Design of a I2C IP Core and Linux Device Drivers

CSE4356-System On Chip Design

Bryan Gonzalez
1001443032

Overview

The goal of the project was to make an I2C module that could be interfaced using a AXI bus. The I2C module was designed to follow the standard for I2C protocol with the flexibillty of communication with simple or complex I2C devices (complex being devices with multiple registers). The module is interfaced using the AXI bus that connects the processor and the FPGA. Lastly, we created the drivers for the I2C module such that any program capable of reading and writing to a file can access and control the module.

Registers

The I2C module has 5 32-bit registers as a part of its memory map: Address, Register, Data, Status and Control.

- Address: Holds the 7-bit address of the device we are communicating with.
- Register: If the device is a complex I2C with different registers, the 8-bit register we are writing to is stored here.
- Data: As we receive data from I2C it is stored in a FIFO (RX FIFO) and if we are sending data, it is stored in a FIFO (TX FIFO) until we start transmitting. When we read from the data register, we get data from the RX FIFO. When we write to the data register, that data is stored in the TX FIFO.
- Status: Holds the state of both FIFOs (overflow, full and empty) as well as the ack error flag and the busy flag. The user should really only be able to read that data stored in this register. Writing to specific bits such as the overflow bits and the ack error will clear those bits.
- Control: This register determines how the I2C protocol is played out. The register determines if we are reading or writing, the number of bytes we are sending/receiving, if we are using a repeated start, if we are using a register, and if we are starting.

The user interfaces with these 5 registers using the AXI bus. When we receive an AXI write request, we fill the register specified by axi_awaddr with what is currently in S_AXI_WDATA. The only exception to this is the status register. if we get a write request for the status register, we check the bits for overflows and the ack error and make clear requests if any of those are high. When writing to the data register, the register is filled but we also make a request to store that value in the TX FIFO. When we receive an AXI read request we just load the axi_rdata with data in the register specifed by raddr. The exception this time being the data register. when we get a request for the data register, we make a read request from the RX FIFO and store that in axi_rdata.

Design - FIFO

Each FIFO can hold 15 8-bit numbers and keep track of those numbers with a read and write index. The output of the FIFO will always be the number at the read index. The FIFO is empty if both indices are the same and full if the write index is one behind the read index. These values are stored in the status register accordingly. When we make a read request, as long as the FIFO is not empty, we increment the read index, changing the output. When we get a write request, we first check if the FIFO is full. If the FIFO is full, we output an overflow error and store that in the status register. Otherwise, we store the data at the input of the FIFO at the write index and increment the write index. The finite state machine (FSM) that manages the I2C protocol sits in between the two FIFOs such that a user can only write to the TX FIFO and read from the RX FIFO. The FSM will manage transmitting the data in the TX FIFO and filling the RX FIFO when receiving.

Design – Finite State Machine

The finite state machine is the heart of the I2C protocol logic and operates at a 200kHz rate. The FSM tracks the control register, SDA input and the phase of each state to determine what the SDA output and SCL should do next. The SDA and SCL outputs are '1' or '0' in the FSM for easy viewing/coding but any '1' is translated to a 'z' at the system top to allow other devices to pull the line low. The FSM has 9 states: IDLE, START, TX_ADDR_WR, TX_REG, TX_DATA , STOP_REPEAT, START_REPEAT, TX_ADDR_RD, RX_DATA and STOP. When starting up, the FSM is in IDLE state and SDA and SCL are high. The phase is constantly being incremented at a 200kHz rate. When there is a change in state, phase is set to 0.

IDLE

- If the start bit of the control register goes high, we make a clear request for the start bit and store byte_count in a separate counter. We then set the state to START.

START: Transmit the start condition. SDA transitioning from high to low while SCL is high

- Phase 0: we make sure that SDA and SCL are high and keep them high for two phases.
- Phase 2: drop SDA low to initiate the start condition.
- Phase 3: if we are reading and we are not using a register we can just skip to TX_ADDR_RD. Otherwise, go to TX_ADDR_WR.

TX_ADDR_WR: Transmit 8 bits. First 7 being the address from the address register and the last bit is low to indicate that we are writing.

- SCL is pulled low on all the even phases and high on the odd phases.

- Phase 0 - 12: on the even phases, set SDA out to address bits (starting with the most significant bit)
- Phase 14: set SDA output to low to indicate we are writing.
- Phase 16: set SDA output to high to allow other devices to pull the SDA line low.
- Phase 17: If the SDA input is low, then another device has transmitted an ACK and we can go to TX_REG if we are using a register or TX_DATA if not using a register. Otherwise, we set the ACK error flag and clear the busy bit in the status register and jump back to the IDLE state.

TX_REG: Transmit the register from the “register” register. register.

- SCL is pulled low on all the even phases and high on the odd phases.
- Phase 0 - 14: on the even phases, set SDA out to register bits (starting with the most significant bit)
- Phase 16: set SDA output to high to allow other devices to pull the SDA line low.
- Phase 17: if SDA input does not go low, we set the ACK error flag and clear the busy bit in the status register and jump back to the IDLE state. Otherwise jump to START_REPEAT if we are reading and using a repeated start, jump to STOP_REPEAT if we are reading and not using a repeated start, or jump to TX_DATA if we are writing.

TX_DATA Transmit the data from the TX FIFO.

- If the counter (from byte count) is 0, then jump to STOP
- SCL is pulled low on all the even phases and high on the odd phases
- Phase 0 - 14: on the even phases, set SDA out to the output of the TX FIFO (starting with the most significant bit)
- Phase 16: set SDA output to high to allow other devices to pull the SDA line low and make a read request to the TX FIFO to indicate that we have read the data.
- Phase 17: if SDA input does not go low, we set the ACK error flag and clear the busy bit in the status register and jump back to the IDLE state. Otherwise we decrement the counter and start TX_DATA again.

