Gisselquist
Technology, LLC

SDIO/EMMC CONTROLLER SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) ieee.org

May 21, 2024

Revision History

Rev.	Date	Author	Description
1.0	8/15/2023	Gisselquist	First release
1.9	2/24/2024	Gisselquist	Pre-release updates, with a new (optional) Wishbone DMA feature
1.91	4/25/2024	Gisselquist	New hardware reset feature added

Contents

Figures

Tables

Preface

After using the SDSPI companion controller for many years, I came across the need for an eMMC controller. Specifically, I needed a controller that could verify that two hardware interfaces were working. The first was a full SDIO interface, the second an eMMC interface,

Since the companion SDSPI IP already existed and was sufficiently low logic, this IP took on a different set of goals. First and foremost, it was to support the full SDIO interface–all of it. Every timing mode was to be supported, as were all commands. This includes the multiple block read and write commands not supported by the SDSPI driver. It does not include any of the low power UHS-II modes used by the newest cards of today. This controller was also built to support eMMC devices–to include both the data strobe and the eMMC device interrupt. Unlike the SDSPI controller, low logic was no longer one of my goals.

Dan Gisselquist, Ph.D.

1. Introduction

This SDIO/eMMC IP core is a redesign of the SDSPI IP, specifically focusing on full feature support. Low-logic, a key goal of the SDSPI companion IP, is no longer a goal. Key measures instead include full feature support and high device throughput. A full list of the features of this IP may be found in Chapt. 2.

For those interested in the basic architecture of how this controller is put together, Chapt. 3 describes its basic components and provides illustrations of how those components work together to accomplish the various actions required of the IP. Chapt. 3 will also discuss the components of the verification architecture.

I am aware that there is an official SDIO controller register set standard. This IP does not support that standard, but rather uses its own register set configuration. One reason for this is simplicity. The official register set standard appeared, upon inspection, more complicated than necessary. Second, since the SDSPI register set worked so well, there didn't seem to be a need to generate something new.

As a result, most of the registers from the SDSPI controller have been replicated here, save that they support SDIO control instead of SPI control. Like the SDSPI controller, this IP has CMD, DATA/ARG, and two FIFO registers. Unlike the SDSPI controller, a fifth register has been added for PHY configuration. This register directly exposes what once was the internal configuration register of the SDSPI controller. Finally, this IP now supports an optional internal DMA capability, and so three additional registers are now assigned to support both the DMA address and transfer length–registers not present in the original SDSPI controller. Chapt 4 discusses how all of these registers can be used to accomplish the basic operations required of any interaction with an external card/chip, while Chapt. 5 goes through the definition of each register in detail.

Finally, this is an Open Source project. The project was initially developed using a combination of external funding and internal research and development dollars. My intention is twofold. First, I intend to use this project to both teach others, as well as for blogging material. From this standpoint, the project exists as part of a portfolio of projects available for others to examine and learn from. My second goal is to use this as a background when providing services to my customers. My typical contract is for an hourly rate only, yet it comes with full access to any IP I have developed. As such, additional work on this IP may be done on a pay-per-hour basis. Where this project supports another project of mine, I will bill that project for any required maintenance. Likewise, if you find this IP almost meets your needs, then feel free to contract with me for any maintenance adjustments or upgrades you might need.

2.

Features

This IP has been designed from the ground up for full SDIO/eMMC feature support. This means that support has been built in for the following specific requirements:

- Full IO support
  - Support for both open-drain and push-pull IOs are available.
  - Both SDR and DDR modes are supported, for clocks up to 2x the system clock.
  - IP supports either 1, 4, or 8 bit bus widths
  - The eMMC data strobe is supported
- Supports both multiblock read and multiblock write commands.

  These are commands that were not supported by the SDSPI IP. They are supported here, at the cost of additional logic, since the goals of this project include both full feature support and high device throughput. Therefore, these features are supported at the cost of more logic.
- Ping-pong FIFO support, so one FIFO may be active on the bus while the other is active in a transaction.
- Internal CRC generation and checking, for both command and data interfaces
- Both RX and Command timeout support. Timeouts may be adjusted at build time.
- (Optional) Card detection support
- Integer clock division support starts at 2x the system clock rate, and can divide the clock all the way down to 1/1,000th of the system clock rate. Thus, for a 100 MHz system clock, clock frequencies supported include 200 MHz, 100 MHz, 50 MHz, 25 MHz, 12.5 MHz, 10 MHz, 8.3 MHz, etc., all the way down to 100 kHz.
- The SDIO clock may be stopped upon request, once all operations have come to a stop. This is useful when working with the multiblock read command, as the clock may be stopped while data is transferred out of the FIFOs. The feature can also be used to minimize emissions and power usage when the clock isn't in use.
- Special support for eMMC's *GO_IRQ_STATE* command. Receive timeout detection is suspended for this command. Further, it is possible to self-send a reply to this command to exit interrupt mode if the device doesn't generate an interrupt fast enough.

- Includes software driver support for use with FATFS
- Both Verilator C++ and all Verilog device verification models are provided
- While the IP is built for big-endian buses, optional little endian support exists
- All of the various IP modules, save for the front end itself, can be formally verified.
- 32-bit Wishbone interface.

  An optional AXI–Lite interface also exists.
- An optional DMA interface exists. This DMA interface can be used to transfer many blocks of data sequentially as part of a single command.

  At present, the DMA interface supports the Wishbone bus only.

Many, although not all, of these features have been tested as part of the 10Gb Ethernet switch KlusterLab project. It is in this project that the software has been tested and proven, and where its utility has been demonstrated.

2.1 Limitations

As of this current release, the IP is not as fully featured as I might like. Several details and features are missing and may (or may not) be developed in the near future as time, funding, and necessity require. The following list, therefore, outlines some of the key limitations of the controller as it exists today. It also provides a roadmap for future development.

- IO features not yet tested in hardware include: OSERDES support, DDR support, and data strobe support. To date, these features have only been tested either formally or in simulation or both.

  This is due to both the voltage requirements of these extra features, as well as the design of the circuit board I am currently testing with. High speed support requires 1.8V not 3.3V, and (worse) requires a voltage switch from 3.3V to 1.8V. Second, the voltage translator we are using limits speeds to 60MHz in order to maintain the ability to support open–drain IO. Similarly, wiring the clock pin to Xilinx's CCLK pin has limited the clock in the devices under test to a maximum of 50 MHz. Finally, success with the data strobe pin really requires that the data strobe be wired to a clock capable pin. I expect these hardware testing limitations will be addressed in the future with a better PCB design, but for the time being this statement outlines hardware testing success to date.

  Please check the hwteststat.png file in the ./ directory for the current status of any hardware testing and any updates on this issue.
- The verification model depends upon an internal memory buffer which can become quite large. Further, it doesn't integrate nicely with the FatFS system.

  Plans exist to replace the memory buffer in the SDIO VIP with a file backing. This should make it possible to model full sized SD cards.

- The included verification test suite doesn't (yet) support testing the software driver.

  A plan exists for integrating the ZipCPU into the verification environment for the purpose of verifying the software driver in simulation.

- There's no eMMC boot support at present.

  This also means there's no support for either CRC acknowledgement or negative acknowledgement tokens.

  Boot support may be added in the future, once I have a requirement for it. For now, I have no requirement for this capability.

- The model doesn't (yet) support all SDIO commands.

  As additional commands are required, they will be added. At present, however, the model is sufficient to verify IP startup and data transfer.

- Although the design has been tested and proven in hardware, not all of those hardware tests can currently be repeated in the included verification infrastructure.

3. Architecture

Four basic architectures are discussed in this chapter. The first is the hardware architecture. This will outline the various hardware components composing the design, and discuss each of their purposes. This is followed by a discussion of the optional DMA architecture component of the overall hardware architecture. The third is the verification architecture, used to test and verify the various components of the design. Finally, this chapter will conclude with a quick review of the software driver architecture that allows a single FATFS library to be able to interact with all three of the design components here: SDSPI, SDIO, and eMMC.

3.1 Controller Architecture

The basic SDIO/eMMC RTL architecture is shown in Fig. 3.1. The design is broken up into two parts at the top level: a hardware independent, all digital logic component whose top level is `sdio.v`, and a hardware dependent front end component whose top level is found in `sdfrontend.v`.

The core controller, found in `sdio.v`, itself is composed of five separate components. The first, `sdwb.v`, is the controller. This component is a wishbone slave. All user interaction goes through this controller. Requests for transfers are made here. Data to be transferred is stored here in one of two FIFOs for transmission to the front end. Data to be received is stored here after reception, to be read out as it becomes available to the CPU. Even when using the DMA, data will first be stored in the FIFO's within the controller before going to the card, or after coming from the card.

A second, alternative, controller implementation exists in `sdaxil.v`. This controller is nearly line for line identical to the `sdwb.v` controller, save that interaction with this controller takes place over AXI-Lite instead of Wishbone.

The controller maintains a PHY register for configuring the front end. This register also controls the clock generator, `sdckgen.v`. This clock generator is responsible for generating the clock used throughout the interface. That clock is internally represented as an 8-bit vector, allowing it to specify zero, one, or up to two clock periods, each either aligned with or 90 degrees offset from the data, per 8-bits. This clock may also be divided to slower speeds as requested by the user. Finally, a user setting allows the clock to turn off when it isn't being used, to save on interface power and to potentially reduce emissions. (Note that this only turns off the IO clock, not any of the system clocks used throughout the design to clock the logic within the design.) Turning off the clock may be required when working with the multiblock read command, to ensure that the FIFO is properly cleared before subsequent blocks are read.

