,
                        addrdben_, addrhen_, addrlen_, addrinc, dataen_,
                        spioreset,
                        set_parerr, set_unsup, set_mbusy,
                        ledregen_, stpen_, seectlen_,
                        brdctlen_, statregen_, bdprregen_       );

input           clkin, sync_rst_;
input   [11:0]  pshreq;
input           id_done, ledtest, seems, brdrw;

output          romcs_, romoe_, mrw;
output          romcsoe, romoeoe, mrwoe;
output          maoe;
output  [7:0]   mdoe;
output          seecsoe, ledoe, stpwctloe;
output  [2:0]   readmuxsel;
output          pload_shift;
output          ackdata, ackload_, ackshift_;
output          spiooe;
output          addrdben_, addrhen_, addrlen_, addrinc, dataen_;
output          spioreset;
output          set_parerr, set_unsup, set_mbusy;
output          ledregen_, stpen_, seectlen_;
output          brdctlen_, statregen_, bdprregen_;
```

```
////////////////////////////////////////////////////////////////
//
// FLIP-FLOPS

reg     [36:0]  state;

reg     [10:0]  timer;

reg             spiooe;

reg             romcs_,
                romoe_,
                mrw;

reg             stpwctloe;

reg             rom_mdoe,
                romabort,
                addrdben_;

reg     [2:0]   readmuxsel;
```

```
////////////////////////////////////////////////////////
//
// These below will not be flip-flops, but are set up
// as REGs for programming (Verilog)

reg     [36:0]  next_state;

reg             clear_timer;

reg             d_spiooe,
                ld_spiooe;

reg             d_romcs_,
                d_romoe_,
                d_mrw;

reg             d_rom_mdoe,
                d_romabort,
                ld_romabort,
                d_addrdben_;

reg     [2:0]   d_readmuxsel;
reg             ld_readmuxsel;

reg             pload_shift;

reg             ackdata,
                ackload_,
                ackshift_,

                addrhen_,
                addrlen_,
                addrinc,
                dataen_,

                set_parerr,
                set_unsup,
                set_mbusy,

                ledregen_,
                stpen_,
                seectlen_,
                brdctlen_,
                statregen_,
                bdprregen_;
```

```
////////////////////////////////////////////////////////
// Internal wires

// ... which later became regs to try and get this turkey up to speed

reg     badparity,
        leddec,
        stpdec,
        romrddec,
        romwrdec,
        seedec,
        brddec,
        spiostatdec,
        badpardec,
        spioreset;

always  @(posedge clkin)

        begin

        badparity       <= #1 ~^pshreg[11:3];

        leddec          <= #1 ( pshreg[8:3] == 6'b00001_0 );
        stpdec          <= #1 ( pshreg[8:3] == 6'b00010_0 );

        romrddec        <= #1 ( pshreg[8:3] == 6'b00011_1 );
        romwrdec        <= #1 ( pshreg[8:3] == 6'b00100_0 );
        seedec          <= #1 ( pshreg[8:4] == 5'b00110 );
        brddec          <= #1 ( pshreg[8:4] == 5'b00111 );

        spiostatdec     <= #1 ( pshreg[8:4] == 5'b10000 );
        badpardec       <= #1 ( pshreg[8:4] == 5'b10001 );

        spioreset       <= #1 ~ledtest & ~|pshreg;      // Not a single 1 in there

        end


// wire badparity = ~^pshreg[10:2];

// wire leddec          = ( pshreg[7:2] == 6'b00001_0 );
// wire stpdec          = ( pshreg[7:2] == 6'b00010_0 );

// wire romrddec        = ( pshreg[7:2] == 6'b00011_1 );
// wire romwrdec        = ( pshreg[7:2] == 6'b00100_0 );
// wire seedec          = ( pshreg[7:3] == 5'b00110 );
// wire brddec          = ( pshreg[7:3] == 5'b00111 );

// wire spiostatdec     = ( pshreg[7:3] == 5'b10000 );
// wire badpardec       = ( pshreg[7:3] == 5'b10001 );


wire    readdec         = pshreg[2];

// assign       spioreset       = ~ledtest & ~|pshreg;  // Not a single 1 in there


////////////////////////////////////////////////////////
// Output enables:


assign ledoe    = sync_rst_;
assign seecsoe  = sync_rst_;
assign romoeoe  = sync_rst_;
assign romcsoe  = sync_rst_;
assign mrwoe    = sync_rst_;
assign maoe     = sync_rst_;


assign  mdoe[7] = rom_mdoe | (seems & ~brdrw & sync_rst_);
assign  mdoe[6] = rom_mdoe | (seems & ~brdrw & sync_rst_);
assign  mdoe[5] = rom_mdoe | (seems & ~brdrw & sync_rst_);
assign  mdoe[4] = rom_mdoe | (seems & sync_rst_);
assign  mdoe[3] = rom_mdoe | (seems & sync_rst_);
assign  mdoe[2] = rom_mdoe | (seems & sync_rst_);
assign  mdoe[1] = rom_mdoe | (seems & sync_rst_);
```

../verilog/OH3_sequencer.v 5

```
assign  mdoe[0] = rom_mdoe;
```

```
////////////////////////////////////////
// STATE definitions
//
// One hot... states are IDed here by what bit is set


`define         spioc_hard_reset                        0

`define         spioc_reset_ack                         1
`define         spioc_send_id                           2
`define         spioc_send_estat                        3

`define         spioc_wait_for_start                    4
`define         spioc_shift_in_cmd                      5

`define         spioc_write_dbits                       6
`define         spioc_ack_lo_1                          7
`define         spioc_ack_lo_2                          8
`define         spioc_ack_hi                            9

`define         spioc_read_byte                         10
`define         spioc_read_byte_ack_lo_1                11
`define         spioc_read_byte_ack_lo_2                12
`define         spioc_read_byte_ack_hi                  13
`define         spioc_read_byte_shift                   14

`define         spioc_write_byte_wait_start             15
`define         spioc_write_byte_shift_data             16
`define         spioc_write_byte                        17


`define         spioc_eeread_wait_lo_addr_start         18
`define         spioc_eeread_shift_lo_addr              19
`define         spioc_eeread_wait_hi_addr_start         20
`define         spioc_eeread_shift_hi_addr              21
`define         spioc_eeread_read_0                     22
`define         spioc_eeread_inc_addr_01                23
`define         spioc_eeread_read_1                     24
`define         spioc_eeread_inc_addr_12                25
`define         spioc_eeread_read_2                     26
`define         spioc_eeread_inc_addr_23                27
`define         spioc_eeread_read_3                     28
`define         spioc_eeread_shift_3                    29

`define         spioc_eewrite_wait_lo_addr_start        30
`define         spioc_eewrite_shift_lo_addr             31
`define         spioc_eewrite_wait_hi_addr_start        32
`define         spioc_eewrite_shift_hi_addr             33
`define         spioc_eewrite_wait_data_start           34
`define         spioc_eewrite_shift_data                35
`define         spioc_eewrite_write_data                36
```

```
////////////////////////////////////////
// FLIP-FLOP switching

// There's a few flip-flops here that I actually DO want to use ASYNC RESET for...

always @ (  posedge clkin or negedge sync_rst_ )

        if      (!sync_rst_)

                begin
                spiooe          <= #1 1'b0;
                rom_mdoe        <= #1 1'b0;
                end

        else    begin
                spiooe          <= #8 ld_spiooe ? d_spiooe : spiooe;
                rom_mdoe        <= #8 d_rom_mdoe;
                end


always @ (  posedge clkin or negedge sync_rst_ )

        if      (!sync_rst_)    stpwctloe       <= #1 1'b0;
        else if (~stpen_)       stpwctloe       <= #1 1'b1;



always @ (  posedge clkin )

        if      (!sync_rst_)

                begin
                state           <= #1 37'h00_0000_0001;
                romcs_          <= #1 1'b1;
                romoe_          <= #1 1'b1;
                mrw             <= #1 1'b1;
                end

        else    begin
                state           <= #1 next_state;
                romcs_          <= #8 d_romcs_;
                romoe_          <= #8 d_romoe_;
                mrw             <= #8 d_mrw;
                end


always @ (  posedge clkin )

        if      (~sync_rst_ | clear_timer)      timer   <= #1 11'b0;
        else                                    timer   <= #1 {timer[9:0], 1'b1};


always @ (  posedge clkin )

        // These don't need a reset at all

        begin
        readmuxsel              <= #1 ld_readmuxsel ? d_readmuxsel : readmuxsel;
        romabort                <= #1 ld_romabort ? d_romabort : romabort;
        addrdben_               <= #1 d_addrdben_;
        end
```