STOP_REPEAT: Transmit a stop condition. SDA transitioning from low to high while SCL is high.

- Phase 0: pull both SDA and SCL low for two phases (one clock cycle. Otherwise the device gets stuck in a state where it holds the line low).
- Phase 2: SCL goes high.
- Phase 4: SDA goes high (indicating the stop condition).
- Phase 5: Jump to START_REPEAT.

START_REPEAT: Transmit the start condition. SDA transitioning fom high to low while SCL is high.

- Phase 0: we make sure that SDA and SCL are high and keep them high for two phases.
- Phase 2: drop SDA low to initiate the start condition.
- Phase 3: jump to TX_ADDR_RD.

TX_ADDR_RD: Transmit 8 bits. First 7 being the address from the address register and the last bit is high to indicate that we are reading.

- SCL is pulled low on all the even phases and high on the odd phases.
- Phase 0 - 12: on the even phases, set SDA out to address bits (starting with the most significant bit)
- Phase 14: set SDA output to high to indicate that we are reading.
- Phase 16: set SDA output to high to allow other devices to pull the SDA line low.
- Phase 17: If a SDA input is low, then another device has transmitted an ACK and we can go to TX_REG if we are using a register or TX_DATA if not using a register. Otherwise, we set the ACK error flag and clear the Busy bit in the status register and jump back to the IDLE state.

RX_DATA: Receive data.

- If the counter (from byte count) is 0, then jump to STOP
- SCL is pulled low on all the even phases and high on the odd phases.
- Phase 0: set SDA output to high so the other device can pull the line low when it starts transmitting.
- Phase 1-15: store the state of SDA input in the input of the RX FIFO (starting with the most significant bit).
- Phase 16: Make a write request to the RX_FIFO. Also, if the counter is 1 (we are on the last byte) we leave SDA high to indicate to the device that we don't want any more data. Otherwise, the device holds the line low after the stop condition. If the counter idicates that we have more bytes to read, we set SDA to low to transmit an ACK and the device can continue transmitting.
- Phase 17: Decrement the counter and start RX_DATA again.

STOP: Transmit a stop condition. SDA transitioning from low to high while SCL is high.

- Phase 0: pull both SDA and SCL low for two phases (one clock cycle. Otherwise the device gets stuck in a state where it holds the line low).
- Phase 2: SCL goes high.
- Phase 4: SDA goes high (indicating the stop condition).

- Phase 5: Jump to IDLE and clear the busy bit.

Design - Linux Device Drivers

The Linux Device Drivers allow users to communicate with the I2C module using the AXI bus. To use the AXI bus, write to memory address 0x43C20000 + an offset for each register (offset = 0 for address, 4 for register, 8 for data, 12 for status, and 16 for control). Except because of virtual memory, we use /dev/mem to ask the kernel for an address that translates to 0x43C20000 and we read and write to that. To make the I2C driver we created a handful of kernel objects and made a kernel module. We made 9 kernel objects the user can interface in /sys/kernel/i2c in theory. In practice, they were stored in a separate directory: /sys/i2c/i2c. The 9 kernel objects are mode, byte_count, reg, address, use_repeated_start, use_register, start, tx_data, and rx_data. All kernel objects should have read and write access except for tx_data and rx_data. tx_data should be write only and rx_data should be read only. The kernel objects we created are just files so ideally, any programming language capable of reading and writing to a file is capable of using our I2C module. Kernel Objects:

- mode
    - Reading: Returns “read” when bit 0 of the control register is 1. Otherwise returns “write.”
    - Writing: If the user writes “read,” then it sets bit 0 of the control register to 1. If the user writes "write,” then it sets bit 0 to a 0.
- byte_count
    - Reading: Returns the number of bytes we are transmitting/recieving (bits 1-4 of the control register)
    - Writing: Sets the number of bytes we are transmitting/receiving (bits 1-4 of the control register)
- reg
    - Reading: Returns the value in the “register” register.
    - Writing: Sets the value in the "register" register.
- address
    - Reading: Returns the value in the address register.
    - Writing: Sets the value in the address register.
- use_register
    - Reading: Returns “true” when bit 5 of the control register is 1. Otherwise returns “false.”
    - Writing: If the user writes “true,” then it sets bit 5 of the control register to 1. If the user writes "false,” then it sets bit 5 to a 0.

- use_repeated_start
    - Reading: Returns “true” when bit 6 of the control register is 1. Otherwise returns “false.”
    - Writing: If the user writes “true,” then it sets bit 6 of the control register to 1. If the user writes "false,” then it sets bit 6 to a 0.
- start
    - Reading: Will return the value in the status register
    - Writing: Writing anything will set bit 7 of the control register to 1, initiating the I2C protocol.
- tx_data
    - Reading: Gives an error because the user does not have reading permission
    - Writing: Loads the written value to the TX FIFO.
- rx_data
    - Reading: Reads a value from the RX FIFO.
    - Writing: Gives an error because the user does not have writing permission

Theory vs Implementation

The final project was tested against the MCP23008, a GPIO expander and complex I2C device. The goal was to configure the device, set the GPIO registers and read the GPIO registers all using the I2C module and kernel module. After some testing, the final project seems to be working as intended. The only error we found again was that my kernel objects were stored in /sys/i2c/i2c instead of /sys/kernel/i2c but the objects still worked the same. Otherwise, we had no issues communicating with the MCP23008 and the signals look good on the oscilloscope.