ATB0 Engineering Document - Controller

SCALE Group
MIT Computer Science and Artificial Intelligence Laboratory

Contents

1 Introduction **5**

2 User Interface **6**
2.1 Accessing SDRAM 6
2.2 Configuration 8
2.2.1 Setting the voltage 8
2.2.2 Measuring the current 10
2.2.3 Calibrating current measurement 10
2.2.4 Setting the generated clock speed 11
2.2.5 Other configuration registers 11
2.3 Communication with the daughtercard 12
2.3.1 Accessing the user pins 12
2.3.2 Asynchronous Host Interface Port (AHIP) 13
2.3.2.1 Asynchronous Transaction Protocol 13
2.3.2.2 Writing and reading with AHIP 14
2.3.2.3 8 bit reads and writes 14
2.3.2.4 AHIP Self-Test 15

3 Implementation details **16**
3.1 Clocking 16
3.1.1 Syncing with the slow clock 17
3.2 Helper modules 17
3.2.1 Shift registers 17
3.2.2 Counter 18
3.3 Controller module 18
3.4 Decode module 19
3.4.1 PLX Interface 19
3.4.2 Decode implementation 20
3.5 SDRAM control module 24
3.5.1 Refresh Timer 25
3.5.2 SDRAM control module implementation. 25
3.6 Voltage Set module 30
3.7 Voltage measure module 33
3.8 Current measure module 35
3.9 Clock module 39
3.10 User pin control module 41
3.11 AHIP module 42
3.12 LGALED module 46

A Pinout listings **48**

List of Figures

1 Block diagram of ATB0. 6
2 ATB0 Memory Map. 7
3 SDRAM memory layout. 7
4 Format of VMR*mn*. 10
5 Calibrating the current. 10
6 Clock register format. 11
7 Status register format. 12
8 AHIP Header format. 14
9 Timing diagram of an AHIP read. 15
10 Timing diagram of an AHIP write. 15
11 Timing diagram of an 8 bit AHIP write and read. 15
12 Shift registers 17
13 Block diagram of the controller module. 18
14 Dataflow through the controller. 19
15 Timing diagram of general PLX access. 19
16 Block diagram of the decode module. 20
17 Timing diagram of a write for decode module. 23
18 Timing diagram of a read for decode module. 24
19 Refresh timer for the SDRAM control module. 25
20 Block diagram of the SDRAM control module. 26
21 Timing diagram of SDRAM initialization. 26
22 Timing diagram of a SDRAM write. 29
23 Timing diagram of a SDRAM read. 30
24 Block diagram of the voltage set module. 32
25 Timing diagram for the voltage set module. 33
26 Block diagram of the voltage measure module. 34
27 Timing diagram for the voltage measure module. 35
28 Block diagram of the current measure module. 36
29 Timing diagram of the current measure module writing to SDRAM memory. 38
30 Block diagram of the clock module. 39
31 Timing diagram for the clock module. 40
32 Block diagram of the User pin control module. 41
33 Block diagram of the AHIP module. 43

List of Tables

1 ATB0 Control Register . . . 9
2 Frequency to M and N values conversion chart. . . . 11
3 Bit assignment of LGA_LED register. . . . 12
4 Meaning of direction bits in the user pin registers. . . . 13
5 AHIP modes. . . . 13
6 AHIP opcodes. . . . 14
7 Decode module wire descriptions. . . . 21
8 Decode module state definitions. . . . 22
9 Module numbers. . . . 22
10 SDRAM control module state definitions. . . . 27
11 SDRAM control module state transitions. . . . 27
12 SDRAM control module wire descriptions. . . . 28
13 Voltage set module state definitions. . . . 30
14 Voltage set module wire descriptions. . . . 31
15 Voltage measure module state definitions. . . . 33
16 Voltage measure module wire descriptions. . . . 34
17 Current measure module state definitions. . . . 35
18 Current measure module wire descriptions. . . . 37
19 Clock module state definitions . . . 39
20 Clock module wire descriptions. . . . 40
21 User pin control module wire descriptions. . . . 42
22 AHIP module wire descriptions. . . . 44
23 AHIP module state definitions. . . . 45
24 AHIP module state transitions. . . . 45

1 Introduction

This document is one in a set of three engineering documents describing the Assam Tester Baseboard Revision 0 (ATB0); this document describes the controller while the other two describe the actual hardware [1] and the software interface [2].

ATB0 is designed to provide a testbed for custom designed circuit boards that require multiple differing power supplies (referred to in this document as the "daughtercard"). ATB0 also provides a communication channel between a host PC and the daughtercard. To do so, ATB0 has the following devices:

Sixteen power supplies. Each power supply can be configured to supply a voltage independently of each other; there are fourteen power supplies that supply a positive voltage between 0 and 4 volts and two that supply a negative voltage between 0 and -4 volts. The voltage of each of the positive power supplies can be read back to ensure proper operation; the current drawn from each positive power supply can also be read for power measurements.

Onboard SDRAM. The onboard SDRAM is 12 bits wide and 64 MWs deep. Its primary purpose to provide temporary storage for power measurements allowing measurements to be performed quickly while testing and read back later, after testing is complete.

Frequency synthesizer. The frequency synthesizer is used to provide a clock to the daugthercard and can be configured to run between 25 MHz and 400 Mhz.

60 I/O pins between ATB0 and the daughtercard. 34 of these pins are used to perform reads and writes on the daughtercard using the AHIP protocol [3]. The remaining 26 are used as individual user defined I/O pins.

Two LEDs. These can be used to provide status flags to the user.

The controller for ATB0 is written in Verilog for the Xilinx on ATB0. Its purpose is two-fold: 1) control the various devices on ATB0, and 2) facilitate communication between the host PC and the daughtercard. All communication between the host PC and the controller is in the form of reads and writes to memory locations. A PLX interface card is used to convey a read or write from the host PC to the controller.

Standard operation involves the user connecting a daughtercard to ATB0 then writing to memory-mapped registers to configure the devices on ATB0 according to the requirements of the daughtercard. The user then performs some task by communicating with the daughtercard, either directly, by reading from or writing to the user pins, or using the AHIP interface. Tasks could include operations such as such as running an application on a synthesized CPU, performing memory operation on a memory controller, etc. While the task is being performed, certain events, as configured by the user, cause the controller to measure the current drawn from each of the power supplies and store those measurements in the onboard SDRAM chips. The user can later read those values from the SDRAM and analyze the power consumption over time of the task.

Figure 1 shows a block diagram of the system. A memory access operation from the host PC is conveyed by the PLX interface card to ATB0. A decode module in the controller receives the operation and decodes it. The decode module then drives the address and data onto the central buses and enables the appropriate module according to the nature of the operation. The voltage set and voltage measure modules communicate directly with the power supplies. The current measure module communicates with both the power supplies and the SDRAM control module to enable it to store current measurements into the onboard

memory. The SDRAM module can receive memory accesses from both the decode module and the current measure module and performs the access to the SDRAM. The AHIP module communicates with the daughtercard through dedicated AHIP pins and the user pin control module allows the user to set and read individual pins connected to the daughtercard. The LED control module drives the LEDs directly and the Clock set module communicates with the frequency synthesizer to provide the clock to the daughtercard.

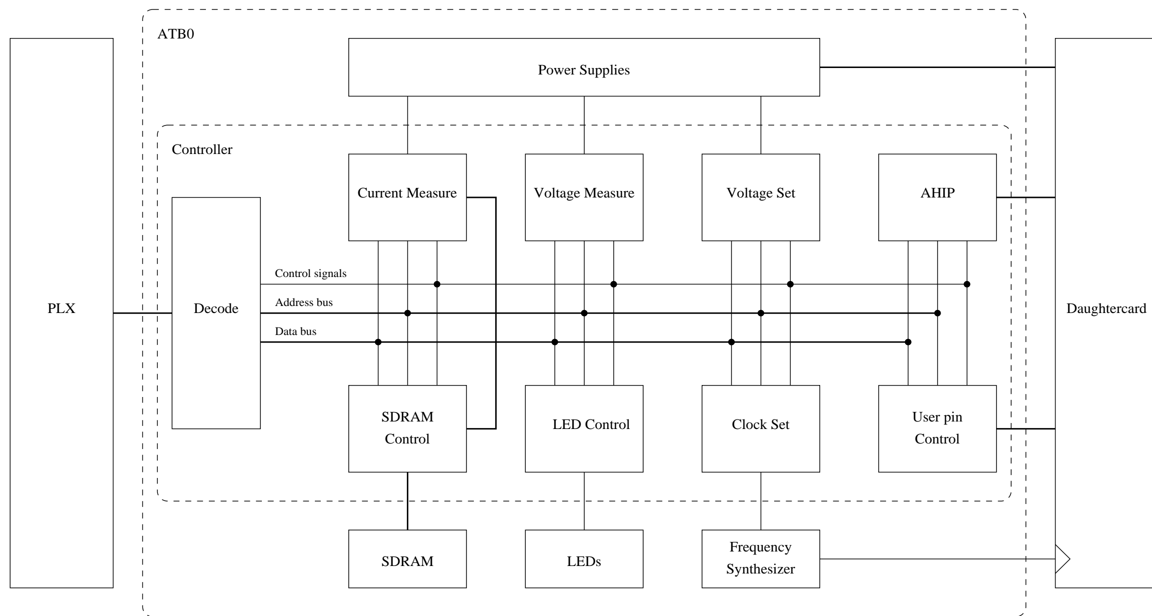

Figure 1: Block diagram of ATB0.

2 User Interface

The user interacts with the controller by reading and writing to memory locations on the host PC. The PLX then forwards these requests to the baseboard and relays the response back to the user. The PLX allows addresses up to 28 bits wide and enforces word addressing by forcing the lower two bits to zero. The controller currently uses 27 of those bits, leaving the top half of possible memory space open for expansion. As shown in Figure 2, memory is divided into three main sections, SDRAM Memory, Control Registers, and AHIP Daughtercard Memory space.

Accessing memory in SDRAM is described in Section 2.1. Configuration of the devices using the control registers is described in Section 2.2. Finally, communicating with the daughtercard is described in Section 2.3.

2.1 Accessing SDRAM

Each word in SDRAM memory space corresponds to two 12-bit values in the actual SDRAM; Figure 3 shows the resulting memory layout, relating SDRAM memory locations to addresses in SDRAM memory space. Since there are 2^{24} locations in SDRAM, 2^{23} four byte words, or 32 MBs, are needed to access all the SDRAM memory, thus the SDRAM memory space in the controller is 32 MBs. To access a word in SDRAM memory space, simply read from or write to the corresponding address in the ATB0 memory space.

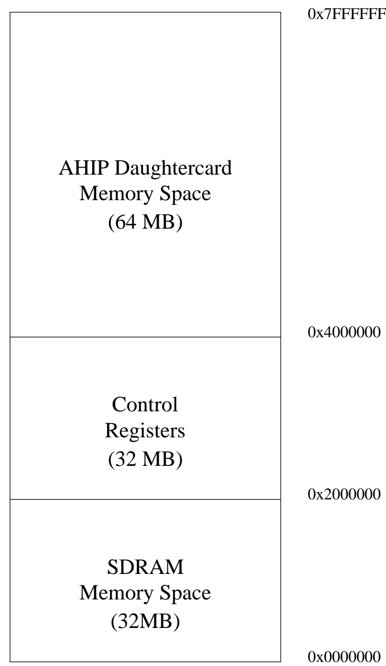

Figure 2: ATB0 Memory Map. Address are relative to the bottom of ATB0 address space.

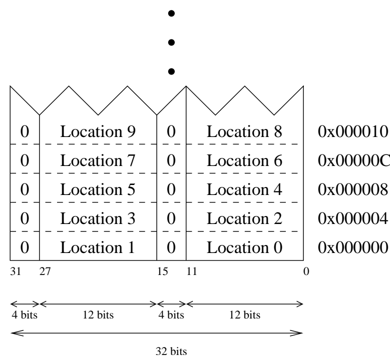

Figure 3: SDRAM memory layout. Addresses are relative to the bottom of SDRAM Memory space.

2.2 Configuration

Configuring ATB0 is accomplished by writing to a number of memory-mapped control registers. The address and functionality of each register is summarized in Table 1. Setting the voltage of the Power Supplies is described in Section 2.2.1. Measuring the current is described in Section 2.2.2. Calibrating the current measurements is described in Section 2.2.3. Setting the clock speed that the frequency synthesizer generates is described in Section 2.2.4. Finally, Section 2.2.5 show how to set refresh rate of the onboard SDRAM chips and the Logic Analyzer and LED outputs.

2.2.1 Setting the voltage

ATB0 is equipped with 16 user-controlled independent power supplies. The desired voltage of power supplies 0 - 13* can be set between 0V and 4.095V†; the actual voltage supplied and the current drawn from the power supply can be measured and read by the user. The desired voltage of power supplies 14 and 15 can be set between 0V and -4.095V but the actual voltage and current can not be measured. Note that the current that each supply is able to produce varies, see the hardware document [1] for more information.

Setting the desired voltage is accomplished by writing the appropriate register (VSR0 - VSR15) with the desired value. The registers are 12 bits wide (thus, only the lower 12 bits of the 32 bits written to the register are used) and their value corresponds linearly to the range of 0V to 4.095V. To convert from a desired voltage into a value to put in the register, divide the desired voltage by the maximum voltage (V_{REF}) and multiply by the maximum value 2^{12}-1 = 4095 as shown in Equation 1.

$$VSRn = \frac{volts}{V_{REF}} * (2^{12} - 1) \approx \frac{volts}{4.095} * 4095 = volts * 1000 \tag{1}$$

This results in a coding that is about $1\frac{mV}{bit}$ since V_{REF} is about 4.095V and thus the voltage set register can be set to approximately 1000 times the desired voltage; however, if precision is necessary, the user should measure V_{REF} and use the actual value in the calculation. Writing to VSR0 - VSR14 will only set the register, it will not cause the power supply to output the desired voltage. Writing to VSR15 both sets VSR15 and causes all 16 power supplies to output the voltage that is contained in their corresponding register.

Once VSR15 is written to and the power supplies are set to the desired values, the actual voltage produced by each power supply can be obtained by simply reading the appropriate voltage measure register (VMR). This returns a 12-bit value that corresponds linearly to the range of 0V to 5V, which is the reference voltage supplied to the ADC used to measure the voltage. Thus, to obtain a voltage from the value read from the VMR, divide by the maximum value (2^{12}-1 = 4095) and multiply by the ADC reference voltage (V_{REF_ADC}) as shown in Equation 2. V_{REF_ADC} is approximately 5V, but like V_{REF}, the user should measure this if precision is required.

$$Volts = \frac{VMR}{2^{12} - 1} * V_{REF_ADC} \approx \frac{VMR}{4095} * 5 \tag{2}$$

*Power supplies are numbered 0 - 15 to make it coincide more nicely with the addressing scheme; however, on the schematics, power supplies are numbered 1 - 16, this numbering scheme is not used anywhere but the schematic, don't let that confuse you.

†"4.095V [is] the theoretical maximum voltage. The actual voltage is limited by the dropout voltage of the regulators, with 3.9V the expected limit."[4] See the hardware document[1] for more information.

Register	Address	Description	Read	Write
VS*n*	0x21000*n*0	Voltage Set Registers - The desired voltage for power supply *n*. Writing to VSR15 commits the voltages to the power supplies.	√	√
VM*n*	0x22000*n*0	Voltage Measure Registers - The actual voltage of Power Supply *n*. A measurement is made each time one of these registers is read.	√	
CM_BURST	0x2300000	Current Measure Burst - Reading this register causes the current being drawn from each power supply specified in the CM_MASK register to be measured and placed in SDRAM memory. The address in SDRAM memory space of the first measurement is returned. Writing sets the address in SDRAM memory space where the next burst will be placed.	√	√
CM_MASK	0x2300004	Current Measure Mask - Each of the lower 14 bits in this register correspondes to a power supply (bit 0 correspondes to Power Supply 0). When CM_BURST is read, each power supply whose bit in this register is 1 has its current measured. (Currently not supported)	√	√
CM*mn*	0x2301*mn*0	Current Measure Registers - The current being drawn from Power Supplies *m* and *n*. A measurement is made each time one of these registers is read.	√	
CLOCK	0x2400000	Clock - Used to set the frequency generated by the onboard frequency synthesizer.	√	√
SDRAM_RT	0x2500000	SDRAM Refresh Timer - The number of clock ticks between each refresh of the SDRAM modules.	√	√
USER_ALL	0x2600000	All user pins - The logical values of all 26 user pins.	√	√
USER_DIR	0x2600004	User pins direction - Whether all 26 user pins are set to be input or output. A high bit indicates the controller is driving that pin (an output).	√	√
USER*p*	0x2601*pp*0	User Pins - Reading returns the logical value of user pin *p* and I/O direction. Writing a 0 or 1 sets the pin as output with the given value and writing a value of 2 resets the pin to be an input.	√	√
LGA_LED	0x2000000	LGA and LED outputs.	√	√
AHIP_MODE	0x2700000	AHIP Mode. Used to allow for test and 8 bit modes.	√	√
STATUS	0x2800000	Status flags	√	

Table 1: ATB0 Control Registers. Addresses are relative to the bottom of the ATB0 address space.

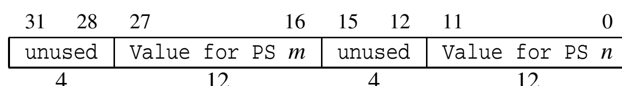

Figure 4: Format of VMR*mn*.

