Robert Wilcox
Mohamed Gnedi
Ibrahim Binmahfood
ECE 540, Kravitz
12/07/2023

Final Project Report

In this project, our team created an interactive basketball game, leveraging the capabilities of the RealDigital BooleanBoard. The essence of the game is straightforward yet engaging: players are tasked with shooting a ball into a hoop. The hoop can be positioned at various locations within the play area, introducing variability and challenge to the game. The given contribution of the project is available on the GitHub project. Mainly Ibrahim was responsible for the detection mechanism, Robert was responsible for the launching mechanism, and Mohamed was responsible for the joystick mechanism.

RTL CODE

Detection mechanism

The **pmod_D_ctrlr.sv** module is a System Verilog Wishbone peripheral rtl module that interfaces the Wishbone bus with the PMOD D expansion connector. Since Xilinx's Vivado doesn't support System Verilog rtl modules directly, a Verilog wrapper file is required. The Verilog wrapper file, **pmod_D_ctrlr_wrapper.v**, instantiates the rtl module **pmod_D_ctrlr.sv**. The **pmod_D_ctrlr.sv** rtl module includes two registers *state* and *echo_pulse*. The *state* register is read if an obstacle is detected or not. The *echo_pulse* register is read to debug if there are any pulses read back from the sensor. These two are accessible in the PMODD_STATE register at 0x80001640. These two registers are 1 bit in size. So only bits [1:0] are used in the PMODD_STATE register. The PMOD D expansion connector, has the following pin mappings for the Boolean board:

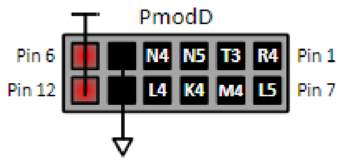


Fig. 1 PMOD D Connector

From these pins, only *N4* and *L4* were used for the HC-SR04 sensor. The HC-SR04 sensor requires a 10 μs pulse input to the Trigger pin. The **ultrasonic_sensor.sv** rtl module which accounts for the trigger pulse generation, echo back pulse, sample rate, and OR filter. The **ultrasonic_sensor.sv** originally by Josh Macfie, developed this module in Verilog. I ported his Verilog module into a System Verilog equivalent module.

This module requires a 64 MHz clock signal as its input. The counter register is 26 bits wide since $2^{26} = 64Mb = 64MHz$. The trigger pulse generation logic are a group of bits sliced out of this *counter* register. Only when the *counter* register has bits [24:10] low, the trigger pulse is output to the trigger pin of the HC-SR04 sensor. The echo back pulse from the sensor requires logic to determine the distance at which an obstacle is located at. Whenever the echo pulse back from the sensor is high, a *echo_pulse_cnt* register increments. Also, this echo pulse back value is held by a *prev_echo_pulse* register. These two echo pulse registers are checked if the current echo pulse back is low and the previous echo pulse is high, then the *echo_pulse_cnt* is held by the *output_cnt* register. The *echo_pulse_cnt* is then reset to 0.

The *output_cnt* register determines the distance an obstacle is located in front of the sensor. This distance is limited to ~10 cm. Whenever the *output_cnt* register is greater than the distance limit, the *status_filter* is shifted 1 bit to the left and bit 0 is set high. If that is not the case, the *status_filter* is shifted 1 bit to the left and bit 0 is set low. This *status_filter* register is used to account for all occurrences of an obstacle detected within the distance limit. Finally the *status_filter* is bitwise ORed to output a true obstacle detected signal.

Launching mechanism

The **servo_controller.v** module is a Verilog implementation designed for FPGA use, primarily to control a servo motor via a Pulse Width Modulation (PWM) output. This module is distinguished by its integration with the Wishbone bus interface, a standard bus architecture in embedded systems, facilitating communication with other system components. The core feature of the module is the generation of a PWM signal, which is controlled by the **pwm_width_reg**, a 32-bit register that dictates the PWM pulse width. The PWM signal itself is produced by comparing a 32-bit counter with the **pwm_width_reg**; the counter increments with each clock cycle and resets after 2,000,000 cycles, thereby defining the PWM frequency.

In terms of interface, the module connects to the Wishbone bus using signals such as **wb_clk** (clock), **wb_rst** (reset), **wb_cyc** (cycle valid), among others, allowing for read and write operations to the PWM width register. Notably, the module features a synchronized reset logic, essential for ensuring stability and preventing glitches by aligning the external reset signal with the Wishbone clock. Moreover, the module handles read and write operations through the Wishbone interface. During write operations, when **wb_we**, **wb_cyc**, and **wb_stb** are active, the **pwm_width_reg** is updated. Conversely, during read operations, the current value in **pwm_width_reg** is output to **wb_dat_o**.

