A SYSTEM TO EAVESDROP ON MARMOSETS

by

Christopher Ness Casebeer

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

November, 2015
# TABLE OF CONTENTS

1. INTRODUCTION AND OVERVIEW ................................................................. 1
   - Introduction ............................................................................................... 1
   - System Features ...................................................................................... 2
   - Physical System Components .................................................................. 4
   - FPGA Design Decision and Background .................................................. 7
   - PCB Development Phases ....................................................................... 8
   - FPGA System Architecture .................................................................... 17
   - Project Responsibilities and Further Sections ....................................... 19

2. MICROSD CONTROLLER ....................................................................... 20
   - SD Card Intro ......................................................................................... 21
     - Microsd_controller ............................................................................... 23
     - Microsd_controller_inner .................................................................... 24
     - Microsd_init ......................................................................................... 25
     - Microsd_data ....................................................................................... 25
     - Microsd_crc_7 .................................................................................... 25
     - Microsd_crc_16 .................................................................................. 25
     - Microsd_buffer .................................................................................. 26
   - SD Mode Commands ............................................................................... 26
   - CRC Engines .......................................................................................... 32
   - SD Card Initialization Flow .................................................................... 34
   - SD Data Flow .......................................................................................... 38
   - Voltage Switch and Line Signaling Levels ............................................. 47
   - Performance and Power Usage ............................................................... 53
     - Further Power Usage .......................................................................... 58
   - Power Consumption of Level Shifted Mode ........................................... 61
   - Error Handling ........................................................................................ 66
   - SPI Mode ................................................................................................ 66
   - SD Loader Entity .................................................................................... 67
   - Operation of SD Host Controller System .............................................. 70
   - Further Developments .......................................................................... 73

3. SEGMENT ASSEMBLY ......................................................................... 75
   - Segment Assembler Format ................................................................... 76
   - Flashblock Entity VHDL Code ............................................................... 81
   - Data Recovery ........................................................................................ 86
   - Flashblock Parser .................................................................................. 90
   - Further Developments .......................................................................... 94
TABLE OF CONTENTS – CONTINUED

Conclusions ................................................................................................ 94

4. INITIAL SYSTEM POWER STUDIES ......................................................... 95
   Analog to Digital Converter Setup and Usage ........................................ 95
   Power Studies ..................................................................................... 98
   Future Developments ......................................................................... 104
   Conclusions ...................................................................................... 105

5. MEMS MICROPHONE ............................................................................ 106
   Cascaded Integrator Comb Filter ....................................................... 110
   VHDL Code and Architecture ............................................................. 112
   Microphone Troubleshooting ............................................................... 114
   Data Logging ..................................................................................... 115
   MEMS Microphone Power Studies .................................................. 120
   Conclusions ...................................................................................... 121
   Further Characterization .................................................................. 122

6. INERTIAL MEASUREMENT UNIT ............................................................ 123
   VHDL Code and Architecture ............................................................. 126
   Targeted Register Set ........................................................................ 134
   Obtained Data .................................................................................. 136
   IMU Power Measurements ................................................................. 144
   Future Implementations .................................................................... 147

7. FINAL REMARKS .................................................................................. 148
   Further Developments ....................................................................... 148
   Future Directions ............................................................................... 148
   Conclusions ...................................................................................... 150

REFERENCES CITED .............................................................................. 152

APPENDICES ............................................................................................. 155
   APPENDIX A: VHDL Code ................................................................. 156
   APPENDIX B: Matlab Code ............................................................... 458
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Level Shifting Power Comparison</td>
<td>66</td>
</tr>
<tr>
<td>3.1 Defined Segment Types</td>
<td>79</td>
</tr>
<tr>
<td>6.1 IMU Initial Testing Register Set</td>
<td>135</td>
</tr>
<tr>
<td>6.2 IMU Current Comparison</td>
<td>146</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Collar System Overview</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>BeMicroCV Sensor Daughter Board</td>
<td>10</td>
</tr>
<tr>
<td>1.3</td>
<td>BeMicroCV Sensor Daughter Board</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Power Monitor Development Board</td>
<td>12</td>
</tr>
<tr>
<td>1.5</td>
<td>Power Monitor Development Board</td>
<td>13</td>
</tr>
<tr>
<td>1.6</td>
<td>Marmoset Collar Board</td>
<td>14</td>
</tr>
<tr>
<td>1.7</td>
<td>Marmoset Collar Board</td>
<td>15</td>
</tr>
<tr>
<td>1.8</td>
<td>Initial Test Sensor Board</td>
<td>16</td>
</tr>
<tr>
<td>1.9</td>
<td>FPGA Sensor Architecture</td>
<td>17</td>
</tr>
<tr>
<td>1.10</td>
<td>FPGA Peripheral Architecture</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>microSD Card Dimensions and Pins</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>microsd_controller HDL system Layout</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>SD Mode Command Format</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>Command Response of the SD Command Line</td>
<td>28</td>
</tr>
<tr>
<td>2.5</td>
<td>Command Send Process of sd_init Core</td>
<td>29</td>
</tr>
<tr>
<td>2.6</td>
<td>CMD0 State Machine Output Logic</td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>R1 SD Mode Response Format</td>
<td>30</td>
</tr>
<tr>
<td>2.8</td>
<td>R6 SD Mode Response Format</td>
<td>31</td>
</tr>
<tr>
<td>2.9</td>
<td>R1 Response Handler</td>
<td>31</td>
</tr>
<tr>
<td>2.10</td>
<td>Tri-State Communication Signals</td>
<td>32</td>
</tr>
<tr>
<td>2.11</td>
<td>Command CRC7</td>
<td>33</td>
</tr>
<tr>
<td>2.12</td>
<td>Data CRC16</td>
<td>33</td>
</tr>
<tr>
<td>2.13</td>
<td>Card Initialization State Diagram</td>
<td>35</td>
</tr>
<tr>
<td>2.14</td>
<td>ACMD41 Format</td>
<td>37</td>
</tr>
<tr>
<td>2.15</td>
<td>microsd_init Core FSM states</td>
<td>38</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES – CONTINUED

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.16</td>
<td>Data Transfer Mode</td>
<td>39</td>
</tr>
<tr>
<td>2.17</td>
<td>microsd_data Core FSM States</td>
<td>40</td>
</tr>
<tr>
<td>2.18</td>
<td>Available SD Function Groups</td>
<td>42</td>
</tr>
<tr>
<td>2.19</td>
<td>CMD6 Command Format</td>
<td>42</td>
</tr>
<tr>
<td>2.20</td>
<td>Multiple Block Write Flow</td>
<td>43</td>
</tr>
<tr>
<td>2.21</td>
<td>Four Bit Data and CRC Append Format</td>
<td>44</td>
</tr>
<tr>
<td>2.22</td>
<td>CRC Okay Response</td>
<td>45</td>
</tr>
<tr>
<td>2.23</td>
<td>Data Response Token</td>
<td>46</td>
</tr>
<tr>
<td>2.24</td>
<td>TXB0106 Bidirectional Level Translator Pin Out</td>
<td>48</td>
</tr>
<tr>
<td>2.25</td>
<td>Level Translation Setup</td>
<td>49</td>
</tr>
<tr>
<td>2.26</td>
<td>ACMD41 Format</td>
<td>50</td>
</tr>
<tr>
<td>2.27</td>
<td>Level Switch Sequence</td>
<td>50</td>
</tr>
<tr>
<td>2.28</td>
<td>Incorrect Clock Handling Voltage Switch</td>
<td>51</td>
</tr>
<tr>
<td>2.29</td>
<td>Clean SD Clock Transition</td>
<td>52</td>
</tr>
<tr>
<td>2.30</td>
<td>SD on Linux Throughput (figure from [7])</td>
<td>54</td>
</tr>
<tr>
<td>2.31</td>
<td>Standby SD Card Current (less than 1mA)</td>
<td>55</td>
</tr>
<tr>
<td>2.32</td>
<td>Energy to Write 1MB of Data. Initial microSD Studies</td>
<td>56</td>
</tr>
<tr>
<td>2.33</td>
<td>Throughput as a function of multiblock count number and frequency</td>
<td>57</td>
</tr>
<tr>
<td>2.34</td>
<td>50MB of MicroSD Writing Using a 128 Multi-Block Write Method</td>
<td>59</td>
</tr>
<tr>
<td>2.35</td>
<td>Throughput decreases occur when the card has not been pre-erased</td>
<td>60</td>
</tr>
<tr>
<td>2.36</td>
<td>Direct SD Write Current</td>
<td>63</td>
</tr>
<tr>
<td>2.37</td>
<td>Level Shifted SD Write Current</td>
<td>64</td>
</tr>
<tr>
<td>2.38</td>
<td>System Power Direct SD Writes</td>
<td>65</td>
</tr>
<tr>
<td>2.39</td>
<td>sd_loader System Position</td>
<td>68</td>
</tr>
<tr>
<td>2.40</td>
<td>microsd_controller HDL system layout</td>
<td>70</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES – CONTINUED

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.41</td>
<td>microsd_controller Generics</td>
</tr>
<tr>
<td>3.1</td>
<td>Segment Assembler (Flashblock)</td>
</tr>
<tr>
<td>3.2</td>
<td>The Defined Block and a Contained Segment</td>
</tr>
<tr>
<td>3.3</td>
<td>send_item FSM states</td>
</tr>
<tr>
<td>3.4</td>
<td>send_block_item FSM states</td>
</tr>
<tr>
<td>3.5</td>
<td>Audio Data Interrupt Handler VHDL Code</td>
</tr>
<tr>
<td>3.6</td>
<td>Status Segment Last Audio FPGA Time Handling</td>
</tr>
<tr>
<td>3.7</td>
<td>Cygwin and Use of dd Command</td>
</tr>
<tr>
<td>3.8</td>
<td>Highlighted Block as Saved to MicroSD</td>
</tr>
<tr>
<td>3.9</td>
<td>Annotated MicroSD Block Recovery</td>
</tr>
<tr>
<td>3.10</td>
<td><code>search_flashblock.m</code> Function Definition</td>
</tr>
<tr>
<td>3.11</td>
<td>IMU Accelerometer Test Recovery</td>
</tr>
<tr>
<td>3.12</td>
<td>Flashblock Assembled and Recovered Audio</td>
</tr>
<tr>
<td>4.1</td>
<td>TI ADS131E08EVM-PDK</td>
</tr>
<tr>
<td>4.2</td>
<td>Single-Ended ADS131E08 Configuration</td>
</tr>
<tr>
<td>4.3</td>
<td>Minimal FPGA Design</td>
</tr>
<tr>
<td>4.4</td>
<td>Total System Current and Voltage</td>
</tr>
<tr>
<td>4.5</td>
<td>Typical Programming Profile Breakdown</td>
</tr>
<tr>
<td>4.6</td>
<td>Minimal System Power</td>
</tr>
<tr>
<td>4.7</td>
<td>Minimal FPGA System Current Usage</td>
</tr>
<tr>
<td>4.8</td>
<td>Max V CPLD Current Usage</td>
</tr>
<tr>
<td>4.9</td>
<td>MicroSD System Current Profile</td>
</tr>
<tr>
<td>5.1</td>
<td>MEMS Microhpone Functional Block Digram</td>
</tr>
<tr>
<td>5.2</td>
<td>MEMS Microphone Chip View</td>
</tr>
<tr>
<td>5.3</td>
<td>INMP621 Pin Out (bottom view)</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>5.4</td>
<td>Encoding and Bitrate Conversion</td>
</tr>
<tr>
<td>5.5</td>
<td>PDM Data Phase</td>
</tr>
<tr>
<td>5.6</td>
<td>Oscilloscope Capture of PDM Bits</td>
</tr>
<tr>
<td>5.7</td>
<td>Microphone Component Interfacing</td>
</tr>
<tr>
<td>5.8</td>
<td>mems_top_16 ports</td>
</tr>
<tr>
<td>5.9</td>
<td>hd_16_ports</td>
</tr>
<tr>
<td>5.10</td>
<td>Recovered Sound Test</td>
</tr>
<tr>
<td>5.11</td>
<td>microSD binary Dump Audio Data</td>
</tr>
<tr>
<td>5.12</td>
<td>Audio Segment Type and Length</td>
</tr>
<tr>
<td>5.13</td>
<td>Microphone Startup and Record Current Profile</td>
</tr>
<tr>
<td>5.14</td>
<td>MEMS Microphone Sleep Mode Current Draw</td>
</tr>
<tr>
<td>6.1</td>
<td>LSM9DS1 Sensors and Directions</td>
</tr>
<tr>
<td>6.2</td>
<td>Shared IMU SPI Bus Structure</td>
</tr>
<tr>
<td>6.3</td>
<td>LSM9DS1 SPI Communication Example</td>
</tr>
<tr>
<td>6.4</td>
<td>LSM9DS1 Pins</td>
</tr>
<tr>
<td>6.5</td>
<td>LSM9DS1_TOP VHDL Architecture Overview</td>
</tr>
<tr>
<td>6.6</td>
<td>IMU Register Set Creation Flow</td>
</tr>
<tr>
<td>6.7</td>
<td>LSM9DS1 Register Map Example</td>
</tr>
<tr>
<td>6.8</td>
<td>LSM9DS1 FIFO in Bypass Mode</td>
</tr>
<tr>
<td>6.9</td>
<td>Gyroscope X Axis Test</td>
</tr>
<tr>
<td>6.10</td>
<td>Gyroscope Y Axis Test</td>
</tr>
<tr>
<td>6.11</td>
<td>Gyroscope Z Axis Test</td>
</tr>
<tr>
<td>6.12</td>
<td>Accelerometer X Axis Test</td>
</tr>
<tr>
<td>6.13</td>
<td>Accelerometer Y Axis Test</td>
</tr>
<tr>
<td>6.14</td>
<td>Accelerometer Z Axis Test</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>6.15</td>
<td>Current Profile IMU XL_G_M</td>
</tr>
<tr>
<td>6.16</td>
<td>IMU Current Profile Without Gyroscope</td>
</tr>
</tbody>
</table>
This masters thesis describes developing a custom digital recording system to record the vocalizations and behavior of marmosets, which are small primates native to the northeast of Brazil. Animal behavior scientists have traditionally studied communications between only a "sender" and a "receiver". Animal communications however are hypothesized to occur in communication networks involving more than just a pair of animals. In this project a miniaturized recording system aimed at acquiring data to study the communication networks of the common marmoset is underway. The acoustic recording collar project aims to develop a wearable recording embedded platform for a freely behaving primate. A custom embedded platform utilizing a field programmable gate array (FPGA) has been developed to prototype the system. A hardware description language (HDL) has been used to create the FPGA architecture for the collar application, which in this case is the VHSIC Hardware Description Language (VHDL). Sensors used and developed for this application include a global positioning system (GPS), inertial measurement unit (IMU), and digital MEMS microphone. These sensors provide position and accurate time information, behaviorally related motion information, and the acoustic environment of the marmoset. This data comprises the Behavioral Acoustic Biome of the marmoset. Storage of the behavioral acoustic biome data occurs on a local microSD flash memory card. A printed circuit board of footprint 1.36 by 1.18 inches has been completed and the system will be soon fitted to a 3D printed collar wearable by the marmoset. Demonstration of sensor data logging to the microSD flash has been completed. Other developments of the embedded system are ongoing. Ultimately, fitting multiple wearable devices across a troop of freely behaving marmosets will allow novel studies of communication networks in the common marmoset to be undertaken.
INTRODUCTION AND OVERVIEW

Introduction

A recording system is being developed to study communication networks in wild marmosets. This recording system will be fit in a "collar" and be lightweight and small. The collar will feature a full embedded computer system which will record the Behavioral Acoustic Biome of the wild marmoset. The behavioral acoustic biome consists of the auditory acoustic vocalizations, the spatial locations, and the behaviorally related movements of the marmoset. Recording will occur over an entire collared troop of marmosets living in Brazil, or as many as can be practically trapped, collared, and then set free. The project requires fitting small sensors into the collar system with a method to store sensor data for later retrieval. The sensors are maintained and interfaced using a field programmable gate array (FGPA) which handles system processing and is a great platform for prototyping the system. The FPGA is central to the collar system design and is a platform which allows great flexibility in system design. Recent advances in cell phone technology have allowed miniaturized technologies to advance and become widely available. Key technologies which have enabled this project include large storage in small form factor made possible by microSD flash cards. The current microSD flash module stores 64GB of data. Sensors based on microelectromechanical systems (MEMS) allow sensing acoustic sounds and spatial movement using very small integrated circuits. Global positioning systems (GPS) are available in small integrated packages allowing interface to a host. The GPS is unique in this system as it provides both positioning information as well as very accurate timestamp information. Accurately timestamping the sensor data gives information on when behavioral events occurred
and will play a significant role in correlation and causation studies of marmoset behavior given the recorded data. As the recording collar will be attached to the marmoset, the design will be powered by a battery system. This battery system will feature a recharge capability through use of a solar cell array positioned on the collar. The battery has a finite amount of energy to power the collar and thus power limitations have been at the forefront of system design since project start. Efforts to select low power devices and develop sensor interfaces with a mind towards energy savings has been used in development of the collar thus far. Figure 1.1 shows the major system sensors and peripherals which comprise the marmoset behavioral acoustic biome recording system.

**System Features**

The collar system will collect sensor data from multiple integrated circuit and MEMS sensors and then will store the data to a microSD flash card. The sensors include a digital microphone to record animal vocalizations and other external sounds, an inertial measurement unit (IMU) for local behaviorally related motion data, and a global positioning system (GPS) to provide position and absolute time data. The system is powered by a battery and is initially power limited. The system will need to recharge the battery via solar cells before any subsequent recordings can occur. As a result, the power usage of all parts of the system has been considered in their selection. One design choice that has a significant impact on the design and power consumption of the system has been the use of an FPGA for the digital system controller. This device was chosen for its development ease and flexibility as a system prototyping platform, both for the current system as well as for future capabilities. The design choice was made knowing that the FPGA can use more
power than alternative microcontrollers. However use of an FPGA allows great flexibility in system design and can implement DRAM controllers that interface with low power DRAM (LPDRAM) chips, which is an interface that is lacking on typical microcontrollers. For the current prototype system board we are testing, we have added power measurement capabilities for all active devices in the system.
Physical System Components

The marmoset collar features numerous peripherals and core system components to realize the goal of recording the behavioral acoustic biome. The peripherals and sensors currently specified for the collar project are briefly described below. Each component is followed by a brief description of its role in the project. Final printed circuit board (PCB) design features all of the mentioned components.

Complex Programmable Logic Device (CPLD) Power Controller Max V CPLD
A CPLD is utilized due to its very low power usage to switch power to all system components including the FPGA. Its responsibilities include programming the FPGA and communicating with the FPGA to directly switch on and off power to all sensor peripherals.

Field Programmable Gate Array (FPGA) Main Processor Cyclone V FPGA
All communications with external peripherals and sensors occurs within this processor. The VHDL entities responsible for encapsulating all the sensor handling including the inertial measurement unit, microphone, and global positioning system are realized as programmed architecture within the Cyclone V. The FPGA also controls the interface to the low power DRAM buffer and moves the data from the DRAM to microSD flash storage.

Inertial Measurement Unit (IMU) ST Microelectronics LSM9DS1
The inertial measurement unit has the capability of measuring 3D rotational change, 3D angular change, and 3D magnetic field strength.

MEMS Microphone Invensense INMP621
The MEMS microphone allows the detection of acoustic pressure waves or sound in the environment. This sensor will record the marmoset’s vocalizations as well as
the marmoset communication network as a whole. When an entire marmoset troop is recorded, communication network studies can be undertaken.

Global Positioning System (GPS) uBlox MAX 7Q

The GPS provides the system with positioning information as well as very accurate time information.

microSDXC memory card SandDisk Ultra 64GB

A SanDisk Ultra microSDXC 64GB flash memory card was used throughout development to achieve the data recording system. A full microSD VHDL interface for serial logging of data at high speed was developed. This is the location of where all stored sensor data is kept.

64 MB Low Power SDRAM Micron Mobile LPSDR SDRAM MT48H32M16LFB4

Early in development of the microSD card it was found that considerable energy saving would be possible if data was buffered before engaging the microSD card. The sensor data is buffered in LPSDRAM before being sent to the microSD card.

Magnetic Memory Everspin MR20H40

Magnetic memory allows persistence of key elements of data between events involving complete loss of power to the device. Important elements of the design such as number of recharge cycles and location of last data write location on the SD card are critical pieces of data that must persist between complete loss of power. Those elements are stored to magnetic memory.

Real Time Clock Maxim DS1371

This component allows alarms to be set which persist through the FPGA power cycle. This part specially allows the collar system to wake and record at predefined times of day.

Battery Fuel Gauge Texas Instruments BQ27520-G3
Information on the battery is provided such as capacity, available energy, and time to empty.

Solar Controller Linear Technology LTC3105

Solar cell output voltage can be stepped to battery recharge voltage levels. The solar cells connected to the DC/DC converter can be operated at their maximum power point through setting the targeted input voltage to the solar controller. Output of the solar controller can also be stepped and maintained to the optimum battery recharge voltage.

Transceiver Integrated Circuit Texas Instruments CX1120

A Texas Instruments (TI) transceiver chip paired with sub 1-Ghz antennae circuitry will allow the system to communicate with a base station. Targeted radio frequency is approximately 440 Mhz. Status updates about the state of the collar will be relayed to researchers. Researchers will also be able to issue commands to the collar.

Power Regulators Linear Technology Buck-Boost LTC3521 and TI Step-Down TPS62085

These components provide the power rails needed for the entire collar. The Cyclone V FPGA requires 1.1V, 1.8V, 2.5V, and 3.3V rails to power its I/O banks and power its internal logic. A buck/boost power regulator and a step-down regulator implementation provide the needed power rails for the FPGA, CPLD and all other peripherals.

FPGA Bitstream Flash Memory Micron Serial NOR Flash Memory N25Q128A

The FPGA loses its programming upon loss of power and thus a bit stream configuration must be stored. A flash memory retains the FPGA bit stream for reprogramming the FPGA upon power on. This flash memory is programmed before deployment to the field and contains the FPGA’s runtime architecture.
Reprogramming of the FPGA from the flash memory is initiated and controlled by the CPLD.

**FPGA Design Decision and Background**

Central to the design is an Altera Cyclone V Field Programmable Gate Array (FPGA) and an Altera Max V Complex Programmable Logic Device (CPLD). The FPGA provides the main computational processing and architecture to handle the vast majority of the marmoset collar project. Its role is similar to the central processing unit of a traditional computer but this processing unit can be designed and redesigned at will by the embedded hardware developer. A hardware description language (HDL), VHDL in this instance, is used to define the embedded logic in the FPGA, which is comprised of interconnected logic elements. VHDL stands for VHSIC Hardware Description Language and VHSIC stands for Very High Speed Integrated Circuit. Logic elements are the basic building blocks of FPGAs and are typically implemented as small memories that act as functional lookup tables, which also contain D flip-flops to capture the outputs. These logic elements are connected together at synthesis by the toolchain that maps the VHDL functionality to a combination of these logic elements. Building up large numbers of these interconnects and logic elements define the runtime function of the FPGA. Architectures described by the VHDL language allow specific tasks to be accomplished and can also be changed at will during the design process. Unlike microprocessors with fixed instruction sets, FPGAs allow custom architectures and operations to be specifically developed for a targeted application. The FPGA and CPLD function in much the same way, with the CPLD having much less resources available to the user but is also developed with VHDL code defining the interconnect of the logical blocks. The
benefit of the CPLD is that it consumes much less power than the FPGA. Both technologies allow building architectures that target specific applications. Given the large amount of Input/Output (IO) pins of the CPLD and FPGA, the auxiliary resources available to the FPGA such as memory blocks, multipliers, and phase locked loops, designers are given great flexibility in determining how to create an optimal architecture for the required design. The FPGA functions at the center of the marmoset collar and controls the sensors and peripherals. The CPLD functions as a system power controller concerned mostly with boot-up and turning on and off peripherals and sensors. Usage of Altera’s toolchain allows compilation and fitting of the architecture created in VHDL to be synthesized, placed, and routed in the FPGA and CPLD fabric for runtime operation. These devices are ideal for their flexibility in developing the optimal custom computer architecture needed for collecting and processing sensor data. The tradeoff is the increase in power consumption which is greater than using low-power microcontrollers. However, microcontrollers limit design flexibility and would be difficult to use with low power SDRAM.

**PCB Development Phases**

A custom miniaturized system which will fit a 3D printed collar targeting the common marmoset is needed to successfully record the behavior of the marmoset. This collar and associated embedded system will be able to record the behavioral acoustic biome of a troop of marmosets within their habitat existing in northeastern Brazil. In the effort to achieve this, several PCB iterations have occurred. These are detailed and described next. I did not conduct the PCB development apart from an initial daughterboard. PCB development was conducted by project team member
Kyler Callahan. A description of the PCB boards frames the project as a whole and explains the places where FPGA and sensor development has occurred.

Early PCB design involved a custom PCB built to interface with the Arrow BeMicroCV Cyclone V development kit [1]. I developed a daughterboard for the BeMicroCV FPGA development board which contained many of the sensors of interest. It was at this time that the MEMS microphone system was first successfully tested. Features of the microSD controller such as level translation were also first developed and tested here. The FPGA was absorbed into the next PCB which was designated the power monitor development board. All PCB design occurred in the Mentor Graphics PADS design suite. The initial BeMicroCV sensor board developed is shown in Figures 1.2 and 1.3.

PCB development went through two more large iterations. A large main development board was made to fully house all components of interest for test and development. It is termed the power monitor development board since on this board all components have measurement points attached to the power inputs of all active devices. Thus the board is termed the power monitor board. This board was larger in size and used to enact design and debug for the next board iteration. It is currently still the ideal choice for active development on the remaining systems due to its size and presence of power sense points. Final characterizations of the microSD controller occurred here. Development of the IMU system was finalized on this board as was the final verification of the MEMS microphone system. Other system peripherals are being actively developed using the power monitor development board. The power monitor board is shown in Figures 1.4 and 1.5.

A final miniaturized board was designed for use on an actual marmoset. This board is the PCB which will reside within the collar/backpack, which the marmoset will wear. The miniaturized system was split into a main board and sensor board.
This functional separation was done to allow future sensor board designs to target other applications. The main collar board houses all core system functionality. These functionalities include the FPGA, CPLD, power regulators, SDRAM, and programming interface. All sensor functionality was moved to the connected sensor daughter board. This board is termed the sensor board. The main board and sensor board connect through two 50 pin interconnects. The main collar board is shown in Figures 1.6 and 1.7.

Base functionality of the marmoset collar board has been tested. The FPGA, CPLD, and power regulator systems are all functional and have been successfully tested. These tests included programming the CPLD and FPGA through their
JTAG programming interfaces. Powering of the devices was also successful indicating proper power regulation. Initial programming of the FPGA by CPLD with bitstream residing in flash was also tested successfully. Counter test programs were used to test functionality here. A very initial sensor board design populated with a MEMS microphone and IMU was also tested successfully when interfaced with the main collar board. That initial sensor board PCB design is shown in Figure 1.8.

Final sensor board design housing all sensors needed for final recording is currently being finalized.
Figure 1.4: Power Monitor Development Board
Figure 1.5: Power Monitor Development Board
Figure 1.6: Marmoset Collar Board
Figure 1.7: Marmoset Collar Board
Figure 1.8: Initial Test Sensor Board
FPGA System Architecture

FPGA system design has been developed to harvest the sensor data and store it permanently to microSD flash. Between a particular sensor IC and microSD flash there exist multiple VHDL entities. The overall system architecture has been developed which moves sensor data towards the microSD flash. In Figure 1.9 sensors which provide data are shown on the left. Data movement occurs left to right. Each sensor is paired with a VHDL defined entity and controller. These controllers are responsible for encapsulating all needed communications and implement the abstraction needed to retrieve and present the sensor data to the segment assembler. The three major sensors are the GPS, IMU, and the microphone.

![Figure 1.9: FPGA Sensor Architecture](image)

The controller developers make use of the associated sensor data sheet and knowledge of the development board layout as well as VHDL and FPGA design
to successfully retrieve sensor data. However there are other peripherals not shown in Figure 1.9 which are also important to the fully functional collar system. Other system components are shown in Figure 1.10.

Figure 1.10: FPGA Peripheral Architecture

The transceiver which sends and receives wireless packets to base station is shown. The real time clock allows the system to wake at specified times. Magnetic memory allows storage of small pieces of important data between power cycles. The MAX V CPLD is termed the power controller within the marmoset collar design. It controls switches attached to all powered devices on the marmoset collar. The CPLD turns off and on all devices through control of these power switches. This includes turning on the FPGA itself. The FPGA signals which devices to turn on once powered
through transmitting a control register to the CPLD over a serial parallel interface (SPI) bus designed between the FPGA and CPLD. The CPLD also is responsible for programming the FPGA upon collar power up where it gets the configuration bit stream from a piece of flash memory residing on the main board.

**Project Responsibilities and Further Sections**

My project responsibilities included creation of the microSD storage interface and controller, IMU interface and controller, the sensor data assembler, MEMS microphone implementation, and physical development board debug. Initial debug of the physical development boards was my responsibility and yielded feedback and future revision information to be passed to the system designer and PCB designer. The following thesis sections describe the systems of the acoustic recording collar project I have directly designed, assembled, and tested myself. These sections are the microSD controller, the MEMS microphone, the IMU, and the segment assembler. The sections give a characterization of the component and an overview of the VHDL architecture developed and tested for that device or function. A section on power monitoring is also included to describe the power monitoring setup and the power needed by the system as a whole as contrasted to the power profile of an individual component. The individual component sections give power characterization of the devices by themselves.
MICROSD CONTROLLER

MicroSD cards allow large data storage up to 200 GB in a small 15 x 11 mm package. This grants the recording collar the ability to record the behavioral acoustic biome of the marmoset for long periods of time. At the current microphone fidelity (56 KHz sample rate), two weeks of continuous audio (two channels) can be stored to the targeted 64GB card. The use of the microSD card with such large data capacities has shifted project limitations completely away from insufficient data storage space utilizing physically large flash cards or hard drives. The ability to communicate with the microSD card at high rates up to 104 MB/s allow handling sensor data input without fear of data loss or buffer overrun. The inclusion of microSD flash solves the issue in a relatively low power footprint of where to store the behavioral acoustic biome as it is recorded.

To allow saving multiple sensor data streams to the marmoset collar, a microSD card VHDL controller was developed. This component was developed through multiple iterations and utilized multiple development boards as the project progressed. As development proceeded, more advanced subsets of the SD protocol were added to the controller. In its current state the microSD controller can record serial amounts of data at high speed utilizing the SD card protocol. This allows data from microphones, inertial measurement units, global positioning systems, and other data to be logged to the physical microSDXC card for later recovery.

The microSD_controller and its sub entities are a complete VHDL component system to write serial data to a microSDXC card. The controller is written in VHDL for use with an FPGA and has been developed with the Altera Quartus II toolchain targeting a Cyclone V FPGA. The microSD_controller accepts data from the host and writes that data to an attached microSD card. The user does not need knowledge
of the SD card protocol as this is abstracted away, hidden in a VHDL state machine that implements the protocol. The user only needs to clock data to the component appropriately and the data will be readable off the SD card in a raw binary format. At present, no file system has been implemented since the logging simply implements a binary stream that is written to flash in a serial fashion. The microSD_controller was written specifically for a data logging application where an FPGA was the center of the platform. Through use of this component, the user can dump blocks of data to the SD card beginning at any desired address. The user must notify the controller how much data will be sent to the SD card. After this data is written, the component will indicate to the host if the write was successful. The final developed system has achieved verified data write and recovery at 11MB/sec using a 50 MHz clock. microSD has the potential to run using a 208 MHz clock and achieve 104MB/sec speeds but this has not been tested on our system yet.

**SD Card Intro**

MicroSD cards have a form factor of 15 mm x 11 mm x 1 mm and weigh 0.5 grams. The SD card has 8 physical connections of interest. The card has 3.3V power and ground connections. SD card communication is done through a custom command response protocol defined by the SD Group in a SD Specifications document [2]. The lines which are involved in communications are the clock, command, and data lines. The clock line provides a clock to the SD card which is used to time command and response alignment. The command line is a bidirectional line. It handles all command and response transfers. Commands handle aspects such as requesting a write, read or erase. Commands are also the method in which the card is initialized to a writeable state. It is not involved in any data movement however. All data bits flow over the
bidirectional data lines. There are four data lines. The lines are often referred to as clk, cmd, and dat0 through dat3. Data flows in parallel over the four data lines bi-directionally in the case of reading or writing.

Data on the microSD card is addressed and handled in blocks. A block is defined by the microSD protocol as 512 bytes. This smallest unit of addressable data (a block of 512 bytes) is a new design of SDHC and SDXC type cards. Previous SDSC cards had sub 512 byte addressing. Thus all data sent to the microSD card through the VHDL entities is handled in 512 byte blocks. As a result, the system has been built with the 512 byte block size in mind. This makes this microSD
controller implementation incompatible with older SDSC cards where addresses are
byte oriented and not block oriented.

The SD protocol consists of numerous commands designated by number. To write
data to the card the host must send commands to the SD card over the command
line appropriately, getting the card past initialization and into the appropriate state
for transferring data to the card.

SD card communications can take many forms at many different speeds. The SD
protocol allows clock speeds up to 208Mhz as well as double data rate configurations.
Our design of the host microSD VHDL controller targets a 50 MHz single data rate
mode where all 4 data lines are used with the ability to signal at 1.8V levels. The 1.8V
signaling level requires external circuitry (detailed later in section Voltage Switch and
Line Signaling Levels). The top level VHDL component microsd_controller.vhd and
its sub entities will be referred to as the host controller. The microsd_controller, its
sub entities and its port connections are shown Figure 2.2.

The VHDL entity microsd_controller encapsulates the communications to initialize
the microSD card as well as write data to the microSD card. A system overview
is now given which gives general descriptions of the blocks which make up the
microsd_controller system. This overview is done now before explanation of SD
protocol and code specifics.

**Microsd_controller**

Microsd_controller is the top level component the VHDL developer will interact
with. This top component instantiates microsd_controller_inner. It also tri-states the
cmd and data lines depending on the state of write enable signals coming from either
microsd_init or microsd_data. This level also handles the microsd_data core control
in relation to the internal data buffer. For example, if the buffer has data ready to be written, microsd_controller engages microsd_data to write that data.

**Microsd_controller_inner**

This inner component has two jobs. One is to create the 400 kHz initialization clock through a clock divider. Microsd_controller_inner also instantiates the two lower components microsd_data and microsd_init and multiplexes communications between them depending on if initialization has completed.
Microsd_init

This core is responsible for issuing all commands related to SD card initialization. This component is responsible for all commands up through issuing CMD3 to the SD card. The microsd_init core also makes decisions on initialization steps based on the desired operating modes. It is during initialization when the switch to 1.8V signaling occurs. The microsd_init core is clocked at 400kHz. Microsd_init also communicates with the microSD card at 400kHz.

Microsd_data

This component is responsible for all data related communications with the SD card. The core is currently capable of exercising a CMD25 multi-block write to stream continuous 512 byte blocks to the card. The number of blocks in any multi-block stream has been coded to be 128 blocks or less depending on the number of blocks currently being transmitted. Other requirements of writing data, such as switching into 4 bit mode and tracking data cyclic redundancy checks, is handled here. The core also contains partial read and erase functionalities.

Microsd_crc_7

This component contains the logic required for generating the CRC7 checksum that is associated with all commands issued to the SD card over the command line. The code herein is a VHDL port of Kay Gorontzi’s online CRC generator and LGPL Verilog code [3].

Microsd_crc_16

All data blocks sent to and received from the card over the data lines are protected by a CRC16 checksum. This engine generates the checksums attached to blocks of
data sent to the card. The code herein is a VHDL port of Kay Gorontzi’s online CRC
generator and LGPL Verilog code [3].

**Microsd_buffer**

This component is a data buffer where data is buffered by the host while the SD
card is busy writing other data. The host will buffer data here quicker than the card
can write the data. The microsd_data core takes its data from microsd_buffer. This
buffer is expandable in size via top level generics. Currently, the default size is 2048
bytes or 4 blocks. An Altera dual-port RAM is used for this buffer. A buffer water
level mark and read and write pointers track the buffer status. The buffer water
level mark controls a buffer full flag reported to the user of the microSD controller
through a top level entity port. The component tracks the number of blocks that
have moved through the buffer in relation to the number of blocks expected. This
buffer also allows the microSD_data entity to exercise a data resend sequence in the
instance where data fails to reach the microSD card correctly as indicated by a CRC
error reported by the SD card.

**SD Mode Commands**

Commands sent to the SD card over the command line follow a certain format as
described in the SD specifications. SD mode commands are illustrated in Figure 2.3.

<table>
<thead>
<tr>
<th>Bit position</th>
<th>47</th>
<th>46</th>
<th>[45:40]</th>
<th>[39:8]</th>
<th>[7:1]</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width (bits)</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>32</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Value</td>
<td>'0'</td>
<td>'1'</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>'1'</td>
</tr>
<tr>
<td>Description</td>
<td>start bit</td>
<td>transmission bit</td>
<td>command index</td>
<td>argument</td>
<td>CRC7</td>
<td>end bit</td>
</tr>
</tbody>
</table>

**Table 7-1: Command Format**

Figure 2.3: SD Mode Command Format
Of note is that a SD card can be operated in one of two modes, SPI mode and SD mode. SPI mode is a compatibility mode for microcontrollers which may only have a SPI communication interface. However, as SPI only offers one data line and the SPI mode is a subset of the full SD mode, SPI is a slower mode and not used in performance applications with modern day SD implementations. Initial developments targeted the SPI mode. However, the SD mode gives access to the full capabilities and performance of the SD card. Early in development the SD mode was targeted as the sole communication mode. Thus all further descriptions target only the SD mode.

The controller in its current state utilizes SD mode communications exclusively. To issue a command to the SD card, a command must be sent to the card with specific format as shown in Figure 2.3. The SPI mode and SD mode command formats are the same. A start bit followed by a transmission bit precedes every command sent to the card. This is common to any transmission to the card. A command index follows the start bits which identifies the command number. A different command is sent to the card by populating the command index field with a different command number. The argument field contains bits unique to each command. These must be looked up in the SD Specifications on a command by command basis. Table 4.7.4 of the SD specifications [2] gives the argument bits and their applications. A CRC7 checksum is then calculated for all bits preceding it and inserted in the CRC7 field. CRC7 includes the start bit and the transmission bit. The final end bit signals to the microSD card that the command has finished and the microSD card will then respond at this point with a response. Command and response of the SD mode protocol is shown in Figure 2.4.

Most commands sent to the SD card are replied to. This involves the SD card clocking back a response which has a format based on the command which was sent.
The command line is bidirectional and the host must tristate its physical pin to be able to receive a response from the SD card while the SD card drives the command line. Some commands do not yield a response, though this is rare. The type and format of the response is determined by the command number sent to the SD card. The reset command 0 is one command which does not cause a response from the SD card. Commands are sent by either the microsd_init state machine or the microsd_data state machine through enabling a support process in the VHDL code. That command send process is shown in Figure 2.5.

A command is prepared to be sent by the command send process of Figure 2.5 as shown below in Figure 2.6.

It is in this way that all commands are built and then sent to the SD card. The write enable on the command line is asserted which allows driving the cmd line by the host. As the VHDL process in Figure 2.5 shifts data in the output_command register, the individual bits show up on the cmd line, as cmd_signal is routed to the cmd line elsewhere in the code. This same scheme allows other SD commands to be sent to the microSD card. Of note is the synchronous movement of data onto the cmd line on the falling edge of the clk. The clk described here is the clock both driving the logic as shown as well as the clock provided to the microSD card.
Figure 2.5: Command Send Process of sd_init Core

Figure 2.6: CMD0 State Machine Output Logic
Responses take several different formats. There are 6 different types of response, named R1, R1b, R2, R3, R6, and R7. By far the most common response is the R1 response. The R1 response is the normal response command. It contains the index of the command being responded to and a section of the microSD card status register. This response can thus be used to check that the card received the numbered command sent as well as the state of the card. Several responses are used only once during initialization, such as the R6 response which is associated with retrieving the relative card address once during initialization. The R1 response has the form shown in Figure 2.7.

<table>
<thead>
<tr>
<th>Bit position</th>
<th>47</th>
<th>46</th>
<th>[45:40]</th>
<th>[39:8]</th>
<th>[7:1]</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width (bits)</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>32</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Value</td>
<td>'0'</td>
<td>'0'</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>'1'</td>
</tr>
<tr>
<td>Description</td>
<td>start bit</td>
<td>transmission bit</td>
<td>command index</td>
<td>card status</td>
<td>CRC7</td>
<td>end bit</td>
</tr>
</tbody>
</table>

Table 4-30: Response R1

Figure 2.7: R1 SD Mode Response Format

The host controller in all instances reads back the appropriate response type. In some instances it parses the card status bits to check for SD card state, ready for data, illegal command, and error states. However, different responses have different fields and associated interpretations. Thus the host controller signals the correct response handling VHDL process code depending on which command has been sent. For example, in the case of an R6 response, the host controller reads the response back to acquire the card’s relative card address, which is then used in subsequent commands. The relative card address can be seen in Figure 2.8.

The R1 response handling VHDL process is shown in Figure 2.9. Engaging this process is done from another state machine residing in the VHDL code.
The lines used for communication between the SD card and the host controller are bidirectional. All communication lines are tri-stated between the host controller and the SD card. The state machines internal to the controller appropriately engage write_en on a bidirectional line to drive it or disable the write_en to put the signal
and top level pin into high impedance. Data can then be read from the pin. The VHDL code to implement this is shown in Figure 2.10.

---

```vhdl
--Internal Tri-State of all bidirectional lines.
tri_state_cmd: process(cmd_write_en_signal)
begin
  if (cmd_write_en_signal = '1') then
    sd_cmd <= cmd_top_signal;
    cmd_top_signal_in <= '1';
  else
    sd_cmd <= 'Z';
    cmd_top_signal_in <= sd_cmd;
  end if;
end process;
```

Figure 2.10: Tri-State Communication Signals

**CRC Engines**

Commands with variable payloads are checked by the SD card for transmission errors using a CRC7 checksum algorithm. If the calculated CRC7 of the command packet which is received by the SD card does not match, the SD card ignores the command. The CRC7 algorithm is shown in Figure 2.11. Data transmitted and received to and from the SD card is protected by CRC16 hashing algorithm described by the SD Specifications and shown in Figure 2.12. Code provided by the generator by Kay Gorontzi was adapted to provide the CRC7 and CRC16 engines of the design [3].

During initialization of the card, all commands and payloads are known at compile time. All CRC7 checksums are hard coded into initialization and into the microsd_init state machine. It is not until the data payloads of the data transfer section of the design and the card is addressed using a relative card address, do the CRC components
Figure 2.11: Command CRC7

VHDL code was adapted from the CRC7 engine compiler by Kay Gorontzi [3]. Slight modification of the code generation was needed to implement the needed CRC7 engine, as the web generator provided Verilog code. Examples of calculated CRC7 see usage. This is also why no CRC engines are instantiated inside the microSD_init machine.
and CRC16 checksums are given in the SD specifications. The checksums generated by the engines could thus be verified. Commands received by the SD card that do not have a matching CRC checksum are not responded to. Data blocks which are received with a bad CRC checksum do give acknowledgement to the host of a transmission error through use of the data response token (SD Specification Section 7.3.3.1).

**SD Card Initialization Flow**

A series of commands and responses must be issued to the SD card before it is ready to accept data which it then in turn writes to flash. This process, while largely documented in the SD Specifications, does leave room for interpretation. A flexible FPGA development platform, along with ample use of a Saleae physical logic analyzer and Altera SignalTap allowed the SD host controller to be layered upon and built over time. The state machines grew through trial and error from inability to pass initialization to robust writing and successful recovery of data from the microSD card.

The SD card is initialized through a series of commands which are not related to the data series of commands. This series of commands is termed the card identification mode. These commands are unique to initializing an SDXC type card.

Before any command is sent to the SD card upon bootup, a number of clock cycles must be provided to the card while the card is powered. The specifications requires 74 clock cycles at the default 400kHz for the card to complete initialization. The microSD_init state machine provides these clock cycles. The initialization clocks are followed by a command 0 (CMD0). Command 0 is a reset that brings all cards to the idle state (idle state command). Unique to command 0 is the control of the data 3 SD card pin. If the pin is driven low during the transmission of CMD0, the SD card will enter an SPI mode. It will begin using the data 0 line as a master out slave in
(MOSI) line instead of using the cmd line bidirectionally to respond to commands. Responses which show up on the data 0 line is a good indication that the data 3 line was not properly driven high during command 0. The SD mode is the mode used in this development and thus the data 3 line is driven high during the transmission of the CMD0 to place the card in SD mode.

Command 0 (CMD0) is followed by command 8. The importance of command 8 (CMD8) is that it detects if the card is in compliance with SD specifications version
2.0, which the card under test is. It also sends a voltage range which the host can operate at. In this specific code, this range is specified as 2.7-3.6V. A check pattern is also sent to the card which it echoes back upon the card both being 2.0 compliant and able to function at the supplied voltage range. The initialization VHDL machine checks for the check pattern in the R7 response that comes back. Command 8 (CMD8) is the first command to issue back a response, which uses the R7 type response.

Before the next command, which is an application specific command, another command must be issued. This command is a CMD55 or APP_CMD. All application specific commands must be preceded by a CMD55. Preceding a command with an APP_CMD allows the command numbering to start over and an expanded command set to be sent to the card. In the R1 response that comes back from the card due to a CMD55, there exists a status register, which gives information on the current state of the card. In my code, I check bit field 5 of the status register which is APP_CMD. It says that the card is now expecting an application specific command, which is correct.

The next series of events are the core of initialization. It is during the initialization command (ACMD41) where the SD card goes busy initializing and the host must wait before the card is ready to resume. ACMD41 is also where several important decisions are made. In the ACMD41 command that is sent out, the host indicates whether it supports SDSC type SD cards or SDHC/SDXC type SD cards. Since I used an SDXC type card in development, this field is set to the SDHC/SDXC type. Another important field is the S18R field. It is here that the host indicates that it wants to switch the signaling voltage. The initialization pathway which follows if this switch signaling bit is set is different. The voltage switch pathway is dedicated to its own section. Further description of initialization, if signaling is not being switched, is described next. Since the ACMD41 command is important and central to the initialization of the SD card, it is shown in Figure 2.14.
Once the first ACMD41 is sent, the card responds with a R3 type response. It is in this response that a busy bit is found. If the busy bit (active low) is set to '1', the host can proceed in initialization. The SD card specifications remark that the initial ACMD41 is to be sent repeatedly until the busy bit is deasserted and set to '1'. The VHDL implementation thus repeatedly issues this ACMD41 command and parses the R3 response until the R3 response comes back that the card is ready. After the card indicates ready, two more command/response pairs are exercised before the card leaves the card identification mode.

Command 2 is the return card identification register command. The response to the command returns the card identification (CID) register. It contains information mostly unique to the card manufacturer. This data is currently ignored in this implementation.

Command 3 (CMD3) is much more important. Command 3 causes the card to respond with a relative card address (RCA), which is a 16 bit address. This address is stored by the microsd_controller core and is used to select a card for data transfer. This feature allows multiple SD cards to exist on the same bus, each selectable via the
RCA. Striping of data across multiple SD cards existing on the same bus or interacting with another SD card while another is busy would be facilitated by addressing the card with the RCA. It is also at this point that commands are no longer known at compile time. Future commands beyond this point must include an RCA. Thus all future commands must make use of the CRC7 engine to complete a command transmission successfully.

At this point, the VHDL codebase switches from the initialization portion of the code to the data portion of the code. The signals which go to the SD card are multiplexed between the initialization and data cores based on whether initialization has finished and an RCA has been saved. This allows segmenting the initialization portion of the code and the data portion of the code into separate entities. The main state machine states of the microsd_init core are shown in Figure 2.15.

![Figure 2.15: microsd_init Core FSM states](image)

SD Data Flow

Now initialization of the microSD card is complete once CMD3 has been sent and the RCA is received. Clocking can now occur at a rate of up to 50MHz when the code crosses into the transfer mode, shown in Figure 2.16. The microsd_data entity
and state machine is clocked at the same speed at which communications and data are sent and received from the SD card.

![State Diagram](image)

**Figure 4-9: SD Memory Card State Diagram (data transfer mode)**

**Figure 2.16: Data Transfer Mode**

Control and execution is now given to the microsd data core. This entity contains a larger state machine which in turn walks through the various states needed to successfully write data to the SD card. The VHDL state machine states currently used to guide the microSD card within the data transfer mode is shown in Figure 2.17.

Upon coming out of card identification mode, the SD card is in a stand-by state. Before sending data to the card, a change of card state from stand-by to transfer state
must be accomplished. This is accomplished through sending a command 7 (CMD7) which is a select/deselect card command.

To send data to the card over all data lines d0-d3, an additional command must be sent to the card. In its default bootup state, the card defaults to using only one data line data 0 (d0). The command that is sent is the ACMD6 SET_BUS_WIDTH command. Since it is an application specific type command, it is preceded with a CMD55. The microSD_data state machine handles both the command format, exercise of the CRC7 engine to append CRC, and interpretation of the R1 response that the SD card sends back for a CMD7 and ACMD6. The state machine checks
that the response R1 status register bits indicate the command was legal and that the command CRC verified correctly.

One more mode transition needs to occur before sending data to the card. The SD card can be set to operate in different SD bus interface speed modes. In the case of sending data to the card at the current tested maximum of 50MHz, a speed interface mode change must occur. The SD card defaults to a bus speed mode of SDR12 which is rated at a maximum clock speed of 25Mhz. To operate at 50MHz, a change to SDR25 bus speed mode must occur. This is done by sending a switch function command, command 6 (CMD6). One function group represents one operating condition in which the SD card can be placed. The function group of interest in moving to SDR25 mode is the access mode function group. Changing this function group is done with CMD6 sent with the mode bit set to switch function. Command 6 (CMD6) has two types, a check and a set type command based on a bit field inside the command. One can either set or check the current function group settings. A set of the function group access mode from SDR12 mode to SDR25 mode allows the SD card to be accessed up to 50MHz. The function groups are shown in Figure 2.18 and are detailed in section 4.3.10 Switch Function Command of the SD Specifications [2].

The argument slice (left column of Figure 2.18) of the set corresponds to the argument sent in the CMD6 command. The general structure can be seen in 4.7.4 Detailed Command Description section of the SD specifications [2].

The SD state transitions needed to write to the microSD card at 50MHz over all 4 data lines is now complete. Data is now written to the card by using a multi-block write command, CMD25. Data is streamed to the SD card over all 4 data lines. The host can stream as many blocks to the SD card as wanted, but the host must wait before sending further blocks if the SD card becomes busy writing data.
<table>
<thead>
<tr>
<th>Arg. Slice</th>
<th>[23:20]</th>
<th>[19:16]</th>
<th>[15:12]</th>
<th>[11:8]</th>
<th>[7:4]</th>
<th>[3:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group No.</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Function name</td>
<td>reserved</td>
<td>reserved</td>
<td>Current Limit</td>
<td>Driver Strength</td>
<td>Command system</td>
<td>Access mode$^1$</td>
</tr>
<tr>
<td>0x0</td>
<td>Default$^2$</td>
<td>Default$^2$</td>
<td>200mA</td>
<td>Default$^2$</td>
<td>Type B</td>
<td>Default$^2$ / SDR12</td>
</tr>
<tr>
<td>0x1</td>
<td>Reserved</td>
<td>Reserved</td>
<td>400mA</td>
<td>Type A</td>
<td>For eC</td>
<td>High-Speed / SDR25</td>
</tr>
<tr>
<td>0x2</td>
<td>Reserved</td>
<td>Reserved</td>
<td>600mA</td>
<td>Type C</td>
<td>Reserved</td>
<td>SDR50</td>
</tr>
<tr>
<td>0x3</td>
<td>Reserved</td>
<td>Reserved</td>
<td>800mA</td>
<td>Type D</td>
<td>OTP</td>
<td>SDR104</td>
</tr>
<tr>
<td>0x4</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>ASSD DD5</td>
</tr>
<tr>
<td>0x5</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x6</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x7</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x8</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x9</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xA</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xB</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xC</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>(eSD)</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xD</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xE</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Reserved</td>
<td>Vendor specific</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xF</td>
<td>No influence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note 1: Bus Speed Mode is alias of Access Mode.
Note 2: "Default" of Function 0 means that a function of just after the card initialized.

Table 4-10: Available Functions

Figure 2.18: Available SD Function Groups

Table 4-28 was added in version 1.10

<table>
<thead>
<tr>
<th>CMD INDEX</th>
<th>type</th>
<th>argument</th>
<th>resp</th>
<th>abbreviation</th>
<th>command description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMD6</td>
<td>adtc</td>
<td>[31] Mode 0: Check function 1: Switch function [30:24] reserved (All '0') [23:20] reserved for function group 6 (0h or Fh) [19:16] reserved for function group 5 (0h or Fh) [15:12] function group 4 for current limit [11:8] function group 3 for drive strength [7:4] function group 2 for command system [3:0] function group 1 for access mode</td>
<td>R1</td>
<td>SWITCH_FUNC</td>
<td>Checks switchable function (mode 0) and switch card function (mode 1). See Chapter 4.3.10.</td>
</tr>
</tbody>
</table>

Figure 2.19: CMD6 Command Format
The CMD25 multi-block allows the host to continue sending blocks after the initial CMD25 indefinitely. The number of blocks in a multi-block stream has a large impact on throughput to the SD card. An SD card can become busy in the middle of a CMD25 multi-block write, whereby the host must wait for the SD card to become available again before sending more blocks. The general flow of a CMD25 multi-block write is shown in Figure 2.20 as it occurs across the bus attached to the SD card.

![Figure 2.20: Multiple Block Write Flow](image)

There is more than one way to check if the SD card is available or busy. The currently used method is a polling method using CMD13 SEND_STATUS command. Upon sending any block and its appended CRC, followed by checking that the card has received the data and its CRC matches, the microsd_data state machine continually sends CMD13’s to the SD card. The SD card remains able to respond to CMD13 via R1 response even while busy. A bit field of the response indicates that the SD card is again ready for data.

It is after this CMD13 busy check that the state machine either sends another 512 byte block or terminates the multi-block write via command 12 (CMD12), STOP_TRANSMISSION. Of note is that the data sent to the SD card when 4 bit data mode is enabled must be parallelized across all 4 data lines. The CRC16 checksum is also sent in parallel. Thus, there is a CRC16 calculated for each of
the 4 data lines in parallel and then appended at the end of each 128 bit data sequence. The microSD data entity reads words from the dual port ram contained in microSD buffer. For each falling edge of the clock during a CMD25 data transmission, the microSD data machine splits the upper nibble of a data byte across the four data lines. In addition to doing this, the state machine also splits the bits into four parallel CRC16 engines. A streaming CRC16 checksum is calculated for each of the four data lines and is appended on each data line after the full split 512 byte block has been sent across the data lines. The beauty of the parallel architecture of the FPGA is
seen here where 4 parallel CRC16 engines can be operating in parallel on the split streaming data.

The SD card checks every data block against the CRC which is appended to that block. If the CRC does not match the data which it received, a data response token indicating data rejected due to CRC error is returned. The multi-block write must be stopped with a command 12 (CMD12) and a new command 25 (CMD25) must be sent.

![ CRC Okay Response Diagram ]

Language within the SD Mode section of the SD specification is not ultimately clear on what the response shown in Figure 2.22 represents. An internet blog suggested that the response token takes on a SPI mode control token format. [4] That format is shown in Figure 2.23.

It was initially questionable about what kind of response was coming back from the SD card upon sending a block. The blog’s information was correct and the SPI
7.3.3.1 Data Response Token

Every data block written to the card will be acknowledged by a data response token. It is one byte long and has the following format:

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
x & x & x & 0 & \text{Status} & 1 \\
\end{array}
\]

The meaning of the status bits is defined as follows:

- '010' - Data accepted.
- '101' - Data rejected due to a CRC error.
- '110' - Data Rejected due to a Write Error

![Figure 2.23: Data Response Token](image)

As mentioned, after a CMD25 multi-block send is completed, the microsd_data core stops transmission by sending a CMD12. If this is the end of a data transfer, the microsd_data returns to a wait state. In this state the FSM also gates and stops the clock provided to the SD card. The microsd_data core will resume operation when
signaled, in turn caused by new data being clocked into the microsd_controller by the
master digital controller contained in the top FPGA architecture.

Voltage Switch and Line Signaling Levels

The SD card has several other functional parameters. One of these is the signaling
level of the data, command and clock lines. The signaling level can be either at 1.8V
or 3.3V. The voltage at which the data transmission will occur can be set by a
top level VHDL generic, signalling_18_en, of microSD_controller. This generic will
change both microSD initialization sequences as well as make use of addition ports
of microSD_controller. Voltage level translation requires a level translator IC and
a voltage switching method to enact this voltage change. This is necessary as the
SD card always begins communication at 3.3V during the initialization. This is a
legacy and compatibility feature of the SD protocol and cannot be avoided. The 1.8V
signaling is a requirement for communicating with the microSD card at rates higher
than 50MHz. As initial power measurements suggested that faster clock speeds would
utilize less energy, level translation and speeds above 50MHz were researched.

To achieve the level translation, the TXB0106 6-Bit Bidirectional Voltage-Level
Translator from Texas Instruments [5] was chosen for initial development of the level
translation. This IC was populated to both the BeMicroCV sensor board designed
by Chris Casebeer (2014) as well as the power monitor development board designed
by Emery Newlon and Kyler Callahan (2014). The level translator makes use of two
reference voltages to enact the level translation. This requires a switching mechanism
to change the voltage which exists between the microSD card and the level translator.
In the instance of the BeMicroCV sensor board, the level translation occurred from
3.3V to 1.8V. In the instance of the power monitor development board, the translation
switched back to 1.8V after an initial level translation from 1.8V to 3.3V during initialization. This turned out to lessen the speed at which the TXB0106 could reliably switch data whereas the 3.3V to 1.8V instance of the BeMicroCV was tested up to 50MHz.

The microsd_controller top level component provides 1.8V and 3.3V data signals. These signals, vc_18_on and vc_33_on are meant to be tied to switch enables controlling the voltage present on the reference voltage port of a level translator. In the instance of the TXB0106, two voltage reference ports VCCA and VCCB set the translation levels. The pin out of the chip is shown in Figure 2.24 for clarity. Here we see the VCCA and VCCB reference ports.

![Figure 2.24: TXB0106 Bidirectional Level Translator Pin Out](image)

Either VCCA or VCCB can be pointed towards the SD card as long as VCCA is less than or equal to VCCB. In the implementation on the power monitor development board, the VCCA was host facing and run at 1.8V. The VCCB side was SD card facing. VCCB voltage was controlled by turning on one of two switches whose output was tied to VCCB. The switches in this instance were the Vishay SiP32413 [6]. This switch was then used to enable either 1.8V or 3.3V to the VCCB port of the level
translator, effectively switching from 3.3V signaling to 1.8V signaling to the SD card. The FPGA output pins stayed at 1.8V signaling levels throughout as the clock, data, and command lines of the level shifted SD circuitry were tied to a FPGA 1.8V bank. Figure 2.25 schematically shows the design which was used.

![Figure 2.25: Level Translation Setup](image)

During initialization is when the communication between the host and the SD card is switched from the default 3.3V level to a 1.8V level. This switch is a requirement for data communications above 50MHz. Operating the SD card in a 1.8V signaling mode differs from operating at a 3.3V signaling mode in the first ACMD41 initialization command sent to the SD card. ACMD41 command format is shown in Figure 2.26.

The first ACMD41 sent by the host contains the payload which the SD card considers. Bit field 32 is set to a ‘1’ when the signaling switch is desired. Figure 2.27 shows the string of command responses which allow transition into a level shifted communication state.

A R3 response is sent by the SD card in response to the final AMCD41 indicating initialization complete. In this R3 response, the busy bit is set to a ‘1’ as discussed above. In this instance, this R3 response will also contain an enabled S18A bit. When
this bit is set to '1', the card is ready and capable of switching to a 1.8V signaling level. One case in which the card will not send back an R3 response with S18A enabled is the case in which the card has already been switched to 1.8V signaling. This can happen when the card is reinitialized without being power cycled having previously been switched to 1.8V signaling. I found that a power cycle is required to return to 3.3V signaling.

At this point a CMD11, voltage_switch, is sent to the SD card. It is after the final bits of the R1 response to CMD 11 are sent by the SD card that the SD card drives the cmd and data lines all low. It is not until the card has sensed the clock voltage change to 1.8V does the card drive the lines high again. It is thought that
the SD card actively senses the voltage level on the clock line to determine when to resume communications with the host. Several important elements of the voltage switch sequence were needed to complete it successfully. Previously, the clock line had been left on during the entirety of the switching sequence. However this can lead to a clock which does not switch cleanly between the two voltages, but can track the decay voltage present on the level translator reference voltage port. The clock being supplied to the SD card is shown in blue in Figure 2.28. Such a mistake is shown in Figure 2.28 whereby a clock is continually supplied to the SD card while switching the level translator voltages.

Figure 2.28: Incorrect Clock Handling Voltage Switch
However, if the clock is gated for a period of time before being re-enabled between the 3.3V being turned off to the voltage reference port and the 1.8V being turned onto the voltage reference port, the SD card will respond correctly and drive the data lines high appropriately signaling that initialization can continue. A clean and successful switch is shown in Figure 2.29. The blue waveform illustrates how the clock is gated between voltage switching.

![Figure 2.29: Clean SD Clock Transition](image)

In the instance of a bad clock transition, the SD card will never respond by driving its data lines high again. The periods of time involved in the voltage switch sequence are absent from the public SD specifications. In this controller implementation, a period of 250us is waited between shutting off the 3.3V switch and turning on the
1.8V switch. This is done by properly controlling the vc_33_on and vc_18_on signals. A period of 320us is waited for the 1.8V level to stabilize before the clock is enabled again. These voltage switch sequences are executed by support VHDL processes enabled by the main state machine of microsd_init. The clock must be shut off and only turned on once again when the level translator reference voltage for the SD card side is stable. These timeout periods are determined partly by the level translation chip and switches used and parameters not disclosed in the public SD specifications. The current strategy and timeouts have proven robust with the current implementation.

The host controller state machines waits for the data lines to be driven high again by the SD card. Only then does initialization continue with CMD2 as previously described. Now communications between the level shifter and the SD card can occur at 1.8V.