```
////////////////////////////////////////////////////////
// NOW to set all the "REGs" that aren't flip-flops

always  @( state or timer
                or pshreg
                or romcs_ or romoe_ or mrw
                or stpwctloe
                or rom_mdoe or romabort
                or readmuxsel
                or id_done or ledtest or seems
                or leddec or stpdec or seedec or brddec
                or romrddec or romwrdec or spiostatdec or badpardec
                or readdec
                or badparity
                or spioreset    )

        begin


        // Set up "defaults"

                next_state      = 37'b0;

                clear_timer     = 1'b0;

                d_spiooe        = 1'b0;
                ld_spiooe       = 1'b0;

                d_romcs_        = 1'b1;
                d_romoe_        = 1'b1;
                d_mrw           = 1'b1;

                d_rom_mdoe      = 1'b0;

                d_romabort      = 1'b0;
                ld_romabort     = 1'b0;

                d_addrdben_     = 1'b1;

                d_readmuxsel    = 3'b000;
                ld_readmuxsel   = 1'b0;

                pload_shift     = 1'b0;

                ackload_        = 1'b1;
                ackdata         = 1'b1;
                ackshift_       = 1'b1;

                addrhen_        = 1'b1;
                addrlen_        = 1'b1;
                addrinc         = 1'b0;
                dataen_         = 1'b1;

                set_parerr      = 1'b0;
                set_unsup       = 1'b0;
                set_mbusy       = 1'b0;

                ledregen_       = 1'b1;
                stpen_          = 1'b1;
                seectlen_       = 1'b1;
                brdctlen_       = 1'b1;
                statregen_      = 1'b1;
                bdprregen_      = 1'b1;
```

```
        if ( spioreset )

                begin
                next_state[`spioc_wait_for_start]       = 1'b1;
                ld_spiooe                               = 1'b1;
                end


        else    begin

                // `spioc_hard_reset :

                        // wait here for timer[3]...

                        next_state[`spioc_hard_reset]
                                = next_state[`spioc_hard_reset]
                                        | ( state[`spioc_hard_reset] & ~timer[3] );

                        // Once timer[3] comes...

                        ackload_        = ackload_ & ~( state[`spioc_hard_reset] & timer[3] );
                        ackdata         = ackdata  & ~( state[`spioc_hard_reset] & timer[3] );

                        ld_spiooe       = ld_spiooe | ( state[`spioc_hard_reset] & timer[3] );

                        d_spiooe        = d_spiooe |  state[`spioc_hard_reset];

                        ld_readmuxsel   = ld_readmuxsel | state[`spioc_hard_reset];

                        d_readmuxsel
                                = d_readmuxsel
                                        | ( {3{state[`spioc_hard_reset]}} & 3'b100 );

                        next_state[`spioc_reset_ack]
                                = next_state[`spioc_reset_ack]
                                        | ( state[`spioc_hard_reset] & timer[3] );


                // `spioc_reset_ack :

                        // Wait here until id_done

                        next_state[`spioc_reset_ack]
                                = next_state[`spioc_reset_ack]
                                        | ( state[`spioc_reset_ack] & ~id_done );

                        // Once id_done comes...

                        ackload_        = ackload_ & ~( state[`spioc_reset_ack] & id_done );

                        pload_shift     = pload_shift | state[`spioc_reset_ack];

                        clear_timer     = clear_timer | state[`spioc_reset_ack];

                        next_state[`spioc_send_id]
                                = next_state[`spioc_send_id]
                                        | ( state[`spioc_reset_ack] & id_done );


                // `spioc_send_id :

                        ackshift_       = ackshift_ & ~state[`spioc_send_id];

                        ld_readmuxsel   = ld_readmuxsel | state[`spioc_send_id];

                        d_readmuxsel    = d_readmuxsel | ( {3{state[`spioc_send_id]}} & 3'b101 );

                        // wait here for timer[9]...

                        next_state[`spioc_send_id]
                                = next_state[`spioc_send_id]
                                        | ( state[`spioc_send_id] & ~timer[9] );

                        // take off when timer[9] comes

                        pload_shift     = pload_shift | ( state[`spioc_send_id] & timer[9] );
```

```
        clear_timer     = clear_timer | ( state['spioc_send_id] & timer[9] );

        next_state['spioc_send_estat]
                = next_state['spioc_send_estat]
                        | ( state['spioc_send_id] & timer[9] );


// 'spioc_send_estat :

        // Sit here shifting until timer[10]

        ackshift_       = ackshift_ & ~( state['spioc_send_estat] & ~timer[10] );

        next_state['spioc_send_estat]
                = next_state['spioc_send_estat]
                        | ( state['spioc_send_estat] & ~timer[10] );

        // When timer[10], bug out

        ld_spiooe       = ld_spiooe | ( state['spioc_send_estat] & timer[10] );

        next_state['spioc_wait_for_start]
                = next_state['spioc_wait_for_start]
                        | ( state['spioc_send_estat] & timer[10] );


// 'spioc_wait_for_start :

        // Sit here until see pshreg[11] = 0 and no ledtest

        next_state['spioc_wait_for_start]
                = next_state['spioc_wait_for_start]
                        | ( state['spioc_wait_for_start] & (pshreg[11] | ledtest) );

        // See pshreg (= 0), this is not a test

        clear_timer
                = clear_timer | state['spioc_wait_for_start];

        next_state['spioc_shift_in_cmd]
                = next_state['spioc_shift_in_cmd]
                        | ( state['spioc_wait_for_start] & ~pshreg[11] & ~ledtest );
```

```
// `spioc_shift_in_cmd :

        // Wait here for timer[8]

        next_state[`spioc_shift_in_cmd]
                = next_state[`spioc_shift_in_cmd]
                        | ( state[`spioc_shift_in_cmd] & ~timer[8] );

        // OK, timer[8]

        set_parerr    *
                = set_parerr
                        | ( state[`spioc_shift_in_cmd] & timer[8] & badparity );


        next_state[`spioc_wait_for_start]
                = next_state[`spioc_wait_for_start]
                        | ( state[`spioc_shift_in_cmd] & timer[8] & badparity );


        ld_readmuxsel
            = ld_readmuxsel | state[`spioc_shift_in_cmd];

        d_readmuxsel
            = d_readmuxsel
                | ( {3{(state[`spioc_shift_in_cmd] & stpdec      )}} & 3'b001 )
                | ( {3{(state[`spioc_shift_in_cmd] & seedec      )}} & 3'b001 )
                | ( {3{(state[`spioc_shift_in_cmd] & brddec      )}} & 3'b010 )
                | ( {3{(state[`spioc_shift_in_cmd] & spiostatdec )}} & 3'b011 )
                | ( {3{(state[`spioc_shift_in_cmd] & badpardec   )}} & 3'b110 );


        next_state[`spioc_write_dbits]
                = next_state[`spioc_write_dbits]
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & ( leddec | stpdec ) );

        next_state[`spioc_read_byte]
                = next_state[`spioc_read_byte]
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & readdec & ( seedec | brddec | spiostatdec | badpardec ) );

        next_state[`spioc_write_byte_wait_start]
                = next_state[`spioc_write_byte_wait_start]
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & ~readdec & ( seedec | brddec | spiostatdec | badpardec ) );


        d_addrdben_ = d_addrdben_
                        & ~( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & ( romrddec | romwrdec ) );

        set_mbusy = set_mbusy
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & seems & ( romrddec | romwrdec ) );


        ld_romabort = ld_romabort | state[`spioc_shift_in_cmd];

        d_romabort = d_romabort | seems;

        // Go throgh ROM states even if SEEMS error

        next_state[`spioc_eeread_wait_lo_addr_start]
                = next_state[`spioc_eeread_wait_lo_addr_start]
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & romrddec );

        next_state[`spioc_eewrite_wait_lo_addr_start]
                = next_state[`spioc_eewrite_wait_lo_addr_start]
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & romwrdec );

        // In case of unsupported command...

        set_unsup = set_unsup
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
```

```
                                & ~leddec
                                & ~stpdec
                                & ~seedec
                                & ~brddec
                                & ~spiostatdec
                                & ~badpardec
                                & ~romrddec
                                & ~romwrdec );

        next_state[`spioc_wait_for_start]
                = next_state[`spioc_wait_for_start]
                        | ( state[`spioc_shift_in_cmd] & timer[8] & ~badparity
                                & ~leddec
                                & ~stpdec
                                & ~seedec
                                & ~brddec
                                & ~spiostatdec
                                & ~badpardec
                                & ~romrddec
                                & ~romwrdec );
```

```
// `spioc_write_dbits :

        // I used readmuxsel[0] to remember if LED or STP was written...

        ledregen_       = ledregen_     & ~( state[`spioc_write_dbits] & ~readmuxsel[0] );
        stpen_          = stpen_        & ~( state[`spioc_write_dbits] & readmuxsel[0] );


        ackload_        = ackload_      & ~state[`spioc_write_dbits];
        ackdata         = ackdata       & ~state[`spioc_write_dbits];

        ld_spiooe       = ld_spiooe     | state[`spioc_write_dbits];
        d_spiooe        = d_spiooe      | state[`spioc_write_dbits];

        next_state[`spioc_ack_lo_1]
                = next_state[`spioc_ack_lo_1] | state[`spioc_write_dbits];

// `spioc_ack_lo_1 :

        next_state[`spioc_ack_lo_2]
                = next_state[`spioc_ack_lo_2] | state[`spioc_ack_lo_1];

// `spioc_ack_lo_2 :

        ackload_        = ackload_      & ~state[`spioc_ack_lo_2];

        next_state[`spioc_ack_hi]
                = next_state[`spioc_ack_hi] | state[`spioc_ack_lo_2];