Furthermore, the module is equipped with debug outputs, namely **debug_counter** and **debug_pwm_width_reg**, which provide real-time insights into the internal state of the module, particularly useful during the development and testing phases. The module's design reflects a focus on precise control of servo motors, a critical requirement in many FPGA-based applications, such as robotics and control systems. The choice of a 2,000,000-cycle reset for the PWM generation counter suggests an adaptation to specific servo requirements and the operating frequency of the Wishbone bus. Overall, the **servo_controller** module exemplifies an efficient and scalable solution for servo motor control within a larger FPGA-based system, highlighting the adaptability and modular design capabilities inherent to FPGA technology.

The **tb_servo_controller.v** module is a testbench designed for testing and verifying the **servo_controller** module, a crucial component in FPGA-based systems for servo motor control. This testbench is structured to simulate a range of operational scenarios that the **servo_controller** might encounter in real-world applications. The primary focus of this testbench is to analyze the behavior of the PWM (Pulse Width Modulation) output under various input conditions, thereby ensuring the module's reliability and correctness.

Key to the testbench's functionality is the simulation of input signals that the **servo_controller** would typically receive. These include the Wishbone clock (**wb_clk**), reset signal (**wb_rst**), cycle valid (**wb_cyc**), strobe (**wb_stb**), write enable (**wb_we**), byte select (**wb_sel**), address (**wb_adr**), and data input (**wb_dat_i**). The testbench initializes and manipulates these signals to emulate different states and transitions, providing a comprehensive environment for testing the **servo_controller**.

Within the testbench, the **servo_controller** is instantiated as the Unit Under Test (UUT). The UUT's inputs and outputs are directly connected to the testbench, allowing for real-time observation and interaction with the **servo_controller**. The test stimulus provided by the testbench commences with the initialization of inputs, followed by de-asserting the reset signal to start the simulation. The testbench then iteratively modifies the PWM width (**wb_dat_i**), observing the response of the UUT to these changes. It also checks for acknowledgments (**wb_ack**) from the UUT after each modification, a critical step in ensuring proper communication and control flow within the system.

Throughout the simulation, the testbench provides real-time feedback on the settings and responses of the UUT. It prints out the high and low durations of the PWM signal, offering a quantitative view of the module's performance.

Upon completing the tests, the testbench concludes the simulation with a notification statement.

In conclusion, the **tb_servo_controller** testbench is a meticulously engineered module that played an indispensable role in the development cycle of the **servo_controller**. By simulating a range of input conditions and analyzing the PWM output, the testbench ensures that the **servo_controller** operates reliably and meets the necessary performance criteria, thereby affirming its suitability for deployment in FPGA-based servo control systems.

The **dc_motor_controller.v** module, designed for FPGA implementation, is a Verilog module developed to manage DC motor operations via Pulse Width Modulation (PWM). Structurally analogous to the servo_controller, this module is specifically optimized for DC motor control, demonstrating the versatility of FPGA applications in various motor control scenarios. It employs the Wishbone bus interface, a prevalent choice in embedded systems, to ensure seamless communication with other system components.

A notable feature of the dc_motor_controller is its ability to generate a PWM signal on dc_pwm_out. This signal is crucial in controlling the speed and direction of a DC motor. The duty cycle of the PWM, which is instrumental in determining the motor's operational characteristics, is governed by the pwm_width_reg. This register stores the width of the PWM pulse and is a key element in modulating the motor's behavior.

The module's integration with the Wishbone bus is facilitated through standard interface signals such as **wb_clk** (clock), **wb_rst** (reset), **wb_cyc** (cycle valid), **wb_stb** (strobe), **wb_we** (write enable), **wb_sel** (byte select), **wb_adr** (address), **wb_dat_i** (data input), **wb_ack** (acknowledge), and **wb_dat_o** (data output). This integration is essential for the module's interaction with other components in the system, providing the necessary interface for configuring and monitoring the PWM width register.

In the PWM generation logic, a 32-bit **counter** is employed. This counter increments with each clock cycle and resets upon reaching the **PWM_MAX_COUNT**, which is set to 50,000. The module modulates the duty cycle of the PWM signal by comparing the value of this counter with the **pwm_width_reg**, thereby controlling the motor's operational parameters.

The module's Wishbone interface logic plays a critical role in handling read and write operations. During write operations, when signals such as **wb_we**, **wb_cyc**, **wb_stb**, and **wb_ack** are active, the **pwm_width_reg** is updated. This update is based on the input data (**wb_dat_i**), interpreted as a percentage of the maximum PWM count. This percentage-based methodology enables intuitive control over the motor speed, making the module user-friendly and adaptable.

The synchronization and reset logic of the module is integral to its stability and reliability. This logic ensures that the external reset signal (**wb_rst**) is synchronized with the Wishbone clock (**wb_clk**), a critical feature in systems where multiple clock domains are present.