Performance and Power Usage

As the SD card controller ultimately targets a power limited design, efforts began immediately to minimize the energy needed to write data to the SD card. An A/D system developed around the Freescale Freedom board allowed initial power measurements of the SD card by measuring the voltage across a known accurate sense resistor in series with the VDD pin of the microSD card under test. Initial questions regarding how the SD card’s power profile changes in relation to how data is written to it were investigated. As data can be written at different speeds and with different commands and lengths, these parameters were investigated early. Test VHDL processes were initially written to write 1MB of data to the microSD card beginning at address 0. The activity was recorded with the A/D system and analyzed
in Matlab. Two primary aspects were considered during these initial tests. The frequency at which data is written to the microSD card and the number of blocks streamed after any one CMD25. An article by Arnn Bergman [7] suggested that multi-block number would have a large effect on SD card throughput. A graph from his article shown in Figure 2.30 shows the improvements in utilizing longer multi-block writes particularly in the linear write instance. This article suggested the use of longer multi-block writes to the microSD controller.

![Figure 2.30: SD on Linux Throughput (figure from [7])](image)

It was also important to realize the general current profile the microSD card utilizes when writing data. A period of current activity is found whenever data is being clocked to the SD card. However when the card is not being interacted with, a very low power utilization mode is engaged. It is this low power mode that allows
a lower duty cycle provided by higher throughput to save energy. The low current standby period is shown in Figure 2.31. This is next to a current plateau where 1MB of data is being written to the card.

![Figure 2.31: Standby SD Card Current (less than 1mA)](image)

To initially examine how the write strategy effects power usage, a series of tests were done. A series of tests were performed and the time and current required to write 1MB were recorded over a range of frequencies and multi-block configurations. Two themes emerged. Less energy is required to write a particular amount of data when the data is written faster and also when more blocks are streamed in on any one CMD25 set. The energy to write 64KB (128 blocks) of buffered data to the microSD card using all four data lines versus writing the same 64KB of data using
individual 512 byte blocks in series using a single data line required approximately one quarter the energy. The difference in these transfers involves the amount of data streamed without additional interleaved commands. A 64KB write, termed a 128 block write, involves 128 512 byte blocks streamed continuously to the card before a stop transmission command. This is compared to a series of single 512 byte block writes where only 512 bytes is transferred to the card before a stop transmission is issued and a new CMD25 must be issued. A combination of both writing to the card at high frequency and streaming longer sets of data to the card increases throughput to the card and allows less time to be spent in an active write state. This is summarized in Figure 2.32.

![Energy to Write 1MB to uSD at 3.3V](image)

**Figure 2.32: Energy to Write 1MB of Data. Initial microSD Studies.**

The effective throughput to the SD card thus largely dictates the amount of time the card is in an active write state consuming energy. If the card is kept out of an active write state, this allows the card to remain in a dormant state more of the time.
A comparison of throughput in MB/s to the frequency of writing and the multi-block count is shown in Figure 2.33.

![Figure 2.33: Throughput as a function of multiblock count number and frequency](image)

The idea that single block writes is inefficient is echoed in the SD specifications. This was found to be very true as shown in Figure 2.33. Multiblock writes of 128 blocks were settled on as Bergmann’s studies showed minimal throughput increase for increases past 128 multiblocks. No testing was done above 128 blocks as performance gains beyond are not necessarily needed for this project. Testing was also initially done to test the frequency rate versus energy consumption hypothesis. It was found that high clocking frequencies of data to the SD card resulted in less energy to write
1MB of data to the card. This is again due to increasing throughput to the SD card, which reduced the time the card was in an active write state.

Further Power Usage

Figure 2.34 shows a typical extended current usage profile of microsd_controller’s writing data to the microSDXC card. A TI A/D ADS131E08 [8] chip residing on an ADS131E08EVM-PDK demonstration kit [9] was used to generate the associated graphs. A test VHDL process was written to engage the microSD card writing 50 1MB sets at 0.4 second intervals. 128 512-byte multiblock writes are now the standard method to write data to the microSD card and is the method used in the below figures. This test process was used to get a better sense of performance as throughput could be measured across pulses and averaged using Matlab. A larger test set as compared to initial 1MB tests both allowed more assurance in calculated power measurements and performance numbers. Such a 50MB test is shown in Figure 2.34.

A short initialization spike of current usage has always been seen in development and across A/D systems. This spike has properties faster than the 16kSPS sampling rate of the ADS13E08 ADC used to investigate the microSD card. An oscilloscope trace of this waveform gives better indication of the duration and amplitude of the current usage during this initialization procedure. However, this current pulse has not caused the microSD to fail initialization in this design. A capacitor placed close to the VDD input of the SD card will help alleviate any 3.3V rail droop is now designed in future revisions of the marmoset collar SD recording system.

Flash is a dynamic system that is not perfectly consistent. A recent talk by Bunnie Huang [10] revealed that microSD flash management is very sophisticated since flash chip silicon yield is not perfect and flash is a high volume low price margin business that is constantly evolving. Internal wear leveling algorithms implemented by an
Figure 2.34: 50MB of MicroSD Writing Using a 128 Multi-Block Write Method

internal microcontroller residing in the microSD card creates a performance profile that is not always identical. For example, it has been found that issuing an erase before doing a data write can be beneficial for throughput. This erase before write may allow the internal microcontroller to remap flash blocks before incoming data arrives, thus decreasing both time needed during the actual data write to position the data on flash as well as decrease the current needed to do so at this time. One interesting test is shown in Figure 2.35 where 50MB of data was written to the microSD card. In this case the card had not been erased beforehand and exhibited both lower throughput in areas of the card as well as higher current usage during these low throughput times. This suggests that some flash block remapping operation or
wear leveling is being actively performed by the embedded microcontroller contained in the microSD flash card that is hidden from the user.

Figure 2.35: Throughput decreases occur when the card has not been pre-erased

Efforts to increase throughput to the card will in general decrease energy usage when writing data to the card. These power studies showed that efforts to increase throughput to the card would results in less energy used. As a result of these studies, a buffering strategy was implemented in the marmoset collar system. The buffering strategy was to add a large 64MB low power SDRAM (LPSDRAM), which provides a low power storage space to store all sensor data until it is ready to be written quickly to the microSD flash. The LPSDRAM chosen was the Micron MT48H32M16LF Mobile LPSDR SDRAM [11]. In this way both high frequency and long multi-block
writes can be done, both of which reduce the energy required to write any amount of data. Using internal FPGA block SRAM would most likely be power intensive and use up large amounts of block ram, which can be used elsewhere in the design, thus LPSDRAM was used instead.

**Power Consumption of Level Shifted Mode**

To achieve level translation, a specific Texas Instruments level translator IC was chosen early on in development. The IC selected was the TI TXB0106 [5]. Initial level translation studies and command response development occurred on the BeMicroCV sensor board. It was not until the appropriate power monitor points were created on the power monitor development board that an appropriate power analysis of level shifted versus non-level shifted communications could be conducted. The power consumption of the level translation was targeted as a potential savings of energy due to lower bank voltages needed and lower signaling levels. When development of the first miniaturized collar sensor board was begun, a proper analysis of the level shifted power savings were needed to determine if keeping these translation parts was warranted especially in the case where the miniaturized sensor board is space limited. The results show that the total system power on average per 1 MB write is 20mW greater when using the level translated system. This meant that the level shifting approach using the addition level shift part consumed significantly more power than operating the signaling at 1.8 V.

The two FPGA images used to test system power between level shifted and non-level shifted are nearly identical. The differences in the level shifted scheme are turning on the level translator and shifting data through it from the FPGA’s 1.8V bank versus the 3.3V bank in the direct writing instance. The power monitor
development board was constructed with an entirely separate non-level shifted SD card system utilizing its own set of pins. In this way a shifted bus and a non-shifted bus each have their own FPGA pins. Both shifted and non-shifted can be tested.

 Initialization differences are taken out of the equation as only pure data operations are considered in reported values. The level shifted initialization of the microSD card takes somewhat longer and involves more command response to and from the FPGA. However only data which occurs post initialization are considered. Between the two data sets, the exact same data is sent to the SD card involving the same sequence of commands.

 Both tests utilized a 40Mhz PLL output for SD clocking. 40Mhz was found to be the fastest data rate at which the TXB0106 could reliably transmit data from the 1.8V FPGA bank to the SD card operating at a level translated 1.8V. This was in contrast to previous level translation tests on the BeMicroCV board which reliably worked up to 50MHz. However in this instance the level shift was occurring from 3.3V to 1.8V. The data speeds of the TI TXB0106 decrease when shifting between two identical lower voltages. Figure 2.36 shows system characteristics during a 50MB test write to the SD card without the use of the level translator. All signaling occurs at 3.3V and a 3.3V bank of the FPGA is utilized. The pulses indicate sending exactly 1 MB of data to the SD card. A total of 50 1MB pulses are sent. Important things to notice here are the current usage of the level translator during this test. It is drawing minimal current (trace in not seen since it is 0, hidden by the other curves).

 Figure 2.37 shows when identical data is written using the level translated system a significant current usage is found at the SD reference voltage port of the level translator.

 The total system power was calculated using the current sensed at the power monitor development main power rail input and the voltage at this rail input.
Figure 2.36: Direct SD Write Current

referenced to ground. This is the point where main system power enters the system and is converted to appropriate rail voltages by the system’s buck boost power regulator. This is done by using two of the A/D inputs. In this way the total system
Figure 2.37: Level Shifted SD Write Current
power could be calculated at each sample taken by the A/D. The system power curve is plotted in Figure 2.38 for the direct SD write instance.

![System Power Direct SD Writes](image)

**Figure 2.38: System Power Direct SD Writes**

A Matlab script was written to extract out the power averages from the pulses in both the shifted and non-shifted instances. In this way the average power used to write 1MB of data was extracted 50 times from the data shown in both the shifted and unshifted instances. This data provides the comparison shown in Table 2.1.

Averaged across 50MB of writes in both instances, the level shifted system draws 20mW more power than in the unshifted instance. Thus using the level shifting system solely as a means to save power wasn’t pursued. The level shifting strategy was dropped from the miniaturized final sensor collar due in part to power restrictions.
Table 2.1: Level Shifting Power Comparison

<table>
<thead>
<tr>
<th>Speed(MHz)</th>
<th>Width</th>
<th>Multiblock</th>
<th>Signaling</th>
<th>Speed (MB/s)</th>
<th>Total_System_Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>4</td>
<td>128</td>
<td>1.8</td>
<td>9.89</td>
<td>771.58</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>128</td>
<td>3.3</td>
<td>9.90</td>
<td>752.98</td>
</tr>
</tbody>
</table>

as well as PCB design space considerations. However, future non power limited high speed SD projects will easily be able to take the developed code and transition into data rates greater than 50Mhz that does require the level shifting.

Error Handling

The host controller has the ability to detect CRC errors in data block transmission. A full retry pathway exists when the host controller recognizes that the SD card has rejected a block of data due to a CRC mismatch. At this point the host controller must stop transmission with a command 12 and wait for the card to signal ready for data, before it resumes transmission again. This capability has been tested successfully. The microsd controller buffer will only overwrite a buffered block upon the SD card signaling CRC success on that particular block. Thus the host controller will resend a CRC mismatch block successfully without loss of data.

SPI Mode

The SD card can also be communicated with in an SPI mode. Many features of this mode are similar to that of the SD mode. The code base which was developed was evolved from an initial SPI VHDL code base by Justin Hogan of Montana State University who adapted this from Andrew Shull’s capstone design project code (2010).
This SPI code base included the ability to read but not write SPI mode data. An SPI mode write sequence was added to this code base before transitioning to the SD mode. Several differences of the SPI mode are the single data line reading and writing, the ability to communicate without CRC checksums, and an absolute maximum clock rate of 50MHz. To approach the fuller capabilities of the SD card, the SD mode communication was developed and completed since this allows larger and faster capacity microSD cards to be used.

**SD Loader Entity**

An entity was written to interface with the output buffer of the SDRAM controller and load the microsd_controller buffer appropriately. The interface of the microsd_controller was originally developed to allow streaming input data, whereby addressing into the microSD_controller buffer was not needed. However for the SD card to write data from an addressable memory buffer, an interface component was developed, called sd_loader. In the current collar revision, the microsd_controller interfaces with sd_loader, which interfaces with the output buffer facing the SDRAM controller. The component sd_loader waits for a data ready signal and a number of bytes to write from the SDRAM controller. Sd_loader then takes this amount of data from the SDRAM output buffer and loads it to the microSD_controller internal buffer. The sd_loader position within the system is shown in Figure 2.39.

Additional functionality of the sd_loader is the ability to read and write a buffer associated with magnetic memory. Magnetic memory allows certain elements of the system to persist through power cycles. One very important piece of system persistence is the last written SD memory block address location. This is particularly the case as the microSD controller has yet to implement robust reading. In an effort
to prevent overwriting data on the microSD card after a power loss event the sd_loader entity will fetch the last successfully written write pointer from a buffer associated with magnetic memory before it engages the microSD_controller. This allows the
data logging to resume without overwriting previous data. Thus the sd_loader will update or read the buffer associated with magnetic memory. This buffer is managed by another entity which in turn reads and writes magnetic memory, which is the magmem_controller.
Figure 2.40 shows the top level abstraction of the design and the ports. The user should clock the design with a clock that ranges between 400kHz to 50Mhz. This clock rate will be the rate at which the internal logic is clocked and the rate at which data is transferred to the card. The user should present the component with the number of blocks (512 bytes) to be written, data\_nblocks, and the start address on the card where the blocks will begin to be written, data\_sd\_start\_address. The initialization 400kHz clock is internally generated based on the clk\_divide generic discussed below.

Data residing on data\_input is latched into the internal buffer on the rising edge of clk and when data\_we is asserted. The data\_full flag will go high when the internal buffer is full and the component can no longer accept data. The user should stop
presenting data immediately and de-assert data_we. Upon receiving the number of blocks data_nbblocks, the component will keep the data_full line high until all data is flushed from buffer and written to the card successfully. The component relies on the idea that data_nbblocks amount of data will come from the host. Upon finishing writing all of data_nbblocks the component will reset the internal buffer and wait for another set of inputs and first rising edge of data_we. The component samples data_sd_start_address and data_nbblocks near the first rising edge of data_we after a previously successful transfer or upon first power up.

The microSD card uses 6 lines to communicate with the host, in this case an FPGA. The lines are clock, command, and data 0 through 3. The associated I/O of the component are labeled sd_clk, sd_cmd and sd_dat respectively. The physical microSD card also requires power (3.3V) and ground. The appropriate data, clock, and command lines are tri-stated internally in this component as these lines are bidirectional. The user of this component must take these inout lines and tie them to bidirectional pins at the top of their design. These top level ports should map to pins physically connected to the appropriate SD card pins.

The component was tested from 400kHz to 50MHz successfully. Timing constraint is critically important when using this component on an FPGA and must include use of the appropriate input and output port constraints.

The component writes data via CMD25 of the SD card command set. CMD25 is a streaming write command. It is called the multi-block write. It was found that write speed and efficiency went up dramatically as the number of blocks streamed after any CMD25 increased. This was tested up to 128 blocks which resulted in the greatest write speed of 11MB/s. Thus if the user specifies greater than 128 blocks to write to the card, the data will be sent in 128 block sets. The component will decrease the blocks sent in the CMD25 stream if data_nbblocks is less than 128.
The component makes use of an internal buffer. The buffer allows temporary storage of data to be written before it is written to the SD card. The buffer size can be specified and the buffer makes use of the Altera specific two-port ram.

The operation of data_current_block_written and sd_block_written_flag allow the user to log the last block (512 bytes) which was written successfully. On the rising edge of output signal sd_block_written_flag the data_current_block_written can be sampled. Data_current_block_written signifies the last block written successfully to the card.

![Figure 2.41: microsd_controller Generics](image)

Six generics exist for microsd_controller. Bufsize_size_g specifies the size of the internal buffer. It must be a multiple of 512 bytes. Hs_sdr25_mode is a single bit which should correspond to the frequency of the input clock. The generic should be set 1 when operating above 25Mhz. This bit changes the initialization process of the card via CMD6, setting the card into a higher speed mode. Clk_divide specifies a divide by value used to divide the clock input down to 400kHz. In the case of a 50MHz clock, a 128 clk_divide is used. This divide should appropriately target a clock speed less than or equal to 400kHz as this is used and needed by initialization. Signalling_18_en is used to enable or disable level shifting. When this bit is set
to ‘1’, the controller will attempt to enter a level shifted state by issuing a different initialization sequence to the card as well as exercising the v.33.on_off and v.18.on_off ports appropriately. Block.size.g should be kept at its current value. It is used to calculate the width of the buffer.level port.

The SD card can communicate with 1 data line or 4 data lines. By default the 4 bit mode is used for increased throughput.

Further Developments

The CMD25 multi-block write pathway is the major focus of development so far. The CMD25 4 bit pathway is the only portion of the design currently utilized. However a framework and partial design exists for reading from the card and erasing the card. The single block read, single block write and erase have all been implemented and tested working previously. The finite state machine paths and associated processes for these functions need maintenance to bring them back to working order. Multi-block read however has never been implemented and would need more work than the others.

To achieve higher throughputs, a much faster level translator would be needed. The current design would scale to higher clock and throughput rates. A new unidirectional level translator has been found which is capable of 300 Mbit throughputs. As the direction of data transmission is known at all times by the host controller, directional control could be done on the level translator. Alternatively, two I/O signals from two different FPGA voltage banks could be used, one at 1.8V and the other at 3.3V and then these signals tristated appropriately.

The microSD_controller was developed purely on the FPGA without the use of a Modelsim testbench. This is due to the fact that the communication protocol for SD
is elaborate. The exact behavior of the SD card is also not completely known from the
SD specifications. Yet a test bench could be written whereby the basics responses of
the SD card could be simulated. This might be of utility to future controller additions.
Additional improvements to the design that could be undertaken are listed below:

- Single Block and Multi-block Read
- Microcontroller Programming Interface
- Multi SD Card Bus Management
- Communication at over 50MHz using faster level translator
- SD Communications Testbench and Emulation
- Testing against other SD manufacturers
SEGMENT ASSEMBLY

The marmoset collar has multiple sensors each producing data at its own predefined rate. The sensor data of interest include two channels of microphone audio samples, inertial measurement unit data, and global positioning system data. A specified data format and VHDL assembly entity was created to store data into a customized format which would be stored on the microSD card. After storage onto the microSD card, a parser extracts the data from the microSD card, which is stored in a binary format. Prior to recording, all attached sensors and the associated information they provide must be known and the data format must also be specified. This data recovery will be downloaded to a PC for later analysis of the sensor data across a troop of marmosets for studying animal network communications.

Figure 3.1: Segment Assembler (Flashblock)

The data assembler for the marmoset collar project, which is central to interfacing with all sensors and assembling that data, has been termed flashblock. Flashblock is
a VHDL entity and its associated filename is flashblock.vhd. Figure 3.1 is a simplified depiction of where flashblock fits into the sensor data stream inside the FPGA.

Flashblock has multiple other duties it must perform beyond packing data. It is central to data handling as it controls and signals when critical events of the system occur. It also has control of when and if data is forced to the microSD flash. During shutdown and critical events (such as low power) it holds a central role in pushing data to the microSD card as well as signaling other components that shutdown is occurring. Sensor data interrupt handling by flashblock is parallel in nature but the associated buffering of the sensor data and assembling of data blocks when parallel data streams arrive is serial.

The core components of the flashblock system have been tested and verified. Successful assembly of audio and IMU data along with backend parsing has been demonstrated on the power monitor development board. Other elements of the flashblock system are still in development as other pieces of the marmoset collar are evolving and being written. The core of the flashblock system and successful end-to-end sensor data retrieval has been successfully demonstrated. The structure has been designed to allow additional arbitrary sensor data to be easily added in the future.

**Segment Assembler Format**

The segment assembler makes use of the microSD card addressable data format termed a *block*. Since the minimal addressable amount of data on the microSDXC card is 512 bytes, which is called a block, data is assembled into block length buffers before being sent downstream by the flashblock segment assembler for storage to microSD flash. Individual sensor samples are much smaller than a full block. Thus shorter
amounts of data are assembled into what are termed data segments. A sequential format was defined whereby blocks of data containing multiple segments would be constructed. Segments hold arbitrary types of data arriving in any order. Segments are composed of the data payload itself and a segment trailer. The segment trailer ends any segment. It identifies the type of segment and the amount of data that preceded the trailer in the segment. It should be noted that there is no header to the segment structure.

To identify the beginning of a block, a block number starts any assembled block. MicroSD flash erases can be done to either erase all ones or all zeros to the microSD flash memory. Thus the block numbers never start from zero or end with all ones. The microSD protocol defines a 32 bit address space that is used by all SD protocol addressing for reads and writes. Thus block numbers are also 32 bits in length. The features of the block as laid out on microSD flash are shown in Figure 3.2. The block structure in Figure 3.2 fills the microSD flash sequentially as data is recorded. Features shown include the concept of the segment and segment trailer, the block number at the beginning of the block, and the final segment trailer which always ends the block.

Many types of segments are defined. They differ in the type and source of data they contain. Each segment contains a segment trailer which includes both the segment type and the segment length. This information allows a parser to both identify the amount and type of data contained within the segment. The segment formatting has evolved from the idea that audio is central to the system and also the fastest data input into the system. The length of a segment can be determined when the segment is closed by keeping track of data formed thus far. This is in contrast to putting segment length information at the beginning of the segment. This would require post editing the length once the length is known, which is impractical for storing
Figure 3.2: The Defined Block and a Contained Segment

streams to flash without buffering. This allows a fast streaming sensor, such as the microphone, to consistently build segments. This method of forming blocks allows the flashblock segment assembler to push data it receives immediately downstream without any buffering needed to assemble a segment. The flashblock system keeps the audio segment open by default and only closes the segment upon being interrupted by another data type. This is done because audio data, which is being sampled at 56 kHz, occurs at a frequency 52 times higher than the next fastest sensor, which is the IMU where the accelerometer and gyroscope can be configured to have sampling rates up to 952 Hz.

The currently defined microSD segments and their payload content as handled by flashblock entity are listed in Table 3.1.
<table>
<thead>
<tr>
<th>Segment Number</th>
<th>Segment Description</th>
<th>Segment Size (Bytes)</th>
<th>Segment Field Number</th>
<th>Segment Field Size (Bytes)</th>
<th>Segment Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Padding Segment</td>
<td>Variable</td>
<td>1</td>
<td>n Bytes</td>
<td>Padding bytes used to fill block gaps</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>4 Bytes</td>
<td>Time of Project Complete if can be intermixed with TCLKsel[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>4 Bytes</td>
<td>Time of Project Complete</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>9 Bytes</td>
<td>FPGA Time; The FPGA time at which the status segment was accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>9 Bytes</td>
<td>Last Accelerometer Sample FPGA Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>9 Bytes</td>
<td>Last Magnetometer Sample FPGA Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>9 Bytes</td>
<td>Last Gyroscope Sample FPGA Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>9 Bytes</td>
<td>Last Temperature Sample FPGA Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>9 Bytes</td>
<td>Last Audio Sample FPGA Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>4 Bytes</td>
<td>Real-Time Clock Counter Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>1 Byte</td>
<td>Number of microphones detected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>2 Bytes</td>
<td>Bit Vector of Devices Turned On</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>2 Bytes</td>
<td>Battery Status</td>
</tr>
<tr>
<td>2</td>
<td>Status Segment</td>
<td>73</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>GPS Time Mark Segment</td>
<td>18</td>
<td>1</td>
<td>9 Bytes</td>
<td>GPS Time stamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>9 Bytes</td>
<td>FPGA Time stamp the GPS Time stamp is repeated</td>
</tr>
<tr>
<td>4</td>
<td>GPS Position Segment</td>
<td>28</td>
<td>1</td>
<td>1 Byte</td>
<td>GPNAV Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>12 Bytes</td>
<td>Signed XYZ ECEF Coordinates</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>4 Bytes</td>
<td>X Position Accuracy Ellipse</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>2 Bytes</td>
<td>Position DOP (Dilution of Precision)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>9 Bytes</td>
<td>FPGA Time Associated with the coordinate</td>
</tr>
<tr>
<td>5</td>
<td>IMU Gyroscope Segment</td>
<td>6</td>
<td>1</td>
<td>6 Bytes</td>
<td>Z,Y,X Gyroscopic</td>
</tr>
<tr>
<td>6</td>
<td>IMU Accelerometer</td>
<td>6</td>
<td>1</td>
<td>6 Bytes</td>
<td>Z,Y,X Accelerometer</td>
</tr>
<tr>
<td>7</td>
<td>IMU Magnetometer</td>
<td>6</td>
<td>1</td>
<td>6 Bytes</td>
<td>Z,Y,X Magnetometer</td>
</tr>
<tr>
<td>8</td>
<td>Audio Segment</td>
<td>Variable</td>
<td>1</td>
<td>n Bytes</td>
<td>Audio Sample Bytes</td>
</tr>
<tr>
<td>10</td>
<td>IMU Temperature</td>
<td>2</td>
<td>1</td>
<td>2 Bytes</td>
<td>Temperature</td>
</tr>
<tr>
<td>11</td>
<td>Event Segment</td>
<td>Variable</td>
<td>1</td>
<td>n Bytes</td>
<td>Event Context</td>
</tr>
<tr>
<td>12</td>
<td>Shutdown Segment</td>
<td>1</td>
<td>1</td>
<td>1 Byte</td>
<td>Return for Shutdown</td>
</tr>
</tbody>
</table>

Table 3.1: Defined Segment Types
A test bench has exercised the flashblock entity capabilities. These capabilities include the status segment, GPS time mark segment, GPS position segment, IMU segments, audio segment, and event segment. The shutdown segment is a recent addition to the defined microSD block structure and has not yet been implemented. Development board testing up to this point has focused on developed and the mentioned sensors thus far. To this point the MEMS microphone and IMU 9 axes have been successfully stored to microSD flash and successfully parsed from the resulting microSD binary file. This testing and recovery has occurred on the power monitor development board. Other segments and sensor data await physical testing and recovery.

The segment assembler flashblock allows for timestamps to be inserted into the microSD block stream for identifying when any particular sample occurred in time. The accurate 1us GPS time stamps and the internal FPGA counter time since reset are both included in the status segment. Of note is that the FPGA time and GPS time mark are 9 byte values which are broken down into week number, milliseconds into week, and nanoseconds into millisecond. This format corresponds to the format used by the uBlox GPS. Inserting both times into the data stream will allow the FPGA timestamp to be synchronized to universal time and to correct for clock drift through use of the less often inserted GPS time stamp, which is the mark of absolute time. Since the sampling rates of all sensor data are known for a particular compiled FPGA system, the sampling times are not stored next to each sensor data but the time of the last sample of each data type are stored in the status segment. The exact sample time of any sensor data can then be determined by reading a status segment and moving back in the sensor stream, associating times with sensor data at that sensor’s regular update interval. This also saves space on the microSD card.
Two primary state machines are utilized to assemble blocks and segments within those blocks. The two state machines are named send_item and send_block_item. The associated states of these two finite state machines (FSM) are shown in Figures 3.3 and 3.4.

The flow of processing that flashblock enacts is described next. The two FSMs interact with one another. The flashblock assembler is largely linear when it is engaged in running through its states. The two finite state machines do not execute concurrently but call on one another. The send_item FSM is responsible for responding to the different types of interrupts and jumping to the appropriate data assembly that actually needs to be done given the type of data interrupt being responded to, deciding if a block is full, or responding to a critical event. Send_item also checks whether space is left in the current block through use of a block byte counter. When the block is full or on receipt of a critical interrupt, the execution
Figure 3.4: send_block_item FSM states

of the send_item halts and execution then jumps to the send_block_item FSM. It is in this state machine that data is handled and pushed down stream towards the
microSD flashcard. Each state of send_block_item fills a std_logic_vector buffer and uses a state to empty this buffer sequentially into the dual port RAM, which interfaces between flashblock and the SDRAM controller. In the fully functional system, the buffer between flashblock and the SDRAM controller would be emptied into physical SDRAM by the SDRAM controller. As the SDRAM controller and SDRAM physical implementation are still under test, flashblock interfaces with the buffer which exists between SDRAM controller and the microSD controller system. The microSD system then empties to microSD flash from that buffer. These buffers, like most buffers in the architecture of the marmoset collar, are two port rams implemented with Altera’s Quartus toolchain. Specifically the Quartus MegaFunction wizard can be utilized to generate these structures. Two port RAMs are random access memories implemented using physical embedded memory in the FPGA fabric, often termed block memory. The two port nature of the buffers allows separate clock domains to interface with the two sides of the memory, allowing data to cross the clock domains. The clock used by the SDRAM controller and the microSD controller do not need to be at the same frequency in the instance of the buffer between SDRAM and microSD. This is one major way data is moved between asynchronous clock domains within the FPGA.

In the particular instance of microphone data input at a sampling rate of 56 kHz, there is a possibility of a missed sample due to flashblock processing another segment. This is an issue if the flashblock entity is not clocked at a sufficiently high rate compared to the date input rates. In order to eliminate any missed data, four circular buffers were developed to buffer all incoming IMU and audio data. As data is received by flashblock from the different data sources, the samples are buffered into these circular buffers. Read and write pointers keep track of the data length residing in the buffers. In the instance of slower data, such as gyroscope, accelerometer, and magnetometer data, there is rarely a time when more than one sample exists in these
buffers. Flashblock is currently clocked at 3.6 MHz, which is a much faster clock than the incoming gyroscope or accelerometer data samples, which have a maximum data rate of 952 Hz. Processing these data samples happens very quickly. However in relation to the 56 kHz input rate of microphone data, multiple audio samples need buffer space to avoid overflow. This is seen in simulation when larger segments such as the status segment are constructed.

The data interrupt handler VHDL process, audio_sample, is responsible for reacting to data interrupts and storing these samples in the circular buffers. A section of this process is shown in Figure 3.5.

```
if (audio_data_rdy_follower /= audio_data_rdy) then
  audio_data_rdy_follower <= audio_data_rdy;

  if (audio_data_rdy = '1') then
    audio_data_process_request <= '1';
    audio_sample_fpga_time <= current_fpga_time;
    end if;

elsif (audio_data_processed_follower /= audio_data_processed) then
  audio_data_processed_follower <= audio_data_processed;

  if (audio_data_processed = '1') then
    audio_data_process_request <= '0';
    audio_data_write <= '1';
    end if;

elsif(audio_written_follower /= audio_written) then
  audio_written_follower <= audio_written;

  if (audio_written = '1') then
    if (circ_buffer_rd_audio = circ_buffer_wr_audio) then
      audio_data_write <= '0';
    end if;
  end if;
end if;
```

Figure 3.5: Audio Data Interrupt Handler VHDL Code

This process is also responsible for tracking read and writes pointers into the circular buffers. When the read and write pointers are not equal this process signals
send_item FSM to begin processing the associated data. The audio_data_rdy interrupt will trigger both processing the data into the circular buffer as well as timestamping that audio sample with the FPGA maintained time that tracks time since the last reset has occurred (Figure 3.5, line 2536). On line 2554 the read and write pointers, when equal, will disengage the audio_data_write bit, thus disengaging the two large state machines item_state and item_block_state. The IMU data are handled in a similar fashion.

As the GPS VHDL code was written by another developer, the fetching of GPS timestamp and GPS position data is handled somewhat differently. These data are held in two RAMs maintained by the GPS controller. Interrupt bits provided by the GPS controller signals that the buffers internal to the GPS controller have been refreshed with new position or timestamp information. These bits signal item_state FSM and then item_block_state FSM to fetch this data at predefined locations from within the GPS controller. This involves exercising address, read enable, data, and clock lines which run out of flashblock and into the GPS controller. These direct signals occur within item_block_state. Bytes from the GPS controller are clocked through flashblock and into the input buffer of SDRAM controller appropriately.

The building of the status segments and event segments involves similar themes. An interrupt handler process apart from the main state machines watches for interrupts and triggers the main state machines to process that data. In the item_block_state FSM the associated data bytes are sequentially written out to the associated SDRAM input buffer. A state is associated with every sub segment status field. For example the last audio sample FPGA time field belonging to the status segment is written out in the item_block_state FSM as shown in Figure 3.6.

The item_block_state FSM state BLOCK_STATE_BUFFER will process a buffered segment field to downstream memory. For example in Figure 3.6, the
byte_buffer is filled with a number of bytes (line 1346). The cur_block_state is set to BLOCK_STATE_BUFFER. After BLOCK_STATE_BUFFER state has finished processing the byte_buffer, the next state BLOCK_STATE_STATUS_RTC_TIME will be jumped to.

Data Recovery

Recovery of the microSD data structures can be achieved with the Unix ’dd’ command line utility. Initially a virtual Linux Debian operating system was used to examine microSD card extractions. However it was discovered that the Cygwin environment for Windows incorporates a port of the dd Unix utility. Cygwin is an environment installed onto a Windows machine which emulates the feel of a Unix system by allowing access to Windows ports of many default Unix programs. It can be used in a similar fashion to extract binary data from an attached device.
102400 blocks are read from the microSD card to a directory on the working host in Figure 3.7. If the above file is examined with a hex editor and viewer such as HxD hex viewer, one can identify elements of the defined microSD structure.

A test whereby both microphone and IMU data were assembled and saved on the microSD card followed by correct parsing has been demonstrated. The test involved exercising the accelerometer. Other axes of the IMU were being recorded, but not necessarily being changed in meaningful or recognizable ways. The MEMS microphone was also on and audio segments were formed. In this particular test the IMU was oriented onto 5 of 6 accelerometer faces to test one earth gravity on each of the accelerometer faces. In this binary extraction magnetometer segments, accelerometer segments, gyroscope segments, audio segments, padding segments, and beginning block numbers were all identified and extracted. Figure 3.8 shows one block out of a 102400 block extraction for the described test. The highlighted length can be seen as 512(0x200) bytes long.

Numerous elements of the defined microSD data structure exist within the hexadecimal binary view and extraction. This block and the larger dump has been successfully parsed to analyze its content. This binary extraction is the culmination of the data collection work thus far on the marmoset collar. Four types of sensor data have been successfully assembled and retrieved from microSD flash. Figure 3.9 is an annotated detailed version of the Figure 3.8 block.

In Figure 3.9 all four types of interspersed data can be seen. A large audio segment is seen identified by type 0x08 and 0x72 (114 bytes) in length. A gyroscope segment of type 0x05 and 0x06(6 bytes) in length is seen near the end. A magnetometer segment is seen identified by type 0x07 and 0x06(6 bytes) in length. An accelerometer segment can be seen identified by type 0x06 and 0x06(6 bytes) in length. A padding segment is seen at the very end of the block. The types are defined in Table 3.1. The blocks
continue on past the annotated block. The next block has a block number which has incremented by one. Of note is that the block number as well as all sensor data is stored little endian on the microSD card. This can be identified by the least significant byte stored at the lower address. This is evident in the block number where the least significant byte increments between blocks corresponding to incrementing the value by one. Also of note is the block before the annotated block also has a padding segment at its end which is preceded by an audio segment 0x2C(44 bytes) in length. Other instances of the above data types can be picked out in the positions...
Figure 3.9: Annotated MicroSD Block Recovery

in between the annotations. It is this type of binary extraction that is parsed by the
Matlab function `search_flashblock()` to search and concatenate the data streams into continuous arrays specific to each data stream type.

If the above binary extraction is parsed with the flashblock Matlab parsing function no gaps in data or lost samples have been seen to occur, which means there have been no jumps in block number or lost blocks. In its current tested form, the data parser does not expect nor handle malformed data. Its current implementation is concerned with properly parsing blocks assuming well-formed data has been sent from the various data sources.

**Flashblock Parser**

A function in Matlab `search_flashblock()` has been written to parse the currently tested and written sensor data. The function is currently sensitive to microphone segments, accelerometer segments, gyroscope segments, and magnetometer segments. The function definition is shown in Figure 3.10.

```
function [s,sequence_number,audio_segment_stream,gyro_segment_stream,...
    accel_segment_stream,mag_segment_stream] = ...
    search_flashblock(filename,length_blocks)

%search_flashblock  Parse data structures from the flashblock assembled
%       adcard data.
% output
% s = Array of all read data as uint8.
% sequence_number = Array of processed block numbers.
% audio_segment_stream = Continuous audio stream. 16bit signed.
% gyro_segment_stream = Continuous Gyroscope Data. 32bits. XYZ.
% accel_segment_stream = Continuous Accelerometer Data. 32bits. XYZ.
% mag_segment_stream = Continuous Magnetometer Data. 32bits. XYZ.
% Input
% filename = Filename of adcard dump.
% length_blocks = Number of blocks to read from file.
```

Figure 3.10: `search_flashblock.m` Function Definition
The function returns concatenated arrays of the associated sensor data. Further processing of the IMU data is handled by another script called \textit{IMU\_read}. In its current form \textit{search\_flashblock} does not separate the axes of a particular IMU sensor stream. The separation and full scale interpretation of the IMU data streams happens in \textit{IMU\_read}. Knowledge of how the IMU was set up via the downloaded register set is needed to interpret the data values in the IMU segments. This is due to the different full scales which can be set on each of the IMU’s sensors. For the following example the full scale of the IMU has been set to 245 degrees per second for gyroscope full scale, 2 earth gravities accelerometer full scale, and 4 gauss magnetometer full scale.

The \textit{search\_flashblock} Matlab function reads in the entirety of a binary microSD card as unsigned 8 bit integers. The processing of blocks occurs by first reading the block number from the block’s first 4 bytes. The processing index is advanced 511 bytes to then process the block by ascending address. As identifying segment information is put at the end of the segment this processing from the segment end to beginning is required. Counters are read from the segment trailer length field and types are read from the segment trailer type field. The constants used to identify segments of particular types as used by flashblock and presented in associated SD card data structure documentation are also used in this function. Processing of a segment is based on the length of data as indicated by the segment trailer length field. Audio data is read in a reverse time order as the block is processed in reverse. Once a block is complete the audio segment is corrected temporally by flipping the built audio segment and concatenating it to the growing audio array.

One can plot the block numbers as extracted by \textit{flashblock\_search}. Missing block numbers or gaps in block numbers is indicative of errors in data flow within the FPGA. Checking the extracted block numbers against an incrementing array in Matlab is a useful data parity check. Currently no malformed segment handling capability had
been programmed into the function. Future revisions may throw out blocks which do
not end in a proper segment trailer or if an unknown segment is found within a block.
An initial data parity check is done by comparing the block numbers read against an
incrementing vector to see that all blocks are present and were written in a sequential
order.

If the example binary dump that has been mentioned is processed with \textit{search\_flashblock}
and those results processed with \textit{IMU\_read}, the following accelerometer test can be
graphed against time.

![Figure 3.11: IMU Accelerometer Test Recovery](image)

The automation of the \texttt{flashblock\_search} and \texttt{IMU\_read} interpreted the stored data
correctly. The confirmation is the presence of 1 earth gravity on the axes during the
test. Incorrect storage or incorrect parsing would likely not yield such a result. If the audio segment recorded at the same time as the accelerometer data above is also processed and played back from this test, identifiable room acoustics are easily identifiable. The associated audio segment parsed out of the same microSD binary extraction is shown in Figure 3.12.

![Figure 3.12: Flashblock Assembled and Recovered Audio](image)

The other 8 IMU axes not shown are also parseable and plottable through use of the `search_flashblock` Matlab function.
Further Developments

Currently the audio and inertial measurement unit data have successfully been interfaced to the flashblock entity and the resulting data structures recorded to microSD flash, dumped from microSD flash, parsed, and interpreted successfully. Other segment types such as GPS or status have yet to be verified on a development board. Other elements of flashblock such as its interaction with the SDRAM controller are yet to be tested as development of the SDRAM controller is underway. The pathways for the status segment, GPS segment, events segment, and SDRAM flush functionalities however have been tested through use of a Matlab HDL Verifier testbench and associated Modelsim simulation.

Conclusions

The VHDL entity flashblock allows data coming from multiple sensors to be assembled into a standard format, which allows new sensors to be easily added in the future. This defined format allows a developed parser to extract sensor data from the microSD card after the sensor data has been assembled and saved to the microSD card. Inserted time stamps within the format allow sensor samples to be marked with a 1 us accurate time. Audio and inertial measurement unit data have been assembled and parsed successfully after having been saved to the microSD card. This has occurred on the collar project’s main development board. Other functionality such as assembling status and GPS information has passed simulation studies and will be physically tested soon.
INITIAL SYSTEM POWER STUDIES

To monitor system power usage, an analog to digital converter (ADC) has been selected and used through development to help characterize the system. The current ADC used is the Texas Instruments ADS131E08 [8], an 8 channel ADC capable of sampling up to 16,000 samples per second using the provided software and demonstration kit. Power monitoring points have been added to all sensors and component rails of the power monitor development board so that an active component’s power footprint can be monitored with the given ADC. Using two of the ADC channels, one can measure voltage at the component’s power pin and current through a sense resistor to characterize power utilization of a sensor or component. Power measurements for the other system components discussed in this thesis have been included in their appropriate sections. This section describes two baseline studies used to identify entire system current draw and power. These measurements give approximate battery duration. One baseline study of power is used to identify how long the selected battery system could possibly last given little to no FPGA system activity. Another test where the microSD card system is active provides another baseline. These baselines studies give an initial estimation as to how long the currently specified battery could last from a full charge.

Analog to Digital Converter Setup and Usage

A performance demonstration kit was used to interface with the TI ADS131E08 A/D integrated circuit. This development package is the ADS131E08EVM-PDK [9]. The demonstration kit housing the A/D IC is shown in Figure 4.1. It is the device which was attached to the power monitor development board.
Demonstration software was used to set the demonstration kit into a suitable sampling mode for interface with the power monitor development board. The demonstration software allows downloading and examining the register set on the ADS131E08. A sampling rate of 1000 samples per second was chosen. Other notable setup was the need for a strong ground connection between the ADC demonstration kit and the ground of the power monitor development board. The ADS131E08 can
function in both differential and single ended modes. The ADS131E08 demonstration kit was setup to function in a single ended mode. The single ended mode can function in voltage ranges about a common mode voltage. The common mode voltage is a function of the voltages applied to the A/D analog supply (AVDD) and analog ground (AVSS) pins. Common mode voltages of interest were rail voltages of the FPGA system as well as an input main system voltage of 4.1V. Rail voltages of attached peripherals are 3.3V, 2.5V, and 1.8V. AVDD and AVSS were set to 5V and 0V to allow the A/D to sample correctly about these voltages. The common mode voltage in relation to the allowable voltage swing is detailed in Figure 4.2.

![Figure 4.2: Single-Ended ADS131E08 Configuration](image)

The sampled voltage reported by the A/D is the difference occurring across the inserted series measured sense resistor. When measuring the current draw a sense resistor is inserted in series with the device being measured. In all instances, accurately measured 1 \( \Omega \) and 2.5 \( \Omega \) sense resistors were used.
Power Studies

The battery currently specified is a 550mAh Lithium Manganese Oxide cylindrical battery. Identifying current draw of the power monitor development board as a whole yields a first approximation of recording times. A baseline FPGA image was programmed whereby all sensors and peripherals were disconnected from their associated power rails and minimal FPGA logic was utilized. The amount of FPGA resources utilized is shown in Figure 4.3.

![Fitter Summary](image)

**Figure 4.3: Minimal FPGA Design**

The current and voltage at the point where power enters the power monitor development board are examined with the A/D. These results are exported from the
included demonstration kit interface software and imported to Matlab for plotting and analysis. Short Matlab scripts take the voltage values exported from the A/D and convert them to current or voltage curves given knowledge of the sampling setup and sense resistors. A system power profile can be plotted using both the voltage and current. Accurately measured resistors were used in these studies to convert a measured voltage difference to a current. A voltage divider was used to take system input voltage of 4.1V and divide it to an acceptable common mode voltage range of the A/D. Figure 4.4 shows the interpreted voltage and current at the input to the power monitor development board in the minimal FPGA system case.

Figure 4.4: Total System Current and Voltage
The above current and voltage profile exhibit times of variation. The profiles exhibit periods where by the CPLD is on alone, when the CPLD is programming the FPGA initially from the bitstream flash, when the FPGA is reprogrammed over JTAG by the user, and when the FPGA starts running the JTAG downloaded configuration bitstream. All phases are exhibited in the Figure 4.5.

![Minimal FPGA System Power](image.png)

**Figure 4.5: Typical Programming Profile Breakdown**

The current and voltage vectors of Figure 4.4 can be used to calculate a system power curve.

The minimal total system power is noted to be approximately 360mW when the development system is in a configuration very minimal in nature. Current draw when
the FPGA is running the minimal system is approximately 86mA. This is shown in Figure 4.7.

Of note is the current usage by the CPLD alone. This is an important figure as this is the current draw when the CPLD alone is managing the board and potentially awaiting a real time clock alarm to resume recording. This is a current draw which will likely always be on. This current draw is approximately 2 mA. This is shown in Figure 4.8.

With a minimal system current draw of 86mA as examined here a 550mAh battery can be expected to last 6.4 hours. This provides a relative maximum running time for the system as configured. The project will most likely not approach recording
time any longer than 6 hours. This is most definitely the case as more subsystems are turned on.

Another FPGA and system configuration can be looked at to identify another approximate recording time given a fully charged 550mAh battery. As shown in the microSD card section, power and current profiles of the microSD card actively writing to flash are given. Figure 4.9 shows a test program whereby the FPGA is writing 1MB amounts of data to the microSD card at predefined intervals. The current measurement point of interest is not inside the current pulses themselves. This is due to the duty cycle of writing data to the microSD card. The duty cycle of data write is less than 2% when considering data throughput of 10MB/s to the microSD card.
and input data rate of the microphone at 112.5KB/s. As the microSD card draws less than 1mA of current when not engaged and duty cycle on writing to the SD card is less than 2%, the current point of interest is not the current pulse. It is the baseline system draw when not engaged in writes. This measurement is shown and marked in Figure 4.9.

At the noted current draw of 126mA, the 550mAh hour battery will last approximately 4.3 hours. This figure gives another approximate recording window given a full battery charge. Again this is an upper estimate as all other systems beyond the microSD and the FPGA architecture needed to program the microSD
were not on during this test. As more sensors and peripherals are turned on the current draw will increase and the power window will decrease.

**Future Developments**

Total system current draw when the GPS, IMU, and MEMS microphone are on has not been examined yet. These studies will provide recording windows which are closer to what will occur in the field. Peripherals still in development such as SDRAM and transceiver must be power characterized as well. A closer examination of the FPGA supply rails and I/O bank supplies, as a function of FPGA architecture
and attached I/O activity, might provide additional insight into low power FPGA design. However another revision of the power monitor development board is needed as some voltage of these voltage rails come directly from unmeasurable power planes. No single point can be intercepted for measurement in these cases.

Conclusions

These power measurements are early estimates that do not include all sensors or peripherals. Power usage will increase as more functionality is added to the design, more resources are added to the FPGA, and other devices are brought online. Further efforts will go into utilizing the power reduction pathways built into the FPGA toolsets as well as implementing low power HDL design practices. Power windows suggested by these measurements show upper estimates of 4 to 6 hours given the currently specified battery. Utilization of the solar cells attached to the collar will provide multiple recording windows due to recharge functionality. Other power reduction avenues could eventually be pursued such as fitting a collar design entirely within a CPLD, targeting a FPGA with fewer logical elements, or examining other lower power FPGA offered by vendors such as MicroSemi. While the FPGA does have a larger quiescent current draw than a microcontroller, the design flexibility has allowed the system to handle a wide array of peripherals chosen to best suit the targeted research as well as allowing SDRAM interfacing.
MEMS MICROPHONE

A primary goal of the marmoset collar project is to record the vocalizations between marmosets in a troop as well as the sound environment around the troop of marmosets. To sense the acoustic biome of the common marmoset, a digital MEMS microphone was selected. The selected microphone is the Invensense INMP621 [12]. The originally targeted MEMS microphone was the Analog Devices ADMP521. Midway through development, the Analog Devices MEMS microphone product line was bought by out Invensense. No development changes were needed. This microphone senses incoming acoustic pressure waves using capacitive MEMS technology. Using a fourth order delta sigma modulation in the analog to digital conversion, the microphone outputs a pulse density modulated (PDM) signal which represent the incoming acoustic waves. The microphone has a frequency response of 45 Hz to more than 20 kHz. This frequency response covers the frequency range of marmoset vocalizations, which has most of its vocalization energy at 7 kHz.

![MEMS Microphone Functional Block Diagram](image)

Figure 5.1: MEMS Microphone Functional Block Diagram
The INMP621 is a microphone based on MEMS process technology and comes in a 4 x 3 x 1 mm package. The microphone lends itself well to digital development as the output has already been digitized into a pulse density modulated (PDM) bitstream. For a PDM signal, the density of the pulses encodes the amplitude of the analog audio waveform. A PDM bitstream can be transmitted using a single trace of a printed circuit board. Delta sigma modulation is used by the MEMS microphone to create the PDM bitstream from the analog input waveform. The quantization noise which is incurred by constraining the continuous analog signal to a discrete digital value (0 or 1) is handled by noise shaping circuitry and placed into the high frequency spectrum provided by the faster sampling rates used for PDM. This noise shaping is inherent to the delta sigma modulation technique. For playback and analysis, pulse
code modulation (PCM) is used for storage on the microSD card. This means a conversion from PDM to PCM must take place. Pulse code modulation encodes the amplitude of the analog waveform using many n-bits. Pulse code modulation samples are not as discretely constrained as in the PDM case. The pulse code modulated word encodes signal amplitude using a multibit word. In the case of a 16 bit PCM word, $2^{16} (65536)$ values can be used to represent the original waveform.

To interface with the microphone, a clock must be supplied by the host FPGA. VDD must be supplied as well as a ground GND connection when populating the part to a PCB. A L/R SELECT line dictates phase of output PDM data bits, allowing two microphones to share a single data line enabling stereo functionality. Finally the DATA pin is driven by the microphone to send PDM bits back to the host. In the final collar PCB design, the DATA and CLK lines are routed to the FPGA 3.3V I/O bank. The Invensense microphone allows sampling and generation of PDM bits up to 3.6 MHz as specified by the data sheet. Dividing the FPGA 50 MHz system clock by 14 allows a 3.6 MHz clock to be generated, which is sent out from the FPGA on the pin tied to the microphone’s clock pin. The delta sigma modulation which is done by the microphone on the incoming acoustic signal creates a varying PDM 1 bit value from an analog signal. The PDM bit stream consists of both the original frequency content as well as the quantization noise created when a continuous analog value is turned into a 1 bit signal. To recover a more useful form of the audio data, the PDM bit stream is low pass filtered and down sampled to produce a pulse code modulated word using a cascaded integrator-comb filter. It is this word which represents the amplitude of the original audio pressure wave. Pulse-code modulation (PCM) is a common format used on compact discs and can be played back easily with Matlab functions. For example the compact disc digital audio standard used to encode music onto compact discs uses a 16-bit PCM encoding utilizing a 44100 Hz sampling rate. To
do this conversion from 1 bit PDM to wider word PCM value a filter is needed. This filter is implemented on the FPGA and the VHDL code is generated using Matlab.

The cascaded integrator-comb (CIC) filter allows the PDM bits streamed from the microphone to be stored in microSD flash as signed pulse-code modulation (PCM) word values. PCM encodes the amplitude of the waveform at a selected sampling rate and the samples are stored in the microSD card as PCM values. The stored PCM word values are then read from the microSD card using a Matlab script and further processed or played back. Combining the MEMS microphone, the VHDL code implemented on the FPGA, microSD code, and Matlab dumping scripts one can record the acoustic environment for later playback and analysis. A PCM 16-bit word sampling rate of 56 kHz has been achieved with this system.

The INMP621 can clock PDM data bits back to the host on either the falling edge or rising edge of the sampling clock provided by the host. This is achieved by hooking the L/R SELECT pins to either ground or VDD. In the current implementations multiple microphones are assembled in parallel, each utilizing the L/R SELECT pin connected to ground. This configuration is termed the right channel configuration and data is valid on the rising edge of the host provided clock. In many of the bus

---

Figure 5.4: Encoding and Bitrate Conversion
structures of other sensors and devices worked with in development of the marmoset collar, data is valid on the rising edge of an associated clock. This convention was kept with the MEMS microphones. Figure 5.5 shows PDM bit valid position in relation to the clock.

![Figure 5.5: PDM Data Phase](image)

The datasheet for the INMP621 remarks that in a single microphone application the DATA line is held for a full clock cycle around the positive edge where data is valid. An oscilloscope capture of the DATA and CLK lines of INMP621 is shown in Figure 5.6. This image represents the configuration of a functional system existing on the power monitor development board. The DATA appears to be held for a full clock cycle around the positive clock edge.

**Cascaded Integrator Comb Filter**

Matlab was used to create a cascaded integrator-comb down sampling filter in which the sampling rate of the microphone is reduced and the PDM bit stream is filtered. A 3.6 MHz 1-bit sampling rate is converted to a 56250 Hz 16-bit word sampling rate. It is this filtering that converts the PDM stream to a PCM stream.
This process is shown in Figure 5.4. The Matlab DSP System Toolbox was used to create the CIC decimator. The CIC decimator incorporates decimation of the input PDM signal as well as low pass filtering of the signal. The Matlab DSP System Toolbox function `fdesign.decimator()` can be used to create a CIC decimator of a certain decimation factor. Low pass filter characteristics is specified as well. The Matlab toolbox that allows the filter design to be converted to VHDL is the Filter Design HDL Coder. The `generatehdl()` function will take a filter design object and create the associated VHDL entity and code which realizes the filtering operation. The output word length of the fixed point CIC decimator was set to a word width of 16-bits. This word is a signed 12-bit integer with 4-bits of fraction in the 16-bit fixed
point number. This PCM value is the one which is then sent to the system segment assembler at a rate of 56250 Hz and is also the value which is stored to microSD flash. The CIC filter is an efficient filtering method for both down and up sampling in part because CIC filters do not require multiplication [13]. In the case of FPGA implementation, this eases construction of the filter significantly.

The delta sigma modulation used to create the PDM bits has quantization noise that is inherent in creating the one bit PDM signal, which is shaped towards higher frequencies. The oversampling of the analog waveform performed by the MEMS microphone creates a larger frequency spectrum in which to shift the quantization noise. A noise transfer function is sometimes used to describe noise shaping characteristics of a delta sigma modulator [14]. The quantization noise can be shaped into the spectrum outside of human hearing range. The low pass filter of the CIC decimator thus eliminates the quantization noise, which is transferred into higher frequencies. This is the general theory of how sigma delta modulation achieves a 1-bit word length.

**VHDL Code and Architecture**

![Figure 5.7: Microphone Component Interfacing](image)

The output of the Filter Design HDL Coder requires some modification and interfacing into the collar project system. The output of the HDL Coder generated a component named Hd_16 as shown in Figure 5.7. As the generated CIC filter expects
signed 1 bit values, a conversion of the unsigned 1 bit values which are received on the FPGA pin tied to the data pin of the microphone is needed. This conversion is done by the encapsulating entity Mems_top_16 as shown in Figure 5.7. The VHDL entity ports maps for both mems_top_16 and hd_16 are shown in Figures 5.8 and 5.9.

```vhdl
generic

entity mems_top_16 is
  port(
    clk : IN std_logic;  -- 3.6MHz Clock
    rst_n : IN std_logic;  -- Negative Reset
    clk_enable : IN std_logic;  -- Clock enable of the CIC Filter
    pdm_bit : IN std_logic;  -- Output of Mem Mix. 1 or 0.
    filter_bit : OUT std_logic_vector(IS DOWMTO 0);  -- Signed FFR Word Out
    clock_out : OUT std_logic -- Divide by 64 clock pulse
  );
end mems_top_16;

entity hd_16 is
  port(
    clk : IN std_logic;  -- 3.6MHz Clock
    reset : IN std_logic;  -- Clock enable of the CIC Filter
    positive_reset : IN std_logic;  -- Positive Reset
    filter_in : IN std_logic_vector(IS DOWMTO 1);  -- sfix2 Signed 1 bit
    filter_out : OUT std_logic_vector(IS DOWMTO 0);  -- sfix16_64 Signed 12 integer bits 4 fractional bits
    ce_out : OUT std_logic -- Divide by 64 clock pulse
  );
end hd_16;
```

Figure 5.8: mems_top_16 ports

Figure 5.9: hd_16 ports

The FPGA pin tied to the MEMS microphone DATA pin is mapped as an input std_logic signal. This pin is tied to the mems_top_16 pdm_bit port (Figure 5.8, line 35). Thus this std_logic signal will be a '0' or '1' upon the rising edge of the 3.6 MHz clock provided to the MEMS microphone by the FPGA. The created CIC filter, Hd_16, expects a signed one bit value. This is shown in the hd_16 port map as the filter_in and its associated comment of sfix2 (Figure 5.9, line 59). The filter_in port described by the HDL coder as a sfix2, requires the mapping of the unsigned 1 bit value to the required signed 2 bit value. The encapsulating mems_top_16 entity
handles this mapping. The outputs of the CIC generated filter hd_16 are mapped directly out through mems_top_16 to the segment assembler. The output ports of the hd_16 generated VHDL entity include a filter_out port which is described as a sfix16_E4 by HDL Coder comments (Figure 5.9, line 60). This represents the 16 bit signed PCM word, of which 4 bits are a fractional part. This word becomes valid on the ce_out pulse of the hd_16 entity (Figure 5.9, line 61). This pulse occurs at the new sampling rate of 56250 Hz, down from the original 3.6 MHz as decimation by 64 times has been completed by the CIC filter. It is this pulse which signals the downstream component segment assembler to sample the PCM value from the filter_out port of mems_top_16 into the currently forming audio segment.

The mems_top_16 entity is clocked with the same 3.6 MHz clock as is provided to the MEMS microphone over the physical CLK line between the FPGA and the microphone. The generated CIC filter clocks its logic at 3.6 MHz and in turn provides pulses occurring with a period of 56250 Hz corresponding to the new sampling rate of 56250 Hz.

Microphone Troubleshooting

Several peculiarities and troubleshooting scenarios arose during the development of the MEMS microphone PCBs. Initial implementation of the MEMS microphone occurred on the BeMicroCV sensor board. On this board the MEMS microphone was interfaced with 3.3V logic and power supply and provided the initial successful test and verification of the MEMS microphone implementation. The power monitor development board design switched the MEMS microphone to 1.8V logic and supply in hopes of power savings. The MEMS microphone was hooked to an I/O pin on a FPGA 1.8V bank. The datasheet of the INMP621 remarks that the part can be operated
at 1.8V. However in the tests on the devices used, the part would only initialize and begin clocking PDM output bits if supplied with 2.0V or greater. It was here that the microphone’s inability to start at 1.8V was found. In future miniaturized sensor boards, microphones are operated and connected to I/O associated with 3.3V FPGA banks. Another finding was the likely intolerance of the MEMS microphone to board wash during PCB assembly. Screaming Circuits PCB assembly house was used to assemble the power monitor development board. After assembly it was found that all populated MEMS microphones were inoperable and upon translation of PDM to PCM and storage to microSD, the MEMS microphones were only encoding noise on their data output pins. Upon removing a microphone from the power monitor development board and reflowing on a brand new MEMS microphone, proper intelligible audio was able to be recorded.

**Data Logging**

Audio data has been logged in several different ways throughout development. Development preceding the segment assembler flashblock used serial binary dumps of audio data to the microSD card. Scripts were developed to read the serially logged data. In many of the initial tests non-structured writes of binary data was streamed to the microSD card. Examination of 16-bit versus 20-bit downsampled words involved placing 16-bit and 20-bit values in larger 32-bit values and storing these words next to each other on the microSD card. After 16-bit downsampling size was decided upon, PCM were stored sequentially to microSD as signed 16-bit words. The way in which audio data will be logged to microSD for the final marmoset recording collar will make use of the developed microSD data format and the processing of the data streams by the VHDL component flashblock. In this way audio data is contained
to an identifiable segment on the microSD card. The microphone data is logged to microphone segments assembled to microSD blocks by the segment assembling system, termed *flashblock.vhd*. The *flashblock_search.m* Matlab script has been written to extract the microphone words from the raw blocks dumped from the microSD flash card. The script finds all audio segments based on trailer information and parses data out of the segment byte by byte. Little endianness and signed values are interpreted from the Matlab unit8 data type read by Matlab inside the *search_flashblock.m* script. The segment assembly section of this thesis describes the data format present on the microSD card as well as how *search_flashblock()* works. After processing dumped binary data from the microSD card with *search_flashblock.m*, samples can be played back in Matlab using the *soundsc()* function at the known sampling rate of 56250 Hz. The PCM words can also be plotted against time.

Figure 5.10 shows an example waveform captured by the MEMS microphone. The PCM waveform shows the audio recording of a common marmoset twitter call played from a desktop speaker directed at the inlet port of the MEMS microphone as it exists on the power monitor development board.

The PDM bits were captured from the microphone and converted using the in place CIC downsampler. The segment assembler created audio segments and encapsulated them into 512 byte blocks for storage to the microSD card. Storage to microSD card by the microSD HDL components allowed the PCM words to be dumped from the microSD card and parsed from the binary dump using the *search_flashblock.m* script. The amplitude values shown represent the exact outputs of the hd_16 CIC filter having been interpreted as signed 16-bit with a 4-bit fractional part. As the words are written to microSD in little endian format, byte wise reads of the binary dump must account for this fact. The little endian uint8 reads must be assembled into 16-bit signed values by the script. Once this is complete the waveform can be
played back using the `soundsc()` function of Matlab with the known sampling rate of 56250 Hz. Sound playback is sensitive and clearly intelligible. Human speech tests have also been played back in Matlab and the speech is completely intelligible. A subjective high degree of sensitivity of the microphone has been demonstrated.