2.2.2 Measuring the current

The current drawn from a power supply can be measured either individually or as a burst with other power supplies. The current from two different power supply can be measured individually by reading from the appropriate Current Measure Register (CM*mn*). The current drawn from power supply *m* is returned in the top 16 bits, and the current drawn from power supply *n* is returns in the lower 16 bits as shown in Figure 4. The value returned for each supply is described in Section 2.2.3. To measure multiple power supplies in a single burst, first set the CM_MASK register so that each power supply to be measured has a 1 in its bit within the mask. The CM_BURST register can optionally be written to to specify where in SDRAM memory space to place the measurements. Reading the CM_BURST register actually performs the measurements and returns the address in SDRAM memory space at which the first measurement was placed.

2.2.3 Calibrating current measurement

Because of hardware issues (see [1]), the current measured by the power supplies is not exactly the current that is being drawn from the power supply by the daughtercard; therefore, it is necessary to adjust the measurements recieved. To calibrate a power supply to determine how to adjust the measurements, load the power supply with two different loads and measure, with lab equipment, the actual current through each load, as well as the value returned by the controller by reading the Current Measure register. These two measurements define two points in a plane with the current on the y-axis and the return value on the x-axis. These two points define a line in that plane. That line defines the relationship between the value returned by the controller as the current measurement and the actual current being drawn by the daughtercard for a single power supply. Figure 5 shows this plane.

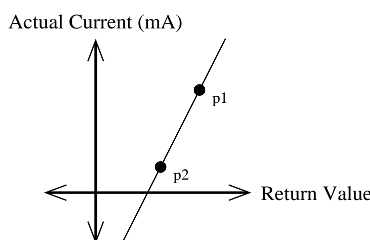

Figure 5: Calibrating the current. The y-axis is the measured current, the x-axis is the value returned by the controller as the current for that supply. $p1$ and $p2$ represent values obtained by placing different loads across the power supply. The line relates the value returned by measuring the current to the actual current drawn by the daughtercard.

Given the above calibration, basic geometry gives us an equation to convert a value returned by the controller to the actual current begin drawn by the daughter card. Equation 3 gives the slope of the line, Equation 4 gives the intercept. Using the slope and the intercept, Equation 5 is the equation needed. More

31 14	13 11	10 9	8 0
unused	test	N	M
18	3	2	9

Figure 6: Clock register format.

Frequency (MHz)	N	M
25 - 50	3	$8 * frequency$
50 - 100	2	$4 * frequency$
100 - 200	1	$2 * frequency$
200 - 400	0	$frequency$

Table 2: Frequency to M and N values conversion chart.

representative names, ratio and offset, can be used for the slope and intercept respectively.

$$slope = \frac{(Measured_1 - Measured_2)}{CMR_1 - CMR_2} \tag{3}$$

$$intercept = Measured_2 - (slope * CMR_2) \tag{4}$$

$$current = (slope * CMR) + intercept = (ratio * CMR) + offset \tag{5}$$

2.2.4 Setting the generated clock speed

ATB0 has a frequency synthesizer that is capable of creating clock signals in the range of 25 to 400 MHz; the frequency can be set up by writing to the CLOCK register whose format is shown in Figure 6. The used 14 bits of the register are split into three fields: `test`, `N`, and `M`. The `test` field is used to choose what test mode to put the frequency synthesizer in and should be set to 0 for normal operation. Table 2 provides values for `N` and `M` to achieve a desired frequency. Note that `M` should always be between 200 and 400. See the frequency synthesizer's datasheet [5] for more information on what these numbers mean and what test modes are available.

2.2.5 Other configuration registers

The STATUS register provides status for each of the modules within the controller and can be used for debugging purposes. As shown in Figure 7, bits 15 to 8 contain the da (data available) signal for each module and bits 7 to 0 contain the done signal for each module. The bits are ordered according to the module numbers in Table 9 (i.e. module 2's done signal is bit 2 and its da signal is bit 10). See the implementation section (Section 3) for more information on module numbers and what the da and done signals mean.

Using other configuration registers is mostly a matter of simply writing a value to the register. Each of the registers can also be read if necessary.

SDRAM Refresh Timer The SDRAM Refresh Timer register (SDRAM_RT) contains the number of clock ticks between each refresh of the SDRAM modules and is independent of anything else. To change this timer simply write a new value to the register, and to check the currently value simply read it. The

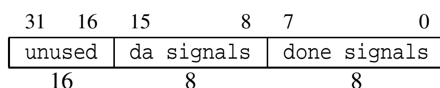

Figure 7: Status register format.

Bit #	Signal
0	LED0
1	LED1
2	LGA0
3	LGA1
4 - 31	unused

Table 3: Bit assignment of LGA_LED register.

SDRAM documentation [6] states that the modules should be refreshed every 15.625μs; the default value of 78 (0x4E) causes a refresh every 15.6μs with a 5 MHz clock. If a 40 MHz clock is used, this value should be set to 625 (0x271) to achieve a refresh every 15.625μs.

LGA and LED outputs The two LGA outputs go to a jumper on the back side of the baseboard such that they can be used as inputs to a logic analyzer. The LED outputs go to the two LEDs on the baseboard. All four of these outputs can be set using the LGA_LED register with the bit assignments shown in Table 3.

2.3 Communication with the daughtercard

The controller allows the user to either manually communicate with the daughtercard using the user pins directly (Section 2.3.1), or read and write to the AHIP Daughtercard Memory area and have the controller take care of forwarding the data to the daughtercard, which it does using AHIP (Section 2.3.2).

2.3.1 Accessing the user pins

User pins can be accessed directly, either individually or collectively in parallel. The USER*p* registers allow access to an individual pin *p*. When a USER*p* register is read, bit 0 contains the logical value currently on the pin, whether it is being driven by the controller or the daughtercard, and bit 1 contains the I/O direction of the pin, a 1 indicates the pin is an output pin and being driven by the controller, a 0 indicates the pin is an input and being driven by the daughtercard (or floating) (as shown in Table 4). When a value of 0 or 1 is written to a USER*p* register, the controller will drive the pin at that value; if a value of 2 is written to a USER*p* register, the controller will stop driving the pin with any value, making it an input pin, available for the daughtercard to drive.

The USER_ALL and USER_DIR registers allow access to all 26 pins at once. Reading USER_DIR returns wether each pin is set as an output or input, each pin's direction is in 1 bit (i.e. pin 0's direction is in bit 0). Writing to USER_DIR sets the direction of all 26 pins according to the corresponding bit in the value written (i.e. if bit 5 of the value written is 1, pin 5 will be set to be an output). Reading the USER_ALL register returns the value of all 26 pins in the low 26 bits of the result. Writing to USER_ALL sets all *output*

Bit value	Meaning
0	Pin is an input and being driven by the daughtercard (or floating).
1	Pin is an output and being drive by the controller.

Table 4: Meaning of direction bits in the user pin registers.

Mode	Number	Description
Normal	0	All reads and writes are normal 32 bit reads and writes.
Test	1	All writes are test writes, reads to address 0x0 are test address reads, reads to any other address are test data reads.
8-bit	2	All reads and writes are normal 8 bit reads and writes.
8-bit test	3	Like Test mode, but all 8 bit reads and writes.

Table 5: AHIP modes.

pins to the corresponding bit in the value written. Pins that are set to be input pins will not be affected by a write to USER_ALL.

2.3.2 Asynchronous Host Interface Port (AHIP)

AHIP is a data communication protocol that facilitates communication between two devices that do not share a common clock. The controller uses AHIP to allow the user to access memory space on the daughtercard directly. To read from or write to a memory location on the daughtercard, the user need only read from or write to the AHIP Daughtercard Memory space on ATB0. The controller performs the host side of AHIP and handles the actual transfer of data to and from the daughtercard, which acts as the slave.

AHIP can be used in one of four modes by setting the AHIP_MODE register to a mode number shown in Table 5. When in normal mode, all reads and writes to AHIP Daughtercard memory space become normal AHIP reads and writes using the standard 32-bit AHIP protocol. When in test mode a write causes a test write, a read from daughtercard address zero causes a test address read and a read from any other daughtercard address causes a test data read, see Section 2.3.2.4 for more information on test mode. When in 8 bit mode, normal reads and writes are performed, but the 8-bit AHIP protocol (Section 2.3.2.3, which uses only the bottom 8 bits of the bus, is used. Finally, in 8 bit test mode, test reads and writes are performed the same as in test mode, but the 8-bit AHIP protocol is used.

The AHIP protocol is described below to facilitate creating a client for the daughtercard that can communicate with ATB0. An example client is included in the source directory (in the ahip_client subdirectory) which is written to be connected to a block memory module on a Virtex II FPGA to create a simple memory system that can be written to and read from using AHIP. ahip_client.v contains the ahip client module for normal 32 bit operation, and ahip8_client.v contains an ahip client that uses an 8 bit bus. Another example can be found in ATC0 [3] which implements the client side of AHIP.

2.3.2.1 Asynchronous Transaction Protocol The protocol uses a 32-bit wide bi-directional bus and two handshake signals, `req`, controlled by the host, and `ack`, controlled by the slave, to transfer data to and from the daughtercard, with the controller acting as the host and the daughtercard as the slave. When idle, both `req` and `ack` are low and the host is responsible for driving the bus so that it does not float. The host starts a transaction by raising the `req` signal and placing a header on the bus in the format shown in Figure 8. The

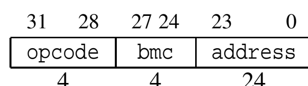

Figure 8: AHIP Header format.

Opcode	Function
0000	Normal Write
0001	Normal Read
1000	Test Write
1001	Test Read (Data)
1101	Test Read (Address)

Table 6: AHIP opcodes.

header contains three fields: a 4-bit opcode field; a 4-bit *burst-mode counter* (bmc) for burst-mode read and write; and a 24-bit address. The burst-mode read/write is not currently implemented by the controller and reserved for future implementations, so the bmc field is set to zero and should be ignored. Table 6 shows the different opcodes available. During normal operation, only the normal read and write are used. Some opcodes are used to perform a self-test of AHIP, as described in Section 2.3.2.4. A variation of the standard protocol that uses an 8-bit data bus is also supported and described in Section 2.3.2.3.

2.3.2.2 Writing and reading with AHIP The host starts a write by placing the header on the bus and raising the req signal. After the slave observes that req is high, it reads the header word and raises the ack signal. After the host receives ack, it places the write data on the bus and lowers req. The slave reads the bus value and then lowers ack. The host will then free the data bus and both the host and the slave return to the idle state. Figure 10 shows the timing diagram of the transaction.

Similar to word write, the host starts a read by placing the header on the bus and raising the req signal. The slave will obtain the header and asserts ack. Once the host sees the ack, it frees the data bus and lowers req. Once the slave is ready with the data, it places the data on the bus and lowers ack. The host reads the data off the bus and raises req again, the slave then frees the data bus and raises ack. The host lowers req, followed by the slave lowering ack, and both return to an idle state. Figure 9 shows the timing diagram of the word read.

2.3.2.3 8 bit reads and writes When used in an 8 bit mode, reads and writes still transfer 32 bits, but they do so 8 bits at a time. The protocol is similar, the host begins a transaction by placing the bottom 8 bits of the header (header[7:0]) onto the bus and raising the req signal. The slave observes that req is high, reads the data from the bus and raises the ack signal. The host places the next 8 bits of the header onto the bus and lowers req, which the slave reads and lowers ack. This process continues until all 4 bytes of the header have been transmitted, one byte per edge. When the slave acknowledges receiving the last byte of the header, if the access is a write, the host continues to send the data word, 1 byte at a time, starting from the bottom 8 bits (data[7:0]), in the same manner until all four bytes have been sent. If the access is a read, the host frees the bus and raises req to signal the slave can begin transmitting the read data back. The slave then transmits the read data back to the host, one byte at a time, in similar manner. When the host raises req to acknowledge receipt of the last byte, the slave raises ack one more time to indicate it it no longer

driving the bus and the protocol finishes just as it did with the 32 bit read. The timing diagram for both an 8 bit write and read is shown in Figure 11, the write is on top.

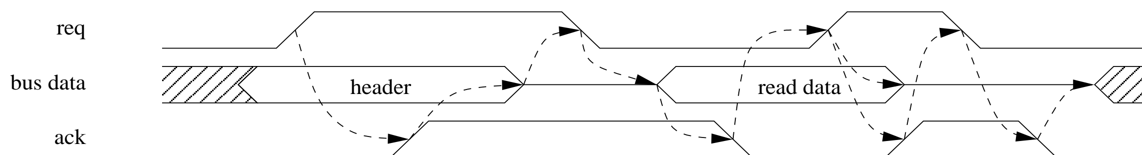

Figure 9: Timing diagram of an AHIP read.

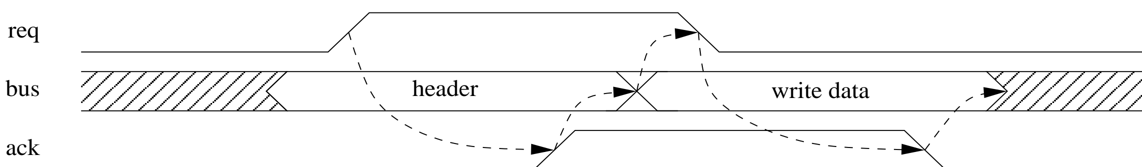

Figure 10: Timing diagram of an AHIP write.

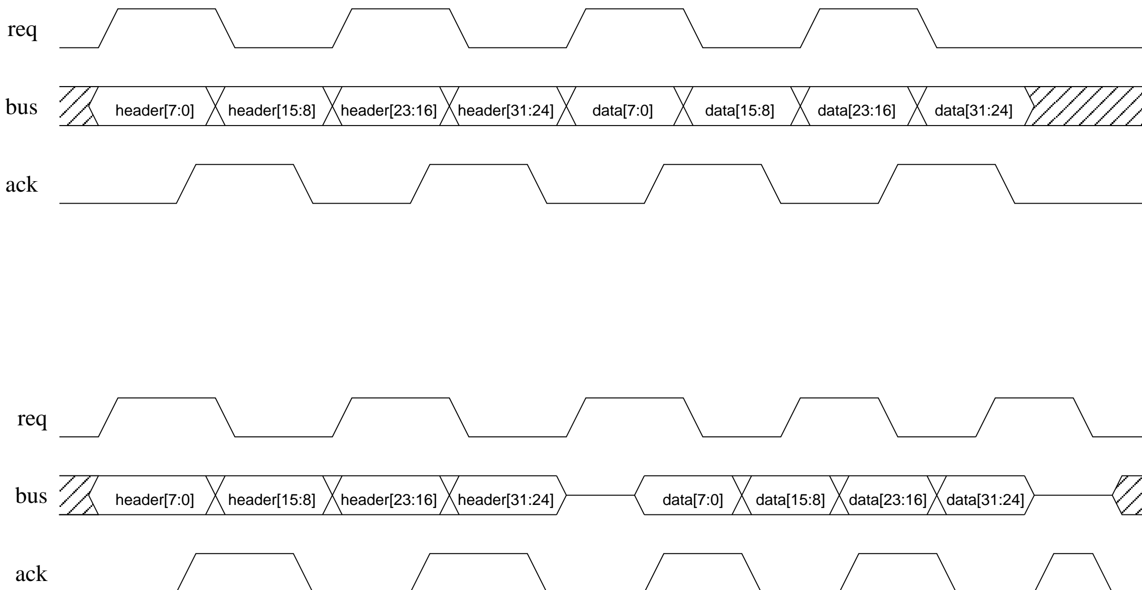

Figure 11: Timing diagram of an 8 bit AHIP write and read. The write is on top and the read is on the bottom.

2.3.2.4 AHIP Self-Test To verify the correctness of AHIP functionality, both in the controller and the implementation on the daughtercard, the top bit of the opcode is used to run AHIP in test mode. The daughtercard must implement AHIP test mode for the test modes to be useful. When in test mode, a slave device will store the address and the value written during a Test Write. A Test Read will then return either the stored address or the stored data. The host can thus verify that both the data and address are being

transmitted correctly by performing a Test Write then a Test Read on both the data and the address and checking that those values correspond to the data and address just written.

3 Implementation details

The ATB0 controller is written in Verilog and consists of the following interconnected modules.

Controller The top level module. It instantiates all other modules, connects them together, divides the clock to provide a slower clock when needed, collects output from all the modules and pipelines it off the chip, and defines the external interface.

Decode Responsible for decoding requests received from the host computer via the PLX interface and forwarding the request on to the correct module depending on the address.

Voltage set Controls the DACs which set the desired voltage on the power supplies.

Voltage measure Controls the ADCs which measure the actual voltage on the power supplies.

Current measure Controls the ADCs which measure the current drawn from the power supplies.

SDRAM control Provides an interface to the on board SDRAM.

Clock Interfaces with the frequency synthesizer.

User pin Sets and reads the user pins.

AHIP Performs the host side of AHIP to perform memory transactions with the daughtercard on behalf of the user.

LGALED A small module which holds the LGA_LED register.

Figure 1 in the Introduction (Section 1) shows a block diagram of how these module interconnect with each other and the rest of the system. In some instances, more than one of the modules requires the same functionality, such as shifting the bits of a register onto a serial data line; in these cases, a seperate "helper" module is defined and instantiated in each module that requires that functionality. The helper modules are:

shiftreg_out Shifts the bits of a register onto a serial data line.

shiftreg_in Shifts bits into a register from a serial data line.

counter A simple counter with an enable signal.