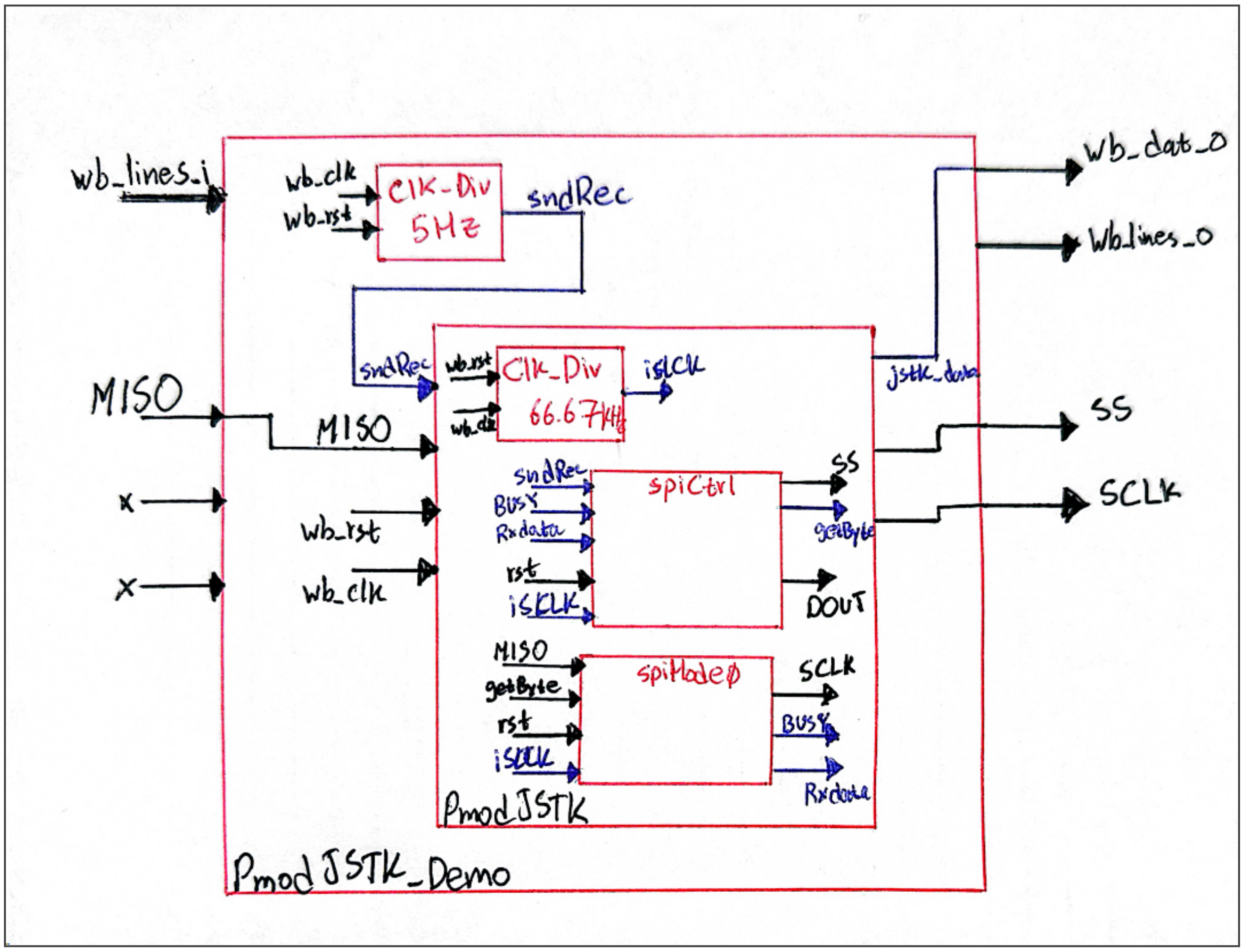
For debugging and monitoring, the module outputs **debug_counter** and **debug_pwm_width_reg**, which reflect the internal state of the **counter** and **pwm_width_reg**. These debug outputs are invaluable for real-time observation and troubleshooting, especially in the development and testing stages.

Lastly, the module's use of the parameter **PWM_MAX_COUNT** to define the maximum PWM cycle count exemplifies the flexibility and adaptability inherent in FPGA-based designs. This parameterization allows the module to be tailored to different DC motor specifications and system clock frequencies.

The **dc_motor_controller** module exemplifies a robust and flexible approach to DC motor control in FPGA environments. Its integration with the Wishbone bus and intuitive control mechanism positions it as a highly adaptable and user-friendly module for a broad spectrum of DC motor applications. The module's design highlights the adaptability and modular nature of FPGA technology, especially in contexts requiring precise and variable motor control capabilities.