Figure 5.11 is a view of the binary data as it exists in a dumped binary file shown in hexadecimal format. The program being used to view the data is the HxD hex editor. The highlighted section above represents a 254 byte long section of converted audio data stored and recovered from the microSD card. The values are 2 byte PCM value representing the amplitudes of the audio waveform. The little endianness can be seen in the LSB that shows more variability than any of the other values.
The segment assembler and microSD block definitions as covered in the segment assembler section dictate the format of the encapsulated data. A segment identifier, 0x08, indicates that the preceding section is audio data. The 0xFE value signifies that...
Figure 5.12: Audio Segment Type and Length

the segment holds 254 bytes of audio data. *The Matlab function search_flashblock.m* can be used to search a binary dump such as the one shown and extract an audio PCM array.
The TI ADS131E08 and associated TI development board ADS131E08EVM-PDK are used to make power measurements on the INMP621 MEMS microphone. Current measurements of the MEMS microphone are as follows. The microphone power measurements are relatively simple as the MEMS microphone functions in two primary states. The microphone is either on or off. The MEMS microphone turns on or off after a clock is present or not present for 32768 cycles. The power measurement points included on the power monitor development board were utilized to measure the MEMS microphone current utilization. The power tests were conducted by physically removing the clock pin from the MEMS microphone using an interconnect wire. When the microphone is actively producing PDM bits, current utilization is 1.28mA as shown in Figure 5.13.

The data sheet for the INMP621 specifies a 1.6mA current draw when the MEMS microphone is operating at 3.3V. In Figure 5.13 we see a sleep mode quiescent current that is approached but is not reached. Another power test is done to see what sleep mode quiescent actually is given the sleep mode current specified in the data sheet. When the microphone enters sleep mode upon the clock line being disabled, the MEMS microphone’s current utilization is 13uA as shown in Figure 5.14. The INMP621 data sheet specifies an 8uA sleep mode current.

The startup current profile shown in the above graphs is reproducible. Like the microSD card the MEMS microphone also has a startup current profile. This suggests the use of a capacitor on the input VCC line to smooth this transient as well as decrease demands of the buck boost regulator and decrease momentary rail droop.
Figure 5.13: Microphone Startup and Record Current Profile

Conclusions

High quality audio data has been implemented and stored to microSD flash on the marmoset collar system. The INMP621 MEMS microphone has been implemented physically on a total of three development boards to date. All have been brought to a functional state. Pairing the MEMS microphone with filter operations realized in VHDL upon the FPGA allows a complete recording solution and will successfully
Figure 5.14: MEMS Microphone Sleep Mode Current Draw

enable the recording of marmoset vocalizations. Robust assembly into segments and writing of the audio data at high speed to microSD allows high sampling rate wide word audio data to be stored without lost samples. This provides an excellent baseline for the first acoustic biome experiments.

Further Characterization

The MEMS microphone could undergo further loudness, input to recorded word dynamic range, and noise floor tests. The actual frequency response of the system is not known beyond the MEMS microphone datasheet.
INERTIAL MEASUREMENT UNIT

A VHDL entity for an inertial measurement unit (IMU) was developed. The targeted IMU was the ST Microelectronics LSM9DS1 [15]. This integrated circuit is based off microelectromechanical systems (MEMS) technology and comes in a small 3.5x3x1 mm package. This inertial measurement unit can measure changes in acceleration, rotation, and change in magnetic field. This is provided by an integrated accelerometer, gyroscope, and magnetometer all using MEMS technology. A visual of the chip and the sense directions is shown in Figure 6.1.

Figure 6.1: LSM9DS1 Sensors and Directions

The IMU is capable of different full scale settings. The initial full scales of interest in this project are +/- 2 earth gravities (g), +/- 4 gauss, and +/- 245 degrees per
second (dps). At these full scale settings the IMU is sensitive down to .061E-3 g acceleration, .14E-3 gauss, and 8.75E-3 dps.

This chip will allow the marmoset collar project to sense primate movement. Changes in acceleration and rotation in reference to the three axis can be sensed and stored to microSD flash. Behavioral information can then be inferred from the extracted data stored on the microSD card. The IMU chip is a 24 pin device and was integrated onto the power monitor development board. Communications with the IMU occurs over a shared serial peripheral interface (SPI) bus. The chip contains functionally partitioned sections. One part of the chip contains the accelerometer and gyroscope and the other the magnetometer. The accelerometer and gyroscope share an addressable register space, while the magnetometer has its own register space. Addresses of the registers in each partition overlap and thus each partition must be communicated with separately. This is done over a shared SPI bus. SPI communications use four lines between a master and a slave. In this instance the master is the FPGA and the slave the IMU. The bus lines involved in SPI are the clock, chip select, master in slave out (MISO), and master out slave in (MOSI) lines. In the instance of the IMU each chip partition has its own MISO pin. The gyroscope and accelerometer have their own MISO line. The magnetometer has its own MISO line. Talking to the chip occurs over a shared MOSI line. This SPI topology is shown in Figure 6.2.

The LSM9DS1 uses a CPOL=0 and CPHA=0 clocking scheme. These SPI options tell us the clock polarity and phase in relation to the data which goes across the SPI bus. In this instance of CPOL and CPHA being 0, data is clocked onto the bus on the falling edge of the master provided clock. Data is latched off the bus on the rising edge of the master provided clock. An example SPI write communication to the accelerometer over the SPI bus is shown in Figure 6.3.
The device which is targeted when the master is sending data over the MOSI line is done using a unique chip select line for the accelerometer and gyroscope or the magnetometer. The physical pins soldered to the PCB pads and routed to FPGA include a chip select pin for the accelerometer and gyroscope and a chip select pin for the magnetometer. In this way data sent to the device over the MOSI can target a partition of the IMU. The device also has two MISO lines, one for each of the partitions. Similar to the chip select lines, the MISO lines must be
correctly multiplexed depending on which partition is being communicated with. A pin description of the IMU of interest is shown in Figure 6.4.

![LSM9DS1 Pins](image)

**Figure 6.4: LSM9DS1 Pins**

**VHDL Code and Architecture**

A VHDL entity and associated Matlab scripts were created to initialize the IMU and retrieve data from it. The outputs of the VHDL entity are data words for each of the 9 axes of the IMU where each sensor has 3 axes. Utilization of previously created SPI communication entities and abstraction layers allowed the IMU VHDL entity to focus on the device itself and sit atop the SPI abstraction design. The LSM9DS1 VHDL architecture overview is shown in Figure 6.5.

Four multipurpose interrupt pins exist on the LSM9DS1. Three of these pins are programmed to represent data ready on each of the three devices. They also can take on different functions such as sensor threshold limit reached. The interrupt pins of the IMU were wired to the FPGA 1.8V bank on the development PCB board. The IMU has been targeted to a 1.8V IO standard through powering the VDDIO pin of
the IMU to 1.8V. These interrupts are brought into the IMU entity of the FPGA and synchronized to the FPGA clock domain using a D flip flop. The IMU entity ports thus include output words for each of the 9 axes of the IMU. The IMU uses 2’s complement 16-bit values to represent the sensed value for any axis. Full scale of any word is programmatically defined by populating a register on the IMU. Thus, the user of the device picks the full scale range for each of the three devices upon writing the correct value to the correct register on startup. Creation of this non-default startup register set is created by a Matlab function which in turn creates a MIF file used to initialize a 1 port ROM on the FPGA. The IMU VHDL entity reads registers from this ROM on startup.

Figure 6.5: LSM9DS1_TOP VHDL Architecture Overview
The flow of Matlab scripts proceeds as follows and is shown in Figure 6.6. 

*LSM9DS1_XL_G_Register_Settings.m* is altered and run to indicate the registers that the user wants to populate on the accelerometer and gyroscope on startup. The same should be done to *LSM9DS1_M_Register_Settings.m*. A list of bit fields is presented in the script for every register existing on the IMU. Inside the scripts the 1 byte registers are broken out into bit fields whereby each bit field is given a short description according to the IMU’s datasheet. The user and developer can in turn alter the bit fields one at a time to target a specific operating state of the IMU. When *LSM9DS1_M_Register_Settings.m* is run it compiles all modified registers and their associated addresses that are different from the power on default values. The MIF file is generated from this list. The MIF file is structured to have a data width of 2 bytes. The first two addresses in the MIF file represent the number of modified gyroscope accelerometer registers and the number of modified magnetometer registers. These two first fields allow the IMU VHDL entity to address each partition of the IMU in turn, knowing how many non-default registers exist in each partition. As the address
spaces overlap this must be done for the IMU VHDL entity to appropriately address each device in turn.

The IMU VHDL component was developed before the IMU was assembled onto a completed power monitor development board. This development occurred through use of the LSM9DS1 data sheet and simulation of the VHDL design. Simulation of the design used Matlab HDL Verifier toolbox to drive Mentor Graphics Modelsim. A Matlab and Modelsim testbench \texttt{LSM9DS1\_top\_tb.m} was written to drive the inputs to the VHDL entity under test and report outputs of the entity. It was in this fashion that development of the IMU was initially completed. Once the IMU was populated to the power monitor development board, a final test and verification of the code was done against the physical integrated circuit.

The IMU VHDL entity is broken into several large sections. The main synchronous state machine is responsible for both initially populating the IMU’s register set as well as reacting to data ready interrupts sent from the IC for each of the 3 devices. Upon startup the state machine reads the IMU initialization register values created by the associated Matlab script and programmed to a 1 port ROM utilizing the Altera Cyclone V embedded memory via a generated memory initialization file (.mif). When the 1 port ROM is instantiated in VHDL, reference to this generated MIF file is added. Startup involves reading a count value from the ROM indicating how many gyroscope and accelerometer register values need to be changed from default. This allows the state machine to read only the correct number of address register value pairs from ROM before moving onto the magnetometer values. Reading data from the ROM requires control of the address, data, and read control lines of the 1 port ROM. The ROM is instantiated internal to the IMU entity.

Figure 6.7 illustrates that the IMU register address is a one byte value and all register values are one byte values. Thus each element of the MIF file has been
defined as 2 bytes wide. In turn, each register address value pair is taken from ROM and sent to the IMU over the SPI MOSI line. Again to target the specific device, the appropriate device chip select line is held low by the VHDL component. This involves either pulling the CS_A/G or CS_M pin of the IMU chip low at the appropriate time. The device’s MISO line is also correctly routed into the SPI abstraction layers at this time. As the SPI abstraction layers spi_command and spi_abstract handle the actual SPI transmission, the chip select and MISO lines from these levels needed to be multiplexed to the appropriate IMU pins. The SPI abstractions are built to only interface with one chip select and MISO line. Thus the IMU entity is responsible for multiplexing the SPI abstraction’s chip select line to the appropriate physical chip select pin existing on the IMU and routing the target MISO line to the SPI abstraction

![Table 21. Accelerometer and gyroscope register address map](image)

**Table 21. Accelerometer and gyroscope register address map**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Register address</th>
<th>Default</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>–</td>
<td>00-03</td>
<td>--</td>
<td>Reserved</td>
</tr>
<tr>
<td>ACT_THS</td>
<td>r/w</td>
<td>04</td>
<td>00000100</td>
<td>00000000</td>
</tr>
<tr>
<td>ACT_DUR</td>
<td>r/w</td>
<td>05</td>
<td>00000101</td>
<td>00000000</td>
</tr>
<tr>
<td>INT_GEN_CFG_XL</td>
<td>r/w</td>
<td>06</td>
<td>00000110</td>
<td>00000000</td>
</tr>
<tr>
<td>INT_GEN_THS_X_XL</td>
<td>r/w</td>
<td>07</td>
<td>00000111</td>
<td>00000000</td>
</tr>
<tr>
<td>INT_GEN_THS_Y_XL</td>
<td>r/w</td>
<td>08</td>
<td>00001000</td>
<td>00000000</td>
</tr>
<tr>
<td>INT_GEN_THS_Z_XL</td>
<td>r/w</td>
<td>09</td>
<td>00001001</td>
<td>00000000</td>
</tr>
<tr>
<td>INT_GEN_DUR_XL</td>
<td>r/w</td>
<td>0A</td>
<td>00001010</td>
<td>00000000</td>
</tr>
<tr>
<td>REFERENCE_G</td>
<td>r/w</td>
<td>0B</td>
<td>00001011</td>
<td>00000000</td>
</tr>
<tr>
<td>INT1_CTRL</td>
<td>r/w</td>
<td>0C</td>
<td>00001100</td>
<td>00000000</td>
</tr>
<tr>
<td>INT2_CTRL</td>
<td>r/w</td>
<td>0D</td>
<td>00001101</td>
<td>00000000</td>
</tr>
<tr>
<td>Reserved</td>
<td>–</td>
<td>0E</td>
<td>--</td>
<td>Reserved</td>
</tr>
<tr>
<td>WHO_AM_L</td>
<td>r</td>
<td>0F</td>
<td>00001111</td>
<td>01101000</td>
</tr>
</tbody>
</table>

Figure 6.7: LSM9DS1 Register Map Example
layers. When any populated register value is sent over the SPI bus, the IMU VHDL state machine stalls until that register send is finished. This allows the multiplexing to occur accurately.

The spi_commands entity has the hand shaking signals command_busy_out and command_done pulse. This allows the IMU VHDL entity to send a full command set to the spi_commands entity and wait for the command to be completely sent to the IMU. This knowledge of when the current command has been finished is needed to correctly multiplex the chip select and MISO lines to the appropriate device being addressed. A single multiplexer select line enables routing of the MISO and chip select lines appropriately to and from either the accelerometer gyroscope or the magnetometer.

A support state machine and process supplements the main state machine by watching for interrupts occurring from the IMU. The data ready interrupt lines are mapped into the entity as inputs and are first synchronized using a D-Flip Flop to properly condition the asynchronous signals. Upon an interrupt, a signal is sent to the main state machine to trigger processing of that particular interrupt. All interrupts are latched and all interrupts are eventually responded to even if they occur simultaneously as in the case of the accelerometer and gyroscope operating at identical sampling rates.

Processing of an interrupt involves reading a set of data output registers existing on the IMU. The register addresses vary for the gyroscope, the accelerometer, or the magnetometer. For each of these internal sensors, there are 6 8-bit registers which need to be read to collect the output data. These are the high and low bytes for each of the X,Y, and Z axes. By default, the ability to burst read register values is enabled on the gyroscope and accelerometer partition. In the instance of reading output data from the accelerometer, the SPI command issued can be formed to be a read type
and target the first accelerometer data register, OUT_X_L_G at address 18h. The
next 5 registers are associated with the accelerometer sensor after this with addresses
ranging from 19h to 1Dh. The addresses are continuous. This allows a burst read to
occur where further clocking of the SPI bus auto increments the addressed register
and the contents of those registers are automatically placed out on the MISO line.
In this way, when any data ready interrupt is seen by the IMU entity, a burst read
of the entire 3D data set is read. In the instance of the magnetometer and its SPI
bus, a burst read bit is part of the SPI read command sent to the IMU. This is in
contrast to the burst read functionality being a register setting for the gyroscope and
accelerometer.

A third support process of the IMU entity is sensitive to the current state of the
main state machine. This process is solely responsible for setting the SPI bus select
multiplexing bit as well as controlling the 1 port ROM control signals. Separate
combinational logic sensitive to the SPI bus multiplexing bit connects the lower SPI
abstraction layers to the correct chip select and MISO lines depending on which
partition of the IMU is currently being communicated with.

As part of the design, the endianness of the data words was considered. In addition
to being able to program the order or endianness of the words which are received from
the IMU via a register set, data sent to the flashblock segment assembler must be in
a correct endianness. Flashblock was developed to expect big endian data and thus
the IMU entity takes little endian word order streamed from the IMU and makes big
endián words before clocking the output data word output ports. Endianness must
be considered throughout the entire data flow all the way to storage on the microSD
card. The 9 axes and the words associated with them are stored in little endian on
the microSD card. The Matlab parser expects this format and the data assembler
flashblock maintains this endián order.
Initial testing of the IMU was conducted with a SPI master test device manufactured by Total Phase. While the VHDL entity had been written, there is utility in quickly testing a populated IC without involving the FPGA. The AARDVARK SPI Host adapter [16] allows connecting a PC to a SPI communicating device such as the LSM9DS1 and issuing SPI commands to it over MOSI and receiving back results over the MISO line. With an input hexadecimal string and knowledge of the IMU communication protocol and register map taken from the data sheet, the IMU can be initially examined without programming the FPGA. This test makes sure that the IMU has been soldered to the board correctly and PCB layout and board design have not introduced errors preventing the IMU from working. An associated Total Phase level shifter board allowed the AARDVARK to interface with the 1.8V signals present on the IMU as the IMU’s I/O rail had been powered with 1.8V. The AARDVARK SPI host adapter is a 3.3V signaling device. Any mistakes made in the VHDL entity code is bypassed with this initial testing and a better initial understanding of the device and how it works can be ascertained. It helps with debugging since if the device is not responding, one can determine if it is the physical device or problems with the VHDL code. The IMU populated and connected on the power monitor development board was tested successfully by reading back the WHOAMI register on the device and verifying that the binary response matches the datasheet value. This was done with the mentioned SPI master device. Both sensor partitions of the IMU have their own WHOAMI registers and both were tested. While an initial understanding of the IMU datasheet provided an initial guess on what register settings would be required for successful initialization of the IMU, it wasn’t until use of the SPI master device that an initialization register set known to work was arrived at. This process involved testing different register configurations using an understanding of the datasheet and viewing the response of the IMU. In
several instances, incorrect register settings resulted in an unresponsive device. A final set of register settings proceeded to enable interrupts and set all three devices into a sampling data mode.

**Targeted Register Set**

A set of register values that setup the IMU for initial testing was arrived at. Knowledge of the data sheet and use of the SPI master testing device allowed for a set register values to be created. The non-default registers that were changed were targeted at programming the interrupts to signal data ready for the accelerometer and gyroscope as well as enabling all three devices from the off state and targeting a sample rate for each device. Sampling rates of 952Hz, the accelerometer and gyroscope maximum, were selected for initial testing. The magnetometer sampling rate was targeted to be 10 samples per second. The LSM9DS1 has an internal FIFO which can be utilized to read previous output words. However the IMU was configured to not use the FIFO and to function in bypass mode. In bypass mode, only the most recent sample for any of the 9 axes is available. As the state machine which is servicing this IMU component is running much faster than any of the device sampling rates there is no possibility in missing a sample. Thus the advantage of using the bypass mode is that samples can be acquired with minimal latency as they become available, whereas the FIFO introduces latency proportional to the buffer depth.

The register set arrived at and programmed to the device for testing is shown in Table 6.1 and described next.

INT1_CTRL at address 0Ch and INT2_CTRL at address 0Dh were changed to enable the INT1_A/G and INT2_A/G interrupt pins. INT1_CTRL was programmed with value 02h to turn on gyroscope data ready on the INT1_A/G pin. INT2_CTRL
CTRL_REG1_G at address 10h was programmed with value C0h. Bits 6 through 8 of this register were changed to value '110' to target a 952 output data rate (ODR) on the gyroscope. This action also effectively turns on the gyroscope and interrupts signaling gyroscope data ready begin to be seen on the associated programmed interrupt pin. This register also contains the settings for full scale on the gyroscope. The defaults are left at 245 degrees per second. In a similar fashion the accelerometer is turned on. CTRL_REG6_XL at address 20h was programmed with value C0h. This programmed
value turns on the accelerometer by programming the bits 6 through 8 with bit string '110' to both turn on the accelerometer and target a 952 ODR on this sensor. The default full scale for the accelerometer is left at 2 G. The magnetometer is turned on by programming CTRL_REG3_M at address 22h with a value of 00h. This enables the magnetometer into continuous-conversion mode where new samples are continually created. They are created and signaled on the magnetometer data ready pin at a 10 Hz interval, which is the default of the CTRL_REG1_M register.

**Obtained Data**

A test procedure utilizing many elements of the power monitor development board was used to successfully capture data from the IMU. The IMU VHDL entity was instantiated into the collar entity used for development on the power monitor development board. Through use of the IMU entity’s startup input signal, the IMU was allowed to only start initialization and data capture upon request by the user. This request was given to the FPGA through use of Altera’s SignalProbe functionality. This debugging feature allows single bits or registers to be actively probed and read through a Quartus graphical user interface. The output words from the IMU entity were output to the flashblock segment assembling entity. The flashblock segment assembling VHDL entity is responsible for assembling the data from multiple sensors with different rates into SD blocks where a block is the minimum addressable amount of data on the microSD card. The IMU test utilized flashblock to assemble proper blocks and store the data onto the microSD card as defined in the recording collar data format (see section Segment Assembly). The microSD controller system comprising of sd_loader and microsd_controller and its sub-entities successfully wrote the IMU data onto the microSD. Utilizing a Cygwin command ‘dd’ and a USB microSD card
reader, the data from the card was dumped to a file on the PC in a binary format. A Matlab function making use of the definitions of the flashblock assembly process and resulting collar microSD data format parsed the data. The parser Matlab function makes use of defined segment types and known data lengths to search the dumped binary file. Further description of the function, flashblock, and microSD card format are discussed in the data assembly section (see section Segment Assembly).

Each data sample of the 9 axes is represented as two bytes of a 2s complement word. If the words are interpreted knowing the programmed full scale settings for the IMU, the words can be interpreted to represent acceleration in earth gravities, rotation in degrees per second, or magnetic field gauss. For the tests done, a full scale of 2 earth gravities, 4 gauss, and 245 degrees per second were chosen. During the flashblock_search function and the imu_read script the endianness of the words saved to flash is reversed and the byte-wise reads of the binary data are transformed into full 16 bit signed integers. Converting two individual byte values into a single 2 byte value is done by multiplying the higher order byte by 256 and adding the lower byte to the upper byte. The sign bit of any word is also checked to see if this word represents a negative value. In this way the raw data saved to the microSD flash card is converted to data representing earth gravity, degrees per second, and gauss. A test on the gyroscope was done to test validity of the collected data. A constant rotation was applied to the power monitor development board about the Z axis. This was done by hand in an accurate way. The IMU state machine was started at the time via the Quartus Signal Probe GUI. After the board was rotated several times about the Z axis, the microSD card was then pulled from the socket. The resulting data is plotted against time in Figures 6.9, 6.10, and 6.11. The rotation about the Z axis of the gyroscope is clearly seen. Rotations about the other axis are not present and this was expected.
In a similar fashion the accelerometer can be tested by aligning the 3 axes of the accelerometer to the earth’s center. The accelerometer should sample one earth gravity in the direction that points towards earth center. The accelerometer test was done by flipping the power monitor development board onto each of its 6 different faces. The IMU entity was started at the appropriate time and the data logged to the microSD card through the flashblock segment and block assembly system. The test provided correct and reasonable data.

Figures 6.12, 6.13, and 6.14 show the accelerometer test results and also indicate correct interpretation and setup of the IMU. At the beginning of the test, the IMU is stationary on a table. An acceleration of 1 gravity can be seen in the positive Z
direction. Upon flipping the IMU upside down, an acceleration of -1 gravity can be seen. A return to 1 gravity in the positive direction is returned to at the end of the test. The IMU was turned on end to test both a positive and negative gravity of the X axis. Only one gravity direction on the Y axis was tested. These tests show an appropriate response from the IMU and suggest a successful program, interpretation, and data flow of the complete system.

Initial development of the marmoset collar is most concerned with angular rate and acceleration data. The magnetometer data is demonstrated successfully read and parsed but interpretation of the data remains.
Figure 6.11: Gyroscope Z Axis Test
Figure 6.12: Accelerometer X Axis Test
Figure 6.13: Accelerometer Y Axis Test
Figure 6.14: Accelerometer Z Axis Test
IMU Power Measurements

The LSM9DS1 data sheet suggests the gyroscope to be the largest draw of power on the IMU. To verify IMU contribution to total system current draw and verify the gyroscope impact on current draw a current measurement of the IMU system was done. This measurement utilized the current sense points added to the power monitor development board. A 1 ohm resistor was put in series with the voltage rail of the IMU. The voltage drop was then measured across the sense resistor using the Texas Instruments ADS131E08 analog to digital converter and associated software. Output from the software was imported to Matlab and current values plotted knowing the resistance value.

Initial current findings show that turning off the gyro reduces the current consumption considerably. Two power configurations were tested. One test had the gyroscope, accelerometer, and magnetometer all turned on. Sampling rates were 952 samples per second for the accelerometer and gyroscope and 10 samples per second for the magnetometer. This is the full register set as previously described and shown in Table 6.1. This power test is shown in Figure 6.15.

Figure 6.15 shows a current pulse near the beginning of sampling. This may possibly be an inrush of current to the IMU upon the switch connected to the IMU being turned on by the CPLD. The active sampling current occurs after this current pulse. This point and points after represent the IMU axes being turned on via register population. The IMU is now sampling and presenting samples from approximately 20 seconds in Figure 6.15.

Another configuration turned the gyroscope off by not populating the CTRL_REG1_G register responsible for turning on the gyroscope. By not populating this register, the default register value is a power down state for the gyroscope. The accelerometer was
left on at 952 samples per second and the magnetometer at 10 samples per second. The current profile of Figure 6.16 was captured.

It can be seen by comparing the two that turning off the gyroscope saves considerable current draw at the LSM9DS1’s VDD pin. If an average of the activity sections shown in Figures 6.15 and 6.16 is taken the results shown in Table 6.2 are arrived at.

The gyroscope appears to draw approximately 5 times more power when enabled. Of note is that these register settings are not power optimized settings. Maximum sampling rates were targeted which most likely implies higher current consumption. The LSM9DS1 datasheet specifically indicates that lower sampling rates on the
gyroscope will require less current. Other separate register settings may allow for lower current profiles of the IMU.
The IMU has the capability to record temperature data as well. Many of the hooks needed to report this temperature data to outside the IMU entity are present. Temperature data was not a high priority and other possibly more accurate sources of temperature data are present on the collar system. Another possible further development would be utilization of the INT_M pin of the IMU and the INT_M port of the IMU entity. Currently this interrupt system is unused and no process checks or responds to this interrupt. The functionality associated with this pin was not needed. Other useful features of the IMU such as threshold detection and filtering stages present in the IMU have not been investigated. The implementation targeted has been concerned with simply obtaining the raw sensed data words to microSD flash for later retrieval and analysis.
FINAL REMARKS

Further Developments

The marmoset recording collar has not been brought to completion although an initial structure for active recording has been developed and demonstrated. There are still areas of the project actively being worked on. These include the SDRAM, GPS, real-time clock and battery monitor systems. The radio transceiver controller and wireless packet structure is under development. Other project aspects such as the data collection as it will exist in Brazil and the communications networks needed to enact this are in development. A dual band communication network is being developed whereby the collar communicates with base stations using sub-1 Ghz radio transmission. A ham radio network then aggregates status data back to a central collection station. A collar which is wearable by the marmoset is also actively being developed through iterative plastic 3D printing. Finally the data must be aggregated across a troop of marmosets and mined in a correlation and causation animal behavior study. A Shannon information theoretic approach will be central to this data mining to look at possible causality between the different time series data sets. The accumulated data will allow researchers to investigate novel questions in animal behavior.

Future Directions

The platform thus far allows for many future experiments and extensions of the base system. While the current platform is initially power limited due to the size of the target animal, future animal studies might target larger animals. A larger animal
and thus larger power budget will increase recording time and also allow the addition of additional sensors to the daughter boards. Experiments targeting a lab setting would be less sensitive to power limitations since researchers could simply exchange batteries between experiments.

FPGAs lend themselves very well to real-time DSP processing, allowing real-time filtering operations to be performed. This platform is also in the process of being targeted as a marmoset cochlear implant preprocessor where it will be performing real-time frequency shifting. This platform will map a marmosets hearing range to the frequencies which a human cochlear processor is designed for, allowing off-the-shelf human cochlear processors to be used with a marmoset. These signal processing frequency shift operations can be done in real-time utilizing the FPGA. Filtering operations done upon the microphone input data can be altered through processing in the FPGA and sent to the cochlear processor or the information can be stored to the microSD flash. A initial digital to analog circuit has also been investigated which would allow the collar system to interface with the analog input of cochlear processors. Using the FPGA systems developed thus far allows many experiments in auditory neurophysiology which up to this point have not been explored.

It is envisioned that neural recording functionality be eventually added to the recording system as well. Small biosense amplifier ASICs could record the neural activity in the auditory cortex. Intan offers biomonitoring integrated circuits allowing neural signals to be digitized where the data is sent via a SPI bus for data acquisition. Correlation between acoustic patterns and neural activity would allow neuroscientists to study the primate auditory cortex in a natural behaving context. This type of study is termed auditory scene analysis. Pairing what the marmoset is hearing with what the auditory nerves are doing will yield much insight into what certain areas of the auditory cortex respond too.
Other sensors are also targeted for further revisions of the sensor board. A MEMS ultrasonic transducers will allow sensing frequencies greater than what is possible now with the current MEMS microphone. Adding this sensor would allow characterization of the auditory scene above 20 kHz via ultrasonic transducer and allow monitoring the full range of marmoset hearing. While the marmoset does not vocalize above 20kHz, marmosets are able to hear up to 40kHz. The need for an ultrasonic microphone is justified here.

Finally, as the design of the collar system has been done in the VHDL hardware description language, a design could be hardened into an application specific integrated circuit (ASIC). Doing so would reduce power consumption considerably if an ASIC targeting a specific usage scenario is warranted.

Conclusions

A powerful and flexible system for recording a significant amount of data on a platform small enough to be used on small primates is being developed. The system is power limited yet it provides enough recording time and processing power to be useful. The power budget and feasibility of collecting data given the FPGA centric system was one of the collar projects initial questions. Development and verification have proven many vital components within the system, given a specific power window. End to end verification of the recording system has been demonstrated. Sensor data has been taken through assembly, storage to the microSD card, and parsed successfully. The microSD, MEMS microphone, and IMU systems have been demonstrated successfully on the power monitor development board. The microSD system has been developed to exercise a large advanced section of the SD command protocol writing data at high speeds. Power studies on the microSD card and developed controller have optimized
the subsystem for this power limited collar design. The inertial measurement unit has been programmed and rotation, acceleration, and magnetic sense retrieved from the IC. The MEMS microphone has been brought to a state whereby audio data can begin to be collected and stored without loss of samples at high bitrates. Debug of multiple development boards from project inception has allowed a place for FPGA development to occur and optimization of new PCB designs.
REFERENCES CITED


APPENDICES
APPENDIX A

VHDL CODE
Filename: microsd_controller.vhd

Description: Source code for microsd serial data logger

Author: Christopher Casebeer

Lab: Dr. Snider

Department: Electrical and Computer Engineering

Institution: Montana State University

Support: This work was supported under NSF award No. DBI-1254309

Creation Date: June 2014

Version 1.0

Modification History (give date, author, description)

None

Please send bug reports and enhancement requests to
Dr. Snider at rksnider@ece.montana.edu

This software is released under

The MIT License (MIT)

Copyright (C) 2014 Christopher C. Casebeer and Ross K. Snider

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;

― Library used for dual port ram.
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

― ! @brief microsd_controller is the top component of the
serial logging design.
― ! @details
― !
― ! @param clk_freq_g Frequency of the input
clk. Used to generate
new clocks and timeout
values.
― ! @param buf_size_g Number of bytes in the
buffer. Must be specified as N * 512 bytes.
― !
― ! @param buf_size_g Size of microsd_controller
internal buffer, bytes.
― ! @param block_size_g Size of a sd card block.
Minimal addressable data size.
― !
― ! @param hs_sdr25_mode_g When operating clk @ 25
Mhz or below, set '0'. @ above 25Mhz set to '1'.
― !
― ! @param clk_divide_g The sd card init clock
should be close to 400k.
― ! Specify this number to
divide the
~400k init clock.
― ! @param signalling_18_en_g Set this bit to either
attempt to change initialization to
switch into 1.8V
signalling mode. This requires a differet ACMD41 in init
― ! as well as control of a
level shifter in use with the lines of the sdcard.
― !
― ! @param rst_n Reset will cause the card
to reinitialize immediately back
to the waiting for data state.

---!

- @param clk Data Transmission Clock. Clock can be from 400kHz to 50MHz depending on timing of target device.

---!

- @param clock_enable Disable the component. The component will finish its current write if writing, and then gate its clock.

---!

- @param data_input Data presented a byte at a time. This is written into the component’s buffer.

---!

- @param data_we Data is clocked into the internal buffer on the rising edge. Host should control this clock appropriately.

---!

- @param data_full Buffer is full. Stop sending data.

---!

- @param data_sd_start_address The beginning block address on the card should be written.

---!

- @param data_nblocks The number of blocks the host intends to send to component and write to SD card.

---!

- @param data_current_block_written This is the sd block address which was last written successfully. It passed CRC response check.

---!

- @param sd_block_written_flag This flag is pulsed on successful block write.

---!

- data_current_block_written is valid.
entity microsd_controller is
  generic (
    clk_freq_g : natural := 50E6;
    buf_size_g : natural := 2048;
    block_size_g : natural := 512;
    hs_sdr25_mode_g : std_logic := '1';
    clk_divide_g : natural := 128;
    signalling_18_en_g : std_logic := '0';
  );
  port ( 
    rst_n : in std_logic;
    clk : in std_logic;
    clock_enable : in std_logic;
    data_input : in std_logic_vector(7 downto 0);
    data_we : in std_logic;
    
    @param sd_clk
    sd_clk wired to IO bank.
    @param sd_cmd
    sd_cmd wired to IO bank.
    @param sd_dat
    sd_dat wired to IO bank.
    @param v_3_3_on_off
    Wired to 3.3 switch on/off control.
    @param v_1_8_on_off
    Wired to 1.8 switch on/off control.
    @param init_start
    Start the init process.
    @param user_led_n_out
    Data FSM encoding used for LEDs.

    level translator
    sd card side bank control.
    level translator
    sd card side bank control.
    level translator
    sd card side bank control.

    Output of switch goes to
    Output of switch goes to
    Output of switch goes to

    Output of switch goes to

    Data FSM encoding used for LEDs.
data_full : out std_logic;

data_sd_start_address : in std_logic_vector(31 downto 0);
data_nblocks : in std_logic_vector(31 downto 0);

data_current_block_written : out std_logic_vector (31 downto 0);

sd_block_written_flag : out std_logic;
buffer_level : out std_logic_vector (natural(trunc(log2(real(buf_size_g/block_size_g)))) downto 0);

sd_clk : out std_logic;

sd_cmd : inout std_logic;

sd_dat : inout std_logic_vector (3 downto 0);

v_3_3_on_off : out std_logic;
v_1_8_on_off : out std_logic;

—Personal Debug for now

init_start : in std_logic;
user_led_n_out : out std_logic_vector(3 downto 0);

ext_trigger : out std_logic;

end microsd_controller;
A chunk of data of N blocks (a block is 512 bytes) is signalled to be written through use of data_nbblocks.
The card expects the N blocks to flow through its internal buffer. The component should be presented with data only on the rising edge of mem_clk and not when data_full is '1';
The number of blocks to be written to the card will be written starting at sd card block address data_sd_start_address.
--data_nbblocks represents the number of blocks (512 bytes each) which will be sent to the component’s buffers and written to the sd card starting at data_sd_start_address. The component will reset its buffers after it has written data_nbblocks blocks.
The component signals with the sd_block_written_flag pulse that the last block written at data_current_block_written address was successful. The bidirectional lines are tri-states internally.
--sd_cmd and sd_dat thus must be inout to top entity and tied to bidirectional pins.
--By default the card will transmit data at 3.3V signalling level.
--To achieve a 1.8V signalling level a level translator must be present between the FPGA GPIO and the sd card.
--The outputs of the switches are tied together and routed to the voltage reference port of the level translator.
--The component will handle the switching of the sd card side supply voltage pin.
--of the level translator. An internal signal of the design can also be changed easily to run the card in 3.3V mode if so desired.

architecture Behavioral of microsd_controller is

component microsd_controller_inner
generic(
clk_divide_g : natural;
clk_freq_g : natural;
signalling_18_en_g : std_logic);
port(
clk : in std_logic;
rst_n : in std_logic;
sd_init_start : in std_logic;
sd_control : in std_logic_vector(7 downto 0);
sd_status : out std_logic_vector(7 downto 0);
block_read_sd_addr : in std_logic_vector(31 downto 0);
block_byte_data : out std_logic_vector(7 downto 0);
block_byte_wren : out std_logic;
block_byte_addr : out std_logic_vector(8 downto 0);
block_write_sd_addr : in std_logic_vector(31 downto 0);
block_write_data : in std_logic_vector(7 downto 0);
um_blocks_to_write : in integer range 0 to 2**16 - 1;
ram_read_address : out std_logic_vector(8 downto 0);
erase_start : in std_logic_vector(31 downto 0);
erase_end : in std_logic_vector(31 downto 0);
dat0 : out std_logic;
dat1 : out std_logic;
dat2 : out std_logic;
dat3 : out std_logic;
cmd : out std_logic;
sclk : out std_logic;
prev_block_write_sd_addr : out std_logic_vector(31 downto 0);
prev_block_write_sd_addr_pulse : out std_logic;

cmd_write_en : out std_logic;
D0_write_en : out std_logic;
D1_write_en : out std_logic;
D2_write_en : out std_logic;
D3_write_en : out std_logic;

cmd_signal_in : in std_logic;
D0_signal_in : in std_logic;
D1_signal_in : in std_logic;
D2_signal_in : in std_logic;
D3_signal_in : in std_logic;

vc_18_on : out std_logic;
vc_33_on : out std_logic;

hs_sdr25_mode_en : in std_logic;
state_leds : out std_logic_vector(3 downto 0);
restart : out std_logic;
init_done : out std_logic;
ext_trigger : out std_logic

--Debug ram. I read data off the sd card and store it here.
--Debuggable as its single clock single port.
-- component ram_1_port IS
  -- PORT
  -- (  
  --   -- address : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
  --   -- clock : IN STD_LOGIC := '1';
  --   -- data : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
  --   -- wren : IN STD_LOGIC ;
  --   -- q : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  -- );
  -- END component;
component microsd_buffer is
    generic(
        buf_size_g : natural;
        block_size_g : natural
    );
    port(
        rst_n : in std_logic;
        clk : in std_logic;
        mem_address : in std_logic_vector (8 downto 0);
        data_out : out std_logic_vector (7 downto 0);
        data_input : in std_logic_vector (7 downto 0);
        -- data_clk : in std_logic;
        data_we : in std_logic;
        data_full : out std_logic;
        sd_write_rdy : out std_logic;
        sd_write_done : in std_logic;
        buffer_reinit_done : out std_logic;
        data_nbblocks : in std_logic_vector (31 downto 0);
        sd_block_written : in std_logic;
        buffer_level : out std_logic_vector (natural (trunc (log2 (real (buf_size_g/block_size_g)) downto 0)));
        init_start : in std_logic
    );
end component;

--- Signals to be sent to card.
signal dat0_top_signal : std_logic;
signal dat1_top_signal : std_logic;
signal dat2_top_signal : std_logic;
signal dat3_top_signal : std_logic;
signal sclk_top_signal : std_logic;
signal cmd_top_signal : std_logic;

--- Tri-State Read Signals
signal cmd_top_signal_in : std_logic;
signal D0_top_signal_in : std_logic;
signal D1_top_signal_in : std_logic;
signal D2_top_signal_in : std_logic;
signal D3_top_signal_in : std_logic;

-- sd_data will leave its APP_WAIT state to execute different commands.
signal sd_control_signal : std_logic_vector(7 downto 0);

-- Byte encoding of the current state of sd_data. Not complete or unique.
signal sd_status_signal : std_logic_vector(7 downto 0);

-- Where on sd card a block will be read.
signal block_read_sd_addr_signal : std_logic_vector(31 downto 0);

-- Read data from SD card memory
signal block_byte_data_top : std_logic_vector(7 downto 0);

-- Signals that a data byte has been read. Ram wr_en.
signal block_byte_wren_top : std_logic;

-- Address to write read data to in ram.
signal block_byte_addr_top : std_logic_vector(8 downto 0);

-- The address for the current CMD25
signal block_write_sd_addr_signal : std_logic_vector(31 downto 0);

-- Data sent to card
signal block_write_data_signal : std_logic_vector(7 downto 0);

-- Number of blocks to be sent in any multiblock write.
signal num_blocks_to_write_signal : integer range 0 to 2**16 - 1;

-- Start address for an erase.
signal erase_start_signal : std_logic_vector(31 downto 0);

-- End address for an erase.
signal erase_end_signal : std_logic_vector(31 downto 0);
-- Once off control signal for data_buffer_to_sd_data_handler process
signal stop_write : std_logic;
-- Used to simply push sd_data current state to Leds.
signal state_leds_top : std_logic_vector(3 downto 0);

-- Enables for the tri-state output
signal cmd_write_en_signal : std_logic;
signal D0_write_en_signal : std_logic;
signal D1_write_en_signal : std_logic;
signal D2_write_en_signal : std_logic;
signal D3_write_en_signal : std_logic;

-- Where sd_data is going to fetch its data from to write to card
signal ram_read_address_top : std_logic_vector(8 downto 0);

-- Counter to keep track of how many blocks have been written to card
-- using multiple cmd25s.
signal num_of_blocks_written : natural;
-- sd_init has finished init.
signal init_done_top : std_logic;

-- Bit used to enable CMD6 hs_sdr25 mode transition
-- before first CMD25 in the data core.
signal hs_sdr25_mode_en : std_logic;

-- ext_trigger bit which can be used for an oscpe.
signal ext_trigger_top : std_logic;

-- Bit signifying that buffer is either full OR data_nbblocks number of
-- blocks has streamed and the card must first flush the buffer to the card.
signal buf_ful_top : std_logic := '0';
-- Bit signifying that buffer has at least 1 writeable block in it.
signal sd_write_rdy_top : std_logic := '0';

-- clk_en gating signal.
signal shut_off : std_logic;

-- Main system clock. Renamed signal to allow shut off with gating.
signal clk_internal : std_logic;

-- A CMD25 stream has finished.
signal sd_write_done_internal : std_logic;

-- data_buffer has finished reinit.
signal buffer_reinit_done_internal : std_logic;

-- sd_block_written_flag. Last block finished crc receive check by the card.
signal sd_block_written_internal : std_logic;

-- Used to detect if the mem_clk has been started by the host.
signal mem_clk_started : std_logic;
signal mem_clk_started_follower : std_logic;
signal data_we_active : std_logic;
signal data_we_active_r : std_logic;
signal data_we_active_r_follower : std_logic;

signal restart_signal : std_logic;

begin

sd_block_written_flag <= sd_block_written_internal;

user_led_n_out(3 downto 0) <= not state_leds_top;

sd_clk <= sclk_top_signal;

i_microsd_controller_inner_0 : microsd_controller_inner
  generic map (
clk_divide_g => clk_divide_g,
clk_freq_g => clk_freq_g,
signalling_18_en_g => signalling_18_en_g
)
port map (clk => clk,
         rst_n => rst_n,
         sd_init_start => init_start,
         sd_control => sd_control_signal,
         sd_status => sd_status_signal,
         block_read_sd_addr => block_read_sd_addr_signal,
         cmd => cmd_top_signal,
         sclk => sclk_top_signal,
         dat0 => dat0_top_signal,
         dat1 => dat1_top_signal,
         dat2 => dat2_top_signal,
         dat3 => dat3_top_signal,
         block_write_sd_addr =>
             block_write_sd_addr_signal,
         block_write_data =>
             block_write_data_signal,
         num_blocks_to_write =>
             num_blocks_to_write_signal,
         block_byte_data => block_byte_data_top,
         block_byte_wren => block_byte_wren_top,
         block_byte_addr => block_byte_addr_top,
         erase_start => erase_start_signal,
         erase_end => erase_end_signal,
         cmd_write_en => cmd_write_en_signal,
         D0_write_en => D0_write_en_signal,
         D1_write_en => D1_write_en_signal,
         D2_write_en => D2_write_en_signal,
         D3_write_en => D3_write_en_signal,
         state_leds => state_leds_top,
         ram_read_address => ram_read_address_top,
         init_done => init_done_top,
hs_sdr25_mode_en    => hs_sdr25_mode_en,
vc_18_on            => V_1_8_ON_OFF,
vc_33_on            => V_3_3_ON_OFF,

cmd_signal_in       => cmd_top_signal_in,
D0_signal_in        => D0_top_signal_in,
D1_signal_in        => D1_top_signal_in,
D2_signal_in        => D2_top_signal_in,
D3_signal_in        => D3_top_signal_in,
restart             => restart_signal,
prev_block_write_sd_addr =>
                      data_current_block_written,
prev_block_write_sd_addr_pulse =>
                      sd_block_written_internal,
ext_trigger         => ext_trigger
);

i_data_buffer_0: microsd_buffer
generic map (    buf_size_g => buf_size_g,
                 block_size_g => block_size_g)
port map (    rst_n => rst_n,
              clk => clk,
              mem_address => ram_read_address_top,
              data_out => block_write_data_signal,
              data_input => data_input,
              data_we => data_we,
              data_clk => data_clk,
              data_full => data_full,
              sd_write_rdy => sd_write_rdy_top,
              sd_write_done => sd_write_done_internal,
              buffer_reinit_done => buffer_reinit_done_internal,
              data_nblocks => data_nblocks,
              sd_block_written => sd_block_written_internal,
              buffer_level => buffer_level,
              init_start => init_start
read_to_ram : altsyncram

-- GENERIC MAP (  
  clock_enable_input_a => "BYPASS",  
  clock_enable_output_a => "BYPASS",  
  init_file => "./matlab scripts/512bytecount_zero.mif",  
  intended_device_family => "Cyclone V",  
  lpm_hint => "ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=DEB",  
  lpm_type => "altsyncram",  
  numwords_a => 512,  
  operation_mode => "SINGLE_PORT",  
  outdata aclr_a => "NONE",  
  outdata reg_a => "CLOCK0",  
  power_up_uninitialized => "FALSE",  
  read_during_write_mode_port_a => "DON'T CARE",  
  widthad_a => 9,  
  width_a => 8,  
  width byt e a => 1  
);  

-- PORT MAP (  
  address_a => block byte addr top,  
  clock0 => clk internal,  
  data_a => block byte data top,  
  wren_a => block byte wren top  
);  

-- Internal Tri-State of all bidirectional lines.  
tri_state_cmd: process (cmd_write_en_signal)  
begin  
  if (cmd_write_en_signal = '1') then  
    sd_cmd <= cmd_top_signal;  
    cmd_top_signal_in <= '1';  
  else  
    sd_cmd <= 'Z';  
    cmd_top_signal_in <= sd_cmd ;  
  end if;  
end process;
tri_state_D0: process(D0_write_en_signal)
begin
if (D0_write_en_signal = '1') then
    sd_dat(0) <= dat0_top_signal;
    D0_top_signal_in <= '1';
else
    sd_dat(0) <= 'Z';
    D0_top_signal_in <= sd_dat(0);
end if;
end process;

tri_state_D1: process(D1_write_en_signal)
begin
if (D1_write_en_signal = '1') then
    sd_dat(1) <= dat1_top_signal;
    D1_top_signal_in <= '1';
else
    sd_dat(1) <= 'Z';
    D1_top_signal_in <= sd_dat(1);
end if;
end process;

tri_state_D2: process(D2_write_en_signal)
begin
if (D2_write_en_signal = '1') then
    sd_dat(2) <= dat2_top_signal;
    D2_top_signal_in <= '1';
else
    sd_dat(2) <= 'Z';
    D2_top_signal_in <= sd_dat(2);
end if;
end process;

tri_state_D3: process(D3_write_en_signal)
begin
if (D3_write_en_signal = '1') then
    sd_dat(3) <= dat3_top_signal;
    D3_top_signal_in <= '1';
else
    sd_dat(3) <= 'Z';
    D3_top_signal_in <= sd_dat(3);
num_blocks_to_write_signal <= to_integer(unsigned(data_nblocks)) when (to_integer(unsigned(data_nblocks)) < 128) else 128;

hs_sdr25_mode_en <= hs_sdr25_mode_g;

block_read_sd_addr_signal <= x"00000000";

data_buffer_to_sd_data_handler: process(rst_n, clk) begin
if (rst_n = '0') then
  stop_write <= '0';
  sd_write_done_internal <= '0';
  num_of_blocks_written <= 0;
  block_write_sd_addr_signal <= x"00000000";
  sd_control_signal <= x"FF";
  sd_write_done_internal <= '0';
  mem_clk_started_follower <= '0';
elsif rising_edge(clk) then
  if (buffer_reinit_done_internal = '1') then
    stop_write <= '0';
    sd_write_done_internal <= '0';
    num_of_blocks_written <= 0;
end if;

if (mem_clk_started_follower /= mem_clk_started) then
  mem_clk_started_follower <= mem_clk_started;
  --On the first rising edge of mem_clk after a buffer reinit
  --(or power on) sample the start address.
  if (mem_clk_started = '1') then
    block_write_sd_addr_signal <= data_sd_start_address;
  end if;
end if;

--If there is data in the buffer, begin
if (sd_write_rdy_top = '1') then
  if (stop_write = '0') then
    --if in idle state change address and data to write.
    if (sd_status_signal = x"01") then
      --Which mode do we select. Single/m
      --x"44" is the 4 bit multiblock path
      sd_control_signal <= x"44";
      --Number of single multiblock writes to do.
      num_of_blocks_written <= num_of_blocks_written +
          num_blocks_to_write_signal;
      --Only send the sd_control_signal once per APP_WAIT.
      stop_write <= '1';
    end if;
  end if;
else
  --We must wait past APP_WAIT of sd_data to change control signal.
  --We wait until CMD12_INIT of microsd_data FSM.
  --Works for both 1 bit and 4 bit writing. They both rely
  --on CMD12 which signifies end of the multiblock write
  if (sd_status_signal = x"48") then
    --In process of writing data_nblock block
    if (num_of_blocks_written = to_integer(unsigned(data_nblocks))) then
      --This system keeps track of how many blocks have been singly written.
      --It will halt writing when the counter reaches X number of blocks.
      stop_write <= '1';
      sd_write_done_internal <= '1';
    end if;
  end if;
end if;
block_write_sd_addr_signal <= std_logic_vector(
  unsigned(
    block_write_sd_addr_signal +
    num_blocks_to_write_signal )
);  
— Keep writing and update the address to write next
num_blocks_to_write.
stop_write <= '0';
end if;
sd_control_signal <= x"FF";
end if;
end if;
end if;
end if;
end process data_buffer_to_sd_data_handler;

-- Process to test one erase. TURN OFF WHEN NOTE IN USE
-- process(clk,rst_n)
-- begin
-- if (rst_n = '0') then
--  stop_write <= '0';
--  sd_control_signal <= x"FF";
-- end if;
-- elsif rising_edge(clk) then
--  if (stop_write = '0') then
--    if (sd_status_signal = x"01") then -- if in idle state change address and data to write
--      sd_control_signal <= x"0E";
--      erase_start_signal <= std_logic_vector(
--        to_unsigned(0, 32));
--      erase_end_signal <= std_logic_vector(
--        to_unsigned(900000, 32));
--      elsif (sd_status_signal = x"E0") then -- Inside erase path
--        sd_control_signal <= x"FF";
--        stop_write <= '1';
--      end if;
--    end if;
--  end if;
-- end process;
---data_clk rising edge detection mechanism for grabbing first rising edge of any nblock set.
---This process will fire first time and then every new nblock thereafter, latching address appropriately.

buffer_reinit_data_clk_reset: process(rst_n, clk)
begin
  if (rst_n = '0') then
    mem_clk_started <= '0';
    data_we_active_r <= '0';
  elsif rising_edge(clk) then
    if(buffer_reinit_done_internal = '1') then
      mem_clk_started <= '0';
      data_we_active_r <= '1';
    elsif (data_we_active = '1') then
      mem_clk_started <= '1';
      data_we_active_r <= '0';
    end if;
  end if;
end process buffer_reinit_data_clk_reset;

data_clk_sense: process(rst_n, clk)
begin
  if(rst_n = '0') then
    data_we_active_r_follower <= '0';
    data_we_active <= '0';
  elsif rising_edge(clk) then
    if (data_we = '1') then
      if (data_we_active_r_follower /= data_we_active_r) then
        data_we_active_r_follower <= data_we_active_r;
        if (data_we_active_r = '1') then
          data_we_active <= '0';
        end if;
      else
        data_we_active <= '1';
      end if;
    end if;
  end if;
end process data_clk_sense;

clk_en_process
-- The internal clk will gate after the sd_data state
-- machine returns to APP_WAIT.
clk_en_process: process(rst_n, clk)
begin
  if (rst_n = '0') then

796  -- shut_off <= '0';
797  elsif rising_edge(clk) then
798      -- if(clock_enable = '0') then
799          -- if (sd_status_signal = x"01") then
800              -- shut_off <= '1';
801          -- end if;
802      -- else
803          -- shut_off <= '0';
804  -- end if;
805  -- end if;
806  -- end process clk_en_process;
807
808
809
810 end Behavioral;

Filename: microsd_controller_inner.vhd

Description: Source code for microsd serial data logger

Author: Christopher Casebeer

Lab: Dr. Snider

Department: Electrical and Computer Engineering

Institution: Montana State University

Support: This work was supported under NSF award No. DBI-1254309

Creation Date: June 2014

Version 1.0

Modification History (give date, author, description)

None

Please send bug reports and enhancement requests to Dr. Snider at rksnider@ece.montana.edu

This software is released under

The MIT License (MIT)

Copyright (C) 2014 Christopher C. Casebeer and Ross K. Snider

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
microsd_controller_inner handles muxing and microsd_init depending on which is active.

@details

@param clk_divide_g Divide value to take clk to ~400kHz for sd card initialization.

@param clk_freq_g Frequency of the input clk.

@param clk Transmission Clock

@param rst_n Start signal from input pushbutton

@param sd_init_start Reset to initial conditions.

@param sd_control Used to select pathway in microsd_data.

@param sd_status Current State of State Machine.

@param block_byte_data Read data from SD card memory

@param block_byte_wren Signals that a data byte has been read.

@param block_read_sd_addr Address to read block from on sd card.

@param block_byte_addr Address to write read data to in ram.

@param block_write_sd_addr Address where block is written on sd card.

@param block_write_data Byte that will be written

@param num_blocks_to_write Number of blocks to be written in CMD25.

@param ram_read_address Where block_write_data is read from.

@param erase_start Start address of erase.
104  ---!
105  ---! @param erase_end Stop address of erase.
106  ---! @param state_leds Used to encode current
107  state to Leds.
108  ---! @param prev_block_write_sd_addr The last
109  address to be successfully written valid on
110  ---!
111  prev_block_write_sd_addr_pulse '1'
112  ---!
113  ---! @param prev_block_write_sd_addr_pulse
114  prev_block_write_sd_addr is valid
115  ---!
116  ---! @param cmd_write_en Tri State Enable
117  ---! @param D0_write_en Tri State Enable
118  ---! @param D1_write_en Tri State Enable
119  ---! @param D2_write_en Tri State Enable
120  ---! @param D3_write_en Tri State Enable
121  ---!
122  ---! @param cmd_signal_in Read value of the tri-
123  stated line.
124  ---! @param D0_signal_in Read value of the tri-
125  stated line.
126  ---! @param D1_signal_in Read value of the tri-
127  stated line.
128  ---! @param D2_signal_in Read value of the tri-
129  stated line.
130  ---! @param D3_signal_in Read value of the tri-
131  stated line.
132  ---!
133  ---! @param init_done Card has passed init phase.
134  ---! @param signalling_18_en Card should go into 1.8V
135  mode during init.
136  ---! @param hs_sdr25_mode_en Card should transition to
137  hs_sdr25 mode before first CMD25.
138  ---!
139  ---! @param dat0 dat0 line out
140  ---! @param dat1 dat1 line out
141  ---! @param dat2 dat2 line out
142  ---! @param dat3 dat3 line out
143  ---! @param cmd cmd line out
144  ---! @param sclk clk sent to sd card
145  ---!
entity microsd_controller_inner is
  generic(
    clk_divide_g : natural;
    clk_freq_g  : natural;
    signalling_18_en_g : std_logic
  );
  port(
    clk         :in std_logic;
    rst_n       :in std_logic;
    sd_init_start :in std_logic;
    sd_control  :in std_logic_vector(7 downto 0)
    );
  sd_status   :out std_logic_vector(7 downto 0)
  );
  block_byte_data  :out std_logic_vector(7 downto 0)
  );
  block_byte_wren  :out std_logic;
  block_read_sd_addr :in std_logic_vector(31 downto 0);
  block_byte_addr  :out std_logic_vector(8 downto 0)
  );
  block_write_sd_addr :in std_logic_vector(31 downto 0);
  block_write_data  :in std_logic_vector(7 downto 0);
  num_blocks_to_write  :in integer range 0 to 2**16 - 1;
  ram_read_address  :out std_logic_vector(8 downto 0);

erase_start : in std_logic_vector(31 downto 0);
erase_end : in std_logic_vector(31 downto 0);
state_leds : out std_logic_vector(3 downto 0);
prev_block_write_sd_addr : out std_logic_vector(31 downto 0);
prev_block_write_sd_addr_pulse : out std_logic;
cmd_write_en : out std_logic;
D0_write_en : out std_logic;
D1_write_en : out std_logic;
D2_write_en : out std_logic;
D3_write_en : out std_logic;
cmd_signal_in : in std_logic;
D0_signal_in : in std_logic;
D1_signal_in : in std_logic;
D2_signal_in : in std_logic;
D3_signal_in : in std_logic;
init_done : out std_logic;
hs_sdr25_mode_en : in std_logic;
vc_18_on : out std_logic;
vc_33_on : out std_logic;

---SD Signals

dat0 : out std_logic;
dat1 : out std_logic;
dat2 : out std_logic;
dat3 : out std_logic;
cmd : out std_logic;
sclk : out std_logic;
restart : out std_logic;

ext_trigger : out std_logic
architecture Behavioral of microsd_controller_inner is

component microsd_init
  generic(
    signalling_18_en_g : std_logic);
  port(
    clk : in std_logic;
    rst_n : in std_logic;
    sd_init_start_in : in std_logic;
    cmd_out : out std_logic;
    dat0_out : out std_logic;
    dat3_out : out std_logic;
    sclk_out : out std_logic;
    D0_signal_in : in std_logic;
    init_done_out : out std_logic;
    cmd_write_en_out : out std_logic;
    D3_write_en_out : out std_logic;
    cmd_signal_in : in std_logic;
    rca_out : out std_logic_vector (15 downto 0);
    restart_out : out std_logic;
    vc_18_on_out : out std_logic;
    vc_33_on_out : out std_logic;
    state_leds_out : out std_logic_vector (3 downto 0);
    ext_trigger_out : out std_logic
  );
end component;

component microsd_data is
  generic(
    clk_freq_g : natural);
  port(
  );
end component;
clk : in std_logic;

rst_n : in std_logic;

sd_control : in std_logic_vector (7 downto 0);

sd_status : out std_logic_vector (7 downto 0);

block_byte_data : out std_logic_vector (7 downto 0);

block_byte_wren : out std_logic;

block_read_sd_addr : in std_logic_vector (31 downto 0);

block_byte_addr : out std_logic_vector (8 downto 0);

block_write_sd_addr : in std_logic_vector (31 downto 0);

block_write_data : in std_logic_vector (7 downto 0);

num_blocks_to_write : in integer range 0 to 2**16 - 1;

ram_read_address : out std_logic_vector (8 downto 0);

erase_start : in std_logic_vector (31 downto 0);

erase_end : in std_logic_vector (31 downto 0);

state_leds : out std_logic_vector (3 downto 0);

prev_block_write_sd_addr : out std_logic_vector (31 downto 0);

prev_block_write_sd_addr_pulse : out std_logic;

cmd_write_en : out std_logic;

D0_write_en : out std_logic;
D1_write_en : out std_logic;
D2_write_en : out std_logic;
D3_write_en : out std_logic;
cmd_signal_in : in std_logic;
D0_signal_in : in std_logic;
D1_signal_in : in std_logic;
D2_signal_in : in std_logic;
D3_signal_in : in std_logic;
card_rca : in std_logic_vector (15 downto 0);
init_done : in std_logic;
h_sdr25_mode_en : in std_logic;

--SD Signals
dat0 : out std_logic;
dat1 : out std_logic;
dat2 : out std_logic;
dat3 : out std_logic;
cmd : out std_logic;
sclk : out std_logic;
restart : out std_logic;
data_send_crc_error : out std_logic;
ext_trigger : out std_logic;

);
end component;

--Signal Declarations

-- Input Signals
signal sd_init_start_signal : std_logic;

-- Clock Signals
signal clk_400k_signal : std_logic;
signal clk_400k_count : integer range 0 to 2**9 - 1;

-- SD_INIT Signals
signal initDat0_signal : std_logic;
signal initCmd_signal : std_logic;
signal initSclk_signal : std_logic;
signal initDat3_signal : std_logic;
signal initDone_signal : std_logic;
signal initState_leds_signal : std_logic_vector (3 downto 0);
signal cmdWrite_en_init : std_logic;
signal D3_write_en_init : std_logic;
signal initRestart : std_logic;
signal ext_trigger_init : std_logic;

-- SD_DATA Signals
signal dataCmd_signal : std_logic;
signal dataSclk_signal : std_logic;
signal dataDat3_signal : std_logic;
signal dataState_leds_signal : std_logic_vector (3 downto 0);
signal dataDat0_signal : std_logic;
signal cmdWrite_en_data : std_logic;
signal D3_write_en_data : std_logic;
signal card_rca_signal : std_logic_vector (15 downto 0);
signal dataRestart : std_logic;
signal ext_trigger_data : std_logic;
signal div_clk : std_logic;
signal div_clk_count : unsigned (7 downto 0);

begin
sd_init_start_signal <= sd_init_start;
init_done <= init_done_signal;
i_sd_init_0: microsd_init
generic map(
signalling_18_en_g => signalling_18_en_g
)
port map(
clk => clk_400k_signal,
rst_n => rst_n,
sd_init_start_in => sd_init_start_signal,
dat0_out => init_dat0_signal,
cmd_out => init_cmd_signal,
sclk_out => init_sclk_signal,
dat3_out => init_dat3_signal,
D0_signal_in => D0_signal_in,
init_done_out => init_done_signal,
cmd_write_en_out => cmd_write_en_init,
D3_write_en_out => D3_write_en_init,
cmd_signal_in => cmd_signal_in,
state_leds_out => init_state_leds_signal,
ext_trigger_out => ext_trigger_init,
vclk18_on_out => vc_18_on,
vclk33_on_out => vc_33_on,
restart_out => init_restart,
rcu_out => card_rca_signal
);

i_sd_data_0: microsd_data
generic map(
clk_freq_g => clk_freq_g
)
port map(
clk => clk,
--clk => div_clk,
rst_n => rst_n,
dat0 => data_dat0_signal,
cmd => data_cmd_signal,
sclk => data_sclk_signal,
dat3 => data_dat3_signal,
dat1 => dat1,
dat2 => dat2,

cmd_write_en => cmd_write_en_data,
cmd_signal_in => cmd_signal_in,
D0_write_en => D0_write_en,
D1_write_en => D1_write_en,
D2_write_en => D2_write_en,
D3_write_en => D3_write_en_data,
D0_signal_in => D0_signal_in,
D1_signal_in => D1_signal_in,
D2_signal_in => D2_signal_in,
D3_signal_in => D3_signal_in,
card_rca => card_rca_signal,
init_done => init_done_signal,
block_read_sd_addr => block_read_sd_addr,
block_byte_data  => block_byte_data,
block_byte_wren  => block_byte_wren,
block_byte_addr  => block_byte_addr,

prev_block_write_sd_addr  => prev_block_write_sd_addr,
prev_block_write_sd_addr_pulse =>
prev_block_write_sd_addr_pulse ,

hs_sdr25_mode_en  => hs_sdr25_mode_en,
block_write_sd_addr  => block_write_sd_addr,
block_write_data  => block_write_data,

sd_control  => sd_control,
sd_status  => sd_status,
state_leds  => data_state_leds_signal,
num_blocks_to_write  => num_blocks_to_write,
erase_start  => erase_start,
ram_read_address  => ram_read_address,
restart  => data_restart,

ext_trigger  => ext_trigger_data,
erase_end  => erase_end
);

with init_done_signal select
  sclk  <= data_sclk_signal when '1',
         init_sclk_signal when others;
with init_done_signal select
  cmd  <= data_cmd_signal when '1',
         init_cmd_signal when others;
with init_done_signal select
  dat0  <= data_dat0_signal when '1',
         init_dat0_signal when others;
with init_done_signal select
  dat3  <= data_dat3_signal when '1',
         init_dat3_signal when others;
with init_done_signal select  
  state_leds <= data_state_leds_signal when '1',  
  init_state_leds_signal when others;

with init_done_signal select  
  cmd_write_en <= cmd_write_en_data when '1',  
  cmd_write_en_init when others;

with init_done_signal select  
  D3_write_en <= D3_write_en_data when '1',  
  D3_write_en_init when others;

with init_done_signal select  
  restart <= data_restart when '1',  
  init_restart when others;

with init_done_signal select  
  ext_trigger <= ext_trigger_data when '1',  
  ext_trigger_init when others;

/*Generate the 400k clock using the clk_divide_g generic.*/
gen_400k : process(rst_n,clk)
begin  
  if (rst_n = '0') then
    clk_400k_signal <= '0';
    clk_400k_count <= 0;
  elsif rising_edge(clk) then
    --Divides by count = (count+1)*2.  
    --50 Mhz --> (clk_divide_g(128)/2 - 1) + 1 * 2,  
    --128/2 -1 == 63. + 1 == 64. * 2 == 128. 50Mhz/128 = .3906  
    MHZ.
    if(clk_400k_count = clk_divide_g/2 - 1) then
      clk_400k_signal <= not clk_400k_signal;
      clk_400k_count <= 0;
    elseif(init_done_signal = '1') then
      clk_400k_signal <= '0';
      clk_400k_count <= 0;
    else
      clk_400k_count <= clk_400k_count + 1;
  end if;
end process;
end if;
end if;
end process gen_400k;
end Behavioral;
Filename: microsd_init.vhd
Description: Source code for microsd serial data logger
Author: Christopher Casebeer
Lab: Dr. Snider
Department: Electrical and Computer Engineering
Institution: Montana State University
Support: This work was supported under NSF award No. DBI-1254309
Creation Date: June 2014

Version 1.0

Modification History (give date, author, description)
None

Please send bug reports and enhancement requests to Dr. Snider at rksnider@ece.montana.edu

This software is released under The MIT License (MIT)
Copyright (C) 2014 Christopher C. Casebeer and Ross K. Snider

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is
—furnished to do so, subject to the following conditions:

—

—The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

—

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Christopher Casebeer
Electrical and Computer Engineering
Montana State University
541 Cobleigh Hall
Bozeman, MT 59717
christopher.casebeer1@msu.montana.edu

Ross K. Snider
Associate Professor
Electrical and Computer Engineering
Montana State University
538 Cobleigh Hall
Bozeman, MT 59717
rksnider@ece.montana.edu

Information on the MIT license can be found at http://opensource.org/licenses/MIT

library IEEE ;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.Numeric_STD.ALL ;
use IEEE.MATH_REAL.ALL ;
---! @brief sd_init is the portion of the microsd_controller which guides the microSD card through initialization

---! @details

---! @param signalling_18_en_g Attempt to switch into 1.8 V signalling

---! @param cmd_out cmd line output.