// `spioc_ack_hi :

        ld_spiooe       = ld_spiooe     | state[`spioc_ack_hi];

        next_state[`spioc_wait_for_start]
                = next_state[`spioc_wait_for_start] | state[`spioc_ack_hi];
```

```
// `spioc_read_byte :

        ackload_        = ackload_      & ~state[`spioc_read_byte];
        ackdata         = ackdata       & ~state[`spioc_read_byte];

        ld_spiooe       = ld_spiooe     | state[`spioc_read_byte];
        d_spiooe        = d_spiooe      | state[`spioc_read_byte];

        next_state[`spioc_read_byte_ack_lo_1]
                = next_state[`spioc_read_byte_ack_lo_1] | state[`spioc_read_byte];


// `spioc_read_byte_ack_lo_1 :

        next_state[`spioc_read_byte_ack_lo_2]
                = next_state[`spioc_read_byte_ack_lo_2]
                        | state[`spioc_read_byte_ack_lo_1];


// `spioc_read_byte_ack_lo_2 :

        pload_shift     = pload_shift   | state[`spioc_read_byte_ack_lo_2];

        ackload_        = ackload_      & ~state[`spioc_read_byte_ack_lo_2];

        next_state[`spioc_read_byte_ack_hi]
                = next_state[`spioc_read_byte_ack_hi]
                        | state[`spioc_read_byte_ack_lo_2];


// `spioc_read_byte_ack_hi :

        ackshift_       = ackshift_     & ~state[`spioc_read_byte_ack_hi];

        clear_timer     = clear_timer   | state[`spioc_read_byte_ack_hi];

        next_state[`spioc_read_byte_shift]
                = next_state[`spioc_read_byte_shift]
                        | state[`spioc_read_byte_ack_hi];


// `spioc_read_byte_shift :

        // Wait, shifting, until timer[9]

        ackshift_       = ackshift_ & ~( state[`spioc_read_byte_shift] & ~timer[9] );

        next_state[`spioc_read_byte_shift]
                = next_state[`spioc_read_byte_shift]
                        | ( state[`spioc_read_byte_shift] & ~timer[9] );

        // When timer[9], move on

        ld_spiooe       = ld_spiooe | ( state[`spioc_read_byte_shift] & timer[9] );

        next_state[`spioc_wait_for_start]
                = next_state[`spioc_wait_for_start]
                        | ( state[`spioc_read_byte_shift] & timer[9] );
```

```
// `spioc_write_byte_wait_start :

        // Wait here for start bit (0) in pshreg[11]

        next_state[`spioc_write_byte_wait_start]
                = next_state[`spioc_write_byte_wait_start]
                        | ( state[`spioc_write_byte_wait_start] & pshreg[11] );

        // Take off when start bit comes

        clear_timer
                = clear_timer | state[`spioc_write_byte_wait_start];

        next_state[`spioc_write_byte_shift_data]
                = next_state[`spioc_write_byte_shift_data]
                        | ( state[`spioc_write_byte_wait_start] & ~pshreg[11] );


// `spioc_write_byte_shift_data :

        // Just wait until timer[8]

        next_state[`spioc_write_byte_shift_data]
                = next_state[`spioc_write_byte_shift_data]
                        | ( state[`spioc_write_byte_shift_data] & ~timer[8] );

        // Then branch per badparity, set_parerr if need be

        set_parerr
                = set_parerr
                        | ( state[`spioc_write_byte_shift_data] & timer[8] & badparity );

        next_state[`spioc_wait_for_start]
                = next_state[`spioc_wait_for_start]
                        | ( state[`spioc_write_byte_shift_data] & timer[8] & badparity );

        next_state[`spioc_write_byte]
                = next_state[`spioc_write_byte]
                        | ( state[`spioc_write_byte_shift_data] & timer[8] & ~badparity );


// `spioc_write_byte :

        seectlen_ = seectlen_
                        & ~( state[`spioc_write_byte] & (readmuxsel == 3'b001) );

        brdctlen_ = brdctlen_
                        & ~( state[`spioc_write_byte] & (readmuxsel == 3'b010) );

        statreqen_ = statreqen_
                        & ~( state[`spioc_write_byte] & (readmuxsel == 3'b011) );

        bdprreqen_ = bdprreqen_
                        & ~( state[`spioc_write_byte] & (readmuxsel == 3'b110) );

        ackload_        = ackload_      & ~state[`spioc_write_byte];
        ackdata         = ackdata       & ~state[`spioc_write_byte];

        d_spiooe        = d_spiooe      | state[`spioc_write_byte];
        ld_spiooe       = ld_spiooe     | state[`spioc_write_byte];

        next_state[`spioc_ack_lo_1]
                = next_state[`spioc_ack_lo_1] | state[`spioc_write_byte];
```

```
// `spioc_eeread_wait_lo_addr_start :

        // Wait for pshreg[11] to get a start bit (0)

        next_state[`spioc_eeread_wait_lo_addr_start]
                = next_state[`spioc_eeread_wait_lo_addr_start]
                        | ( state[`spioc_eeread_wait_lo_addr_start] & pshreg[11] );

        // Then roll out

        clear_timer
                = clear_timer | state[`spioc_eeread_wait_lo_addr_start];

        next_state[`spioc_eeread_shift_lo_addr]
                = next_state[`spioc_eeread_shift_lo_addr]
                        | ( state[`spioc_eeread_wait_lo_addr_start] & ~pshreg[11] );


// `spioc_eeread_shift_lo_addr :

        // Just sit here until timer[8]

        next_state[`spioc_eeread_shift_lo_addr]
                = next_state[`spioc_eeread_shift_lo_addr]
                        | ( state[`spioc_eeread_shift_lo_addr] & ~timer[8] );

        // Then hit addlen_ AND addrinc (special load and drop 1:0)
        // contemplate setting parity arror and romabort, move on

        addrlen_ = addrlen_
                        & ~( state[`spioc_eeread_shift_lo_addr] & timer[8] );

        addrinc = addrinc
                        | ( state[`spioc_eeread_shift_lo_addr] & timer[8] );

        set_parerr = set_parerr
                        | ( state[`spioc_eeread_shift_lo_addr] & timer[8] & badparity );

        ld_romabort = ld_romabort
                        | ( state[`spioc_eeread_shift_lo_addr] & timer[8] & badparity );

        d_romabort = d_romabort | state[`spioc_eeread_shift_lo_addr];

        next_state[`spioc_eeread_wait_hi_addr_start]
                = next_state[`spioc_eeread_wait_hi_addr_start]
                        | ( state[`spioc_eeread_shift_lo_addr] & timer[8] );


// `spioc_eeread_wait_hi_addr_start :

        // Wait for pshreg[11] to get a start bit (0)

        next_state[`spioc_eeread_wait_hi_addr_start]
                = next_state[`spioc_eeread_wait_hi_addr_start]
                        | ( state[`spioc_eeread_wait_hi_addr_start] & pshreg[11] );

        // Then roll out

        clear_timer
                = clear_timer | state[`spioc_eeread_wait_hi_addr_start];

        next_state[`spioc_eeread_shift_hi_addr]
                = next_state[`spioc_eeread_shift_hi_addr]
                        | ( state[`spioc_eeread_wait_hi_addr_start] & ~pshreg[11] );


// `spioc_eeread_shift_hi_addr :

        // Just sit here until timer[8]

        next_state[`spioc_eeread_shift_hi_addr]
                = next_state[`spioc_eeread_shift_hi_addr]
                        | ( state[`spioc_eeread_shift_hi_addr] & ~timer[8] );

        // Then hit addhen_, consider parity, and BRANCH per parity/romabort
```

```
                // status

                addrhen_ = addrhen_
                        & ~( state['spioc_eeread_shift_hi_addr] & timer[8] );

                set_parerr = set_parerr
                        | ( state['spioc_eeread_shift_hi_addr] & timer[8] & badparity );


                next_state['spioc_wait_for_start]
                        = next_state['spioc_wait_for_start]
                        | ( state['spioc_eeread_shift_hi_addr] & timer[8] &
                                (badparity | romabort)  );

                next_state['spioc_eeread_read_0]
                        = next_state['spioc_eeread_read_0]
                        | ( state['spioc_eeread_shift_hi_addr] & timer[8] &
                                ~badparity & ~romabort  );


                // I don't think it will hurt to clear timer w/o checking parity or romabort

                clear_timer = clear_timer
                        | ( state['spioc_eeread_shift_hi_addr] & timer[8] );


        // 'spioc_eeread_read_0 :

                // Do this stuff any time in this state

                ld_spiooe = ld_spiooe | state['spioc_eeread_read_0];
                d_spiooe = d_spiooe | state['spioc_eeread_read_0];

                ackload_ = ackload_ & ~state['spioc_eeread_read_0];