Joystick mechanism

The joystick is implemented in the FPGA using the **PmodJSTK_Demo** module. This module utilizes two sub-modules: the 5Hz clock divider **(ClkDiv_5Hz)** and a joystick controller & communication module **(PmodJSTK).**

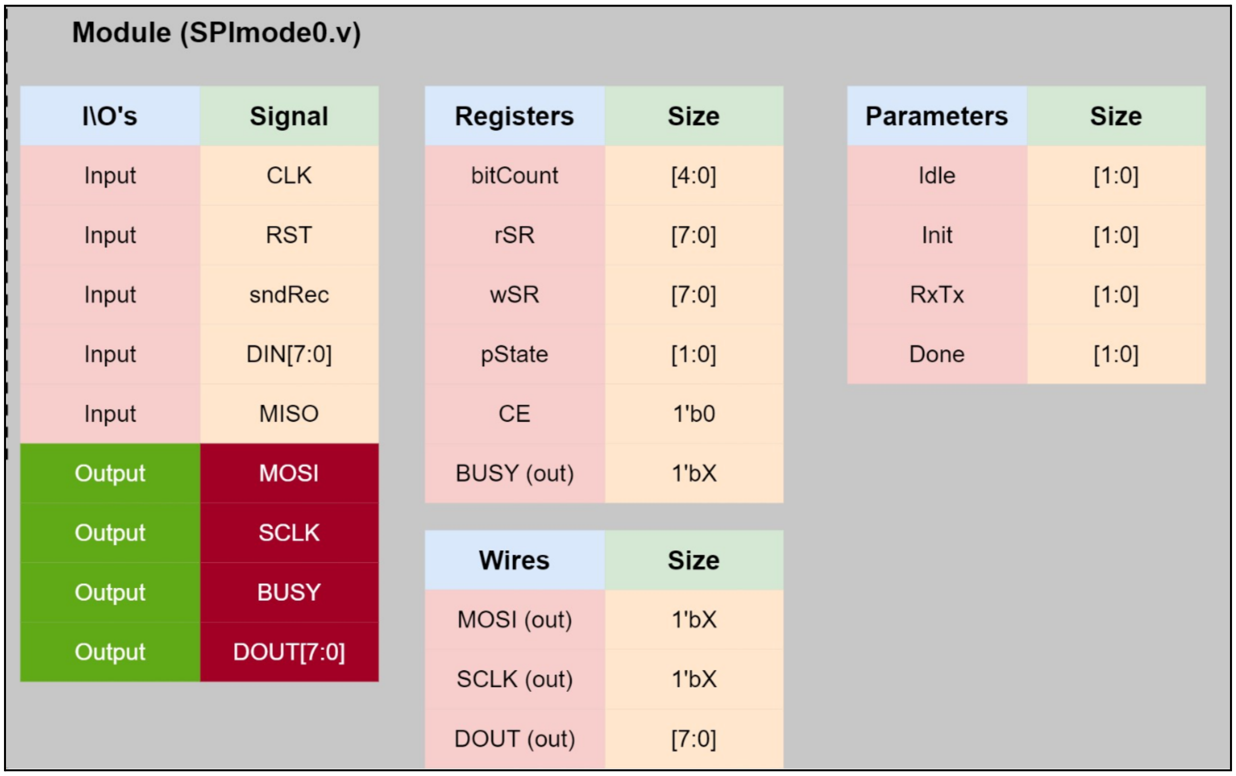


Block diagram with all the sub modules in (PmodJSTK_Demo.v)

- **ClkDiv_5Hz** is used to generate send & receive requests from the **PmodJSTK** module. Five requests per send.
- **PmodJSTK** provides the joystick XY-data to the Wishbone bus output, and it obtains the data by implementing three sub-modules:

1. **ClkDiv_66_67kHz**: 66.67kHz clock divider. It inputs the system clock from the Wishbone bus and outputs an internal serial clock (iSLCK) to manage the SPI communication.
2. **spiMode0**: This module provides the interface for sending and receiving data to and from the PmodJSTK; SPI mode 0 is used for communication. The master (Boolean Board), through the SPI controller and Wishbone bus, reads the data

on the MISO input on rising edges, and the slave (PmodJSTK) reads the data on the MOSI output on rising edges. The MOSI function is eliminated from this implementation to reduce the project's complexity, and the outputs provided are LEDs, which we have in abundance on our board.



Module (SPImode0.v)