---! @param dat0_out dat0 line output.

---! @param dat3_out dat3 line output.

---! @param sclk_out Sclk sent to card.

---! @param D0_signal_in dat0 tri-state line read.

---! @param init_done_out SD Initialization Complete During Init

---! @param signalling_18_en_in 1.8V Switch Enable

---! @param cmd_write_en_out Enable Write on CMD line

---! @param D3_write_en_out dat3 write enable

---! @param cmd_signal_in cmd tri-state line read

---! @param rca_out Relative Card Address passed to sd_data.

---! @param restart_out Command response failures exceeded
---!
@param vc_18_on_out Enable 1.8V signal to level translator
---!
@param vc_33_on_out Enable 3.3V signal to level translator
---!
@param state_leds_out State indication for leds
---!
@param ext_trigger_out External trigger bit.
---!
if need arise.

entity microsd_init is
  generic(
    signalling_18_en_g : std_logic
  );
  port(
    clk : in std_logic;
    rst_n : in std_logic;
    sd_init_start_in : in std_logic;
    cmd_out : out std_logic;
    dat0_out : out std_logic;
    dat3_out : out std_logic;
    sclk_out : out std_logic;
    D0_signal_in : in std_logic;
    init_done_out : out std_logic;
    cmd_write_en_out : out std_logic;
    D3_write_en_out : out std_logic;
    cmd_signal_in : in std_logic;
    rca_out : out std_logic_vector(15 downto 0);
    restart_out : out std_logic;
    vc_18_on_out : out std_logic;
    vc_33_on_out : out std_logic;
    state_leds_out : out std_logic_vector(3 downto 0);
    ext_trigger_out : out std_logic
  );
end microsd_init;
microsd_init is responsible for handling sd card initialization which happens only once per card power on. This includes switching the card into 1.8V mode as well as handing off RCA address for all future commands.

architecture Behavioral of microsd_init is

component microsd_crc_7 is
    port(
        bitval :in std_logic;
        enable :in std_logic;
        clk :in std_logic;
        rst :in std_logic;
        crc_out : out std_logic_vector(6 downto 0)
    );
end component;

--FSM Signals
type sd_ctrl_state is (START_WAIT, PWR_ON, CMD0_INIT, CMD0_SEND, CMD0_READ, CMD8_INIT, CMD8_SEND, CMD8_READ, CMD55_INIT, CMD55_SEND, CMD55_READ, ACMD41_INIT, ACMD41_SEND, ACMD41_READ, CMD2_INIT, CMD2_SEND, CMD2_READ, CMD3_INIT, CMD3_SEND, CMD3_READ, CMD11_INIT, CMD11_SEND, CMD11_READ, CMD11_READ_PAUSE, VOLTAGE_SWITCH, VOLTAGE_SWITCH_WAIT, ERROR, IDLE);

--State signals
signal next_state :sd_ctrl_state; -- FSM next-state variable
signal current_state :sd_ctrl_state; -- FSM current-state variable

-- PWR_ON_DELAY0 Process Signals
signal pwr_on_delay_en :std_logic;
signal pwr_on_delay_count :integer range 0 to 2**8 - 1;
```
189 signal pwr_on_delay_done : std_logic;
190
191 -- CMD_SEND0 Process Signals
192 signal command_load_en : std_logic;
193 signal output_command : std_logic_vector(47 downto 0);
194 signal command_send_en : std_logic;
195 signal command_send_done : std_logic;
196 signal command_send_bit_count : integer range 0 to 2**6 - 1;
197 signal cmdstartbit : std_logic;
198
199 -- 48-bit SD Card command register
200 signal command_signal : std_logic_vector(47 downto 0);
201
202 -- CMD_READ_R1_RESPONSE0 Signals
203 signal read_bytes : std_logic_vector(47 downto 0);
204 signal read_r1_response_done : std_logic;
205 signal r1_response_bytes : std_logic_vector(47 downto 0);
206 signal read_r1_response_en : std_logic;
207 signal response_1_status : std_logic_vector(31 downto 0);
208 signal response_1_current_state_bits : std_logic_vector(3 downto 0);
209
210 -- CMD_READ_R2_RESPONSE0 Signals
211 signal r2_response_bytes : std_logic_vector(135 downto 0);
212 signal read_r2_response_done : std_logic;
213 signal read_r2_bytes : std_logic_vector(135 downto 0);
214 signal read_r2_response_en : std_logic;
215
216 -- CMD_READ_R3_RESPONSE0 Signals
217 signal r3_response_bytes : std_logic_vector(47 downto 0);
218 signal read_r3_response_done : std_logic;
219 signal read_r3_bytes : std_logic_vector(47 downto 0);
220 signal read_r3_response_en : std_logic;
221
222 -- CMD_READ_R6_RESPONSE0 Signals
```
signal r6_response_bytes : std_logic_vector(47 downto 0);
signal read_r6_response_done : std_logic;
signal read_r6_bytes : std_logic_vector(47 downto 0);
signal read_r6_response_en : std_logic;
signal response_6_status : std_logic_vector(15 downto 0);

--- CMD_READ_R7_RESPONSE0 Signals
signal r7_response_bytes : std_logic_vector(47 downto 0);
signal read_r7_response_done : std_logic;
signal read_r7_bytes : std_logic_vector(47 downto 0);
signal read_r7_response_en : std_logic;

--- SD Card I/O signals
signal dat0_signal : std_logic;
signal dat3_signal : std_logic;
signal cmd_signal : std_logic;
signal cmd_signal_in_signal : std_logic;

--- Command Payload Signals
--- This bit indicates whether or not the SD card is SDSC(0),
--- or SDHC/SDXC(1). Might better be a generic. Was always set
to '1'
--- during development as I was using a SDXC card.
signal sd_hcs_bit : std_logic := '1';

--- Default RCA addressed used with ACMD41 on init
signal card_rca_signal : std_logic_vector (15 downto 0);

--- Voltage Switch Signals
--- Once off signal to track number of
--- d0–d3 transitions during voltage translate.
signal d0_signal_in_follower : std_logic;
signal d0_signal_in_count : unsigned (1 downto 0);
signal voltage_switch_done : std_logic;
signal voltage_switch_en_signal : std_logic;
---CRC7 Signals
signal crc7_bitval_signal : std_logic;
signal crc7_rst_signal : std_logic;
signal crc7_signal : std_logic_vector (6 downto 0);
signal crc7_gen_en : std_logic;
signal crc7_done : std_logic;
signal crc7_send_bit_count : integer range 0 to 2**6 - 1;

---Debug Signals
signal init_counter : unsigned (31 downto 0);

---Sclk Enable
signal sclk_en : std_logic;

---Voltage Switch Signals
---Wait time between 3.3V off and 1.8V on
---Wait time on 1.8V turn on to sclk turned back on (stabilize 1.8V)
signal voltage_switch_lag_counter : unsigned (11 downto 0);
signal voltage_switch_18_stabilize : unsigned (11 downto 0);
---The voltage switch has finished.
signal voltage_switch_run : std_logic;

---Wait after cmd_11_read to turn the clk off.
signal cmd_11_read_done_pause_counter : unsigned (11 downto 0);
signal cmd_11_read_done_pause_en : std_logic;
signal cmd_11_read_done_pause_done : std_logic;

attribute noprune : boolean;
attribute noprune of response_1_status : signal is true;
attribute noprune of response_1_current_state_bits : signal is true;
attribute noprune of crc7_done : signal is true;
attribute noprune of init_counter : signal is true;
begin
process (clk, rst_n)
begin
if (rst_n = '0') then
    current_state <= START_WAIT;
elsif (rising_edge(clk)) then
    current_state <= next_state;
end if;
end process;

---NEXT-STATE LOGIC PROCESS

process (current_state, sd_init_start_in, pwr_on_delay_done, command_send_done, read_r1_response_done, read_r7_response_done, read_r3_response_done, read_r2_response_done, read_r6_response_done,
begin
next_state <= current_state;  — Default to current state

case current_state is

when START_WAIT =>
  — Wait for active-low pushbutton press to start initialization process
  — or a low pulse of sd_init_start_in.
  if (sd_init_start_in = '0') then
    next_state <= PWR_ON;
  end if;
  — Send > 74 clock cycles with all lines pulled up.
  — Hold over from SPI mode.
when PWR_ON =>
  if (pwr_on_delay_done = '1') then
    next_state <= CMD0_INIT;
  end if;
  — CMD0. Reset the SD Card. GO_IDLE_STATE

when CMD0_INIT =>
  next_state <= CMD0_SEND;

when CMD0_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD0_READ;
  end if;

when CMD0_READ =>
  next_state <= CMD8_INIT;

  — CMD8
  — Sends SD Memory Card interface
  — condition, which includes host supply
  — voltage information and asks the card
  — whether card supports voltage.
  — Reserved bits shall be set to '0'.
  — Send CMD8 (SEND_IF_COND) to the SD card.
  — Expand the SD card instruction set.
when CMD8_INIT =>
next_state <= CMD8_SEND;

when CMD8_SEND =>
if (command_send_done = '1') then
  next_state <= CMD8_READ;
end if;

when CMD8_READ =>
if (read_r7_response_done = '1') then
  — This is checking R7 response for echo back pattern.
  — Card will not respond if voltage level is not accepted.
  if (r7_response_bytes(15 downto 8) = "10101010") then
    next_state <= CMD55_INIT;
  else
    next_state <= ERROR;
  end if;
  — elsif (cmd_resend_en = '1') then
  — next_state <= CMD8_INIT;
  end if;

  — Send CMD55 (APP_CMD) to the SD card to preface impending application-specific command.
when CMD55_INIT =>
  next_state <= CMD55_SEND;

when CMD55_SEND =>
if (command_send_done = '1') then
  next_state <= CMD55_READ;
end if;

when CMD55_READ =>
if (read_r1_response_done = '1') then
  — This bit is the APP_CMD bit of the status register.
  — The other bit set on first CMD55 response is READY FOR DATA.
  if (r1_response_bytes(13) = '1') then
    next_state <= ACMD41_INIT;
  else
    next_state <= ERROR;
  end if;
  — elsif (cmd_resend_en = '1') then
  — next_state <= CMD55_INIT;
  end if;
  — Send ACMD41 (SD_SEND_OP_COND) Here we send
—host capacity support info and to begin card
initialization process.
See output logic for contents.
—ACMD41 is sent repeatedly with the same setup as the
first ACMD41 sent.
The card becomes initialized and ready to continue when
the R3 response
—which contains the OCR register has bit 31 set to a ’1’,
the Card power
—up status bit. Only then do we continue.
—ACMD41 currently doesn’t have any command resend error
handling.

when ACMD41_INIT =>

next_state <= ACMD41_SEND;

when ACMD41_SEND =>

if (command_send_done = '1') then

next_state <= ACMD41_READ;
end if;

when ACMD41_READ =>

if (read_r3_response_done = '1') then

—OCR (31) of R3 gets set to 1 when the init phase
is over.

—We check for that bit in the R3 response here.

if (r3_response_bytes(39) = '1') then

—If switching voltages is a go, proceed to CMD11.

—Signified by bit 24 of OCR, Switching to 1.8V
Accepted.

—**NOTE** bit 24 of OCR will come back ’0’ if the
card
—is already in 1.8V mode from reinitializing w/o
—doing a power cycle.

if (r3_response_bytes(32) = '1') then

next_state <= CMD11_INIT;

else

next_state <= CMD2_INIT;
end if;

—Otherwise keeps sending CMD55/ACMD41. Do not change
ACMD41.

else

—Go back and send another CMD55 followed by
ACMD41

next_state <= CMD55_INIT;
469        end if;
470
471        -- elsif(cmd_resend_en = '1') then
472        -- next_state <= CMD55_INIT;
473        end if;
474
475
476        -- Voltage switch command. Depends on the S18A bit in the
477        -- ACMD41 response.
478        -- If 1, Voltage switch is possible .... send CMD 11.
479        when CMD11_INIT =>
480            next_state <= CMD11_SEND;
481
482        when CMD11_SEND =>
483            if (command_send_done = '1') then
484                next_state <= CMD11_READ;
485            end if;
486            -- Returning R1 type response means the card starts voltage
487            -- switch sequence.
488        when CMD11_READ =>
489            if (read_r1_response_done = '1') then
490                next_state <= CMD11_READ_PAUSE;
491                -- elsif(cmd_resend_en = '1') then
492                -- next_state <= CMD11_INIT;
493            end if;
494            -- Returning R1 type response
495            -- means the card starts voltage switch sequence.
496        when CMD11_READ_PAUSE =>
497            if (cmd_11_read_done_pause_done = '1') then
498                next_state <= VOLTAGE_SWITCH;
499                -- elsif(cmd_resend_en = '1') then
500                -- next_state <= CMD11_INIT;
501            end if;
502
503        when VOLTAGE_SWITCH =>
504            -- Voltage switch sequence is done when d0-d3 goes high.
505            -- d0 going high detection is handled in other process.
506            -- Card waits for level translation to finish as the card
507            -- itself
508            -- detects 1.8V signalling on the lines.
509            -- Turn the clk back on (output logic) after the 1.8V
510            -- comes on.
511            if (voltage_switch_run = '1') then
next_state <= VOLTAGE_SWITCH_WAIT;
end if;
when VOLTAGE_SWITCH_WAIT =>
  --Voltage switch sequence is done when d0–d3 goes high.
  --d0 going high detection is handled in other process.
  --Card waits for level translation to finish as the card itself
  --detects 1.8V signalling on the lines.
  --Here I turn the selk back on and wait.
  --This way the clk is turned off momentarily.
if (voltage_switch_done = '1') then
  next_state <= CMD2_INIT;
end if;

--Get CID (unique card identification from card).
when CMD2_INIT =>
  next_state <= CMD2_SEND;
when CMD2_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD2_READ;
  end if;

--Nothing is done with the CID at this time.
when CMD2_READ =>
  if (read_r2_response_done = '1') then
    next_state <= CMD3_INIT;
    -- elsif (cmd_resend_en = '1') then
    --   next_state <= CMD2_INIT;
  end if;

--Get address of the card (RCA).
--All commands after this need this RCA number.
when CMD3_INIT =>
  next_state <= CMD3_SEND;
when CMD3_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD3_READ;
  end if;
when CMD3_READ =>
  --RCA is stored on r6 done.
  --RCA is stored in the r6 read response process.
  if (read_r6_response_done = '1') then
Card is in standby mode Binary Coded Decimal == 3 means standby.

Initial CMD3 yields BCD 2 == INIT.

I am waiting for card to be in standby here.

Not needed most likely as card state in response is at the time of cmd issue.

So the card is most likely in standby after the first CMD3. Two CMD3’s are sent anyway.

if (response_6_status(12 downto 9) = "0011") then
  next_state <= IDLE;
else
  next_state <= CMD3_INIT;
end if;

elsif (cmd_resend_en = '1') then
  next_state <= CMD3_INIT;
end if;

when ERROR =>
  next_state <= ERROR;

when IDLE =>
  next_state <= IDLE;

end case;

begin
  -- Default Signal Values
  pwr_on_delay_en <= '0';
  command_load_en <= '0';
  command_send_en <= '0';
read_r1_response_en <= '0';
read_r2_response_en <= '0';
read_r3_response_en <= '0';
read_r6_response_en <= '0';
read_r7_response_en <= '0';
command_signal <= x"FFFFFFFFFFFFF";
init_done_out <= '0';
crc7_gen_en <= '0';
rc_a_out <= x"0000";
voltage_switch_en_signal <= '0';
D3_write_en_out <= '0';
ext_trigger_out <= '0';
clk_en <= '1';

cmd_11_read_done_pause_en <= '0';
case current_state is
when START_WAIT =>
  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  state_leds_out <= "1010";
  state_leds_out <= "1110";
  cmd_write_en_out <= '0';
when PWR_ON =>
  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  pwr_on_delay_en <= '1';
  state_leds_out <= "0000";
  cmd_write_en_out <= '1';
when CMD0_INIT => -- Set up the FSM to transmit CMD0 to the SD card

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

  -- Initialize the command data with CMD0 contents
  command_signal <= '0' & '1' & "000000" & x"00000000" & "1001010" & '1';
  command_load_en <= '1';
  state_leds_out <= "0001";
  cmd_write_en_out <= '0';

when CMD0_SEND =>

  cmd_signal <= output_command(47);

  -- Initialize the command data with CMD0 contents
  command_signal <= '0' & '1' & "000000" & x"00000000" & "1001010" & '1';
  dat0_signal <= '1';
  dat3_signal <= '1';
  command_send_en <= '1';
  state_leds_out <= "0010";
  cmd_write_en_out <= '1';

  -- Get into SD mode by holding dat3/CS_N '1' during sending CMD0.
  -- Do not rely on pullups to get to the card. Actively drive
  -- this line in case a level translator is inserted.
  D3_write_en_out <= '1';

when CMD0_READ =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

  -- read_r1_response_en <= '1';
  state_leds_out <= "0011";
when CMD8_INIT =>
  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  command_signal <= '0' & '1' & "001000" & x"00000"
  & "0001" & "10101010" & "1000011" & '1';
  command_load_en <= '1';
  state_leds_out <= "0100";
  cmd_write_en_out <= '0';

when CMD8_SEND =>
  cmd_signal <= output_command(47);
  command_signal <= '0' & '1' & "001000" & x"00000"
  & "0001" & "10101010" & "1000011" & '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  command_send_en <= '1';
  state_leds_out <= "0101";
  cmd_write_en_out <= '1';
when CMD8_READ =>
  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  read_r7_response_en <= '1';
  state_leds_out <= "0110";
  cmd_write_en_out <= '0';

---

--CMD55(APP_CMD) This command is sent before any ACMD.
--This simply tells the card that an expanded command is coming next.

---

when CMD55_INIT =>
  cmd_signal <= '1';
  dat3_signal <= '1';
  command_signal <= '0' & '1'
    & "110111" & x"00000000" & "0110010" & '1';
  command_load_en <= '1';
  state_leds_out <= "0111";
  cmd_write_en_out <= '0';

when CMD55_SEND =>
  cmd_signal <= output_command(47);
  command_signal <= '0' & '1' & "110111" & x"00000000" & "0110010" & '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  command_send_en <= '1';
  state_leds_out <= "1000";
  cmd_write_en_out <= '1';
when CMD55_READ =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

  read_r1_response_en <= '1';
  state_leds_out <= "1001";
  cmd_write_en_out <= '0';

—————

—ACMD41(SD_SEND_OP_COND) Main init command. 
—Power Savings, Signalling Level and Card Type Specified. 
—Hosts repeatedly issues this command until card is ready 
—(response bit indicates ready. OCR(31))
—————

when ACMD41_INIT =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

  if(signalling_18_en_g = '1') then
  —More bits of ACMD41 can be set. Page 23 of SD Spec. 
  —Card kept to SDXC and Power saving. 
  —1.8V or 3.3V signalling is decided here. 
  —FF80 references 23 downto 8 of OCR register. 
  command_signal <= '0' & '1' & "101001" & '0' 
  & sd_hcs_bit & "000001"
  & x"FF80" & x"00" & "0001000" & '1';
  else
  command_signal <= '0' & '1' & "101001" & '0' 
  & sd_hcs_bit & "000000"
  & x"FF80" & x"00" & "0001011" & '1';
  end if;

  command_load_en <= '1';
  state_leds_out <= "1010";
when ACMD41_SEND =>

cmd_signal <= output_command(47);

if(signalling_18_en_g = '1') then

    More bits of ACMD41 can be set. Page 23 of SD Spec.
    Card kept to SDXC and Power saving.
    1.8V or 3.3V signalling is decided here.
    FF80 references 23 downto 8 of OCR register.
    command_signal <= '0' & '1' & "101001" & '0'
    & sd_hcs_bit & "000001"
    & x"FF80" & x"00" & "0001000" & '1';
else

    command_signal <= '0' & '1' & "101001" & '0'
    & sd_hcs_bit & "000000"
    & x"FF80" & x"00" & "0001011" & '1';

end if;

dat0_signal <= '1';
dat3_signal <= '1';

command_send_en <= '1';
state_leds_out <= "1011";

when ACMD41_READ =>

cmd_signal <= '1';
dat3_signal <= '1';
dat0_signal <= '1';

read_r3_response_en <= '1';
state_leds_out <= "1100";

end if;

--------
CMD11(VOLTAGE SWITCH COMMAND/RESPONSE)

when CMD11_INIT =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

  command_signal <= '0' & '1'
                 & "001011" & x"00000000" & "0111011" & '1';

  command_load_en <= '1';
  state_leds_out <= "1101";

  cmd_write_en_out <= '0';

when CMD11_SEND =>

  cmd_signal <= output_command(47);

  command_signal <= '0' & '1'
                 & "001011" & x"00000000" & "0111011" & '1';

  dat0_signal <= '1';
  dat3_signal <= '1';

  command_send_en <= '1';
  state_leds_out <= "1110";

  cmd_write_en_out <= '1';

when CMD11_READ =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

  read_r1_response_en <= '1';
  state_leds_out <= "0001";
when CMD11_READ_PAUSE =>

  cmd_signal <= '1';
dat3_signal <= '1';
dat0_signal <= '1';
state_leds_out <= "0001";
cmd_write_en_out <= '0';
cmd_11_read_done_pause_en <= '1';

-- Custom state. This is where the level shifter is switched to new voltage level.
-- The SD card halts with d0 low until the levels are changed.

when VOLTAGE_SWITCH =>

  cmd_signal <= '1';
dat3_signal <= '1';
dat0_signal <= '1';
voltage_switch_en_signal <= '1';
state_leds_out <= "0001";
sclk_en <= '0';
ext_trigger_out <= '0';
cmd_write_en_out <= '0';

when VOLTAGE_SWITCH_WAIT =>

  cmd_signal <= '1';
dat3_signal <= '1';
dat0_signal <= '1';
voltage_switch_en_signal <= '1';
when CMD2_INIT =>
  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  command_signal <= '0' & '1' & "000010" & x"00000000" & "0100110" & '1';
  command_load_en <= '1';
  state_leds_out <= "1101";
  cmd_write_en_out <= '0';

when CMD2_SEND =>
  cmd_signal <= output_command(47);
  command_signal <= '0' & '1' & "000010" & x"00000000" & "0100110" & '1';
  dat0_signal <= '1';
  dat3_signal <= '1';
  command_send_en <= '1';
  state_leds_out <= "1110";
  cmd_write_en_out <= '1';

when CMD2_READ =>
  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  read_r2_response_en <= '1';
  state_leds_out <= "0001";
217

    cmd_write_en_out <= '0';

------------

--CMD3(GET RCA Relative Card Address)

------------

------------CMD3 w/ CRC gen

when CMD3_INIT =>

    cmd_signal <= '1';
    dat3_signal <= '1';
    dat0_signal <= '1';
    --IMPORTANT: The CRC7 is defaulted to x"01". It will be filled
    --by CRC7GEN and CMD_SEND.
    --The 1 is the end transmission bit.
    command_signal <= '0' & '1' & "000011" & x"0000000001";
    command_load_en <= '1';
    state_leds_out <= "0010";

    cmd_write_en_out <= '0';

when CMD3_SEND =>

    cmd_signal <= output_command(47);
    --IMPORTANT: The CRC7 is defaulted to x"01". It will be filled
    --by CRC7GEN and CMD_SEND.
    --The 1 is the end transmission bit.
    command_signal <= '0' & '1' & "000011" & x"0000000001";
    dat0_signal <= '1';
    dat3_signal <= '1';
    command_send_en <= '1';
    state_leds_out <= "0011";
    cmd_write_en_out <= '1';
    crc7_gen_en <= '1';
    --This command is sent with a CRC7 generated for it.
    --CRC7 is not hardcoded.
    --crc7_gen_en enabled while sending command
if you want CRC7 appended during send. Not absolutely needed here, but
always needed after we received variable RCA and start including
— it inside every command thereafter.

when CMD3_READ =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

  read_r6_response_en <= '1';
  state_leds_out <= "0100";

  cmd_write_en_out <= '0';

when ERROR =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';
  state_leds_out <= "1111";

  cmd_write_en_out <= '0';

when IDLE =>

  cmd_signal <= '1';
  dat3_signal <= '1';
  dat0_signal <= '1';

— Initialization process is done. This will trigger the DATA
— mode on the mux one level up.
  init_done_out <= '1';
  state_leds_out <= "1000";

  cmd_write_en_out <= '0';
— Now that INIT is done, send the RCA signal to the sd_data code.
  rca_out <= card_rca_signal;
end case;
end process;

--SD Card Protocol requires that 74 SD clock cycles come before CMD0 on
--startup. This is also a requirement of SPI mode, but cs_n is held low in
--that instance.

power_on_delay: process(rst_n, clk)
begin
  if (rst_n = '0') then
    pwr_on_delay_count <= 0;
    pwr_on_delay_done <= '0';
  elsif rising_edge(clk) then
    if (pwr_on_delay_en = '1') then
      if (pwr_on_delay_count = 152) then
        pwr_on_delay_done <= '1';
      else
        pwr_on_delay_done <= '0';
        pwr_on_delay_count <= pwr_on_delay_count + 1;
      end if;
    else
      pwr_on_delay_count <= 0;
      pwr_on_delay_done <= '0';
    end if;
  end if;
end process power_on_delay;

--SEND ANY CMD with CRC7 APPEND

cmd_send: process(rst_n, clk)
begin
  if (rst_n = '0') then
    cmdstartbit <= '0';
    command_send_done <= '0';
    command_send_bit_count <= 0;
    output_command <= x"FFFFFFFFFFFF";
  elsif falling_edge(clk) then
    if (command_load_en = '1') then
cmdstartbit <= '1';
end if;
command_send_done <= '0'; -- Default Value
if (command_send_en = '1') then
  if (command_send_bit_count = 48) then
    command_send_done <= '1';
    command_send_bit_count <= 0;
  elsif (crc7_done = '1') then
    output_command <= crc7_signal & '1' & output_command (39 downto 0);
    command_send_bit_count <= command_send_bit_count + 1;
  elsif (cmdstartbit = '1') then
    output_command <= command_signal;
    cmdstartbit <= '0';
  else
    command_send_bit_count <= command_send_bit_count + 1;
    output_command <= output_command (46 downto 0) & '1';
  end if;
else
  output_command <= x" FFFFFFFFFFFF";
end if;
end if;
end process cmd_send;

--CRC7 Gen and Append Process. Used only for CMD3 in this init code.
--All other CRC7 portions of commands are hard coded.
crc7_bitval_signal <= output_command (47);
crc7_done <= '1' when (crc7_send_bit_count = 40) else '0';
crc7_rst_signal <= '1' when (crc7_gen_en = '1') else '0';
crc7_gen: process(rst_n, clk)
begi
  if (rst_n = '0') then
    crc7_send_bit_count <= 0;
  elsif falling_edge(clk) then
    if (crc7_gen_en = '1') then
      crc7_send_bit_count <= crc7_send_bit_count + 1;
    else
      crc7_send_bit_count <= 0;
    end if;
  end if;
end process crc7_gen;
-- Below are incoming response handlers which sample the cmd lines
-- for responses sent after commands.

```
cmd_read_r1_response: process(rst_n, clk)
begin
  if (rst_n = '0') then
    read_r1_response_done <= '0';
    read_bytes <= x"FFFFFFFFFFFFFFF";
    r1_response_bytes <= x"FFFFFFFFFFFFFFF";
    response_1_status <= (others => '0');
    response_1_current_state_bits <= (others => '0');
  elsif rising_edge(clk) then
    read_r1_response_done <= '0';
    if (read_r1_response_en = '1') then
      if (read_bytes(47) = '0') then
        read_r1_response_done <= '1';
        r1_response_bytes <= read_bytes;
      end if;
      response_1_status <= read_bytes(39 downto 8);
      response_1_current_state_bits <= read_bytes(20 downto 17);
    else
      read_bytes <= read_bytes(46 downto 0) &
        cmd_signal_in_signal;
    end if;
  end if;
end process cmd_read_r1_response;
```

```
cmd_read_r2_response: process(rst_n, clk)
begin
  if (rst_n = '0') then
    read_r2_response_done <= '0';
    read_r2_bytes <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
    r2_response_bytes <= x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
  elsif rising_edge(clk) then
    read_r2_response_done <= '0';
    if (read_r2_response_en = '1') then
```
if (read_r2_bytes(135) = '0') then
    read_r2_response_done <= '1';
    r2_response_bytes <= read_r2_bytes;
    read_r2_bytes <= (others => '1');
else
    read_r2_bytes <= read_r2_bytes(134 downto 0) & cmd_signal_in_signal;
end if;
end if;
end if;
end process cmd_read_r2_response;

cmd_read_r3_response: process(rst_n, clk)
begin
    if (rst_n = '0') then
        read_r3_response_done <= '0';
        read_r3_bytes <= x"FFFFFFFFFFFFF";
        r3_response_bytes <= x"FFFFFFFFFFFFF";
    elsif rising_edge(clk) then
        read_r3_response_done <= '0';
        if (read_r3_response_en = '1') then
            if (read_r3_bytes(47) = '0') then
                read_r3_response_done <= '1';
                r3_response_bytes <= read_r3_bytes;
                read_r3_bytes <= x"FFFFFFFFFFFFF";
            else
                read_r3_bytes <= read_r3_bytes(46 downto 0) & cmd_signal_in_signal;
            end if;
        end if;
    end if;
end process cmd_read_r3_response;

cmd_read_r6_response: process(rst_n, clk)
begin
    if (rst_n = '0') then
        read_r6_response_done <= '0';
        read_r6_bytes <= x"FFFFFFFFFFFFF";
        r6_response_bytes <= x"FFFFFFFFFFFFF";
        card_rca_signal <= (others => '0');
        response_6_status <= (others => '0');
elsif rising_edge(clk) then
  read_r6_response_done <= '0';
  if (read_r6_response_en = '1') then
    if (read_r6_bytes(47) = '0') then
      read_r6_response_done <= '1';
      r6_response_bytes <= read_r6_bytes;
      --Store the RCA returned for later use in all commands
      card_rca_signal <= read_r6_bytes(39 downto 24);
      --Debug utility. Handy for looking at the R6 status bits.
      response_6_status <= read_r6_bytes(23 downto 8);
      read_r6_bytes <= x"FFFFFFF";
    else
      read_r6_bytes <= read_r6_bytes(46 downto 0) &
        cmd_signal_in_signal;
    end if;
  end if;
end if;
end process cmd_read_r6_response;

cmd_read_r7_response: process(rst_n, clk)
begin
  if (rst_n = '0') then
    read_r7_response_done <= '0';
    r7_response_bytes <= x"FFFFFFF";
    read_r7_bytes <= x"FFFFFFF";
  elsif rising_edge(clk) then
    read_r7_response_done <= '0';
    if (read_r7_response_en = '1') then
      if (read_r7_bytes(47) = '0') then
        read_r7_response_done <= '1';
        r7_response_bytes <= read_r7_bytes;
        read_r7_bytes <= x"FFFFFFF";
      else
        read_r7_bytes <= read_r7_bytes(46 downto 0) &
          cmd_signal_in_signal;
      end if;
    end if;
  end if;
end process cmd_read_r7_response;
—Here we wait for $d_0$ to return high and stay high before continuing with init.

—$d_0$ will only return high after sd card senses new signalling levels.

—The card is sensing the voltage level of the clk line most likely.

—After the R1 response post CMD11, the card will drop $d_0$.

—$d_0$ only returns to high after the line voltages have changed.

```
Voltage_Switch_Complete: process(rst_n, clk) begin
    if (rst_n = '0') then
        d0_signal_in_count <= "00";
        d0_signal_in_follower <= '1';
        voltage_switch_done <= '0';
    elsif rising_edge(clk) then
        if (voltage_switch_en_signal = '1') then
            --Follower signals allow detecting a transition once
            --and not multiple times.
            if (d0_signal_in_follower /= d0_signal_in) then
                d0_signal_in_follower <= d0_signal_in;
                if (d0_signal_in = '1') then
                    d0_signal_in_count <= d0_signal_in_count + 1;
                end if;
            end if;
        else
            d0_signal_in_count <= "00";
            d0_signal_in_follower <= '1';
        end if;
    else
        if (d0_signal_in_count = 1) then
            voltage_switch_done <= '1';
        else
            voltage_switch_done <= '0';
        end if;
    end if;
end process Voltage_Switch_Complete;
```

—DEBUG COUNTER

—Counter used for measuring time sd card spends in INIT state.

```
COUNT_INIT_TIME: process(rst_n, clk)
```
begin
if (rst_n = '0') then
  init_counter <= (others => '0');
elsif rising_edge(clk) then
  if (current_state /= START_WAIT) then
    if (current_state = IDLE) then
    else
      init_counter <= init_counter + 1;
    end if;
  end if;
end if;
end process COUNT_INIT_TIME;

--Here the voltage on the signal lines is changed through
--interacting with the switch
--controlling the voltage level pin of the level translator.

--Info on the length of switch is NOT provided in the sd spec.
--Try wait 6 ms.
--1 400kHz tick is 2.5us.

voltage_switch_process: process(rst_n, clk) --Sclk is ~400 kHz.
begin
if(rst_n = '0') then
  vc_33_on_out <= '1';
  vc_18_on_out <= '0';
  voltage_switch_run <= '0';
  voltage_switch_lag_counter <= (others => '0');
  voltage_switch_18_stabilize <= (others => '0');
elsif rising_edge(clk) then
  if (voltage_switch_run = '0') then
    if (voltage_switch_en_signal = '1') then
      if (voltage_switch_lag_counter =
          to_unsigned(100, voltage_switch_lag_counter
          'length')) then
        if (voltage_switch_18_stabilize =
            to_unsigned(160,
            voltage_switch_18_stabilize'length))
          then
            voltage_switch_run <= '1';
      end if;
    end if;
  end if;
end process;
vc_33_on_out <= '0';
vc_18_on_out <= '1';
else
  voltage_switch_18_stabilize <=
    voltage_switch_18_stabilize + 1;
vc_33_on_out <= '0';
voltage_switch_18_stabilize <=
    voltage_switch_18_stabilize + 1;
vc_18_on_out <= '1';
end if;
else
vc_33_on_out <= '0';
voltage_switch_run <= '0';
voltage_switch_lag_counter <=
    voltage_switch_lag_counter + 1;
end if;
else
vc_33_on_out <= '1';
voltage_switch_run <= '0';
end if;
end if;

end process voltage_switch_process;

--Pause for 1ms. After R1 from CMD 11 read.
cmd_11_read_pause: process(rst_n, clk)
  -- Sclk is ~400kHz.
begins
  if (rst_n = '0') then
    cmd_11_read_done_pause_done <= '0';
cmd_11_read_done_pause_counter <= (others => '0');
  elsif rising_edge(clk) then
    if (cmd_11_read_done_pause_en = '1') then
      if (cmd_11_read_done_pause_counter =
        to_unsigned(20,cmd_11_read_done_pause_counter'
length)) then
        cmd_11_read_done_pause_done <= '1';
      else
        cmd_11_read_done_pause_counter <=
          cmd_11_read_done_pause_counter + 1;
      end if;
    end if;
  end if;
end if;

end process cmd_11_read_pause;

end Behavioral;
Filename: microsd_data.vhd
Description: Source code for microsd serial data logger
Author: Christopher Casebeer
Lab: Dr. Snider
Department: Electrical and Computer Engineering
Institution: Montana State University
Support: This work was supported under NSF award No. DBI-1254309
Creation Date: June 2014

-- Modification History (give date, author, description)
None

-- Please send bug reports and enhancement requests
to Dr. Snider at rksnider@ece.montana.edu

-- This software is released under
-- The MIT License (MIT)

Copyright (C) 2014 Christopher C. Casebeer and Ross K. Snider

Permission is hereby granted, free of charge, to any person
obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without
limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell
copies of the Software, and to permit persons to whom the
Software is
furnished to do so, subject to the following conditions:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;
microsd_data is the portion of microsd_controller which handles data communications with the microSD card.

@details

@param clk_freq_g Used to determine command time outs.
@param clk Logic Clock and Data Transmission Clock
@param rst_n Active Low Reset
@param sd_control Used to select pathway in Data Component.
@param sd_status Current State of State Machine.

@param block_byte_data Read data from SD card memory
@param block_byte_wren Signals that a data byte has been read.
@param block_read_sd_addr Address to read block from on sd card.
@param block_byte_addr Address to write read data to in ram.
@param block_write_sd_addr Address where block is written on sd card.
@param block_write_data Byte that will be written
@param num_blocks_to_write Number of blocks to be written in CMD25.
@param ram_read_address Where block_write_data is read from.

@param erase_start Start address of erase.
@param erase_end Stop address of erase.
@param state_leds Used to encode current state to Leds.
---! @param prev_block_write_sd_addr The last address to be valid on
---! prev_block_write_sd_addr_pulse '1'
---! @param prev_block_write_sd_addr_pulse prev_block_write_sd_addr is valid
---! @param cmd_write_en Tri State Enable
---! @param D0_write_en Tri State Enable
---! @param D1_write_en Tri State Enable
---! @param D2_write_en Tri State Enable
---! @param D3_write_en Tri State Enable
---! @param cmd_signal_in Read value of the tri-stated line.
---! @param D0_signal_in Read value of the tri-stated line.
---! @param D1_signal_in Read value of the tri-stated line.
---! @param D2_signal_in Read value of the tri-stated line.
---! @param D3_signal_in Read value of the tri-stated line.
---! @param card_rca Card RCA is passed from init.
---! @param init_done Card has passed init phase.
---! @param hs_sdr25_mode_en Card should transition to hs_sdr25 mode before first CMD25.
---! @param voltage_switch_en Enable voltage switching sequence.
---! @param dat0 dat0 line out
---! @param dat1 dat1 line out
---! @param dat2 dat2 line out
---! @param dat3 dat3 line out
---! @param cmd cmd line out
---! @param sclk clk sent to sd card
---! @param restart Card should be restarted at upper level
---! @param data_send_crc_error Card has sensed a data CRC error
---! @param ext_trigger External trigger bit.
Used to trigger an Oscilloscope
if need arise.

entity microsd_data is
  generic(
    clk_freq_g : natural
  );
  port(
    clk : in std_logic;
    rst_n : in std_logic;
    sd_control : in std_logic_vector(7 downto 0);
    sd_status : out std_logic_vector(7 downto 0);
    block_byte_data : out std_logic_vector(7 downto 0);
    block_byte_wren : out std_logic;
    block_read_sd_addr : in std_logic_vector(31 downto 0);
    block_write_sd_addr : in std_logic_vector(31 downto 0);
    block_write_data : in std_logic_vector(7 downto 0);
    num_blocks_to_write : in integer range 0 to 2**16 -1;
    ram_read_address : out std_logic_vector(8 downto 0);
    erase_start : in std_logic_vector(31 downto 0);
    erase_end : in std_logic_vector(31 downto 0);
state_leds : out std_logic_vector (3 downto 0);

prev_block_write_sd_addr : out std_logic_vector (31 downto 0);
prev_block_write_sd_addr_pulse : out std_logic;

cmd_write_en : out std_logic;
D0_write_en : out std_logic;
D1_write_en : out std_logic;
D2_write_en : out std_logic;
D3_write_en : out std_logic;

cmd_signal_in : in std_logic;
D0_signal_in : in std_logic;
D1_signal_in : in std_logic;
D2_signal_in : in std_logic;
D3_signal_in : in std_logic;

-- SD Signals

card_rca : in std_logic_vector (15 downto 0);

init_done : in std_logic;
hs_sdr25_mode_en : in std_logic;

voltage_switch_en : out std_logic;
dat0 : out std_logic;
dat1 : out std_logic;
dat2 : out std_logic;
dat3 : out std_logic;
cmd : out std_logic;
sclk : out std_logic;
restart : out std_logic;
data_send_crc_error : out std_logic;
ext_trigger : out std_logic;

end microsd_data;
microsd_data is responsible for writing, reading, and erase of the card.

It also handles switching into 4 bit mode and also switching into different speed modes.

architecture Behavioral of microsd_data is

component microsd_crc_7 is
  port (
    bitval : in std_logic;
    enable : in std_logic;
    clk : in std_logic;
    rst : in std_logic;
    crc_out : out std_logic_vector(6 downto 0)
  );
end component;

component microsd_crc_16 is
  port (
    bitval : in std_logic;
    enable : in std_logic;
    clk : in std_logic;
    rst : in std_logic;
    crc_out : out std_logic_vector(15 downto 0)
  );
end component;

component counter_ram IS
  PORT(
    address : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
    clock : IN STD_LOGIC := '1';
    data : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    wren : IN STD_LOGIC;
    q : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
  );
END component;

-- FSM State Variables

type sd_data_state is (ENTRY,
APP_WAIT,
CMD7_INIT, CMD7_SEND, CMD7_READ,
CMD6_INIT, CMD6_SEND, CMD6_READ,
CMD6_INIT_4, CMD6_SEND_4, CMD6_READ_4,
CMD17_INIT, CMD17_SEND, CMD17_READ, CMD17_READ_DATA,
CMD24_INIT, CMD24_SEND, CMD24_READ, CMD24_DATA, CMD24_DATA_INIT,
CMD25_INIT, CMD25_SEND, CMD25_READ, CMD25_DATA, CMD25_DATA_INIT,
CMD25_DATA_READ_12, CMD25_DATA_READ_13MULTI,
CMD25_INIT_4, CMD25_SEND_4, CMD25_READ_4, CMD25_DATA_4,
CMD25_DATA_4_READ_TOKEN,
CMD25_DATA_4_READ_CRC_SUCCESS,
CMD25_DATA_4_READ_DECIDE,
CMD25_DATA_4_RESEND,
CMD25_INIT_4_RESEND, CMD25_SEND_4_RESEND,
CMD55_INIT, CMD55_SEND, CMD55_READ,
CMD55_INIT_ACMD6, CMD55_SEND_ACMD6, CMD55_READ_ACMD6,
CMD55_INIT_ACMD42, CMD55_SEND_ACMD42, CMD55_READ_ACMD42,
CMD55_INIT_ACMD13, CMD55_SEND_ACMD13, CMD55_READ_ACMD13,
CMD32_INIT, CMD32_SEND, CMD32_READ,
CMD33_INIT, CMD33_SEND, CMD33_READ,
CMD38_INIT, CMD38_SEND, CMD38_READ,
ACMD23_INIT, ACMD23_SEND, ACMD23_READ,
ACMD6_INIT, ACMD6_SEND, ACMD6_READ,
ACMD13_INIT, ACMD13_SEND, ACMD13_READ,
ACMD42_INIT, ACMD42_SEND, ACMD42_READ,
---APP_CMD17_DATA, APP_CMD18_DATA, APP_CMD12_INIT,
---APP_CMD12_SEND, APP_CMD12_WAIT, APP_CMD12_READ,
---CMD9_INIT, CMD9_SEND, CMD9_READ,
CMD12_INIT, CMD12_SEND, CMD12_READ,
CMD12_INIT_ABORT, CMD12_SEND_ABORT, CMD12_READ_ABORT,
CMD13_INIT, CMD13_SEND, CMD13_READ,
CMD13_INIT_TIMEOUT_REC, CMD13_SEND_TIMEOUT_REC,
CMD13_READ_TIMEOUT_REC,
CMD13_INIT_MULTI, CMD13_SEND_MULTI, CMD13_READ_MULTI,
CMD13_INIT_MULTI_4, CMD13_SEND_MULTI_4, CMD13_READ_MULTI_4,
DELAY,
---APP_CMD12CLEAR_BUFFER, APP_CMD17_CLEAR_BUFFER,
APP_ACK_WAIT,
IDLE, ERROR);
signal current_state : sd_data_state;
signal next_state : sd_data_state;
signal return_state : sd_data_state;

Return state is a way for exiting a shared CMD, such as CMD55!

signal return_state : sd_data_state;

-- SD Data FSM Status and Control Signals
signal sd_status_signal : std_logic_vector (7 downto 0);
signal sd_control_signal : std_logic_vector (7 downto 0);

-- DELAY0 Process Signals
signal delay_done : std_logic;
signal delay_count : integer range 0 to 2**8 - 1;
signal delay_en : std_logic;

-- CMD_SEND0 Process Signals
signal command_load_en : std_logic;
signal output_command : std_logic_vector (47 downto 0);
signal command_send_en : std_logic;
signal command_send_done : std_logic;
signal command_send_bit_count : integer range 0 to 2**6 - 1;

-- CMD_READ_R1_RESPONSE0 Signals
signal read_bytes : std_logic_vector (47 downto 0);
signal read_r1_response_done : std_logic;
signal r1_response_bytes : std_logic_vector (47 downto 0);
signal read_r1_response_en : std_logic;
signal response_1_status : std_logic_vector (31 downto 0);
signal response_1_current_state_bits : std_logic_vector (3 downto 0);

-- CMD_READ_R2_RESPONSE0 Signals
signal r2_response_bytes : std_logic_vector (135 downto 0);
signal read_r2_response_done : std_logic;
signal read_r2_bytes : std_logic_vector(135 downto 0);
signal read_r2_response_en : std_logic;

-- CMD_READ_R6_RESPONSE0 Signals
signal r6_response_bytes : std_logic_vector(47 downto 0);
signal read_r6_response_done : std_logic;
signal read_r6_bytes : std_logic_vector(47 downto 0);
signal read_r6_response_en : std_logic;
signal response_6_status : std_logic_vector(15 downto 0);

-- Data Response Token Signals.
signal reading_data_token_byte : std_logic_vector(7 downto 0);
signal read_data_token_byte : std_logic_vector(7 downto 0);
signal read_data_token_response_done : std_logic;
signal read_data_token_response_en : std_logic;

--- SINGLE_BLOCK_READ0 Process Signals
signal CMD6_D0_read_done : std_logic;
signal block_byte_count : unsigned(9 downto 0);

--- Wren line on ram.
signal block_byte_wren_signal : std_logic;
signal block_bit_count : integer range 0 to 2**3 - 1;
signal block_start_flag : std_logic;

--- Data being read
signal block_byte_data_signal : std_logic_vector(7 downto 0);

--- Data done being read and presented to the outside inbetween bytes.
signal block_byte_data_signal_out : std_logic_vector(7 downto 0);

--- Sync bit 0 of read to block_bit_count 0. Otherwise off by one.
signal start_read_bit : std_logic;
signal singleblock_read_done : std_logic;

--- Where to store data as its read from sd card. 512 locations.
signal ram_write_address_signal : std_logic_vector(8 downto 0);
—Used to enable process involved in single block read.
signal block_read_process_en : std_logic;

—SD Card I/O signals
signal CMD_signal : std_logic;
signal dat0_signal : std_logic;
signal dat1_signal : std_logic;
signal dat2_signal : std_logic;
signal dat3_signal : std_logic;

—40-bit SD Card command register used for form cmd.
signal command_signal : std_logic_vector(47 downto 0);

—CRC7 signals
—Next bit for CRC engine
signal crc7_bitval_signal : std_logic;
signal crc7_rst_signal : std_logic;
signal crc7_signal : std_logic_vector(6 downto 0);
signal crc7_gen_en : std_logic;
signal crc7_done : std_logic;

—48 bits to count before appending CRC7
signal crc7_send_bit_count : integer range 0 to 2**6 - 1;

—CRC16_D0 signals
—Next bit for CRC engine
signal crc16_bitval_signal_D0 : std_logic;
signal crc16_rst_signal_D0 : std_logic;
—Constant change output of CRC16
signal crc16_signal_D0 : std_logic_vector(15 downto 0);
—Register to capture finished CRC16
signal crc16_signal_D0_fin : std_logic_vector(15 downto 0);
signal crc16_gen_en_D0 : std_logic;
signal crc16_done_D0 : std_logic;
signal crc16_send_bit_count_D0 : integer range 0 to 2**4 - 1;

—512 bits to count before appending CRC16
signal crc16_send_byte_count_D0 : integer range 0 to 2**9 - 1;

--CRC16_D1 signals
signal crc16_bitval_signal_D1 : std_logic;
signal crc16_rst_signal_D1 : std_logic;
signal crc16_signal_D1 : std_logic_vector(15 downto 0);
signal crc16_signal_D1_fin : std_logic_vector(15 downto 0);
signal crc16_gen_en_D1 : std_logic;
signal crc16_done_D1 : std_logic;
signal crc16_send_bit_count_D1 : integer range 0 to 2**4 - 1;
signal crc16_send_byte_count_D1 : integer range 0 to 2**9 - 1;

--CRC16_D2 signals
signal crc16_bitval_signal_D2 : std_logic;
signal crc16_rst_signal_D2 : std_logic;
signal crc16_signal_D2 : std_logic_vector(15 downto 0);
signal crc16_signal_D2_fin : std_logic_vector(15 downto 0);
signal crc16_gen_en_D2 : std_logic;
signal crc16_done_D2 : std_logic;
signal crc16_send_bit_count_D2 : integer range 0 to 2**4 - 1;
signal crc16_send_byte_count_D2 : integer range 0 to 2**9 - 1;

--CRC16_D3 signals
signal crc16_bitval_signal_D3 : std_logic;
signal crc16_rst_signal_D3 : std_logic;
signal crc16_signal_D3 : std_logic_vector(15 downto 0);
signal crc16_signal_D3_fin : std_logic_vector(15 downto 0);
signal crc16_gen_en_D3 : std_logic;
signal crc16_done_D3 : std_logic;
signal crc16_send_bit_count_D3 : integer range 0 to 2**4 - 1;
signal crc16_send_byte_count_D3 : integer range 0 to 2**9 - 1;
Store the SD cards relative card address

```
signal card_rca_signal : std_logic_vector(15 downto 0) := x"0000";
```

Master Block write process signals.

```
--Enable for single block writes.
signal block_write_process_en : std_logic;

--Address for reading 512 byte ram of tracking collar system
--Starts at address 0.
signal ram_read_address_signal : std_logic_vector(8 downto 0);

--Keep track of number of bits written while writing
signal wr_block_bit_count : integer range 0 to 2**4 - 1;

--Keep track of number of bytes written while writing
signal wr_block_byte_count : integer range 0 to 2**10;

--Byte to be written to SD Card
signal wr_block_byte_data : std_logic_vector(7 downto 0);

--Flag signalling append of 4 bit crc16.
signal append_crc_4_bit : std_logic;

--Used to insert start bit into the data send.
signal start_bit : std_logic;

--Switch for data load on first byte of multiblock send.
signal load : std_logic;

--Single block done flag
signal block_write_done : std_logic;

--Number of blocks that have been written. Checked to flag
--multiblock write done.
signal num_blocks_written : integer range 0 to 2**16 - 1;

--A Multiblock write has finished
signal multiblock_write_done : std_logic;

--Enable for turning on multiblock writes
signal multiblock_en : std_logic;

--Segmenting out 4 bit writing control signals.
— Enable 4 bit multiblock write process

```vhdl
signal block_write_process_en_4 : std_logic;

— FSM Flow Signals
— Flag to only execute the ACMD6 wide mode switch
— once vs after every exit from APP_WAIT.
```

```vhdl
signal widedone : std_logic;

— Flag to only execute after the CMD6 Access Mode Switch has been completed.
```

```vhdl
signal ac_mode_switch_done : std_logic;

— Tri-State Signals In from SD card
— Tri-state read CMD signal
```

```vhdl
signal cmd_signal_in_signal : std_logic;

— Tri-state read D0 signal
```

```vhdl
signal D0_signal_in_signal : std_logic;

— Block write data alignment signals
— Used to align first bit of any command.
```

```vhdl
signal cmdstartbit : std_logic;

— Last block successfull written and pulse generation signals
— Internal last block written signal
```

```vhdl
signal block_write_sd_addr_interal : std_logic_vector (31 downto 0);

— Flag to indicate that we’ve passed data token read in the CMD25 instance.
```

```vhdl
signal block_success : std_logic;
signal block_success_follower : std_logic;

— Flag associated with CMD25_DATA_INIT_4.
— Used to track inter block transmissions and current block count.
```

```vhdl
signal new_block_write : std_logic;

— Flag associated with CMD25_DATA_INIT_4.
— Used to track inter block transmissions and current block count.
```

```vhdl
signal new_block_write_follower : std_logic;

— Flag associated with CMD25_INIT_4.
— Used to track first block of multiblock transmission.
signal first_block_of_multiblock : std_logic;

--Command Timeout Signals
signal cmd_resend_en : std_logic;

--Calculation of 100ms timeout counter.
constant cmd_timeout : natural := integer(trunc((real( clk_freq_g )) * 100.0E-3));

signal cmd_response_timeout : natural;
signal error_count : unsigned (5 downto 0);
signal restart_response : std_logic;
signal cmd_resend_timer_en : std_logic;
signal cmd_error_rate : unsigned (4 downto 0);

--CRC Error Data Resend Signals
signal resend : std_logic;

--Once resend pulses high, resending stays high until block_success.

--This avoids a latch in the output logic.
signal resending : std_logic;
signal resend_f : std_logic;
signal resend_count : unsigned(5 downto 0);
signal restart_crc : std_logic;
signal data_error_rate : unsigned(3 downto 0);

attribute noprune : boolean;
attribute noprune of response_1_status : signal is true;
attribute noprune of response_1_current_state_bits : signal is true;
attribute noprune of cmd_error_rate : signal is true;
attribute noprune of resend_count : signal is true;

--DEBUG COUNTERS. Used in profilings inter-write delays.
signal cmd13multi_counter : unsigned(31 downto 0);
signal cmd13multi_counter_reg : unsigned(31 downto 0);
signal cmd13multi_counter_done : std_logic;
signal CMD25_number : unsigned (10 downto 0);
signal CMD25_number_1 : std_logic;
signal cmd13multi_counter_done_1 : std_logic;
signal cmd12_13_counter : unsigned (31 downto 0);
signal cmd12_13_counter_reg : unsigned (31 downto 0);
signal cmd25_setup_counter : unsigned (31 downto 0);
signal cmd25_setup_counter_reg : unsigned (31 downto 0);

begin

--CRC used for commands.
i_sd_crc_7_0: microsd_crc_7
port map (bitval => crc7_bitval_signal,
           enable => crc7_gen_en,
           clk => clk,
           rst => crc7_rst_signal,
           crc_out => crc7_signal);

--CRC used for data.
i_sd_crc_16_0: microsd_crc_16
port map (bitval => crc16_bitval_signal_D0,
           enable => crc16_gen_en_D0,
           clk => clk,
           rst => crc16_rst_signal_D0,
           crc_out => crc16_signal_D0);

--CRC used for data.
i_sd_crc_16_1: microsd_crc_16
port map (bitval => crc16_bitval_signal_D1,
           enable => crc16_gen_en_D1,
           clk => clk,
           rst => crc16_rst_signal_D1,
           crc_out => crc16_signal_D1);
--CRC used for data.
i_sd_crc_16_2: microsd_crc_16
port map ( bitval => crc16_bitval_signal_D2,
            enable => crc16_gen_en_D2,
            clk => clk,
            rst => crc16_rst_signal_D2,
            crc_out => crc16_signal_D2 );

--CRC used for data.
i_sd_crc_16_3: microsd_crc_16
port map ( bitval => crc16_bitval_signal_D3,
            enable => crc16_gen_en_D3,
            clk => clk,
            rst => crc16_rst_signal_D3,
            crc_out => crc16_signal_D3 );

--Debug Rams to Characterize Inner Multiblock Write Waits.
-- counter_ram_instant : counter_ram
-- PORT MAP
-- ( address => std_logic_vector(CMD25_number),
      clock => clk,
      data => std_logic_vector(cmd13multi_counter_reg),
      wren => cmd13multi_counter_done
      q =>
      -- );

-- counter_ram_instant_1 : counter_ram
-- PORT MAP
-- ( address => std_logic_vector(CMD25_number),
      clock => clk,
      data => std_logic_vector(cmd12_13_counter_reg),
      wren => cmd13multi_counter_done
      q =>
      -- );
counter_ram_instance_2 : counter_ram

PORT MAP

address => std_logic_vector(CMD25_number),
clock => clk,
data => std_logic_vector(cmd25_setup_counter_reg),
wren => cmd13multi_counter_done

q =>

process(clk)
begin
if (falling_edge(clk)) then
  dat0 <= dat0_signal;
  dat1 <= dat1_signal;
  dat2 <= dat2_signal;
  dat3 <= dat3_signal;
end if;
end process;

-- I/O Signal Assignments

-- Shut the clock to the sd card off when in the APP_WAIT state
sclk <= '0' when (sd_status_signal = x"01") else clk;
CMD <= CMD_signal;

sd_control_signal <= sd_control;

cmd_signal_in_signal <= cmd_signal_in;
D0_signal_in_signal <= D0_signal_in;
ram_read_address <= ram_read_address_signal;

block_byte_data <= block_byte_data_signal_out;
block_byte_wren <= block_byte_wren_signal;
block_byte_addr <= ram_write_address_signal;
prev_block_write_sd_addr <= block_write_sd_addr_internal;

---

STATE MEMORY PROCESS

---

state_transition: process (clk, rst_n)
begin
  if (rst_n = '0') then
    current_state <= ENTRY;
  elsif (rising_edge(clk)) then
    current_state <= next_state;
  end if;
end process;

---

NEXT-STATE LOGIC PROCESS

---

next_state_logic: process (current_state, init_done, delay_done, sd_control_signal, command_send_done, read_r1_response_done, block_write_done, multiblock_write_done, hs_sdr25_mode_en, ac_mode_switch_done, widedone, read_data_token_reponse_done, CMD6_DO_read_done, read_data_token_byte, cmd_resend_en)
begin
  next_state <= current_state;  -- Default to current state
  case current_state is
    when ENTRY =>
      if (init_done = '1') then
        next_state <= DELAY;
      end if;
      -- Early on in development a delay after initialization proved useful.
when DELAY =>
  if (delay_done = '1') then
    — Transition to SD's "Transfer State" where we can read and write from.
    next_state <= CMD7_INIT;
  end if;

—With some slight modifications the block read, single block
—write, multiblock single bit, multiblock with preerase, and erase can be
—brought functional. Focus on 1.8V, 4 bit, CMD25 development
—resulted in these other pathways going somewhat unmaintained.
—Multiblock read however has never been worked on.
—The commands for these operations are already researched.
—Simply exercising the appropriate process enable
—bits and tweaking FSM pathways would result in these other
—paths becoming operational.
—Only the CMD25 pathway is fully functional at this moment.