Commands issued to the controller will be forwarded to the `sdcmd.v` module. This module is responsible for all interactions taking place over the CMD wire. The module can generate 48-bit commands, beginning with a start bit and ending with a CRC and a stop bit. It can then recognize

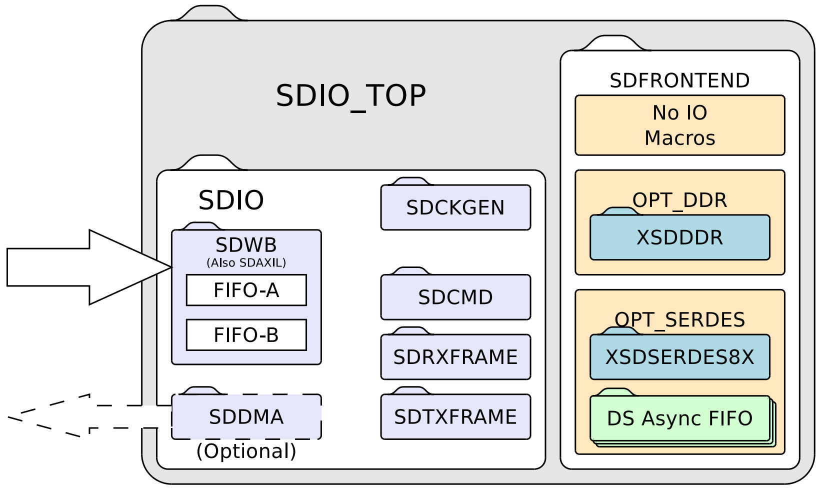

Figure 3.1: SDIO/eMMC Controller RTL Architecture

either 48-bit or 136-bit command responses. Responses are checked against both the CRC and frame errors. If the device fails to return a response within a given timeout window, the response may also timeout with an error, after which the command controller will no longer expect any response until the next command is sent.

Fig. 3.2 shows both the basic command flow, and the two possible response flows. For commands, a data register within `sdwb.v` is first set, then the command is issued. The `sdcmd.v` module organizes both into a 48-bit word to be transmitted. The `sdcmd.v` module then looks for one of two responses, either a 48–bit response which would place the 32-bit argument back in a data register, or a 136–bit response which would be placed into one of the two FIFOs in `sdwb.v`.

One type of controller command is a memory command. These commands send a 48-bit command to the device, followed by sending data to or receiving data from the device.

The `sdrxframe.v` component is activated when data is expected from the device on the data wires. It's purpose is to collect the data returned from the front end controller, and re-form that data into commands to write to one of the FIFOs in `sdwb.v`. If a full frame, as defined by the controller, is not received prior to a timeout, or if a CRC error is detected within that frame, the receive component will report an error and give up in order to wait for its next command.

Similarly, the `sdtxframe.v` command is activated when data is to be sent over the data wires to the device. In this case, data flows from one of the FIFOs in `sdwb.v` to `sdtxframe.v`, gets formatted for the front end to output, a start bit, CRC, and stop bit are added, and all get sent to the front end. This basic flow is shown in Fig. 3.3.

Let's now turn our attention to the device dependent front end.

The front end was initially defined for full IO support, all the way up to HS400 and the eMMC data strobe. Driven from a 100 MHz clock, this means that this front end was designed to support up to two device clocks per 100 MHz system clock, and to have data sent on each edge of these clocks. This forces a requirement of 4-data bits output per 100 MHz system clock cyle. The clock, however, also needs to be (potentially) offset by 90 degrees from that data. As a result, all I/O

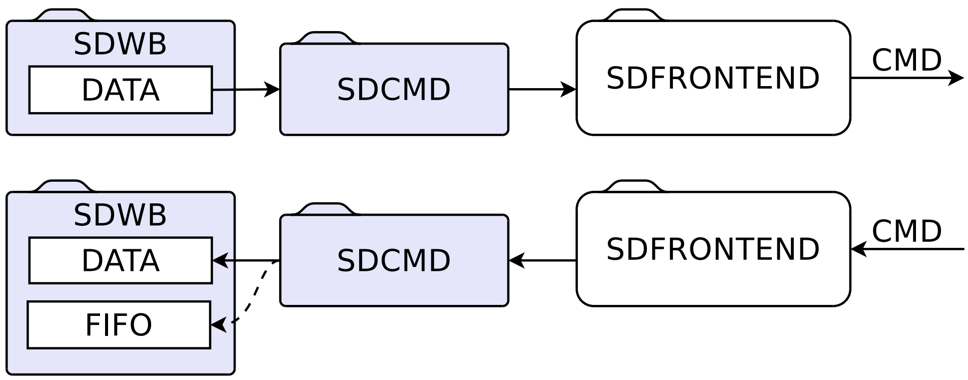

Figure 3.2: SDIO/eMMC CMD pin data flow

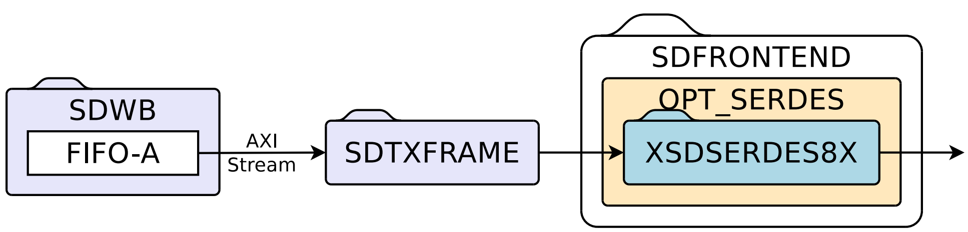

Figure 3.3: SDIO/eMMC transmit data flow, to the device

using this particular front end capability, called the OPT_SERDES capability in Fig. 3.1, takes place using Xilinx's 8:1 OSERDES elements. Incoming bits are either sent to an asynchronous FIFO, to support the eMMC data strobe, or they are sampled via 1:8 ISERDES elements. In the case of the ISERDES inputs, sample timing is controlled via both the output clock rate, as well as a user controlled digital delay, allowing sample alignment with clock precision of one eighth of a system clock. Hence, even without the full speed HS200 support, this subsample re-alignment may still be quite valuable when supporting higher speeds. All Xilinx specific SERDES interactions are captured in an xsdserdes8x.v module.

Logic resources throughout the design may be reduced, however, by switching to a simpler front end architecture. Two other architectures are available: the OPT_DDR architecture and a plain architecture that doesn't use any IO macros–save for the IOBUF macro used for controlling any tristate capabilities.

The OPT_DDR architecture is roughly the same as the OPT_SERDES architecture, save the DDR IO elements are used to generate both clock and data instead of the SERDES elements. This should make it possible to drive the IO pins at the full clock speed using SDR, or at half the clock speed using DDR. In this case, all Xilinx specific IDDR and ODDR interactions are captured in an xsdddr.v module.

The no macro architecture was built in order to support an architecture where the SD_CK pin was routed through the Xilinx CCK pin, and hence through an STARTUPE2 primitive. In this case, no IO macros are available for controlling the clock, and all IOs are controlled directly from logic. The fastest clock this architecture can generate is half the system clock rate.

These constitute the basic components of the controller, both hardware independent and hardware dependent.

3.2 DMA Architecture

The DMA is built from several parts. The first key component to the DMA is found in the control module, sdwb.v. The controller is responsible for sectioning transfers into per-block requests, and forwarding data to and from the DMA module itself. The rest of the DMA logic is found in the DMA module, sddma.v. This module is shown in Fig. 3.4.

When using the DMA, data continues to go through the FIFOs in the controller module. The difference is that these FIFOs will be automatically sourced from, or emptied into, an external DMA module which will then handle copies from, or to, memory. Hence, for a write operation, the user will request the write operation and then the controller will tell the DMA to supply data for one of its FIFOs. Once the FIFO is filled, the controller will send it to the external device. Then, while the data is being transmitted to the device, the controller will continue to request source data capture to supply any subsequent data transfer requirements. Likewise, for a read operation, the user will request the operation first, followed by a block getting transferred to the controller's FIFO. The read will then continue filling the second FIFO will the first one is drained by the DMA. If both FIFO's fill faster than the DMA can empty them, then the IO clock will be stopped to wait for the next FIFO to be drained.

That's a high level overview, from the perspective of a DMA enabled control module. Let's now turn our attention to how the DMA supplies and deals with the data streams given to it.