3.1 Clocking

One of the signals sent from the PLX daughtercard is a clock that is used as a global clock for the entire controller; it is called CXCOE in the schematics (Chip Transmission Line Clock and Output Enable). A slightly skewed second clock, called HCLK (Host Clock) in the schematics, is also sent but never used in the current implementaiton of the controller. This global clock is laballed "clk" in the block diagrams in this section.

Multiple modules (voltage set, voltage measure, current measure, and clock) interact with on board components that can not run at the frequency of the global clock, thus a clock that is eight times slower is generated for these components. It is labelled "sclk" in the block diagrams.

3.1.1 Syncing with the slow clock

The FSM's of the modules that interact with the on board components that require a slower clock must run at the speed of the global clock for communication with the host PC to work. It is therefore necessary to get in sync with the slow clock before communicating with the on board component. To avoid repetition, the process is described here.

When leaving the idle state, the FSM checks to see if the slow clock is low or high, if it is high it enters the SYNC1 state and waits for it to go low. Once the slow clock is low, it enters SYNC2 and waits for it to go high. It then moves onto the next state at the beginning of the slow clock's cycle. A timing diagram of this can be seen in Figure 25 which shows the timing for the voltage set module.

3.2 Helper modules

Because they are used in many of the main modules, the helper modules are described here first.

3.2.1 Shift registers

The shift registers act like ordinary shift registers, except they only shift once every 8 clock cycles because every modules that uses them interacts with a device running on a slower clock. They both use a 3 bit counter to slow the shifting down. The size of the shift register is variable and determined by the module that instantiates it using Verilog parameters.

The shiftreg_out has three inputs, an input value the width of the register, a shift signal, and a clock signal; it has one single bit output. When shift is low, the input is latched into the register on each positive edge of the clock, the counter is disabled, and the output is tied low. When shift goes high the counter is started and the output switches to the most significant bit of the register. The register maintains its value until the count reaches 7, at which point it shifts it contents left by one bit, shifting a 0 into the LSB. The process continues while shift is held high.

The shiftreg_in has the same ports, except the input value is a single bit and the output value is the width of the register. The shift input is used as the enable signal to the counter. When the count equals 7, the input bit is shifted into the LSB of the register, otherwise the register remains unchanged. When shift is low, the counter does not count and thus never equals 7, so no shifting occurs.

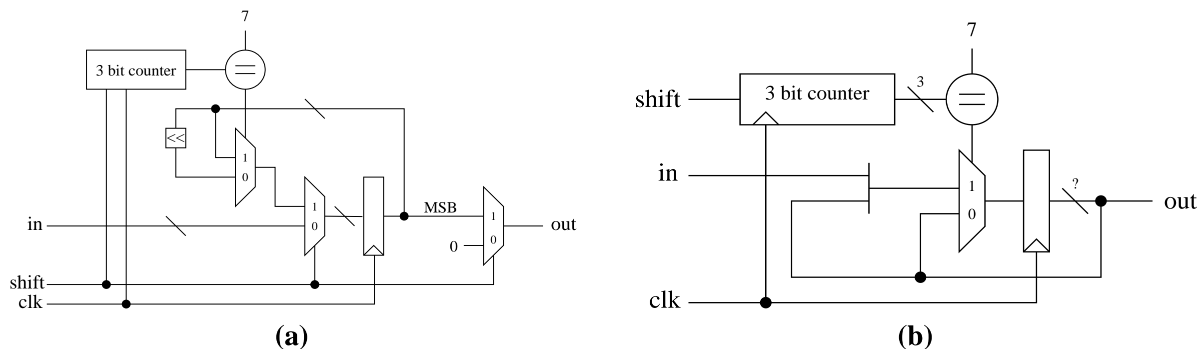

Figure 12: (a) shiftreg_out block diagram. (b) shiftreg_in block diagram.

3.2.2 Counter

The counter is a variable width counter with an enable signal. The width is determined by the module that instantiates it using Verilog parameters. When enable is low, the count stays at zero. When enable is high the count increases by one each rising edge of the clock. The implementation is straightforward.

3.3 Controller module

The controller module is the top level module of the ATB0 controller. Its port are the interface with the PLX interface card as well as the rest of the components on the baseboard, meaning each port in the controller module is an actual pin on the FPGA. Figure 13 is a high level block diagram of the controller module and how it connects everything together. It does not show each module's control signals or the connections between the modules and the external components on the baseboard. Figure 14 illustrates the flow of data through the controller. A memory access request comes in to the controller through the HADS, HLWNR, and HAD signals, is decoded by the decoder and forwarded on to a specific module via the address, enable, w_nr, and data lines. The address bus to the modules is 25 bits wide because the address space requires 27 bits to access and two of those bits are always zero due to word alignment. The modules perform the requested task and communicates with the decode module using the done and da (data available) signals. When the module is done performing its task, the data from the correct module is selected from the data outputs of all the modules using the datasel signal; hadsel then selects to output this data to HAD. The data is sent back to the PLX using the HLRDY, HXDIR, and HAD signals. These three signals are all registered before being sent to the PLX for speed. While the module is performing the task, hadsel allows the status lines from the decoder to be driven onto data_bus and consequently onto HAD. The HXDIR signal is used to enable a tri-state driver which drives the data_bus onto HAD. The protocol of using these signals is described in Section 3.4 when the decode module is described.

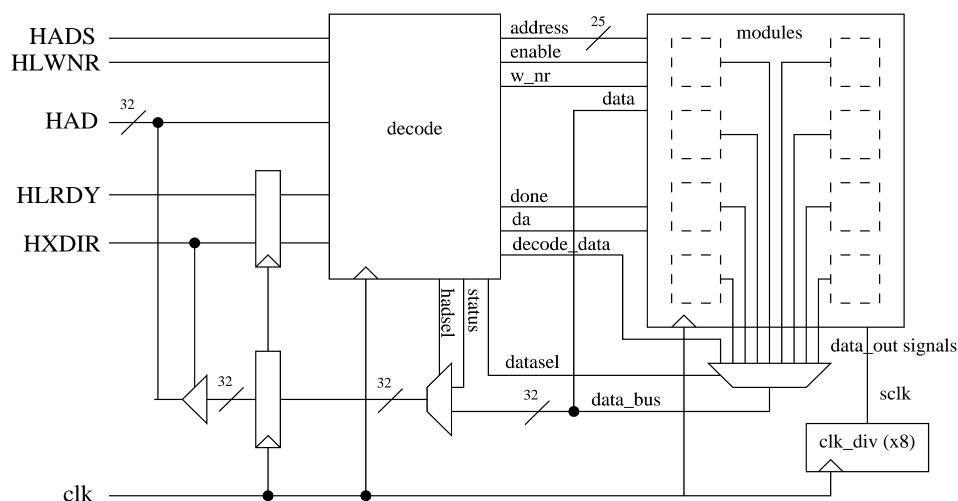

Figure 13: Block diagram of the controller module.

The controller also instantiates a clock divider which is used by many of the components on the baseboard. The clock divider is implemented using a single 3 bit register that increments by one each rising edge of the input clock. The top bit of the register is used as the slow output clock.

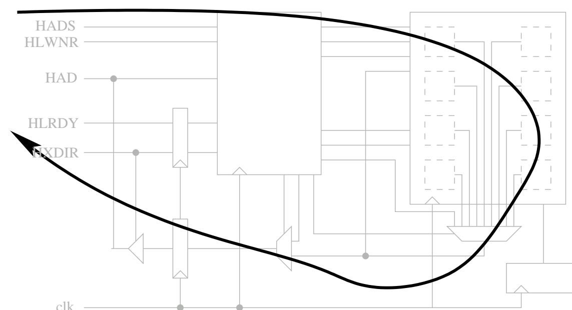

Figure 14: Dataflow through the controller.

3.4 Decode module

The decode module is responsible for communicating with the PLX interface card and controlling all other modules. Inputs from the PLX interface are HADS (Host Adddress/Data Strobe), HLWNR (Host Local Write Not Read), and HAD (Host Adress Data bus). The module also receives a done signal and a da (data available) signal from each module in the controller. There are two outputs to the PLX, HXDIR (Host Transmission Direction), and HLRDY (Host Local Ready). Control signals to each module are a w_nr (write not read) and enable signal. The module has a data output to drive the data bus with and an address bus that goes to each module. Section 3.4.1 describes the PLX interface and Section 3.4.2 describes how the module operates.

3.4.1 PLX Interface

The controller is designed to receive memory access commands from a PLX interface card, with the PLX the bus master and the controller the bus slave. The protocol uses a 32-bit multiplexed address/data bus (HAD), an address/data strobe signal (HADS), a write/read signal (HLWNR), and a ready signal (HLRDY). The PLX begins a transaction by driving HAD with an address, setting HLWNR appropriately, and dropping HADS. This sends the address to the controller. The PLX then waits for the controller to drop HLRDY. If the operation is a write, when the controller drops HLRDY the PLX drives HAD with the write data until HLRDY goes high again. If the operation is a read, when the controller drops HLRDY the controller drives HAD with the read data for the PLX to read. Waiting for the HLRDY signal to drop allows the controller to delay the read or write until it is ready. Figure 15 shows the timing of the transaction.

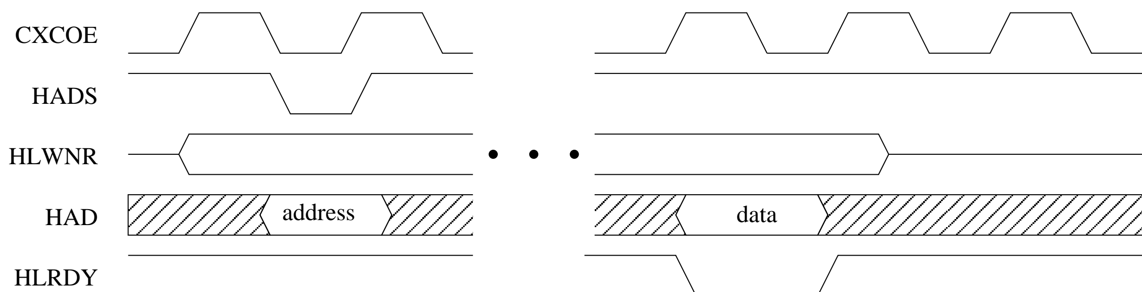

Figure 15: Timing diagram of general PLX access.

3.4.2 Decode implementation

The decode module is one of the more complex modules in the controller. It is responsible for communicating with the PLX and controlling all other modules. Figure 16 shows a block diagram of the control module, with Table 7 providing an alphabetized description of each wire. At the heart of the module, like most of the modules in the controller, is an FSM which performs each step necessary during a read or a write. Table 8 shows each state with its output.

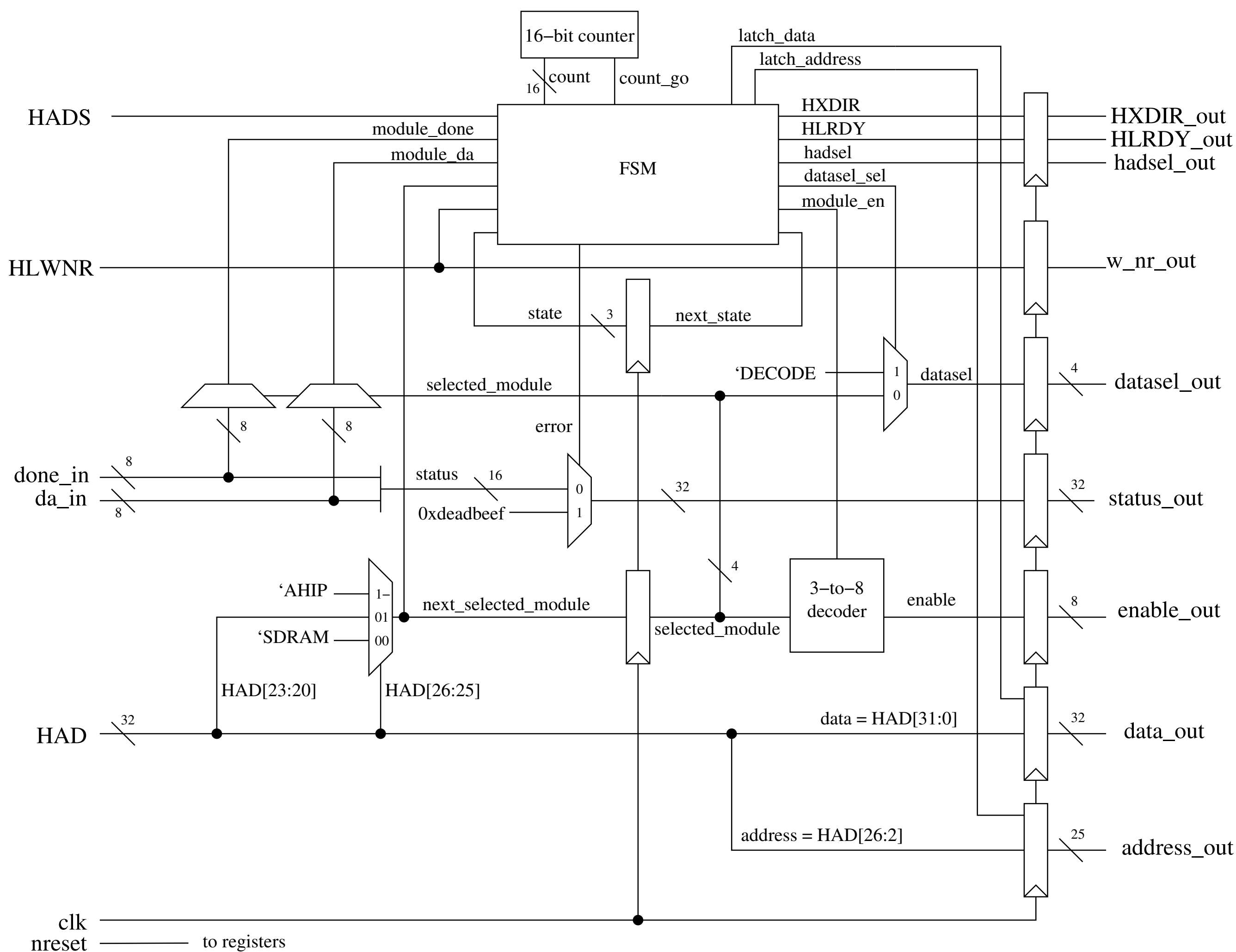

Figure 16: Block diagram of the decode module.

The module processes memory accesses from the PLX as they arrive. An access is initiated when the FSM is idle and the HADS signal is dropped. If HADS is dropped and the FSM is not idle, it is ignored. When the HADS signal is dropped the FSM raises the latch_address signal to latch the address to the address_out bus and examines the address to determine which module should handle the access. To do this, each module is given a number from 0 to 7 as shown in Table 9. If the address is in the SDRAM or AHIP memory spaces, the appropriate module is saved in the selected_module register. If the address is in the control register space, bits 20 to 23 of the address are used as the selected module, this work because the

Signal	Description
address	Output to modules. Used to drive the address bus that goes to each module.
clk	Global clock. Used as clock input to all registers.
count	Input to FSM. 16-bit count. Incremented by one each clock period when count_go is high.
count_go	Output from FSM. Instructs the counter to count.
cs_address	Input from current measure module. Address to drive the address bus with when the current measure module is writing directly to the SDRAM.
cs_we	Input from current measure module. Used to drive the address from the current measure module onto the address bus instead of the address from the PLX. The only signal not from the FSM.
da_in	Input from modules. One signal from each module indicates wheter or not the data supplied by the module is available or not.
data	Output to modules. Used to drive the data_bus when datasel is DECODE.
datasel_sel	Output from FSM. This determine whether the datasel_out output is hardwired to DECODE or is driven with the selected_module register.
done_in	Input from modules. One signal from each module indicates whether or not that module is in an idle state. (i.e. done with a transaction).
enable	Output to modules. One signal to each module which enable that module.
error	Output from FSM. This is used to drive the status bus with the value 0xdeadbeef.
HAD	Input from PLX. Mulitplexed bus that carries both the address and data from the PLX.
HADS	Input from the PLX. Used to initiate a memory access.
hadsel	Output to main controller module. Used to determine if the data bus or the status signal should be driven to HAD on a read. High means the status is driven.
HLRDY	Output to PLX. Used to signal to the PLX that the controller is ready to either send or receive data. When this is low, the data on the bus is valid.
HLWNR	Input from the PLX. Used to determine if memory access is a read or a write.
HXDIR	Output to PLX. Used to determine if the controller of the PLX should drive the HAD bus. High means the controller is driving the bus.
latch_address	Output from FSM. The enable signal for the address_out register. When high, address_out latches the address portion of the HAD bus. This is also the enable signal for the selected_module register.
latch_data	The enable signal for the data_out register. When high the data_out register latches the HAD bus.
module_da	Input to FSM. One of the da_in signals selected by selected_module.
module_done	Input to FSM. One of the done_in signal selected by selected_module.
module_en	This is used to enable the currently selected module. When high, the enable signal going to the module saved in the selected_module register is driven high.
nreset	Global reset. Used to reset all register.
selected_module	An internal register. Contains the module number of the most recent access from the PLX. It determines this using the address on the HAD bus. If the address is in the SDRAM memory space or AHIP memory space it is the number corresponding to those modules, if the address is in the control register space, it obtains the module number from the address itself. Enabled by the latch_address signal.
status	A status word to send back to the user. Contains all the done and da signals from the modules.
w_nr	Output to the modules. The HLWNR signal is simply forwarded to each module to indicate whether the access is a write or a read.