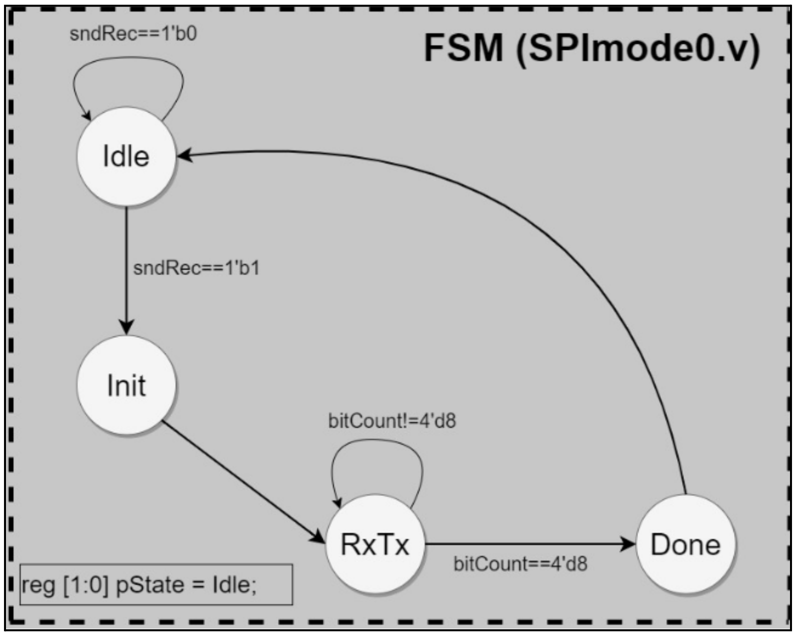
I\O's	Signal
Input	CLK
Input	RST
Input	sndRec
Input	DIN[7:0]
Input	MISO
Output	MOSI
Output	SCLK
Output	BUSY
Output	DOUT[7:0]

Registers	Size
bitCount	[4:0]
rSR	[7:0]
wSR	[7:0]
pState	[1:0]
CE	1'b0
BUSY (out)	1'bX

Wires	Size
MOSI (out)	1'bX
SCLK (out)	1'bX
DOUT (out)	[7:0]

Parameters	Size
Idle	[1:0]
Init	[1:0]
RxTx	[1:0]
Done	[1:0]

The I/Os, wires, registers, and parameters of the module are listed below:
The FSM is as follows:



Control flow:
* The **spiMode0** operates with 4 states for both the Read & Write operation.
* **Idle** is the default state where all the variables are set to 0 and the Read Shift Register maintains its value. The state changes to the **Init** state if a Send-Receive request is set to 1.
* **Init** state maintains the Read Shift Register value and keeps other variables set to zero except for the BUSY signal. The BUSY signal is used to inform other SPI controllers that it's reading or writing data from and to registers. Init transitions automatically to the **RxTx** state.
* **RxTx** appends a bit from the MISO line to the Read Shift Register if the serial clock is enabled. It also writes to the peripheral, but since the functionality isn't implemented, no outputs are detected in the joystick and the bitcount reaches 8 which leads to a transition to the **Done** state.
* **Done** disables the serial clock and maintains the BUSY states and loops back to the **Idle** state.
* The Read Shift Register data is always assigned to DOUT which will provide the read data to the SPI controller module.

3. **spiCtrl**: This component manages all data transfer requests, and manages the data bytes being sent to the PmodJSTK. However, we're not implementing the send part as it doesn't serve the purpose of the project.

   The I/Os, wires, registers, and parameters of the module are listed below:

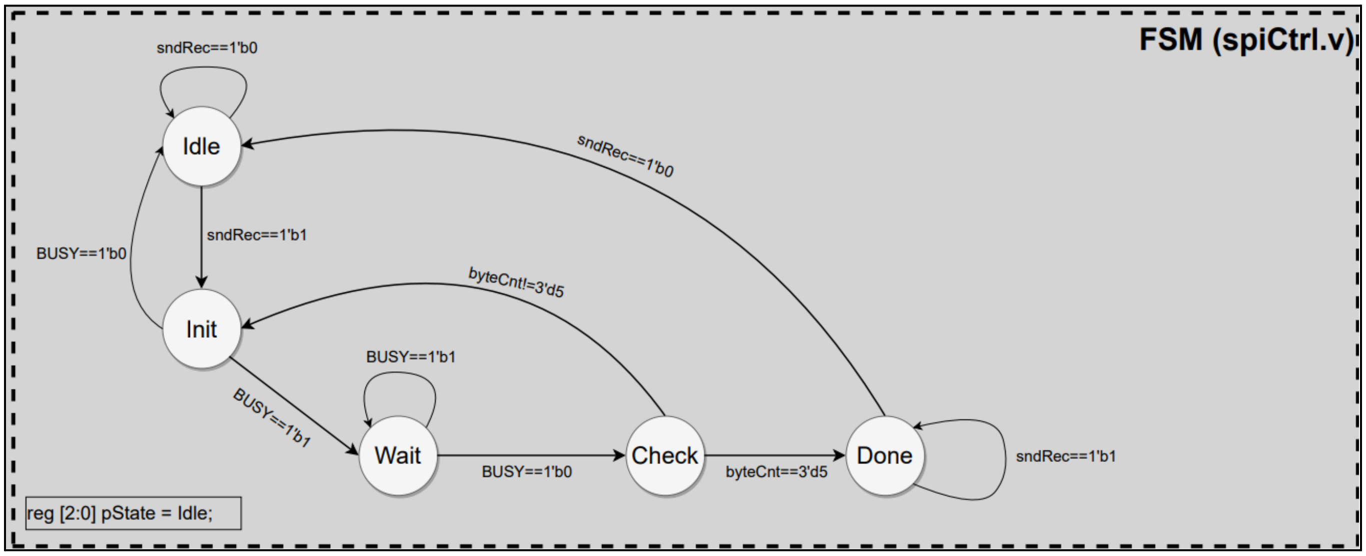
Module (spiCtrl.v)