Data arrives in the DMA module from one of three sources. First, if both OPT_ISTREAM and the MSB of the DMA address are set, data may arrive from an AXI stream. If used, this most-significant

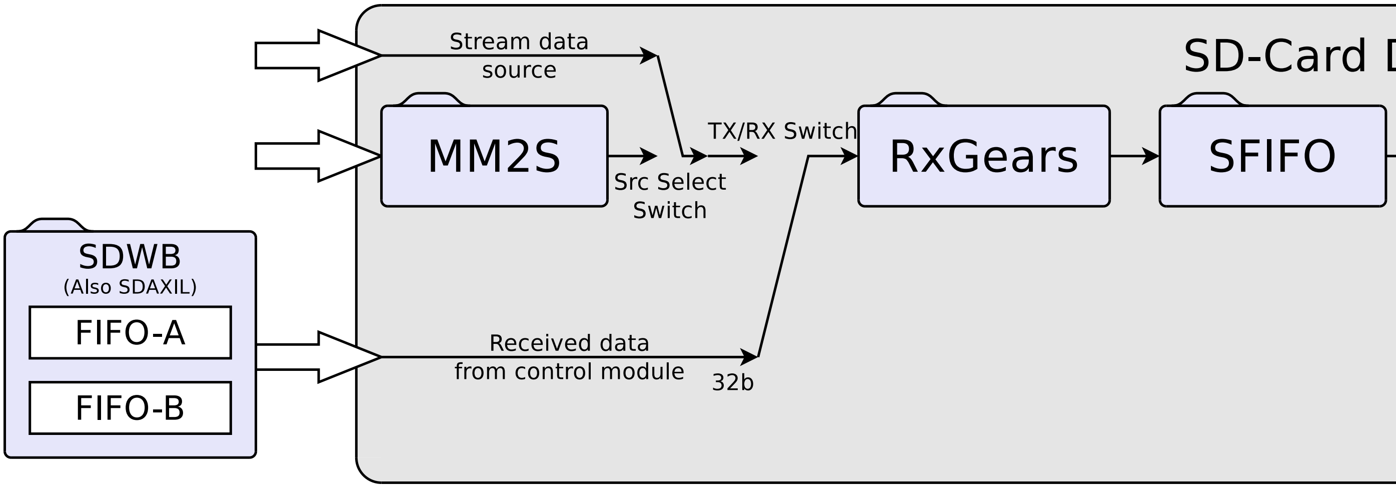

Figure 3.4: DMA Block Diagram

bit is one wider than the address width of the devices bus. Alternatively, data may be read from memory using the sddma_zipmms.v component. Finally, when reading from the card, data will arrive first in the controller module, and then be forwaded from it once the respective buffer is full.

These sources may have different data widths. Data arriving from memory will arrive packed in bits that may be up to the full bus width in size. Data arriving from the controller will be 32b wide. Data arriving from the stream port may be either one of these widths. The receiving gearbox, sddma_rxgears.v, will therefore take this data stream and pack it into beats at the widest of these widths.

From here, data enters into a FIFO. This guarantees that the memory source never needs to stall, or otherwise provide backpressure on the bus–something not supported by the Wishbone interface. The FIFO also allows data to arrive at wide widths, only to be read at slower (32b) rates if necessary.

From the FIFO, the data width is adjusted in the transmit, or outgoing, gearbox, sddma_txgears.v. This component optionally adjusts the data width back down to 32b if required. Either the stream output, or the controller interface may require this width. If going back to memory, the width will be left at the full size of the bus.

The final step in the DMA process is to forward the data stream to one of three ultimate destinations. When writing to the card, the ultimate destination will send the data to the controller. When reading from the card, data may either be written to memory or to the AXI stream interface. The AXI stream interface will be used when/if both OPT_OSTREAM is set and the most significant address bit are set.

A small amount of logic exists in the sddma.v module to control the switching between sources and outputs, as well as to make sure block boundaries are respected.

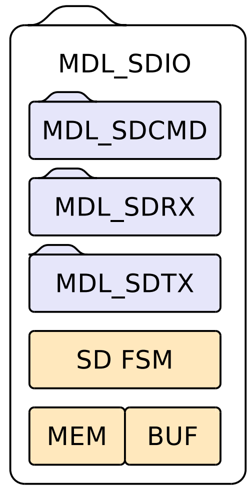

Figure 3.5: SDIO Verification IP

3.3 Verification Architecture

The verification architecutre is built around a model of the downstream device. One such model is shown in Fig. 3.5. A second, similar, Verilog model exists for modeling the eMMC interface. These Verification IP modules contains full featured submodules for handling the command wire, receiving data, and transmitting data. These three submodules are designed to correspond to their host controller counterparts. Further, these three submodules have been built to be independent of whether the controller is modeling an SDIO or an eMMC interface.

The smarts of this Verfication IP (VIP) are found in the finite state machine (FSM) at the top level of `mdl_sdio.v`. This state machine responds to commands, and generates responses to them. A data transfer buffer exists for sending or receiving data across the interface. A second memory within the VIP, shown as `MEM` in Fig. 3.5, captures the persistent memory found in the SDIO (or eMMC) device.

The entire Verilog test bench infrastructure is diagrammed in Fig. 3.6. In this setup, a test script drives a Wishbone Bus Functional Model (BFM). The BFM issues 32b bus requests, which are then resized to the full width of the bus. From here, requests go into a crossbar, and get routed to either an SDIO (via a width downsizer), an eMMC controller (via a width downsizer), or a block RAM model (at the full bus width). This then allows the test script to interact with the IP, which then drives interactions with one of the two backend VIP device models. If the DMA's are present, they can also be used to transfer data to or from the RAM via the bus interconnect. What tests get accomplished at this point is then dependent upon the test script. Two test libraries are also available, one for SD cards and one for eMMC cards. These libraries help define common constants and supporting common test script tasks which can be called by the test script itself.

Two versions of this top-level interface exist. The first supports all Wishbone interactions. It can be found in `tb_wb.v`. A second top level exists for testing AXI interactions. This is found in `tb_axi.v`.

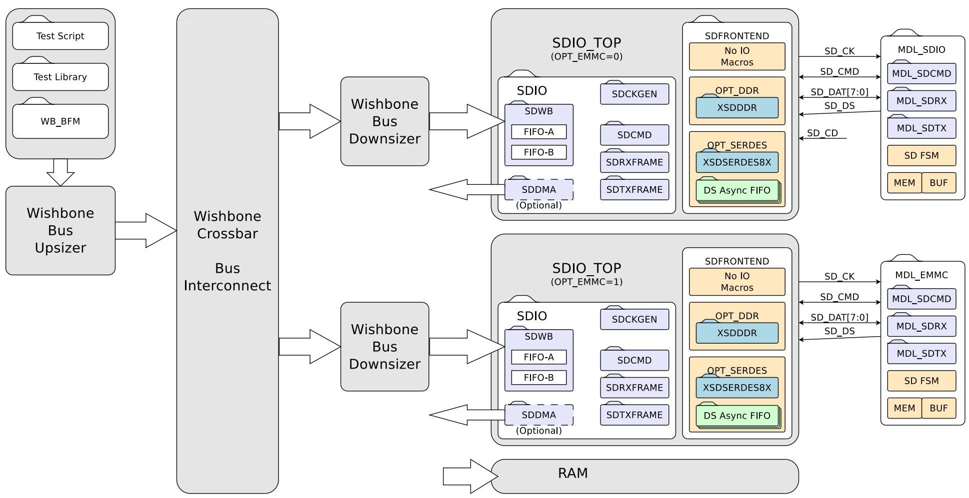

Figure 3.6: Test-bench infrastructure block diagram

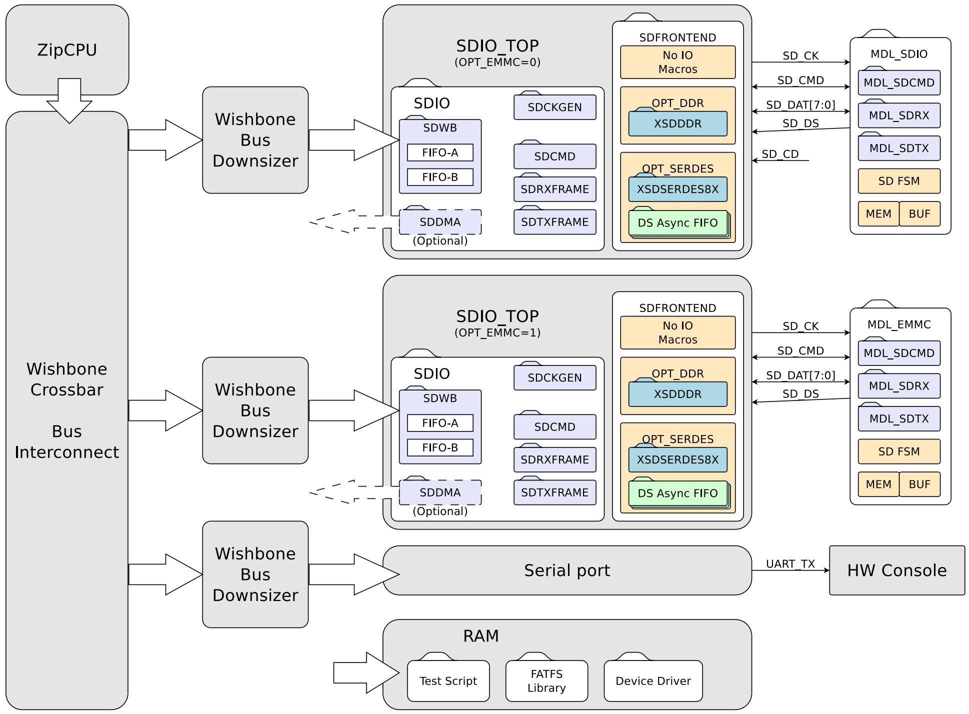

Figure 3.7: All Verilog top-level model

A third test bench infrastructure, shown in Fig. 3.7, is also envisioned. This infrastructure will contain a soft–core ZipCPU, a serial console, and memory. In this test architecture, the test script exists as a software program supported by both the FATFS library and the software driver. Once complete, the purpose of this testbench infrastructure will be to specifically test and verify the software driver in its native (software) form, compiled for and then running on a soft–core CPU. In this case, that softcore CPU will be the ZipCPU.