Table 7: Decode module wire descriptions.

state	latch enables data	address	datasel_sel	hadsel	HLRDY	HXDIR	module_en	count_go	error	next_state
IDLE	0	!HADS	1	1	1	1	0	0	0	[1]
WAIT_START	0	0	1	1	1	1	0	1	0	[2]
WRITE	0	0	1	1	0	1	0	0	0	WAIT_WRITE
WAIT_WRITE	count == 1	0	1	1	1	1	count == 1	1	0	[3]
READ	0	0	0	0	1	0	1	0	0	STALL
STALL	0	0	0	0	1	0	0	0	0	WAIT_READ
WAIT_READ	0	0	0	0	1	0	0	1	0	[4]
SEND	0	0	0	[5]	0	0	0	0	0	IDLE
TIMEOUT	0	0	0	HLWNR	0	HLWNR	0	0	1	IDLE

[1] next_state = HADS ? IDLE :
(next_selected_module == DECODE) ? SEND : WAIT_START
[2] next_state = module_done ? HLWNR ? WRITE : READ :
(count = timeout) ? TIMEOUT : WAIT_START
[3] next_state = count == 1 ? IDLE : WAIT_WRITE
[4] next_state = module_da ? SEND :
(count == timeout) ? TIMEOUT : WAIT_READ
[5] hadsel = (selected_module == DECODE)

Table 8: Decode module state definitions.

Module	Number
Voltage Set	0
Voltage Measure	1
Current EMasure	2
Clock	3
SDRAM	4
USER pin	5
LGALED	6
AHIP	7
DECODE	8

Table 9: Module numbers.

control registers are placed in memory such that bits 20 to 23 of the address contain the module number. If the selected module is the decode module, the FSM goes straight to the SEND state to send the status word. Otherwise it goes to the WAIT_START state.

In the WAIT_START state, the FSM waits for the selected module to raise its done signal in case the module is busy finishing a previous operation. During this state the counter is going and if the count reaches a specified timeout (currently 0xFFFF) the FSM goes to the TIMEOUT state. When the selected module's done signal is high, the FSM goes to either the WRITE or READ state depending on the value of HLWNR.

In the WRITE state, the FSM drops the HLRDY signal for one cycle and moves immediately to the WAIT_WRITE state. Here it waits one cycle for the HLRDY signal to make it off the baseboard and to the PLX (it must go through two registers). In the second cycle in the WAIT_WRITE state the latch_data signal is raised to latch the data from the PLX and module_en signal is raised to enable the selected module. Following the second cycle in the WAIT_WRITE state, the FSM goes back to IDLE ready to process another request while the module that just received the write command is processing the write. Figure 17 shows a timing diagram of this process.

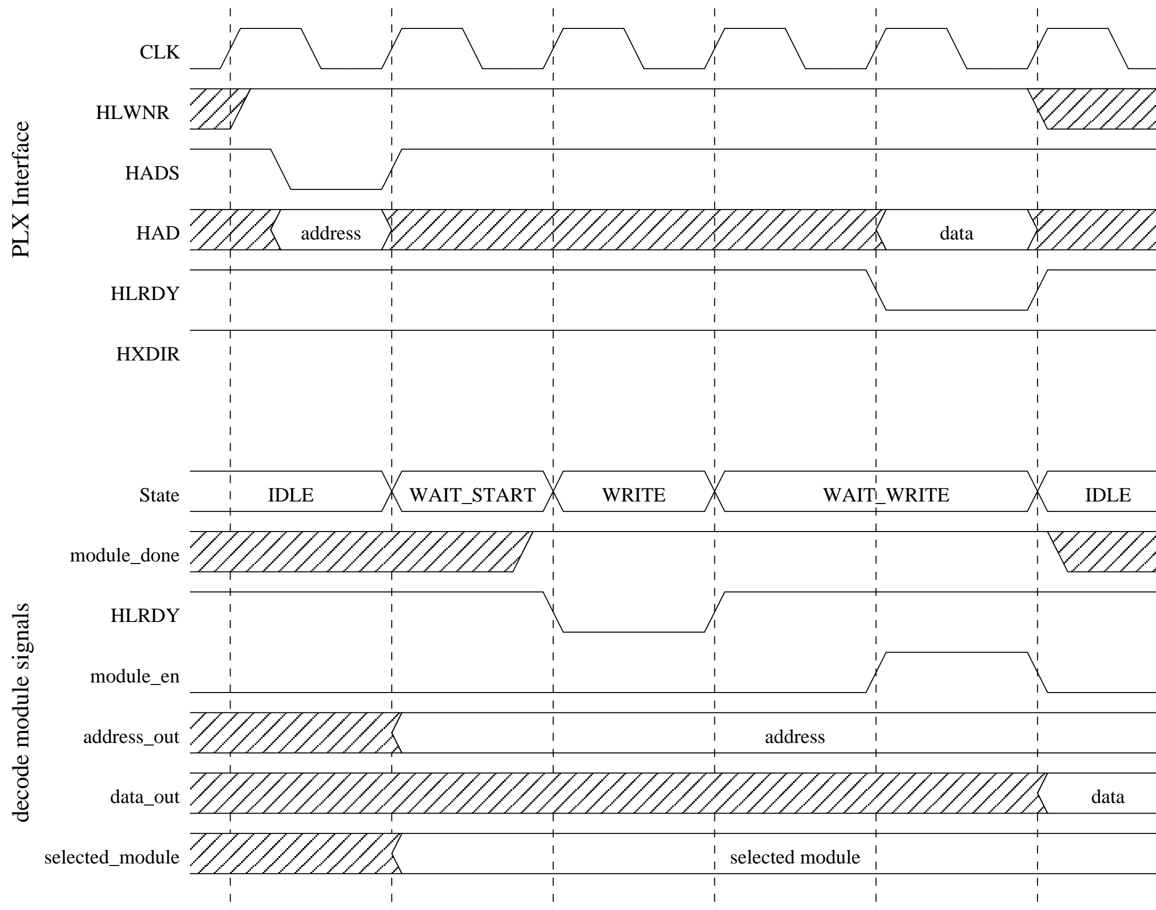

Figure 17: Timing diagram of a write for decode module.

During the READ state, the module_en signal is raised, enabling the selected module. The FSM then waits a cycle in the STALL state to give the module enough time to lower its da signal and for that da signal to make it back to the decoder. After waiting a cycle the FSM enters the WAIT_READ state in which it waits for the module to raise its da signal to indicate that the read data is available. Like the WAIT_START state, this state is timed by the counter and can timeout. Once the module raises its da signal, the FSM moves to the SEND state. In the SEND state, HXDIR is lowered to make the HAD bus driven by the controller and the HLRDY signal is lowered to tell the PLX the data is being sent on the bus. If the selected module is the decode module, hadsel is used to send the status word, otherwise the data bus, which is the output from the currently selected module, is sent to the PLX. After the SEND state the FSM goes back to IDLE. Figure 18 shows a timing diagram of this process.

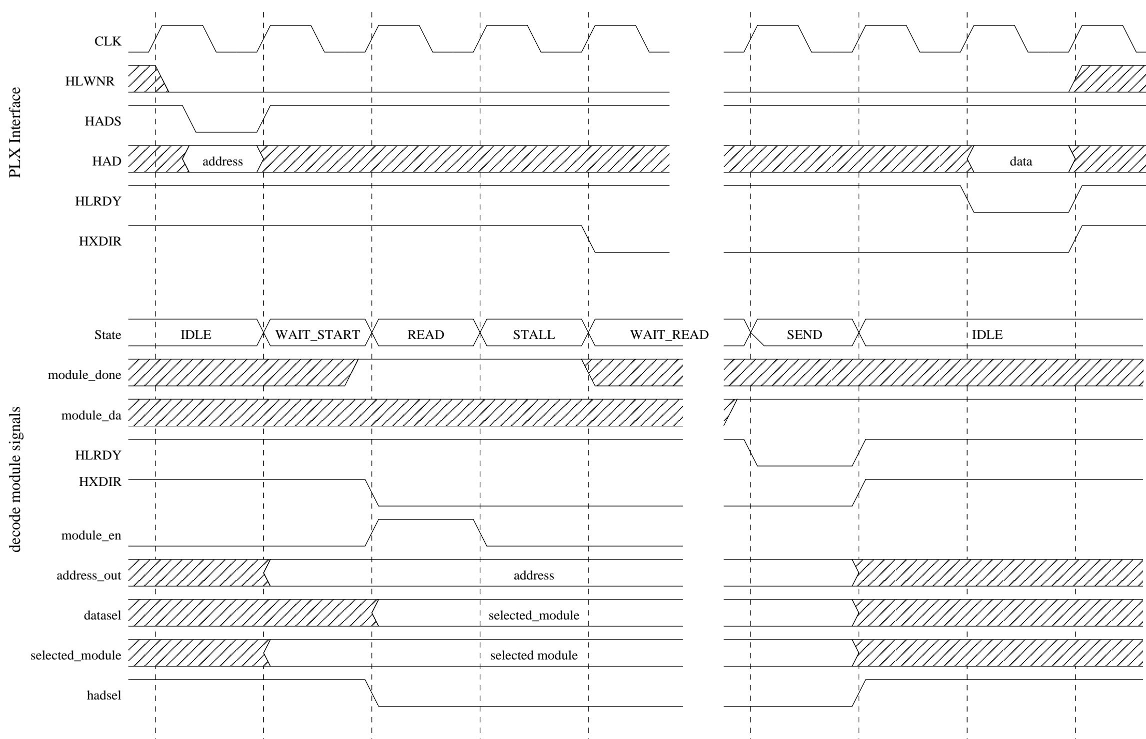

Figure 18: Timing diagram of a read for decode module.

In the TIMEOUT state, HLRDY is dropped so the PLX will stop waiting for the controller. If the access is a read, the hadsel and error signal are used to return 0xdeadbeef. If the access is a write, the write is never performed. Following the TIMEOUT state the controller returns to IDLE.

3.5 SDRAM control module

The SDRAM control module is responsible for all communication with the SDRAM chips on the baseboard. The SDRAM chips need to be regularly refreshed so the module uses a seperate refresh timer to keep track of when a refresh should occur, this is described in Section 3.5.1 then the operation of the actual SDRAM control module is described in Section 3.5.2.

3.5.1 Refresh Timer

The refresh timer has three main inputs, a clock, reset, and maximum value, and one output, an expired signal. It consists of a simple counter that counts up by one each clock cycle until the count equals the maximum value given. When the count reaches the maximum value it stops counting and raises the expired output. It holds expired high until the reset signal is raised at which point it resets the counter to 0, and starts over. A block diagram is shown in Figure 19.

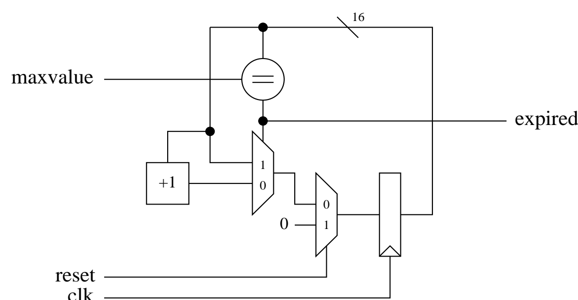

Figure 19: Refresh timer for the SDRAM control module.

3.5.2 SDRAM control module implementation.

The SDRAM control module performs the necessary initialization of the SDRAM chips when reset and performs reads and writes to the SDRAM during use. Along with the standard interface to the decode module, the SDRAM control modules controls all signals going to the SDRAM chips. The global clock is used as the clock for the SDRAM chips and the CKE (clock enable) input to the SDRAM is tied high so the clock is always activated. The input/output mask, SDDQM, to the SDRAM is tied low so SDDQ is never masked. The three command inputs, SDWE_B, SDCAS_B, and SDRAS_B, are controlled by the FSM within the SDRAM control module. The bank address (SDBA), address (SDA), and data (SDDQ) inputs to the SDRAM come from internal registers, as controlled by the FSM.

Figure 20 shows a block diagram of the SDRAM control module, with Table 12 providing an alphabetized description of each wire. Table 10 shows each state with its output. Because it is often necessary to wait a specific number of cycles, three WAIT states are defined in the FSM and an additional wait_state register is added. When in the wait states, the FSM counts down from WAIT3 to WAIT2 to WAIT1 then returns to the state saved in the the wait_state register.

After a reset, the FSM goes through an initialization process to prepare the SDRAM chips for use, this follows the process described in the SDRAM data sheets [6]. After issuing an initial NOP command in the INIT state, in the PRECHARGE_ALL state, sda_sel is set to precharge_all (00b) to output the address 0x400 to indicate the SDRAM should precharge all banks, a PRECHARGE command is then issued to the SDRAM. The FSM waits three cycles to ensure all banks are precharged then issues two REFRESH commands with a 3 cycle delay after each. Finally, a LMR command is issued loading the mode register with the value of 0x021 by setting sda_sel to mode (01b). This tells the SDRAM that the controller wants sequential bursts of length two and a CAS latency of two (this means that data appears two cycles after a read command is issued). After the mode register is loaded, the FSM goes to an IDLE state. Figure 21 shows a timing diagram of this process.

When idle, the FSM waits for a memory access to come from ether the decode module or the current

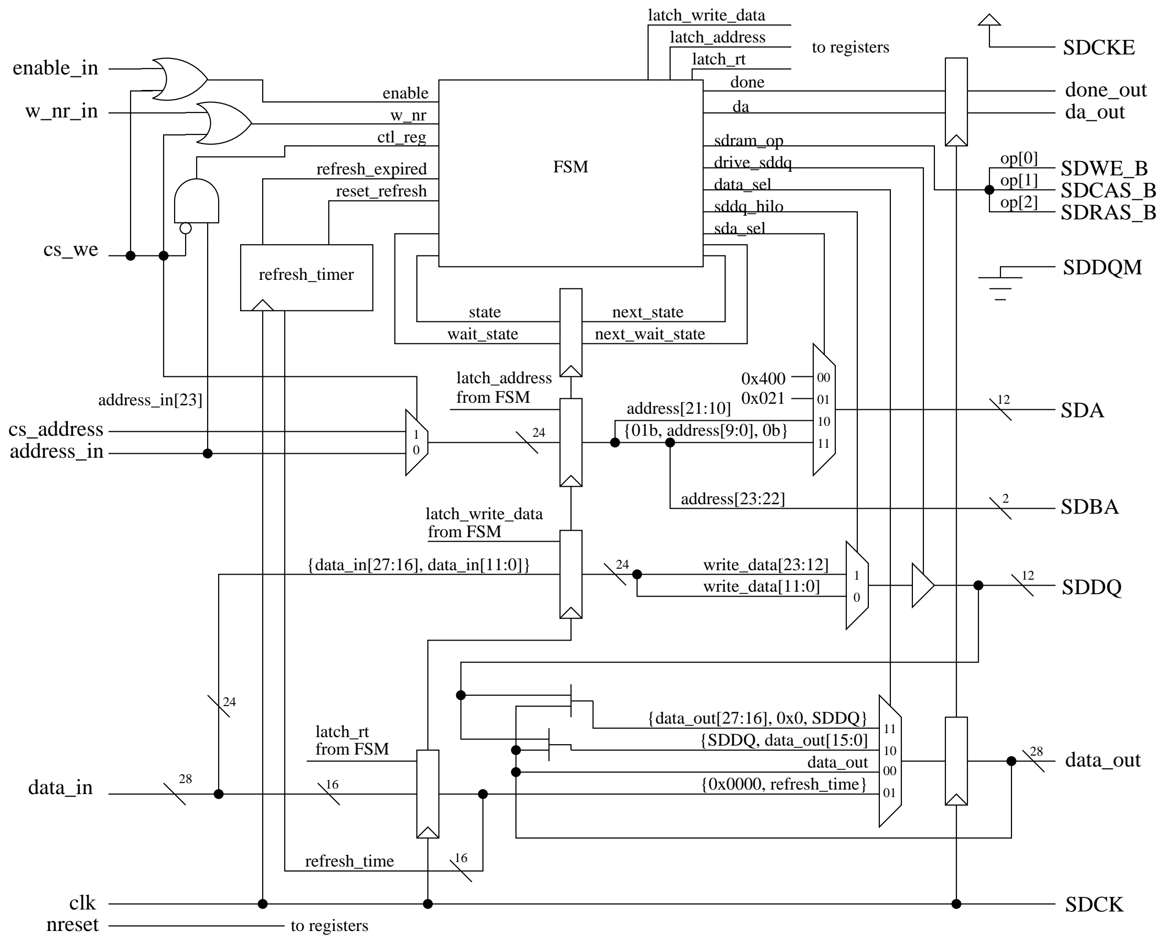

Figure 20: Block diagram of the SDRAM control module.