—MAIN STATE MACHINE BRANCH
when APP_WAIT =>
  case sd_control_signal is
    — Perform a block read
    when x"01" =>
      next_state <= CMD17_INIT;
    —Perform a multiple block read
    —when x"02" =>
    —next_state <= APP_CMD18_INIT;
    —Perform a block write
    when x"03" =>
      next_state <= CMD24_INIT;
    — Perform a multiblock write single bit
    when x"04" =>
      —If the hs_sdr25 bit set, insert a CMD6 to switch to sdr25 mode.
      if (hs_sdr25_mode_en = '1') then
        if (ac_mode_switch_done = '0') then
          next_state <= CMD6_INIT;
--Else start writing.
else
    next_state <= CMD25_INIT;
end if;
else
    next_state <= CMD25_INIT;
end if;

-- Perform a multiblock write with pre-erase
when x"05" =>
    -- return_state <= ACMD23_INIT;
    next_state <= CMD55_INIT;
    -- Perform a multiblock erase.
    when x"0E" =>
        next_state <= CMD32_INIT;
    end when;
end when;

-- Perform a multiblock write 4 bit
---**This is the only pathway currently in operation**---
when x"44" =>
    if (widedone = '0') then
        -- Do a wide mode (4 bit switch).
        -- This requires a CMD55 followed by an ACMD6.
        next_state <= CMD55_INIT_ACMD6;
        -- hs_sdr25_mode_en check and switch is tied into the
        -- next state logic. Unlike in the single bit case above.
    else
        -- If already in 4 bit mode, start or continue writing.
        next_state <= CMD25_INIT_4;
    end if;
end when;
when others =>
    next_state <= APP_WAIT;
end case;

---CMD24(WRITE_SINGLE_BLOCK)
---Single block write.
---Not efficient (10x slower) compared to multiblock (128) writes.
when CMD24_INIT =>
  next_state <= CMD24_SEND;

when CMD24_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD24_READ;
  end if;

when CMD24_READ =>
  if (read_r1_response_done = '1') then
    next_state <= CMD24_DATA_INIT;
  end if;

when CMD24_DATA_INIT =>
  next_state <= CMD24_DATA;

when CMD24_DATA =>
  if(block_write_done = '1') then
    next_state <= CMD13_INIT;
  end if;

-------------
CMD25(WRITE_MULTIPLE_BLOCK)
-------------
The multiblock write command.
-------------
Begins a streaming write to the
-------------
sd card. This is the 1 bit pathway.
-------------

when CMD25_INIT =>
  if(multiblock_write_done = '1') then
    next_state <= CMD12_INIT;
  else
    next_state <= CMD25_SEND;
  end if;

when CMD25_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD25_READ;
  end if;
when CMD25_READ =>
  if (read_r1_response_done = '1') then
    --RDY FOR DATA BIT
    if (response_1_status(8) = '1') then
      next_state <= CMD25_DATA_INIT;
    else
      next_state <= ERROR;
    end if;
  end if;
when CMD25_DATA_INIT =>
  next_state <= CMD25_DATA;
when CMD25_DATA =>
  if (multiblock_write_done = '1') then
    next_state <= CMD25_DATA_READ_12;
  elsif (block_write_done = '1') then
    next_state <= CMD25_DATA_READ_13MULTI;
  end if;
when CMD25_DATA_READ_12 =>
  if (read_data_token_response_done = '1') then
    --Check that data was received okay.
    --SPI Data token of SD card mode.
    if(read_data_token_byte(3 downto 1) = "010") then
      next_state <= CMD12_INIT;
    else
      next_state <= ERROR;
    end if;
  else
    next_state <= CMD25_DATA_READ_12;
  end if;
when CMD25_DATA_READ_13MULTI =>
  if (read_data_token_reponse_done = '1') then
    --Check that data was received okay.
    --SPI Data token of SD card mode.
    if (read_data_token_byte(3 downto 1) = "010") then
      next_state <= CMD13_INIT_MULTI;
    else
      next_state <= ERROR;
end if;
else
next_state <= CMD25_DATA_READ_13MULTI;
end if;

-------------
---CMD25(WRITE_MULTIPLE_BLOCK)
---The multiblock write command.
---Begins a streaming write to the sd card. This is the 4 bit pathway.
-------------

when CMD25_INIT_4 =>
if(multiblock_write_done = '1') then
next_state <= CMD12_INIT;
else
next_state <= CMD25_SEND_4;
end if;
when CMD25_SEND_4 =>
if (command_send_done = '1') then
next_state <= CMD25_READ_4;
end if;
when CMD25_READ_4 =>
if (read_r1_response_done = '1') then
  if (response_1_status(8) = '1') then  ——RDY FOR DATA BIT
    next_state <= CMD25_DATA_INIT_4;
  else
    next_state <= ERROR;
  end if;
  ——This will not work in the resend instance.
  ——Must jump back to either resend or none resend.
  ——elsif(cmd_resend_en = '1') then
  ——next_state <= CMD25_INIT_4;
end if;
when CMD25_DATA_INIT_4 =>
next_state <= CMD25_DATA_4;
when CMD25_DATA_4 =>
  if(block_write_done = '1') then
    next_state <= CMD25_DATA_4_READ_TOKEN;
  end if;

when CMD25_DATA_4_READ_TOKEN =>
  if(read_data_token_response_done = '1') then
    -- Check that data was received okay. SPI Data token of SD card mode.
    if(read_data_token_byte(3 downto 1) = "010") then
      next_state <= CMD25_DATA_4_READ_CRC_SUCCESS;
    else
      -- Else crc token indicates failure.
      -- Interesting note here. If a crc error occurs mid
      -- multiblock, all further blocks are ignored! Must resend
      -- CMD25.
      next_state <= CMD25_DATA_4_RESEND;
    end if;
  else
    next_state <= CMD25_DATA_4_READ_TOKEN;
  end if;

  -- Resend bit set in this state to trigger address handling due to error.
when CMD25_DATA_4_RESEND =>
  next_state <= CMD12_INIT_ABORT;

when CMD25_INIT_4_RESEND =>
  if(multiblock_write_done = '1') then
    next_state <= CMD12_INIT;
  else
    next_state <= CMD25_SEND_4_RESEND;
  end if;

  -- This CMD25 contains the address we wish to try write again.
when CMD25_SEND_4_RESEND =>
  if(command_send_done = '1') then
    next_state <= CMD25_READ_4;
  end if;
when CMD25_DATA_4_READ_CRC_SUCCESS =>
  next_state <= CMD25_DATA_4_READ_DECIDE;
when CMD25_DATA_4_READ_DECIDE =>
  if (multiblock_write_done = '1') then
    next_state <= CMD12_INIT;
  else
    next_state <= CMD13_INIT_MULTI_4;
  end if;

---CMD13(SEND_STATUS)
---Card Status is returned.
---This is used to check if the card
---is ready for data before sending
---another block to the card in a
---multiblock stream/write.

---CMD13’s are broken into single bit and multi bit and
---inbetween
---blocks and at the end of a CMD25/CMD12
---CMD13 Returns card status field in an R1 response.
---Used for checking
---that card is READY_FOR_DATA inbetween blocks of
---a multiblock write primarily.
---Other bits of interest are APP_CMD, ILLEGAL COMMAND, etc.

when CMD13_INIT =>
  next_state <= CMD13_SEND;
when CMD13_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD13_READ;
  end if;
when CMD13_READ =>
  if (read_r1_response_done = '1') then
      --Card Status READY FOR DATA Page 76 SD Spec.
      if (response_1_status(8) = '1') then
          next_state <= APP_WAIT;
      else
          next_state <= CMD13_INIT;
      end if;
  end if;

--This is an example recovery path for command crc error
--error recovery.

when CMD13_INIT_TIMEOUT_REC =>
  next_state <= CMD13_SEND_TIMEOUT_REC;
when CMD13_SEND_TIMEOUT_REC =>
  if (command_send_done = '1') then
      next_state <= CMD13_READ_TIMEOUT_REC;
  end if;
when CMD13_READ_TIMEOUT_REC =>
  if (read_r1_response_done = '1') then
      --Card Status READY FOR DATA Page 76 SD Spec.
      if (response_1_status(8) = '1') then
          next_state <= return_state;
      else
          next_state <= CMD13_INIT_TIMEOUT_REC;
      end if;
  end if;

--CMD13(SEND_STATUS)
--SD Status is returned.
--This is used to check if the
--card is ready for data before
--sending another block to the
--card in a multiblock stream/write.
--**The CMD13 for the 1 bit multiblock path**
CMD13 string of commands used with CMD25 Multiblock write.

Used between blocks of a multiblock write.

when CMD13_INIT_MULTI =>
  next_state <= CMD13_SEND_MULTI;

when CMD13_SEND_MULTI =>
  if (command_send_done = '1') then
    next_state <= CMD13_READ_MULTI;
  end if;

when CMD13_READ_MULTI =>
  if (read_r1_response_done = '1') then
    Card Status READY FOR DATA
    if (response_1_status (8) = '1') then
      next_state <= CMD25_DATA_INIT;
    else
      next_state <= CMD13_INIT_MULTI;
    end if;
  end if;

-----------

CMD13(SEND STATUS)

SD Status is returned.

This is used to check if the card is ready for data before sending another block to the card in a multiblock stream/write.

-----------

The 4 bit multiblock path

CMD13 string of commands used with CMD25 Multiblock write.

Used between blocks of a multiblock write.

when CMD13_INIT_MULTI_4 =>
  next_state <= CMD13_SEND_MULTI_4;

when CMD13_SEND_MULTI_4 =>
  if (command_send_done = '1') then
    next_state <= CMD13_READ_MULTI_4;
  end if;

when CMD13_READ_MULTI_4 =>
  if (read_r1_response_done = '1') then
if (response_1_status(8) = '1') then 
   Card Status
   READY_FOR_DATA
   if (resending = '0') then
      next_state <= CMD25_DATA_INIT_4;
   else
      next_state <= CMD25_INIT_4_RESEND;
   end if;
else
   next_state <= CMD13_INIT_MULTI_4;
end if;

---

---CMD7(SELECT/DESELECT_CARD)
---Take card from Standby to Transfer State.
---
---After init, take the card from standby to transfer mode.
---A nice state diagram exists in the SD SPEC. Page 27.
---Command to take SD card from standby mode to transfer mode,
---from which data writing is initiated.
when CMD7_INIT =>
   next_state <= CMD7_SEND;

when CMD7_SEND =>
   if (command_send_done = '1') then
      next_state <= CMD7_READ;
   end if;

when CMD7_READ =>
   if (read_r1_response_done = '1') then
      ---If the bad command or crc error flags are not set.
      if (response_1_status(23 downto 22) = "00") then
         next_state <= APP_WAIT;
      else
         next_state <= CMD7_INIT;
      end if;
   end if;
   ---An example recovery scenario. Additional paths needed
   ---for the entire protocol but this is proof that this scheme
   ---will work.
elsif (cmd_resend_en = '1') then
  return_state <= CMD7_INIT;
  next_state <= CMD13_INIT_TIMEOUT_REC;
end if;

---CMD6(SWITCH_FUNC)
---Put card into HS_SDR25 Mode for 25–50Mhz.
---CMD6 is the function switch command. Page 41 SD Spec 3.01
---1 bit pathway
---CMD6 is the function switch command.
---We can change speeds and current levels. See page 44 Sd Spec 3.01
when CMD6_INIT =>
  next_state <= CMD6_SEND;
when CMD6_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD6_READ;
  end if;
when CMD6_READ =>
  --CMD6 inquiry and set function both included 512 bits sent
  --back to the host over D0. We must wait for these even
  --we don’t do anything with them currently.
  --Response was checked and written up to verify it was
done correctly.
  if (CMD6_D0_read_done = '1') then
    next_state <= CMD25_INIT;
  end if;

---CMD6(SWITCH_FUNC)
---Put card into HS_SDR25 Mode for 25–50Mhz.
---CMD6 is the function switch command. Page 41 SD Spec 3.01
—4 bit pathway
—CMD6 is the function switch command.
—We can change speeds and current levels.
—See page 44 Sd Spec 3.01

when CMD6_INIT_4 =>
  next_state <= CMD6_SEND_4;

when CMD6_SEND_4 =>
  if (command_send_done = '1') then
    next_state <= CMD6_READ_4;
  end if;

when CMD6_READ_4 =>
  --CMD6 inquiry and set function both included
  --512 bits sent to the host over D0.
  --We must wait for these bit before proceeding,
  --even if they aren't checked
  --with program
  if (CMD6_D0_read_done = '1') then
    next_state <= CMD25_INIT_4;
  end if;
  elsif (cmd_resend_en = '1') then
    next_state <= CMD6_INIT_4;
  end if;

--

--CMD32/33/38 are the ERASE COMMANDS
--
--CMD32(ERASE_WR_BLK_START) Set start of erase.

when CMD32_INIT =>
  next_state <= CMD32_SEND;

when CMD32_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD32_READ;
  end if;

when CMD32_READ =>
  if (read_r1_response_done = '1') then
if (response_1_status(23 downto 22) = "00") then
  next_state <= CMD33_INIT;
else
  next_state <= ERROR;
end if;

---CMD33(ERASE_WR_BLK_END) Set end of erase.

when CMD33_INIT =>
  next_state <= CMD33_SEND;
when CMD33_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD33_READ;
  end if;
when CMD33_READ =>
  if (read_r1_response_done = '1') then
    if (response_1_status(23 downto 22) = "00") then
      next_state <= CMD38_INIT;
    else
      next_state <= ERROR;
    end if;
  end if;

---CMD38(ERASE_WR_BLK_END) Erase!

when CMD38_INIT =>
  next_state <= CMD38_SEND;
when CMD38_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD38_READ;
  end if;
when CMD38_READ =>
  if (read_r1_response_done = '1') then
    if (response_1_status(23 downto 22) = "00") then
      next_state <= CMD13_INIT;
    else
      next_state <= ERROR;
    end if;
  end if;
end if;
end if;

--CMD12(STOP_TRANSMISSION)
--Stop a multiblock write transmission.

--CMD12 is the stop data send command. Used to cap a
--multiblock stream during a CMD25.
when CMD12_INIT =>
  next_state <= CMD12_SEND;
when CMD12_SEND =>
  if (command_send_done = '1') then
    next_state <= CMD12_READ;
  end if;
when CMD12_READ =>
  if (read_r1_response_done = '1') then
    --check for COM_CRC_ERROR and ILLEGAL COMMAND
    if (response_1_status(23 downto 22) = "00") then
      next_state <= CMD13_INIT;
    else
      next_state <= ERROR;
    end if;
  end if;

--These CMD12's are jumped to following a data response
token
--indicating anything besides succesfull receive of a data
--block by the sd card.
when CMD12_INIT_ABORT =>
  next_state <= CMD12_SEND_ABORT;
when CMD12_SEND_ABORT =>
  if (command_send_done = '1') then
    next_state <= CMD12_READ_ABORT;
  end if;
when CMD12_READ_ABORT =>
  if (read_r1_response_done = '1') then
    --check for COM_CRC_ERROR and ILLEGAL COMMAND
    if (response_1_status(23 downto 22) = "00") then
      next_state <= CMD13_INIT_MULTI_4;
else
    next_state <= ERROR;
end if;
end if;

---CMD55(APP_CMD)
--This command is sent before any ACMD.
--This simply tells the card that an expanded command
--is coming next.
---
-- Send CMD55 (APP_CMD) to the SD card to preface
impending
--- application-specific command
when CMD55_INIT =>
    next_state <= CMD55_SEND;
when CMD55_SEND =>
    if (command_send_done = '1') then
        next_state <= CMD55_READ;
    end if;
when CMD55_READ =>
    if (read_r1_response_done = '1') then
        -- This bit is the APP_CMD bit of the status register.
        -- The other bit set on CMD55 response is READY FOR
        -- DATA.
        if (r1_response_bytes(13) = '1') then
            -- next_state <= return_state;
        else
            next_state <= ERROR;
        end if;
    end if;
---ACMD23(SET_WR_BLK_ERASE_COUNT)
--Used to pre-erase before a write.
---
when ACMD23_INIT =>
    next_state <= ACMD23_SEND;
when ACMD23_SEND =>
if (command_send_done = '1') then
    next_state <= ACMD23_READ;
end if;

when ACMD23_READ =>
    if (read_r1_response_done = '1') then
        -- Checksum is okay and command valid.
        if (response_1_status(23 downto 22) = "00") then
            next_state <= CMD25_INIT;
        else
            next_state <= ERROR;
        end if;
    end if;

when ACMD6_INIT =>
    next_state <= ACMD6_SEND;

when ACMD6_SEND =>
    if (command_send_done = '1') then
        next_state <= ACMD6_READ;
    end if;

when ACMD6_READ =>
    if (read_r1_response_done = '1') then
        -- Checksum is okay and command valid.
        if (response_1_status(23 downto 22) = "00") then
            if (hs_sdr25_mode_en = '1') then
                next_state <= CMD6_INIT_4;
            else
                next_state <= CMD25_INIT_4;
            end if;
        else
            next_state <= ERROR;
        end if;
    else
        next_state <= ERROR;
    end if;

elseif (cmd_resend_en = '1') then
    -- next_state <= ACMD6_INIT;
end if;

-------------
when ACMD13_INIT =>
  next_state <= ACMD13_SEND;

when ACMD13_SEND =>
  if (command_send_done = '1') then
    next_state <= ACMD13_READ;
  end if;

when ACMD13_READ =>
  -- 512 bits sent to the host over D0. We must wait for these.
  if (CMD6_D0_read_done = '1') then
    next_state <= CMD25_INIT_4;
  end if;

when ACMD42_INIT =>
  next_state <= ACMD42_SEND;

when ACMD42_SEND =>
  if (command_send_done = '1') then
    next_state <= ACMD42_READ;
  end if;

when ACMD42_READ =>
  if (read_r1_response_done = '1') then
    if (response_1_status(23 downto 22) = "00") then
      next_state <= CMD55_INIT_ACMD13;
else
    next_state <= ERROR;
end if;

--CMD17(READ_SINGLE_BLOCK) Read a single block.
--Not completely working but close.

when CMD17_INIT =>
    next_state <= CMD17_SEND;
when CMD17_SEND =>
    if (command_send_done = '1') then
        next_state <= CMD17_READ;
    end if;
when CMD17_READ =>
    if (read_r1_response_done = '1') then
        next_state <= CMD17_READ_DATA;
    end if;
when CMD17_READ_DATA =>
    if (singleblock_read_done = '1') then
        next_state <= APP_WAIT;
    end if;
when IDLE =>
    next_state <= IDLE;
when ERROR =>
    next_state <= ERROR;

--The following CMD55s are pathways to get to particular APP CMDS.
--Each APP CMD has its one CMD55 pathway.
--This method was done as creating a
--return_state enumeration caused warnings initially.
--Send CMD55 (APP_CMD) to the SD card to preface impending
--application-specific command
when CMD55_INIT_ACMD6 =>
   next_state <= CMD55_SEND_ACMD6;
when CMD55_SEND_ACMD6 =>
   if (command_send_done = '1') then
      next_state <= CMD55_READ_ACMD6;
   end if;
when CMD55_READ_ACMD6 =>
   if (read_r1_response_done = '1') then
      -- This bit is the APP_CMD bit of the status register.
      -- The other bit set on first CMD55 response is READY FOR DATA.
      if (r1_response_bytes(13) = '1') then
         next_state <= ACMD6_INIT;
      else
         next_state <= ERROR;
      end if;
      -- elsif (cmd_resend_en = '1') then
      --   next_state <= CMD55_INIT_ACMD6;
      end if;
   -- Send CMD55 (APP_CMD) to the SD card
   -- to preface impending application-specific command
when CMD55_INIT_ACMD42 =>
   next_state <= CMD55_SEND_ACMD42;
when CMD55_SEND_ACMD42 =>
   if (command_send_done = '1') then
      next_state <= CMD55_READ_ACMD42;
   end if;
when CMD55_READ_ACMD42 =>
   if (read_r1_response_done = '1') then
      -- This bit is the APP_CMD bit of the status register.
      -- The other bit set on first CMD55 response is READY FOR DATA.
      if (r1_response_bytes(13) = '1') then
         next_state <= ACMD42_INIT;
      else
         next_state <= ERROR;
      end if;
      -- elsif (cmd_resend_en = '1') then
      --   next_state <= CMD55_INIT_ACMD42;
      end if;
Send CMD55 (APP_CMD) to the SD card to preface impending application-specific command

when CMD55_INIT_ACMD13 =>
  next_state <= CMD55_SEND_ACMD13;

when CMD55_SEND_ACMD13 =>
  if (command_send_done = '1') then
    next_state <= CMD55_READ_ACMD13;
  end if;

when CMD55_READ_ACMD13 =>
  if (read_r1_response_done = '1') then
    if (r1_response_bytes(13) = '1') then
      next_state <= ACMD13_INIT;
    else
      next_state <= ERROR;
    end if;
  end if;

end case;

end process;

process (current_state)
begin

  -- Default Signal Values
  command_load_en <= '0';
  command_send_en <= '0';
  read_r1_response_en <= '0';
  read_r6_response_en <= '0';
command_signal <= x"FFFFFFFFFFFFFF";
sd_status_signal <= x"FF";
crc7_gen_en <= '0';
crc16_gen_en_D0 <= '0';
crc16_gen_en_D1 <= '0';
crc16_gen_en_D2 <= '0';
crc16_gen_en_D3 <= '0';
block_write_process_en <= '0';
block_write_process_en_4 <= '0';
multiblock_en <= '0';
block_read_process_en <= '0';
read_data_token_reponse_en <= '0';
crc16_bitval_signal_D0 <= '0';
crc16_bitval_signal_D1 <= '0';
crc16_bitval_signal_D2 <= '0';
crc16_bitval_signal_D3 <= '0';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
card_rca_signal <= card_rca;
resend <= '0';
block_success <= '0';
ext_trigger <= '0';
ew_block_write <= '0';
case current_state is
  ENTRY is the state where the FSM sits while init is done.
  when ENTRY =>
CMD_signal <= '1';

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

card_rca_signal <= card_rca;
state_leds <= "0000";
sd_status_signal <= x"00";

---

---APP_WAIT is the central state from
---which a control signal launches
---the state machine.
---

when APP_WAIT =>

CMD_signal <= '1';

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

state_leds <= "0001";
sd_status_signal <= x"01";
---CMD24(Send Single Block).
---Send a single block to the sd card.

when CMD24_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_signal <= '0' & '1' & "011000" & block_write_sd_addr & x"01";
command_load_en <= '1';

state_leds <= "0010";
sd_status_signal <= x"02";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD24_SEND =>

CMD_signal <= output_command(47);

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_send_en <= '1';
crc7_gen_en <= '1';

state_leds <= "0011";
sd_status_signal <= x"03";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD24_READ =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0100";
sd_status_signal <= x"04";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD24_DATA_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
state_leds <= "1111";
sd_status_signal <= x"05";
cmd_write_en <= '0';
D0_write_en <= '1';
D1_write_en <= '1';
D2_write_en <= '1';
D3_write_en <= '1';

when CMD24_DATA =>

dat0_signal <= wr_block_byte_data(7);
CMD_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

block_write_process_en <= '1';

state_leds <= "0101";
sd_status_signal <= x"06";
cmd_write_en <= '0';
D0_write_en <= '1';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc16_gen_en_D0 <= '1';

custom state. Delay inserted when coming into DATA FSM.
Card was found unresponsive immediately after init.

when DELAY =>

CMD_signal <= '1';

state_leds <= "0110";
sd_status_signal <= x"0D";
cmd_write_en <= '0';

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
delay_en <= '1';
---CMD13(SEND_STATUS)
---SD Status is returned. This is used to check if the card is
---ready for data before sending another block to the card in
---a multiblock stream/write.
---This CMD13 is used once after CMD12 and not inbetween
---blocks of multiblock write.

when CMD13_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

--IMPORTANT: The CRC7 is defaulted to x"01".
--It will be filled by CRC7GEN and cmdsend processes.
command_signal <= '0' & '1' & "001101"
& card_rca_signal & x"000001";
command_load_en <= '1';
state_leds <= "0100";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD13_SEND =>

CMD_signal <= output_command(47);
--IMPORTANT: The CRC7 is defaulted to x"01".
--It will be filled by CRC7GEN and CMD_SEND
command_signal <= '0' & '1' & "001101"
& card_rca_signal & x"000001";
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "0101";
cmd_write_en <= '1';
crc7_gen_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD13_READ =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  read_r1_response_en <= '1';
  state_leds <= "0110";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
  sd_status_signal <= x"68";

——CMD7(SELECT/DESELECT_CARD)
——Take card from Standby to Transfer State.

when CMD7_INIT =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_signal <= '0' & '1' & "000111"
               & card_rca_signal & x"000001";
  command_load_en <= '1';
  state_leds <= "0100";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
when CMD7_SEND =>

    CMD_signal <= output_command(47);
    command_signal <= '0' & '1' & "000111"
                & card_rca_signal & x"000001";
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
    dat3_signal <= '1';
    command_send_en <= '1';
    state_leds <= "0101";
    cmd_write_en <= '1';
    D0_write_en <= '0';
    D1_write_en <= '0';
    D2_write_en <= '0';
    D3_write_en <= '0';
    crc7_gen_en <= '1';

when CMD7_READ =>

    CMD_signal <= '1';
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
    dat3_signal <= '1';
    read_r1_response_en <= '1';
    state_leds <= "0110";
    cmd_write_en <= '0';
    D0_write_en <= '0';
    D1_write_en <= '0';
    D2_write_en <= '0';
    D3_write_en <= '0';
    ext_trigger <= '1';

—−CMD6(SELECT/DESELECT_CARD)
—−Put card into HS_SDR25 Mode for 25−50Mhz.
CMD6 is the function switch command. Page 41 SD Spec 3.01

when CMD6_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

-- This command it attempting an actual switch "80"
    rather
-- than a check [most sig bit] and changing
-- to high speed mode, "FFFFF1", the '1' being the mode
    switch.
command_signal <= '0' & '1' & "000110"
    & x"80" & x"FFFFF1" & x"01";

command_load_en <= '1';
state_leds <= "0100";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD6_SEND =>

CMD_signal <= output_command(47);

-- This command it attempting an actual switch "80"
    rather
-- than a check [most sig bit] and changing
-- to high speed mode, "FFFFF1", the '1' being the mode
    switch.
command_signal <= '0' & '1' & "000110"
    & x"80" & x"FFFFF1" & x"01";

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "0101";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc7_gen_en <= '1';

when CMD6_READ =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
block_read_process_en <= '1';

——CMD6(SWITCH_FUNC)
—Put card into HS_SDR25 Mode for 25–50Mhz.

——CMD6 is the function switch command. Page 41 SD Spec 3.01
—4 bit pathway

when CMD6_INIT_4 =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

—This command does a function switch "80" rather
—than a function check [most sig bit].
—We change to high speed mode, "FFFFF1",
—the '1' being the mode switch.
command_signal <= '0' & '1' & "000110"
 & x"80" & x"FFFFF1" & x"01";
command_load_en <= '1';
state_leds <= "0100";

when CMD6_SEND_4 =>
  CMD_signal <= output_command(47);
  --This command it attempting an actual switch "80"
  --than a check [most sig bit] and changing
  --to high speed mode, "FFFFF1", the '1' being the mode
  --switch.
  command_signal <= '0' & '1' & "000110" & x"80"
  & x"FFFFF1" & x"01";
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_send_en <= '1';
  state_leds <= "0101";
  --Here I am driving all the lines into a known state.
  --I am sensing the data lines start bit,
  --so they need to be in a known state.
  --Hopefully these lines won't droop
  --(or level translator won't change them)
  --before the sd card starts to drive them.
  cmd_write_en <= '1';
  D0_write_en <= '1';
  D1_write_en <= '1';
  D2_write_en <= '1';
  D3_write_en <= '1';
  crc7_gen_en <= '1';

when CMD6_READ_4 =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
278

2063  dat2_signal <= '1';
2064  dat3_signal <= '1';
2065  read_r1_response_en <= '1';
2066  state_leds <= "0110";
2067  cmd_write_en <= '0';
2068  D0_write_en <= '0';
2069  D1_write_en <= '0';
2070  D2_write_en <= '0';
2071  D3_write_en <= '0';
2072  block_read_process_en <= '1';
2073
2074
2075  _______________________________________________________________________
2076  --ACMD13(SD_STATUS)
2077  --Send back the sd status.
2078  --Only used for debug currently.
2079  --512 status bits will come back on the D0 line.
2080  _______________________________________________________________________
2081
2082  when ACMD13_INIT =>
2083
2084      CMD_signal <= '1';
2085      dat0_signal <= '1';
2086      dat1_signal <= '1';
2087      dat2_signal <= '1';
2088      dat3_signal <= '1';
2089      command_signal <= '0' & '1' & "001101" & x"00" & x"000000" & x"01";
2090
2091      command_load_en <= '1';
2092      state_leds <= "0100";
2093      cmd_write_en <= '0';
2094      D0_write_en <= '0';
2095      D1_write_en <= '0';
2096      D2_write_en <= '0';
2097      D3_write_en <= '0';
2098
2099  when ACMD13_SEND =>
2100
2101      CMD_signal <= output_command(47);
2102      dat0_signal <= '1';
2103      dat1_signal <= '1';
2104      dat2_signal <= '1';
2105      dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "0101";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc7_gen_en <= '1';

when ACMD13_READ =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

--ACMD13 sends back 512 bit SD STATUS register on D0.
block_read_process_en <= '1';

--------
--ACMD42(SET CLR CARD DETECT)
--Program/Deprogram the pullup resistor on D3.
--Tested but never used.
--------
--Never implemented, but played with during debugging.

when ACMD42_INIT =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

--Least significant bit '0' in 32 bit stuff bits is disabling
command_signal <= '0' & '1' & "101010" & x"00" & x"000000" & x"01";

command_load_en <= '1';
state_leds <= "0100";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when ACMD42_SEND =>
  CMD_signal <= output_command(47);
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_send_en <= '1';
  state_leds <= "0101";
  cmd_write_en <= '1';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
  crc7_gen_en <= '1';

when ACMD42_READ =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  read_r1_response_en <= '1';
  state_leds <= "0110";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
CMD32(ERASE_WR_BLK_START)

--- Set start of erase.

when CMD32_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "100000" & erase_start & x "01";

command_load_en <= '1';
state_leds <= "0100";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD32_SEND =>

CMD_signal <= output_command(47);
command_signal <= '0' & '1' & "100000" & erase_start & x "01";
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "0101";
sd_status_signal <= x"E0";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc7_gen_en <= '1';
when CMD32_READ =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
sd_status_signal <= x"E0";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

---CMD33(ERASE_WR_BLK_END)
---Set end of erase.
when CMD33_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "100001" & erase_end & x"01";

command_load_en <= '1';
state_leds <= "0100";
sd_status_signal <= x"E0";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD33_SEND =>
CMD_signal <= output_command(47);
command_signal <= '0' & '1' & "100001" & erase_end & x"01";
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "0101";
sd_status_signal <= x"E0";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc7_gen_en <= '1';

when CMD33_READ =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
sd_status_signal <= x"E0";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc7_gen_en <= '1';

when CMD38_INIT =>

——CMD38(ERASE_WR_BLK_END)
——Erase!

when CMD38_INIT =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "100110" & x"0000000001";

command_load_en <= '1';
state_leds <= "0100";
sd_status_signal <= x"E0";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD38_SEND =>

    CMD_signal <= output_command(47);
    command_signal <= '0' & '1' & "100110" & x"0000000001";
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
    dat3_signal <= '1';
    command_send_en <= '1';
    state_leds <= "0101";
    sd_status_signal <= x"E0";
    cmd_write_en <= '1';
    D0_write_en <= '0';
    D1_write_en <= '0';
    D2_write_en <= '0';
    D3_write_en <= '0';

crc7_gen_en <= '1';

when CMD38_READ =>

    CMD_signal <= '1';
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
\begin{verbatim}
  dat3_signal <= '1';
  read_r1_response_en <= '1';
  state_leds <= "0110";
  sd_status_signal <= x"E0";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';

---CMD25(WRITE_MULTIPLE_BLOCK)
---The multiblock write command. Begins a streaming
---write to the sd card. This is the 1 bit pathway.

when CMD25_INIT =>

  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_signal <= '0' & '1' & "011001" &
                   block_write_sd_addr & x"01";
  command_load_en <= '1';

  state_leds <= "0010";
  sd_status_signal <= x"02";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
  sd_status_signal <= x"40";

  multiblock_en <= '1';

when CMD25_SEND =>

  CMD_signal <= output_command(47);
\end{verbatim}
command_signal <= '0' & '1' & "011001" &
    block_write_sd_addr & x"01";

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_send_en <= '1';
crc7_gen_en <= '1';

state_leds <= "0011";
sd_status_signal <= x"03";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"41";

state_leds <= "0100";
sd_status_signal <= x"04";
cmd_write_en <= '0';
D0_write_en <= '1';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"42";

state_leds <= "0100";
sd_status_signal <= x"04";
cmd_write_en <= '0';
D0_write_en <= '1';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"42";

when CMD25_READ =>

    CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0100";
sd_status_signal <= x"04";
cmd_write_en <= '0';
D0_write_en <= '1';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"42";

when CMD25_DATA_INIT =>

    CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
state_leds <= "1111";
sd_status_signal <= x"05";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"43";

multiblock_en <= '1';

when CMD25_DATA =>

dat0_signal <= wr_block_byte_data(7);
crc16_bitval_signal_D0 <= wr_block_byte_data(7);
crc16_gen_en_D0 <= '1';
CMD_signal <= '1';

dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
block_write_process_en <= '1';
multiblock_en <= '1';
state_leds <= "0101";
sd_status_signal <= x"06";
cmd_write_en <= '0';
D0_write_en <= '1';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc16_gen_en_D0 <= '1';
sd_status_signal <= x"44";

when CMD25_DATA_READ_12 =>

read_data_token_response_en <= '1';
CMD_signal <= '1';
state_leds <= "0101";
sd_status_signal <= x"06";

block_write_process_en <= '1';
multiblock_en <= '1';
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

sd_status_signal <= x"44";

when CMD25_DATA_READ_13MULTI =>
read_data_token_reponse_en <= '1';
CMD_signal <= '1';
state_leds <= "0101";
sd_status_signal <= x"06";

block_write_process_en <= '1';
multiblock_en <= '1';

when CMD25_INIT_4 =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';

---CMD25(WRITE_MULTIPLE_BLOCK)
---The multiblock write command.
---Begins a streaming write to the sd card. This is the 4 bit pathway.
2548    dat3_signal <= '1';
2549
2550    command_signal <= '0' & '1' & "011001" &
2551        block_write_sd_addr & x"01";
2552    command_load_en <= '1';
2553
2554    state_leds <= "0010";
2555    sd_status_signal <= x"02";
2556    cmd_write_en    <= '0';
2557    D0_write_en    <= '0';
2558    D1_write_en    <= '0';
2559    D2_write_en    <= '0';
2560    D3_write_en    <= '0';
2561    sd_status_signal <= x"40";
2562    multiblock_en <= '1';
2563
2564    when CMD25_SEND_4 =>
2565        CMD_signal <= output_command(47);
2566        command_signal <= '0' & '1' & "011001" &
2567            block_write_sd_addr & x"01";
2568
2569    dat0_signal <= '1';
2570    dat1_signal <= '1';
2571    dat2_signal <= '1';
2572    dat3_signal <= '1';
2573
2574    command_send_en <= '1';
2575    crc7_gen_en <= '1';
2576
2577    state_leds <= "0011";
2578    sd_status_signal <= x"03";
2579    cmd_write_en    <= '1';
2580    D0_write_en    <= '0';
2581    D1_write_en    <= '0';
2582    D2_write_en    <= '0';
2583    D3_write_en    <= '0';
2584    sd_status_signal <= x"41";
2585
2586    multiblock_en <= '1';
2587
2588    when CMD25_READ_4 =>
2589        when CMD25_READ_4 =>
2590
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0100";
sd_status_signal <= x"04";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"42";
multiblock_en <= '1';

when CMD25_DATA_INIT_4 =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
state_leds <= "1111";
sd_status_signal <= x"05";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"43";
multiblock_en <= '1';
new_block_write <= '1';

when CMD25_DATA_4 =>

if (append_crc_4_bit = '0') then
dat0_signal <= wr_block_byte_data(4);
else
dat0_signal <= crc16_signal_D0_fin(15);
end if;
if (append_crc_4_bit = '0') then
    dat1_signal <= wr_block_byte_data(5);
--Point to test crc errors in stream and resend.
--To disable to not increment data_error_rate.
elsif (data_error_rate = to_unsigned(9, data_error_rate'length)) then
    dat1_signal <= '1';
else
    dat1_signal <= crc16_signal_D1_fin(15);
end if;

if (append_crc_4_bit = '0') then
    dat2_signal <= wr_block_byte_data(6);
else
    dat2_signal <= crc16_signal_D2_fin(15);
end if;

if (append_crc_4_bit = '0') then
    dat3_signal <= wr_block_byte_data(7);
else
    dat3_signal <= crc16_signal_D3_fin(15);
end if;
--Send data into CRC components
    crc16_bitval_signal_D0 <= wr_block_byte_data(4);
crc16_bitval_signal_D1 <= wr_block_byte_data(5);
crc16_bitval_signal_D2 <= wr_block_byte_data(6);
crc16_bitval_signal_D3 <= wr_block_byte_data(7);
    CMD_signal <= '1';
block_write_process_en_4 <= '1';
multiblock_en <= '1';
state_leds <= "0101";
sd_status_signal <= x"06";
cmd_write_en <= '0';
D0_write_en <= '1';
D1_write_en <= '1';
D2_write_en <= '1';
D3_write_en <= '1';
crc16_gen_en_D0 <= '1';
crc16_gen_en_D1 <= '1';
crc16_gen_en_D2 <= '1';
crc16_gen_en_D3 <= '1';
Read the data response token that comes after a sent block.

Jump to appropriate CMD (12/13) depending if we want to send more blocks.

when CMD25_DATA_4_READ_TOKEN =>
  read_data_token_response_en <= '1';
  CMD_signal <= '1';
  state_leds <= "0101";
  sd_status_signal <= x"06";
  multiblock_en <= '1';
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
  sd_status_signal <= x"44";

--CRC SUCCESS states.
--These are included to detect if the last block was received correctly.

when CMD25_DATA_4_READ_CRC_SUCCESS =>
  CMD_signal <= '1';
  state_leds <= "0101";
  sd_status_signal <= x"06";
multiblock_en <= '1';

cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

sd_status_signal <= x"44";
block_success <= '1';

when CMD25_DATA_4_READ_DECIDE =>

CMD_signal <= '1';
state_leds <= "0101";
sd_status_signal <= x"06";

multiblock_en <= '1';

cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

sd_status_signal <= x"44";

when CMD25_DATA_4_RESEND =>

CMD_signal <= '1';
state_leds <= "0101";
sd_status_signal <= x"06";
resend <= '1';

multiblock_en <= '1';

cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

sd_status_signal <= x"44";

when CMD25_INIT_4_RESEND =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_signal <= '0' & '1' & "011001"
& block_write_sd_addr_internal & x"01";
command_load_en <= '1';

state_leds <= "0010";
sd_status_signal <= x"02";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

sd_status_signal <= x"40";

multiblock_en <= '1';

when CMD25_SEND_4_RESEND =>

CMD_signal <= output_command(47);
command_signal <= '0' & '1' & "011001"
& block_write_sd_addr_internal & x"01";

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_send_en <= '1';
crc7_gen_en <= '1';

state_leds <= "0011";
sd_status_signal <= x"03";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"41";
multiblock_en <= '1';
when ERROR =>
  CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
state_leds <= "1111";
  state_val <= x"11";
  sd_status_signal <= x"08";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
when IDLE =>
  CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
state_leds <= "1000";
  state_val <= x"12";
  sd_status_signal <= x"09";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
CMD13(SEND_STATUS)
Card Status is returned.
This is used to check if the card is ready for data before sending another block to the card in a multiblock stream/write.

when CMD13_INIT_MULTI =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_signal <= '0' & '1' & "001101"
  & card_rca_signal & x"000001";
  command_load_en <= '1';
  state_leds <= "0100";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
  multiblock_en <= '1';
  sd_status_signal <= x"45";

when CMD13_SEND_MULTI =>
  CMD_signal <= output_command(47);
  command_signal <= '0' & '1' & "001101"
  & card_rca_signal & x"000001";
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_send_en <= '1';
  state_leds <= "0101";
  cmd_write_en <= '1';
  crc7_gen_en <= '1';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
multiblock_en <= '1';
sd_status_signal <= x"46";

when CMD13_READ_MULTI =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  read_r1_response_en <= '1';
  state_leds <= "0110";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
  multiblock_en <= '1';
  sd_status_signal <= x"47";

---CMD13(SEND_STATUS)
---Card Status is returned. This is used to check
---if the card is ready for data before sending another
---block to the card in a multiblock stream/write.
---4 bit path.

when CMD13_INIT_MULTI_4 =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_signal <= '0' & '1' & "001101"
    & card_rca_signal & x"000001";
  command_load_en <= '1';
  state_leds <= "0100";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';
  multiblock_en <= '1';
2948   sd_status_signal <= x"45";
2949
2950
2951
2952   when CMD13_SEND_MULTI_4 =>
2953
2954       CMD_signal <= output_command(47);
2955       command_signal <= '0' & '1' & "001101"
2956       & card_rca_signal & x"000001";
2957       dat0_signal <= '1';
2958       dat1_signal <= '1';
2959       dat2_signal <= '1';
2960       dat3_signal <= '1';
2961       command_send_en <= '1';
2962       state_leds <= "0101";
2963       cmd_write_en <= '1';
2964       crc7_gen_en <= '1';
2965       D0_write_en <= '0';
2966       D1_write_en <= '0';
2967       D2_write_en <= '0';
2968       D3_write_en <= '0';
2969       multiblock_en <= '1';
2970       sd_status_signal <= x"46";
2971
2972   when CMD13_READ_MULTI_4 =>
2973
2974       CMD_signal <= '1';
2975       dat0_signal <= '1';
2976       dat1_signal <= '1';
2977       dat2_signal <= '1';
2978       dat3_signal <= '1';
2979       read_r1_response_en <= '1';
2980       state_leds <= "0110";
2981       cmd_write_en <= '0';
2982       D0_write_en <= '0';
2983       D1_write_en <= '0';
2984       D2_write_en <= '0';
2985       D3_write_en <= '0';
2986       multiblock_en <= '1';
2987       sd_status_signal <= x"47";
2988
2989
2990   ———CMD12(STOP_TRANSMISSION)
2991   ——Stop a multiblock write transmission.
when CMD12_INIT =>

    CMD_signal <= '1';
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
    dat3_signal <= '1';
    command_signal <= '0' & '1' & "001100" & x"0000000001";
    command_load_en <= '1';
    state_leds <= "0100";
    cmd_write_en <= '0';
    D0_write_en <= '0';
    D1_write_en <= '0';
    D2_write_en <= '0';
    D3_write_en <= '0';
    sd_status_signal <= x"48";

when CMD12_SEND =>

    CMD_signal <= output_command(47);
    command_signal <= '0' & '1' & "001100" & x"0000000001";
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
    dat3_signal <= '1';
    command_send_en <= '1';
    state_leds <= "0101";
    cmd_write_en <= '1';
    crc7_gen_en <= '1';
    D0_write_en <= '0';
    D1_write_en <= '0';
    D2_write_en <= '0';
    D3_write_en <= '0';
    sd_status_signal <= x"49";

when CMD12_READ =>

    CMD_signal <= '1';
    dat0_signal <= '1';
    dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"4A";

when CMD12_INIT_ABORT =>
  CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "001100" & x"0000000001";
command_load_en <= '1';
state_leds <= "0100";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"58";
multiblock_en <= '1';

when CMD12_SEND_ABORT =>
  CMD_signal <= output_command(47);
  command_signal <= '0' & '1' & "001100" & x"0000000001";
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "0101";
cmd_write_en <= '1';
crc7_gen_en <= '1';
when CMD12_READ_ABORT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"59";
multiblock_en <= '1';

--- CMD55(APP_CMD)

--- This command is sent before any ACMD.
--- This simply tells the card that an expanded command is coming next.

when CMD55_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "110111" & card_rca_signal & x"000001";
command_load_en <= '1';
state_leds <= "0111";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_SEND =>
  CMD_signal <= output_command(47);
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  command_send_en <= '1';
  state_leds <= "1000";
  crc7_gen_en <= '1';
  cmd_write_en <= '1';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';

when CMD55_READ =>
  CMD_signal <= '1';
  dat0_signal <= '1';
  dat1_signal <= '1';
  dat2_signal <= '1';
  dat3_signal <= '1';
  read_r1_response_en <= '1';
  state_leds <= "1001";
  cmd_write_en <= '0';
  D0_write_en <= '0';
  D1_write_en <= '0';
  D2_write_en <= '0';
  D3_write_en <= '0';

--- ACMD23(SET_WR_BLK_ERASE_COUNT)
--- Used to pre-erase before a write.
when ACMD23_INIT =>
  CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "010111" & x"00" & '0' & 
  std_logic_vector(to_unsigned(num_blocks_to_write,23)) & 
x"01";
command_load_en <= '1';
state_leds <= "0111";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
when ACMD23_SEND =>
  CMD_signal <= output_command(47);
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "1000";
crc7_gen_en <= '1';
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
when ACMD23_READ =>
  CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

read_r1_response_en <= '1';
state_leds <= "1001";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

---------
--ACMD6(SET_BUS_WIDTH)
--Switch to 4 bit mode. Use D1–D3 now.
---------

when ACMD6_INIT =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

--Bit 0 indicates bus width. 0 = 1 bit 1 = 4 bit.
--The rest are stuff bits. Page 66 SD Protocol.
command_signal <= '0' & '1' & "000110"
& x"000000" & "000000" & "10" & x"01";
command_load_en <= '1';
state_leds <= "0111";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when ACMD6_SEND =>

CMD_signal <= output_command(47);
--Bit 0 indicates bus width. 0 = 1 bit 1 = 4 bit.
--The rest are stuff bits. Page 66 SD Protocol.
command_signal <= '0' & '1' & "000110"
& x"000000" & "000000" & "10" & x"01";
3261  dat0_signal <= '1';
3262  dat1_signal <= '1';
3263  dat2_signal <= '1';
3264  dat3_signal <= '1';
3265
3266  command_send_en <= '1';
3267  state_leds <= "1000";
3268  crc7_gen_en <= '1';
3269  cmd_write_en <= '1';
3270  D0_write_en <= '0';
3271  D1_write_en <= '0';
3272  D2_write_en <= '0';
3273  D3_write_en <= '0';
3274
3275  when ACMD6_READ =>
3276
3277
3278  CMD_signal <= '1';
3279  dat0_signal <= '1';
3280  dat1_signal <= '1';
3281  dat2_signal <= '1';
3282  dat3_signal <= '1';
3283
3284  read_r1_response_en <= '1';
3285  state_leds <= "1001";
3286  cmd_write_en <= '0';
3287  D0_write_en <= '0';
3288  D1_write_en <= '0';
3289  D2_write_en <= '0';
3290  D3_write_en <= '0';
3291
3292
3293
3294
3295
3296
3297
3298  ---CMD17(READ_SINGLE_BLOCK) Read a single block.
3299
3300
3301
3302  when CMD17_INIT =>
3303
3304  CMD_signal <= '1';
3305  dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "010001"
& block_read_sd_addr & x"01";

command_load_en <= '1';
state_leds <= "0100";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD17_SEND =>

CMD_signal <= output_command(47);
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "0101";
sd_status_signal <= x"43";
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
crc7_gen_en <= '1';

when CMD17_READ =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD17_READ_DATA =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
state_leds <= "0110";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
block_read_process_en <= '1';

-------------------

-- Different exit paths for CMD55 are below.
-- ACMD's must be preceded with CMD55.
-------------------

when CMD55_INIT_ACMD6 =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "110111"
& card_rca_signal & x"000001";
command_load_en <= '1';
state_leds <= "01111";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_SEND_ACMD6 =>
CMD_signal <= output_command(47);
command_signal <= '0' & '1' & "110111"
   & card_rca_signal & x"000001";

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_send_en <= '1';
state_leds <= "1000";
crc7_gen_en <= '1';
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_READ_ACMD6 =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

read_r1_response_en <= '1';
state_leds <= "1001";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_INIT_ACMD42 =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_signal <= '0' & '1' & "110111"
   & card_rca_signal & x"000001";
command_load_en <= '1';
state_leds <= "0111";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_SEND_ACMD42 =>
CMD_signal <= output_command(47);
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

command_send_en <= '1';
state_leds <= "1000";
crc7_gen_en <= '1';
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_READ_ACMD42 =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';

read_r1_response_en <= '1';
state_leds <= "1001";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_INIT_ACMD13 =>
CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_signal <= '0' & '1' & "110111"
& card_rca_signal & x"000001";
command_load_en <= '1';
state_leds <= "0111";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_SEND_ACMD13 =>
CMD_signal <= output_command(47);

dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
command_send_en <= '1';
state_leds <= "1000";
crc7_gen_en <= '1';
cmd_write_en <= '1';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';

when CMD55_READ_ACMD13 =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "1001";
when CMD13_INIT_TIMEOUT_REC =>

    CMD_signal <= '1';
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
    dat3_signal <= '1';

    --IMPORTANT: The CRC7 is defaulted to x"01".
    --It will be filled by CRC7GEN and cmdsend processes.
    command_signal <= '0' & '1' & "001101"
    & card_rca_signal & x"000001";
    command_load_en <= '1';
    state_leds <= "0100";
    cmd_write_en <= '0';
    D0_write_en <= '0';
    D1_write_en <= '0';
    D2_write_en <= '0';
    D3_write_en <= '0';

when CMD13_SEND_TIMEOUT_REC =>

    CMD_signal <= output_command(47);
    --IMPORTANT: The CRC7 is defaulted to x"01".
    --It will be filled by CRC7GEN and CMD_SEND
    command_signal <= '0' & '1' & "001101"
    & card_rca_signal & x"000001";
    dat0_signal <= '1';
    dat1_signal <= '1';
    dat2_signal <= '1';
    dat3_signal <= '1';
    command_send_en <= '1';
    state_leds <= "0101";
    cmd_write_en <= '1';
    crc7_gen_en <= '1';
    D0_write_en <= '0';
    D1_write_en <= '0';
when CMD13_READ_TIMEOUT_REC =>

CMD_signal <= '1';
dat0_signal <= '1';
dat1_signal <= '1';
dat2_signal <= '1';
dat3_signal <= '1';
read_r1_response_en <= '1';
state_leds <= "0110";
cmd_write_en <= '0';
D0_write_en <= '0';
D1_write_en <= '0';
D2_write_en <= '0';
D3_write_en <= '0';
sd_status_signal <= x"68";

end case;
end process;

-- Delay Process
-- Used upon first coming into microsd_data.

first_delay : process(rst_n, clk) is
begin
  if(rst_n = '0') then
    delay_count <= 0;
    delay_done <= '0';
  elsif rising_edge(clk) then
    if (delay_en = '1') then
      if (delay_count = 8) then
        delay_done <= '1';
      else
        delay_count <= delay_count + 1;
      end if;
    else
      delay_count <= delay_count + 1;
      delay_done <= '0';
    end if;
  end if;
else
    delay_count <= 0;
    delay_done <= '0';
end if;
end if;
end process first_delay;

--- Overview of data flow, crc7 gen, and sampling.
--- Data is moved on the negative edge of the clock so it's ready for
--- sampling on the positive edge of the clock at the SD Card.
--- SD card is strictly
--- rising edge sampling, apart from more advanced ddr modes.
--- The crcs are updated at the same time that data is sampled,
--- on the rising edge of the clock.
--- The final bit of a crc is quick. The data must be sampled on
--- the rising edge, and the crc appended and put on line for
--- sampling on following rising edge. This is how it is currently done.

--- CRC7 process. --- Enabled on ANY CMDXX_SEND state. This
--- process
--- is compact. The crc7 is generated streaming and readied within
--- 1/2 clock cycle for append to streaming data.

crc7_bitval_signal <= output_command(47);
crc7_done <= '1' when (crc7_send_bit_count = 40) else '0';
crc7_rst_signal <= '1' when (crc7_gen_en = '1') else '0';
crc7_gen: process(rst_n, clk)
beginn
    if (rst_n = '0') then
        crc7_send_bit_count <= 0;
    elsif falling_edge(clk) then
        if (crc7_gen_en = '1') then
            crc7_send_bit_count <= crc7_send_bit_count + 1;
        else
            crc7_send_bit_count <= 0;
        end if;
    end if;
end process crc7_gen;
---CRC16 process. ---Enabled during data transmit CMD24.DATA/ CMD25.DATA

CRC16_GEN0: process(rst_n, clk)
begin
if (rst_n = '0') then
  crc16_rst_signal_D0 <= '0';
  crc16_send_bit_count_D0 <= 0;
  crc16_done_D0 <= '0';
  crc16_send_byte_count_D0 <= 0;
else
  if falling_edge(clk) then
    if (crc16_gen_en_D0 = '1') then
      if (crc16_send_bit_count_D0 = 7) then
        crc16_send_byte_count_D0 <= crc16_send_byte_count_D0 + 1;
        crc16_send_bit_count_D0 <= 0;
        crc16_rst_signal_D0 <= '1';
        crc16_done_D0 <= '0';
      else
        crc16_rst_signal_D0 <= '1';
        crc16_send_bit_count_D0 <= crc16_send_bit_count_D0 + 1;
        crc16_done_D0 <= '0';
      end if;
    else
      crc16_rst_signal_D0 <= '0';
      crc16_send_bit_count_D0 <= 0;
    end if;
  else
    if (crc16_send_bit_count_D0 = 7) then
      if (crc16_send_byte_count_D0 = 511) then --Assert done
        crc16_done_D0 <= '1';
        crc16_send_bit_count_D0 <= 0;
        crc16_send_byte_count_D0 <= 0;
      end if;
    end if;
  end if;
end if;
end process CRC16_GEN0;
---CRC16 process. ---Enabled during data transmit CMD24_DATA/ CMD25_DATA

CRC16_GEN1: process(rst_n, clk)
begin
  if (rst_n = '0') then
    crc16_done_D1 <= '0';
    crc16_rst_signal_D1 <= '0';
    crc16_send_bit_count_D1 <= 0;
    crc16_send_byte_count_D1 <= 0;
  elsif falling_edge(clk) then
    if (crc16_gen_en_D1 = '1') then
      if (crc16_send_bit_count_D1 = 7) then
        crc16_send_byte_count_D1 <= crc16_send_byte_count_D1 + 1;
        crc16_rst_signal_D1 <= 0;
        crc16_done_D1 <= '0';
      else
        crc16_rst_signal_D1 <= '1';
        crc16_send_bit_count_D1 <= crc16_send_bit_count_D1 + 1;
        crc16_done_D1 <= '0';
      end if;
    else
      crc16_rst_signal_D1 <= '0';
      crc16_send_bit_count_D1 <= 0;
    end if;
  else ——rising edge of clock
  if (crc16_send_bit_count_D1 = 7) then
    if (crc16_send_byte_count_D1 = 511) then ——Assert done
      crc16_done_D1 <= '1';
      crc16_send_bit_count_D1 <= 0;
      crc16_send_byte_count_D1 <= 0;
    end if;
  end if;
end process CRC16_GEN1;

---CRC16 process. ---Enabled during data transmit CMD24_DATA/ CMD25_DATA
CRC16_GEN2: process(rst_n, clk)
begin
if (rst_n = '0') then
  crc16_done_D2 <= '0';
crc16_rst_signal_D2 <= '0';
crc16_send_bit_count_D2 <= 0;
crc16_send_byte_count_D2 <= 0;
elsif falling_edge(clk) then
  if (crc16_gen_en_D2 = '1') then
    if (crc16_send_bit_count_D2 = 7) then
      crc16_send_byte_count_D2 <= crc16_send_byte_count_D2 + 1;
crc16_send_bit_count_D2 <= 0;
crc16_rst_signal_D2 <= '1';
crc16_done_D2 <= '0';
    else
      crc16_rst_signal_D2 <= '1';
crc16_send_bit_count_D2 <= crc16_send_bit_count_D2 + 1;
crc16_done_D2 <= '0';
    end if;
  else
    crc16_rst_signal_D2 <= '0';
crc16_send_bit_count_D2 <= 0;
  end if;
else  —— rising edge of clock
  if (crc16_send_bit_count_D2 = 7) then
    if (crc16_send_byte_count_D2 = 511) then  —— Assert done
      crc16_done_D2 <= '1';
crc16_send_bit_count_D2 <= 0;
crc16_send_byte_count_D2 <= 0;
    end if;
  end if;
end if;
end process CRC16_GEN2;

—–CRC16 process. —–Enabled during data transmit CMD24_DATA/CMD25_DATA
CRC16_GEN3: process(rst_n, clk)
begin
if (rst_n = '0') then
  crc16_done_D3 <= '0';
crc16_rst_signal_D3 <= '0';
crc16_send_bit_count_D3 <= 0;
```vhdl
    crc16_send_byte_count_D3 <= 0;
    elsif falling_edge(clk) then
        if (crc16_gen_en_D3 = '1') then
            if (crc16_send_bit_count_D3 = 7) then
                crc16_send_byte_count_D3 <= crc16_send_byte_count_D3 + 1;
                crc16_send_bit_count_D3 <= 0;
                crc16_rst_signal_D3 <= '1';
                crc16_done_D3 <= '0';
            else
                crc16_rst_signal_D3 <= '1';
                crc16_send_bit_count_D3 <= crc16_send_bit_count_D3 + 1;
                crc16_done_D3 <= '0';
            end if;
        else
            crc16_rst_signal_D3 <= '0';
            crc16_send_bit_count_D3 <= 0;
        end if;
    else
        --rising edge of clock
        if (crc16_send_bit_count_D3 = 7) then
            if (crc16_send_byte_count_D3 = 511) then
                --Assert done
                crc16_done_D3 <= '1';
                crc16_send_bit_count_D3 <= 0;
                crc16_send_byte_count_D3 <= 0;
            end if;
        end if;
    end if;
end process CRC16_GEN3;

--Process used and engaged to sent any command to the --SD card.

cmd_send: process(rst_n, clk)
begin
    if (rst_n = '0') then
        command_send_done <= '0';
        command_send_bit_count <= 0;
        output_command <= x"FFFFFFFFFFFFFFF";
```
cmdstartbit <= '0';
cmd_error_rate <= to_unsigned(0, cmd_error_rate'length);
elsif falling_edge(clk) then
  if (command_load_en = '1') then
    cmdstartbit <= '1';
  end if;
  command_send_done <= '0';
  if (command_send_en = '1') then
    if (command_send_bit_count = 48) then
      command_send_done <= '1';
      command_send_bit_count <= 0;
    elsif (crc7_done = '1') then
      —Below is a framework for introducing crc errors into
      —specific commands. This is taken out for now. Not
      —all cmds
      —have recovery next state logic.
      —if (current_state = CMD7_SEND) then
      —if (cmd_error_rate = 0) then
      —output_command <= "1010101" & '1' & output_command
       (39 downto 0);
      —command_send_bit_count <= command_send_bit_count +
       1;
      —cmd_error_rate <= cmd_error_rate + 1;
      —else
      output_command <= crc7_signal & '1' & output_command
       (39 downto 0);
      command_send_bit_count <= command_send_bit_count + 1;
      —cmd_error_rate <= cmd_error_rate + 1;
      —else
      —output_command <= crc7_signal & '1' & output_command
       (39 downto 0);
      —command_send_bit_count <= command_send_bit_count +
       1;
      —end if;
    end if;
  elsif (cmdstartbit = '1') then
    output_command <= command_signal;
    cmdstartbit <= '0';
  else
    command_send_bit_count <= command_send_bit_count + 1;
    output_command <= output_command(46 downto 0) & '1';
  end if;
else
  output_command <= x"FFFFFFFFFFFFFF";
end if;
end if;
end process cmd_send;

--Below are incoming response handlers which sample the cmd lines
--for response sent after commands.

read_r1_response: process(rst_n, clk) begin
if (rst_n = '0') then
read_r1_response_done <= '0';
response_1_status <= (others => '1');
response_1_current_state_bits <= (others => '1');
r1_response_bytes <= x"FFFFFFFFFFFFFF";
read_bytes <= x"FFFFFFFFFFFFFF";
elif rising_edge(clk) then
read_r1_response_done <= '0';
if (read_r1_response_en = '1') then
if (read_bytes(47) = '0') then
read_r1_response_done <= '1';
r1_response_bytes <= read_bytes;  --Debug register to look at card status field of r1 response
response_1_status <= read_bytes(39 downto 8);  --A debug register to look at
response_1_current_state_bits <= read_bytes(20 downto 17);
read_bytes <= x"FFFFFFFFFFFFFF";
else
read_bytes <= read_bytes(46 downto 0) &
cmd_signal_in_signal;
end if;
end if;
end if;
end process read_r1_response;

read_r6_response: process(rst_n, clk) begin
if (rst_n = '0') then
read_r6_response_done <= '0';
r6_response_bytes <= x"FFFFFFFFFFFFFF";
read_r6_bytes <= x"FFFFFFFFFFFFFF";
elif rising_edge(clk) then
read_r6_response_done <= '0';
if (read_r6_response_en = '1') then
  if (read_r6_bytes(47) = '0') then
    read_r6_response_done <= '1';
    r6_response_bytes <= read_r6_bytes;
    -- card_rca_signal <= read_r6_bytes(39 downto 24);
    -- RCA is passed from INIT.
    -- response_6_status <= read_r6_bytes(23 downto 8);
    read_r6_bytes <= x"FFFFFFFFFFFF";
  else
    read_r6_bytes <= read_r6_bytes(46 downto 0) &
      cmd_signal_in_signal;
  end if;
end if;
end if;
end process read_r6_response;

--
-- Single Block WRITE and Multi Block Write Card Data
-- This is the block writing process. It handles both single block and
-- multi block writing to the sd card. Supporting processes such as the
-- crc gen processes aid it, but the actual data movement is here.
--
master_block_write: process(rst_n, clk) is
begin
  if (rst_n = '0') then
    wr_block_bit_count <= 0;
    wr_block_byte_count <= 0;
    -- Best to initialized lines and data to F as start bit
    -- is always '0' in communications.
    wr_block_byte_data <= x"FF";
    load <= '1';
    ram_read_address_signal <= "000000000";
    append_crc_4_bit <= '0';
    start_bit <= '0';
    -- falling edge of sclk.
  elsif falling_edge(clk) then
    if (block_write_process_en = '1' or
        block_write_process_en_4 = '1') then
Make sure start bit goes through.
if (start_bit = '0') then
    wr_block_byte_data <= x"00";
    start_bit <= '1';
end if;