3.4 Software Driver Architecture

At present, there exist three software drivers and one glue layer for each. The three drivers are found in the `sw/` directory. These respective drivers are:

- `sdspidrv.c`: A software driver for the SPI version of this controller. The SPI version of this controller is described in the SDSPI user guide.
- `sdiodrv.c`: A software driver for the SDIO controller, described in this user guide. This controller is designed to interact with an SD Card.
- `emmcdrv.c`: A software driver for the eMMC controller. This is designed to interact with the same SDIO/eMMC controller discussed in this user guide. However, the SDIO and eMMC

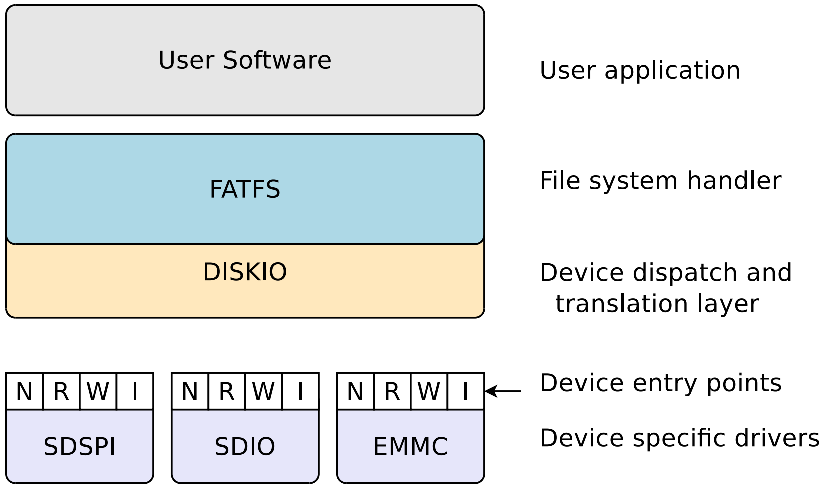

Figure 3.8: Software Stack

protocols require different commands to interact with their respective devices. As a result, in spite of any similarities, the eMMC software driver differs significantly from the SDIO software driver.

- `diskiodrvr.h` provides a common definition of a device "driver", which can be used across all of the device drivers listed above.

  More on this in a moment.

- `diskio.c` is a shared/common dispatch glue layer. Commands from the FATFS subsystem naturally land here, where they are then dispatched to the appropriate controller.

Put together, this IO subsystem has the various layers shown in Fig. 3.8. User software interacts with the FATFS layer. FATFS then interacts with the device dispatch layer. The device dispatch layer then instantiates and interacts with one (or more) of the device drivers beneath it.

Each device driver has four entry points–"methods" in Object Oriented terminology. The first method is used to initialize the driver and set up its data structures. After that, there are methods for reading and writing. Finally, there's an IOCTL method which can be used for other tasks such as querying the size of the device or the size of a data block.

4.

Operation

At its most basic level, interacting with the controller simply involves configuring the PHY, setting the data you wish to send, and then sending a command to the controller. This needs to take place, however, in many different contexts. Some commands don't expect responses, whereas others return 48 or even 136–bits. Some commands are followed by data transmission, whereas others do not. This section therefore walks through several examples of commands, illustrating the range of commands and responses that can be issued using this controller.

4.1 Initial PHY setup

The first step to any operation is to make sure that the PHY register is appropriately set up for the clock speed desired, the transfer rate desired, and whether or not the design will be using open–drain or push–pull IOs. Any changes made to this register will be applied immediately, save for any changes to the clock rate. Clock rate changes will take place starting at the top of the next clock cycle. Hence, once a clock rate change has been made, it makes sense to wait until the current clock matches the requested clock rate.

The following, for example, will set up a 100 kHz clock rate using open–drain IOs and 512 byte sector sizes. This is appropriate for beginning communication with the device.

```
const   SDIOCK_100KHZ = 0x000000fc,
        SPEED_SLOW    = SDIOCK_100KHZ,
        SPEED_512B    = 0x09000000;

_sdio->sd_phy = SPEED_SLOW | SECTOR_512B;
while(SPEED_SLOW != (_sdio->sd_phy & 0x0ff))
        ;
```

4.2 GO IDLE: Commands that don't get responses

Sending a command to the device is as easy as setting the command's 32'bit argument, and then writing the appropriate command to the command register. All command writes set bits [7:6] to 2'b01. Non-command writes will set these bits to something different. At the same time you write a command to the controller, you'll also want to tell the controller what type of response to expect–whether to expect no response at all, an 48-bit (R1) response, or a 136-bit (R2) response.

When issuing a command, you also want to be aware of the error bit. If the error bit gets set, no commands will be accepted until the error bit is cleared. Hence, you'll want to clear the error bit at the beginning of any command sequence, and check it when you are done to see where any error took place.

Finally, in the case of the first command used when interacting with a device, you'll want to clear the controller's card-removed bit. You can then check this bit later, at your convenience, to know if the card has ever been removed and (if so) if you need to start your initialization sequence again from the `SEND_GO_IDLE` command.

Put together, this would look like:

```
const   SDIO_REMOVED = 0x00040000,
        SDIO_ERR     = 0x00008000,
        SDIO_RNONE   = 0x00000000,
        SDIO_CMD     = 0x00000040;

_sdio->sd_data = 0;
_sdio->sd_cmd  = SDIO_REMOVED | SDIO_CMD | SDIO_RNONE | SDIO_ERR;
```

The next step is to wait for the command to complete. This can be done either by polling the controller,

```
const   SDIO_BUSY    = 0x00106800;

unsigned        st;

st = _sdio->sd_cmd;
while(st & SDIO_BUSY)
        st = _sdio->sd_cmd;
```

or by waiting for a controller interrupt.

```
// SDIO_INT is the interrupt number assigned to the
// design.  It's set external to the controller.
unsigned        st;

st = _sdio->sd_cmd;
while(st & SDIO_BUSY) {
        CLEAR_INT(SDIO_INT);    // Clear the interrupt signal
        WAIT_INT(SDIO_INT);     // Wait for the given interrupt
}
```

4.3 SEND IF COND: R1 commands

The next type of command the controller understands are what are called R1 commands. These are commands that expect a 48-bit response–whether or not that response truly contains an R1 register or any other register.

The first R1 command in the startup script is the SEND IF COND command, also known as CMD8. This command requires a special argument. We'll allow that argument to arrive via the variable `ifcond`. This variable is first placed in the `sd_data` register, and then the command is issued.

```
const   SDIO_R1      = 0x00000100,
        SDIO_READREG = SDIO_R1 | SDIO_CMD;

_sdio->sd_data = ifcond;
_sdio->sd_cmd = SDIO_READREG+8;

wait_while_busy();
```

Once the command returns, the first 8-bits of the 48-bit return may be read from the `sd_cmd` register. If the protocol has been followed and the device has returned a value, then bits [7:6] of this register will now be zero. The next 32-bits may be read from the `sd_data` register, to know how the device has responded. The last 8-bits, both CRC and stop bit, are not returned by the controller. Instead, the error bit and error code may be checked to know if the CRC is correct and the stop bit valid.

4.4 ALL SEND CID: R2 commands

The last type of command is the R2 command. These commands read 136-bits from the device. The first 8-bits of these are read into the command register, as with the R1 commands. However, the next 128-bits will not fit into the data register. Instead, they are returned into one of the two FIFOs.

Sending such a command follows the same script as before. The difference is that we could now specify, in our command, which FIFO to write the results into. We'll use FIFO A, the first FIFO, which is the default for this example.

```
const   SDIO_R2      = 0x00000200,
        SDIO_READREG = SDIO_R2 | SDIO_CMD;

_sdio->sd_data = 0;
_sdio->sd_cmd = (SDIO_ERR|SDIO_READR2) + 2;

wait_while_busy();
```

If this particular command works, it is likely to set the CRC error bit. This is normal for a CMD2, although not so normal for other commands.

Once the command completes, the CID register can be read from the FIFO 32-bits at a time.

```
unsigned        d_CID[4];

d_CID[0] = _sdio->sd_fifa;
d_CID[1] = _sdio->sd_fifa;
d_CID[2] = _sdio->sd_fifa;
d_CID[3] = _sdio->sd_fifa;
```

If the controller has been configured for big–endian operation, then the MSB byte of each FIFO result will be the first byte read from the device.

The `d_CID` values may then be decoded and processed by the software driver.

4.5 Adjusting the PHY

Once any cards have been properly enumerated, the PHY may be adjusted for higher speed. The first speed change below simply changes the clock speed to 25MHz, and the IOs from open–drain to push–pull.

```
const   SDIOCK_25MHZ = 0x00000003,
        SPEED_512B   = 0x09000000;

_sdio->sd_phy = SECTOR_512B | SDIOCK_25MHZ | SDIO_PUSHPULL;
```

Once we request a clock change, we may then need to wait a while. Specifically, the clock will only change in between clock periods. Hence, if the clock is set for 100 kHz operation, we may need to wait up to 10 μs for the clock to switch. We can tell when the switch is complete by reading back the lower 8-bits from the PHY register to know the currently set clock speed.

```
// Wait for the clock change
while(SDIOCK_25MHz != (_sdio->sd_phy & 0x0ff))
        ;
```

A later PHY adjustment might even include switching to 4–bit data mode.

```
const   SDIO_W4 = 0x00000400;

_sdio->sd_phy = SECTOR_512B | SDIOCK_25MHZ | SDIO_PUSHPULL
                        | SDIO_W4;
```

In this final example, we don't need to wait for the change to take place. The interface was already idle, so it took place immediately.