I\O's	Signal
Input	CLK
Input	RST
Input	sndRec
Input	BUSY
Input	DIN[7:0]
Input	RxData[7:0]
Output	SS
Output	getByte
Output	sndData[7:0]
Output	DOUT[39:0]

Registers	Size
pState	[2:0]
byteCnt	[2:0]
tmpSR	[39:0]

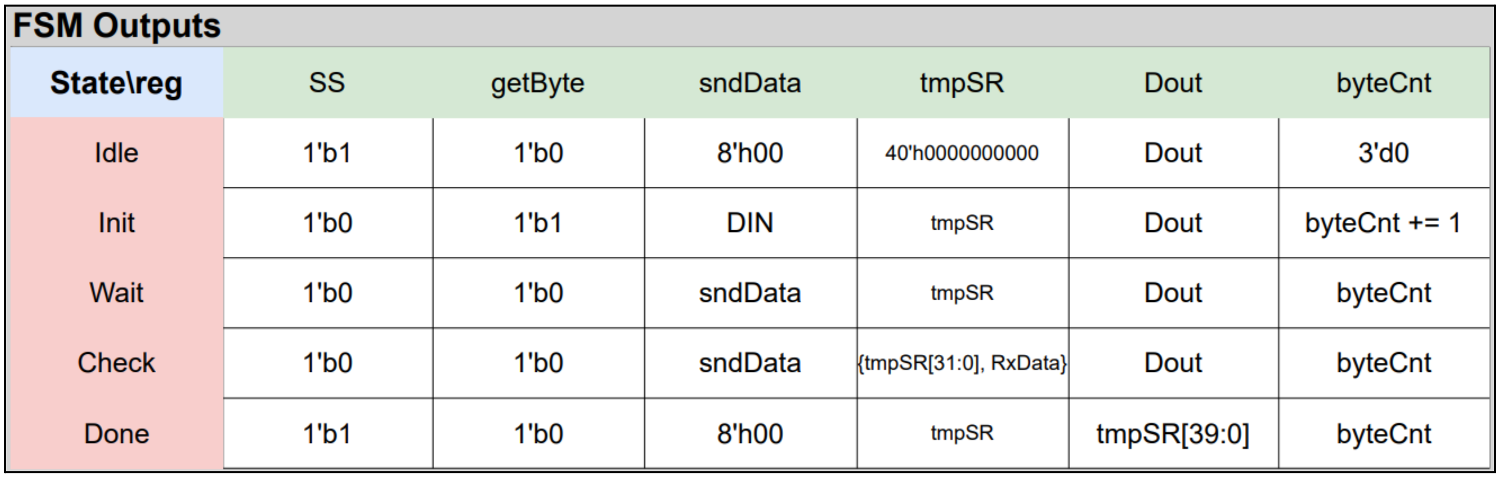
Parameters	Size
Idle	[2:0]
Init	[2:0]
Wait	[2:0]
Check	[2:0]
Done	[2:0]
byteEndVal	3'd5

The FSM is as follows:
Control flow:



* The **spiCtrl** operates with 5 states for both the Send & Receive operation from the **spiMode0** module and shares the received data with the top module to the Wishbone bus.
* **Idle** is the default state where all the variables are set to 0, no slave devices are selected, and output data is retained. It transitions to the **Init** state when Send-Receive request is set to 1.
* **Init** state starts with enabling the slave device and initializing a data transfer request with **spiMode0** module. It retains the temporary registers and output data. If it receives a BUSY signal from **spiMode0,** it transitions to the **Wait** state and increases the byte count. If it doesn't, it shifts back to the **Idle** state.
* **Wait** state keeps enabling the slave device and stops initializing a data transfer request with **spiMode0** module. It retains the temporary registers, output data, and byte count. If the BUSY input signal is set to 0, indicating that the data read is done, it transitions to the **Check** state, if not, it remains in the **Wait** state.
* **Check** state keeps enabling the slave device and with no data transfer request with **spiMode0** module. It retains byte count and output data. It also appends the new byte received from **spiMode0** in the temporary register, and it checks if it received the 5 bytes to move to the **Done** state. If it didn't get the 5 bytes, it'll move back to the **Init** state to re-initialize the data transfer request.
* **Done** disables the slave device and retains all values except for the output. The output gets the temporary register data which then is assigned to the Wishbone output. If Send-Receive is set to 0, it loops back to the **Idle** state. If Send-Receive is set to 1, it remains in **Done**.