                // Do this until timer[8] gets set

                ackdata = ackdata & ~( state['spioc_eeread_read_0] & ~timer[8] );

                d_romcs_ = d_romcs_ & ~( state['spioc_eeread_read_0] & ~timer[8] );
                d_romoe_ = d_romoe_ & ~( state['spioc_eeread_read_0] & ~timer[8] );

                next_state['spioc_eeread_read_0]
                        = next_state['spioc_eeread_read_0]
                        | ( state['spioc_eeread_read_0] & ~timer[8] );

                // When timer[8] comes...

                pload_shift = pload_shift | ( state['spioc_eeread_read_0] & timer[8] );

                next_state['spioc_eeread_inc_addr_01]
                        = next_state['spioc_eeread_inc_addr_01]
                        | ( state['spioc_eeread_read_0] & timer[8] );


        // 'spioc_eeread_inc_addr_01 :

                ackshift_ = ackshift_ & ~state['spioc_eeread_inc_addr_01];

                addrinc = addrinc | state['spioc_eeread_inc_addr_01];

                clear_timer = clear_timer | state['spioc_eeread_inc_addr_01];

                next_state['spioc_eeread_read_1]
                        = next_state['spioc_eeread_read_1]
                        | state['spioc_eeread_inc_addr_01];


        // 'spioc_eeread_read_1 :

                ackshift_ = ackshift_ & ~state['spioc_eeread_read_1];

                // Do this until timer[8] gets set

                d_romcs_ = d_romcs_ & ~( state['spioc_eeread_read_1] & ~timer[8] );
                d_romoe_ = d_romoe_ & ~( state['spioc_eeread_read_1] & ~timer[8] );
```

```
                next_state[`spioc_eeread_read_1]
                        = next_state[`spioc_eeread_read_1]
                                | ( state[`spioc_eeread_read_1] & ~timer[8] );

                // When timer[8] comes...

                pload_shift = pload_shift | ( state[`spioc_eeread_read_1] & timer[8] );

                next_state[`spioc_eeread_inc_addr_12]
                        = next_state[`spioc_eeread_inc_addr_12]
                                | ( state[`spioc_eeread_read_1] & timer[8] );


        // `spioc_eeread_inc_addr_12 :

                ackshift_ = ackshift_ & ~state[`spioc_eeread_inc_addr_12];

                addrinc = addrinc | state[`spioc_eeread_inc_addr_12];

                clear_timer = clear_timer | state[`spioc_eeread_inc_addr_12];

                next_state[`spioc_eeread_read_2]
                        = next_state[`spioc_eeread_read_2]
                                | state[`spioc_eeread_inc_addr_12];


        // `spioc_eeread_read_2 :

                ackshift_ = ackshift_ & ~state[`spioc_eeread_read_2];

                // Do this until timer[8] gets set

                d_romcs_ = d_romcs_ & ~( state[`spioc_eeread_read_2] & ~timer[8] );
                d_romoe_ = d_romoe_ & ~( state[`spioc_eeread_read_2] & ~timer[8] );

                next_state[`spioc_eeread_read_2]
                        = next_state[`spioc_eeread_read_2]
                                | ( state[`spioc_eeread_read_2] & ~timer[8] );

                // When timer[8] comes...

                pload_shift = pload_shift | ( state[`spioc_eeread_read_2] & timer[8] );

                next_state[`spioc_eeread_inc_addr_23]
                        = next_state[`spioc_eeread_inc_addr_23]
                                | ( state[`spioc_eeread_read_2] & timer[8] );


        // `spioc_eeread_inc_addr_23 :

                ackshift_ = ackshift_ & ~state[`spioc_eeread_inc_addr_23];

                addrinc = addrinc | state[`spioc_eeread_inc_addr_23];

                clear_timer = clear_timer | state[`spioc_eeread_inc_addr_23];

                next_state[`spioc_eeread_read_3]
                        = next_state[`spioc_eeread_read_3]
                                | state[`spioc_eeread_inc_addr_23];


        // `spioc_eeread_read_3 :

                ackshift_ = ackshift_ & ~state[`spioc_eeread_read_3];

                // Do this until timer[8] gets set

                d_romcs_ = d_romcs_ & ~( state[`spioc_eeread_read_3] & ~timer[8] );
                d_romoe_ = d_romoe_ & ~( state[`spioc_eeread_read_3] & ~timer[8] );

                next_state[`spioc_eeread_read_3]
                        = next_state[`spioc_eeread_read_3]
                                | ( state[`spioc_eeread_read_3] & ~timer[8] );

                // When timer[8] comes...
```

```
        pload_shift = pload_shift | ( state[`spioc_eeread_read_3] & timer[8] );

        clear_timer = clear_timer | ( state[`spioc_eeread_read_3] & timer[8] );

        next_state[`spioc_eeread_shift_3]
                = next_state[`spioc_eeread_shift_3]
                        | ( state[`spioc_eeread_read_3] & timer[8] );


// `spioc_eeread_shift_3 :

        ackshift_ = ackshift_ & ~( state[`spioc_eeread_shift_3] & ~timer[10] );

        next_state[`spioc_eeread_shift_3]
                = next_state[`spioc_eeread_shift_3]
                        | ( state[`spioc_eeread_shift_3] & ~timer[10] );


        ld_spiooe = ld_spiooe | ( state[`spioc_eeread_shift_3] & timer[10] );

        next_state[`spioc_wait_for_start]
                = next_state[`spioc_wait_for_start]
                        | ( state[`spioc_eeread_shift_3] & timer[10] );
```

```
// `spioc_eewrite_wait_lo_addr_start :

        // Wait for pshreg[11] to get a start bit (0)

        next_state[`spioc_eewrite_wait_lo_addr_start]
                = next_state[`spioc_eewrite_wait_lo_addr_start]
                        | ( state[`spioc_eewrite_wait_lo_addr_start] & pshreg[11] );

        // Then roll out

        clear_timer
                = clear_timer | state[`spioc_eewrite_wait_lo_addr_start];

        next_state[`spioc_eewrite_shift_lo_addr]
                = next_state[`spioc_eewrite_shift_lo_addr]
                        | ( state[`spioc_eewrite_wait_lo_addr_start] & ~pshreg[11] );


// `spioc_eewrite_shift_lo_addr :

        // Just sit here until timer[8]

        next_state[`spioc_eewrite_shift_lo_addr]
                = next_state[`spioc_eewrite_shift_lo_addr]
                        | ( state[`spioc_eewrite_shift_lo_addr] & ~timer[8] );


        addrlen_ = addrlen_
                        & ~( state[`spioc_eewrite_shift_lo_addr] & timer[8] );


        set_parerr = set_parerr
                        | ( state[`spioc_eewrite_shift_lo_addr] & timer[8] & badparity );


        ld_romabort = ld_romabort
                        | ( state[`spioc_eewrite_shift_lo_addr] & timer[8] & badparity );

        d_romabort = d_romabort | state[`spioc_eewrite_shift_lo_addr];


        next_state[`spioc_eewrite_wait_hi_addr_start]
                = next_state[`spioc_eewrite_wait_hi_addr_start]
                        | ( state[`spioc_eewrite_shift_lo_addr] & timer[8] );


// `spioc_eewrite_wait_hi_addr_start :

        // Wait for pshreg[11] to get a start bit (0)

        next_state[`spioc_eewrite_wait_hi_addr_start]
                = next_state[`spioc_eewrite_wait_hi_addr_start]
                        | ( state[`spioc_eewrite_wait_hi_addr_start] & pshreg[11] );

        // Then roll out

        clear_timer
                = clear_timer | state[`spioc_eewrite_wait_hi_addr_start];

        next_state[`spioc_eewrite_shift_hi_addr]
                = next_state[`spioc_eewrite_shift_hi_addr]
                        | ( state[`spioc_eewrite_wait_hi_addr_start] & ~pshreg[11] );


// `spioc_eewrite_shift_hi_addr :

        // Just sit here until timer[8]

        next_state[`spioc_eewrite_shift_hi_addr]
                = next_state[`spioc_eewrite_shift_hi_addr]
                        | ( state[`spioc_eewrite_shift_hi_addr] & ~timer[8] );


        addrhen_ = addrhen_
                        & ~( state[`spioc_eewrite_shift_hi_addr] & timer[8] );
```

```
        set_parerr = set_parerr
                | ( state['spioc_eewrite_shift_hi_addr] & timer[8] & badparity );

        ld_romabort = ld_romabort
                | ( state['spioc_eewrite_shift_hi_addr] & timer[8] & badparity );

        d_romabort = d_romabort | state['spioc_eewrite_shift_hi_addr];

        next_state['spioc_eewrite_wait_data_start]
                = next_state['spioc_eewrite_wait_data_start]
                | ( state['spioc_eewrite_shift_hi_addr] & timer[8] );


// 'spioc_eewrite_wait_data_start :

        // Wait for pshreg[11] to get a start bit (0)

        next_state['spioc_eewrite_wait_data_start]
                = next_state['spioc_eewrite_wait_data_start]
                | ( state['spioc_eewrite_wait_data_start] & pshreg[11] );

        // Then roll out

        clear_timer
                = clear_timer | state['spioc_eewrite_wait_data_start];

        next_state['spioc_eewrite_shift_data]
                = next_state['spioc_eewrite_shift_data]
                | ( state['spioc_eewrite_wait_data_start] & ~pshreg[11] );