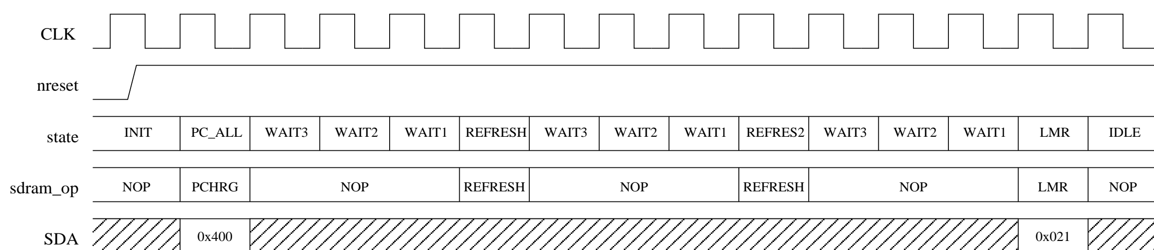

Figure 21: Timing diagram of SDRAM initialization.

				latch enables						
state	sdram_op	data_sel	sda_sel	write_data	rt	address	sddq_hilo	drive_sddq	da	done
INIT	NOP	-	-	0	0	0	-	0	0	0
PRECHARGE_ALL	PRECHARGE	-	precharge_all	0	0	0	-	0	0	0
INIT_REFRESH	REFRESH	-	-	0	0	0	-	0	0	0
INIT_REFRESH2	REFRESH	-	-	0	0	0	-	0	0	0
INIT_LMR	LMR	-	mode	0	0	0	-	0	0	0
IDLE	[1]	[2]	-	[3]	[4]	[3]	-	0	![3]	![3]
BEGIN_WRITE	ACTIVE	save	row	0	0	0	-	0	1	0
WRITE_1	WRITE	save	column	0	0	0	1	1	1	0
WRITE_2	NOP	save	-	0	0	0	0	1	1	0
BEGIN_READ	ACTIVE	save	row	0	0	0	-	0	0	0
READ_ISSUE	READ	save	column	0	0	0	-	0	0	0
READ_1	NOP	get_hi	-	0	0	0	-	0	0	0
READ_2	NOP	get_lo	-	0	0	0	-	0	1	1
WAIT3	NOP	save	-	0	0	0	-	0	0	0
WAIT2	NOP	save	-	0	0	0	-	0	0	0
WAIT1	NOP	save	-	0	0	0	-	0	0	0

[1] sdram_op = refresh_expired ? REFRESH : NOP
[2] data_sel = (enable & ctl_reg & !w_nr) ? rt : save
[3] latch_write_data, latch_address, !da, !done = (enable & !ctl_reg)
[4] latch_rt = (enable & ctl_reg & w_nr)

Table 10: SDRAM control module state definitions.

State	Next State	Next Wait State
INIT	PRECHARGE_ALL	-
PRECHARGE_ALL	WAIT3	INIT_REFRESH
INIT_REFRESH	WAIT3	INIT_REFRESH2
INIT_REFRESH2	WAIT3	INIT_LMR
INIT_LMR	IDLE	-
IDLE	[1]	-
BEGIN_WRITE	WAIT1	WRITE_1
WRITE_1	WRITE_2	-
WRITE_2	WAIT1	IDLE
BEGIN_READ	WAIT1	READ_ISSUE
READ_ISSUE	WAIT1	READ_1
READ_1	READ_2	-
READ_2	WAIT1	IDLE
WAIT3	WAIT2	wait_state
WAIT2	WAIT1	wait_state
WAIT1	wait_state	-

[1] next_state = (enable & !refresh_expired & !ctl_reg) ? (w_nr ? BEGIN_WRITE : BEGIN_READ) : IDLE

Table 11: SDRAM control module state transitions.

Signal	Description
address	This is the last saved address and is split into the three parts: address[23:22] is the bank, address[21:10] is the row, and address[9:0] is the column.
address_in	Input from the decode module, this is saved (latched) as address whenever latch_address is high and cs_we is low.
clk	Global clock. Used as clock input to all registers and sent to the SDRAM as SDCK.
cs_address	Input from current measure module. Address to write to when cs_we goes high.
cs_we	Input from current measure module. Indicates that the current measure module wants to write what is on the data bus to cs_address.
ctl_reg	Used to determine if an access is reading or writing to a control register (namely, the refresh time) or actual SDRAM memory.
da	Output to decode module. Indicates when data_out is valid.
data_in	Input from decode module. Value to write to the SDRAM or set the refresh time to.
data_out	Output to PLX. Contains either the latest value read from SDRAM or the value of the refresh time.
data_sel	Internal control signal. Used to determine what value is latched into data_out. There are four choices: the refresh time, keep the same value, keep the low 16 bits and get the high 16 bits from SDDQ, or keep the high 16 bits and get the low 16 bits from SDDQ.
done	Output to decode module. Indicates that the FSM is idle and ready for a memory acceess.
drive_sddq	Internal control signal. Used as the enable pin to the tri-state driver on SDDQ to determine if the module should drive SDDQ or not.
enable	Input to FSM. Begins a memory access; either cs_we going high or the enable_in signal from the decode module raises enable.
latch_address	Enable signal to address register. When this is high, the address register latches address_in on the the rising edge of the clock.
latch_rt	Enable signal to the refresh time register. When this is high, the refresh_time register latches data_in on the rising edge of the clock.
latch_write_data	Enable signal to write_data register. When this is high, the write_data register latches the appropriate bits of data_in on the rising edge of the clock.
nreset	Global reset. Used to reset all registers.
refresh_expired	Input from refresh timer. Indicates that the timer has reached the maxvalue, which is refresh_time, and the SDRAMs should be refreshed.
refresh_time	The time the user has set to be the number of ticks between refreshes of the SDRAM. Used as the maxvalue input to the refresh timer.
reset_refresh	Input into the refresh timer, indicates that a refresh has occured and the refresh timer should reset and start counting again.
sda_sel	Used to selected what is output to SDA. There are four choices: the command to precharge all banks (0x400), the mode to use (0x021), the row (address[21:10]) or the column (01b, address[9:0], 0b)
SDA	Output to SDRAM. The address bus.
SDBA	Output to SDRAM. The bank address. Obtained from the highest two bits of the last saved address.
SDCAS_B	Output to SDRAM. One of three control signals to the SDRAM that make up the opcode (the others are SDWE_B and SDRAS_B).
SDCK	Output to SDRAM. The clock to the SDRAM.
SDCKE	Output to SDRAM. The clock enable signal, used to put the SDRAM in a low-power standby mode, tied high to indicate SDRAM should always be ready.
SDDQ	Output to SDRAM. The data bus, used for both input and output.
sddq_hilo	Internal control signal. Used to determine wether the high or low bits of the saved write_data is driven to SDDQ when drive_sddq is high.
SDDQM	Output to SDRAM. The mask for SDDQ, tied to ground to indicate SDDQ should always be used.
sdram_op	Output from FSM. Split into SDWE_B, SDCAS_B, and SDRAS_B. Forms the opcode sent to the SDRAM.
SDRAS_B	Output to SDRAM. One of three control signals to the SDRAM that make up the opcode (the others are SDWE_B and SDCAS_B).
SDWE_B	Output to SDRAM. One of three control signals to the SDRAM that make up the opcode (the others are SDRAS_B and SDCAS_B).
write_data	Internal register. The data that will be written to the SDRAM.
wait_state	Internal register. The state to go to after waiting in the wait states.
w_nr	Input to FSM. Used to determine if a memory access is a write or a read, cs_we can force this high, otherwise it follows w_nr_in from the decode module.

Table 12: SDRAM control module wire descriptions.

measure module. To allow the current measure module to write to the SDRAM, cs_we and cs_address are used. When cs_we goes high, enable and w_nr both go high and cs_address is sent to the address register instead of address_in; thus a write from the current measure module looks just like a write from the user. While waiting for an access, if refresh_expired goes high it issues a REFRESH command and resets the refresh timer. When enable is raised indicating an access request has arrived, it checks the ctl_reg signal to see if the access is to the control register (the refresh time) or to SDRAM. If the access is to the control register and a write, latch_rt is raised to latch the new refresh time and the FSM remains idle. If the access is to the control register and a read, then data_sel is set to rt (01b) to output to refresh time and the FSM remains idle. If the access is to SDRAM it latches both the write_data and the address and goes to either BEGIN_WRITE or BEGIN_READ depending on w_nr.

In BEGIN_WRITE, the ACTIVE command is sent to the SDRAM with the top two bits of the latched address as the bank (SDBA) and the next 12 bits as the row to activate (sent on SDA by setting sda_sel to row (10b)). It waits a cycle for the correct bank and row to be activated then goes to the WRITE_1 state. In WRITE_1, the WRITE command is issued to the SDRAM with the the bottom 11 bits of SDA the bottom 10 bits of the latched address multiplied by two because there are two SDRAM locations for every 32-bit word in SDRAM address space. Bit 10 of SDA is set to 1 to indicate we want the SDRAM chip to auto precharge when done, and bit 11 is unused and set to 0. All this is accomplished by setting sda_sel to column (11b). During WRITE_1, sddq_hilo and drive_sddq are high to drive the high 12 bits of the latched write_data onto SDDQ. Next, in WRITE_2, a NOP is issued and sddq_hilo is dropped, keeping drive_sddq high, to drive SDDQ with the low 12 bits of write_data, finishing the burst. The FSM waits one more cycle for the bank to be precharged and then returns to the IDLE state. Figure 22 shows a timing diagram of this process.

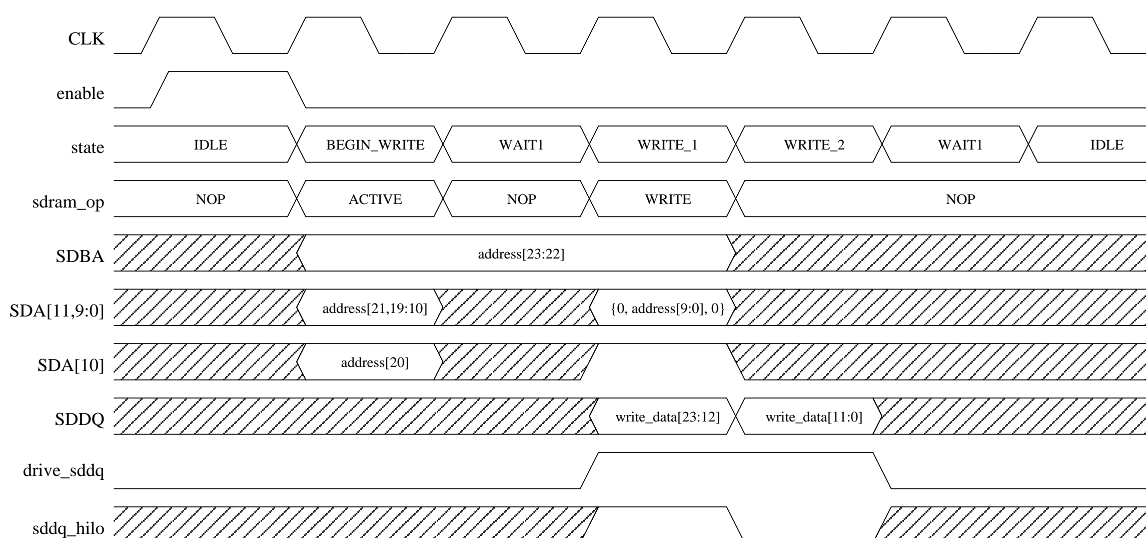

Figure 22: Timing diagram of a SDRAM write.

In BEGIN_READ, the ACTIVE command is sent to the SDRAM with the top two bits of the latched address as the bank (SDBA) and the next 12 bits as the row to activate (sent on SDA), just like BEGIN_WRITE. It waits a cycle for the correct bank and row to be activated then goes to the READ_ISSUE state. In READ_ISSUE the READ command is issued to the SDRAM with SDA selecting the appropriate column and auto precharge as it was during a write. After waiting a cycle for the data to arrive, in READ_1, data_sel is get_hi (10b) to latch the high word from SDDQ into data_out. Then in READ_2 data_sel is get_lo (11b) to

state	done	PVSCSB	sr_shift	vsreg_we	count_go	next_state
IDLE	1	1	0	(enable & w_nr)	0	[1]
SYNC1	0	1	0	0	0	sclk ? SYNC1 : SYNC2
SYNC2	0	1	0	0	0	sclk ? ZEROS : SYNC2
ZEROS	0	0	0	0	1	(count[6:0] == 0x1F) ? SHIFT : ZEROS
SHIFT	count == 0x7FF	0	1	0	1	[2]

[1] next_state = (enable & w_nr & (address == 0xF)) ? (sclk ? SYNC1 : SYNC2) : IDLE
[2] next_state = (count[6:0] == 0x7F) ? (count[10:7] == 0xF) ? IDLE : ZEROS : SHIFT

Table 13: Voltage set module state definitions.

latch the low word from SDDQ into data_out. The FSM waits one more cycle for the bank to be precharged and then returns to the IDLE state. Figure 23 shows a timing diagram of this process.

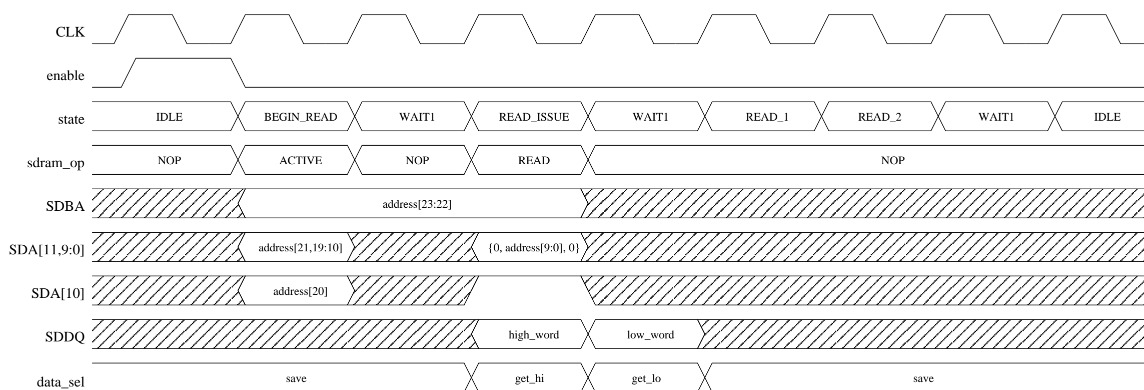

Figure 23: Timing diagram of a SDRAM read.

3.6 Voltage Set module

The voltage set module interacts with the 16 DACs on ATB0 that are used to set the voltage level for each of the 16 power supplies. There are 16 registers to hold the values for each of the power supplies (VSR0 - VSR15). Writing to any of them except VSR15 just sets the register value, it does not set the actual voltage. Writing to VSR15 sets the VSR15 register then initiates the scanning-in process described below to actually set the voltage on each power supply. Reading a voltage set register just returns the value in the register, it does not return the actual voltage of the power supply, the voltage measure module can be used for that.

The DACs used are daisy chained together, and for each chip, "the data at DIN appears at DOUT, delayed by 16 clock cycles plus one clock width" [7]. Therefore, by shifting 256 bits into the first DAC, with values in the order they are daisy chained together, the module sets all 16 of them. This is accomplished with a shiftreg_out module. The ordering of the chips is: 9, 1, 5, 15, 11, 7, 3, 13, 12, 2, 6, 10, 14, 4, 0, 8 (starting from the last in the chain, going to first). See the DAC's datasheet [7] or the hardware document [1] for more information. Another caveat is the DACs can not run at 40 MHz, 15 MHz is the fatest they will go, so we use the slow clock provided by the controller module.

Figure 24 shows a block diagram of the voltage set module, with Table 14 providing an alphabetized description of each wire. The module uses a FSM to perform the scanning-in to the DACs. Table 13 shows each state with its output.

Signal	Description
address	Input from decode module. Contains the power supply number of the access and is used as an address into the register file.
clk	Global clock. Used as clock input to all registers.
count	Input to FSM and used to selected which register in the register file to load into the shift register during shifting. From an 11-bit counter that is incremented by one each clock period when count_go is high.
count_go	Output from FSM. Instructs the counter to count.
da_out	Output to decode module. Tied high to indicate data is always available.
data_in	Input from decode module. Value to set the register to when writing.
data_out	Output to PLX. Always contains the register at the address on the address bus (one cycle later).
done	Output to decode module. Indicates that the FSM is idle and ready for a memory access.
enable	Input from decode module. Begins a memory access.
nreset	Global reset. Used to reset all registers.
PVSCK	Power Voltage Set Clock. Clock to the voltage setting DACs.
PVSCSB	Power Voltage Set Chip Select (Bar). Used to instruct the voltage set DACs to latch the incoming serial data into their internal registers. Active low.
PVSDI9	Power Voltage Set Data In (#9). Serial output to DAC that sets desired voltage for the power supply 9. This is the beginning of the chain of DACs that sets the voltage for all power supplies.
PVSRSB	Power Voltage Set Reset (Bar). Used to reset the voltage set DACs. Active low.
sclk	Slow clock from controller module.
sr_in	Input to the shift register. This is one of the 16 registers in the register file.
sr_shift	Input to the shift register. This instructs the shift register to shift its data onto the serial data line PVSDI9. When this is low, the shift register latches its input, sr_in.
vsreg_we	Write enable input to the register file. When high, data_in is latched onto the register at address.
w_nr	Input from decode. Determines whether an access is a write or a read.

Table 14: Voltage set module wire descriptions.