Table with all the variable changes in the different states (spiCtrl.v)



FSM Outputs

State\reg	SS	getByte	sndData	tmpSR	Dout	byteCnt
Idle	1'b1	1'b0	8'h00	40'h0000000000	Dout	3'd0
Init	1'b0	1'b1	DIN	tmpSR	Dout	byteCnt += 1
Wait	1'b0	1'b0	sndData	tmpSR	Dout	byteCnt
Check	1'b0	1'b0	sndData	{tmpSR[31:0], RxData}	Dout	byteCnt
Done	1'b1	1'b0	8'h00	tmpSR	tmpSR[39:0]	byteCnt

In Summary, the joystick is connected to the board through the Pmod A port and it only makes use of three SPI lines (MISO, SCLK, SS) because we're not writing to the joystick registers. The 5Hz clock divider controls the frequency of the Send-Receive requests from the peripheral, and the 66.67kHz controls the serial clock to sync the incoming data. The spiMode0 processes the incoming bits from the MISO line and sends the received data as a byte to the spiCtrl which transfers it to the upper module and Wishbone bus.

C code

VGA Display

The VGA display peripheral functions and definitions are provided in **vga.h/vga.c**. Inside **vga.h**, the VGA display peripheral includes two main registers for setting the initial row and columns to display a character and the character to display at those coordinates. There are 6 functions that are implemented in **vga.c**. The *vga_display_welcome()*, sets coordinates and a welcome string input to the *_str2ascii()* function. This *_str2ascii()* function takes the input string and indexes it for certain characters available in the VGA peripheral character ROM. In between writing to the VGA display peripheral registers, a delay is set. This is to slow down writing the string of characters for the naked eye to see it as one whole word on the screen. The same behavior of *vga_display_welcome()* is present in *vga_display_miss()* and *vga_display_hit()*. However for *vga_display_score()*, the score is translated from an integer to its respective ASCII value.

Button Input

The buttons peripheral functions and definitions are provided in **bttn.h/bttn.c**. Inside **bttn.h**, the button peripheral includes three main registers for setting which buttons as input or floating. There are 3 functions that are implemented in **bttn.c**. The *bttn_init()*, sets the button peripheral as floating or input. The *bttn_read()*, reads from the button peripheral input register **RGPIO1_IN** and returns it as a 32 bit unsigned integer. The *bttn_write()*, writes to the button output register **RGPIO1_OUT**.

Switch Input

The switch peripheral functions and definitions are provided in **switch.h/switch.c**. Inside **switch.h**, the switch peripheral includes three main registers for setting which switches as input or floating. There are 3 functions that are implemented in **switch.c**. The *switch_init()*, sets the switch peripheral as floating or input. The *switch_read()*, reads from the switch peripheral input register **RGPIO0_IN** and returns it as a 32 bit unsigned integer. The *switch_write()*, writes to the button output register **RGPIO0_OUT.**

Detection mechanism

The sensor peripheral functions and definitions are provided in **hc_sr04.h/hc_sr04.c**. Inside **hc_sr04.h**, the sensor peripheral includes 1 register for setting which has two bits to access the *state* and *echo* bits. There are 3 functions that are implemented in **hc_sr04.c**. The *get_echo_pulse()*, reads the sensor peripheral if an echo pulse was present or not. The *get_status()*, reads the sensor peripheral if an actual obstacle was detected or not. The *hc_sr04_debug()*, writes both the *state* and *echo* bits to the serial terminal to help for debugging purposes.

Launching Mechanism

The C file **motor_control.c** is designed for FPGA-based DC motor control, utilizing direct memory-mapped I/O operations for hardware interaction. It includes **motor_control.h** for necessary declarations and defines **DC_CTRL_BASE_ADDR** as the base address for the DC motor controller, set to **0x80001704**. The file introduces two macros, **READ_PER** and **WRITE_PER**, for performing read and write operations to hardware registers using volatile pointers, ensuring direct and unoptimized access to hardware.

The core functionality is provided by two functions: **turnOnDCMotor** and **turnOffDCMotor**. **turnOnDCMotor** writes a value of 0 to the DC control register at the base address to activate the motor. Conversely, **turnOffDCMotor** writes a value of 100 to the same register to deactivate the motor. The reason the duty cycle is apparently inverted is due to the use of a PNP transistor in the motor switching circuit. This results in the motor turning on when the signal line is low. These functions enable simple and effective control of the DC motor through software, demonstrating a practical application of memory-mapped I/O in FPGA systems. More values between 0 and 100 could be added to control the motor speed more precisely (than simply on or off).
The C file **servo_control.c** is designed for controlling a servo motor in an FPGA-based system, using memory-mapped I/O for interaction with hardware. The file includes standard headers like **<stdint.h>, <stdio.h>**, and <**stdbool.h**>, along with custom headers **motor_control.h**, **buttons.h**, and **joystick_wrapper.h** for specific hardware control functionalities.