// 'spioc_eewrite_shift_data :

        // Just sit here until timer[8]

        next_state['spioc_eewrite_shift_data]
                = next_state['spioc_eewrite_shift_data]
                | ( state['spioc_eewrite_shift_data] & ~timer[8] );

        dataen_ = dataen_
                & ~( state['spioc_eewrite_shift_data] & timer[8] );

        set_parerr = set_parerr
                | ( state['spioc_eewrite_shift_data] & timer[8] & badparity );

        next_state['spioc_wait_for_start]
                = next_state['spioc_wait_for_start]
                | ( state['spioc_eewrite_shift_data] & timer[8] &
                        (badparity | romabort) );

        next_state['spioc_eewrite_write_data]
                = next_state['spioc_eewrite_write_data]
                | ( state['spioc_eewrite_shift_data] & timer[8] &
                        ~badparity & ~romabort );

        d_romcs_ = d_romcs_
                & ~( state['spioc_eewrite_shift_data] & timer[8] &
                        ~badparity & ~romabort );

        // I don't think it will hurt to clear timer w/o checking parity or romabort

        clear_timer = clear_timer
                | ( state['spioc_eewrite_shift_data] & timer[8] );


// 'spioc_eewrite_write_data :

        d_romcs_ = d_romcs_ & ~state['spioc_eewrite_write_data];
        d_rom_mdoe = d_rom_mdoe | state['spioc_eewrite_write_data];
        ld_spiooe = ld_spiooe | state['spioc_eewrite_write_data];
        d_spiooe = d_spiooe | state['spioc_eewrite_write_data];
        ackload_ = ackload_ & ~state['spioc_eewrite_write_data];

        d_mrw = d_mrw & ~( state['spioc_eewrite_write_data] & ~timer[4] );
```

```
                    ackdata = ackdata & ~( state['spioc_eewrite_write_data] & ~timer[4] );

                    next_state['spioc_eewrite_write_data]
                            = next_state['spioc_eewrite_write_data]
                                    | ( state['spioc_eewrite_write_data] & ~timer[4] );

                    next_state['spioc_ack_hi]
                            = next_state['spioc_ack_hi]
                                    | ( state['spioc_eewrite_write_data] & timer[4] );


            end     // of the enormous thing you do if SPIORESET is not asserted

        end     // of the huge "always" block


endmodule
```

../verilog/spioc.v 1

```
`timescale      1ns/10ps

// SPIOC.V
//
// This is the "top level" file for the SPIOC chip

module spioc (

                CLK,
                P_RST_,

                B_RST_,

                SPIO,

                LED_,
                STPCTL,

                MA17, MA16,
                MA15, MA14, MA13, MA12, MA11, MA10, MA9, MA8,
                MA7, MA6, MA5, MA4, MA3, MA2, MA1, MA0,

                MD7, MD6, MD5, MD4, MD3, MD2, MD1, MD0,

                MCS_,
                MOE_,
                MWR_,

                SEECS,

                IDPAL,
                ID7, ID6, ID5, ID4, ID3, ID2, ID1, ID0,
                ESBRD,

                S_RST_,
                TRMSHI,
                TRMSLO,

                XCBLDT,
                CBLDT   );


input           CLK;
input           P_RST_;

output          B_RST_;

inout           SPIO;

output          LED_;
output          STPCTL;

inout           MA17, MA16;
output          MA15, MA14, MA13, MA12, MA11, MA10, MA9, MA8;
output          MA7, MA6, MA5, MA4, MA3, MA2, MA1, MA0;

inout           MD7, MD6, MD5, MD4, MD3, MD2, MD1, MD0;

inout           MCS_;
output          MOE_;
inout           MWR_;

inout           SEECS;

input           IDPAL;
input           ID7, ID6, ID5, ID4, ID3, ID2, ID1, ID0;
input           ESBRD;

input           S_RST_;
input           TRMSHI;
input           TRMSLO;
input           XCBLDT;
input           CBLDT;
```

```
wire    [17:0]  ma_out;

wire    [17:16] ma_in;

wire    [7:0]   md_out,
                mdoe,
                md_in,
                iddat_in;


iobuffers buffers_1 (   .pad_clkin          (CLK),
                        .clkin              (clkin),

                        .pad_pci_reset_     (P_RST_),
                        .reset_             (reset_),
                        .pad_brd_reset_     (B_RST_),

                        .pad_spio           (SPIO),
                        .spio_in            (spio_in),
                        .spio               (spio_out),
                        .spiooe             (spiooe),

                        .pad_led_           (LED_),
                        .led_               (led_out_),
                        .ledoe              (ledoe),

                        .pad_stpwctl        (STPCTL),
                        .stpwctl            (stpwctl_out),
                        .stpwctloe          (stpwctloe),

                        .pad_hi_ma          ( {MA17, MA16} ),
                        .pad_ma             ( {MA15, MA14, MA13, MA12, MA11, MA10, MA9,
                                               MA8, MA7, MA6, MA5, MA4, MA3, MA2, MA1, MA0}),
                        .ma_in              (ma_in),
                        .ma                 (ma_out),
                        .maoe               (maoe),

                        .pad_md             ( {MD7, MD6, MD5, MD4, MD3, MD2, MD1, MD0} ),
                        .md_in              (md_in),
                        .md                 (md_out),
                        .mdoe               (mdoe),

                        .pad_romcs_         (MCS_),
                        .romcs_in_          (romcs_in_),
                        .romcs_             (romcs_out_),
                        .romcsoe            (romcsoe),

                        .pad_romoe_         (MOE_),
                        .romoe_             (romoe_out_),
                        .romoeoe            (romoeoe),

                        .pad_mrw            (MWR_),
                        .mrw_in             (mrw_in),
                        .mrw                (mrw_out),
                        .mrwoe              (mrwoe),

                        .pad_seecs          (SEECS),
                        .seecs_in           (seecs_in),
                        .seecs              (seecs_out),
                        .seecsoe            (seecsoe),

                        .pad_idpal          (IDPAL),
                        .idpal_in           (idpal_in),

                        .pad_iddat          ( {ID7, ID6, ID5, ID4, ID3, ID2, ID1, ID0} ),
                        .iddat_in           (iddat_in),

                        .pad_esbrd          (ESBRD),
                        .esbrd_in           (esbrd_in),

                        .pad_scsi_rst_      (S_RST_),
                        .scsi_rst_in_       (scsi_rst_in_),

                        .pad_termsense_b    (TRMSHI),
                        .termsense_b_in     (termsense_b_in),
```

../verilog/spioc.v 3

```
                    .pad_termsense_a        (TRMSLO),
                    .termsense_a_in         (termsense_a_in),

                    .pad_x_connsense        (XCBLDT),
                    .x_connsense_in         (x_connsense_in),

                    .pad_connsense          (CBLDT),
                    .connsense_in           (connsense_in)          );


spioc_core  core_1  (   .clkin              (clkin),
                        .reset_             (reset_),

                        .spio               (spio_out),
                        .spiooe             (spiooe),
                        .spio_in            (spio_in),

                        .led_               (led_out_),
                        .ledoe              (ledoe),

                        .stpwctl            (stpwctl_out),
                        .stpwctloe          (stpwctloe),

                        .ma                 (ma_out),
                        .maoe               (maoe),
                        .ma_in              (ma_in),

                        .md                 (md_out),
                        .mdoe               (mdoe),
                        .md_in              (md_in),

                        .romcs_             (romcs_out_),
                        .romcsoe            (romcsoe),
                        .romcs_in_          (romcs_in_),

                        .romoe_             (romoe_out_),
                        .romoeoe            (romoeoe),

                        .mrw                (mrw_out),
                        .mrwoe              (mrwoe),
                        .mrw_in             (mrw_in),

                        .seecs              (seecs_out),
                        .seecsoe            (seecsoe),
                        .seecs_in           (seecs_in),

                        .idpal_in           (idpal_in),
                        .iddat_in           (iddat_in),
                        .esbrd_in           (esbrd_in),

                        .scsi_rst_in_       (scsi_rst_in_),
                        .termsense_b_in     (termsense_b_in),
                        .termsense_a_in     (termsense_a_in),
                        .x_connsense_in     (x_connsense_in),
                        .connsense_in       (connsense_in)          );

endmodule
```

```
`timescale      1ns/10ps

module GTECH_MUX4 (  d0 , d1 , d2 , d3 , sel0 , sel1 , out ) ;

input d0 , d1 , d2 , d3 , sel0 , sel1 ;

output out ;

reg out ;

wire [3:0] data ;

wire [1:0] sel ;

assign data = {d3 , d2 , d1 , d0 } ;

assign sel = { sel1 , sel0 } ;

always @ ( data or sel )
begin
        out <= data[sel] ;
end

endmodule
```

```
`timescale      1ns/10ps

// Module to load, hold, increment ROM ADDRESS bits
//
// Here's a special trick: If BOTH addlen_ and addrinc are asserted,
// do a special load where the 2 low bits get jammed to zero
//
// Here's another... if SEEMS is asserted, send out "BRDRW" on MA[15]

module rom_addr (       out,
                        reset_,
                        addrdben_,
                        addrhen_,
                        addrlen_,
                        addrinc,
                        clkin,
                        in,