Load the first byte. Increment counter accordingly.
elsif (load = '1') then
    wr_block_byte_data <= block_write_data;
    if (resending = '0') then
        ram_read_address_signal <= std_logic_vector(
            unsigned(
                ram_read_address_signal) +
            1);
    end if;
    load <= '0';
else
    if (block_write_process_en = '1') then
        wr_block_bit_count <= wr_block_bit_count + 1;
    else
        wr_block_bit_count <= wr_block_bit_count + 4;
    end if;
    if (block_write_process_en = '1') then
        if (wr_block_byte_count = 511) then
            wr_block_byte_data <= crc16_signal_D0 (15 downto 8);
        end if;
        if (wr_block_byte_count = 512) then
            wr_block_byte_data <= crc16_signal_D0_fin(7 downto 0);
        end if;
        elsif (wr_block_byte_count = 513) then
            wr_block_byte_data <= x"FF";
        end if;
    end if;
end else;
else
    wr_block_byte_data <= block_write_data;  --
    wr_block_byte_count <= wr_block_byte_count + 1;
    wr_block_bit_count <= 1;
    ram_read_address_signal <= std_logic_vector(
        unsigned(
            ram_read_address_signal
        ) + 1);
end if;
else  -- must be in 4 bit mode.

if (wr_block_byte_count = 511) then

    --Append the first 8 bits of the crc16
    append_crc_4_bit <= '1';
    --Grab/Store the entire crc16 at this moment
    crc16_signal_D0_fin <= crc16_signal_D0;
    crc16_signal_D1_fin <= crc16_signal_D1;
    crc16_signal_D2_fin <= crc16_signal_D2;
    crc16_signal_D3_fin <= crc16_signal_D3;

    wr_block_byte_count <= wr_block_byte_count + 1;
    wr_block_bit_count <= 1;

elsif (wr_block_byte_count = 512) then

    --Keep shifting the 16 bits values across the bytes border.
    --Append stop bit onto the crc16.
    crc16_signal_D0_fin <= crc16_signal_D0_fin(14 downto 0) & '1';
    crc16_signal_D1_fin <= crc16_signal_D1_fin(14 downto 0) & '1';
    crc16_signal_D2_fin <= crc16_signal_D2_fin(14 downto 0) & '1';
    crc16_signal_D3_fin <= crc16_signal_D3_fin(14 downto 0) & '1';
    wr_block_byte_count <= wr_block_byte_count + 1;
    wr_block_bit_count <= 1;

elsif (wr_block_byte_count = 513) then

    append_crc_4_bit <= '0';
if (block_write_process_en = '1') then
    wr_block_bit_count <= wr_block_bit_count + 1;
    wr_block_byte_data <= wr_block_byte_data(6 downto 0) & '1';
else --must be in 4 bit mode.
    if (append_crc_4_bit = '1') then
        crc16_signal_D0_fin <= crc16_signal_D0_fin(14 downto 0) & '1';
        crc16_signal_D1_fin <= crc16_signal_D1_fin(14 downto 0) & '1';
        crc16_signal_D2_fin <= crc16_signal_D2_fin(14 downto 0) & '1';
        crc16_signal_D3_fin <= crc16_signal_D3_fin(14 downto 0) & '1';
        wr_block_bit_count <= wr_block_bit_count + 1;
    else
        --4 bit mode will need to increment by 4
        wr_block_bit_count <= wr_block_bit_count + 4;
        --In four bit mode we shift by 4 bits instead of 1.
        wr_block_byte_data <= wr_block_byte_data(3 downto 0) & "1111";
        end if;
    end if;
else
324

```vhdl
    wr_block_bit_count <= 0;
    wr_block_byte_count <= 0;
    wr_block_byte_data <= x"FF";
    load <= '1';
    ram_read_address_signal <= "000000000";
    append_crc_4_bit <= '0';
    start_bit <= '0';
    end if;
    end if;
end process master_block_write;

--
-- Block Write Done Signals
-- Signal that one block has finished writing
-- Multi block is the next process
--
block_write_d: process(rst_n, clk) is
begin
  if (rst_n = '0') then
    block_write_done <= '0';
    num_blocks_written <= 0;
  elsif falling_edge(clk) then
    if (block_write_process_en = '1') then
      -- Account for stop bit.
      if (wr_block_byte_count = 514 and wr_block_bit_count = 1) then
        block_write_done <= '1';
        if (multiblock_en = '1') then
          -- Increment how many blocks have finished.
          -- Important in multiblock write.
          num_blocks_written <= num_blocks_written + 1;
        end if;
      end if;
    elseif(block_write_process_en_4 = '1') then
      if (wr_block_byte_count = 514 and wr_block_bit_count = 4) then
        -- Signal that an entire 512 byte block has finished.
        block_write_done <= '1';
      end if;
    else
      -- We need to keep num_blocks_written around inbetween CMD25,
      -- but reset once we leave multiblock_en zone.
```
if (multiblock_en = '1' and block_success = '1') then
    num_blocks_written <= num_blocks_written + 1;
elsif (multiblock_en = '1') then
    num_blocks_written <= num_blocks_written;
    block_write_done <= '0';
else
    block_write_done <= '0';
    num_blocks_written <= 0;
end if;
end if;
end process block_write_d;

--- Multiblock Write Done
--- Signal that the entire multiblock has finished
--- Done by counting the number of single blocks written
---

multiblock_wr_done: process (rst_n, clk) is
begin
    if (rst_n = '0') then
        multiblock_write_done <= '0';
    elsif rising_edge (clk) then
        if (multiblock_en = '1') then
            if (num_blocks_written = num_blocks_to_write) then
                if (block_success = '1') then
                    multiblock_write_done <= '1';
                end if;
            else
                multiblock_write_done <= '0';
            end if;
        else
            multiblock_write_done <= '0';
        end if;
    end if;
end process multiblock_wr_done;

--- Block Read Card Data
--- Process responsible for sampling data
--- line when reading from card
---
--- Shift Register and Start Byte Sync

read_shifter: process(rst_n, clk) is
begin

if (rst_n = '0') then
    block_byte_data_signal <= x"FF";
    block_bit_count <= 0;
    block_start_flag <= '0';
    start_read_bit <= '0';
elsif rising_edge(clk) then
    if (block_read_process_en = '1') then
        if (block_start_flag = '1') then
            --Sample the D0 line
            --Shift the current value in but only if start bit has passed.
            block_byte_data_signal <= block_byte_data_signal(6 downto 0)
                & D0_signal_in_signal;
            if (block_bit_count = 7) then
                block_bit_count <= 0;
            else
                --Sync the start bit. Sync bit 0 to block_bit_count 0.
                --One off mechanism for the very first bit of a block read.
                if (start_read_bit = '1') then
                    block_bit_count <= block_bit_count + 1;
                else
                    start_read_bit <= '1';
                end if;
            end if;
        else
            if (D0_signal_in_signal = '0') then
                block_start_flag <= '1';
                block_bit_count <= 0;
            end if;
        end if;
    else
        block_byte_data_signal <= x"FF";
        block_bit_count <= 0;
        block_start_flag <= '0';
        start_read_bit <= '0';
    end if;
else
    block_byte_data_signal <= x"FF";
    block_bit_count <= 0;
    block_start_flag <= '0';
    start_read_bit <= '0';
end if;
end if;
end process read_shifter;

-- Memory Write Enable Signal Generation
-- Byte Counter, and
-- Address Counter for Single Block Read.
--
read_lines_handler: process(rst_n, clk) is
begin
if (rst_n = '0') then
    block_byte_wren_signal <= '0';
    -- Rather than do another once off flag. I'll roll over on first increment
    -- to byte 0 address. Otherwise byte 0 is written to address 1.
    ram_write_address_signal <= (others => '1');
    block_byte_count <= "0000000000";
elsif falling_edge(clk) then
    -- If we've enabled the read process
    if (block_read_process_en = '1') then
        -- If we've gotten past the start bit 0.
        if (block_start_flag = '1') then
            -- If at bit 7 of transmit.
            if (block_bit_count = 7) then
                -- DO NOT WRITE THE CRCs to ram. Byte 512 and 513 are
                -- CRCs.
                if (block_byte_count = 512) then
                    block_byte_wren_signal <= '0';
                    -- DO NOT WRITE THE CRCs to ram. Byte 512 and 513
                    -- are CRCs.
                elsif (block_byte_count = 513) then
                    block_byte_wren_signal <= '0';
                else
                    -- Else enable wr_en on a ram.
                    block_byte_wren_signal <= '1';
                end if;
            end if;
        end if;
    end if;
    -- Copy data to the output register.
    block_byte_data_signal_out <= block_byte_data_signal;
    -- Increment Internal bytes read counter. Up to 512 and
    -- 513 for CRC.
    block_byte_count <= block_byte_count + 1;
    -- Increment Ram write address. Up to 511.
Starts at 511 and rolls over at beginning byte 0.

```vhdl
ram_write_address_signal <= std_logic_vector(unsigned(
    ram_write_address_signal) + 1);
else
    block_byte_wren_signal <= '0';
end if;
else
    block_byte_wren_signal <= '0';
end if;
else
    block_byte_wren_signal <= '0';
    -- Rather than do another once off flag . . . . .
    -- I'll roll over on first increment to byte 0 address.
    -- Otherwise byte 0 is written to address 1.
    ram_write_address_signal <= (others => '1');
    block_byte_count <= "0000000000";
end if;
end if;
end process read_lines_handler;

-- Block Read Done Generation Process
--
block_read_done: process(rst_n, clk) is
begin
    if (rst_n = '0') then
        CMD6_D0_read_done <= '0';
singleblock_read_done <= '0';
elsif(rising_edge(clk)) then
    if (block_read_process_en = '1') then
        if (block_byte_count = 66) then
            CMD6_D0_read_done <= '1';
singleblock_read_done <= '0';
            -- CMD 17 read done flag.
        elsif (block_byte_count = 514) then
            singleblock_read_done <= '1';
            CMD6_D0_read_done <= '0';
        else
            CMD6_D0_read_done <= '0';
singleblock_read_done <= '0';
        end if;
    end if;
```
else
    CMD6_D0_read_done <= '0';
    singleblock_read_done <= '0';
end if;
end if;
end process block_read_done;

−−
−− Track wide mode (4 bit) switch complete −−
−−
wide_mode_switch_done: process(rst_n, clk)
begin
    if (rst_n = '0') then
        widedone <= '0';
    elsif rising_edge(clk) then
        if (current_state = ACMD6_READ) then
            widedone <= '1';
        end if;
    end if;
end process wide_mode_switch_done;

−−
−− Track Change into HS/SDR25 Mode via CMD6 −−
−−
ac_mode_switch_done_process: process(rst_n, clk)
begin
    if (rst_n = '0') then
        ac_mode_switch_done <= '0';
    elsif rising_edge(clk) then
        if (current_state = CMD6_READ) then
            ac_mode_switch_done <= '1';
        end if;
    end if;
end process ac_mode_switch_done_process;

−−
−− Data Response Token Handler −−
−− After sending write data to the sd card −−
−− via CMD25, a response comes back immediately −−
−− on the D0 line. This response is a Data Response −−
−− token indicating data accepted or not −−
−− This is not immediately evident in the SD protocol −−
−− and the formation of the response is under the SPI −−
-- section of the manual. PAGE 132 of 3.01 spec. --

READ_DATA_TOKEN_RESPONSE: process(rst_n, clk)
begin
if (rst_n = '0') then
    reading_data_token_byte <= x"FF";
    read_data_token_response_done <= '0';
    read_data_token_byte <= x"FF";
elsif rising_edge(clk) then
    read_data_token_response_done <= '0';
    if (read_data_token_response_en = '1') then
        if (reading_data_token_byte(4) = '0' and reading_data_token_byte(0) = '1') then
            read_data_token_response_done <= '1';
            reading_data_token_byte <= reading_data_token_byte(6 downto 0) & d0_signal_in_signal;
        end if;
        else
            reading_data_token_byte <= x"FF";
        end if;
    end if;
end process READ_DATA_TOKEN_RESPONSE;

--
--Cmd response timeout process and explanation.

--If a response is not received in ~100ms, resend the command.
--If total error count becomes too high, pipe out restart signal
--The resend bit is checked in the next state logic. Not all response have this check but most do. The CMD25_4 pathways of importance
--have the check currently.
The below process does indeed work. However after a CRC command timeout

--- a cmd 13 must be sent to clear the status register bits before the card

--- will accept the error'd command again. A cmd_resend_en can be added

--- to next state logic to enter CMD13_INIT_TIMEOUT_REC along with an

--- appropriate setting of return_state to return to. The only command this was
tested on was a resend of CMD7 but the path does indeed work.

--- needs to be added to all commands where simply resending the command again

--- will be the appropriate action. APPCMDS and commands sent in and around

--- data will probably need more care.

--- APPCMDS might be handled by resending CMD55 if that errors and resenting

--- CMD55 if the APPCMD itself fails.

cmd_response_timeout_handler: process(rst_n, clk)
begin

if (rst_n = '0') then

  cmd_response_timeout <= 0;
  cmd_resend_en <= '0';
  error_count <= (others => '0');
  restart_response <= '0';
  cmd_resend_timer_en <= '0';

elsif rising_edge(clk) then

  if (command_send_done = '1') then
    cmd_resend_timer_en <= '1';
    elsif (command_load_en = '1') then
    cmd_resend_timer_en <= '0';
    end if;

  if (cmd_resend_timer_en = '1') then
    --100ms timeout
    if (cmd_response_timeout = cmd_timeout) then
      cmd_resend_en <= '1';
      cmd_resend_timer_en <= '0';
    end if;

end if;
if (error_count = to_unsigned(63,error_count'length))
then
    restart_response <= '1';
end if;
else
    cmd_response_timeout <= cmd_response_timeout + 1;
end if;
else
    cmd_resend_en <= '0';
    cmd_response_timeout <= 0;
end if;
end if;
end process cmd_response_timeout_handler;

-- Data Block Write CRC Error Handling
--
-- Track the number of data resends due to crc errors.
-- Below is the data crc error process. In it we set the
-- flag to alter state machine pathways following a block crc
-- failure.
-- We also increment the data_error_rate to introduce errors
-- into the
-- written data at interval.
-- To disable the error insertion, simply comment
-- data_error_rates out in
-- this process.
-- CRC errors in a multiblock stream are handled by sending a
-- CMD12 followed
-- by an alternative CMD25 with the address of the last block
-- which caused
-- the failure. The data crc error handling is local to
-- microsd_data. The above
-- processes which involve buffers are immune to block resends
-- at this level.
-- This is a very good thing.
resend_count_tracker : process(rst_n, clk)
begin
    if (rst_n = '0') then
        resend_f <= '0';
        resend_count <= (others => '0');
        resending <= '0';
    end if;
end process resend_count_tracker;
elsif rising_edge(clk) then
  if (resend_f /= resend) then
    resend_f <= resend;
    if (resend = '1') then
      resending <= '1';
      data_error_rate <= data_error_rate + 1;
      if (resend_count = to_unsigned(63, resend_count'))
        restart_crc <= '1';
      end if;
    end if;
  end if;
end if;
end process resend_count_tracker;

-- Debug Process to Characterize Inner Multiblock Write Waits.
inner_multiblock_wait: process(clk)
begin
  if (rst_n = '0') then
    -- Async reset of this debug process causes problems in
    the build. Leaving out for now.
    -- cmd13multi_counter_done_1 <= '0';
    -- cmd13multi_counter_done <= '0';
    -- cmd13multi_counter <= to_unsigned(0,32);
    -- cmd12_13_counter <= to_unsigned(0,32);
    -- cmd25_setup_counter <= to_unsigned(0,32);
    -- cmd12_13_counter_reg <= to_unsigned(0,32);
    -- cmd13multi_counter_reg <= to_unsigned(0,32);
    -- cmd25_setup_counter_reg <= to_unsigned(0,32);
    -- CMD25_number_1 <= '0';
    if rising_edge(clk) then
      if (init_done = '1') then
        if (current_state = CMD13_INIT_MULTI_4
            OR current_state = CMD13_READ_MULTI_4
            OR current_state = CMD13_SEND_MULTI_4) then
          --
        end if;
      end if;
    end if;
  end if;
end process inner_multiblock_wait;
cmd13multi_counter <= cmd13multi_counter + 1;
cmd13multi_counter_reg <= cmd13multi_counter;
cmd13multi_counter_done_1 <= '1';

elsif (current_state = CMD13_INIT
  OR current_state = CMD13_READ
  OR current_state = CMD13_SEND
  OR current_state = CMD12_INIT
  OR current_state = CMD12_SEND
  OR current_state = CMD12_READ) then
  cmd12_13_counter <= cmd12_13_counter + 1;
cmd12_13_counter_reg <= cmd12_13_counter;
cmd13multi_counter_done_1 <= '1';

  --Note: This will not include pre_app_wait stuff, but this is
  --more realistic for a post init based delays study.
elseif (current_state /= APP_WAIT) then
  --This will include all states that
  --are not APP_WAIT/CMD12/CMD13/CMD13Multi.
  --Includes everything not CMD12/CMD13 related including
  --point 1 which is run up to app_wait.
cmd25_setup_counter <= cmd25_setup_counter + 1;
cmd25_setup_counter_reg <= cmd25_setup_counter;
cmd13multi_counter_done_1 <= '1';

else--Current state must be app_wait.

if(cmd13multi_counter_done_1 = '1') then
  cmd13multi_counter_done_1 <= '0';
cmd13multi_counter_done <= '1';
  CMD25_number_1 <= '1';
else
  -- Increase the CMD25 counter AFTER we have
  -- stored the previous values.
  if (CMD25_number_1 = '1') then
    CMD25_number <= CMD25_number + 1;
    CMD25_number_1 <= '0';
  end if;
  cmd13multi_counter_done_1 <= '0';
cmd13multi_counter_done <= '0';
cmd13multi_counter <= to_unsigned(0,32);
cmd12_13_counter <= to_unsigned(0,32);
cmd25_setup_counter <= to_unsigned(0,32);
end if;
end if;
end if;
−−
end if;
−−
end if;
end if;
end process inner_multiblock_wait;

−−
−−
data_current_block_written
−− and
−− sd_block_written_flag
−− generation.
−− Generate the last successful address written global
−− output as well as the valid pulse which accompanies it.
−−

address_success : process (rst_n, clk)
begin
if (rst_n = '0') then
first_block_of_multiblock <= '0';
prev_block_write_sd_addr_pulse <= '0';
block_write_sd_addr_internal <= (others => '0');
elseif rising_edge(clk) then
-- Use a follower to do the following only once per change.
if (block_success_follower /= block_success) then
block_success_follower <= block_success;
-- We have passed the data token check successfully
-- based on hitting CRC Success States. Create pulse
-- and increment block count.
if(block_success = '1') then
prev_block_write_sd_addr_pulse <= '1';
else
prev_block_write_sd_addr_pulse <= '0';
end if;
end if;
if (new_block_write_follower /= new_block_write) then
  new_block_write_follower <= new_block_write;
end if;

-- Handling of the block_write_sd_addr_internal is delicate around
-- resending of blocks. However if a resend events doesn’t happen on the
-- very first block of the multiblock this process will not be effected.
if (new_block_write = '1' and resending /= '1') then
  -- Here we only increment the success address if the first block
  -- of a multiblock has passed. This allows for address 0 to be passed out.
  if (first_block_of_multiblock = '1') then
    block_write_sd_addr_interal <= block_write_sd_addr;
    first_block_of_multiblock <= '0';
  else
    block_write_sd_addr_interal <= std_logic_vector(unsigned(block_write_sd_addr_interal) + 1);
  end if;
end if;

end if;

-- If we have returned to APP_WAIT reset the first block flag.
if (SD_status_signal = x"01") then
  first_block_of_multiblock <= '1';
end if;

end process address_success;

end Behavioral;
— CRC Generation Unit – Linear Feedback Shift Register implementation
— (c) Kay Gorontzi, GHSi.de, distributed under the terms of LGPL
—
— Simple port from Verilog to VHDL
— by Christopher Casebeer from http://ghsi.de/CRC/index.php?
  Polynom=10001001&Message=48000001AA

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity microsd_crc_7 is
  port(
    bitval :in std_logic;
    enable :in std_logic;
    clk :in std_logic;
    rst :in std_logic;
    crc_out :out std_logic_vector(6 downto 0)
  );
end microsd_crc_7;

architecture Behavioral of microsd_crc_7 is

signal crc : std_logic_vector(6 downto 0);
signal inv : std_logic;
signal sclk_follower : std_logic;

begin

crc_out <= crc;
inv <= BITVAL XOR CRC(6);

process(clk)
begin
if(rising_edge(clk)) then
if (rst = '0') then
   CRC <= "00000000";
else if (enable = '1') then
   CRC(6) <= CRC(5);
   CRC(5) <= CRC(4);
   CRC(4) <= CRC(3);
   CRC(3) <= CRC(2) XOR inv;
   CRC(2) <= CRC(1);
   CRC(1) <= CRC(0);
   CRC(0) <= inv;
end if;
end if;
end process;
end Behavioral;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

--brief crc16 generation for appending to any sd data sent over the data lines
--details
--param clk Component Clock
--param bitval The current bit input into crc engine
--param enable Enable bit for the engine.
--param rst Active low reset. Zeroes the crc.
--param crc_out The current crc engine output

entity microsd_crc_16 is
port(
  bitval :in std_logic;
  enable :in std_logic;
  clk :in std_logic;
  rst :in std_logic;
  crc_out :out std_logic_vector(15 downto 0)
);
end microsd_crc_16;

architecture Behavioral of microsd_crc_16 is
```vhdl
signal crc : std_logic_vector(15 downto 0);
signal inv : std_logic;

begin

crc_out <= crc;
inv <= BITVAL XOR CRC(15);

process(clk)
begin
if(rising_edge(clk)) then
 if (rst = '0') then
   CRC <= x"0000";
 elsif (enable = '1') then
   CRC(15) <= CRC(14);
   CRC(14) <= CRC(13);
   CRC(13) <= CRC(12);
   CRC(12) <= CRC(11) XOR inv;
   CRC(11) <= CRC(10);
   CRC(10) <= CRC(9);
   CRC(9) <= CRC(8);
   CRC(8) <= CRC(7);
   CRC(7) <= CRC(6);
   CRC(6) <= CRC(5);
   CRC(5) <= CRC(4) XOR inv;
   CRC(4) <= CRC(3);
   CRC(3) <= CRC(2);
   CRC(2) <= CRC(1);
   CRC(1) <= CRC(0);
   CRC(0) <= inv;
 end if;
end if;
end process;

end Behavioral;
```
Filename: microsd_buffer.vhd
Description: Source code for microsd serial data logger
Author: Christopher Casebeer
Lab: Dr. Snider
Department: Electrical and Computer Engineering
Institution: Montana State University
Support: This work was supported under NSF award No. DBI-1254309
Creation Date: June 2014
Version 1.0

Modification History (give date, author, description)
None

Please send bug reports and enhancement requests to
Dr. Snider at rksnider@ece.montana.edu

This software is released under
The MIT License (MIT)

Copyright (C) 2014 Christopher C. Casebeer and Ross K. Snider

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Christopher Casebeer
---
Electrical and Computer Engineering
Montana State University
541 Cobleigh Hall
Bozeman, MT 59717
christopher.casebeer@msu.montana.edu

Ross K. Snider
---
Associate Professor
Electrical and Computer Engineering
Montana State University
538 Cobleigh Hall
Bozeman, MT 59717
rksnider@ece.montana.edu

Information on the MIT license can be found at http://opensource.org/licenses/MIT

! @brief Data buffer of the microsd_controller
! @details The host puts data here while the card is busy writing data.
--- @param buf_size_g Number of bytes in the double buffer.
--- Must be specified as N * 512 bytes.
--- @param block_size_g Size of a sd card block. Minimal addressable data size.
--- @param clk Data Transmission Clock. Clock can be from 400kHz to 100MHz depending on timing of target device.
--- @param rst_n Reset to initial conditions.
--- @param mem_address microsd_data's address into the buffer.
---
--- @param data_out Data presented a byte at a time out of the buffer.
--- @param mem_output Data presented a byte at a time into the buffer.
--- @param mem_clk Data is clocked into the circular buffer on the rising edge.
---
--- @param sd_write_rdy Go bit for the sd card to start writing.
--- @param buf_ful Buffer is full. Signal to the host.
---
--- @param sd_write_done N blocks has been written. Reset the buffer to accept more data.
---
--- @param buffer_reinit_done Signal to microsd_controller that buffer reinit is done.
---
--- @param data_nbblocks N blocks that host will send to card.
--- @param sd_block_written Block received successfully by the sd card.
---
--- @param init_start Init start pushbutton or poweron.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_STD.ALL;
use IEEE.MATH_REAL.ALL;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

entity microsd_buffer is
  generic(
    buf_size_g : natural := 2048;
    block_size_g : natural := 512
  );
  port(
    clk : in std_logic;
    rst_n : in std_logic;
    mem_address : in std_logic_vector (8 downto 0);
    data_out : out std_logic_vector (7 downto 0);
    data_input : in std_logic_vector (7 downto 0);
    data_clk : in std_logic;
    data_we : in std_logic;
    data_full : out std_logic;
    sd_write_rdy : out std_logic;
    sd_write_done : in std_logic;
    buffer_reinit_done : out std_logic;
    data_nblocks : in std_logic_vector (31 downto 0);
    sd_block_written : in std_logic;
    buffer_level : out std_logic_vector (natural(trunc(log2(real(buf_size_g/block_size_g)))) downto 0);
    init_start : in std_logic
  );
end microsd_buffer;
--microsd_buffer takes data from the host on rising edge of mem_clk.
--It store into a 2 port ram. Upon the first block written into the buffer,
--sd_data is signalled to begin writing.
--microsd_buffer keeps track of the number of blocks which have been written to buffer.
--When data_nblocks have have been written, the component will raise buf_full and signal
--the host to stop writing data.

architecture Behavioral of microsd_buffer is

--The buffer is made up of blocks (512 bytes) and bytes in those blocks.
--The buf_size_g generic is a multiple of blocks ie 2048 = 4 * 512.
--In the case of buf_size_g = 2048, 4 addressable blocks exist.

--constant block_size_g : natural := 512;
constant max_level : natural := buf_size_g/block_size_g;

--Data level in the buffer.
signal level : unsigned (natural(trunc(log2(real(buf_size_g/block_size_g))))) downto 0);

signal ram_byte_address_wr : unsigned (natural(trunc(
    log2(real(block_size_g-1)))) downto 0);
signal ram_block_address_wr : unsigned (natural(trunc(
    log2(real(buf_size_g/block_size_g-1)))) downto 0);
signal ram_byte_address_rd : unsigned (natural(trunc(
    log2(real(block_size_g-1)))) downto 0);
signal ram_block_address_rd : unsigned (natural(trunc(
    log2(real(buf_size_g/block_size_g-1)))) downto 0);
signal buf_ful_internal : std_logic;

-- The buffer has at least one block in it. Signal microsd_controller to start write.
signal sd_write_rdy_internal : std_logic;

signal buffer_reinit_done_internal : std_logic;

signal ram_byte_address_wr_follower : unsigned (natural(
  trunc(log2(real(
    trunc(block_size_g - 1))))
) downto 0);

-- Keep track of number of blocks pushed into buffer.
-- When N blocks have been pushed, keep BUF_FUL high as to prevent more data coming in until sd is done.
signal num_blocks_through_buffer : unsigned(31 downto 0);

signal init_started : std_logic;

-- Buffer_clock is inverted system clock.
signal buffer_clock : std_logic;

-- Concatenations for the addresses into the memory. Avoid modelsim warnings.
signal address_a_internal : std_logic_vector(natural(  
  trunc(log2(real(buf_size_g - 1))))
) downto 0);

signal address_b_internal : std_logic_vector(natural(  
  trunc(log2(real(buf_size_g - 1))))
) downto 0);

begin
  ram_byte_address_rd <= unsigned(mem_address);
  data_full <= buf_ful_internal;
  sd_write_rdy <= sd_write_rdy_internal;
buffer_reinit_done <= buffer_reinit_done_internal;

buffer_clock <= not clk;

address_a_internal <= std_logic_vector(ram_block_address_wr)
& std_logic_vector(ram_byte_address_wr);

address_b_internal <= std_logic_vector(ram_block_address_rd)
& std_logic_vector(ram_byte_address_rd);

internal_buffer.instant : altsyncram
GENERIC MAP (
address_aclr_b => "NONE",
address_reg_b => "CLOCK1",
clock_enable_input_a => "BYPASS",
clock_enable_input_b => "BYPASS",
clock_enable_output_b => "BYPASS",
intended_device_family => "Cyclone V",
lpm_type => "altsyncram",
numwords_a => buf_size_g,
numwords_b => buf_size_g,
operation_mode => "DUAL_PORT",
outdata_aclr_b => "NONE",
outdata_reg_b => "CLOCK1",
power_up_uninitialized => "FALSE",
widthad_a => address_a_internal'length,
widthad_b => address_b_internal'length,
width_a => 8,
width_b => 8,
width_byteena_a => 1
)
PORT MAP (
address_a => address_a_internal,
clock0 => buffer_clock,
data_a => data_input,
rden_b => '1',
wren_a => data_we,
address_b => address_b_internal,
clock1 => clk,
q_b => data_out
);

--Start the process upon init.start.
startup:process(rst_n,clk)
begin
if (rst_n = '0') then
  init_started <= '0';
elseif rising_edge(clk) then
  if(init_start = '0') then
    init_started <= '1';
  end if;
end if;
end process;

--Read and write pointer handling.
read_handling: process(rst_n, clk)
begin
if (rst_n = '0') then
  level <= to_unsigned(0, level'length);
  num_blocks_through_buffer <= to_unsigned(0, num_blocks_through_buffer'length);
  buffer_reinit_done_internal <= '0';
  ram_byte_address_wr_follower <= to_unsigned(0, ram_byte_address_wr_follower'length);
  ram_block_address_rd <= to_unsigned(0, ram_block_address_rd'length);
  sd_write_rdy_internal <= '0';
elsif rising_edge(clk) then
  if (init_started = '1') then
    if (ram_byte_address_wr_follower /= ram_byte_address_wr) then
      ram_byte_address_wr_follower <= ram_byte_address_wr;
      --Increase the level only once (use of followers) when ram_byte_address_wr is 511.
      --Also increment the num of blocks through buffer.
      if (ram_byte_address_wr = to_unsigned(block_size_g - 1, ram_byte_address_wr'length)) then
        level <= level + 1;
        num_blocks_through_buffer <= num_blocks_through_buffer + 1;
      end if;
    end if;
  end if;
  --If the last block was received by the card successfully.
  if (sd_block_written = '1') then

Do not decrease level until reinit if we’ve written Nblocks.

This will keep buf_full high.

if (num_blocks_through_buffer = unsigned(data_nbblocks)) then
    ram_block_address_rd <= ram_block_address_rd + 1;
    -- Else decrease the level of the buffer.
    -- Move to reading the next block in the buffer.
elsif (ram_byte_address_rd = to_unsigned(0, ram_byte_address_rd’length)) then
    level <= level - 1;
    ram_block_address_rd <= ram_block_address_rd + 1;
end if;
end if;

-- If the last block was received okay, then reinit the buffer
-- level and the blocks count that has flowed through buffer.
if (sd_write_done = '1') then
    buffer_reinit_done_internal <= '1';
    level <= to_unsigned(0, level’length);
    num_blocks_through_buffer <= (others => '0');
else
    buffer_reinit_done_internal <= '0';
end if;
if (num_blocks_through_buffer /= to_unsigned(0, num_blocks_through_buffer’length)) then
    sd_write_rdy_internal <= '1';
else
    sd_write_rdy_internal <= '0';
end if;
end if;
end process read_handling;

-- Write handling is occurring in a different clock domain than the
-- read handling. This has been tested at 3.6Mhz write -> 50Mhz read.
write_handling: process(rst_n, clk)
begin
   if (rst_n = '0') then
      ram_block_address_wr <= to_unsigned(0, ram_block_address_wr'length);
      ram_byte_address_wr <= to_unsigned(0, ram_byte_address_wr'length);
      buf_ful_internal <= '0';
   elsif rising_edge(clk) then
      if (init_started = '1') then
         --Push out buffer level synchronously.
         buffer_level <= std_logic_vector(level);
         if (level /= to_unsigned(max_level, level'length)) then
            if (level = to_unsigned(max_level - 1, level'length)) then
               buf_ful_internal <= '1';
            else
               buf_ful_internal <= '0';
            end if;
         end if;
      end if;
      if (data_we = '1') then
         if (level /= to_unsigned(max_level, level'length)) then
            ram_byte_address_wr <= ram_byte_address_wr + 1;
            if (ram_byte_address_wr = to_unsigned(block_size_g - 1, ram_byte_address_wr'length)) then
               ram_block_address_wr <= ram_block_address_wr + 1;
            end if;
         end if;
      end if;
   end if;
end process;
end Behavioral;
This program is free software: you can redistribute it and
/or modify
it under the terms of the GNU General Public License as
published by
the Free Software Foundation, either version 3 of the
License, or
(at your option) any later version.
This program is distributed in the hope that it will be
useful,
but WITHOUT ANY WARRANTY; without even the implied
warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See
the
GNU General Public License for more details.
You should have received a copy of the GNU General Public
License
along with this program. If not, see <http://www.gnu.org/
licenses/>.
Chris Casebeer
Electrical and Computer Engineering
Montana State University
610 Cobleigh Hall
Bozeman, MT 59717
chrisopher.casebee1@msu.montana.edu
![image]

---! @brief Initialize and Control the ST Microelectronics LSM9DS1 IMU.
---! @details
---!
---! @param IMU_AXIS_WORD_LENGTH_BYTES The size of one axis of IMU device in bytes
---!
---! @param command_used_g SPI_COMMANDS GENERIC
---! @param address_used_g SPI_COMMANDS GENERIC
---!
---! @param command_width_bytes_g SPI_COMMANDS GENERIC
---! @param address_width_bytes_g SPI_COMMANDS GENERIC
---! @param data_length_bit_width_g SPI_COMMANDS GENERIC
---!
---!
---!
---!
---! @param clk System clock which drives entity.
---! @param rst_n Active Low reset to reset entity
---! @param startup '1' causes state machine to once off push all the IMU over the SPI bus.
---! @param current_fpga_time The current system wide FPGA time. Used for time stamping fetched data.
---!
---! @param gyro_data_rdy Signal to flashblock that new gyro data is ready.
---! @param accel_data_rdy Signal to flashblock that new accel data is ready.
---! @param mag_data_rdy Signal to flashblock that new mag data is ready.
---! @param temp_data_rdy Signal to flashblock that new temp data is ready.
---!
---! @param gyro_data_x 2 byte word for gyro X axis (2’s complement) (Big endian)
---! @param gyro_data_y 2 byte word for gyro Y axis (2’s complement) (Big endian)
---! @param gyro_data_z 2 byte word for gyro Z axis (2’s complement) (Big endian)
67 ‒‒! @param accel_data_x 2 byte word for accel X axis (2’s complement) (Big endian)
68 ‒‒! @param accel_data_y 2 byte word for accel Y axis (2’s complement) (Big endian)
69 ‒‒! @param accel_data_z 2 byte word for accel Z axis (2’s complement) (Big endian)
70 ‒‒! @param mag_data_x 2 byte word for mag X axis (2’s complement) (Big endian)
71 ‒‒! @param mag_data_y 2 byte word for mag Y axis (2’s complement) (Big endian)
72 ‒‒! @param mag_data_z 2 byte word for mag Z axis (2’s complement) (Big endian)
73 ‒‒! @param temp_data 2 byte word for temp X axis (2’s complement) (Big endian)
74 ‒‒!
75 ‒‒! @param sclk SCLK of the SPI interface. Tie to SCL/SPC of LSM9DS1.
76 ‒‒! @param mosi MOSI. Tie to SDA/SDI/SDO of LSM9DS1.
77 ‒‒! @param miso_XL_G MISO of the XL_G SPI interface. Tie to SDO,A/G of LSM9DS1.
78 ‒‒! @param miso_M MISO of the M SPI interface. Tie to SDO,M of LSM9DS1.
79 ‒‒! @param cs_XL_G CS_N of the XL_G SPI interface. Tie to CS,A/G of LSM9DS1.
80 ‒‒! @param cs_M sclk of the SPI interface. Tie to CS,M of LSM9DS1.
81 ‒‒!
82 ‒‒! @param INT_M INT_M pin of the LSM9DS1
83 ‒‒! @param DRDY_M DRDY_M pin of the LSM9DS1
84 ‒‒! @param INT1_A_G INT1_A/G pin of the LSM9DS1
85 ‒‒! @param INT2_A_G INT2_A/G pin of the LSM9DS1
86 ‒‒!
87 ‒‒! @param gyro_fpga_time FPGA time associated with a gyro_data_rdy and its word.
88 ‒‒! @param accel_fpga_time FPGA time associated with a accel_data_rdy and its word.
89 ‒‒! @param mag_fpga_time FPGA time associated with a mag_data_rdy and its word.
90 ‒‒! @param temp_fpga_time FPGA time associated with a temp_data_rdy and its word.
91 ‒‒!
This piece of VHDL code is to be used with the following two MATLAB scripts.

-- LSM9DS1_XL_G_Register_Settings.m
-- LSM9DS1_M_Register_Settings.m

Running these two scripts in the above order will generate an associated mif file
-- LSM9DS1_Register_Settings_Startup_Memory.mif

This file is read on entity startup to send the non-default registers to both the
-- Accelerometer Gyroscope portion of the IMU over its SPI lines. Then the
-- non default registers are sent to the magnetometer. The IMU
-- is split
-- into two separate devices each with its own register map
-- and SPI chip select line.
-- Thus commands go to each device separately. The
-- accelerometer/gyroscope device (XL_G)
-- and the magnetometer device (M) are handled separately.

The startup registers are sent first for the XL_G and then
-- for the M. Both register
-- sets are pulled from the mif file generated by the scripts.

LSM9DS1 makes use of the spi abstraction layer spi_commands
-- This entity allows
-- pushing a command set to a slave device followed by payload
-- bytes.

The spi_commands only allows for one MISO connection and
-- one CS_N connection. However
-- with the IMU there are MISO/CS_N for each device. A
-- multiplexer selects which
-- line the output of SPI_COMMANDS goes to depending on
-- SPI_SELECT_XL_G_M. During
The interrupt pins on the IMU are programmable. The state machine has been programmed to recognize INT1_A_G as gyroscope data ready and INT2_A_G as accelerometer data rdy.

The DRDY_M is negative logic by default. This requires an inversion on the upper port map of this entity. INT1_A_G and INT2_A_G are positive logic by default. They must be enabled on startup and may also be programmed to negative logic, but are not currently.

-- A dual port ram is used.
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

LIBRARY GENERAL;
USE GENERAL.UTILITIES_PKG.ALL; —— Use General Purpose Libraries
USE GENERAL.GPS_Clock_pkg.ALL; —— Use GPS Clock information.

entity LSM9DS1_top is
Generic (IMU_AXIS Word_LENGTH_BYTES : natural := 2;
command_used_g : std_logic := '1';
address_used_g : std_logic := '0';
command_width_bytes_g : natural := 1;
address_width_bytes_g : natural := 1;
data_length_bit_width_g : natural := 10
)

Port ( clk : in std_logic ;
        rst_n : in std_logic ;
        startup : in std_logic;
        current_fpga_time : in std_logic_vector ( gps_time_bytes_c*8-1 downto 0);
        gyro_data_rdy : out std_logic;
        accel_data_rdy : out std_logic;
        mag_data_rdy : out std_logic;
        temp_data_rdy : out std_logic;
        gyro_data_x : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        gyro_data_y : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        gyro_data_z : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        accel_data_x : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        accel_data_y : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        accel_data_z : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        mag_data_x : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        mag_data_y : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
        mag_data_z : out std_logic_vector( IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
temp_data : out std_logic_vector (IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
sclk : out std_logic;
mosi : out std_logic;
miso_XL_G : in std_logic;
miso_M : in std_logic;
cs_XL_G : out std_logic;
cs_M : out std_logic;
INT_M : in std_logic;
DRDY_M : in std_logic;
INT1_A_G : in std_logic;
INT2_A_G : in std_logic;

gyro_fpga_time : out std_logic_vector (gps_time_bytes_c*8-1 downto 0);
accel_fpga_time : out std_logic_vector (gps_time_bytes_c*8-1 downto 0);
mag_fpga_time : out std_logic_vector (gps_time_bytes_c*8-1 downto 0);
temp_fpga_time : out std_logic_vector (gps_time_bytes_c*8-1 downto 0)

end entity LSM9DS1_top;

architecture behavior of LSM9DS1_top is

type IMU_STATE is (
    IMU_STATE_WAIT,
    IMU_STATE_INIT.FETCH_XL_G.NUM_SET_UP,
    IMU_STATE_INIT.FETCH_XL_G.NUM,
    IMU_STATE_INIT.FETCH_M.NUM_SET_UP,
    IMU_STATE_INIT.FETCH_M.NUM,
    IMU_STATE_INIT.FETCH_M.M.NUM_SET_UP,
    IMU_STATE_INIT.FETCH_M.M.NUM,
    IMU_STATE_FINAL.M.NUM_SET_UP,
    IMU_STATE_FINAL.M.NUM,
    IMU_STATE_FINAL.M.M.NUM_SET_UP,
    IMU_STATE_FINAL.M.M.NUM,
    INT_M, DRDY_M, INT1_A_G, INT2_A_G
);

end architecture behavior of LSM9DS1_top;
IMU_STATE_INIT_FETCH_M_NUM,

IMU_STATE_INIT_FETCH_XL_G,
IMU_STATE_INIT_WRITE_XL_G,
IMU_STATE_INIT_WAIT_XL_G,

IMU_STATE_INIT_FETCH_M,
IMU_STATE_INIT_WRITE_M,
IMU_STATE_INIT_WAIT_M,

IMU_STATE_FETCH_XL_SETUP,
IMU_STATE_FETCH_XL,
IMU_STATE_FETCH_XLDONE,

IMU_STATE_FETCH_G_SETUP,
IMU_STATE_FETCH_G,
IMU_STATE_FETCH_G_DONE,

IMU_STATE_FETCH_M_SETUP,
IMU_STATE_FETCH_M,
IMU_STATE_FETCH_M_DONE

signal cur_imu_state : IMU_STATE;
signal next_imu_state : IMU_STATE;

-- Address where the number of registers to be initialized (number changed from default)
-- for the magnetometer and the accelerometer/gyroscope is stored.
-- 2 8 bit numbers exist at memory locations 0 and 1. These represent the number of
-- non default XLG and M registers which exist in this memory.
constant imu_initbuffer_xl_g_num_loc_c : std_logic_vector(7 downto 0) := x"00";
constant imu_initbuffer_m_num_loc_c : std_logic_vector(7 downto 0) := x"01";
constant imu_initbuffer_data_start_loc_c : std_logic_vector(7 downto 0) := x"02";

--Number of registers associated with a sensor and its data.
--2 bytes for each axis of any single sensor.
constant XL_G_M_register_count_c : natural := 6;
constant IMU_REGISTER_SIZEgetBytes : natural := 1;

--The beginning of the XYZ registers for the IMU devices.
--The accelerometer XYZ registers start as x"28". They go for 6 bytes.
constant XL_data_begin_addr : natural := 40;  --x"28"
constant G_data_begin_addr : natural := 24;  --x"18"
constant M_data_begin_addr : natural := 40;  --x"28"

--First bit of the LSM9DS1 command dictates if a read or a write occurs.
--This is dictated in the device datasheet SPI protocol.
constant RD_EN_BIT : std_logic := '1';
constant WR_EN_BIT : std_logic := '0';

--Magnetometer multiple register read enable bit.
constant MS_AUTOINCREMENT_BIT : std_logic := '1';

--These numbers are stored as 16 bit values in memory.
signal xl_g_init_number : unsigned (7 downto 0);
signal m_init_number : unsigned (7 downto 0);
signal spi_commands_complete : unsigned( 7 downto 0);

--Signals for the one port init buffer.
signal imu_initbuffer_address_std : std_logic_vector(7 downto 0);
signal imu_initbuffer_address : unsigned(7 downto 0);
signal imu_initbuffer_rd_en : std_logic;
signal imu_initbuffer_wr_en : std_logic;
signal imu_initbuffer_q : std_logic_vector(15 downto 0);
signal imu_initbuffer_data : std_logic_vector(15 downto 0);
signal initbuffer_clk : std_logic;
signal gyro_sample : std_logic_vector(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);
signal accel_sample : std_logic_vector(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);
signal mag_sample : std_logic_vector(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);
signal temp_sample : std_logic_vector(IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);

--- Interrupt tracking associated signals
signal INT_M_follower : std_logic;
signal INT_M_req : std_logic;
signal INT_M_processed : std_logic;
signal INT_M_processed_follower : std_logic;

signal INT1_A_G_follower : std_logic;
signal gyro_req : std_logic;
signal gyro_processed : std_logic;
signal gyro_processed_follower : std_logic;

signal INT2_A_G_follower : std_logic;
signal accel_req : std_logic;
signal accel_processed : std_logic;
signal accel_processed_follower : std_logic;

signal DRDY_M_follower : std_logic;
signal DRDY_M_req : std_logic;
signal DRDY_M_processed : std_logic;
signal DRDY_M_processed_follower : std_logic;

--- Intermediate port mapping signals of the spi_commands entity
  These are
signal command_spi_signal : std_logic_vector(command_width_bytes_g*8-1 downto 0);
signal address_spi_signal : std_logic_vector(address_width_bytes_g*8-1 downto 0);
signal address_en_spi_signal : std_logic;
signal data_length_spi_signal : std_logic_vector(data_length_bit_width_g - 1 downto 0);
signal master_slave_data_spi_signal : std_logic_vector(7 downto 0);
signal master_slave_data_rdy_spi_signal : std_logic;
signal master_slave_data_ack_spi_signal : std_logic;
signal master_slave_data_ack_spi_signal_follower : std_logic;
signal command_busy_spi_signal : std_logic;
signal command_done_spi_signal : std_logic;
signal command_done_spi_signal_follower : std_logic;
signal slave_master_data_spi_signal : std_logic_vector(7 downto 0);
signal slave_master_data_ack_spi_signal : std_logic;

 signal startup_en : std_logic;
signal startup_follower : std_logic;

-- Processed signals allow servicing the startup signal in.
signal startup_processed : std_logic;
signal startup_processed_follower : std_logic;

-- Startup complete indicate that main state machine can begin
-- looking for interrupts.
signal startup_complete : std_logic;

-- These signals
signal miso_signal : std_logic;
signal cs_n_signal : std_logic;

-- SPI Routing signals
-- '0' is XL_G, '1' is M
signal SPI_SELECT_XL_G_M : std_logic;

-- Byte counts related to transferring bytes.
signal byte_count : unsigned (data_length_bit_width_g -1 downto 0);
signal byte_number : unsigned (data_length_bit_width_g -1 downto 0);

-- Counts used to keep track of read bytes off MISO.
signal byte_read_count : unsigned (data_length_bit_width_g -1 downto 0);
signal byte_read_number : unsigned (data_length_bit_width_g -1 downto 0);
signal INT1_A_G_sync : std_logic;
signal INT2_A_G_sync : std_logic;
signal DRDY_M_sync : std_logic;

component spi_commands is
generic(
  command_used_g : std_logic := '1';
  address_used_g : std_logic := '0';
  command_width_bytes_g : natural := 1;
  address_width_bytes_g : natural := 1;
  data_length_bit_width_g : natural := 10;
  cpol_cpha : std_logic_vector(1 downto 0) := "00"
);
port(
  clk : in std_logic;
  rst_n : in std_logic;
  command_in : in std_logic_vector(command_width_bytes_g*8-1 downto 0);
  address_in : in std_logic_vector(address_width_bytes_g*8-1 downto 0);
  address_en_in : in std_logic;
  data_length_in : in std_logic_vector(data_length_bit_width_g - 1 downto 0);
  master_slave_data_in : in std_logic_vector(7 downto 0);
  master_slave_data_rdy_in : in std_logic;
  master_slave_data_ack_out : out std_logic;
  command_busy_out : out std_logic;
  command_done : out std_logic;
  slave_master_data_out : out std_logic_vector(7 downto 0);
  slave_master_data_ack_out : out std_logic;
  mosi : in std_logic;
  miso : out std_logic;
  sclk : out std_logic;
  cs_n : out std_logic
);
end component;
begin
imu_initbuffer_address_std <= std_logic_vector(
    imu_initbuffer_address);
initbuffer_clk <= not clk;

spi_commands_slave_XL_G : spi_commands
generic map (  
    command_used_g => command_used_g ,  
    address_used_g => address_used_g ,  
    command_width_bytes_g => command_width_bytes_g ,  
    address_width_bytes_g => address_width_bytes_g ,  
    data_length_bit_width_g => data_length_bit_width_g ,  
    cpol_cpha => "00"
)
port map (  
    clk => clk ,  
    rst_n => rst_n ,  
    command_in => command_spi_signal ,  
    address_in => address_spi_signal ,  
    address_en_in => address_en_spi_signal ,  
    data_length_in => data_length_spi_signal ,  
    master_slave_data_in => master_slave_data_spi_signal ,  
    master_slave_data_rdy_in => master_slave_data_rdy_spi_signal ,  
    master_slave_data_ack_out => master_slave_data_ack_spi_signal ,  
    command_busy_out => command_busy_spi_signal ,  
    command_done => command_done_spi_signal ,  
    slave_master_data_out => slave_master_data_spi_signal ,  
    slave_master_data_ack_out => slave_master_data_ack_spi_signal ,  
    miso => miso_signal ,  
    mosi => mosi ,  
    sclk => sclk ,  
    cs_n => cs_n_signal
);
This memory holds the non-default registers for the IMU. These are set on startup. This memory is initialized with a mif file.

```vhdl
altsyncram_component : altsyncram
GENERIC MAP (  
clock_enable_input_a => "BYPASS",  
clock_enable_output_a => "BYPASS",  
init_file => "LSM9DS1_Register_Settings_Startup_Memory.mif",  
intended_device_family => "Cyclone V",  
lpm_hint => "ENABLE_RUNTIME_MOD=NO",  
lpm_type => "altsyncram",  
numwords_a => 256,  
operation_mode => "SINGLE_PORT",  
outdata_aclr_a => "NONE",  
outdata_reg_a => "UNREGISTERED",  
power_up_uninitialized => "FALSE",  
read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",  
widthad_a => 8,  
width_a => 16,  
width_byteena_a => 1  
)
PORT MAP (  
address_a => imu_initbuffer_address_std,  
clock0 => initbuffer_clk,  
data_a => imu_initbuffer_data,  
wren_a => imu_initbuffer_wr_en,  
rden_a => imu_initbuffer_rd_en,  
q_a => imu_initbuffer_q  
);
```

---

@brief Interact with the LSM9DS1 IMU. Make use of SPI abstractions.

---!

@details Do the following things.

---!

Initialize the registers of the IMU which differ from default.

---!

Respond to XL, G, and M data rdy interrupts.

---!

Receive these data and push them out top of entity.

---!
LSM9DS1_state_machine: process (clk, rst_n)
begin
  if (rst_n = '0') then
    byte_count <= to_unsigned(0, byte_count'length);
    byte_number <= to_unsigned(0, byte_number'length);
    cur_imu_state <= IMU_STATE_WAIT;
    command_spi_signal <= (others => '0');
    master_slave_data_spi_signal <= (others => '0');
    address_en_spi_signal <= '0';
    data_length_spi_signal <= (others => '0');
    master_slave_data_rdy_spi_signal <= '0';
    xl_g_init_number <= (others => '0');
    m_init_number <= (others => '0');
    accel_sample <= (others => '0');
    accel_data_x <= (others => '0');
    accel_data_y <= (others => '0');
    accel_data_z <= (others => '0');
    gyro_sample <= (others => '0');
    gyro_data_x <= (others => '0');
    gyro_data_y <= (others => '0');
    gyro_data_z <= (others => '0');
    mag_sample <= (others => '0');
    mag_data_x <= (others => '0');
    mag_data_y <= (others => '0');
    mag_data_z <= (others => '0');
  end if;
end process;
```
byte_read_number <= to_unsigned(0, byte_number'length);

DRDY_M_processed <= '0';
accel_processed <= '0';
gyro_processed <= '0';
startup_processed <= '0';

imu_initbuffer_address <= (others => '0');
startup_complete <= '0';

command_done_spi_signal_follower <= '0';

elsif (clk'event and clk = '1') then
    -- Default signal states.
    master_slave_data_rdy_spi_signal <= '0';

    if (startup_processed = '1' and startup_processed_follower = '1') then
        startup_processed <= '0';
    end if;

    if (gyro_processed = '1' and gyro_processed_follower = '1') then
        gyro_processed <= '0';
    end if;

    if (accel_processed = '1' and accel_processed_follower = '1') then
        accel_processed <= '0';
    end if;

    if (DRDY_M_processed = '1' and DRDY_M_processed_follower = '1') then
        DRDY_M_processed <= '0';
    end if;
```
case cur_imu_state is

  when IMU_STATE_WAIT =>

    if (startup_en = '1') then
      cur_imu_state <= IMU_STATE_INIT_FETCH_XL_G_NUM_SETUP;
    elsif (startup_complete = '1') then
      --INT1_A_G has been set to G data ready.
      if (gyro_req = '1') then
        cur_imu_state <= IMU_STATE_FETCH_G_SETUP;
      --INT2_A_G has been set to XL data ready.
      elsif (accel_req = '1') then
        cur_imu_state <= IMU_STATE_FETCH_XL_SETUP;
      elsif (DRDY_M_req = '1') then
        cur_imu_state <= IMU_STATE_FETCH_M_SETUP;
      end if;
    end if;

  when IMU_STATE_INIT_FETCH_XL_G_NUM_SETUP =>

    imu_initbuffer_address <= unsigned(
      imu_initbuffer_xl_g_num_loc_c);
    cur_imu_state <= IMU_STATE_INIT_FETCH_XL_G_NUM;
    --Startup_processed moved here to allow enough time
    --for startup_en
    --to go low before IMU_STATE_WAIT checks it again.
    startup_processed <= '1';

  when IMU_STATE_INIT_FETCH_XL_G_NUM =>

    byte_count <= to_unsigned(0, byte_count'length);
    byte_number <= resize(unsigned(imu_initbuffer_q),
      byte_number'length);

    --I only read 8 bits of data here from a 16 bit spot. I do this because
    --I use this number in addressing into a 256 spot ram
    --and I do not need
    --bits 15 through 8.
    xl_g_init_number <= unsigned(imu_initbuffer_q(7 downto 0));
    cur_imu_state <= IMU_STATE_INIT_FETCH_XL_G;
imu_initbuffer_address <= unsigned(
imu_initbuffer_data_start_loc_c);

when IMU_STATE_INIT_FETCH_XL_G =>

if(byte_count = byte_number) then
  cur_imu_state <= IMU_STATE_INIT_FETCH_M_NUM_SETUP;
else
  cur_imu_state <= IMU_STATE_INIT_WRITE_XL_G;
end if;

when IMU_STATE_INIT_WRITE_XL_G =>

if (command_busy_spi_signal = '0') then
  command_spi_signal <= WR_EN_BIT & imu_initbuffer_q(14 downto 8);
  master_slave_data_spi_signal <= imu_initbuffer_q(7 downto 0);
  address_en_spi_signal <= '0';
  data_length_spi_signal <= std_logic_vector(to_unsigned(IMU_REGISTER_SIZE_BYTES,
data_length_spi_signal'length));
  imu_initbuffer_address <= imu_initbuffer_address + 1;
  master_slave_data_rdy_spi_signal <= '1';
  cur_imu_state <= IMU_STATE_INIT_WAIT_XL_G;
  byte_count <= byte_count + 1;
end if;

when IMU_STATE_INIT_WAIT_XL_G =>

-- Wait for a register to complete going out before continuing.
-- This is necessary for cs_n multiplexing.
-- This is necessary as no spi_command ack's come back for command only.
-- if (command_done_spi_signal_follower /=
  command_done_spi_signal) then
--  command_done_spi_signal_follower <=
  command_done_spi_signal;
if(command_done_spi_signal = '1') then
  cur_imu_state <= IMU_STATE_INIT_FETCH_XL_G;
end if;

when IMU_STATE_INIT_FETCH_M_NUM_SETUP =>
  imu_initbuffer_address <= unsigned(immu_initbuffer_m_num_loc_c);
  cur_imu_state <= IMU_STATE_INIT_FETCH_M_NUM;

when IMU_STATE_INIT_FETCH_M_NUM =>
  byte_count <= to_unsigned(0,byte_count'length);
  byte_number <= resize(unsigned(imu_initbuffer_q),
                      byte_number'length);
  m_init_number <= unsigned(imu_initbuffer_q(7 downto 0));
  cur_imu_state <= IMU_STATE_INIT_FETCH_M;
  imu_initbuffer_address <= unsigned(imu_initbuffer_data_start_loc_c) + xl_g_init_number ;

when IMU_STATE_INIT_FETCH_M =>
  if(byte_count = byte_number) then
    -- Don't service interrupts till we've set up the registers.
    startup_complete <= '1';
    cur_imu_state <= IMU_STATE_WAIT;
  else
    cur_imu_state <= IMU_STATE_INIT_WRITE_M;
  end if;

when IMU_STATE_INIT_WRITE_M =>
  if(command_busy_spi_signal = '0') then
    command_spi_signal <= WR_EN_BIT & imu_initbuffer_q (14 downto 8);
    master_slave_data_spi_signal <= imu_initbuffer_q(7 downto 0);
    address_en_spi_signal <= '0';
data_length_spi_signal <= std_logic_vector(to_unsigned(IMU_REGISTER_SIZE_BYTES, data_length_spi_signal'length));
imu_initbuffer_address <= imu_initbuffer_address + 1;
master_slave_data_rdy_spi_signal <= '1';
byte_count <= byte_count + 1;
cur_imu_state <= IMU_STATE_INIT_WAIT_M;
end if;

when IMU_STATE_INIT_WAIT_M =>

-- Wait for a register to complete going out before continuing.
-- This is necessary for cs_m multiplexing.
-- This is necessary as no spi_command ack's come back for command only.
-- if (command_done_spi_signal_follower /= command_done_spi_signal) then
  -- command_done_spi_signal_follower <= command_done_spi_signal;
  if (command_done_spi_signal = '1') then
    cur_imu_state <= IMU_STATE_INIT_FETCH_M;
  end if;
-- end if;

when IMU_STATE_FETCH_XL_SETUP =>

byte_read_count <= to_unsigned(0, byte_count'length);
byte_read_number <= to_unsigned(XL_G_M_register_count_c, byte_number'length);
-- Allows the first byte to go to the 0 address.
-- Address is incremented on first read byte.
if (command_busy_spi_signal = '0') then
  command_spi_signal <= RD_EN_BIT & std_logic_vector(to_unsigned(XL_data_begin_addr, command_spi_signal'length - 1));
  address_en_spi_signal <= '0';
data_length_spi_signal <= std_logic_vector(to_unsigned(XL_G_M_register_count_c, data_length_spi_signal'length));
master_slave_data_spi_signal <= x"00";
master_slave_data_rdy_spi_signal <= '1';
cur_imu_state <= IMU_STATE_FETCH_XL;
end if;

-- After pushing a command set and first data byte to the SPI_COMMANDS entity, we then wait for
-- 6 associated MISO bytes to come back to us. We must remember that the slave_master_data_ack_spi_signal
-- only goes high on the data/payload portion of the SPI stream. No ack's come back associated with command
-- or address bytes sent out the SPI bus. We can thus just count 6 ack's and know that we grabbed the 6 XYZ bytes.
-- This is what is happening here.
-- This assumes that the address is auto incrementing.

when IMU_STATE_FETCH_XL =>

if(byte_read_count = byte_read_number) then
  cur_imu_state <= IMU_STATE_FETCH_XL_DONE;
  accel_processed <= '1';
elsif (slave_master_data_ack_spi_signal = '1') then
  -- Big endian/Little Endian (Configurable with register 22h on the XL/G).
  -- Little endian.
  accel_sample <= slave_master_data_spi_signal &
                  accel_sample(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 8);
  -- Big endian.
  -- accel_sample <= accel_sample(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0) &
  -- slave_master_data_spi_signal;
  byte_read_count <= byte_read_count + 1;
end if;

if (master_slave_data_ack_spi_signal_follower /=
    master_slave_data_ack_spi_signal) then
  master_slave_data_ack_spi_signal_follower <=
    master_slave_data_ack_spi_signal;
Push x00 to the slave to receive the READ bytes back.

```vhdl
if(master_slave_data_ack_spi_signal = '1') then
  master_slave_data_spi_signal <= x"00";
  master_slave_data_rdy_spi_signal <= '1';
end if;
else
  master_slave_data_rdy_spi_signal <= '0';
end if;
```

Assume little endian input for IMU_STATE_FETCH_XL. Generates Big endian words.