4.6 Reading a sector

Once the SD card has been set up, the next step may be to read a sector of data. In general, this is as simple as enabling the FIFO when issuing the command, and then reading the data from it once the command has been complete.

The SDIO driver first, however, checks whether or not the card has been removed. This is appropriate, lest the card has been swapped unbeknownst to the controller.

```
if  _sdio->sd_cmd & SDIO_REMOVED)
        return CARD_REMOVED_ERR;
```

If the card has not been removed, then we can issue a command to read a block. This involves setting the `SDIO_MEM` bit and (potentially) the FIFO bit to select which FIFO we are interested in.

```
const   SDIO_MEM     = 0x00000800,
        SDIO_READBLK = (SDIO_CMD|SDIO_R1|SDIO_MEM)+17;

_sdio->sd_data = sector;        // Assumes a high capacity card
_sdio->sd_cmd = SDIO_READBLK;
```

```
        wait_while_busy();
```

Once the read completes, we can check for any errors.

Assuming the read completed without error, we can then read the data back out of the controller.

```
        for(int k=0; k<512/sizeof(uint32_t); k++)
                buf[k] = _sdio->sd_fifa;
```

Note that the FIFO's are always 32-bit aligned. Whether or not the buffer is aligned, or whether or not the CPU supports unaigned accesses in the case where it isn't, aren't controller issues.

If no DMA is used, it will likely cost the CPU 128 memory load cycles, followed by another 128 memory store cycles to read this data out.

The ZipCPU's DMA may also be used to read from this controller if desired for higher speed. In this case, the read will cost 128 clock cycles (plus latency) to read the memory out of the controller, and another cycles equivalent to 512 divided by the bus width to write the result to memory.

4.7 Writing a sector

Writing a sector to the device is very similar to reading, however, there are a few minor key differences. The first key difference is that the memory needs to be loaded into the FIFO first, before it can be sent to the device.

```
        for(int k=0; k<512/sizeof(uint32_t); k++)
                _sdio->sd_fifa = buf[k];
```

The second key difference is that the controller doesn't know when to write to the device or when to read from it. In general, the controller doesn't know the meanings of any of the commands. Therefore, if a command needs to use the FIFOs, the controller needs to be told. Likewise, if the data is supposed to flow from the FIFO to the device instead of from the device to the FIFO, the controller also needs to be told. For this reason, a separate bit needs to be set in addition to the previous ones–one to indicate that this is a write and not a read.

```
        const   SDIO_WRITE    = 0x00000400,
                SDIO_WRITEBLK = (SDIO_CMD | SDIO_R1 | SDIO_ERR
                                | SDIO_WRITE | SDIO_MEM) + 24;

        _sdio->sd_data = sector;        // Assumes a high density card
        _sdio->sd_cmd  = SDIO_WRITEBLK;

        wait_while_busy();
```

4.8 Writing multiple sectors

Writing multiple sectors to the device is very similar to writing a single sector. Indeed, the instructions for writing the first sector are identical, save that the command is now a CMD25 instead of a CMD24. Things become different when it comes to writing subsequent sectors, and then when it comes to ending transmission.

Let's walk through these changes one by one.

As before, we still have to read our data into the FIFO before we can issue the write command.

```
	s = 0;

	unsigned *src = &buf[s*512];

	for(int w=0; w<512/sizeof(uint32_t); w++)
		_sdio->sd_fifa = src[w];
```

We then issue a command 25 to write multiple blocks to the device.

```
	const	SDIO_FIFO	= 0x00001000;

	_sdio->sd_data = sector;
	_sdio->sd_cmd  = (SDIO_CMD | SDIO_R1 | SDIO_ERR
				| SDIO_WRITE | SDIO_MEM) + 25;
```

Then we come to our first change: we don't wait yet for the transfer to complete. Instead, we load the next sector into the second FIFO. In general, we'll alternate which FIFO we use.

```
	for(s=1; s<sector_count; s++) {
		unsigned *src = &buf[s*512];

		if (s&1) {
			for(int w=0; w<512/sizeof(uint32_t); w++)
				_sdio->sd_fifb = src[w];
		} else {
			for(int w=0; w<512/sizeof(uint32_t); w++)
				_sdio->sd_fifa = src[w];
		}
```

Once the second sector has been transferred to FIFO memory, we can then wait for the current write transaction to complete and issue a new write request for the next sector.

```
		while(_sdio->sd_cmd & SDIO_BUSY)
			;

		_sdio->sd_cmd = (SDIO_WRITE | SDIO_MEM)
				+ ((s & 1) ? SDIO_FIFO : 0)
```

At this point, we repeat until we've written all of the sectors we wish to write to. Once we are done, we can send a command to STOP_TRANSMISSION, expecting an R1 response.

```
	// Send a STOP_TRANSMISSION request
	_sdio->sd_data = (RCA << 16);	// EMMC requires the RCA here
	_sdio->sd_cmd = (SDIO_CMD | SDIO_R1 | SDIO_ERR) + 12;
```

4.9 Reading multiple sectors

Reading multiple sectors is almost identical to writing multiple sectors, save that the operation proceeds in reverse: we issue the read command first, and then copy data out of the FIFOs second.

First, the command is issued to read multiple sectors, given a starting sector.

```
        _sdio->sd_data = sector;        // Assumes a high capacity card
        _sdio->sd_cmd = (SDIO_CMD|SDIO_R1|SDIO_MEM) + 18;
```

We then wait for that command to complete.

```
        while(_sdio->sd_cmd & SDIO_BUSY)
                ;
```

Once complete, we immediately switch FIFOs and issue a command to store the read data from the next sector into the next FIFO. This is also a good time to check for any read errors that may have taken place while reading the last sector.

```
        for(s=1; !err && s<sector_count; s++) {
                while(_sdio->sd_cmd & SDIO_BUSY)
                        ;

                if (0 != _sdio->sd_cmd & SDIO_ERR)
                        err = 1; // No more commands to issue on error
                if (s + 1 < sector_count && !err) {
                        // Immediately start the next request
                        _sdio->sd_cmd = SDIO_MEM + ((s&1) ? 0:SDIO_FIFO) + 18;
                }
```

This second command must be issued before the device starts sending the second sector of data.

Once the second read request is away, we have until it finishes to copy the data out of the first FIFO.

```
                unsigned *dst = &buf[s*512];

                if (s&1) {
                        for(int w=0; w<512/sizeof(uint32_t); w++)
                                dst[w] = _sdio->sd_fifa;
                } else {
                        for(int w=0; w<512/sizeof(uint32_t); w++)
                                dst[w] = _sdio->sd_fifb;
                }
```

These operations are time sensitive, and could create a race condition. Specifically, the host must request the next sector before the SD card starts to send it.

It is recommended to reduce this time pressure by setting the clock shutdown bit in the PHY. If this bit is set, the SDIO/eMMC clock will stop toggling when not in use. This should allow the software driver enough time to finish copying out any data, and to issue a second read command before the data starts showing up.[1]

[1] There is an unwritten requirement here, however, that the card must provide at least a break of a couple of system clocks plus at least one SDIO clock in order for this clock stop to happen in time.

Once all transfers have completed, a stop transmission command needs to be sent for the card to stop sending more data.

```
        while(_sdio->sd_cmd & SDIO_BUSY)
                ;

        // Send a STOP_TRANSMISSION request
        _sdio->sd_data = (RCA << 16);   // EMMC requires the RCA here
        _sdio->sd_cmd = (SDIO_CMD | SDIO_R1 | SDIO_ERR) + 12;
```

The software driver can then copy the last sector out of the FIFO.

```
        unsigned *dst = &buf[s*512];

        if (s&1) {
                for(int w=0; w<512/sizeof(uint32_t); w++)
                        dst[w] = _sdio->sd_fifa;
        } else {
                for(int w=0; w<512/sizeof(uint32_t); w++)
                        dst[w] = _sdio->sd_fifb;
        }
```

Do note that we are reading the FIFOs in reverse. When `s==0`, we request FIFO zero. However, when we come back to read from that FIFO `s` will have advanced, so it will look like we are reading opposite of what we have requested even though this isn't the case.

4.10 Using the DMA to write multiple sectors

It's even easier to write multiple sectors using the DMA. In this case, you only need to make one request and then wait for it to complete.

```
        _sdio->sd_dma_address = (char *)src;
        _sdio->sd_dma_length = sector_count;
        _sdio->sd_data = sector;        // Assumes a high capacity card
        _sdio->sd_cmd  = (SDIO_CMD | SDIO_R1 | SDIO_ERR | SDIO_DMA
                                | SDIO_WRITE | SDIO_MEM) + 25;
```

All that's required next is to wait for the operation to complete.

```
        while(_sdio->sd_cmd & SDIO_BUSY)
                ;
```

There's no need to send any `STOP_TRANSMISSION` command, since the DMA will send it automatically once all sector data has been transferred to the card.