                        seems,
                        brdrw   );


input reset_, addrdben_, addrhen_, addrlen_, addrinc, clkin ;

input [7:0] in ;

input   seems,
        brdrw;

output  [17:0]   out ;


reg [17:0] ad_reg ;


assign  out[17:16] = ad_reg[17:16];
assign  out[15]    = seems ? brdrw : ad_reg[15];
assign  out[14:0]  = ad_reg[14:0];


always @ ( posedge clkin or negedge reset_ )

        if      (!reset_)       ad_reg  <= #1 18'h0AAAA;

        else    begin

                if      ( ~addrdben_ )

                        ad_reg[17:16]   <= #1 in[6:5];

                if      ( ~addrhen_ )

                        ad_reg[15:8]    <= #1 in;


                if      ( ~addrlen_ &  ~addrinc )

                        ad_reg[7:0]     <= #1 in;

                else if ( ~addrlen_ &  addrinc )        // both asserted... trick load
```

```
                begin
                ad_reg[7:2]     <= #1 in[7:2];
                ad_reg[1:0]     <= #1 2'b00;
                end

        else if ( addrlen_ &  addrinc )         // just increment

                ad_reg[1:0]     <= #1 (ad_reg[1:0] + 2'b01);

        end

endmodule
```

```
`timescale      1ns/10ps

// 8 bit register with enable
module regen8 ( out , reset_ , enable_ , clkin , in ) ;

input reset_ , enable_ , clkin ;

input [7:0] in ;

output [7:0] out ;

reg [7:0] out ;

wire [7:0] in ;

always @ (  posedge clkin )
begin
        if (!reset_)
                out <= #1 {8{1'b0}} ;
        else
                out <= #1  enable_ ? out : in ;
end


endmodule
```

```
`timescale      1ns/10ps

module stp ( stpwctl , reset_ , stpwctlen_ , clkin , delayreg ,
        spioreset ) ;

output stpwctl ;

input reset_ , stpwctlen_ , clkin , spioreset ;

input [1:0] delayreg ;

wire [1:0] delayreg ;

reg stpwen , stpwlevel ;

GTECH_MUX4 gtech_mux4_1 (       .d0     (delayreg[0]),
                                .d1     (stpwen),
                                .d2     (1'b0),
                                .d3     (1'b0),
                                .sel0   (stpwctlen_),
                                .sel1   (spioreset),
                                .out    (muxout)        );

always @ (  posedge clkin )

        if      (!reset_)

                begin
                stpwen          <= #1 1'b0 ;
                stpwlevel       <= #1 1'b0 ;
                end

        else    begin
                stpwen          <= #1 muxout ;
                stpwlevel       <= #1 stpwctlen_ ? stpwlevel : delayreg[1] ;
                end

assign stpwctl = stpwlevel ^ stpwen ;

endmodule
```

```
`timescale      1ns/10ps

module spiostatreg (    errledenab,
                        spioparerr,
                        unsupported,
                        membusy,
                        errled_,

                        clkin,
                        reset_,
                        set_parerr,
                        set_unsup,
                        set_mbusy,
                        spioreset,
                        ld_en_,
                        ld_data7,
                        ld_data210      );

input           clkin,
                reset_;

input           set_parerr,
                set_unsup,
                set_mbusy,
                spioreset,
                ld_en_,
                ld_data7;

input   [2:0]   ld_data210;


output  errledenab,
        spioparerr,
        unsupported,
        membusy,
        errled_;

reg     errledenab,
        spioparerr,
        unsupported,
        membusy;

assign  errled_ = ~( errledenab & (spioparerr | unsupported | membusy) );

always @ ( posedge clkin )

        if      ( !reset_ )                     errledenab      <= #1 1'b0;
        else if ( spioreset )                   errledenab      <= #1 1'b0;
        else if ( ~ld_en_ )                     errledenab      <= #1 ld_data7;


always @ ( posedge clkin )

        if      ( !reset_ )                     spioparerr      <= #1 1'b0;
        else if ( spioreset )                   spioparerr      <= #1 1'b0;
        else if ( set_parerr )                  spioparerr      <= #1 1'b1;
        else if ( ~ld_en_ & ld_data210[2] )     spioparerr      <= #1 1'b0;   // A ONE writes it off


always @ ( posedge clkin )

        if      ( !reset_ )                     unsupported     <= #1 1'b0;
        else if ( spioreset )                   unsupported     <= #1 1'b0;
        else if ( set_unsup )                   unsupported     <= #1 1'b1;
        else if ( ~ld_en_ & ld_data210[1] )     unsupported     <= #1 1'b0;   // A ONE writes it off


always @ ( posedge clkin )

        if      ( !reset_ )                     membusy <= #1 1'b0;
        else if ( spioreset )                   membusy <= #1 1'b0;
        else if ( set_mbusy )                   membusy <= #1 1'b1;
        else if ( ~ld_en_ & ld_data210[0] )     membusy <= #1 1'b0;   // A ONE writes it off
```

```
endmodule
```

```
`timescale      1ns/10ps

module badparreg (      clkin,
                        reset_,
                        spioreset,
                        lden_,
                        d_in,

                        badpar  );

input           clkin,
                reset_,
                spioreset,
                lden_,
                d_in;

output          badpar;


reg     badpar;


always @ ( posedge clkin )

        begin
        if      (!reset_)       badpar  <= #1 1'b0;
        else if (spioreset)     badpar  <= #1 1'b0;
        else if (!lden_)        badpar  <= #1 d_in;
        end

endmodule
```

```
`timescale      1ns/10ps

module ledreg ( clkin,
                reset_,
                spioreset,
                ledregen_,
                delayreg,
                errled_,
                spio_in,

                led_,
                ledtest );


input           clkin,
                reset_,
                spioreset;

input           ledregen_;
input   [1:0]   delayreg;

input           errled_;

input           spio_in;

output          led_,
                ledtest;


reg ledbit_, ledtest;


always @ (  posedge clkin )

        if      (!reset_)         ledtest <= #10 1'b0 ;
        else if (!ledregen_ )     ledtest <= #10 delayreg[1] ;


always @ (  posedge clkin )

        if      (!reset_)       ledbit_ <= #1 1'b0 ;
        else if (spioreset)     ledbit_ <= #1 1'b0 ;
        else if (!ledregen_)    ledbit_ <= #1 delayreg[0] ;


assign led_ = ledtest ?  spio_in : ( ledbit_ & errled_ );

endmodule
```

```
`timescale      1ns/10ps

module sp_seectl ( seems , seecs , seeck , seedo , reset_ , seectlen_ ,
        clkin , delayreg5 , delayreg3 , delayreg2 , delayreg1 , spioreset ) ;

output seems , seecs , seeck , seedo  ;

input reset_ , seectlen_ , clkin , delayreg5 , delayreg3 , delayreg2 ,
        delayreg1 , spioreset  ;


reg seems , seecs, seeck , seedo ;


always @ (  posedge clkin )

        if      ( !reset_ )

                begin
                seems   <= #1 1'b0;
                seecs   <= #1 1'b0;
                seeck   <= #1 1'b0;
                seedo   <= #1 1'b0;
                end

        else if ( spioreset )

                begin
                seems   <= #1 1'b0;
                seecs   <= #1 1'b0;
                seeck   <= #1 1'b0;
                seedo   <= #1 1'b0;
                end

        else if ( !seectlen_ )

                begin
                seems   <= #1 delayreg5;
                seecs   <= #1 delayreg3;
                seeck   <= #1 delayreg2;
                seedo   <= #1 delayreg1;
                end


endmodule
```

```
`timescale      1ns/10ps

module sp_brdctl ( brddat7 , brddat6 , brddat5 , brdstb , brdcs , brdrw ,
        reset_ , brdctlen_ , clkin , delayreg7 , delayreg6 , delayreg5 ,
        delayreg4 , delayreg3 , delayreg2 , spioreset ) ;

output brddat7 , brddat6 , brddat5 , brdstb , brdcs , brdrw ;

input reset_ , brdctlen_ , clkin , delayreg7 , delayreg6 , delayreg5 ,
        delayreg4 , delayreg3 , delayreg2 , spioreset  ;

reg brddat7 , brddat6 , brddat5 , brdstb , brdcs , brdrw ;

GTECH_MUX4 gtech_mux4_1 ( delayreg7 , brddat7 , 1'b0, 1'b0, brdctlen_ ,
        spioreset , muxout7 ) ;

always @ (  posedge clkin )
begin
        if (!reset_)
                brddat7 <= #1 1'b0 ;
        else
                brddat7 <= #1 muxout7;
end

GTECH_MUX4 gtech_mux4_2 ( delayreg6 , brddat6 , 1'b0, 1'b0, brdctlen_ ,
        spioreset , muxout6 ) ;

always @ (  posedge clkin )
begin
        if (!reset_)
                brddat6 <= #1 1'b0 ;
        else
                brddat6 <= #1 muxout6;
end