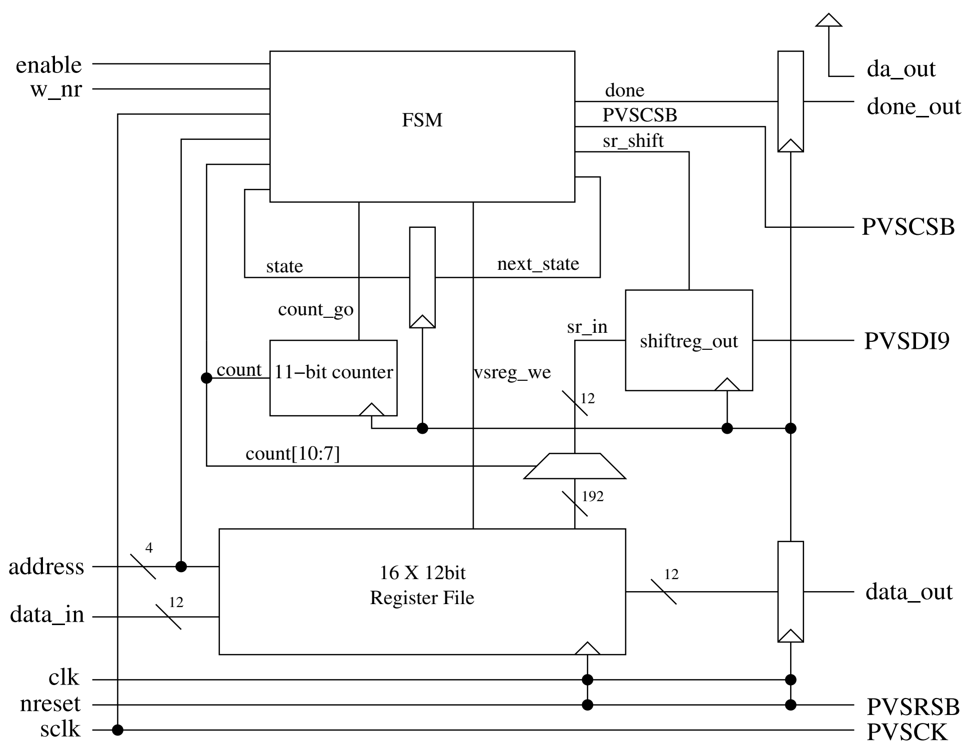

Figure 24: Block diagram of the voltage set module.

The FSM is not used when the user reads and writes to the registers. When both enable and w_nr are high, vsreg_we is raised and the data is latched into the appropriate register, selected by address. The vsreg_we register is part of the FSM to ensure that the registers are only written to when the FSM is idle and not shifting. Since data_out is always driven with the register selected by address, no additional work is needed to perform a read of a register, the data available signal is always high.

When register 15 is written to, the FSM performs the scanning in process. First the FSM gets insync with the slower clock that the DACs use as described in Section 3.1.1. Once synced, the scanning in process occurs during the ZEROS and SHIFT state and PVSCSB is therefore low during both of these states. The counter is also going while scanning in. During the ZEROS state, four zeroes are outputted by holding sr_shift low (causing the shiftreg to hold it's output low) for four sclk cycles (0x1F normal cycles). While waiting for the zeros to complete, the shift register is loaded with the next register value to shift out, based on the value of the count. Because the sclk is 8 times slower than the global clock each bit takes 8 cycles to complete, and each power supply requires 16 bits to be "shifted" in, totalling 128 cycles per power supply. Therefore, the top 4 bits of the count (count[10:7]) can be used to count what power supply's value is being shifted in (i.e. if count[10:7] is 0, VSR9 is loaded into the shift register, if count[10:7] is 2, VSR5 is loaded into the shift register). After waiting 4 sclk cycles in the ZEROS state, the FSM enters the SHIFT state and instructs the shift register to shift out its newly latched value, one bit every 8 cycles. After 12 sclk cycles (or a total of 0x7F normal cycles), the FSM goes back to the ZEROS state and repeats the process. This is continued until the counter reaches 0x7FF, marking the end of the scanning in. The FSM then returns to the IDLE state. Figure 25 shows a timing diagram of the beginning of this process.

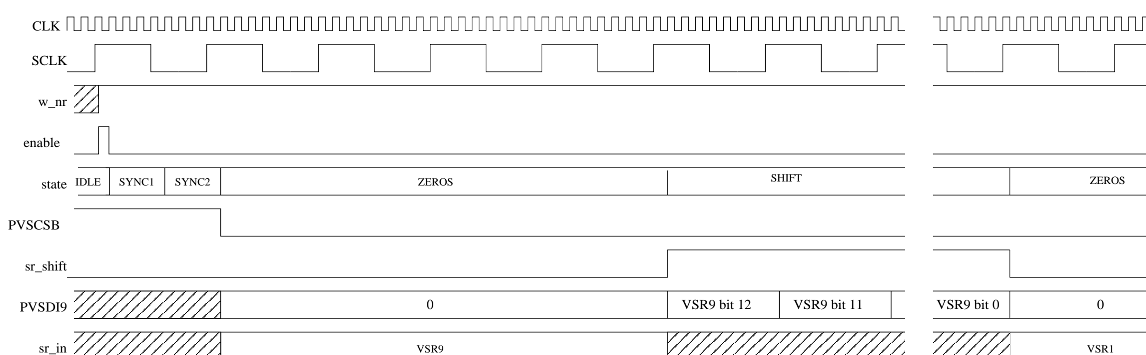

Figure 25: Timing diagram for the voltage set module. Shows the beginning of the scanning in process, will continue to switch between the ZEROS and SHIFT state until all power supplies have been finished.

state	done	da	count_go	sr_shift	latch_address	convert	next_state
IDLE	!read	!read	0	0	read	0	read ? sclk ? SYNC1 : SYNC2 : IDLE
SYNC1	0	0	0	0	0	0	sclk ? SYNC1 : SYNC2
SYNC2	0	0	0	0	0	0	sclk ? SYNC2 : CONV
CONVERT	0	0	1	0	0	1	(count == 0x1) ? SHIFT : CONV
SHIFT	0	0	1	1	0	0	(count == 0xD) ? SAMPLE_NEXT : SHIFT
SAMPLE_NEXT	0	1	1	0	0	0	(count == 0xF) ? IDLE : SAMPLE_NEXT

Table 15: Voltage measure module state definitions.

3.7 Voltage measure module

The voltage measure module interacts with the 14 ADCs on ATB0 that are used to measure the actual voltage of each of the positive power supplies. All 14 ADCs have an individual convert signal and all share a single serial data line. The module uses a single shift register, but from the user's point of view there are 14 different registers, VMR0 - VMR13, one for each power supply. Writes to any register are ignored. When the user reads a register, the module instructs the specified power supply's voltage measure ADC to convert the voltage to a 12-bit value and then reads that value into the shift register. When the shifting has completed, the newly obtained value is sent to the user. A FSM is used to step through the process. Figure 26 shows a block diagram of the module, Table 16 contains an alphabetized description of each wire, and Table 15 shows each state with its outputs.

When the FSM is idle, it waits for enable to go high with w_nr low to indicate a read. When this happens, latch_address is raised to save the address of the read and the FSM gets insync with the slower clock as described in Section 3.1.1. When done syncing, it moves to the CONVERT state and raises the convert signal, causing the PVMCVB to the ADC specified by the saved address to go low; the counter is also started in the CONVERT state. The count into the FSM is the top 4 bits of the actual count, so it is incremented once every 8 cycles, the same rate at which the bits are shifted in. After waiting two sclk cycles in the CONVERT state while the ADC converts the voltage to a value, the FSM moves into the SHIFT state for 12 sclk cycles and raises sr_shift to instruct the shift register to shift the value coming in on PVMD into its register. When done shifting, the data is available so the da signal is raised, but the FSM moves into SAMPLE_NEXT state to wait two sclk cycles for the ADC to get the next sample before raising done and becoming idle, this is to insure another access to the same ADC does not follow too closely. After two sclk cycles in SAMPLE_NEXT, the FSM returns to the IDLE state. Figure 27 shows a timing diagram of this procedure.

Figure 26: Block diagram of the voltage measure module.

Signal	Description
address	Input to decoder. Saved address used to determine which ADC's convert signal to drop when convert is raised.
address_in	Input from decode module. Saved when an access occurs to be used later.
clk	Global clock. Used as clock input to all registers.
count	The top 4 bits of the 7 bit counter.
count_go	Output from FSM. Instructs the counter to count.
da	Output to decode module. Informs decode when data is available to send to PLX.
data_out	Output to PLX. Always contains the current value of the shift register.
done	Output to decode module. Informs decode that FSM is idle and ready for an access.
enable	Input from decode module. Begins a memory access.
latch_address	Output from FSM. Enable signal to address register.
nreset	Global reset. Used to reset all registers.
PVMCK	Power Voltage Measure Clock. Clock to the voltage measure ADCs.
PVMCVB	Power Voltage Measure Convert (Bar). Used to instruct the voltage measure ADCs to convert the voltage on their input to their internal register.
PVMD	Power Voltage Measure Data. Serial input from voltage measure ADCs.
read	Input to FSM. High when enable is high and w_nr is low to indicate a read access from the decode module.
sclk	Slow clock from controller module.
sr_shift	Input to the shift register. This instructs the shift register to shift in data from PVMD. (One bit every 8 cycles).
w_nr	Input from decode module. Determines whether an access is a write or a read.

Table 16: Voltage measure module wire descriptions.

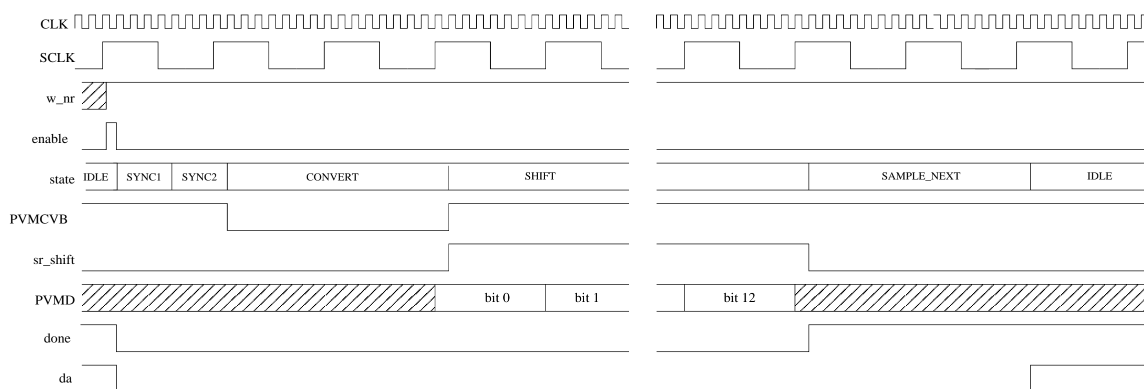

Figure 27: Timing diagram for the voltage measure module. This is the timing of the value from the ADC being shifted into the shift register.

state	done	da	count_go	sr_shift	convert	write_sdram	sdram_we	next_state
IDLE	!conv	!conv	0	0	0	0	0	conv ? sclk ? SYNC1 : SYNC2 : IDLE
SYNC1	0	0	0	0	0	0	0	sclk ? SYNC1 : SYNC2
SYNC2	0	0	0	0	0	0	0	sclk ? CONVERT : SYNC2
CONVERT	0	0	1	0	1	0	0	(bit* == 0x1) ? SHIFT : CONVERT
SHIFT	0	0	[1]	1	0	0	0	[2]
SAMPLE_NEXT	0	1	1	0	0	0	0	(bit == 0x1) ? IDLE : SAMPLE_NEXT
WRITE_SDRAM	0	0	1	0	0	1	[3]	(bit == 0x7) ? IDLE : WRITE_SDRAM

* bit = count[6:3]
[1] count_go = !(bit == 0xD)
[2] next_state = (bit == 0xD ? address[10] ? SAMPLE_NEXT : WRITE_SDRAM : SHIFT
[3] sdram_we = (count[2:0] == 0x7)
latch_address = (state == IDLE & enable) | write_sdram
latch_sda = (state == IDLE & enable & w_nr & address_in == 0x0) | (write_sdram & count[2:0] == 0x0 & count[6:3] != 0x0)

Table 17: Current measure module state definitions.

3.8 Current measure module

The current measure module interacts with the 14 ADCs on ATB0 that are used to measure the current drawn from each of the positive power supplies. One convert signal controls all 14 ADCs and each of them have an individual serial data line. The module works much like the voltage measure module except that it has 14 shift registers instead of just one and does not need a decoder to use the correct convert signal to the ADCs, as there is only one. It also has the added functionality of writing the measurements to the onboard SDRAM when requested. From the users point of view, there is one register (CM_BURST) that holds the address in SDRAM memory space where measurements are put and initiates a measurement of all power supplies when read, there are also 196 ($14 * 14$) registers (CM*mn*) that can be used to read any two power supplies at once. The CM_MASK register is not currently implemented.

When the user reads either CM_BURST or one of the CM*mn* registers, the module instructs each power supply's current measure ADC to convert the voltage from the current sensor to a 12-bit value and then reads those values into the 14 shift registers. When the shifting has completed, if CM_BURST was read, each of the 14 measurements are stored into SDRAM and the address they were stored at is returned to the user; if a CM*mn* register was read, the two values requested are returned to the user.

As usual, a FSM is used to step through the process. Figure 28 shows a block diagram of the module, Table 18 contains an alphabetized description of each wire, and Table 17 shows each state with its outputs.

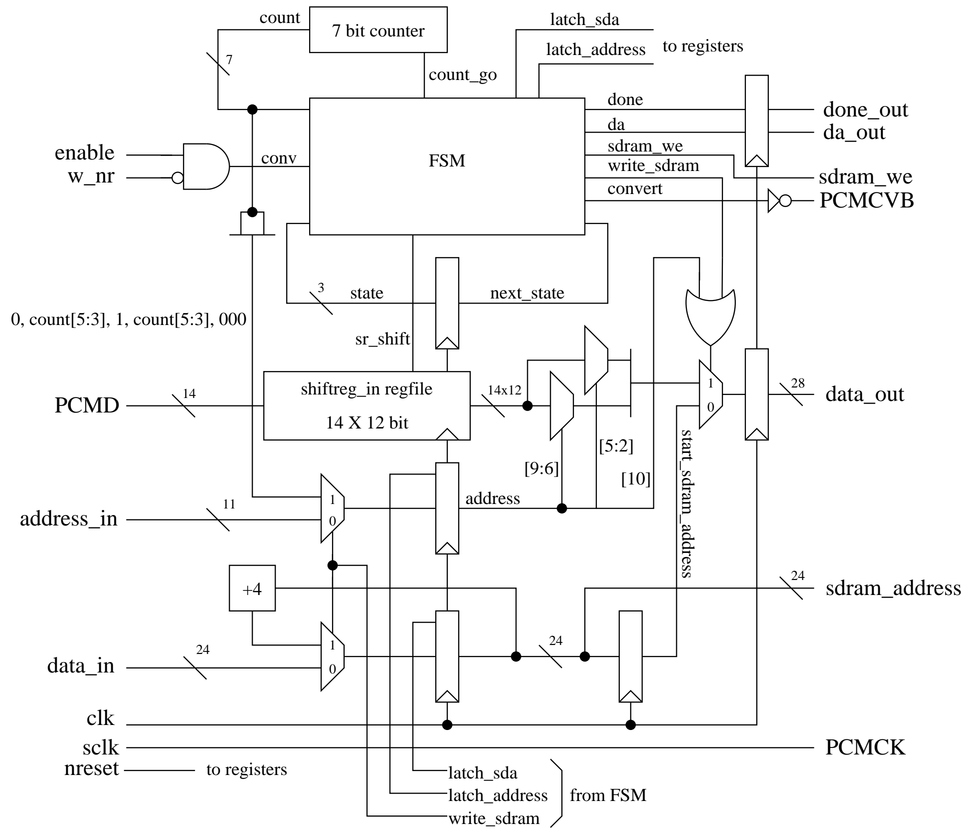

Figure 28: Block diagram of the current measure module.

Signal	Description
address	Internal register. Contains the address of the current memory access. Used to determine what to sent back to the user on the data_out bus.
address_in	Input from decode module. Saved when an access occurs to be used later. The LSB is used immediately when enable is raised to determine if the access requires a conversion or is just reading a control register.
clk	Global clock. Used as clock input to all registers.
count	The count of the 7 bit counter.
count_go	Output from FSM. Instructs to the counter to count.
conv	Obtained from enable, w_nr, and the LSB of address_in. When high, indicates to the FSM that new measurements (a conversion) are required.
convert	Output from FSM. Inverted to obtain the PCMCVB signal.
da	Output to decode module. Informs decode when data is available to send to PLX.
data_in	Input from decode module. Used to set the sdram_address where new measurements should be placed in SDRAM memory.
data_out	Output to PLX. Contains either the value of the two shift registers pointed to by address or start_sdram_address.
done	Output to decode module. Informs decode that FSM is idle and ready for an access.
enable	Input from decode module. Begins a memory access.
latch_address	Output from FSM. Enable signal to address register. When high the address register latches either address_in or an address created from the count when writing to the SDRAM.
latch_sda	Output from FSM. Enable signal to sdram_address register. When high the sdram_address register latches either data_in from the user or sdram_address plus 4 when writing to the SDRAM.
nreset	Global reset. Used to reset all register.
PCMCK	Power Current Measure Clock. Clock to the current measure ADCs.
PCMCVB	Power Current Measure Convert (Bar). Used to instruct the current measure ADCs to convert the voltage on their input to their internal register. Active low.
PCMD	Power Current Measure Data. Individual serial input from each of the 14 current measure ADCs.
sdram_address	Output to decode and SDRAM control modules. Used as the address to write data_out to when sdram_we is high.
sdram_we	Output to decode and SDRAM control modules. Used to tell those module that the current measure module wants to write to SDRAM. When high, the value on data_out is written to the SDRAM address sdram_address.
sclk	Slow clock from controller module.
sr_shift	Input to shift registers. This instructs the shift registers to shift in data from PCMD. (One bit every 8 cycles.)
w_nr	Input from decode module. Determines whether an access is a write or a read.
write_sdram	Output from FSM. Control signal used to instruct rest of module that we are writing to the SDRAM and to behave accordingly. When high, address is obtained from the count, data_out is determined by the address, and sdram_address increments by 4.