Unlike using an external DMA, there's no requirement to first read from the bus before writing the data back to the bus. Hence, the number of clock cycles required for a single sector of this write operation will be given by 512 divided by the bus width, times any throughput delays, plus the bus/memory latency. This will be nearly twice as fast as when using an external DMA, such as the ZipCPU's DMA.

4.11 Reading multiple sectors with the DMA

As with writing to the SD card using the DMA, reading from the card is just a matter of setting four configuration registers and then waiting.

```
const   SDIO_DMA  = 0x00002000;

_sdio->sd_dma_address = (char *)dst;
_sdio->sd_dma_length = sector_count;
_sdio->sd_data = sector;
_sdio->sd_cmd  = (SDIO_CMD | SDIO_R1 | SDIO_ERR | SDIO_DMA) + 18;
```

As with DMA writing, all that's required next is to wait for the operation to complete.

```
while(_sdio->sd_cmd & SDIO_BUSY)
        ;
```

Again, there's no need to send any `STOP_TRANSMISSION` command, since the DMA will send it automatically once all sector data has been transferred to the card.

Unlike the multisector read command above, there's no reason to adjust the clock stop bit of the PHY register. When using the DMA to read, the clock may be stopped automatically as necessary between commands.

4.12 GO IRQ STATE

The one command that the IP knows by number is the GO IRQ State command, CMD40. This command needs to be treated special, because it has no timeout. It also needs to be treated special, so that the controller can end the command if necessary.

The first step in the GO IRQ STATE command is to switch to an open–drain IO mode. This will likely also mean that we need to switch clock speeds as well.

```
_sdio->sd_phy = (_sdio->sd_phy & ~(SDIO_PUSHPULL | 0x0ff))
                | SDIOCK_25MHZ;
```

We can now issue the GO IRQ STATE command.

```
_sdio->sd_data = 0;
_sdio->sd_cmd  = SDIO_READREG + 40;
```

At this point, we need to wait for either an SDIO controller interrupt, indicating that the SDIO command has completed and an R1 response has been read, or a timeout indicating something has gone wrong and we don't wish to wait any longer.

If the response has been read, then the command is complete and we are done. We can clean up by restoring the PHY to its original mode. No other actions are required.

If no response has been read and the software decides not to wait any longer, then we need to create a self–reply to this command.

```
_sdio->sd_data = 0;
_sdio->sd_cmd  = 40;
```

Once this command completes, we are done and can return the PHY to its original mode. No valid data will be returned from this command.

There exists the possibility that this command will step ontop of an actual return from the eMMC card. Electrically, this should be okay since we've returned to open drain mode. If the parameter `OPT_COLLISION` is set, then the front end will detect a collision and automatically back off. If not, the collision may result in a command response with a bad CRC. Either way, the card will leave its IRQ state.

5. Registers

As mentioned in the last two chapters, the SDIO IP supports five registers, and three additional DMA–only registers. These are shown in Tbl. 5.1. Of these registers, the command register, CMD,

Name	Address	Width	Access	Description
CMD	`0x00`	32	R/W	SDSPI Command and status register
ARG	`0x01`	32	R/W	SDSPI return data/argument register
FIFO[0]	`0x02`	32	R/W	FIFO[0] data
FIFO[1]	`0x03`	32	R/W	FIFO[1] data
PHY	`0x04`	32	R/W	Front end configuration register
DMA Addr	`0x05`	32	R/W	Reserved for DMA address bits[31:0]
DMA Addr	`0x06`	32	R/W	Reserved for DMA address bits[63:32]
DMA LEN	`0x07`	32	R/W	Reserved for DMA transfer length

Table 5.1: I/O Peripheral Registers

initiates and provides feedback from all commands to/from the external device, so we'll spend most of our time discussing it.

5.1 CMD Register

Only writes to the CMD register will initiate interactions with the external device. As a result, it is central to how the entire IP operates. The CMD register itself is composed of several packed bit fields, as shown in Fig. 5.1. The various fields of this register are explained in Tbls. 5.2 and 5.3.

Starting at the top, any time $\mathrm{R_{ST}}$ is set, the hardware reset pin output will be active (low). If set, the hardware reset will be set for a minimum of $100\mu s$. This pin must be cleared manually, allowing the external reset pin to be controlled by the CPU. While in reset, other command requests will be ignored.

31 30 29 28 27 26	25	24	23	22	21	20	19	18	17 16	15	14	13	12	11	10	9 8	7	6 5 4 3 2 1 0
(Rsrvd)	$\mathrm{R_{ST}}$	$\mathrm{B_E}$	C_r	E_r	E_c	C_B	P	R	C_C	E	B	D	I	M	W	T		CMD

Figure 5.1: CMD Register fields

Name	Width	Description
R_{ST}	1	Hardware reset. Set to one to assert (lower) the `o_hwreset_n` wire. Internal logic will insist on a minimum $1\mu s$ (i.e. 100 clock) reset duration. This feature is only enabled if `OPT_HWRESET` is set.
D_E	1	DMA Bus Error. True if the DMA encounters a bus error while transferring data. This will also be set if the DMA address wraps during the operation.
C_r	1	Receive error code. On any receive error, indicates the cause of the error.
E_r	1	Receive error. Set if the E bit is set due to a data reception error.
E_c	1	Command error. Set if the E bit is set due to an error in command wire processing.
C_B	1	Card is busy. This is a separate indication from whether or not the controller is busy. Specifically, certain commands expect R1b R1b responses. Following one of these commands, the card may assert a busy signal by lowering DAT[0]. If it does, this bit will be set until the card releases DAT[0].
P	1	No card present. Set if the card detect bit indicates no card is currently present. This bit will automatically clear when a card is inserted. This facility is only available if the `OPT_CARD_DETECT` parameter set to enable it.
R	1	Card removed. This bit will be set if both the `OPT_CARD_DETECT` parameter has been set, and the card detected bit has dropped since a card was last acknowledged. This bit will only be cleared by the user writing a one to this bit to clear it. As a result, it can be used to detect if a card has been inserted and thus needs to be set up.
C_C	2	Command error code. Indicates the type of error, if any, received from the last command. This is separate from any potential bus or receive errors.
E	1	Error. The last command or data receipt completed with error. Once set, further commands will be ignored until this bit is cleared. Write a one to this bit to clear it. Errors may be cleared concurrent with new transactions requests.
B	1	Busy. Either a command or data transfer is currently in progress.
D	1	DMA Enable. Set this bit to enable a DMA operation as part of an issued command.
I	1	FIFO selection. Keep clear to use FIFO A. Set to use FIFO B. When using the DMA, the DMA will begin with the selected FIFO, but then alternate between FIFOs on a block by block basis.

Table 5.2: CMD Register field descriptions, part one

Name	Width	Description
M	1	Memory operation. If set, the requested command also includes a data transfer to or from the FIFOs. If bit[6] is also set, this transfer will take place once the assicated command is complete. Otherwise, if bit[6] is clear, the memory operation will take place without an associated prior command and response. This is useful for supporting multi–block transfers.
W	1	Write operation. If this bit is set for a memory operation, then data will be transferred from a FIFO to the device. Otherwise, and memory operation is a read operation, which will transfer data from the device to the FIFO.
T	1	Response type. Indicates whether a response is expected, and whether or not that response will have 48'bits or 136'bits. See Tbl. 5.5 for more info.
CMD	8	The command. Writes with bits [7:6] set to 2'b01 will forward the other six bits of the command to the external device. Writes with bit [7] set can be used to communicate with the controller without going to the device as well. Writes with both bits clear and the memory request bit set can be used to make requests of the memory channel without also requesting a preceding command transfer. Once any requested command transfer completes, these 8-bits will be replaced by the first 8–bits of the reply. Most such replies will typically just clear bit [6].

Table 5.3: CMD Register field descriptions, part two

D	M	R2	[7:6]	Description
?	0	0	2'b00	If `OPT_EMMC` is set, a command response will be sent. This is useful for ending an IRQ waiting period if the IRQ hasn't taken place.
1	?	0	2'b00	No command will be sent, but a DMA operation will begin.
?	1	0	2'b00	No command will be sent, but a memory operation will begin.
?	?	?	2'b01	A command will be sent to the card.
1	?	0	2'b01	The command will be followed by a DMA transaction.
0	1	0	2'b01	The command will be followed by a memory transaction using the FIFOs.
0	0	1	2'b01	The command response will be placed into the selected FIFO.
?	?	?	2'b1x	No commands will issue, nor DMA nor memory operations commence. Writes of this type may be used to reset the user FIFO position, to clear any error flags, or card removed bit(s)
`32'h5200_0000`				Soft reset request. This will resets the controller internally. Any ongoing operations will be aborted. No indication of this reset will be sent to the external card.

Table 5.4: Types of writes to the CMD register

Commands are issued to the device by writing to this register with bits [7:6] set to 2'b01 and the reset bit is clear. There are some exceptions to this rule:

- If the error bit is set, then any command that doesn't also set bit 15 (clear error) will be ignored.
- If the device is already busy with another command, then any new command will be ignored.
- If the FIFOs are busy with a previous command, then any new command that would use the FIFOs will also be ignored.
- If the DMA is busy, then all new commands will be ignored. (Controller reset commands will still abort ongoing DMA operations.)
- If bits [7:6] are both zero, a new command will be issued if the device was configured with eMMC support (OPT_EMMC=1), no FIFO activity is requested, and no response is expected. This is to support the 'self-reply' capability designed to force a slave to exit the IRQ state.