Key constants are defined at the beginning: **SERVO_CTRL_BASE_ADDR** as the base address for the servo controller (**0x80001600**), and **PWM_WIDTH_REG_OFFSET** (**0x04**) for the offset to the PWM width register. **PWM_WIDTH_REG** combines these to specify the address for the PWM width register. Two macros, **READ_PER** and **WRITE_PER**, facilitate direct read and write operations to hardware registers using volatile pointers, ensuring unoptimized access for reliable hardware control.

The file sets **WIDTH_MIN** and **WIDTH_MAX** as the limits for the PWM width, defining the servo's range of motion. **DELAY_PER_STEP** determines the delay in each increment or decrement step during servo movement. A global flag, **stopRequested**, is used to control the flow of the servo movement functions.

The **delay_loop** function implements a delay mechanism using a loop with a **nop** (no operation) assembly instruction to prevent loop optimization by the compiler. This function is utilized in the main servo movement functions (**servoMoveCCW** and **servoMoveCW**) to control the speed of the servo's motion.

servoMoveCCW and **servoMoveCW** are the core functions for moving the servo counterclockwise and clockwise, respectively. They read the current PWM width, then increment or decrement it within the defined range (**WIDTH_MIN** to **WIDTH_MAX**), checking for a stop condition or a middle joystick position to break the loop. These functions demonstrate controlled servo movement with sensitivity to user input (joystick position) and a software-based stop mechanism.

Lastly, **servoStop** provides a simple interface to stop the servo motion by setting **stopRequested** to true, demonstrating an effective way to integrate software-based control in hardware operations. Overall, this file exemplifies a practical approach to FPGA-based servo motor control, leveraging memory-mapped I/O, loop control mechanisms, and user input integration.

Joystick

The c file **joystick_control.c**, structured for an FPGA-based system, integrates joystick input, motor control, and seven-segment display functionalities. It includes standard libraries and custom headers like **servo_control.h** and **motor_control.h**, indicating its control over specific hardware elements.

The file defines memory-mapped addresses for hardware components. **JSTK_CTRL_REG** is the address for the joystick control, **SEVEN_SEG_DISPLAY_EN_REG_0** and **SEVEN_SEG_DISPLAY_Digits_REG** are for the seven-segment display, and **GPIO_SWs**, **GPIO_LEDs**, **GPIO_SW_INOUT**, **GPIO_Buttons**, **GPIO_BT_INOUT** are for GPIO control. **READ_GPIO** and **WRITE_GPIO** macros facilitate direct hardware register access.

A function **HEX_TO_DEC** converts hexadecimal to decimal values. Functions **displayOn7Seg** and **enableSevSeg** manage operations on the seven-segment display. **displayOn7Seg** writes to the display's digit register, and **enableSevSeg** activates the display.

The function **readJoystick** extracts the joystick's X-axis data from **JSTK_CTRL_REG**. This setup exemplifies integration of input processing, output display, and motor control in an FPGA context.

Wrapper files for the joystick file were created in order to identify the direction of the joystick, as well as to write the position values to the seven segment display.

Main program

The **main_shooter.c** file represents a comprehensive application for an FPGA-based system, integrating various hardware controls and sensor inputs. The file begins by including standard libraries such as <**stdio.h**> and <**stdbool.h**>, along with multiple custom headers like **joystick_wrapper.h**, **motor_control.h**, **servo_control.h**, and others. These headers suggest the file's interaction with a range of hardware components, including motors, buttons, joysticks, VGA display, and ultrasonic sensors.

The **DEBOUNCE_DELAY** macro is defined to adjust the debounce timing for button presses. A **debounceDelay** function is implemented, using a loop with a **nop** (no operation) assembly instruction to ensure the delay loop isn't optimized away by the compiler. This function is essential for accurate button press detection.

In the **main** function, the program initializes the state of the DC motor to off and displays a welcome message using the VGA. The program then initializes various variables, including **score** for tracking the game's score and arrays to read button states and process sensor inputs.

The main loop of the program is designed to continuously check and respond to inputs from buttons and the joystick. The joystick inputs are processed to control the movement of a servo motor (**servoMoveCCW**, **servoMoveCW**, **servoStop**), and joystick movements are logged using the **write_x_data** function. Button presses are debounced using the **debounceDelay** function, toggling the state of the DC motor and updating **motor_state**.

The program also includes functionality for sensor detection and score updating. It reads the status of an ultrasonic sensor multiple times, tracking changes in the sensor's output. Based on these readings, the program increments the score if the sensor detects an object and displays a corresponding message on the VGA display.

In summary, this C file orchestrates a complex interaction between various hardware components in an FPGA-based system. It demonstrates real-time input processing from joysticks and buttons, controls servo and DC motors, and integrates sensor data to update and display scores, showcasing a multifaceted application typical in embedded systems and gaming applications. Demo and GitHub