Table 18: Current measure module wire descriptions.

When the FSM is idle, it waits for the user to read from CM_BURST or a CM*mn* register, this is done by waiting for the conv signal to go high, indicating that enable is high and w_nr is low, which means a read is requested and a conversion is necessary. If the user is writing to CM_BURST then latch_sda is raised to cause sdram_address to latch the new address on data_in and the FSM remains IDLE. All other writes are ignored.

When conv goes high, the FSM raises latch_address to save the address and drops done and da to indicate that it is busy and data_out is not valid. The FSM then gets insync with the slower clock as described in Section 3.1.1 and moves to the CONVERT state. In CONVERT, it raises the convert signal, causing the PCMCVB to the ADCs to go low. Using the counter, the FSM waits two sclk cycls in CONVERT then moves to the SHIFT state for 12 sclk cycle and raises sr_shift to instruct the shift registers to shift in the value coming in on PCMD into their registers. During the shifting state, start_sdram_address latches sdram_address so it is available to output after a write to SDRAM when sdram_address has changed. On the last cycle of shifting, the counter is reset to prepare for a possible write to SDRAM and the address is checked; if the user is reading from a CM*mn* register, the FSM waits two additional sclk cycles in SAMPLE_NEXT to allow the ADC to sample the next voltage before another read is performed. Because address[10] is high, data_out will be the value of the two shift registers pointed to by the saved address. This part of the process is nearly identical to the voltage measure timing shown in Figure 27.

When done shifting, if the user is reading CM_BURST, the FSM goes into the WRITE_SDRAM state. During the WRITE_SDRAM state, the module is in an 8 cycle loop controlled by the counter. Each loop iteration writes two measurements into an SDRAM word; writing only once every 8 cycles give the SDRAM time to perform the write and return IDLE. Within the loop, the counter is used to generate an address to output. Bits 9 to 6 of the address are count[5:3] followed by a 1, and bits 5 to 2 of the address are count[5:3] followed by a 0. This way as the count progresses upward, the address bits used to determine data_out move through 0x10, 0x32, 0x54, etc. on up to 0xCB, changing every time around the 8 cycle loop. At the beginning of the loop, sdram_address is also incremented by one (except for the first iteration, when it is not incremented). The address into the SDRAM control module is a word address, not a byte address, so incrementing this address by one takes us to the next word in SDRAM memory space. The next cycle in the loop, data_out contains the values of the appropriate shift registers. Then at the end of a loop iteration (when count[2:0] is 7) the sdram_we signal is raised to tell the SDRAM controller to write the value on data_out to sdram_address. A timing diagram of two loop iterations is shown in Figure 29. This method was chosen as opposed to performing a burst write to the SDRAM to avoid dealing with bank and row boundaries.

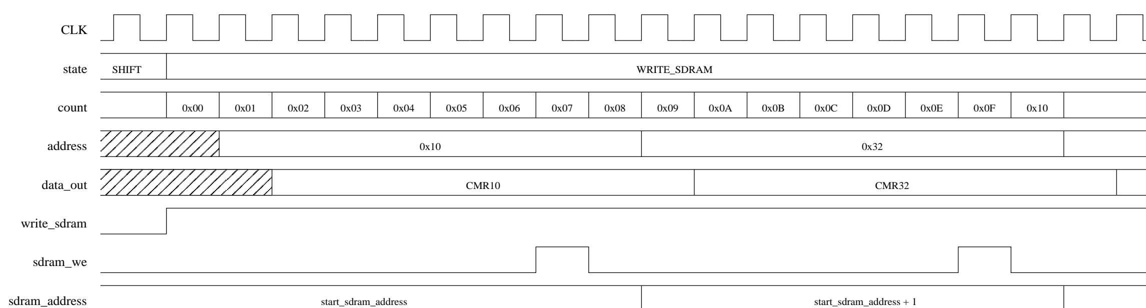

Figure 29: Timing diagram of the current measure module writing to SDRAM memory.

state	done	CKSL	sr_shift	count_go	latch_data	next_state
IDLE	!write	0	0	0	write	write ? (sclk ? SYNC1 : SYNC2) : IDLE
SYNC1	0	0	0	0	0	sclk ? SYNC1 : SYNC2
SYNC2	0	0	0	0	0	sclk ? WAIT : SYNC2
WAIT	0	0	0	!(count == 0x2)	0	(count == 0x2) ? SHIFT : WAIT
SHIFT	0	1	1	1	0	(count == 0x70) ? IDLE : SHIFT

Table 19: Clock module state definitions

3.9 Clock module

The clock module interacts with a programmable frequency synthesizer to set the frequency of the clock sent to the daughtercard. A 14 bit control register is used to configure the chip (CLOCK). When the register is written to the value written to the register is immediately shifted into the frequency synthesizer and saved for later reference. When the register is read, the last value sent to the synthesizer is returned. As usual, an FSM is used to step through the process. Figure 30 shows a block diagram of the module, Table 20 contains an alphabetized description of each wire, and Table 19 shows each state with its output.

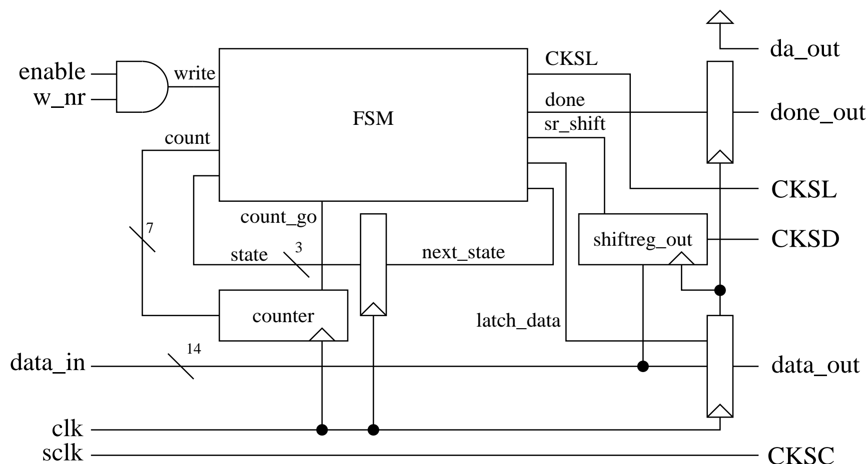

Figure 30: Block diagram of the clock module.

When the FSM is idle, it waits for enable to go high with w_nr high to indicate a write operation. When this happens latch_data is raised to latch the data for future read back and done is dropped to indicate the module is no longer idle. After syncing up with the slower clock of the frequency synthesizer as described in Section 3.1.1, the module starts the counter and waits an additional three cycles in the WAIT state. It does this so the shift register shifts in the middle of the CKSC cycle to help meet the long hold time of the frequency synthesizer. After waiting the three cycles (normal cycles, not sclk cycles), the FSM enters the shift state where it raises CKSL to instruct the frequency synthesizer to load in a new value, raises sr_shift to instruct the shift register to shift out its value onto CKSD, and waits another 14 sclk cycles (or 0x70 normal cycles) while the shift registers shifts. It then returns to IDLE. Figure 31 shows a timing diagram of this process.

Signal	Description
CKSC	Clock Set Clock. Clock sent to the frequency synthesizer.
CKSD	Clock Set Data. Serial data signal to the frequency synthesizer.
CKSL	Clock Set Load. Instructs the frequency synthesizer to load the data coming in on CKSD. Active high.
clk	Global clock. Used as clock input to all registers.
count	Input to FSM. A 7 bit counter, incremented by one each clock edge if count_go is high
count_go	Output from FSM. Instructs the counter to count.
da_out	Output to decode. Tied high to indicate data is always available.
data_in	Input from decode module. The value to shift into the frequency synthesizer.
data_out	Output to PLX. Contains the last value shift into the frequency synthesizer.
done_out	Output to decode module. Used to indicate when the FSM is idle.
enable	Input from decode module. Begins a memory access.
latch_data	Output from FSM. Enable signal to data_out register. data_out latches data_in on the rising edge of the clock if this is high.
sclk	Slow clock from the controller module.
sr_shift	Output from FSM. Instructs the shift register to shift its value out onto CKSD.
w_nr	Input from decode module. Determines whether an access is a write or a read.
write	Input to FSM. This is obtained by anding enable and w_nr and indicates when the decode module issues a write access.

Table 20: Clock module wire descriptions.

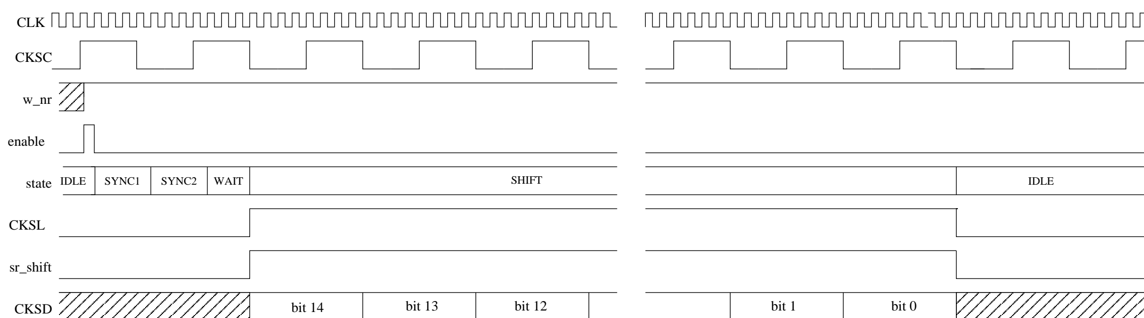

Figure 31: Timing diagram for the clock module.

3.10 User pin control module

The user pin control module is responsible for handling the 26 user pins connected to the daughtercard. It allows the user to set both the direction of the pin (i.e. whether the controller or the daughtercard drive the pin) and the value of the pin if the controller is driving it. The user can read each pin's value whether it is being driven by the controller or daughtercard and each pin's direction, as well as all pin values and directions at once. It is one of the few modules in the controller that does not use a FSM. Figure 32 is a block diagram of the module and Table 21 is an alphabetized description of each wire.

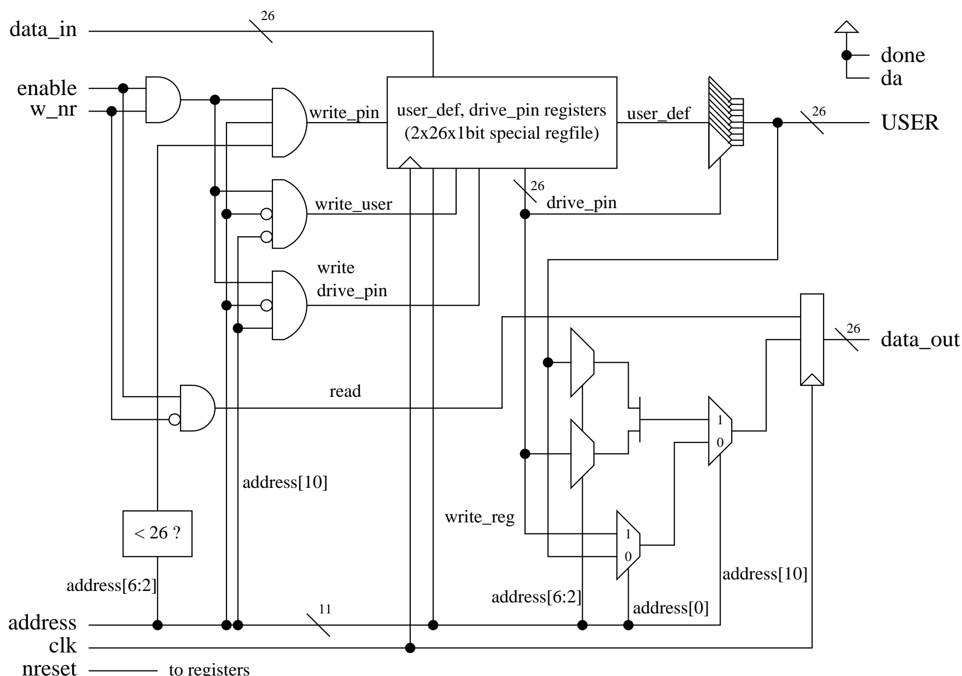

Figure 32: Block diagram of the User pin control module.

The module uses a "special" register file that acts much like two separate register files, each with 26 1 bit register, but allows for some special features as described below. One register file contains the 26 user_def registers, each register is the user defined value for one pin. The other register file contains the 26 drive_pin register, each register is the direction of one pin. The output of the combined register file is two busses, one containing each drive_pin register and one containing each user_def register. The user_def bus is separate from the USER bus, which is the actual value on the user pin whether it is being driven by the controller or daughtercard. 26 seperate tri-state drivers are used to drive each pin of the USER bus with the corresponding user_def register if the corresponding drive_pin register is high.

Reads are split into two categories: reads of a "register" (reading USER_ALL or USER_DIR) and a read of a user pin (one of the USER*p* registers). If the read is to a register, address[0] selects either the drive_pin or USER bus depending on if USER_ALL or USER_DIR is read. Address[10] then selects to send the selected bus to the data_out register where it is latched into data_out and sent as the output. If the read is to a user pin, address[6:2] selects a drive_pin register and a bit from the USER bus, puts the drive_pin register in bit 1 and the bit from USER in bit 0 of a new bus and address[10] selects that bus to latch into data_out and send as the output. The data_out register keeps its value unless a read occurs, at which point it

Signal	Description
address	Input from decode module. Address[10] determines if the lower bits select a pin number or a register. Address[6:2] hold the pin number and address[1:0] is the register number.
addr_okay	A signal that indicates if address[6:2] is below 26 and thus a valid pin number.
clk	Global clock. Used as clock input to all registers.
da	Output to decode module. Tied high to indicate data is always available.
data_in	Input from decode module. Value to set the pin(s) to.
data_out	Output to PLX. Contains either value and direction of a user pin, all values, or all directions. Latches new value on positive edge of the clock when read is high.
done	Output to decode module. Tied high to indicate module is always idle.
drive_pin	A register that contains one bit for each pin that determines whether or not the module should drive the pin or leave it in high impedance.
enable	Input from decode module. Begins a memory access.
nreset	Global reset. Used to reset all registers.
read	Indicates that a memory read is requested. Used an enable signal to the data_out register.
USER	26 I/O pins connected to the daughtercard.
user_def	The user defined value for each pin. A bit in this register is driven onto the User pin if the corresponding bit in drive_pin is high.
w_nr	Input from decode module. Determines whether an access is a write or a read.
write_pin	Indicates a write has been requested to a user pin register and that the pin number is valid.
write_reg	Indicates a write has been requested to a register.

Table 21: User pin control module wire descriptions.

latches the new value on the rising edge of the clock.

The register file knows how to handle writes to any of the registers, which is why it is called “special”. If a user pin is written to, address[6:2] is checked to make sure it is a valid pin number. If data_in is 0 or 1, then the user_def register at address[6:2] is set to data_in[0] and the drive_pin register at address[6:2] is set to 1. If data_in is 2, then the drive_pin register at address[6:2] is set to 0 to reset the pin to be an input. If the USER_ALL register is written to, all user_def registers are set with the bits from data_in and all drive_pin registers are set to 1. If the USER_DIR register is written to, all drive_pin registers are set to 0 to reset them all to inputs.

3.11 AHIP module

The AHIP module is responsible for communicating with the daughtercard using the Asynchronous Host Interface Port (AHIP) protocol (see Section 2.3.2.1). The module can be used in one of four modes: normal mode, test mode, 8 bit mode, and 8 bit test mode. The module has one control register which holds the current mode. Writes to the AHIP daughtercard address space are forwarded to this module by the decode module. This module is then responsible for performing the memory access to the daughtercard using AHIP and returning the result. Figure 33 shows a block diagram of the module with Table 22 providing an alphabetized description of each wire. An FSM is used to walk through the protocol, Table 23 provides definition of each state and Table 24 shows the state transitions.