GTECH_MUX4 gtech_mux4_3 ( delayreg5 , brddat5 , 1'b0, 1'b0, brdctlen_ ,
        spioreset , muxout5 ) ;

always @ (  posedge clkin )
begin
        if (!reset_)
                brddat5 <= #1 1'b0 ;
        else
                brddat5 <= #1 muxout5;
end

GTECH_MUX4 gtech_mux4_4 ( delayreg4 , brdstb , 1'b0, 1'b0, brdctlen_ ,
        spioreset , muxout4 ) ;

always @ (  posedge clkin )
begin
        if (!reset_)
                brdstb <= #1 1'b0 ;
        else
                brdstb <= #1 muxout4;
end

GTECH_MUX4 gtech_mux4_5 ( delayreg3 , brdcs , 1'b0, 1'b0, brdctlen_ ,
        spioreset , muxout3 ) ;

always @ (  posedge clkin )
begin
        if (!reset_)
                brdcs <= #1 1'b0 ;
```

```
        else
                brdcs <= #1 muxout3;
end


GTECH_MUX4 gtech_mux4_6 ( delayreg2 , brdrw , 1'b0, 1'b0, brdctlen_ ,
        spioreset , muxout2 ) ;

always @ (  posedge clkin )
begin
        if (!reset_)
                brdrw <= #1 1'b0 ;
        else
                brdrw <= #1 muxout2;
end


endmodule
```

shreg12.v 1

```
`timescale      1ns/10ps

// 12 bit parallel loadable, parallel out shift register
// shifts from 11 to 0

// LATER added some new tricks:
//
// "pload_shift" will load DATA into [8:1], START BIT into [0],
// then on the next clock (as bits [8:0] start shifting out) PARITY
// (generated here) will be loaded into [8] and 111 into [11:9]. This will
// be equivalent to loading {111, PARITY, DATA, START BIT} all at once
// (but easier to time out)


module shreg12  (       pshregout ,
                        reset_ ,
                        clkin ,
                        spio_in ,
                        spiooe ,
                        pload_shift ,
                        badpar ,
                        pdatain ) ;

output  [11:0]  pshregout ;

input           reset_ ,
                clkin ,
                spio_in ,
                spiooe ,
                pload_shift ,
                badpar ;

input   [7:0]   pdatain ;


reg     after_pload;

always  @(posedge clkin)

        after_pload     <= #1 (pload_shift & reset_);


wire    parity = ~badpar ^ (^pshregout[8:1]);


reg     [11:0]  pshregout;

always @ (  posedge clkin )

    if      (!reset_)       pshregout <= #1 12'b111111111111 ;
    else if (pload_shift)   pshregout <= #1 {3'b111, pdatain, 1'b0};                // load
    else if (after_pload)   pshregout <= #1 {3'b111, parity, pshregout[8:1]};       // fix parity
    else                    pshregout <= #1 {(spio_in | spiooe), pshregout[11:1]}; // shift


endmodule
```

```
`timescale      1ns/10ps

// BOBSENSE.V
//
// Senses termination (for "auto-term") and connectors installed

module  bobsense  (     clkin,
                        sync_rst_,
                        spioreset,

                        scsi_rst_,
                        termsense_b,
                        termsense_a,
                        x_connsense,
                        connsense,

                        seems,
                        brdcs,
                        brdrw,

                        sense_brddat    );

input           clkin,
                sync_rst_,
                spioreset,

                scsi_rst_,
                termsense_b,
                termsense_a,
                x_connsense,
                connsense,

                seems,
                brdcs,
                brdrw;

output  [1:0]   sense_brddat;

reg     scsi_sync_1,
        scsi_sync_2,

        term_b_ff_1,
        term_a_ff_1,

        term_b_ff_2,
        term_a_ff_2,

        x_conn_ff_1,
        x_conn_ff_2,

        conn_ff_1,
        conn_ff_2;

reg     [4:0]   sample_timer;

reg     [1:0]   state;

reg     [1:0]   next_state,
                sense_brddat;

wire    sample_pulse    = (&sample_timer);      // All ones

always  @(posedge clkin)

        begin
        scsi_sync_1     <= #1 scsi_rst_;
        scsi_sync_2     <= #1 scsi_sync_1;

        state           <= #1 next_state;
```

```
        x_conn_ff_1     <= #1 x_connsense;
        x_conn_ff_2     <= #1 x_conn_ff_1;

        conn_ff_1       <= #1 connsense;
        conn_ff_2       <= #1 conn_ff_1;
        end


always  @(posedge clkin)

        if      ( ~sync_rst_ | spioreset | scsi_sync_2 )

                sample_timer    <= #1 5'b0;

        else    sample_timer    <= #1 (sample_timer + 1'b1);


always  @(posedge clkin)

        if      ( ~sync_rst_ | spioreset )

                begin
                term_b_ff_1     <= #1 1'b1;
                term_a_ff_1     <= #1 1'b0;
                term_b_ff_2     <= #1 1'b1;
                term_a_ff_2     <= #1 1'b0;
                end

        else if ( sample_pulse )

                begin
                term_b_ff_1     <= #1 termsense_b;
                term_a_ff_1     <= #1 termsense_a;
                term_b_ff_2     <= #1 term_b_ff_1;
                term_a_ff_2     <= #1 term_a_ff_1;
                end


// STATE DEFINITION (comment)
//
// 00 - means the next BRDLOGIC read is going to read the termstat
// 01 - come here while the first read (termstat) is underway
// 11 - come here after the first read completes, from then on read connsense


// next_state and sense_brddat are programming regs, NOT flip-flops...

always  @( sync_rst_ or spioreset
                or scsi_sync_2
                or state
                or term_b_ff_2 or term_a_ff_2
                or x_conn_ff_2 or conn_ff_2
                or seems or brdcs or brdrw
         )

        begin

        if      ( state == 2'b00 )

                begin
                sense_brddat    = { term_b_ff_2, term_a_ff_2 };

                if      ( ~sync_rst_ | spioreset | ~scsi_sync_2 )       next_state = 2'b00;
                else if ( seems & brdcs & brdrw )                       next_state = 2'b01;
                else                                                    next_state = 2'b00;
                end

        else if ( state == 2'b01 )

                begin
                sense_brddat    = { term_b_ff_2, term_a_ff_2 };

                if      ( ~sync_rst_ | spioreset | ~scsi_sync_2 )       next_state = 2'b00;
                else if ( seems & brdcs & brdrw )                       next_state = 2'b01;
                else                                                    next_state = 2'b11;
```

```
                end

        else    // ( state probably == 2'b11 )

                begin
                sense_brddat    = { x_conn_ff_2, conn_ff_2 };

                if      ( ~sync_rst_ | spioreset | ~scsi_sync_2 )       next_state = 2'b00;
                else                                                    next_state = 2'b11;
                end

        end


endmodule
```

../verilog/iddat.v 1

```
`timescale      1ns/10ps


// IDDAT.v
//
// This is based on a shift register and a timer.
//
// The shift register resets to all ones, then shifts in from the "PAL" pin.
// The shift register stops shifting when the first zero gets to bit [0].
//
// The timer resets to F then counts down to 0. If the timer reaches zero and NO zeroes
// have been shifted in from the PAL yet, then JAM a parallel thing into the shift register.
//
// What to jam depends on the level of idpal AT RESET TIME (as remembered by idpal_at_rst)
// If zero, use the chip built-in value, if one use the "parallel pin" values.
// There's a special bandaid to make sure the shift register doesn't load zeroes at all
// if idpal is low all the time.
//

module iddat (  clkin,
                sync_rst_,
                reset_,

                idpal_in,
                iddat_in,
                esbrd_in,

                ma_in,
                romcs_in_,
                mrw_in,

                seecs_in,

                id_done,
                iddatbyte,
                estatbyte       ) ;


input           clkin,
                sync_rst_,
                reset_;

input           idpal_in;
input   [7:0]   iddat_in;
input           esbrd_in;

input   [17:16] ma_in;

input           romcs_in_,
                mrw_in;

input           seecs_in;


output          id_done;

output  [7:0]   iddatbyte,
                estatbyte;


reg     [16:0]  shifter;

reg     [3:0]   timer;

reg     [1:0]   front_end;


reg     [17:16] ma_at_rst;

reg             idpal_at_rst,
                romcs_at_rst,
                mrw_at_rst,
                seecs_at_rst;
```

```
wire    shifter_all_1   = &shifter;

wire    timer_0         = ~|timer;

wire    shift_in        = front_end[0] | ~idpal_at_rst; // Force to 1 if idpal_at_rst == 0

wire    [1:0]   rom_size = ~romcs_at_rst ? 2'b11                                    // NO ROM
                                         : ~ma_at_rst[16]       ? 2'b10             // ma[16] low -> 64K
                                         : ~ma_at_rst[17]       ? 2'b01             // ma[17] low -> 128K
                                                                : 2'b00;            // 256K


// OK, if there is no PAL will be "jamming" in either default or from pins

wire    [16:0]  jam_in = ~idpal_at_rst  ? 17'b1110_0000_0101_0101_0     // Internal default