Puts big endian on the xyz ports.

```vhdl
when IMU_STATE_FETCH_XL_DONE =>
  accel_data_z <= accel_sample(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 2*IMU_AXIS_WORD_LENGTH_BYTES*8);
  accel_data_y <= accel_sample(2*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 1*IMU_AXIS_WORD_LENGTH_BYTES*8);
  accel_data_x <= accel_sample(1*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);
  cur_imu_state <= IMU_STATE_WAIT;
when IMU_STATE_FETCH_G_SETUP =>
  byte_read_count <= to_unsigned(0, byte_count 'length);
  byte_read_number <= to_unsigned(XL_G_M_register_count_c, byte_number 'length);
  if (command_busy_spi_signal = '0') then
    command_spi_signal <= RD_EN_BIT & std_logic_vector(to_unsigned(G_data_begin_addr, command_spi_signal 'length - 1));
    address_en_spi_signal <= '0';
    data_length_spi_signal <= std_logic_vector(to_unsigned(XL_G_M_register_count_c, data_length_spi_signal 'length));
    master_slave_data_spi_signal <= x"00";
    master_slave_data_rdy_spi_signal <= '1';
    cur_imu_state <= IMU_STATE_FETCH_G;
```
373

end if;

when IMU_STATE_FETCH_G =>

if (byte_read_count = byte_read_number) then
  cur_imu_state <= IMU_STATE_FETCH_G_DONE;
  gyro_processed <= '1';
elsif (slave_master_data_ack_spi_signal = '1') then
  -- Big endian/Little Endian (Configurable with register 22h on the XL/G).
  -- Little endian.
  gyro_sample <= slave_master_data_spi_signal &
    gyro_sample(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 8);
  -- Big endian.
  -- gyro_sample <= gyro_sample(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0) &
    slave_master_data_spi_signal;
  byte_read_count <= byte_read_count + 1;
end if;

if (master_slave_data_ack_spi_signal_follower /=
  master_slave_data_ack_spi_signal) then
  master_slave_data_ack_spi_signal_follower <=
    master_slave_data_ack_spi_signal;
  -- Push x00 to the spi slave to receive the READ bytes back.
  if (master_slave_data_ack_spi_signal = '1') then
    master_slave_data_spi_signal <= x"00";
    master_slave_data_rdy_spi_signal <= '1';
  end if;
else
  master_slave_data_rdy_spi_signal <= '0';
end if;

-- Assume little endian input for IMU_STATE_FETCH_G.
--- Generates Big endian words.
when IMU_STATE_FETCH_G_DONE =>
  gyro_data_z <= gyro_sample(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 2*IMU_AXIS_WORD_LENGTH_BYTES*8);
  gyro_data_y <= gyro_sample(2*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 1*IMU_AXIS_WORD_LENGTH_BYTES*8);
374

gyro_data_x <= gyro_sample(1*IMU_AXIS_WORD_LENGTH_BYTES *8-1 downto 0);
cur_imu_state <= IMU_STATE_WAIT;

when IMU_STATE_FETCH_M_SETUP =>

byte_read_count <= to_unsigned(0, byte_count 'length);
byte_read_number <= to_unsigned(XL_G_M_register_count_c, byte_number 'length);

if (command_busy_spi_signal = '0') then
  command_spi_signal <= RD_EN_BIT & MS_AUTOINCREMENT_BIT & std_logic_vector(
    to_unsigned(M_data_begin_addr, command_spi_signal 'length - 2));
  address_en_spi_signal <= '0';
  data_length_spi_signal <= std_logic_vector(
    to_unsigned(XL_G_M_register_count_c, data_length_spi_signal 'length));
  master_slave_data_spi_signal <= x"00";
end if;

when IMU_STATE_FETCH_M =>

if (byte_read_count = byte_read_number) then
  cur_imu_state <= IMU_STATE_FETCH_M_DONE;
  DRDY_M_processed <= '1';
elsif (slave_master_data_ack_spi_signal = '1') then
  --Big endian/Little Endian (Configurable with register 22h on the XL/G.
  --Little endian.
  mag_sample <= slave_master_data_spi_signal &
    mag_sample(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 8);
  --Big endian.
  --mag_sample <= mag_sample(3*IMU_AXIS_WORD_LENGTH_BYTES *8-1–8 downto 0) & slave_master_data_spi_signal;
byte_read_count <= byte_read_count + 1;
end if;

if (master_slave_data_ack_spi_signal_follower /=
    master_slave_data_ack_spi_signal) then
    master_slave_data_ack_spi_signal_follower <=
    master_slave_data_ack_spi_signal;
    --Push x00 to the spi slave to receive the READ
    bytes back.
    if(master_slave_data_ack_spi_signal = '1') then
        master_slave_data_spi_signal <= x"00";
        master_slave_data_rdy_spi_signal <= '1';
    end if;
    else
        master_slave_data_rdy_spi_signal <= '0';
    end if;

when IMU_STATE_FETCH_M_DONE =>
    mag_data_z <= mag_sample(3*IMU_AXIS_WORD_LENGTH_BYTES
        *8-1 downto 2*IMU_AXIS_WORD_LENGTH_BYTES*8);
    mag_data_y <= mag_sample(2*IMU_AXIS_WORD_LENGTH_BYTES
        *8-1 downto 1*IMU_AXIS_WORD_LENGTH_BYTES*8);
    mag_data_x <= mag_sample(1*IMU_AXIS_WORD_LENGTH_BYTES
        *8-1 downto 0);
    cur_imu_state <= IMU_STATE_WAIT;
end case ;
end if;
end process LSM9DS1_state_machine ;

--
--! @brief The output logic for the cur_imu_state state
    machine.
--! @details Signals associated with certain states are set /
    deset here.
--! Signals of importance include rd_en on the 2
    port memory
--! and the SPI one to many mux select bit.
---! @param clk  Take action on positive edge.
---! @param rst_n  rst_n to initial state.
---!
---!

imu_machine_output:  process (cur_imu_state)
begin

imu_initbuffer_rd_en <= '0';
gyro_data_rdy <= '0';
accel_data_rdy <= '0';
mag_data_rdy <= '0';
imu_initbuffer_wr_en <= '0';

--Set default bus to XL_G.
SPI_SELECT_XL_G_M <= '0';

case cur_imu_state is

when IMU_STATE_WAIT =>

when IMU_STATE_INIT_FETCH_XL_G_NUM_SETUP =>
imu_initbuffer_rd_en <= '1';
when IMU_STATE_INIT_FETCH_XL_G_NUM =>
imu_initbuffer_rd_en <= '1';

when IMU_STATE_INIT_FETCH_M_NUM_SETUP =>
imu_initbuffer_rd_en <= '1';
when IMU_STATE_INIT_FETCH_M_NUM =>
imu_initbuffer_rd_en <= '1';

when IMU_STATE_INIT_FETCH_XL_G =>
imu_initbuffer_rd_en <= '1';
when IMU_STATE_INIT_WRITE_XL_G =>
imu_initbuffer_rd_en <= '1';
SPI_SELECT_XL_G_M <= '0';
when IMU_STATE_INIT_WAIT_XL_G =>
SPI_SELECT_XL_G_M <= '0';

when IMU_STATE_INIT_FETCH_M =>
imu_initbuffer_rd_en <= '1';
when IMU_STATE_INIT_WRITE_M =>
  imu_initbuffer_rd_en <= '1';
  SPI_SELECT_XL_G_M <= '1';
when IMU_STATE_INIT_WAIT_M =>
  SPI_SELECT_XL_G_M <= '1';

when IMU_STATE_FETCH_XL_SETUP =>
  SPI_SELECT_XL_G_M <= '0';
when IMU_STATE_FETCH_XL =>
  SPI_SELECT_XL_G_M <= '0';
when IMU_STATE_FETCH_XL_DONE =>
  accel_data_rdy <= '1';

when IMU_STATE_FETCH_G_SETUP =>
  SPI_SELECT_XL_G_M <= '0';
when IMU_STATE_FETCH_G =>
  SPI_SELECT_XL_G_M <= '0';
when IMU_STATE_FETCH_G_DONE =>
  gyro_data_rdy <= '1';

when IMU_STATE_FETCH_M_SETUP =>
  SPI_SELECT_XL_G_M <= '1';
when IMU_STATE_FETCH_M =>
  SPI_SELECT_XL_G_M <= '1';
when IMU_STATE_FETCH_M_DONE =>
  mag_data_rdy <= '1';
end case;
end process imu_machine_output;

--
@brief data_rdy_catch

@details Catch the data_rdy_interrupts coming from the IMU.

Log the fpga_time immediately.

Then signal state machine to read the appropriate data from IMU over SPI.

@param clk Take action on positive edge.

@param rst_n rst_n to initial state.

data_rdy_catch: process (clk, rst_n)
begin
  if rst_n = '0' then
    DRDY_M_req <= '0';
    gyro_req <= '0';
    accel_req <= '0';
    startup_processed_follower <= '0';
    INT1_A_G_follower <= '0';
    INT2_A_G_follower <= '0';
    DRDY_M_follower <= '0';
    startup_en <= '0';
    gyro_processed_follower <= '0';
    accel_processed_follower <= '0';
    DRDY_M_processed_follower <= '0';
    command_done_spi_signal_follower <= '0';
    INT1_A_G_sync <= '0';
    INT2_A_G_sync <= '0';
    DRDY_M_sync <= '0';
  end if;
end process;
elsif clk'event and clk = '1' then
  -- Synchronize asynch interrupts.
  INT1_A_G_sync <= INT1_A_G;
  INT2_A_G_sync <= INT2_A_G;
  DRDY_M_sync <= DRDY_M;

  if (startup_follower /= startup) then
    startup_follower <= startup;
    if (startup = '1') then
      startup_en <= '1';
    end if;
  elsif (startup_processed_follower /= startup_processed) then
    startup_processed_follower <= startup_processed;
    if (startup_processed = '1') then
      startup_en <= '0';
    end if;
  end if;

  -- DRDY_M is always associated with magnetometer data.
  if (DRDY_M_follower /= DRDY_M_sync) then
    DRDY_M_follower <= DRDY_M_sync;
    if (DRDY_M_sync = '1') then
      mag_fpga_time <= current_fpga_time;
      DRDY_M_req <= '1';
    end if;
  elsif (DRDY_M_processed_follower /= DRDY_M_processed) then
    DRDY_M_processed_follower <= DRDY_M_processed;
    if (DRDY_M_processed = '1') then
      DRDY_M_req <= '0';
    end if;
  end if;
This assumes that data_rdy on INT1_A_G_follower has been programmed to be gyro data rdy. Please check associated matlab startup register definition scripts and the documentation.

if (INT1_A_G_follower /= INT1_A_G_sync) then
    INT1_A_G_follower <= INT1_A_G_sync;
    if (INT1_A_G_sync = '1') then
        gyro_fpga_time <= current_fpga_time;
        gyro_req <= '1';
    end if;
elsif (gyro_processed_follower /= gyro_processed) then
    gyro_processed_follower <= gyro_processed;
    if (gyro_processed = '1') then
        gyro_req <= '0';
    end if;
end if;

This assumes that data_rdy on INT2_A_G_follower has been programmed to be accel data rdy. Please check associated matlab startup register definition scripts and the documentation.

if (INT2_A_G_follower /= INT2_A_G_sync) then
    INT2_A_G_follower <= INT2_A_G_sync;
    if (INT2_A_G_sync = '1') then
        accel_fpga_time <= current_fpga_time;
        accel_req <= '1';
    end if;
elsif (accel_processed_follower /= accel_processed) then
    accel_processed_follower <= accel_processed;
    if (accel_processed = '1') then
        accel_req <= '0';
    end if;
end if;

end if;
end process data_rdy_catch;

When SPI_SELECT_XL_G_M is '0', set up SPI bus to XL_G device.

When SPI_SELECT_XL_G_M is '1', set up SPI bus to M device.
with SPI_SELECT_XL_G_M select
cs_M <= cs_n_signal when '1',
     '1' when others;

with SPI_SELECT_XL_G_M select
cs_XL_G <= cs_n_signal when '0',
          '1' when others;

with SPI_SELECT_XL_G_M select
miso_signal <= miso_M when '1',
        miso_XL_G when others;

end behavior ;
library IEEE ;
use IEEE.STD_LOGIC_1164.ALL ;
use IEEE.NUMERIC_STD.ALL ;

-- This program is free software: you can redistribute it and
/or modify
-- it under the terms of the GNU General Public License as
published by
-- the Free Software Foundation, either version 3 of the
License, or
-- (at your option) any later version.
-- This program is distributed in the hope that it will be
useful,
-- but WITHOUT ANY WARRANTY; without even the implied
warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See
the
-- GNU General Public License for more details.
-- You should have received a copy of the GNU General Public
License
-- along with this program. If not, see <http://www.gnu.org/
licenses/>.

--
-- Chris Casebeer
-- Electrical and Computer Engineering
-- Montana State University
-- 610 Coblleigh Hall
-- Bozeman, MT 59717
-- christopher.casebeer1@msu.montana.edu

--
use IEEE.MATH_REAL.ALL;  ---! Use Real math.

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

LIBRARY GENERAL ;
USE GENERAL.UTILITIES_PKG.ALL;  --- Use General Purpose Libraries
USE GENERAL.GPS_Clock_pkg.ALL;  --- Use GPS Clock information.
USE GENERAL.gps_message_ctl_pkg.ALL;
USE GENERAL.msg_ubx_tim_tm2_pkg.ALL;
USE GENERAL.msg_ubx_nav_sol_pkg.ALL;
USE GENERAL.compile_start_time_pkg.ALL;

---! @brief Assemble blocks of sensor data. Send to buffer.

---! @details

---! @param FPGA_TIME_LENGTH_BYTES  ---! Length of FPGA TIME COUNTER IN BYTES
---! @param TIME_BYTES  ---! Bytes of current time.
---! @param EVENTBYTES  ---! Bytes of event counts.

---! @param RTC_TIME_BYTES  ---! Real Time Clock Word Length Bytes
---! @param NUM_MICS_ACTIVE  ---! Number of Microphones Current Active.

---! @param counter_data_size_g  ---! Event counter system data size.
---! @param counter_address_size_g  ---! Event counter system address size.
---! @param counters_g  ---! Event counter number of events defined.
---! @param GPS_BUFFER_BYTES  --! Size of GPS core buffer

---! @param IMU_AXIS_WORD_LENGTH_BYTES  --! One IMU axis length bytes

---! @param SDRAM_INPUT_BUFFER_BYTES  --! Size of input buffer to SDRAM Controller

---! @param AUDIO_WORD_BYTES  --! Size of an audio word in bytes.

---!

---! @param clock_sys  System clock that drives the fastest ops.

---! @param reset  Reset to initial conditions.

---! @param clk_enable  Clock enable. Used to power on the machine for now.

---!

---! @param last_seqno  The last sequence number.

---! @param log_status  Log Status. Will push a status packet.

---! @param current_fpga_time  FPGA time from the FPGA time component

---! @param log_events  Log Events Flag

---!

---! @param gyro_data_rdy  xyz words from IMU ready

---! @param accel_data_rdy  xyz words from IMU ready

---! @param mag_data_rdy  xyz words from IMU ready

---! @param temp_data_rdy  temp word from IMU ready

---!

---! @param gyro_data_x  x word from IMU gyro (Big endian)

---! @param gyro_data_y  y word from IMU gyro (Big endian)

---! @param gyro_data_z  z word from IMU gyro (Big endian)

---!

---! @param accel_data_x  x word from IMU gyro (Big endian)

---! @param accel_data_y  y word from IMU gyro (Big endian)

---! @param accel_data_z  z word from IMU gyro (Big endian)
---! @param mag_data_x x word from IMU gyro (Big endian)
---! @param mag_data_y y word from IMU gyro (Big endian)
---! @param mag_data_z z word from IMU gyro (Big endian)

---! @param temp_data Temp word from the IMU.

---Audio information.
---! @param audio_data_rdy Audio word rdy
---! @param audio_data Audo word data (Big endian)

---Input Multi Buffer Interface
---! @param flashblock_inbuf_data Data input to the
sdram input buffer
---! @param flashblock_inbuf_wr_en Write enable to the
sdram input buffer
---! @param flashblock_inbuf_clk Clk to the sdram
input buffer
---! @param flashblock_inbuf_addr Address bus to the
sdram input buffer

---! @param flashblock_gpsbuf_addr Address into GPS
core memory
---! @param flashblock_gpsbuf_rd_en Read Enable into
GPS core memory
---! @param flashblock_gpsbuf_clk
---! @param gpsbuf_flashblock_data Data from gps core
memory.

---! @param posbank Position GPS memory
change flag.

---! @param tmbank Time mark GPS memory
change flag.

---! @param gyro_fpga_time FPGA time associated
with a gyro sample.
---! @param accel_fpga_time FPGA time associated
with a accel sample.
---! @param mag_fpga_time FPGA time associated
with a mag sample.
@param temp_fpga_time FPGA time associated with a temp sample.

@param rtc_time Real Time Clock time word input into flashblock.

@param flashblock_counter_rd_addr Read address into the counter component.

@param flashblock_counter_wr_addr Write address into the counter component.

@param flashblock_counter_rd_en Read enable into the counter component.

@param flashblock_counter_wr_en Write enable into the counter component.

@param flashblock_counter_clk Write enable into the counter component.

@param flashblock_counter_data Data into the counter component.

@param counter_flashblock_data Data from the counter component.

@param flashblock_sdram_2k_accumulated Signal 2k accumulated flag relayed to the sdram_controller.

@param dw_en direct write mode output towards sdram controller.

@param dw_amt_valid

@param dw_amt_bytes

@param force_wr_en Force Write Enable to sdram controller. Flush physical memory.

@param sdram_empty When physical ram is empty sdram controller signals sdram_empty.

@param crit_event

@param blocks_past_crit

@param device_byte A byte to add to the flash block.

@param device_byte_ready The flash byte should be written now.
What does this component do?

This component is responsible for assembling segments from all the sensor inputs into the system. The data structures assembled are described in associated documentation (SD_Card_Structure_v1.x.pdf).

Two primary state machines direct flow. One state machine send_block_item is primarily responsible for pushing data out of the component and into the addressed buffer. Different states push different types of data. State transition order dictate the order in which the data is pushed out. Data structure is again noted in associated documentation.

The send_item state machine is a smaller state machine. It is more involved with prioritizing incoming interrupts and jumped to the correct place in send_block_item.

A large process audio_sample is concerned with catching the incoming data and putting it into circular buffers. Send_block_item state machine then reads from theses buffers to assembled the sensor segments. Data is read from circular buffer until read pointer equals write pointer.

How does force write and direct write mode work?

Force write is enabled in a critical event. This will empty all of the physical ram. This is done by the sdram controller. However this takes some time. Blocks will continue to form while sdram is emptying and being written to the sd card.

Thus the number blocks which form after the critical event was received must be
kept track of. One component who must know this information is the sdcard_loader.

Force write is kept on until the sdcard_controller replies with an empty flag.
At this point direct write mode is entered. This mode bypasses the physical sd ram

---IMU words are presented by the IMU controller in big endian format.
They are written out of the flashblock starting high byte of x, low byte of x,

---Words are taken in Big endian. (IMU and Audio)
These are then written out in little endian order inside the data segments.
They then end up little endian on the microSD card.

---TODO:
Fetch last sequence number from magram buffer.
Implement overflow detection on the circular buffers.

entity FlashBlock is

Generic (FPGA_TIME_LENGTH_BYTES : natural := 9; ! Length of FPGA TIME COUNTER IN BYTES
TIME_BYTES : natural := 9 ; ! Bytes of current time.
EVENT_BYTES : natural := 2 ; ! Bytes of event counts.
RTC_TIME_BYTES : natural := 4;
NUM_MICS_ACTIVE : natural := 1;
counter_data_size_g : natural := 8 ;
counter_address_size_g : natural := 9 ;
counters_g : natural := 10 ;
GPS_BUFFER_BYTES : natural := 512;
IMU_AXIS_WORD_LENGTH_BYTES : natural := 2;
SDRAM_INPUT_BUFFER_BYTES : natural := 4096;
AUDIO_WORD_BYTES : natural := 2
)
Port (
clock_sys : in std_logic ;
reset : in std_logic ;
clk_enable : in std_logic ;
last_seqno : in std_logic_vector (31 downto 0);

-- Status information.
log_status : in std_logic ;
current_fpga_time : in std_logic_vector (gps_time_bytes_c*8-1 downto 0);
log_events : in std_logic ;

-- Flash device byte.
device_byte : out std_logic_vector (7 downto 0);
device_byte_ready : out std_logic ;
gyro_data_rdy : in std_logic;
accel_data_rdy : in std_logic;
mag_data_rdy : in std_logic;
temp_data_rdy : in std_logic;

gyro_data_x :in std_logic_vector(
IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
gyro_data_y :in std_logic_vector(
IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
gyro_data_z :in std_logic_vector(
IMU_AXIS WORD_LENGTH_BYTES*8 - 1 downto 0);
accel_data_x :in std_logic_vector(
IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
accel_data_y :in std_logic_vector(
IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
accel_data_z : in std_logic_vector(
  IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);

mag_data_x : in std_logic_vector(
  IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
mag_data_y : in std_logic_vector(
  IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);
mag_data_z : in std_logic_vector(
  IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);

temp_data : in std_logic_vector(
  IMU_AXIS_WORD_LENGTH_BYTES*8 - 1 downto 0);

audio_data_rdy : in std_logic;
audio_data : in std_logic_vector(
  AUDIO_WORD_BYTES*8 - 1 downto 0);

-- Input Multi Buffer Interface
flashblock_inbuf_data : out std_logic_vector(7
downto 0);
flashblock_inbuf_wr_en : out std_logic;
flashblock_inbuf_clk : out std_logic;

-- In the case of a 2*2k multi buffer, upper 3 bits will
index through 0-15 (512 byte blocks)
-- Only increment on the entire address is needed.
flashblock_inbuf_addr : out std_logic_vector(
  natural(trunc(log2(real(SDRAM_INPUT_BUFFER_BYTES-1)))))
downto 0);

flashblock_gpsbuf_addr : out std_logic_vector(
  natural(trunc(log2(real(GPS_BUFFER_BYTES-1)))))
downto 0);
flashblock_gpsbuf_rd_en : out std_logic;
flashblock_gpsbuf_clk : out std_logic;
gpsbuf_flashblock_data : in std_logic_vector(7
downto 0);

posbank : in std_logic;
tmbank : in std_logic;

gyro_fpga_time : in std_logic_vector (gps_time_bytes_c*8-1
downto 0);
accel_fpga_time : in std_logic_vector (gps_time_bytes_c*8-1
downto 0);
mag_fpga_time : in std_logic_vector (gps_time_bytes_c*8-1 downto 0);
temp_fpga_time : in std_logic_vector (gps_time_bytes_c*8-1 downto 0);
rtc_time : in std_logic_vector (RTC_TIME_BYTES*8-1 downto 0);
flashblock_counter_rd_addr : out std_logic_vector (counter_address_size_g-1 downto 0);
flashblock_counter_wr_addr : out std_logic_vector (counter_address_size_g-1 downto 0);
flashblock_counter_rd_en : out std_logic;
flashblock_counter_wr_en : out std_logic;
flashblock_counter_clk : out std_logic;
flashblock_counter_lock : out std_logic;
flashblock_counter_data : out std_logic_vector (counter_data_size_g-1 downto 0);
counter_flashblock_data : in std_logic_vector (counter_data_size_g-1 downto 0);
flashblock_sdram_2k_accumulated : out std_logic;

-- SDRAM Controller Interfacing
dw_en : out std_logic;
dw_amt_valid : out std_logic;
dw_amt_bytes : out std_logic_vector(natural(trunc(log2(real(SDRAM_INPUT_BUFFER_BYTES-1))))) downto 0);
force_wr_en : out std_logic;
sdram_empty : in std_logic;
crit_event : in std_logic;
blocks_past_crit : out std_logic_vector(7 downto 0)
);

end entity FlashBlock ;

---! Flash blocks are written as 512 bytes with no continuation of data between
---! blocks.
Each flash block starts with a 4 byte sequence number between $00000001$ and $FFFFFFFE$. This is the address width of the microSD card. The bit patterns $00000000$ and $FFFFFFFF$ are not used as either can be used to fill unwritten blocks when the device is formatted. Thus a valid sequence number indicates that the block has been written.

Blocks are written sequentially. No blocks will normally be written after an unwritten block. (This latter situation may occur if a write fails leaving a possibly unusable block in an unwritten state.)

The data in each block is written in segments. At the end of each segment is the segment trailer which identifies the type of segment that precedes it. Segment trailers always contain the length of the segment (excluding the length of the segment trailer itself).

Putting them at the end of the segments allows segments to be written as their data accumulates rather than requiring the data to be buffered so that the length can be determined and written before the data is. The end of each block (bytes 510 and 511) will always be a segment trailer. If no valid trailer is found here it indicates that the block is invalid and must be ignored.

Documentation of the flash block segment layout is documented in `SD_Card_Structure_v1.x.doc`.

The architecture behavior of FlashBlock is

```
constant BLOCK_SIZE : natural := 512 ;
```
--- Segment trailer size in bytes for all segment trailers.
constant SEG_TRAILER_SIZE : natural := 2;

---! Block Segment IDs.
constant BLOCK_SEG_UNUSED : std_logic_vector (7 downto 0) := x"01";
constant BLOCK_SEG_STATUS : std_logic_vector (7 downto 0) := x"02";
constant BLOCK_SEG_GPS_TIME_MARK : std_logic_vector (7 downto 0) := x"03";
constant BLOCK_SEG_GPS_POSITION : std_logic_vector (7 downto 0) := x"04";
constant BLOCK_SEG_IMU_GYRO : std_logic_vector (7 downto 0) := x"05";
constant BLOCK_SEG_IMU_ACCEL : std_logic_vector (7 downto 0) := x"06";
constant BLOCK_SEG_IMU_MAG : std_logic_vector (7 downto 0) := x"07";
constant BLOCK_SEG_IMU_TEMP : std_logic_vector (7 downto 0) := x"0A";
constant BLOCK_SEG_EVENT : std_logic_vector (7 downto 0) := x"0B";
constant BLOCK_SEG_SHUTDOWN : std_logic_vector (7 downto 0) := x"0C";
constant BLOCK_SEG_AUDIO : std_logic_vector (7 downto 0) := x"08";

---! Sequence number of the current block.
constant BLOCK_SEQNO_BYTES : natural := 4;
signal block_seqno : unsigned (BLOCK_SEQNO_BYTES *8-1 downto 0);

---! Padding control.
--- No longer needed if padding segment is always added to block.

-- constant PAD_SEG_MIN_SIZE : natural := 1 + SEG_TRAILER_SIZE;
constant PAD_SEG_MIN_SIZE : natural := SEG_TRAILER_SIZE;
constant PAD_SEG_MAX_LENGTH : natural := 255 ;
constant PADDING_BYTE : std_logic_vector (7 downto 0) := (others => '0') ;
signal block_padding_length : unsigned (7 downto 0) ;

--- ! System Status Version, current time, event counts, ---! and tens of microseconds between audio samples.

constant STATUS_VER_BYTES : natural := 2 ;
constant STATUS_AUD_BYTES : natural := 2 ;
constant STATUS_COMPILE_BYTES : natural := 4;
constant STATUS_COMMIT_BYTES : natural := 4;
constant NUM_ACTIVE_MICS_BYTES : natural := 1;
constant DEVICES_ON_VECTOR_BYTES : natural := 2;
constant BATTERY_STATUS_BYTES : natural := 2;
constant OPERATING_STATE_BYTES : natural := 2;

-- constant STATUS_VERSION_NO : std_logic_vector ( STATUS_VER_BYTES*8−1 downto 0)
   -- := x”0001” ;
---Each IMU sample segment is 1 sample long followed by the trailer.

constant IMU_GYRO_SEG_BYTES : natural :=
   3*IMU_AXIS_WORD_LENGTH_BYTES ;
constant IMU_ACCEL_SEG_BYTES : natural :=
   3*IMU_AXIS_WORD_LENGTH_BYTES ;
constant IMU_MAG_SEG_BYTES : natural :=
    3*IMU_AXIS_WORD_LENGTH_BYTES ;
constant IMU_TEMP_SEG_BYTES : natural :=
    IMU_AXIS_WORD_LENGTH_BYTES ;

constant SEG_LEN_GYRO : unsigned (7 downto 0) :=
    TO_UNSIGNED (IMU_GYRO_SEG_BYTES, 8) ;
constant SEG_LEN_ACCEL : unsigned (7 downto 0) :=
    TO_UNSIGNED (IMU_ACCEL_SEG_BYTES, 8) ;
constant SEG_LEN_MAG : unsigned (7 downto 0) :=
    TO_UNSIGNED (IMU_MAG_SEG_BYTES, 8) ;
constant SEG_LEN_TEMP : unsigned (7 downto 0) :=
    TO_UNSIGNED (IMU_TEMP_SEG_BYTES, 8) ;

constant GPS_NAV_SOL_BYTES : natural :=
    gps_time_bytes_c + msg_ubx_nav_sol_ramused_c ;
constant SEG_LEN_GPS_NAV_SOL : unsigned (7 downto 0) :=
    TO_UNSIGNED (GPS_NAV_SOL_BYTES, 8) ;
constant GPS_TIM_TM2_BYTES : natural :=
    gps_time_bytes_c + msg_ubx_tim_tm2_ramused_c ;
constant SEG_LEN_GPS_TIM_TM2 : unsigned (7 downto 0) :=
    TO_UNSIGNED (GPS_TIM_TM2_BYTES, 8) ;

constant STATUS_SEG_BYTES : natural :=
    STATUS_COMPILE_BYTES + STATUS_COMMIT_BYTES + 6*
    gps_time_bytes_c
    + RTC_TIME_BYTES + NUM_ACTIVE_MICS_BYTES ;
constant BLOCK_LEN_STATUS : unsigned (7 downto 0) :=
469   TO_UNSIGNED (STATUS_SEG_BYTES, 8);  
470  
471  
472  
473   constant SHUTDOWN_REASON_BYTES : natural := 1;  
474  
475   constant SHUTDOWN_SEG_BYTES : natural :=  
476         SHUTDOWN_REASON_BYTES;  
477  
478   constant BLOCK_LEN_SHUTDOWN : unsigned (7 downto 0) :=  
479         TO_UNSIGNED (SHUTDOWN_SEG_BYTES, 8);  
480  
481  
482  
483   signal status_written : std_logic;  
484   signal status_written_follower : std_logic;  
485   signal log_status_follower : std_logic;  
486   signal write_status : std_logic;  
487   signal write_status_follower : std_logic;  
488  
489  
490  
491   signal events_written : std_logic;  
492   signal events_written_follower : std_logic;  
493   signal events_data_write : std_logic;  
494   signal log_events_follower : std_logic;  
495  
496  
497  
498   --! Device byte send to Flash flag.  
499   signal device_byte_sending : std_logic ;  
500  
501  
502  
503   --! Byte of a data block to send and flag indicating it is ready to send.  
504   signal block_byte : std_logic_vector (7 downto 0) ;  
505   signal block_byte_ready : std_logic ;  
506   signal block_bytes_left : unsigned (9 downto 0);  
507  
508  
509  
510   --! Acknowledgement that the data block byte is being sent.  
511   signal block_byte_ready_follower : std_logic ;
—! Number of bytes in the current audio segment.

signal audio_seg_length : unsigned (7 downto 0);

—! Number of bytes in the current events segment.

signal events_seg_length : unsigned (7 downto 0);
signal events_checked : unsigned (7 downto 0);
signal start_follower : std_logic;

—! Flash block writing states.

type BlockState is
  BLOCK_STATE_WAIT,
  BLOCK_STATE_BUFFER,
  BLOCK_STATE_SEQNO,
  BLOCK_STATE_PADDING,
  BLOCK_STATE_PADDING_ADD,
  BLOCK_STATE_SEG_PAD,
  BLOCK_STATE_LEN_PAD,

  -- BLOCK_STATE_EVENTS,
  BLOCK_STATE_SEG_ST,
  BLOCK_STATE_LEN_ST,
  BLOCK_STATE_AUDIO_PREFETCH,
  BLOCK_STATE_AUDIO_LAST,
  BLOCK_STATE_SEG_AUD,
  BLOCK_STATE_LEN_AUD,
  BLOCK_STATE_GYRO_PREFETCH,
  BLOCK_STATE_GYRO,
  BLOCK_STATE_GYRO_SEG_ST,
  BLOCK_STATE_GYRO_LEN_ST,
  BLOCK_STATE_ACCEL_PREFETCH,
  BLOCK_STATE_ACCEL,
  BLOCK_STATE_ACCEL_SEG_ST,
  BLOCK_STATE_ACCEL_LEN_ST,
  BLOCK_STATE_MAG_PREFETCH,
  BLOCK_STATE_MAG,
  BLOCK_STATE_MAG_SEG_ST,
  BLOCK_STATE_MAG_LEN_ST,
BLOCK_STATE_TEMP_PREFETCH,
BLOCK_STATE_TEMP,
BLOCK_STATE_TEMP_SEG_ST,
BLOCK_STATE_TEMP_LEN_ST,
BLOCK_STATE_GPS_NAV_SOL_SETUP,
BLOCK_STATE_GPS_NAV_SOL_FETCH,
BLOCK_STATE_GPS_NAV_SOL_POSTIME_SETUP,
BLOCK_STATE_GPS_NAV_SOL_POSTIME_FETCH,
BLOCK_STATE_NAV_SOL_SEG_ST,
BLOCK_STATE_NAV_SOL_LEN_ST,
BLOCK_STATE_GPS_TIM_TM2_SETUP,
BLOCK_STATE_GPS_TIM_TM2_FETCH,
BLOCK_STATE_GPS_TIM_TM2_MARKTIME_SETUP,
BLOCK_STATE_GPS_TIM_TM2_MARKTIME_FETCH,
BLOCK_STATE_TIM_TM2_SEG_ST,
BLOCK_STATE_TIM_TM2_LEN_ST,
BLOCK_STATE_STATUS_COMPILE,
BLOCK_STATE_STATUS_COMMIT,
BLOCK_STATE_STATUS_FPGA_TIME,
BLOCK_STATE_STATUS_ACCEL_TIME,
BLOCK_STATE_STATUS_MAG_TIME,
BLOCK_STATE_STATUS_GYRO_TIME,
BLOCK_STATE_STATUS_TEMP_TIME,
BLOCK_STATE_STATUS_AUDIO_TIME,
BLOCK_STATE_STATUS_RTC_TIME,
BLOCK_STATE_STATUS_MICS,

--- States used to enact direct write and force write modes.
BLOCK_STATE_DIRECT,
BLOCK_STATE_DIRECT_RESET,
BLOCK_STATE_DIRECT_END,
BLOCK_STATE_FORCE,
BLOCK_STATE_FORCE_DONE,
BLOCK_STATE_PADDING_IMM,
BLOCK_STATE_PADDING_ADD_IMM,
BLOCK_STATE_SEG_PAD_IMM,
BLOCK_STATE_LEN_PAD_IMM,
BLOCK_STATE_EVENTS_SETUP,
BLOCK_STATE_EVENTS_FETCH,
BLOCK_STATE_EVENTS_SEG_ST,
BLOCK_STATE_EVENTS_LEN_ST
signal cur_block_state : BlockState;
signal next_block_state : BlockState;
signal end_block_state : BlockState;

--! Flash item writing states.

type ItemState is (ITEM_STATE_WAIT, ITEM_STATE_CHECK_SPACE, ITEM_STATE_NEW_BLOCK, ITEM_STATE_PADDING_END, ITEM_STATE_AUDIO_END, ITEM_STATE_AUDIO_BYTE, ITEM_STATE_STATUS, ITEM_STATE_GYRO, ITEM_STATE_ACCEL, ITEM_STATE_MAG, ITEM_STATE_TEMP, ITEM_STATE_GPS_POS, ITEM_STATE_GPS_TIME_MARK, ITEM_STATE_EVENTS, ITEM_STATE_FORCE, ITEM_STATE_FORCE_DONE, ITEM_STATE_PAUSE);

signal cur_item_state : ItemState;
signal next_item_state : ItemState;
signal end_item_state : ItemState;

--! Number of bytes needed in the current block.

signal bytes_needed : unsigned (7 downto 0);
constant EVENT_SAMPLE_BYTES : natural := 2;

--! Audio input processing signals and constants.

signal audio_written : std_logic; --! An audio byte has been written.
signal audio_written_follower : std_logic; --! Checks for changes to written.
signal audio_byte_count : unsigned (2 downto 0) ;

--!  
bytes with data

signal audio_data_rdy_follower : std_logic;

signal gyro_data_rdy_follower : std_logic;
signal accel_data_rdy_follower : std_logic;
signal mag_data_rdy_follower : std_logic;
signal temp_data_rdy_follower : std_logic;

signal gyro_data_write : std_logic;
signal accel_data_write : std_logic;
signal mag_data_write : std_logic;
signal temp_data_write : std_logic;

signal gyro_data_write_follower : std_logic;
signal accel_data_write_follower : std_logic;
signal mag_data_write_follower : std_logic;
signal temp_data_write_follower : std_logic;

signal gyro_written : std_logic;
signal accel_written : std_logic;
signal mag_written : std_logic;
signal temp_written : std_logic;

signal gyro_written_follower : std_logic;
signal accel_written_follower : std_logic;
signal mag_written_follower : std_logic;
signal temp_written_follower : std_logic;

signal audio_data_write : std_logic;
signal audio_data_write_follower : std_logic;

signal gps_time_data_write : std_logic;
signal gps_pos_data_write : std_logic;

signal gps_pos_written : std_logic;
signal gps_time_written : std_logic;
signal gps_pos_written_follower : std_logic;
signal gps_time_written_follower : std_logic;
signal tmbank_follower : std_logic;
signal posbank_follower : std_logic;

signal audio_data_process_request : std_logic;
signal audio_data_process_request_follower : std_logic;

signal audio_data_processed : std_logic;
signal audio_data_processed_follower : std_logic;

signal gyro_data_process_request : std_logic;
signal gyro_data_process_request_follower : std_logic;

signal gyro_data_processed : std_logic;
signal gyro_data_processed_follower : std_logic;

signal accel_data_process_request : std_logic;
signal accel_data_process_request_follower : std_logic;

signal accel_data_processed : std_logic;
signal accel_data_processed_follower : std_logic;

signal mag_data_process_request : std_logic;
signal mag_data_process_request_follower : std_logic;

signal mag_data_processed : std_logic;
signal mag_data_processed_follower : std_logic;

signal temp_data_process_request : std_logic;
signal temp_data_process_request_follower : std_logic;

signal temp_data_processed : std_logic;
signal temp_data_processed_follower : std_logic;
signal gyro_sample : std_logic_vector(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);
signal accel_sample : std_logic_vector(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);
signal mag_sample : std_logic_vector(3*IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);
signal temp_sample : std_logic_vector(IMU_AXIS_WORD_LENGTH_BYTES*8-1 downto 0);

signal gyro_sample_fpga_time : std_logic_vector(gps_time_bytes_c*8-1 downto 0);
signal accel_sample_fpga_time : std_logic_vector(gps_time_bytes_c*8-1 downto 0);
signal mag_sample_fpga_time : std_logic_vector(gps_time_bytes_c*8-1 downto 0);
signal temp_sample_fpga_time : std_logic_vector(gps_time_bytes_c*8-1 downto 0);

signal audo_sample_fpga_time : std_logic_vector(gps_time_bytes_c*8-1 downto 0);

signal imu_sample_time_buffer : std_logic_vector(((3*IMU_AXIS_WORD_LENGTH_BYTES)+(gps_time_bytes_c))*8 -1 downto 0);

signal flashblock_inbuf_addr_internal : unsigned(natural(trunc(log2(real(SDRAM_INPUT_BUFFER_BYTES-1)))) downto 0);
signal flashblock_gpsbuf_addr_internal : unsigned(natural(trunc(log2(real(GPS_BUFFER_BYTES-1)))) downto 0);

signal flashblock_counter_wr_addr_internal : unsigned(counter_address_size_g-1 downto 0);
signal flashblock_counter_rd_addr_internal : unsigned(counter_address_size_g-1 downto 0);

/* Ultimately here we want to generically specify multiple circular buffers within one space of ram.*/
/* We divide up the space into a number of buffers and then have read and write pointers into each specific circular buffer.*/
Oft forgotten but.....if we want to address $2^4 - 1$ things, we can do this with (3 downto 0)

$2^4 - 1$ allows (3 downto 0) to encode 0–15.

We can use this to calculate any address. Or calculate two address spaces.

Here I calculate an upper set of bits which allow for N circular buffers.

I then calculate a lower set of bits to allow for N samples in a buffer.

I can select which buffer I am in, by simple counting in the upper bits.

constant CIRC_BUFFER_BYTES : natural := 1024;
constant CIRC_BUFFER_SAMPLES : natural := 16;
constant CIRC_BUFFER_WIDTH_BYTES : natural := 8;

signal circbuffer_flashblock_data_internal : std_logic_vector(CIRC_BUFFER_WIDTH_BYTES*8-1 downto 0);
signal flashblock_circbuffer_data_internal : std_logic_vector(CIRC_BUFFER_WIDTH_BYTES*8-1 downto 0);
signal flashblock_circbuffer_wr_en_internal : std_logic;
signal flashblock_circbuffer_rd_en_internal : std_logic;

signal circ_buffer_rd_audio : unsigned(natural(trunc(log2(real(CIRC_BUFFER_SAMPLES-1))))) downto 0);
signal circ_buffer_wr_audio : unsigned(natural(trunc(log2(real(CIRC_BUFFER_SAMPLES-1))))) downto 0);

signal circ_buffer_rd_accel : unsigned(natural(trunc(log2(real(CIRC_BUFFER_SAMPLES-1))))) downto 0);
signal circ_buffer_wr_accel : unsigned(natural(trunc(log2(real(CIRC_BUFFER_SAMPLES-1))))) downto 0);
signal circ_buffer_rd_gyro : unsigned(natural(trunc(log2(real(CIRC_BUFFER_SAMPLES-1))))) downto 0);

These signals allow addressing into one of the circular buffers. We only need to address the number of samples interested in.

These are unsigned and incrementable. They also rollover.
These are the partial addresses which are sent to the ram. They take assignment from the unsigned values above (ie circ_buffer_rd_accel) which store the current positions in the circular buffers.

signal flashblock_circbuffer_sample_rd : std_logic_vector (natural(trunc(log2(real(CIRC_BUFFER_SAMPLES-1)))) downto 0);
signal flashblock_circbuffer_buffer_rd : std_logic_vector (natural(trunc(log2((CIRC_BUFFER_BYTES/CIRC_BUFFER_WIDTH_BYTES)/CIRC_BUFFER_SAMPLES-1)))) downto 0);
signal flashblock_circbuffer_sample_wr : std_logic_vector (natural(trunc(log2(real(CIRC_BUFFER_SAMPLES-1)))) downto 0);
signal flashblock_circbuffer_buffer_wr : std_logic_vector (natural(trunc(log2((CIRC_BUFFER_BYTES/CIRC_BUFFER_WIDTH_BYTES)/CIRC_BUFFER_SAMPLES-1)))) downto 0);
These are simply used to simply always store the concat of the
flashblock_circbuffer_sample_rd &
flashblock_circbuffer_buffer_rd
Used as input to the two port ram. This cures a modelsim error whereby the input
port map to a 2port ram cannot be a concatenation.

signal flashblock_circbuffer_wr_address :
  std_logic_vector(flashblock_circbuffer_buffer_wr'length +
  flashblock_circbuffer_sample_wr'length -1 downto 0);
signal flashblock_circbuffer_rd_address :
  std_logic_vector(flashblock_circbuffer_buffer_wr'length +
  flashblock_circbuffer_sample_wr'length -1 downto 0);
signal circbuffer_clk : std_logic;

constant circbuffer_audio_select : std_logic_vector(
  flashblock_circbuffer_buffer_rd'length - 1 downto 0)
:= std_logic_vector(
  to_unsigned(0,flashblock_circbuffer_buffer_rd'length));
constant circbuffer_accel_select : std_logic_vector(
  flashblock_circbuffer_buffer_rd'length - 1 downto 0)
:= std_logic_vector(
  to_unsigned(1,flashblock_circbuffer_buffer_rd'length));
constant circbuffer_gyro_select : std_logic_vector(
  flashblock_circbuffer_buffer_rd'length - 1 downto 0)
:= std_logic_vector(
  to_unsigned(2,flashblock_circbuffer_buffer_rd'length));
constant circbuffer_mag_select : std_logic_vector(
  flashblock_circbuffer_buffer_rd'length - 1 downto 0)
:= std_logic_vector(
  to_unsigned(3,flashblock_circbuffer_buffer_rd'length));
constant circbuffer_temp_select : std_logic_vector(
  flashblock_circbuffer_buffer_rd'length - 1 downto 0)
:= std_logic_vector(
  to_unsigned(4,flashblock_circbuffer_buffer_rd'length));
Signals relating to servicing direct write and force modes of sdram.

-- SDRAM can only service 2k at a time, so this is why 2k is of interest.

alias accumulated_2k : std_logic is flashblock_inbuf_addr_internal(11);
alias address_2k  : unsigned(10 downto 0) is flashblock_inbuf_addr_internal(10 downto 0);
signal accumulated_2k_follower          : std_logic;

signal dw_en_internal : std_logic;

signal block_padding_length_imm       : unsigned (7 downto 0);

signal crit_event_follower           : std_logic;
signal crit_event_write              : std_logic;

signal crit_event_written            : std_logic;
signal crit_event_written_follower   : std_logic;

New signals related to allowing continued block formation during force_wr.

signal force_wr_en_signal            : std_logic;
signal blocks_past_crit_signal       : unsigned (7 downto 0);

signal sdram_empty_follower          : std_logic;
signal empty_done                    : std_logic;
signal empty_serviced                : std_logic;
signal empty_serviced_follower       : std_logic;

--!

@brief General means to insert multi-byte items into the block.
A number of multi-byte items may need to be inserted into the block. A buffer large enough to handle all of them is defined and a signal to contain the number of bytes of the current transfer is defined as well. The maximum natural function determines the size that the buffer will need to be.

```
type NaturalArray is array (natural range <> of natural;

function MAX_NATURAL (tbl : NaturalArray) return natural is
  variable max_value : natural := 0;
  begin
    for index in tbl'range loop
      if max_value < tbl (index) then
        max_value := tbl (index);
      end if;
    end loop;
    return max_value;
  end MAX_NATURAL;

constant BYTE_LENGTH_TBL : NaturalArray :=
  ( TIME_BYTES, BLOCK_SEQNO_BYTES, STATUS_VER_BYTES, EVENT_BYTES, STATUS_AUD_BYTES, 3*IMU_AXIS_WORD_LENGTH_BYTES, RTC_TIME_BYTES, NUM_ACTIVE_MICS_BYTES, STATUS_COMPILE_BYTES, STATUS_COMMIT_BYTES, gps_time_bytes_c);

constant BYTE_BUFFER_SIZE : natural := MAX_NATURAL (BYTE_LENGTH_TBL);

signal byte_buffer : std_logic_vector (BYTE_BUFFER_SIZE*8-1 downto 0);
```

--- Multi-byte transfer signals.
signal byte_count : unsigned (7 downto 0);
signal byte_number : unsigned (7 downto 0);

begin
    -- IMU and Audio data are buffered here in a circular manner.
circbuffer : altsyncram
    GENERIC MAP (
        address_aclr_b => "NONE",
        address_reg_b => "CLOCK0",
        clock_enable_input_a => "BYPASS",
        clock_enable_input_b => "BYPASS",
        clock_enable_output_b => "BYPASS",
        intended_device_family => "Cyclone V",
        lpm_type => "altsyncram",
        numwords_a => 128,
        numwords_b => 128,
        operation_mode => "DUAL_PORT",
        outdata_aclr_b => "NONE",
        outdata_reg_b => "UNREGISTERED",
        power_up_uninitialized => "FALSE",
        rdcontrol_reg_b => "CLOCK0",
        read_during_write_mode_mixed_ports => "DONT_CARE",
        widthad_a => 7,
        widthad_b => 7,
        width_a => 64,
        width_b => 64,
        width_byteena_a => 1
    )
    PORT MAP (
        address_a => flashblock_circbuffer_wr_address,
        clock0 => circbuffer_clk,
        data_a => flashblock_circbuffer_data_internal,
        rden_b => flashblock_circbuffer_rd_en_internal,
        wren_a => flashblock_circbuffer_wr_en_internal,
        address_b => flashblock_circbuffer_wr_address,
        q_b => circbuffer_flashblock_data_internal
    );
Event counters need to be locked when they are moved one byte at a time. This prevents a counter spanning two blocks from being changed while it is being moved. Until byte-by-byte moving is being done locking is not needed.

```vhdl
flashblock_inbuf_data <= block_byte;
flashblock_inbuf_wr_en <= device_byte_sending;
flashblock_inbuf_clk <= not clock_sys;

-- Take the internal memory pointers and map them out.
flashblock_inbuf_addr <= std_logic_vector(flashblock_inbuf_addr_internal);
flashblock_gpsbuf_addr <= std_logic_vector(flashblock_gpsbuf_addr_internal);
flashblock_counter_rd_addr <= std_logic_vector(flashblock_counter_rd_addr_internal);
flashblock_counter_wr_addr <= std_logic_vector(flashblock_counter_wr_addr_internal);

circbuffer_clk <= not clock_sys;
flashblock_circbuffer_rd_address <= flashblock_circbuffer_buffer_rd & flashblock_circbuffer_sample_rd;
flashblock_circbuffer_wr_address <= flashblock_circbuffer_buffer_wr & flashblock_circbuffer_sample_wr;
dw_en <= dw_en_internal;
force_wr_en <= force_wr_en_signal;
blocks_past_crit <= std_logic_vector(blocks_past_crit_signal);

--! @brief Send a byte to the Flash device when one is ready.
```
Send device byte: process (clock_sys, reset)
begin
  if (reset = '0') then
    device_byte_sending <= '0';
    block_byte_ready_follower <= '0';
    device_byte <= (others => '0');
    flashblock_inbuf_addr_internal <= (others => '0');
    accumulated_2k_follower <= '0';
    flashblock_sdram_2k_accumulated <= '0';
    block_bytes_left <= TO_UNSIGNED (BLOCK_SIZE, block_bytes_left'length);
  elsif (clock_sys'event and clock_sys = '1') then
    if (clk_enable) = '1' then
      device_byte_sending <= '0';
      flashblock_sdram_2k_accumulated <= '0';
      if (device_byte_sending = '1') then
        flashblock_inbuf_addr_internal <= flashblock_inbuf_addr_internal + 1;
        if (accumulated_2k_follower /= accumulated_2k) then
          accumulated_2k_follower <= accumulated_2k;
          flashblock_sdram_2k_accumulated <= '1';
        end if;
      end if;
    end if;
  end if;
end send_device_byte;
end if;

if (block_byte_ready_follower /= block_byte_ready)
then
  block_byte_ready_follower <= block_byte_ready;
end if;

if (block_byte_ready = '1') then
  device_byte_sending <= '1';
  device_byte <= block_byte;
  if (block_bytes_left = 1) then
    block_bytes_left <= TO_UNSIGNED (BLOCK_SIZE,
    block_bytes_left'length);
  else
    block_bytes_left <= block_bytes_left - 1;
  end if;
end if;

end process send_device_byte;

---
---! @brief Send a data item to the Flash block.
---! @details This state machine handles the actual data movement of the assembly. Data is fetched out of buffers, either the entities circular buffer or out of the GPS’s buffer, and then placed into a byte shift buffer which sends the data out to the inbuf of sdram. Other functionality includes the ability to add segments such as initial block number and padding.
---
---! @param clock_sys Take action on positive edge.
---! @param reset Reset to initial state.
--
send_block_item: process (clock_sys, reset)
begin
if reset = '0' then
  cur_block_state <= BLOCK_STATE_WAIT;
  end_block_state <= BLOCK_STATE_WAIT;
  block_byte_ready <= '0';
  block_byte <= (others => '0');
  --clear_events <= '0';
  audio_seg_length <= (others => '0');
  audio_written <= '0';
  gyro_written <= '0';
  accel_written <= '0';
  mag_written <= '0';
  temp_written <= '0';
  gps_time_written <= '0';
  gps_pos_written <= '0';
  status_written <= '0';
  flashblock_counter_rd_addr_internal <= (others => '0');
  flashblock_counter_wr_addr_internal <= (others => '0');
  events_written <= '0';
  flashblock_counter_rd_en <= '0';
  flashblock_counter_wr_en <= '0';
  flashblock_counter_lock <= '0';
  events_checked <= (others => '0');
  events_seg_length <= (others => '0');
  flashblock_gpsbuf_rd_en <= '0';
circ_buffer_rd_audio <= (others => '0');
circ_buffer_rd_gyro <= (others => '0') ;  
circ_buffer_rd_accel <= (others => '0') ;  
circ_buffer_rd_mag <= (others => '0') ;  
circ_buffer_rd_temp <= (others => '0') ;  

flashblock_circbuffer_buffer_rd <= (others => '0') ;  
flashblock_circbuffer_sample_rd <= (others => '0') ;  
flashblock_circbuffer_rd_en_internal <= '0';  
crit_event_written <= '0';  
dw_en_internal <= '0';  
dw_amt_valid <= '0';  
dw_amt_bytes <= (others => '0') ;  
force_wr_en_signal <= '0';  

force_wr_en_signal <= '0';  
empty_serviced <= '0';  
blocks_past_crit_signal <= to_unsigned(0, blocks_past_crit_signal ' length);  

elsif clock_sys 'event and clock_sys = '1' then  
if (clk_enable) = '1' then  
  -- Clear asserted signals when they have been acknowledged.  
  if (block_byte_ready = '1' and block_byte_ready_follower = '1') then  
    block_byte_ready <= '0' ;  
  end if ;  
  -- Acknowledge that audio written has been received by lower process.  
  -- This is simply acknowledgement.  
  if (audio_written = '1' and audio_written_follower = '1') then  
    audio_written <= '0' ;  
  end if ;
if (gyro_written = '1' and gyro_written_follower = '1')
    then
        gyro_written <= '0';
    end if;

if (accel_written = '1' and accel_written_follower = '1')
    then
        accel_written <= '0';
    end if;

if (mag_written = '1' and mag_written_follower = '1')
    then
        mag_written <= '0';
    end if;

if (temp_written = '1' and temp_written_follower = '1')
    then
        temp_written <= '0';
    end if;

if (gps_pos_written = '1' and gps_pos_written_follower = '1')
    then
        gps_pos_written <= '0';
    end if;

if (gps_time_written = '1' and gps_time_written_follower = '1')
    then
        gps_time_written <= '0';
    end if;

if (status_written = '1' and status_written_follower = '1')
    then
        status_written <= '0';
    end if;

if (events_written = '1' and events_written_follower = '1')
    then
        events_written <= '0';
    end if;

if (crit_event_written = '1' and crit_event_written_follower = '1')
    then
        crit_event_written <= '0';
    end if;
if (empty_serviced_follower = '1' and empty_serviced = '1')
  then
    empty_serviced <= '0';
  end if;

if block_byte_ready = '0' and block_byte_ready_follower = '0' then

  -- Send another byte from an item.
  case cur_block_state is

    -- Switch to the next state if there is one.
    when BLOCK_STATE_WAIT =>
      cur_block_state <= next_block_state;

    -- Write out the bytes in the byte buffer.
    when BLOCK_STATE_BUFFER =>
      --
      block_byte <= byte_buffer (7 downto 0);
      block_byte_ready <= '1';

      if byte_count = byte_number then
        cur_block_state <= end_block_state;
      else
        byte_count <= byte_count + 1;
        byte_buffer (byte_buffer'length-8-1 downto 0) <=
                    byte_buffer (byte_buffer'length-1 downto 8);
      end if;

    -- Write out the block header sequence number.
    when BLOCK_STATE_SEQNO =>
      byte_buffer (BLOCK_SEQNO_BYTES*8-1 downto 0) <=
        STD_LOGIC_VECTOR (block_seqno);
      byte_number <= TO_UNSIGNED (BLOCK_SEQNO_BYTES,
                                  byte_number',
                                  length);
  end case;

byte_count <= TO_UNSIGNED (1, byte_count', length);
end_block_state <= BLOCK_STATE_WAIT;
cur_block_state <= BLOCK_STATE_BUFFER;

-- Write out padding.

when BLOCK_STATE_PADDING =>
cur_block_state <= BLOCK_STATE_PADDING_ADD;
byte_count <= TO_UNSIGNED (1, byte_count', length);

when BLOCK_STATE_PADDING_ADD =>
  -- CC byte count init to 1. Then increments within this state.
  if (block_padding_length = 0) then
cur_block_state <= BLOCK_STATE_SEG_PAD;
elsif byte_count = block_padding_length then
cur_block_state <= BLOCK_STATE_SEG_PAD;
block_byte <= PADDING_BYTE;
block_byte_ready <= '1';
else
  byte_count <= byte_count + 1;
  block_byte <= PADDING_BYTE;
  block_byte_ready <= '1';
end if;

when BLOCK_STATE_SEG_PAD =>
cur_block_state <= BLOCK_STATE_LEN_PAD;
  --Send the segment trailer ID.
  block_byte <= BLOCK_SEG UNUSED;
  block_byte_ready <= '1';

when BLOCK_STATE_LEN_PAD =>
  -- If we are in direct wr mode, we need to check to see if we
  -- can leave it after ending another block. Padding is always inserted
  -- at the end of a block.
  if (dw_en_internal = '1') then
cur_block_state <= BLOCK_STATE_DIRECT;
else
  -- Here I increment the blocks_past_crit signal.
  if (force_wr_en_signal = '1') then
blocks_past_crit_signal <= blocks_past_crit_signal + 1;
end if;
cur_block_state <= BLOCK_STATE_WAIT;
end if;
--Send the segment trailer ID.
block_byte <= STD_LOGIC_VECTOR (block_padding_length);
block_byte_ready <= '1';

--Write out the status segment. This is a long string
--of states. Each state is responsible for one field of the status
--segment. The docs for sd_card_structure are available with this code.

when BLOCK_STATE_STATUS_COMPILE =>
  byte_buffer (STATUS_COMPILE_BYTES*8-1 downto 0) <=
  std_logic_vector(to_unsigned(compile_timestamp_c,
    STATUS_COMPILE_BYTES*8));
  byte_number <= TO_UNSIGNED (STATUS_COMPILE_BYTES,
    byte_number'length);
  byte_count <= TO_UNSIGNED (1, byte_count'length);
end_block_state <= BLOCK_STATE_STATUS_COMMIT;
cur_block_state <= BLOCK_STATE_BUFFER;
status_written <= '1';

--Write out the system status time.

when BLOCK_STATE_STATUS_COMMIT =>
  byte_buffer (STATUS_COMMIT_BYTES*8-1 downto 0) <=
  std_logic_vector(to_unsigned(commit_timestamp_c,
    STATUS_COMMIT_BYTES*8));
  byte_number <= TO_UNSIGNED (STATUS_COMMIT_BYTES,
    byte_number'length);
  byte_count <= TO_UNSIGNED (1, byte_count'length);
end_block_state <= BLOCK_STATE_STATUS_FPGA_TIME;
cur_block_state <= BLOCK_STATE_BUFFER;

when BLOCK_STATE_STATUS_FPGA_TIME =>
  byte_buffer (gps_time_bytes_c*8-1 downto 0) <=
    current_fpga_time;
  byte_number <= TO_UNSIGNED (gps_time_bytes_c
    , byte_number'
    length);
  byte_count <= TO_UNSIGNED (1, byte_count'
    length);
  end_block_state <= BLOCK_STATE_STATUS_ACCEL_TIME ;
  cur_block_state <= BLOCK_STATE_BUFFER ;

when BLOCK_STATE_STATUS_ACCEL_TIME =>
  byte_buffer (gps_time_bytes_c*8-1 downto 0) <=
    accel_fpga_time;
  byte_number <= TO_UNSIGNED (gps_time_bytes_c
    , byte_number'
    length);
  byte_count <= TO_UNSIGNED (1, byte_count'
    length);
  end_block_state <= BLOCK_STATE_STATUS_MAG_TIME ;
  cur_block_state <= BLOCK_STATE_BUFFER ;

when BLOCK_STATE_STATUS_MAG_TIME =>
  byte_buffer (gps_time_bytes_c*8-1 downto 0) <=
    mag_fpga_time;
  byte_number <= TO_UNSIGNED (gps_time_bytes_c
    , byte_number'
    length);
  byte_count <= TO_UNSIGNED (1, byte_count'
    length);
  end_block_state <= BLOCK_STATE_STATUS_GYRO_TIME ;
  cur_block_state <= BLOCK_STATE_BUFFER ;

when BLOCK_STATE_STATUS_GYRO_TIME =>
  byte_buffer (gps_time_bytes_c*8-1 downto 0) <=
    gyro_fpga_time;
  byte_number <= TO_UNSIGNED (gps_time_bytes_c
    ,
byte_count <= TO_UNSIGNED (1, byte_count', length) ;
end_block_state <= BLOCK_STATE_STATUS_TEMP_TIME ;
cur_block_state <= BLOCK_STATE_BUFFER ;

when BLOCK_STATE_STATUS_TEMP_TIME =>
  byte_buffer (gps_time_bytes_c*8-1 downto 0) <=
    temp_fpga_time ;
  byte_number <= TO_UNSIGNED (gps_time_bytes_c ,
    byte_number', length) ;
  byte_count <= TO_UNSIGNED (1, byte_count', length) ;
end_block_state <= BLOCK_STATE_STATUS_AUDIO_TIME ;
cur_block_state <= BLOCK_STATE_BUFFER ;

when BLOCK_STATE_STATUS_AUDIO_TIME =>
  byte_buffer (gps_time_bytes_c*8-1 downto 0) <=
    audio_sample_fpga_time ;
  byte_number <= TO_UNSIGNED (gps_time_bytes_c ,
    byte_number', length) ;
  byte_count <= TO_UNSIGNED (1, byte_count', length) ;
end_block_state <= BLOCK_STATE_STATUS_RTC_TIME ;
cur_block_state <= BLOCK_STATE_BUFFER ;

when BLOCK_STATE_STATUS_RTC_TIME =>
  byte_buffer (RTC_TIME_BYTES*8-1 downto 0) <=
    rtc_time ;
  byte_number <= TO_UNSIGNED (RTC_TIME_BYTES ,
    byte_number', length) ;
  byte_count <= TO_UNSIGNED (1, byte_count', length) ;
end_block_state <= BLOCK_STATE_STATUS_MICS ;
cur_block_state <= BLOCK_STATE_BUFFER ;

when BLOCK_STATE_STATUS_MICS =>
byte_buffer (NUM_ACTIVE_MICS_BYTES*8-1 downto 0) <=
std_logic_vector(to_unsigned(NUM_MICS_ACTIVE, NUM_ACTIVE_MICS_BYTES*8));
byte_number <= TO_UNSIGNED (NUM_ACTIVE_MICS_BYTES, byte_number',
byte_count <= TO_UNSIGNED (1, byte_count',
end_block_state <= BLOCK_STATE_SEG_ST;
cur_block_state <= BLOCK_STATE_BUFFER;

-- Write out the system status segment ID and length.
when BLOCK_STATE_SEG_ST =>
cur_block_state <= BLOCK_STATE_LEN_ST;
block_byte <= BLOCK_SEG_STATUS;
block_byte_ready <= '1';

when BLOCK_STATE_LEN_ST =>
block_byte <= STD_LOGIC_VECTOR (BLOCK_LEN_STATUS);
block_byte_ready <= '1';

if(crit_event_write = '1') then
cur_block_state <= BLOCK_STATE_PADDING_IMM;
else
cur_block_state <= BLOCK_STATE_WAIT;
end if;