While most writes to the command register will initiate commands or data interactions with the external device, there are several exceptions to this rule as shown in Tbl. 5.4. Other than soft reset requests, all writes to the command register will be ignored while it is busy.

Commands may expect one of four types of responses, as shown in Tbl. 5.5.

If the memory bit is set, then the data FIFOs will be used in the operation. Which FIFO is used will be controlled by the FIFO ID bit. "Write" transactions, those that transfer data from the FIFO to the card, are identified by also setting the 'W' bit, otherwise all FIFO transactions will be read transactions by default.

The "D" bit is reserved for operations that would enable the (not yet written, not yet integrated) internal DMA.

Resp Type	Description
2'b00	No response expected
2'b01	R1 (48b) response expected
2'b10	R2 (136b) response expected. Returned data will be placed into the selected FIFO.
2'b11	R1b (48b+BUSY) response expected

Table 5.5: CMD response types

CMD Err Code	Description
2'b00	Timeout. The card has not responded. This will also be the return code for commands not requiring any response.
2'b01	Okay. No error was encounted, and the card returned a valid response.
2'b10	CRC Error. The response returned by the card did not have a valid CRC. Not all commands will have valid CRCs. Check the SDIO and eMMC specifications to know when to expect one.
2'b11	Frame error. Set if the CRC passes, but the stop bit of any command reply is not also set.

Table 5.6: CMD response types

As soon as a command has been requested, the busy bit will be set. It will remain set while the command is being sent. If the command expects a response, then it will remain set while the controller awaits the response. If the command also expects a data transfer, then the busy bit will remain set while the data transfer takes place. Once the entire operation, command, reply, and optional data transfer, has completed the busy bit will be lowered.

The controller will return one of four results for any reply request, as shown in Tbl. 5.6. In general, any non–zero response indicates that a start bit was received.

If the `OPT_CARD_DETECT` parameter is set, then the controller will be responsive to a "card detect" bit. Whether or not a card is currently present can be determined by the "No card present" bit. If ever a card is removed from the system, this bit will be set. A second bit, the card removed bit, will also be set if no card is present. Once a card is inserted, the "No card present" bit will clear automatically leaving only the "Card removed" bit to be cleared manually. This helps to insure that software can always detect any time a card has been removed: the "Card removed" bit will be set until cleared.

If the `OPT_CARD_DETECT` parameter is clear, then the "No card present" bit as well as the "Card removed" bit will always read zero.

If, following any R1b command, the card holds the DAT[0] line low, this will be reflected in the "Card busy" bit. Once the card raises DAT[0], the "Card busy" line will be cleared automatically and will not reassert until the next R1b command.

If, following any command, the card holds the DAT[0] line low, this will also be reflected in the "Card busy" bit.

Rx Err Code	Description
1'b0	Watchdog timeout. Set if no bits are received within the watchdog timeout period. This will get set both if no start bits are received, as well as if the clock (or data strobe) stops mid-reception.
1'b1	CRC error. Set if a sector is received with an invalid CRC. May also be an indication that the received sector had started to arrive before the command to receive that sector had been given.

Table 5.7: Potential receive error codes

The error bit may be set from either a command or a data receiver error. The E_r and E_c bits exist so you can know which of the two (or both) produced the current error.

Receive error codes are listed in Tbl. 5.7.

5.2 Argument register

According to both SDIO and eMMC specifications, all commands sent to the device have a 48–bit format. The first 8-bits specify a start bit followed by the 7–bit command itself, where commands from the host begin with a 2'b01 (including the start bit). The next 32-bits constitute an argument to the command. The argument register controls these next 32-bits.

Writes to this register, prior to issuing a command, will set the argument for the next command.

Following any 48–bit reply from the device, the 32–bit argument from that reply may be read from this register.

5.3 FIFO Registers

As with the SDSPI controller, the SDIO/eMMC controller maintains two (nominally) 512 byte memory areas called FIFOs.[1] Reads from the card will write data into one of the two FIFO's, whereas writes to the card will read data out from one of the FIFO's. Which FIFO the card uses is determined by the *I* bit in the CMD register (above).

Further, upon any write to the CMD register, the FIFO address will be set to point to the beginning of the FIFO.

The purpose of the FIFOs is to allow one to issue a command to read into one FIFO, then when that command is complete to read into a second FIFO. While the second command is ongoing, a CPU or DMA may read the data out of the first FIFO and place it wherever into memory. Then, when the second read is complete, a third read may be issued into the first buffer while the data is read out of the second and so forth.

This interleaving approach, sometimes called ping-pong buffering, can also be used for writing: A multi–sector write might start by first writing into one FIFO. The write command would then be issued, followed by writing into the second (now unused) FIFO. Once the first write completes, a second write command may be issued while the first FIFO is refilled for a third command and

[1] Actual FIFO size is determined by the `LGFIFO` parameter.

31–28	27–24	23	22	21	20–16	15	14	13	12	11–10	9	8	7–0
LgFSz	LgBlk	V_C	V_R	D_C	Shift	nC	C9	P_C	P_D	W	D_S	D_D	CLKDIV

Figure 5.2: PHY Front end configuration register

so forth. Feel free to examine the various include software drivers for examples of how this may be accomplished.

One item to note before closing: there is only one internal address register when accessing the FIFO from the wishbone bus. Attempts to read from or write to either FIFO from the wishbone bus will increment this address register. Interleaved read, or write attempts, such as reading one item from FIFO[0] and writing another item to FIFO[1], will each increment the internal address pointer so that the result is likely to be undesirable. For this reason, it is recommended that only one FIFO be read from or written by the wishbone bus at a time.

5.4 PHY Register

The PHY register controls the SPI clock rate, the sector size, and any other PHY IO options, such as the width of the bus or whether command and data wires are in push-pull or open drain modes. Specifically, with regards to the FIFO size, the PHY register controls how many bytes will be written into the FIFO (which is really of a fixed size) before expecting CRC when receiving data, or equivalently how many bytes to read out of the FIFO before adding a CRC to any transmission. The fields of this register are shown in Fig. 5.2. and explained in Tbl. 5.8.

The CKDIV field needs the most explanation. This field is used to control a digital clock divider. Particular high speed frequencies do not go through the divider. All other frequencies may be expressed as divisions of 50 MHz, as shown in Tbl. 5.10. Attempts to use the high frequency values, but without high frequency support, will be quietly ignored. Indeed, this can be one way of telling whether or not controller support exists for either `OPT_SERDES` or `OPT_DDR`.

This controller supports both SDR and DDR IO modes. DDR IO, however, requires that the clock be offset from the data by 90 degrees rather than the nominal 180 degrees.[2] While the `OPT_SERDES` driver will support 90 degree clock and data offsets at all frequencies, the `OPT_DDR` driver will only do so at 25 MHz and below. Any attempt to enable DDR mode will automatically place the clock in its 90 degree offset mode. Alternatively, the 90 degree offset mode may be enabled specifically–even if not required.

The eMMC controller contains the option of using a data strobe to clock the return data. This requires `OPT_EMMC`. This mode may be enabled in the PHY register. Doing so will force all data bits returned by the external device to be clocked using this return data strobe.

The SDIO/eMMC IO clock may be turned off when not in use. To turn this clock off, set the clock shutdown bit in the PHY register. Note that the DMA may automatically shutdown the clock when reading as well.

The LgBlk field sets the log, base two, of the any memory transfer size. The actual transfer length will be will be 2^{LGBLK} bytes. The WB controller will not allow this to be set larger than the

[2] DDR support may also require 1.8V IO. Check the SDIO/EMMC specifications and your board support for more information.

Name	Description	
LgFSz	4	Reflects the `LGFIFO` parameter, and hence the log based two of the maximum transfer block size.
LgBlk	4	Log based two of the transfer block size. Set to 9 for 512 Byte transfers, 4 for 16‘Byte transfers, etc. Cannot be set below 2.
V_O	1	Read only value reflecting the parameter `OPT_1P8V`. If set, indicates that the controller is capable of driving an output value to direct hardware to switch to a lower voltage.
V_R	1	1.8 Volt request. Set to transition to 1.8 Volts, or leave clear to operate at 3.3 Volts. Only relevant if both `OPT_1P8V` is set, and external hardware is present to allow voltage changes. Once set, this register cannot be cleared without either a hardware reset or the card assertion pin (if supported) being withdrawn.
D_C	1	Set for enhanced data strobe support. If set, both the command and data pins will use the data strobe as a clock pin. Will only be set if D_S and D_D are also set.
Shift	5	Sample shift. This is used to determine IO sampling. The digtal clock output signal will be delayed by one clock plus this many eighth clock units to determine the receive sample point when not using the data strobe. This value is likely to need to be adjusted from its simulation/demonstration settings when operating at high speeds in order to reflect IO delays contained in either the front end IO drivers, the external device, or delays across the circuit board itself.
nC	1	Clock shutdown. Set in order to deactivate the clock between commands, or leave clear for a continuously operating clock.
C9	1	Clock–90. Set in order to offset the clock from the data by ninety degrees, or leave clear for the clock to be offset from the data by 180 degrees. Any attempt to use DDR data transfer will automatically set this bit.
P_C	1	Push–Pull CMD. Set in order to drive the CMD wire via a push–pull scheme, or leave clear to use open–drain for CMD transfers.
P_D	1	Push–Pull Data. Set to drive data wires via a push–pull scheme, or leave clear to use open–drain for data transfer.
W	2	Width. Controls how many data bits are used in any data transaction. Set to 0 to use DAT[0] only, 2’b01 to use DAT[3:0], and 2’b10 to use DAT[7:0]. See also Tbl. 5.9.
D_S	1	Data strobe. Set to one to use the eMMC data strobe, clear otherwise. Requires DDR mode set, so D_D must be high. Parameters `OPT_EMMC` and `OPT_DS` must also be set to use this capability.
D_D	1	DDR. Set to one to transmit on both edges of the clock. Clear to send data on the positive edge of the clock only.
CKDIV	8	Clock division control. See Tbl. 5.10 for more information.