                                        : { 1'b1,               // ESTAT[7]
                                            rom_size[1],        // [6]
                                            1'b1,               // [5]
                                            esbrd_in,           // [4]
                                            seecs_at_rst,       // [3]
                                            ~mrw_at_rst,        // [2]
                                            rom_size[0],        // [1]
                                            1'b0,               // [0]

                                            iddat_in,           // ID[7:0]

                                            1'b0    };          // "stop shifting" bit


always  @(posedge clkin)

        front_end       <= #1 { idpal_in, front_end[1] };       // Free-running, no reset


always  @(posedge clkin)

        if      ( ~sync_rst_ )                  shifter <= #1 17'b11111111_11111111_1;
        else if ( timer_0 & shifter_all_1 )     shifter <= #1 jam_in;
        else if ( shifter[0] )                  shifter <= #1 { shift_in, shifter[16:1] };


always  @(posedge clkin)

        if      ( ~sync_rst_ )  timer   <= #1 4'b1111;
        else if ( ~timer_0 )    timer   <= #1 (timer - 4'b0001);


always  @(posedge reset_)       // NON-CLOCK-TRIGGERED FLIP-FLOPS

        begin
        idpal_at_rst    <= #1 idpal_in;
        ma_at_rst       <= #1 ma_in;
        romcs_at_rst    <= #1 romcs_in_;
        mrw_at_rst      <= #1 mrw_in;
        seecs_at_rst    <= #1 seecs_in;
        end


assign  id_done = ~shifter[0];

assign  iddatbyte = shifter[8:1];

assign  estatbyte = shifter[16:9];


endmodule
```

APPENDIX C

```
/*****************************************************
*************************

Function:

Read/write  decoder

*****************************************************
************************/
module sprwdec ( csadr_, csren_, cdadr_, cdwen_,
crbusy, por, softcmden, rd1b,
     rd1d, rd1e, rdbrdctl_, rdseectl_, rdspiodat_,
spiobsy, spiobsy_,
     wrbrdctl_, wrspiocap_, wrseectl_, wrspioctl_,
wrspiodat_ );

input     [7:0] csadr_, cdadr_;
input     csren_, cdwen_, crbusy, por, softcmden;

output    rd1b;               // reading address 1b
output    rd1d;               // reading address 1d
output    rd1e;               // reading address 1e
output    rdbrdctl_;          // read BRDCTL (1D)
output    rdseectl_;          // read SEECTL (1E)
output    rdspiodat_;         // read SOFTDAT (1E)
output    spiobsy;       // CIO read
output    spiobsy_;      // CIO read
output    wrbrdctl_;          // write BRDCTL (1D)
output    wrspiocap_;         // write ESTAT (1B)
output    wrseectl_;          // write SEECTL (1E)
output    wrspioctl_;         // write SPIOCTL (1E)
output    wrspiodat_;         // write SPIODAT (1D)

//-----------------------------------------------------
------------------------
```

```
// csadr_ decode

// complement signals
nbuf02 iv0 ( .NQ(softcmden_), .A(softcmden) );
inv01 iv1 ( .NQ(csadr4), .A(csadr_[4]) );
inv01 iv2 ( .NQ(csadr3), .A(csadr_[3]) );
inv01 iv3 ( .NQ(csadr2), .A(csadr_[2]) );
inv01 iv4 ( .NQ(csadr1), .A(csadr_[1]) );
inv01 iv5 ( .NQ(csadr0), .A(csadr_[0]) );
inv01 iv6 ( .NQ(csren), .A(csren_) );

// block decode: 00011
and05 an0 ( .Q(blkrd), .A(csadr_[7]), .B(csadr_[6]),
.C(csadr_[5]),
     .D(csadr4), .E(csadr3) );

// register decode: 011, 101, 110
nand03 an1 ( .NQ(nrd1b), .A(csadr_[2]), .B(csadr1   ),
.C(csadr0   ) );
nand03 an2 ( .NQ(nrd1d), .A(csadr2   ), .B(csadr_[1]),
.C(csadr0   ) );
nand03 an3 ( .NQ(nrd1e), .A(csadr2   ), .B(csadr1   ),
.C(csadr_[0]) );
nand03 an4 ( .NQ(dec1b1d1e), .A(nrd1b), .B(nrd1d),
.C(nrd1e) );

nbuf02 nb0 ( .NQ(rd1b), .A(nrd1b) );
nbuf02 nb1 ( .NQ(rd1d), .A(nrd1d) );
nbuf02 nb2 ( .NQ(rd1e), .A(nrd1e) );

// set and reset busy
nand03 an5 ( .NQ(adrok_), .A(blkrd), .B(dec1b1d1e),
.C(csren) );
or02   or0 ( .Q(setrbsy_), .A(adrok_), .B(crbusy) );
nor02  nr0 ( .NQ(rstrbsy_), .A(csren_), .B(por) );
```

```
    nrslt nl0 ( .NQ(bsy_), .Q(bsy), .NS(setrbsy_),
    .NR(rstrbsy_) );
    buf0104 bf0 ( .Q(spiobsy), .A(bsy) );
    buf0103 bf1 ( .Q(spiobsy_), .A(bsy_) );
5
    // read strobes
    nand03 nd10 ( .NQ(rdbrd_), .A(rd1d), .B(spiobsy),
    .C(softcmden_) );
    nand03 nd11 ( .NQ(rdsee_), .A(rd1e), .B(spiobsy),
10  .C(softcmden_) );
    nand03 nd12 ( .NQ(rdspiodat_), .A(rd1d), .B(spiobsy),
    .C(softcmden) );

    buf0102 iv30 ( .Q(rdbrdctl_), .A(rdbrd_) );
15  buf0102 iv31 ( .Q(rdseectl_), .A(rdsee_) );

    //-------------------------------------------------
    -------------------------
    // write decode
20
    inv01  iv11 ( .NQ(cdadr4), .A(cdadr_[4]) );
    inv01  iv12 ( .NQ(cdadr3), .A(cdadr_[3]) );
    inv01  iv13 ( .NQ(cdadr2), .A(cdadr_[2]) );
    inv01  iv14 ( .NQ(cdadr1), .A(cdadr_[1]) );
25  inv01  iv15 ( .NQ(cdadr0), .A(cdadr_[0]) );
    nbuf02 iv16 ( .NQ(cdwen), .A(cdwen_) );

    // block decode: 00011
    and05 an10 ( .Q(blkwr), .A(cdadr_[7]), .B(cdadr_[6]),
30  .C(cdadr_[5]),
        .D(cdadr4), .E(cdadr3) );

    // register decode: 011, 101, 110
    and03 an11 ( .Q(wr1b), .A(cdadr_[2]), .B(cdadr1   ),
35  .C(cdadr0   ) );
```

```
and03 an12 ( .Q(wr1d), .A(cdadr2   ), .B(cdadr_[1]),
.C(cdadr0   ) );
and03 an13 ( .Q(wr1e), .A(cdadr2   ), .B(cdadr1   ),
.C(cdadr_[0]) );

and02 an14 ( .Q(wspiocap), .A(blkwr), .B(wr1b) );
and03 an15 ( .Q(wbrdctl),  .A(blkwr), .B(wr1d),
.C(softcmden_) );
and03 an16 ( .Q(wseectl),  .A(blkwr), .B(wr1e),
.C(softcmden_) );
and03 an17 ( .Q(wspiodat), .A(blkwr), .B(wr1d),
.C(softcmden) );
and03 an18 ( .Q(wspioctl), .A(blkwr), .B(wr1e),
.C(softcmden) );

and02 an19 ( .Q(wrspiocap), .A(cdwen), .B(wspiocap) );
and02 an1a ( .Q(wrbrdctl), .A(cdwen), .B(wbrdctl) );
and02 an1b ( .Q(wrseectl), .A(cdwen), .B(wseectl) );
and02 an1c ( .Q(wrspiodat), .A(cdwen), .B(wspiodat) );
and02 an1d ( .Q(wrspioctl), .A(cdwen), .B(wspioctl) );

nbuf02 nb19 ( .NQ(wrspiocap_), .A(wrspiocap) );
nbuf02 nb1a ( .NQ(wrbrdctl_), .A(wrbrdctl) );
nbuf02 nb1b ( .NQ(wrseectl_), .A(wrseectl) );
nbuf02 nb1c ( .NQ(wrspiodat_), .A(wrspiodat) );
nbuf02 nb1d ( .NQ(wrspioctl_), .A(wrspioctl) );

endmodule
```



Claims:-

1. A method for serially transferring information on a serial port input-output line between a first integrated circuit and a second integrated circuit comprising:

transmitting an initialization signal on said serial port input-output line from said second integrated circuit in response to a power-on reset signal; and

driving a signal on said serial port input-output line by said second integrated circuit, subsequent to said initialization signal, only in response to a signal from said first integrated circuit on said serial port input-output line.

2. The method of Claim 1 wherein said second integrated circuit drives an acknowledge signal on said serial port input-output line within a predetermined time of receiving said signal from said first integrated circuit.

3. The method of Claim 2 further comprising waiting by said first integrated circuit for completion of said acknowledge signal from said second integrated circuit before driving another signal on said serial port input-output line by said first integrated circuit.