When the user writes to the AHIP_MODE address, address[24] is low and w_nr is high, so when enable is raised by the decode module, latch_mode goes high and the value on data_in, which is the new mode, is latched onto the mode register for future use. The FSM remains idle. When the user reads from the AHIP_MODE register, address[24] is low and w_nr is low, so when enable is raised by the decode module,

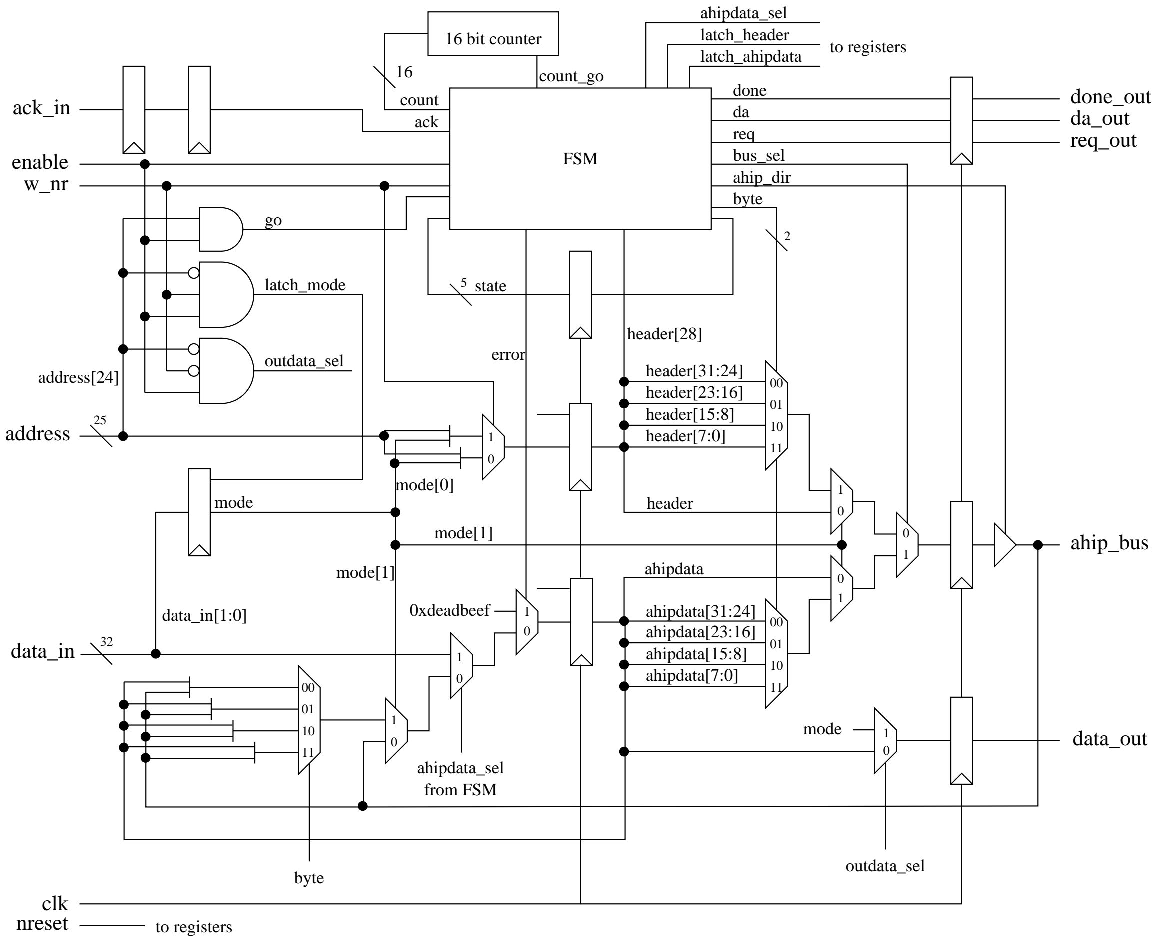

Figure 33: Block diagram of the AHIP module.

Signal	Description
ack	Input to FSM, one of two control signals used during AHIP.
ack_in	Asynchronous input from daughtercard. Synchronized through two flip flops to make ack.
address	Input from the decode module. The top bit (bit 24) is used to determine if a memory access is to the mode register or AHIP daughtecard memory space, the bottom 24 bits are used, along with the current mode and w_nr, to generate the AHIP header.
ahip_bus	32 bit I/O bus connected to the daughtercard, used as both input and output to transfer data.
ahipdata	An internal register used to latch either the outgoing data to send to the daughtercard or the incoming data being received from the daughtercard.
ahipdata_sel	Output from FSM. Used to determine if ahipdata is latched with data_in or the data from the daughtercard that is on the ahip_bus.
ahip_dir	Output from FSM. Used as the enable signal to the tri-state driver on ahip_bus. When this is high, either the header or ahipdata is driven to ahip_bus. When low, ahip_bus is not driven by the controller.
bus_sel	Output from FSM. Used to select whether the header of ahipdata is driven to ahip_bus when ahip_dir is high.
byte	Output from FSM. Used during 8 bit operation to determine which byte of the header or ahipdata to drive to ahip_bus. Also used to determine which byte of ahipdata to replace with ahip_bus when performing a read.
clk	Global clock. Used as clock input to all registers.
count	16 bit counter value. Used to detect timeouts.
count_go	Output from FSM. Instructs counter to count.
da	Data avaialable output to controller. Indicates value on data_out is valid.
data_in	Input from the decode module. Data that is to be sent to the daughtercard.
data_out	Output to PLX. Either the current mode or the last read value of ahipdata, sent to the user via the PLX.
done	Output to controller. Indicates module is done with a transaction and idle.
enable	Input from decode module. Instructs the module to perform the access requested.
error	Output from FSM. Indicates the module has timed out, used to latch 0xdeadbeef into ahipdata to send to the user if the operation that timed out was a read.
go	Input to FSM. And of address[24] and enable, indicates an access to an address in AHIP daughtecard space has been requested and the FSM should perform an AHIP transaction.
header	Internal register. The header to send at the beginning of an AHIP transaction.
latch_ahipdata	Output from FSM. Enable signal to ahipdata register.
latch_header	Output from FSM. Enable signal to header register.
latch_mode	Indicates a write to the mode register. Enable signal to mode register.
mode	Internal register. Indicates the current mode of operation. Bit 0 is high if in a test mode, bit 1 is high if in an 8 bit mode.
nreset	Global reset. Used to reset all registers.
req	Output from FSM to daughtercard. One of two control signal used in the AHIP protocol.
w_nr	Input from decode module. Indicates whether a memory access is a write or a read.

Table 22: AHIP module wire descriptions.

	latch enables										
state	da	done	req	header	ahipdata	ahip_data_sel	bus_sel	ahip_dir	count_go	error	byte
IDLE	go	1	0	1	go & w_nr	go & w_nr	0	1	0	0	0
ISSUE	0	0	1	0	0	-	0	1	1	-	0
R_GETDATA	0	0	0	0	1	0	-	0	1	0	0
R_ACK	1	0	1	0	0	-	-	0	1	-	0
R_DONE	1	0	0	0	0	-	0	1	1	-	0
W_SENDDATA	1	0	0	0	0	-	1	1	1	-	0
ISSUE1	0	0	1	0	0	-	0	1	1	-	0
ISSUE2	0	0	0	0	0	-	0	1	1	-	1
ISSUE3	0	0	1	0	0	-	0	1	1	-	2
ISSUE4	0	0	0	0	0	-	0	1	1	-	3
R8_GETDATA1	0	0	1	0	1	0	-	0	1	0	0
R8_GETDATA2	0	0	0	0	1	0	-	0	1	0	1
R8_GETDATA3	0	0	1	0	1	0	-	0	1	0	2
R8_GETDATA4	0	0	0	0	1	0	-	0	1	0	3
W8_SENDDATA1	1	0	1	0	0	-	1	1	1	-	0
W8_SENDDATA2	1	0	0	0	0	-	1	1	1	-	1
W8_SENDDATA3	1	0	1	0	0	-	1	1	1	-	2
W8_SENDDATA4	1	0	0	0	0	-	1	1	1	-	3
TIMEOUT	0	0	0	0	1	0	-	0	0	1	0

Table 23: AHIP module state definitions.

State	Next State
IDLE	go ? mode[1] ? ISSUE1 : ISSUE : IDLE
ISSUE	ack ? header[28] ? R_GETDATA : W_SENDDATA : ISSUE
R_GETDATA	ack ? R_GETDATA : R_ACK
R_ACK	ack ? R_DONE : R_ACK
RDONE	ack ? R_DONE : IDLE
W_SENDDATA	ack ? W_SENDDATA : IDLE
ISSUE1	ack ? ISSUE2 : ISSUE1
ISSUE2	ack ? ISSUE2 : ISSUE3
ISSUE3	ack ? ISSUE4 : ISSUE3
ISSUE4	ack ? ISSUE4 : header[28] ? R8_GETDATA1 : W8_SENDDATA1
R8_GETDATA1	ack ? R8_GETDATA2 : R8_GETDATA1
R8_GETDATA2	ack ? R8_GETDATA2 : R8_GETDATA3
R8_GETDATA3	ack ? R8_GETDATA4 : R8_GETDATA3
R8_GETDATA4	ack ? R8_GETDATA4 : R_ACK
W8_SENDDATA1	ack ? W8_SENDDATA2 : W8_SENDDATA1
W8_SENDDATA2	ack ? W8_SENDDATA2 : W8_SENDDATA3
W8_SENDDATA3	ack ? W8_SENDDATA4 : W8_SENDDATA3
W8_SENDDATA4	ack ? W8_SENDDATA4 : IDLE
TIMEOUT	IDLE

Table 24: AHIP module state transitions.

outdata_sel goes high and the contents of the mode register are latched onto the data_out register to send the current mode back to the user. Again, the FSM remains idle.

While in the IDLE state, ahip_dir is high to drive the ahip_bus with header because the host is responsible for driving the AHIP bus when no transactions are taking place. When the user reads or writes to an address within the AHIP daughtercard address space, address[24] is high, so when the decode module raises the enable signal, go goes high and the FSM begins processing the transaction. First the header is created using w_nr and mode[0] to create the correct opcode, the bmc field is set to zero, and the low 24 bits of address are used for the address. This all happens on the cycle enable goes high, while the FSM is idle by raising latch_header, latch_ahipdata and ahip_data_sel are also raised if the operation is a write to save the data to write that is on data_in. The FSM then moves to the ISSUE state where it raises req and waits for ack to go high. Once the AHIP operation has been issued, bit 28 is checked to see if the operation is a write or a read (w_nr may not be valid any more). If the operation is a write, the FSM goes to the W_SENDDATA state. If the operation is a read, the FSM goes to R_GETDATA state.

In the W_SENDATA state, req is dropped, ahip_dir remains high to drive ahip_bus, and bus_sel is also raised high to send ahipdata to the ahip_bus. The FSM waits for ack to be dropped to indicate the slave received the data and then returns to the IDLE state.

In the R_GETDATA state, req is dropped, latch_ahipdata is raised with ahip_data_sel 0 to latch the data coming in on ahip_bus into ahipdata, and ahip_dir is dropped to allow the slave to drive ahip_bus. Some invalid value may be latched onto ahipdata during this state, but in the cycle that ack is dropped by the slave, the correct value will be latched into ahipdata and the FSM can move onto the R_ACK state where it raises req to inform the slave that it received the data. During this state, ahip_dir is still low because the slave is still driving the bus. The FSM waits in R_ACK until the slave raises ack once again to indicate it is no longer driving the bus, the FSM then move to the R_DONE state where it drops req, starts driving the bus again and waits for ack to be dropped to signal the end of the transaction, it then returns to the IDLE state.

During each of the states, besides IDLE, the counter is going. If the counter ever reaches its maximum value, the FSM goes to the TIMEOUT state where it raises error to return 0xdeadbeef to the user if a read was in progress. The maximum value of the counter is used because we want the timeout to be very long, if the operation was a read and the controller stops waiting for a response too soon, the AHIP client on the daughtercard could get stuck waiting for the controller to acknowledge receiving the data, which would never happen.

If the module is in an 8 bit mode, the ISSUE, R_GETDATA, and W_SENDDATA states are split into 4 seperate states that perform the necessary action on a single byte. When in these stages, a change in ack moves the FSM to the next state and req is switched from state to state. A read if finished the same way whether it was done using an 8 bit bus or a 32 bit bus.

Figures 9, 10, and 11 in Section 2.3.2 show timing diagrams for all AHIP transactions.

3.12 LGALED module

The LGALED module is simply a register that latches its input when enable and w_nr are high. It always drives both its data out signal to the PLX and the LED and LGA signals to the baseboard with the contents of the register. Its da and done signals are tied high.

References

[1] Jared Casper. *ATB0 Engineering Document - Hardware*. `assam cvs: atb0/doc/hardware`.

[2] Jared Casper. *ATB0 Engineering Document - Software*. `assam cvs: atb0/doc/software`.

[3] Krste Asanović Kenneth Barr Albert Ma Michael Zhang. *ATC0 Engineering Document*. `assam cvs: atc0soft/doc/atc0design`.

[4] James Beck. *IRAM & BRASS Test Chip Testing Plan*. `assam cvs: atb0/doc/www`.

[5] *Programmable Frequency Synthesizer (SY89429A) datasheet*. `assam cvs: atb0/doc/datasheets/SY89429a_programmableFreqSynth.pdf`.

[6] *Micron Synchronous DRAM (MT48LC16M4A2) datasheet*. `assam cvs: atb0/doc/datasheets/64MSDRAM_C.pdf`.

[7] *Maxim +5V, Low-Power, Voltage-Output, Serial 12-Bit DACs (MAX531) datasheet*. `assam cvs: atb0/doc/datasheets/MAX531_DAC.pdf`.

A Pinout listings

Pin	DTX#	ATB0 Xilinx	J1	J2	ATC0	ATC0 LA*	ADB0 Xilinx	ADB0 LA
ahip_bus0	52	215	55		69	1/0/7	AN4	1/11/51
ahip_bus1	53	214	56		71	1/1/11	AP6	2/11/52
ahip_bus2	50	217	57		72	1/2/15	AN5	1/10/47
ahip_bus3	51	216	58		74	1/3/19	AP7	2/10/48
ahip_bus4	48	220	59		75	1/4/23	AJ9	1/9/43
ahip_bus5	49	218	60		77	1/5/27	AK11	2/9/44
ahip_bus6	46	223	63		78	1/6/31	AJ10	1/8/39
ahip_bus7	47	221	64		80	1/7/35	AK10	2/8/40
ahip_bus8	44	225	65		81	1/8/39	AM2	1/7/35
ahip_bus9	45	224	66		83	1/9/43	AL9	2/7/36
ahip_bus10	34	236	79		53	1/10/47	AK9	1/2/15
ahip_bus11	35	235	80		51	1/11/51	AG11	2/2/16
ahip_bus12	32	238	81		50	1/12/55	AM6	1/1/11
ahip_bus13	33	237	82		48	1/13/59	AP4	2/1/12
ahip_bus14	30	3	83		47	1/14/63	AL6	1/0/7
ahip_bus15	31	239	84		45	1/15/67	AP5	2/0/8
ahip_bus16	28	51		84	44	4/0/8	AP12	4/0/8
ahip_bus17	29	50		83	42	4/1/12	AP9	3/0/7
ahip_bus18	26	53		82	41	4/2/16	AP11	4/1/12
ahip_bus19	27	52		81	39	4/3/20	AN8	3/1/11
ahip_bus20	18	71		72	24	4/4/24	AP13	4/5/28
ahip_bus21	19	70		71	23	4/5/28	AL11	3/5/27
ahip_bus22	16	73		68	21	4/6/32	AD16	4/6/32
ahip_bus23	17	72		67	20	4/7/36	AN12	3/6/31
ahip_bus24	14	76		66	18	4/8/40	AD17	4/7/36
ahip_bus25	15	74		65	17	4/9/44	AN11	3/7/35
ahip_bus26	12	78		64	15	4/10/48	AK14	4/8/40
ahip_bus27	13	77		63	14	4/11/52	AJ13	3/8/39
ahip_bus28	10	81		60	12	4/12/56	AK13	4/9/44
ahip_bus29	11	79		59	11	4/13/60	AJ14	3/9/43
ahip_bus30	8	84		58	9	4/14/64	AL16	4/10/48
ahip_bus31	9	82		57	8	4/15/68	AL13	3/10/47
ahip_req	0	95		48	104	1/clk/79	AM15	4/14/64
ahip_ack	1	94		47	106	1/clk/80	AM13	3/14/63
user0	56	209	49		93 (IO39)	2/1/12	AE13	1/13/59
user1	57	208	50		90 (IO40)	2/2/16	AH12	2/13/60
user2	54	213	51		89 (IO41)	2/3/20	AE12	1/12/55
user3	55	210	52		87 (IO42)	2/4/24	AH13	2/12/56
user4	42	228	67		68 (fsel0)	2/6/32	AN3	1/6/31
user5	43	226	68		65 (IO43)	2/7/36	AL10	2/6/32
user6	40	230	71		63 (IO44)	2/8/40	AM7	1/5/27
user7	41	229	72		62 (IO45)	2/9/44	AN6	2/5/28
user8	38	232	73		60 (IO46)	2/10/48	AM8	1/4/23
user9	39	231	74		59 (IO47)	2/11/52	AN7	2/4/24
user10	36	234	75		57 (IO48)	2/12/56	AJ8	1/3/19
user11	37	233	76		54 (fsel1)	2/13/60	AG12	2/3/20
user12	24	55		80	38 (ext_rst)	3/0/7	AN14	4/2/16
user13	25	54		79	35 (IO49)	3/3/19	AG13	3/2/15
user14	22	66		76	33 (IO50)	3/4/23	AN13	4/3/20
user15	23	65		75	30 (IO51)	3/5/27	AG14	3/3/19
user16	20	69		74	29 (IO52)	3/6/31	AP14	4/4/24
user17	21	67		73	27 (IO53)	3/7/35	AM11	3/4/23
user18	6	86		56	3 (IO54)	3/10/47	AL17	4/11/52
user19	7	85		55	120 (IO37)	3/11/51	AL12	3/11/51
user20	4	88		52	117 (IO38)	3/12/55	AJ17	4/12/56
user21	5	87		51			AF14	3/12/55
user22	2	93		50			AJ16	4/13/60
user23	3	92		49			AF15	3/13/59
user24	59	206	48				AJ11	2/14/64
user25	58	207	47				AM9	1/14/63

*Logic analyzer pins are given in the format *Pod/Bit/Pin*. For example, **ahip_bus4** is the 9th bit on the Odd side of Logic Analyzer 0 (or pod 1) of ADB0 which is pin 43 of the Logic Analyzer connection, therefore it reads 1/9/43.