Table 5.8: PHY Front end configuration register fields

Width	Description
2'b00	1b data width. Only DAT[0] will be used to transfer data.
2'b01	4b data width. DAT[3:0] will be used to transfer data.
2'b10	8b data width. DAT[7:0] will be used to transfer data. Only used in eMMC protocols.
2'b11	Reserved/unused.

Table 5.9: Width enumeration

CKDIV	Clock Frequency
8'h0	200 MHz clock. Requires `OPT_SERDES`.
8'h1	100 MHz clock. Requires `OPT_DDR`.
8'h2	50 MHz clock
8'h3	25 MHz clock
8'h4	12.5 MHz clock
N	Sets clock frequency to $25/(N-2)$ MHz.

Table 5.10: Clock division settings

size of the FIFO, as reported by the LgFSz field, so any attempt to set this value past its maximum can be used to read back what the maximum value actually is.

5.5 Soft Resets

The controller may be reset at any time by writing a 32'h52000000 to the CMD register.

Any ongoing transactions will be canceled at that time and the controller will return immediately to idle. Note that this reset *does not* send any commands to the card, it simply resets the controller. Any ongoing operations will be halted mid-operation. If the card is in the middle of a multi-block operation, an additional stop transmission command, followed by an appropriate wait of (roughly) one block length, may be required to bring the card back into a usable state.

Alternatively, a `GO_IDLE` command may be sent at any time–even if a command response is pending or an IO operation is ongoing.

5.6 DMA Usage

An optional DMA may be integrated into this controller. Additional registers are required to use the DMA. These include the DMA length register, and the DMA address register. To use the DMA, these two additional registers must first be set.

The DMA length register controls the number of *blocks* the DMA will transfer. The block length can be set in the PHY register, and it is required to be a power of two. Once the DMA has transferred this many blocks, the DMA operation will end.

The DMA address register controls where on the bus data will be drawn from or written to. During a write operation, data will be read from the bus and written to the card. During a read operation, data will be read from the card and written to the bus. The direction of the operation is controlled by the command word when the DMA operation is requested.

To request a DMA operation, set the DMA bit when requesting a command. A typical read DMA command will set the command bit, request an R1 response, and request a multiblock read. A typical write DMA operation will request an R1 response, set the write bit, the DMA bit, and request a multiblock write from the card.

6.

Interrupts

An interrupt may be generated for any of the following reasons:

1. Any time a command completes, and any anticipated response is either returned or timed out. The exception to this rule is that no interrupt will be generated if either the DMA is active, or if subsequent IO to or from FIFOs is still pending and the command request completes without error.
2. If `OPT_DMA` is set, and the DMA is included, then an interrupt will be generated once a DMA operation is completed. During the DMA operation, however, other interrupts will be suppressed.
3. An interrupt will be generated in non-DMA mode any time a sector is finished being sent to the controller.
4. An interrupt will be generated in non-DMA mode any time a sector finishes being received by the controller.
5. If the `OPT_CARD_DETECT` parameter is set, then an interrupt will be generated any time an acknowledged card is removed.
6. If the `OPT_CARD_DETECT` parameter is set, then an interrupt will also be generated any time a card is inserted. This interrupt, however, will only clear once the "Card removed" bit is also cleared.

All but the last of these reasons will generate a single interrupt pulse, one clock wide.

7. Wishbone Datasheet

Tbl. 7.1 describes the Wishbone command bus interface. It is required by the wishbone specification,

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Slave, (Block/pipelined) Read/Write
Port size	32–bit
Port granularity	32–bit
Maximum Operand Size	32–bit
Data transfer ordering	(Optional)
Clock constraints	(See below)
Signal Names	Signal Name: Wishbone Equivalent i_clk: CLK_I i_wb_cyc: CYC_I i_wb_stb: STB_I i_wb_we: WE_I i_wb_addr: ADR_I i_wb_data: DAT_I i_wb_sel: SEL_I o_wb_ack: ACK_O o_wb_stall: STALL_O o_wb_data: DAT_O

Table 7.1: Wishbone Slave Datasheet

and so it is included here. Note that all wishbone operations may be pipelined, to include FIFO operations, for speed.

The particular constraint on the clock is not really a wishbone constraint, but rather an SD–Card constraint. Not all cards can handle clocks faster than 25 MHz. For this reason, the wishbone clock, which forms the master clock for this entire controller, must be divided down so that the SPI clock is within the limits the card can handle.

When the `OPT_DMA` parameter is also set, a Wishbone master interface will also be enabled, as outlined in Tbl. 7.2. Tbl. 7.2

Description	Specification
Revision level of wishbone	WB B4 spec
Type of interface	Master, (Block/pipelined) Read/Write
Port size	32–bit or larger
Port granularity	8–bit
Maximum Operand Size	Full bus width
Data transfer ordering	(Optional)
Clock constraints	(Same)
Signal Names	Signal Name / Wishbone Equivalent: o_clk — CLK_O o_dma_cyc — CYC_O o_dma_stb — STB_O o_dma_we — WE_O o_dma_addr — ADR_O o_dma_data — DAT_O o_dma_sel — SEL_O i_dma_ack — ACK_I i_dma_stall — STALL_I i_dma_data — DAT_I

Table 7.2: DMA Wishbone Master Datasheet

8.

Clocks

All of the clocks within this system are based upon the speed of the system clock, `i_clk`. This clock determines all other clock speeds within the design.

Name	Source	Rates (MHz)		Description
		Max	Min	
i_clk	Input	104	96	System clock. Nominally at 100 MHz.
i_hsclk	Input			I/OSERDES clock. Must be 4x the i_clk rate, and must come from the same PLL or MMCM as i_clk. This clock input is only used if `OPT_SERDES` is set to cause the design to use the 8x SERDES based front end that requires it.
o_ck	Output	200	0.1	SDIO/eMMC interface clock. This signal is output over the interface at a frequency given by the clock divider controller. May also be turned off between commands.

The `i_clk` speed has been nominally fixed at 100 MHz. Based upon this speed, the controller can generate 200 MHz, 100 MHz, and 50 MHz IO clocks. Other clocks may be generated by integer divisions of 50 MHz.

Other clock speeds may also be possible. However, the design operates primarily on a single clock. Other clock rates will therefore impact both bus and IO operations together.

9. I/O Ports

Table. 9.1 lists the various ports used by this controller, excluding DMA ports. Inputs and outputs

Port	Width	Direction	Description
i_clk	1	Input	System Clock, nominally 100 MHz
i_hsclk	1	Input	SERDES Clock. Must be 4x the system clock.
i_reset	1	Input	System reset. Active high.
?_wb_ Wishbone Slv			Wishbone bus slave signals
?_dma_ Wishbone Master			DMA Wishbone bus master signals
o_ck	1	Output	SDIO/eMMC protocol clock signal
io_cmd	1	In/Out	SDIO/eMMC command wire
io_dat	NUMIO	In/Out	1, 4, or 8 SDIO/eMMC data lines
i_ds	1	Input	eMMC data strobe
i_card_detect	1	Input	High if the hardware detects a card, low o.w. Ignored if `OPT_CARD_DETECT` is not also set.
o_hwreset_n	1	Output	Low if requesting a hardware reset, high otherwise. Requires `OPT_HWRESET` be set.
o_1p8v	1	Output	High if requesting 1.8 Volt IO. Requires `OPT_1P8V` be set.
o_int	1	Output	An interrupt line to the CPU controller
o_debug	32	Output	See Verilog for details

Table 9.1: List of IO ports

are divided into sections: the master clock, the wishbone bus, the SDIO/eMMC interface to the card, and four other wires. Of these, the last two chapters discussed the wishbone bus interface and the clocks. The SDIO/eMMC interface should be fairly straightforward, so we'll move on and discuss the other four wires.

The `i_card_detect` wire should come from any card detection circuitry if present. This is an active high input. If `OPT_CARD_DETECT` is set, this wire will be to set both the "Card not present" and "Card removed" bits.

The eMMC standard includes a negative edge triggered hardware reset. If `OPT_HWRESET` is set, this reset is captured by `o_hwreset_n`.

The 1.8 Volt request line, o_1p8v, is used by the controller to indicate it's desire to switch from 3.3 Volts to 1.8 Volts. The OPT_1P8V parameter should be set low if voltage switching is not supported.

The interrupt line, o_int, has been discussed in Chapt. 6.

The final bus of 32–wires, o_debug, is defined internally and used when/if necessary to debug the core and watch what is going on within it. These wires may be left unconnected in most implementations, as they are not necessary for using the actually controller.