-- Immediately pad out segment. Do not return to item machine.
when BLOCK_STATE_PADDING_IMM =>
cur_block_state <= BLOCK_STATE_PADDING_ADD_IMM;
byte_count <= TO_UNSIGNED (1, byte_count',
if block_bytes_left >= PAD_SEG_MAX_LENGTH + SEG_TRAILER_SIZE + PAD_SEG_MIN_SIZE then
block_padding_length_imm <= TO_UNSIGNED (PAD_SEG_MAX_LENGTH, block_padding_length’length);

elsif (block_bytes_left = PAD_SEG_MAX_LENGTH + SEG_TRAILER_SIZE + 1) then

block_padding_length_imm <= TO_UNSIGNED (PAD_SEG_MAX_LENGTH - 1, block_padding_length’length);

else

— We are writing to end of block, signal crit_event_written.
— Needed to pad all the way to end of block.
— Max pad segment size is 255.

crit_event_written <= '1';
block_padding_length_imm <=
block_bytes_left (block_padding_length’length -1 downto 0) - SEG_TRAILER_SIZE; — Keep the compiler from complaining — that the number is too big to fit.

end if;

when BLOCK_STATE_PADDING_ADD_IMM =>

if block_padding_length_imm = 0 then
cur_block_state <= BLOCK_STATE_SEG_PAD_IMM;
elsif byte_count = block_padding_length_imm then
cur_block_state <= BLOCK_STATE_SEG_PAD_IMM;
block_byte <= PADDING_BYTE;
block_byte_ready <= '1';
else
byte_count <= byte_count + 1;
block_byte <= PADDING_BYTE;
block_byte_ready <= '1';
end if;
when BLOCK_STATE_SEG_PAD_IMM =>
  cur_block_state <= BLOCK_STATE_LEN_PAD_IMM;
  block_byte <= BLOCK_SEG_UNUSED;
  block_byte_ready <= '1';

when BLOCK_STATE_LEN_PAD_IMM =>
  -- Checking if we actually wrote all the way to end of block.
  if (crit_event_write = '1') then
    cur_block_state <= BLOCK_STATE_PADDING_IMM;
  else
    cur_block_state <= BLOCK_STATE_FORCE;
  end if;

-- cur_block_state <= BLOCK_STATEFORCE;
-- Send the segment trailer ID.
block_byte <= STD_LOGIC_VECTOR (block_padding_length_imm);
block_byte_ready <= '1';

-- Force write is enabled. The calculation of the 2k boundary
-- cannot occur here. Force write will take some amount of time. Thus
-- the next complete block formed will be the first dw_en pulse.
when BLOCK_STATE_FORCE =>
  blocks_past_crit_signal <= to_unsigned(0, blocks_past_crit_signal'length);
  force_wr_en_signal <= '1';
  cur_block_state <= BLOCK_STATE_WAIT;

when BLOCK_STATE_FORCE_DONE =>
  empty_serviced <= '1';
  force_wr_en_signal <= '0';
  -- if (address_2k =0) then
  --  -- Reset dw_en internal to '0' by going to state BLOCK_STATE_DIRECT_RESET.
  --  -- This serves only to help sdloader.
  -- cur_block_state <= BLOCK_STATE_DIRECT_RESET;
  -- force_wr_en_signal <= '0';
— Wait for sdram_empty before any assert of dw_en. This way
— sdloader will have gotten the last data_rdy and
data_nbytes
— associated with the force_wr before calculating
critical block
— number.
— dw_amt_bytes <= std_logic_vector(to_unsigned(0,
dw_amt_bytes'length));
— Turn on dw_en. This way, sd_loader will sense
and sampled blocks_past_crit
— calculating the critical block.
dw_en_internal <= '1';
— dw_amt_valid <= '1';
cur_block_state <= BLOCK_STATE_WAIT;

— else
— cur_block_state <= BLOCK_STATE_DIRECT;
— force_wr_en_signal <= '0';
— dw_amt_bytes <= std_logic_vector(to_unsigned(0,
dw_amt_bytes'length));
— dw_en_internal <= '1';
— dw_amt_valid <= '1';
— end if;

— The normal processing of more blocks simply
— accumulates dw_amt_bytes
— and pulses the new amt to the sdram_controller.
— This continues until a multiple of 2k has been
— written out and
— direct write mode is turned off.
when BLOCK_STATE_DIRECT =>
if (address_2k =0) then
dw_en_internal <= '1';
dw_amt_valid <= '1';
dw_amt_bytes <= std_logic_vector(to_unsigned(2048,
dw_amt_bytes'length));
cur_block_state <= BLOCK_STATE_DIRECT_END;
else
dw_en_internal <= '1';
dw_amt_valid <= '1';
dw_amt_bytes <= std_logic_vector(resize(address_2k,
dw_amt_bytes'length));
cur_block_state <= BLOCK_STATE_DIRECT_RESET;
end if;

-- A better way of doing this probably exists.
    However I wanted to
    -- keep the pulse off next to where it is used. The
dw_amt_valid
    -- has to go out in the address_2k=0 case as well.
when BLOCK_STATE_DIRECT_RESET =>
dw_amt_valid <= '0';
cur_block_state <= BLOCK_STATE_WAIT;

when BLOCK_STATE_DIRECT_END =>
dw_amt_valid <= '0';
dw_en_internal <= '0';
cur_block_state <= BLOCK_STATE_WAIT;

-- Write out the oldest audio byte after previous
    write has
    -- been acknowledged.
when BLOCK_STATE_AUDIO_PREFETCH =>

flashblock_circbuffer_buffer_rd <=
circbuffer_audio_select;
flashblock_circbuffer_sample_rd <= std_logic_vector(
    circ_buffer_rd_audio);
circ_buffer_rd_audio <= circ_buffer_rd_audio + 1;
flashblock_circbuffer_rd_en_internal <= '1';
cur_block_state <= BLOCK_STATE_AUDIO_LAST;

when BLOCK_STATE_AUDIO_LAST =>
    if (audio_written = '0' and
        audio_written_follower = '0') then
        -- Depending on buffer level, write the audio word.
        -- Might want to turn off rd_en by default in the
        upper process area.
        -- Multiple states will possibly make use of rd_en.
        flashblock_circbuffer_rd_en_internal <= '0';
    
byte_buffer (AUDIO_WORD_BYTES*8-1 downto 0) <=
circbuffer_flashblock_data_internal(
    AUDIO_WORD_BYTES*8-1 downto 0);
byte_number <= TO_UNSIGNED (AUDIO_WORD_BYTES, byte_number'\n                    length);
byte_count <= TO_UNSIGNED (1, byte_count'\n                    length);
end_block_state <= BLOCK_STATE_WAIT;
cur_block_state <= BLOCK_STATE_BUFFER;
audio_seg_length <= audio_seg_length +
                    to_unsigned(AUDIO_WORD_BYTES, audio_seg_length'\n                    length);

--Big no no. For multibyte data you must go to
--BLOCK_STATE_BUFFER
--and have it send the block byte ready!
--block_byte_ready <= '1';
audio_written <= '1';
end if;

when BLOCK_STATE_SEG_AUD =>
cur_block_state <= BLOCK_STATE_LEN_AUD;
block_byte <= BLOCK_SEG_AUDIO;
block_byte_ready <= '1';

when BLOCK_STATE_LEN_AUD =>
cur_block_state <= BLOCK_STATE_WAIT;
block_byte <= STD_LOGIC_VECTOR (\n                    audio_seg_length);
block_byte_ready <= '1';
audio_seg_length <= (others => '0');

when BLOCK_STATE_GYRO_PREFETCH =>
flashblock_circbuffer_buffer_rd <=
circbuffer_gyro_select;
flashblock_circbuffer_sample_rd <= std_logic_vector(\n                    circ_buffer_rd_gyro);
circ_buffer_rd_gyro <= circ_buffer_rd_gyro + 1;
flashblock_circbuffer_rd_en_internal <= '1';
cur_block_state <= BLOCK_STATE_GYRO;

when BLOCK_STATE_GYRO =>

flashblock_circbuffer_rd_en_internal <= '0';

byte_buffer (IMU_GYRO_SEG_BYTES*8-1 downto 0) <=
circbuffer_flashblock_data_internal( IMU_GYRO_SEG_BYTES*8-1 downto 0);

byte_number <= TO_UNSIGNED (IMU_GYRO_SEG_BYTES,
                    byte_number',
                    length);

byte_count <= TO_UNSIGNED (1, byte_count',
                   length);

end_block_state <= BLOCK_STATE_GYRO_SEG_ST ;
cur_block_state <= BLOCK_STATE_BUFFER ;
gyro_written <= '1' ;

---Append Gyro Segment Type Trailer
when BLOCK_STATE_GYRO_SEG_ST =>
cur_block_state <= BLOCK_STATE_GYRO_LEN_ST ;
block_byte <= BLOCK_SEG_IMU_GYRO ;
block_byte_ready <= '1' ;

---Append Gyro Segment Length Trailer
when BLOCK_STATE_GYRO_LEN_ST =>
cur_block_state <= BLOCK_STATE_WAIT ;
block_byte <= STD_LOGIC_VECTOR (SEG_LEN_GYRO) ;
block_byte_ready <= '1' ;

when BLOCK_STATE_ACCEL_PREFETCH =>

flashblock_circbuffer_buffer_rd <=
circbuffer_accel_select;
flashblock_circbuffer_sample_rd <= std_logic_vector(
circ_buffer_rd_accel);
circ_buffer_rd_accel <= circ_buffer_rd_accel + 1;
flashblock_circbuffer_rd_en_internal <= '1';
cur_block_state <= BLOCK_STATE_ACCEL ;

when BLOCK_STATE_ACCEL =>

flashblock_circbuffer_rd_en_internal <= '0';
byte_buffer (IMU_ACCEL_SEG_BYTES*8-1 downto 0) <=
circbuffer_flashblock_data_internal(  
  IMU_ACCEL_SEG_BYTES*8-1 downto 0);  
byte_number <= TO_UNSIGNED(  
  IMU_ACCEL_SEG_BYTES,  
  byte_number',  
  length);  
byte_count <= TO_UNSIGNED(1, byte_count',  
  length);  
end_block_state <= BLOCK_STATE_ACCEL_SEG_ST;  
cur_block_state <= BLOCK_STATE_BUFFER;  
accel_written <= '1';  

when BLOCK_STATE_ACCEL_SEG_ST =>  
cur_block_state <= BLOCK_STATE_ACCEL_LEN_ST;  
block_byte <= BLOCK_SEG_IMU_ACCEL;  
block_byte_ready <= '1';  

---Append Gyro Segment Length Trailer  
when BLOCK_STATE_ACCEL_LEN_ST =>  
cur_block_state <= BLOCK_STATE_WAIT;  
block_byte <= STD_LOGIC_VECTOR(  
  SEG_LEN_ACCEL);  
block_byte_ready <= '1';  

when BLOCK_STATE_MAG_PREFETCH =>  
flashblock_circbuffer_buffer_rd <=  
circbuffer_mag_select;  
flashblock_circbuffer_sample_rd <= std_logic_vector(  
  circ_buffer_rd_mag);  
circ_buffer_rd_mag <= circ_buffer_rd_mag + 1;  
flashblock_circbuffer_rd_en_internal <= '1';  
cur_block_state <= BLOCK_STATE_MAG;  

when BLOCK_STATE_MAG =>  
flashblock_circbuffer_rd_en_internal <= '0';  
byte_buffer (IMU_MAG_SEG_BYTES*8-1 downto 0) <=  
circbuffer_flashblock_data_internal(  
  IMU_MAG_SEG_BYTES*8-1 downto 0);
byte_number <= TO_UNSIGNED (IMU_MAG_SEG_BYTES,
byte_number',
length);
byte_count <= TO_UNSIGNED (1, byte_count',
length);
end_block_state <= BLOCK_STATE_MAG_SEG_ST;
cur_block_state <= BLOCK_STATE_BUFFER;
mag_written <= '1';

when BLOCK_STATE_MAG_SEG_ST =>
cur_block_state <= BLOCK_STATE_MAG_LEN_ST;
block_byte <= BLOCK_SEG_IMU_MAG;
block_byte_ready <= '1';

---Append Gyro Segment Length Trailer
when BLOCK_STATE_MAG_LEN_ST =>
cur_block_state <= BLOCK_STATE_WAIT;
block_byte <= STD_LOGIC_VECTOR (SEG_LEN_MAG);
block_byte_ready <= '1';

when BLOCK_STATE_TEMP_PREFETCH =>
flashblock_circbuffer_buffer_rd <=
circbuffer_temp_select;
flashblock_circbuffer_sample_rd <= std_logic_vector(
circ_buffer_rd_temp);
circ_buffer_rd_temp <= circ_buffer_rd_temp + 1;
flashblock_circbuffer_rd_en_internal <= '1';
cur_block_state <= BLOCK_STATE_TEMP;

when BLOCK_STATE_TEMP =>
flashblock_circbuffer_rd_en_internal <= '0';

byte_buffer (IMU_TEMP_SEG_BYTES*8-1 downto 0) <=
circbuffer_flashblock_data_internal (IMU_TEMP_SEG_BYTES*8-1 downto 0);
byte_number <= TO_UNSIGNED (IMU_TEMP_SEG_BYTES,
byte_number',
length);
byte_count <= TO_UNSIGNED (1, byte_count'length);
end_block_state <= BLOCK_STATE_TEMP_SEG_ST;
cur_block_state <= BLOCK_STATE_BUFFER;
temp_written <= '1';

when BLOCK_STATE_TEMP_SEG_ST =>
cur_block_state <= BLOCK_STATE_TEMP_LEN_ST;
block_byte <= BLOCK_SEG_IMU_TEMP;
block_byte_ready <= '1';

when BLOCK_STATE_TEMP_LEN_ST =>
cur_block_state <= BLOCK_STATE_WAIT;
block_byte <= STD_LOGIC_VECTOR (SEG_LEN_TEMP);
block_byte_ready <= '1';

when BLOCK_STATE_GPS_NAV_SOL_SETUP =>
cur_block_state <= BLOCK_STATE_GPS_NAV_SOL_FETCH;
flashblock_gpsbuf_rd_en <= '1';
gps_pos_written <= '1';
flashblock_gpsbuf_addr_internal <= TO_UNSIGNED (msg_ram_base_c + msg_ubx_nav_sol_ramaddr_c + if_set (posbank, msg_ubx_nav_sol_ramused_c), flashblock_gpsbuf_addr_internal'length);
byte_number <= TO_UNSIGNED (msg_ubx_nav_sol_ramused_c, byte_number'length);
byte_count <= TO_UNSIGNED (1, byte_count'length);

when BLOCK_STATE_GPS_NAV_SOL_FETCH =>
if (byte_count = byte_number) then
  cur_block_state <= BLOCK_STATE_GPS_NAV_SOL_POSTIME_SETUP;
flashblock_gpsbuf_rd_en <= '0';
—Send the last byte.
block_byte_ready <= '1';
block_byte <= gpsbuf_flashblock_data;

else
  byte_count <= byte_count + 1;
  block_byte_ready <= '1';
  block_byte <= gpsbuf_flashblock_data;
  flashblock_gpsbuf_addr_internal <= flashblock_gpsbuf_addr_internal + 1;
end if;

when BLOCK_STATE_GPS_NAV_SOL_POSTIME_SETUP =>
cur_block_state <= BLOCK_STATE_GPS_NAV_SOL_POSTIME_FETCH;
flashblock_gpsbuf_rd_en <= '1';
flashblock_gpsbuf_addr_internal <= TO_UNSIGNED (msg_ram_base_c +
msg_ram_postime_addr_c +
if_set (posbank,
msg_ram_postime_size_c),
flashblock_gpsbuf_addr_internal'length);

byte_number <= TO_UNSIGNED (gps_time_bytes_c +
  byte_number'length);
byte_count <= TO_UNSIGNED (1, byte_count'length);

when BLOCK_STATE_GPS_NAV_SOL_POSTIME_FETCH =>
  if (byte_count = byte_number) then
    cur_block_state <= BLOCK_STATE_NAV_SOL_SEG_ST;
    flashblock_gpsbuf_rd_en <= '0';
    --Send the last byte.
    block_byte_ready <= '1';
    block_byte <= gpsbuf_flashblock_data;
  else
    byte_count <= byte_count + 1;
    block_byte_ready <= '1';
    block_byte <= gpsbuf_flashblock_data;
    flashblock_gpsbuf_addr_internal <= flashblock_gpsbuf_addr_internal + 1;
when BLOCK_STATE_NAV_SOL_SEG_ST =>
cur_block_state <= BLOCK_STATE_NAV_SOL_LEN_ST;
block_byte <= BLOCK_SEG_GPS_POSITION;
block_byte_ready <= '1';

when BLOCK_STATE_NAV_SOL_LEN_ST =>
cur_block_state <= BLOCK_STATE_WAIT;
block_byte <= STD_LOGIC_VECTOR (SEG_LEN_GPS_NAV_SOL);
block_byte_ready <= '1';

when BLOCK_STATE_GPS_TIM_TM2_SETUP =>
cur_block_state <= BLOCK_STATE_GPS_TIM TM2_FETCH;
flashblock_gpsbuf_rd_en <= '1';
gps_time_written <= '1';
flashblock_gpsbuf_addr_internal <= TO_UNSIGNED (msg_ram_base_c +
msg_ubx_tim_tm2_ramaddr_c +
if_set (tmbank,
msg_ubx_tim_tm2_ramused_c),
flashblock_gpsbuf_addr_internal'length);
byte_number <= TO_UNSIGNED (msg_ubx_tim_tm2_ramused_c,
byte_number'length);
byte_count <= TO_UNSIGNED (1, byte_count'length);

when BLOCK_STATE_GPS_TIM_TM2_FETCH =>
if (byte_count = byte_number) then
cur_block_state <= BLOCK_STATE_GPS_TIM TM2_MARKTIME_SETUP;
flashblock_gpsbuf_rd_en <= '0';
—Send the last byte.
block_byte_ready <= '1';
block_byte <= gpsbuf_flashblock_data;
else
    byte_count <= byte_count + 1;
    block_byte_ready <= '1';
    block_byte <= gpsbuf_flashblock_data;
    flashblock_gpsbuf_addr_internal <= flashblock_gpsbuf_addr_internal + 1;
end if;

when BLOCK_STATE_GPS_TIM_TM2_MARKTIME_SETUP =>

    cur_block_state <= BLOCK_STATE_GPS_TIM_TM2_MARKTIME_FETCH;
    flashblock_gpsbuf_rd_en <= '1';
    flashblock_gpsbuf_addr_internal <= TO_UNSIGNED ( msg_ram_base_c +
        msg_ram_marktime_addr_c +
        if_set ( tmbank,
                msg_ram_marktime_size_c),
        flashblock_gpsbuf_addr_internal'length) ;

    byte_number <= TO_UNSIGNED ( msg_ram_marktime_size_c ,
        byte_number'length);
    byte_count <= TO_UNSIGNED (1, byte_count'length);

when BLOCK_STATE_GPS_TIM_TM2_MARKTIME_FETCH =>

if (byte_count = byte_number) then
    cur_block_state <= BLOCK_STATE_TIM_TM2_SEG_ST;
    flashblock_gpsbuf_rd_en <= '0';
    --Send the last byte.
    block_byte_ready <= '1';
    block_byte <= gpsbuf_flashblock_data;
else
    byte_count <= byte_count + 1;
    block_byte_ready <= '1';
    block_byte <= gpsbuf_flashblock_data;
    flashblock_gpsbuf_addr_internal <= flashblock_gpsbuf_addr_internal + 1;
end if;
when BLOCK_STATE_TIM_TM2_SEG_ST =>
  cur_block_state <= BLOCK_STATE_TIM_TM2_LEN_ST;
  block_byte <= BLOCK_SEG_GPS_TIME_MARK;
  block_byte_ready <= '1';

when BLOCK_STATE_TIM_TM2_LEN_ST =>
  cur_block_state <= BLOCK_STATE_WAIT;
  block_byte <= STD_LOGIC_VECTOR(SEG_LEN_GPS_TIM_TM2);
  block_byte_ready <= '1';

when BLOCK_STATE_EVENTS_SETUP =>
  flashblock_counter_lock <= '1';
  flashblock_counter_rd_en <= '1';
  cur_block_state <= BLOCK_STATE_EVENTS_FETCH;

when BLOCK_STATE_EVENTS_FETCH =>
  flashblock_counter_rd_en <= '0';
  flashblock_counter_wr_en <= '0';

-- If we can't fit an event, a trailer, and a min pad, then cap
-- the current segment.
if (block_bytes_left >= EVENT_SAMPLE_BYTES +
  SEG_TRAILER_SIZE + PAD_SEG_MIN_SIZE) then
  if (events_checked = to_unsigned(counters_g,
    events_checked'length)) then
    -- All the events defined have been checked.
    Reset counter.
    cur_block_state <= BLOCK_STATE_EVENTS_SEG_ST;
    flashblock_counter_rd_addr_internal <= (others => '0');
    events_written <= '1';
    events_checked <= (others => '0');
elsif (counter_flashblock_data =
    std_logic_vector(to_unsigned(0,
    counter_flashblock_data'length))) then
    -- The counter is zero, proceed on.
    flashblock_counter_rd_addr_internal <=
        flashblock_counter_rd_addr_internal + 1;
    flashblock_counter_rd_en <= '1';
    events_checked <= events_checked + 1;
else
    -- The counter is not zero. Store the counter
    position and
    -- its value. Also zero it.
    events_seg_length <= events_seg_length + 2;
    events_checked <= events_checked + 1;
    byte_buffer (EVENTSAMPLEBYTES*8-1 downto 0) <=
        counter_flashblock_data
        &
        std_logic_vector(
            flashblock_counter_rd_addr_internal
            (7 downto 0));
    byte_number <= TO_UNSIGNED(
        EVENTSAMPLEBYTES,byte_number'length);
    byte_count <= TO_UNSIGNED (1, byte_count'length);
    flashblock_counter_wr_en <= '1';
    flashblock_counter_wr_addr_internal <=
        flashblock_counter_rd_addr_internal;
    flashblock_counter_data <= (others => '0');
if (events_checked /= to_unsigned(couters_g,
    events_checked'length) + 1) then
    flashblock_counter_rd_addr_internal <=
        flashblock_counter_rd_addr_internal + 1;
end if;
flashblock_counter_rd_en <= '1';

end_block_state <= BLOCKSTATEEVENTSFETCH;
cur_block_state <= BLOCKSTATEBUFFER;

end if;
else
cur_block_state <= BLOCK_STATE_EVENTS_SEG_ST;
end if;

when BLOCK_STATE_EVENTS_SEG_ST =>
cur_block_state <= BLOCK_STATE_EVENTS_LEN_ST;
block_byte <= BLOCK_SEG_EVENT;
block_byte_ready <= '1';

when BLOCK_STATE_EVENTS_LEN_ST =>
cur_block_state <= BLOCK_STATE_WAIT;
block_byte <= STD_LOGIC_VECTOR (events_seg_length);
block_byte_ready <= '1';
events_seg_length <= (others => '0');

end case;
end if;
end if;
end if;
end process send_block_item;

send_item: process (clock_sys, reset)
begin
if (reset = '0') then
  cur_item_state <= ITEM_STATE_WAIT;
  next_item_state <= ITEM_STATE_WAIT;
  end_item_state <= ITEM_STATE_WAIT;
  next_block_state <= BLOCK_STATE_WAIT;
  write_status_follower <= '0';
  audio_data_write_follower <= '0';
  block_seqno <= unsigned(last_seqno);
elsif (clock_sys'event and clock_sys = '1') then
  if (clk_enable) = '1' then
    -- Wait until the block item process is idle before
    -- starting another
    -- item send.
    -- A lag between next_block_state reaching
    -- cur_block_state requires
    -- use to check that no next_block_state is queued.
    -- Always return here when off send_block_item is engaged.
    -- Wait until the previous block is serviced AND count
    -- updated before
    -- continuing.
    if cur_block_state /= BLOCK_STATE_WAIT or block_byte_ready = '1' then
      next_block_state <= BLOCK_STATE_WAIT;
    else
      -- Send another item.
      case cur_item_state is
      when ITEM_STATE_PAUSE =>
        cur_item_state <= next_item_state;
      when ITEM_STATE_WAIT =>
        -- Start a new block.
        if (block_bytes_left = BLOCK_SIZE) then
          cur_item_state <= ITEM_STATE_NEW_BLOCK;
        end if;
      end case;
  end if;
end if;
 elsif (crit_event_write = '1') then
    if (cur_block_state = BLOCK_STATE_WAIT and
        next_block_state = BLOCK_STATE_WAIT) then
        if (audio_seg_length = 0) then
            bytes_needed <= TO_UNSIGNED (STATUS_SEG_BYTES, bytes_needed'length);
            cur_item_state <= ITEM_STATE_CHECK_SPACE;
            end_item_state <= ITEM_STATE_FORCE;
        else
            cur_item_state <= ITEM_STATE_AUDIO_END;
        end if;
    end if;
elsif (empty_done = '1') then
    if (cur_block_state = BLOCK_STATE_WAIT and
        next_block_state = BLOCK_STATE_WAIT) then
        cur_item_state <= ITEM_STATE_FORCE_DONE;
    end if;
elsif (events_data_write = '1') then
    if (cur_block_state = BLOCK_STATE_WAIT and
        next_block_state = BLOCK_STATE_WAIT) then
        if (audio_seg_length = 0) then
            bytes_needed <= TO_UNSIGNED (EVENT_SAMPLE_BYTES, bytes_needed'length);
            cur_item_state <= ITEM_STATE_CHECK_SPACE;
            end_item_state <= ITEM_STATE_EVENTS;
        else
            cur_item_state <= ITEM_STATE_EVENTS;
            end_item_state <= ITEM_STATE_EVENTS;
        end if;
cur_item_state <= ITEM_STATE_AUDIO_END;
end if;
end if;

-- The interrupt is left high until it is serviced.
elseif (audio_data_write = '1') then
    -- Only proceed if we REALLY are in BLOCK_STATE_WAIT
    if (cur_block_state = BLOCK_STATE_WAIT and
        next_block_state = BLOCK_STATE_WAIT) then
        -- This path will never run.
        if (audio_seg_length /= 0 and
            block_bytes_left = SEG_TRAILER_SIZE) then
            bytes_needed <= TO_UNSIGNED (AUDIO_WORD_BYTES,
                                         bytes_needed',
                                         length);
            cur_item_state <= ITEM_STATE_CHECK_SPACE;
        else
            bytes_needed <= TO_UNSIGNED (AUDIO_WORD_BYTES,
                                           bytes_needed',
                                           length);
            cur_item_state <= ITEM_STATE_AUDIO_END;
        end if;
    end if;
end if;

-- If the audio block is at its max length of 255, cap it.
-- to_unsigned(+1) to avert the overflow.
elseif ((to_unsigned(AUDIO_WORD_BYTES,
                       audio_seg_length'length+1) +
                   audio_seg_length) > 255) then
    cur_item_state <= ITEM_STATE_AUDIO_END;
    else
        bytes_needed <= TO_UNSIGNED (AUDIO_WORD_BYTES,
                                      bytes_needed',
                                      length);
        cur_item_state <= ITEM_STATE_CHECK_SPACE;
elsif (gyro_data_write = '1') then
  if (cur_block_state = BLOCK_STATE_WAIT and
      next_block_state = BLOCK_STATE_WAIT) then
    if (audio_seg_length = 0) then
      bytes_needed <= TO_UNSIGNED (IMU_GYRO_SEG_BYTES,
                                  bytes_needed
                                  length);
      cur_item_state <= ITEM_STATE_CHECK_SPACE ;
      end_item_state <= ITEM_STATE_GYRO ;
    else
      cur_item_state <= ITEM_STATE_AUDIO_END ;
      end if ;
  end if;
elsif (accel_data_write = '1') then
  if (cur_block_state = BLOCK_STATE_WAIT and
      next_block_state = BLOCK_STATE_WAIT) then
    if (audio_seg_length = 0) then
      bytes_needed <= TO_UNSIGNED (IMU_ACCEL_SEG_BYTES,
                                  bytes_needed
                                  length)
      cur_item_state <= ITEM_STATE_CHECK_SPACE ;
      end_item_state <= ITEM_STATE_ACCEL ;
    else
      cur_item_state <= ITEM_STATE_AUDIO_END ;
      end if ;
  end if;
elsif (mag_data_write = '1') then
    if (cur_block_state = BLOCK_STATE_WAIT and
        next_block_state = BLOCK_STATE_WAIT) then
        if (audio_seg_length = 0) then
            bytes_needed <= TO_UNSIGNED (IMU_MAG_SEG_BYTES, bytes_needed 'length);
            cur_item_state <= ITEM_STATE_CHECK_SPACE;
            end_item_state <= ITEM_STATE_MAG;
        else
            cur_item_state <= ITEM_STATE_AUDIO_END;
        end if;
    end if;
elsif (temp_data_write = '1') then
    if (cur_block_state = BLOCK_STATE_WAIT and
        next_block_state = BLOCK_STATE_WAIT) then
        if (audio_seg_length = 0) then
            bytes_needed <= TO_UNSIGNED (IMU_TEMP_SEG_BYTES, bytes_needed 'length);
            cur_item_state <= ITEM_STATE_CHECK_SPACE;
            end_item_state <= ITEM_STATE_TEMP;
        else
            cur_item_state <= ITEM_STATE_AUDIO_END;
        end if;
    end if;
elsif (gps_pos_data_write = '1') then
    if (cur_block_state = BLOCK_STATE_WAIT and
        next_block_state = BLOCK_STATE_WAIT) then
        if (audio_seg_length = 0) then
            bytes_needed <= TO_UNSIGNED (GPS_NAV_SOL_BYTES, bytes_needed 'length);
            cur_item_state <= ITEM_STATE_CHECK_SPACE;
            end_item_state <= ITEM_STATE_MAG;
        else
            cur_item_state <= ITEM_STATE_AUDIO_END;
        end if;
cur_item_state <= ITEM_STATE_CHECK_SPACE;
end_item_state <= ITEM_STATE_GPS_POS;
else
  cur_item_state <= ITEM_STATE_AUDIO_END;
end if;
end if;
elsif (gps_time_data_write = '1') then
  if (cur_block_state = BLOCK_STATE_WAIT and
      next_block_state = BLOCK_STATE_WAIT) then
    if (audio_seg_length = 0) then
      bytes_needed <= TO_UNSIGNED (
        GPS_TIM_TM2_BYTES,
        bytes_needed'length) ;
      cur_item_state <= ITEM_STATE_CHECK_SPACE;
      end_item_state <= ITEM_STATE_GPS_TIME_MARK;
    else
      cur_item_state <= ITEM_STATE_AUDIO_END;
    end if;
  end if;
elsif (write_status = '1') then
  if (cur_block_state = BLOCK_STATE_WAIT and
      next_block_state = BLOCK_STATE_WAIT) then
    if (audio_seg_length = 0) then
      bytes_needed <= TO UNSIGNED ( 
        STATUS_SEG_BYTES, 
        bytes_needed'length) ;
      cur_item_state <= ITEM_STATE_CHECK_SPACE;
      end_item_state <= ITEM_STATE_STATUS;
    else
      cur_item_state <= ITEM_STATE_AUDIO_END;
    end if;
  end if;
elsif (write_status = '1') then
  if (cur_block_state = BLOCK_STATE_WAIT and
      next_block_state = BLOCK_STATE_WAIT) then
    if (audio_seg_length = 0) then
      bytes_needed <= TO UNSIGNED ( 
        STATUS_SEG_BYTES, 
        bytes_needed'length) ;
      cur_item_state <= ITEM_STATE_CHECK_SPACE;
      end_item_state <= ITEM_STATE_STATUS;
else
  cur_item_state <= ITEM_STATE_AUDIO_END;
end if;

end if;

else
  cur_item_state <= ITEM_STATE_AUDIO_END;
end if;

end if;

when ITEM_STATE_CHECK_SPACE =>
  if (block_bytes_left >= bytes_needed
        + SEG_TRAILER_SIZE
        + PAD_SEG_MIN_SIZE) then — or
  cur_item_state <= end_item_state;
  when ITEM_STATE_CHECK_SPACE =>
  if (block_bytes_left >= bytes_needed
        + SEG_TRAILER_SIZE
        + PAD_SEG_MIN_SIZE) then — or
  cur_item_state <= end_item_state;
— If an audio segment is open we need to close it.

else
  cur_item_state <= ITEM_STATE_PAUSE;
  next_item_state <= ITEM_STATE_PADDING_END;

  if (audio_seg_length /= 0) then
    next_block_state <= BLOCK_STATE_SEG_AUD;
  end if;
end if;

— Add an audio byte to the block.
when ITEM_STATE_AUDIO_BYTE =>
  cur_item_state <= ITEM_STATE_PAUSE;
  next_item_state <= ITEM_STATE_WAIT;

  next_block_state <= BLOCK_STATE_AUDIO_PREFETCH;
— End the block with an audio segment.

when ITEM_STATE_AUDIO_END =>
  cur_item_state <= ITEM_STATE_WAIT ;
  next_block_state <= BLOCK_STATE_SEG_AUD ;

— End the block with padding segments.

when ITEM_STATE_PADDING_END =>
  next_block_state <= BLOCK_STATE_PADDING ;

if ( block_bytes_left > PAD_SEG_MAX_LENGTH + SEG_TRAILER_SIZE ) then
  block_padding_length <= TO_UNSIGNED ( PAD_SEG_MAX_LENGTH,
                             block_padding_length' length ) ;
else
  cur_item_state <= ITEM_STATE_WAIT ;
  block_padding_length <=
  — Padding Segment Trailer ALWAYS at end of block.
  block_bytes_left (block_padding_length' length-1 downto 0) -
  SEG_TRAILER_SIZE ; — Keep the compiler from complaining
  — that the number is too big to fit .
endif ;

— Write out the status segment.

when ITEM_STATE_STATUS =>
  cur_item_state <= ITEM_STATE_WAIT ;
  next_block_state <= BLOCK_STATE_STATUS_COMPILE ;
when ITEM_STATE_GYRO =>
  cur_item_state <= ITEM_STATE_WAIT ;
  next_block_state <= BLOCK_STATE_GYRO_PREFETCH ;
when ITEM_STATE_ACCEL =>
  cur_item_state <= ITEM_STATE_WAIT;
  next_block_state <= BLOCK_STATE_ACCEL_PREFETCH;

when ITEM_STATE_MAG  =>
  cur_item_state <= ITEM_STATE_WAIT;
  next_block_state <= BLOCK_STATE_MAG_PREFETCH;

when ITEM_STATE_TEMP  =>
  cur_item_state <= ITEM_STATE_WAIT;
  next_block_state <= BLOCK_STATE_TEMP_PREFETCH;

-- Start a new block.

-- I insert extra pause state here to give cur_block_state
-- time to transition and keep us from triggering
ITEMSTATE_NEW_BLOCK
-- twice.
when ITEM_STATE_NEW_BLOCK =>
  cur_item_state <= ITEM_STATE_PAUSE;
  next_item_state <= end_item_state;
  next_block_state <= BLOCK_STATE_SEQNO;
  block_seqno <= block_seqno + 1;

when ITEM_STATE_GPS_POS  =>
  cur_item_state <= ITEM_STATE_PAUSE;
  next_item_state <= end_item_state;
  next_block_state <= BLOCK_STATE_GPS_NAV_SOL_SETUP;

when ITEM_STATE_GPS_TIME_MARK  =>
  cur_item_state <= ITEM_STATE_PAUSE;
  next_item_state <= end_item_state;
  next_block_state <= BLOCK_STATE_GPS_TIM_TM2_SETUP;

when ITEM_STATE_EVENTS =>
cur_item_state <= ITEM_STATE_PAUSE;
next_item_state <= end_item_state;
next_item_state <= ITEM_STATE_WAIT;
next_block_state <= BLOCK_STATE_EVENTS_SETUP;

when ITEM_STATE_FORCE_DONE =>
cur_item_state <= ITEM_STATE_PAUSE;
next_item_state <= end_item_state;
next_item_state <= ITEM_STATE_WAIT;
next_block_state <= BLOCK_STATE_FORCE_DONE;

-- Here we jump into BLOCK_STATE_COMPILE. We’ll check for
-- crit event flag later once status segment is built.

when ITEM_STATE_FORCE =>
cur_item_state <= ITEM_STATE_PAUSE;
next_item_state <= end_item_state;
next_item_state <= ITEM_STATE_WAIT;
next_block_state <= BLOCK_STATE_FORCE_DONE;

end case;
end if;
end if;
end if;
end process send_item;

---
---
---! @brief Determine when a status segment should be
---! @details Initiate a status segment write when requested.
---! This process also initiates handling critical
---! @param clock_sys Take action on positive edge.
---! @param reset Reset to initial state.
update_status: process (clock_sys, reset)
begin
  if reset = '0' then
    log_status_follower <= '0';
    write_status     <= '0';
    status_written_follower <= '0';
    crit_event_write  <= '0';
    crit_event_follower <= '0';
    crit_event_written_follower <= '0';
  elsif clock_sys'event and clock_sys = '1' then
    if (clk_enable) = '1' then
      if (status_written_follower /= status_written) then
        status_written_follower <= status_written;
        if(status_written = '1') then
          write_status     <= '0';
        end if;
      end if;
      elsif (log_status_follower /= log_status) then
        log_status_follower <= log_status;
        if log_status = '1' then
          write_status     <= '1';
        end if;
      end if;
      if (crit_event_written_follower /= crit_event_written) then
        crit_event_written_follower <= crit_event_written;
        if(crit_event_written = '1') then
          crit_event_write  <= '0';
        end if;
      elsif (crit_event_follower /= crit_event) then
        crit_event_follower <= crit_event;
      end if;
    end if;
  end if;
end process;
if (crit_event = '1') then
    crit_event_write <= '1';
end if;

end if;

end if;

end process update_status;

---
---
---! @brief Handle events trigger.
---! Do not deassert interrupt until serviced.
---!
---! @details
---!
---! @param clock_sys Take action on positive edge.
---! @param reset Reset to initial state.
---

handle_events: process (clock_sys, reset)
begin
    if reset = '0' then
        events_data_write <= '0';
        events_written_follower <= '0';
        log_events_follower <= '0';
    elsif (clock_sys ' event and clock_sys = '1') then
        if (clk_enable) = '1' then
            if (events_written_follower /= events_written) then
                events_written_follower <= events_written;
                if (events_written = '1') then
                    events_data_write <= '0';
                end if;
            elsif (log_events_follower /= log_events) then
                log_events_follower <= log_events;
            end if;
        end if;
    end if;
end process handle_events;
if log_events = '1' then
    events_data_write <= '1';
end if;
end if;
end process handle_events;

−−
−−
−−
−−

−−! @brief Receive IMU and Audio data. Place into circular buffers. After
−−! the data is placed in circ buffers, signal state machine to
−−! process that data out of circ buffer and into block.
−−! @details No circular buffer overrun detection has been coded yet. That is a to do.
−−!
−−!
−−!
−−!

−−! @param clock_sys Take action on positive edge.
−−! @param reset Reset to initial state.
−−
−−

audio_sample: process (clock_sys, reset)
begin
    if reset = '0' then
        audio_byte_count <= (others => '0');
        audio_written_follower <= '0';
        audio_data_write <= '0';
        gyro_data_write <= '0';
        gyro_written_follower <= '0';
        circ_buffer_wr_audio <= (others => '0');
        circ_buffer_wr_gyro <= (others => '0');
        circ_buffer_wr_accel <= (others => '0');
        circ_buffer_wr_mag <= (others => '0');
circ_buffer_wr_temp <= (others => '0');
accel_data_write <= '0';
accel_written_follower <= '0';
mag_data_write <= '0';
mag_written_follower <= '0';
temp_data_write <= '0';
temp_written_follower <= '0';
audio_data_process_request <= '0';
audio_data_process_request_follower <= '0';
audio_data_processed <= '0';
audio_data_processed_follower <= '0';
gyro_data_process_request <= '0';
gyro_data_process_request_follower <= '0';
gyro_data_processed <= '0';
gyro_data_processed_follower <= '0';
accel_data_process_request <= '0';
accel_data_process_request_follower <= '0';
accel_data_processed <= '0';
accel_data_processed_follower <= '0';
mag_data_process_request <= '0';
mag_data_process_request_follower <= '0';
mag_data_processed <= '0';
mag_data_processed_follower <= '0';
temp_data_process_request <= '0';
temp_data_process_request_follower <= '0';
temp_data_processed <= '0';
temp_data_processed_follower <= '0';
flashblock_circbuffer_wr_en_internal <= '0';
flashblock_circbuffer_buffer_wr <= (others => '0');
flashblock_circbuffer_sample_wr <= (others => '0');
flashblock_circbuffer_data_internal <= (others => '0');

sdram_empty_follower <= '0';
empty_serviced_follower <= '0';
empty_done <= '0';
start_follower <= '0';

elsif clock_sys'event and clock_sys = '1' then
  if (clk_enable) = '1' then
    flashblock_circbuffer_wr_en_internal <= '0';
    if (audio_data_processed_follower = '1' and audio_data_processed = '1') then
      audio_data_processed <= '0';
    end if;
    if (gyro_data_processed_follower = '1' and gyro_data_processed = '1') then
      gyro_data_processed <= '0';
    end if;
    if (accel_data_processed_follower = '1' and accel_data_processed = '1') then
      accel_data_processed <= '0';
    end if;
    if (mag_data_processed_follower = '1' and mag_data_processed = '1') then
      mag_data_processed <= '0';
    end if;
    if (temp_data_processed_follower = '1' and temp_data_processed = '1') then
      temp_data_processed <= '0';
    end if;
  end if;
end if;
if (audio_data_rdy_follower /= audio_data_rdy) then
    audio_data_rdy_follower <= audio_data_rdy;

if (audio_data_rdy = '1') then
    audio_data_process_request <= '1';
    audo_sample_fpga_time <= current_fpga_time;
end if;

elsif (audio_data_processed_follower /= audio_data_processed) then
    audio_data_processed_follower <= audio_data_processed;
    if (audio_data_processed = '1') then
        audio_data_process_request <= '0';
        audio_data_write <= '1';
    end if;
end if;

elsif (audio_written_follower /= audio_written) then
    audio_written_follower <= audio_written;
    if (audio_written = '1') then
        if (circ_buffer_rd_audio = circ_buffer_wr_audio) then
            audio_data_write <= '0';
        end if;
    end if;
end if;

if (gyro_data_rdy_follower /= gyro_data_rdy) then
    gyro_data_rdy_follower <= gyro_data_rdy;

if (gyro_data_rdy = '1') then
    gyro_data_process_request <= '1';
end if;

elsif (gyro_data_processed_follower /= gyro_data_processed) then
    gyro_data_processed_follower <= gyro_data_processed;
    if (gyro_data_processed = '1') then
        gyro_data_process_request <= '0';
        gyro_data_write <= '1';
    end if;
end if;

elsif (gyro_written_follower /= gyro_written) then
    gyro_written_follower <= gyro_written;
if (gyro_written = '1') then
  if (circ_buffer_rd_gyro = circ_buffer_wr_gyro) then
    gyro_data_write <= '0';
  end if;
end if;

if (accel_data_rdy_follower /= accel_data_rdy) then
  accel_data_rdy_follower <= accel_data_rdy;
  if (accel_data_rdy = '1') then
    accel_data_process_request <= '1';
    end if;
  elsif (accel_data_processed_follower /= accel_data_processed) then
    accel_data_processed_follower <= accel_data_processed;
    if (accel_data_processed = '1') then
      accel_data_process_request <= '0';
      accel_data_write <= '1';
      end if;
    elsif (accel_written_follower /= accel_written) then
      accel_written_follower <= accel_written;
      if (accel_written = '1') then
        if (circ_buffer_rd_accel = circ_buffer_wr_accel) then
          accel_data_write <= '0';
        end if;
      end if;
    end if;
  end if;
end if;

if (mag_data_rdy_follower /= mag_data_rdy) then
  mag_data_rdy_follower <= mag_data_rdy;
  if (mag_data_rdy = '1') then
    mag_data_process_request <= '1';
    end if;
  elsif (mag_data_processed_follower /= mag_data_processed) then
    mag_data_processed_follower <= mag_data_processed;
    if (mag_data_processed = '1') then
      mag_data_process_request <= '0';
    end if;
  end if;
end if;
mag_data_write <= '1';
end if;

elsif (mag_written_follower /= mag_written) then
  mag_written_follower <= mag_written;
  if (mag_written = '1') then
    if (circ_buffer_rd_mag = circ_buffer_wr_mag) then
      mag_data_write <= '0';
    end if;
  end if;
end if;

if (temp_data_rdy_follower /= temp_data_rdy) then
  temp_data_rdy_follower <= temp_data_rdy;
  if (temp_data_rdy = '1') then
    temp_data_process_request <= '1';
    end if;
  elsif (temp_data_processed_follower /= temp_data_processed) then
    temp_data_processed_follower <= temp_data_processed;
    if (temp_data_processed = '1') then
      temp_data_process_request <= '0';
      temp_data_write <= '1';
    end if;
  elsif (temp_written_follower /= temp_written) then
    temp_written_follower <= temp_written;
    if (temp_written = '1') then
      if (circ_buffer_rd_temp = circ_buffer_wr_temp) then
        temp_data_write <= '0';
      end if;
    end if;
  end if;

if (sdram_empty_follower /= sdram_empty) then
  sdram_empty_follower <= sdram_empty;
  if (sdram_empty = '1') then
    empty_done <= '1';
  end if;
elsif (empty_serviced_follower /= empty_serviced) then
  empty_serviced_follower <= empty_serviced;
if( empty_serviced = '1') then
    empty_done <= '0';
end if;
end if;

if ( audio_data_process_request_follower /=
    audio_data_process_request ) then
    audio_data_process_request_follower <=
        audio_data_process_request;
    if ( audio_data_process_request = '1') then
        circ_buffer_wr_audio <= circ_buffer_wr_audio + 1;
        flashblock_circbuffer_wr_en_internal <= '1';
        flashblock_circbuffer_buffer_wr <=
            circbuffer_audio_select;
        flashblock_circbuffer_sample_wr <= std_logic_vector( circ_buffer_wr_audio);
        flashblock_circbuffer_data_internal <=
            std_logic_vector(resize(unsigned(audio_data),
                flashblock_circbuffer_data_internal'length));
        audio_data_processed <= '1';
    end if;
end if;

elsif ( gyro_data_process_request_follower /=
    gyro_data_process_request ) then
    gyro_data_process_request_follower <=
        gyro_data_process_request;
    if ( gyro_data_process_request = '1') then
        circ_buffer_wr_gyro <= circ_buffer_wr_gyro + 1;
        flashblock_circbuffer_wr_en_internal <= '1';
        flashblock_circbuffer_buffer_wr <=
            circbuffer_gyro_select;
        flashblock_circbuffer_sample_wr <=
            std_logic_vector(circ_buffer_wr_gyro);
        flashblock_circbuffer_data_internal <=
            std_logic_vector(resize(unsigned(gyro_data_x &
                gyro_data_y & gyro_data_z)
                ,flashblock_circbuffer_data_internal'length));
        gyro_data_processed <= '1';
    end if;
end if;
elsif (accel_data_process_request_follower /= accel_data_process_request) then
  accel_data_process_request_follower <= accel_data_process_request;
  if (accel_data_process_request = '1') then
    circ_buffer_wr_accel <= circ_buffer_wr_accel + 1;
    flashblock_circbuffer_wr_en_internal <= '1';
    flashblock_circbuffer_buffer_wr <=
      circbuffer_accel_select;
    flashblock_circbuffer_sample_wr <=
      std_logic_vector(circ_buffer_wr_accel);
    flashblock_circbuffer_data_internal <=
      std_logic_vector(resize(unsigned(accel_data_x &
        accel_data_y & accel_data_z),
        flashblock_circbuffer_data_internal' length));
    accel_data_processed <= '1' ;
  end if;
elsif (mag_data_process_request_follower /= mag_data_process_request) then
  mag_data_process_request_follower <= mag_data_process_request;
  if (mag_data_process_request = '1') then
    circ_buffer_wr_mag <= circ_buffer_wr_mag + 1;
    flashblock_circbuffer_wr_en_internal <= '1';
    flashblock_circbuffer_buffer_wr <=
      circbuffer_mag_select;
    flashblock_circbuffer_sample_wr <=
      std_logic_vector(circ_buffer_wr_mag);
    flashblock_circbuffer_data_internal <=
      std_logic_vector(resize(unsigned(mag_data_x &
        mag_data_y & mag_data_z),
        flashblock_circbuffer_data_internal' length));
    mag_data_processed <= '1' ;
  end if;
elsif (temp_data_process_request_follower /= temp_data_process_request) then
  temp_data_process_request_follower <= temp_data_process_request;
if (temp_data_process_request = '1') then
  circ_buffer_wr_temp <= circ_buffer_wr_temp + 1;
  flashblock_circbuffer_wr_en_internal <= '1';
  flashblock_circbuffer_buffer_wr <=
    circbuffer_temp_select;
  flashblock_circbuffer_sample_wr <=
    std_logic_vector(circ_buffer_wr_temp);
  flashblock_circbuffer_data_internal <=
    std_logic_vector(resize(unsigned(temp_data)
    ,flashblock_circbuffer_data_internal 'length'));
  temp_data_processed <= '1' ;
end if;
end if;
end process audio_sample ;

—
—
—@brief Track new GPS data. GPS data is read from ram in
  GPS entity.
—!
  This data exists in double buffers with the most
  recently updated
—!
  buffer signalled by either a tmbank or posbank
toggle.
—!
  This process watches for those toggles and then
  signals for state
—!
  machines to process the data out of those
  buffers and into block.
—@details
—!
—@param clock_sys Take action on positive edge.
—!
—@param reset Reset to initial state.
—
—
read_in_GPS_data: process (clock_sys, reset)
begin
  if (reset = '0') then
    gps_pos_data_write <= '0' ;
    gps_pos_written_follower <= '0' ;
  end if;
end process;
elsif (clock_sys 'event and clock_sys = '1') then
  if (clk_enable) = '1' then
    if (gps_time_written_follower /= gps_time_written) then
      gps_time_written_follower <= gps_time_written;
      if (gps_time_written = '1') then
        gps_time_data_write <= '0';
      end if;
      --If tmbank changes the data has been refreshed.
    elsif tmbank_follower /= tmbank then
      tmbank_follower <= tmbank;
      gps_time_data_write <= '1';
    end if;
    if (gps_pos_written_follower /= gps_pos_written) then
      gps_pos_written_follower <= gps_pos_written;
      if (gps_pos_written = '1') then
        gps_pos_data_write <= '0';
      end if;
      --If posbank changes the data has been refreshed.
    elsif posbank_follower /= posbank then
      posbank_follower <= posbank;
      gps_pos_data_write <= '1';
    end if;
  end if;
end if;
end process read_in_GPS_data;
end behavior;
APPENDIX B

MATLAB CODE
function [a,sequence_number,audio_segment_stream
  gyro_segment_stream ...
  accel_segment_stream,mag_segment_stream] = ...
  search_flashblock(filename,length_blocks)

search_flashblock Parse data structures from the
  flashblock assembled
% Output
  a = Array of all read data as unit8.
  sequence_number = Array of processed block numbers.
  audio_segment_stream = Continuous audio stream. 16
    bit signed.
  gyro_segment_stream = Continuous Gyroscope Data. 32
    bits. XYZ.
  accel_segment_stream = Continuous Accelerometer
    Data. 32 bits. XYZ.
  mag_segment_stream = Continuous Magnetometer Data.
    32 bits. XYZ.
% Input
  filename = Filename of sdcard dump.
  length_blocks = Number of blocks to read from file.

fid = fopen(filename);
a = uint8(fread(fid,(512*length_blocks),'uint8'));

% Constants pulled in from flashblock.vhd.
BLOCK_SEQNOBYTES = 4;
BLOCK_SEQNOBYTES = 4;
BLOCK_SIZE = 512;
SEG_TRAILER_SIZE = 2;
AUDIO_WORD_BYTES = 2;

IMU_AXIS_WORD_LENGTH_BYTES = 2;
IMU_GYRO_SEG_BYTES = 3*IMU_AXIS_WORD_LENGTH_BYTES;
IMU_ACCEL_SEG_BYTES = 3*IMU_AXIS_WORD_LENGTH_BYTES;
IMU_MAG_SEG_BYTES = 3*IMU_AXIS_WORD_LENGTH_BYTES;

% All the defined segment identifiers.
% Taken from flashblock.vhd.
PADDING_BYTE = uint8 (sscanf('00 ', '%x'));

BLOCK_SEG_UNUSED = uint8 (sscanf('01 ', '%x'));
BLOCK_SEG_STATUS = uint8 (sscanf('02 ', '%x'));
BLOCK_SEG_GPS_TIME_MARK = uint8 (sscanf('03 ', '%x'));
BLOCK_SEG_GPS_POSITION = uint8 (sscanf('04 ', '%x'));
BLOCK_SEG_IMU_GYRO = uint8 (sscanf('05 ', '%x'));
BLOCK_SEG_IMU_ACCEL = uint8 (sscanf('06 ', '%x'));
BLOCK_SEG_IMU_MAG = uint8 (sscanf('07 ', '%x'));
BLOCK_SEG_IMU_TEMP = uint8 (sscanf('0A ', '%x'));
BLOCK_SEG_EVENT = uint8 (sscanf('0B ', '%x'));
BLOCKSEG_AUDIO = uint8 (sscanf('08 ', '%x'));

sequence_number = [];  % Used to hold an audio segment’s data.
audio_segment_block = [];
% Used to hold an the entire audio stream
audio_segment_stream = [];
% Cell array to hold inner block segments (audio_segment_blocks.)
audio_cell = {};
k=1;
gyro_segment_stream = [];
accel_segment_stream = [];
mag_segment_stream = [];

while (k < length(a))
    if (mod(k-1,BLOCK_SIZE)==0)
        segment = uint32 (a(k+BLOCK_SEQNO_BYTES-1:1:k));
        segment = segment(1)*(256^3) + segment(2)*(256^2) +
        segment(3)*256 + segment(4);
        sequence_number = [sequence_number segment];
        % Jump to end of block. Process in reverse.
        block_start = k;
\( k = k + 511; \)

```matlab
while \((k \neq \text{block\_start} + \text{BLOCK\_SEQNO\_BYTES} - 1)\)

%Jump the segment trailer. No data.
%Normal for a padding segment of zero length.
if \((a(k) == 0)\)
    segment\_length = a(k);
    %Jump the segment trailer. No data.
    k = k - \text{double}(\text{segment\_length}) - \text{SEG\_TRAILER\_SIZE};
else
    segment\_length = a(k);
    if \((a(k-1) == \text{BLOCK\_SEG\_UNUSED})\)
        %Jump padding
        k = k - \text{double}(\text{segment\_length}) - \text{SEG\_TRAILER\_SIZE};
    elseif \((a(k-1) == \text{BLOCK\_SEG\_IMU\_GYRO})\)
        k = k - \text{SEG\_TRAILER\_SIZE};
        for \(l = 1:\text{IMU\_GYRO\_SEG\_BYTES}\)
            gyro\_segment\_stream = [ gyro\_segment\_stream a(k) ];
            k = k - 1;
        end
    elseif \((a(k-1) == \text{BLOCK\_SEG\_IMU\_ACCEL})\)
        k = k - \text{SEG\_TRAILER\_SIZE};
        for \(l = 1:\text{IMU\_ACCEL\_SEG\_BYTES}\)
            accel\_segment\_stream = [ accel\_segment\_stream a(k) ];
            k = k - 1;
        end
    elseif \((a(k-1) == \text{BLOCK\_SEG\_IMU\_MAG})\)
        k = k - \text{SEG\_TRAILER\_SIZE};
        for \(l = 1:\text{IMU\_MAG\_SEG\_BYTES}\)
            mag\_segment\_stream = [ mag\_segment\_stream a(k) ];
            k = k - 1;
        end
    elseif \((a(k-1) == \text{BLOCK\_SEG\_AUDIO})\)
        end\_sample = k - \text{SEG\_TRAILER\_SIZE};
        begin\_sample = k - \text{SEG\_TRAILER\_SIZE} - \text{double}(\text{segment\_length} - 1);
        for \(m = \text{begin\_sample}:\text{double}(\text{AUDIO\_WORD\_BYTES}):\text{end\_sample}\)
            segment = int32(a(m+\text{AUDIO\_WORD\_BYTES} - 1:m));
    end
end
```
segment = segment(1)*(256^1) + segment(2);
if (bitand(segment,2^15)==2^15)
  %Sign bit set. Convert to appropriate negative.
  segment = segment - 2^16;
end
audio_segment_block = [
  audio_segment_block segment];
end
k = begin_sample -1;
audio_cell = [audio_cell {
  audio_segment_block }];
audio_segment_block = [];
end
end
end
end
%Jump to start of next block.
k = k+509;
%Flip the segment ordering as they are processed from the end of the block.
audio_segment_stream = [audio_segment_stream audio_cell{
  end:-1:1}];
audio_cell = {};
end
end
%Data sequencing check.
if isequal(sequence_number,[1:1:length_blocks])
  sprintf(’There are no missing blocks’)
end
fclose(fid);
%IMU_read  Parse IMU multiwords returned by search_flashblock.m

% Output
% Sample vectors for each of the 9 IMU axes. Accelerometer/Magnetometer and Gyroscope.
% Input
% G and XL and MAG. IMU 6 byte XYZ values.
% Each axis is returned by search_flashblock.m Big Endian X(2) Y(2) Z(2)

%Set the full scales in which the IMU was recording.
FULL_SCALE_G = 245;
FULL_SCALE_XL = 2;
FULL_SCALE_M = 4;

g_x = [];
g_y = [];
g_z = [];
xl_x = [];
xl_y = [];
xl_z = [];
mag_x = [];
mag_y = [];
mag_z = [];

k=1;

while (k < length(G))
    segment_x = int32(G(k:k+1));
    segment_y = int32(G(k+2:k+3));
    segment_z = int32(G(k+4:k+5));
    segment_x = segment_x(1)*(2^15) + segment_x(2);
    if (bitand(segment_x,2^15)==2^15)
        %Sign bit set. Convert to appropriate negative.
        segment_x = segment_x - 2^16;
    end
    segment_y = segment_y(1)*(2^15) + segment_y(2);
    if (bitand(segment_y,2^15)==2^15)
        segment_y = segment_y - 2^16;
    end
end
464

\[ \text{segment}_z = \text{segment}_z(1) \times (256^1) + \text{segment}_z(2); \]

\[ \text{if } (\text{bitand}(\text{segment}_z, 2^{15}) == 2^{15}) \]

\[ \text{segment}_z = \text{segment}_z - 2^{16}; \]

\[ \text{end} \]

\[ \text{g}_x = [\text{g}_x \text{ double}(\text{segment}_x)]; \]
\[ \text{g}_y = [\text{g}_y \text{ double}(\text{segment}_y)]; \]
\[ \text{g}_z = [\text{g}_z \text{ double}(\text{segment}_z)]; \]

\[ k = k + 6; \]

\[ \text{end} \]

\[ k = 1; \]

\[ \textbf{while} (k < \text{length}(\text{XL})) \]

\[ \text{segment}_x = \text{int32}(\text{XL}(k:k+1)); \]
\[ \text{segment}_y = \text{int32}(\text{XL}(k+2:k+3)); \]
\[ \text{segment}_z = \text{int32}(\text{XL}(k+4:k+5)); \]
\[ \text{segment}_x = \text{segment}_x(1) \times (256^1) + \text{segment}_x(2); \]

\[ \text{if } (\text{bitand}(\text{segment}_x, 2^{15}) == 2^{15}) \]

\[ \% \text{Sign bit set. Convert to appropriate negative.} \]
\[ \text{segment}_x = \text{segment}_x - 2^{16}; \]

\[ \text{end} \]

\[ \text{segment}_y = \text{segment}_y(1) \times (256^1) + \text{segment}_y(2); \]
\[ \text{if } (\text{bitand}(\text{segment}_y, 2^{15}) == 2^{15}) \]
\[ \text{segment}_y = \text{segment}_y - 2^{16}; \]

\[ \text{end} \]

\[ \text{segment}_z = \text{segment}_z(1) \times (256^1) + \text{segment}_z(2); \]
\[ \text{if } (\text{bitand}(\text{segment}_z, 2^{15}) == 2^{15}) \]
\[ \text{segment}_z = \text{segment}_z - 2^{16}; \]

\[ \text{end} \]

\[ \text{x}_l = [\text{x}_l \text{ double}(\text{segment}_x)]; \]
\[ \text{x}_l = [\text{x}_l \text{ double}(\text{segment}_y)]; \]
\[ \text{x}_l = [\text{x}_l \text{ double}(\text{segment}_z)]; \]

\[ k = k + 6; \]
end

k=1;
while(k < length(MAG))

    segment_x = int32(MAG(k:k+1));
    segment_y = int32(MAG(k+2:k+3));
    segment_z = int32(MAG(k+4:k+5));
    segment_x = segment_x(1)*(256^1) + segment_x(2);
    if (bitand(segment_x,2^15)== 2^15)
        \%Sign bit set. Convert to appropriate negative.
        segment_x = segment_x - 2^16;
    end
    segment_y = segment_y(1)*(256^1) + segment_y(2);
    if (bitand(segment_y,2^15)== 2^15)
        segment_y = segment_y - 2^16;
    end
    segment_z = segment_z(1)*(256^1) + segment_z(2);
    if (bitand(segment_z,2^15)== 2^15)
        segment_z = segment_z - 2^16;
    end

    mag_x = [mag_x double(segment_x)];
    mag_y = [mag_y double(segment_y)];
    mag_z = [mag_z double(segment_z)];

    k=k+6;

end

g_x = g_x * (1/2^15) * FULL_SCALE_G;
g_y = g_y * (1/2^15) * FULL_SCALE_G;
g_z = g_z * (1/2^15) * FULL_SCALE_G;
xl_x = xl_x * (1/2^15) * FULL_SCALE_XL;
xl_y = xl_y * (1/2^15) * FULL_SCALE_XL;
xl_z = xl_z * (1/2^15) * FULL_SCALE_XL;
mag_x = mag_x * (1/2^15) * FULL_SCALE_M;
mag_y = mag_y * (1/2^15) * FULL_SCALE_M;
mag_z = mag_z * (1/2^15) * FULL_SCALE_M;