Simple AHB Lite to CAN bus Interface Module

Final Report

Submission Date: 12/13/2017
Tuesday 11:30am Lab (Lab Section 1)
TA: Yinghan Long

Prepared by:
Vaibhav Ramachandran, Miguel Kulisic, Sang Hun Kim

1. Executive Summary

This project aims to build a CAN bus to AHB bus interface module. This is an essential part in the electronics of any car. It facilitates communication between the sensors and the microprocessor and the transfer of commands from the microprocessor to the engine modules. Essentially, this is a way for an SoC design to converse with different electronics. This interface module makes communications between different standards easier and allows the microprocessor to be implemented in more designs. There are not many processors that are capable of connecting to a CAN bus. This project will make the connection between an SoC and CAN protocol possible. This design is ideally suited for ASIC implementations, because this module is only required to perform one task, translation between an AHB bus and a CAN bus interface. There is no need to reprogram this for any other purposes so there is no reason to use a non-ASIC design. Another major factor that justifies the design of this ASIC module is cost effectiveness. Since ASIC circuits only need to perform one action, they tend to require less components which makes them cheaper. Also, less components means less surface area which is another advantage of using ASIC. The final and perhaps the most important advantage is speed. This process needs to be as fast as possible to reduce the amount time it takes for the sensor inputs to be interpreted by the microprocessor and for the modules to receive the commands of the microprocessor. Using an ASIC will best satisfy these requirements.

Successful design of the CAN to AHB bus interface module will require the following resources:

- ARM AMBA AHB-Lite SoC bus standard documentation
- Reference Standard Cell simulation library for Mapped Design Verification
- Reference Standard Cell technology library for Final Design Layout Verification
- Verilog HDL Simulation and Design synthesis toolchain
- Controller Area Network (CAN) bus standard documentation

The following document contents will describe:

- Intended usage expectations and constraints for the design, in brief. (Section 2)
- Intended main implementation architecture. (Section 3)

2. Design Specifications

2.1 System Usage

2.1.1 System Usage Diagram

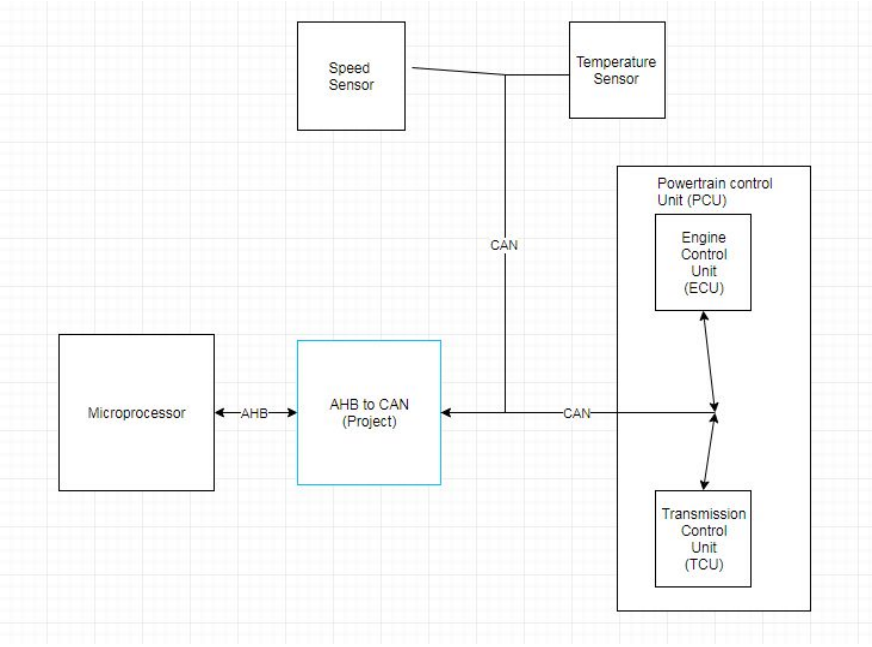


Figure 1: Example system usage diagram for Powertrain Control Unit

The diagram above shows an example of how the AHB to CAN bus interface module is intended to be used. In this diagram, it can be seen that the microprocessor requires an AHB bus connection and the rest of the modules (Speed sensor, temperature sensor, ECU, TCU) are connected to the CAN bus. The sensors will only send data into the microprocessor via the interface module while the Engine Control Unit (ECU) and Transmission Control Unit (TCU) will both send and receive data from the microprocessor. The key steps of this operation are:

1. Receive information from the sensor through the CAN bus.
2. Transform this information from the CAN protocol to the AHB protocol for the microprocessor to understand it.
3. Receive a response from the microprocessor through the AHB bus.
4. Transform this operation from the AHB protocol to the CAN protocol for either the ECU or TCU to understand.
5. Receive response from ECU or TCU through the CAN bus.
6. Transform this information from the CAN protocol to the AHB protocol for the microprocessor to understand it.
7. Send response from the microprocessor if needed.
8. Repeat above steps continuously until necessary.

2.1.2 Implemented Standard(s) and Algorithm(s)

- System-on-Chip (SoC) bus standards
    - ARM AMBA AHB-Lite
        - Reads data sent from microprocessor
        - Writes data to microprocessor
        - Up to 64 bit data bus
        - Burst transfers supported
            - Bursts of up to 16 values are supported for command input
                - Sample input values are pushed into a 16 slot FIFO queue
                - Result values are popped from a 16 slot FIFO queue
        - Transmission of data on clock edge signals
- Non System-on-Chip (Non-SoC) bus standards
    - Controller Area Network (CAN)
        - Reads data sent from CAN nodes
        - Writes data to specific nodes
        - Up to 64 bit data bus
        - 11 bit identifier used to establish priority
            - 0's override 1's
        - Transmission of data on clock edge signals

2.2 Design Pinout

Table 1: Base Frame Format for Standard CAN Bus

Note: All This signals are send through a 1 bit line to the CAN Bus. The bits represent how many bits each signal sends through this 1 bit line. Only applies to Table 1.

Signal Name	Type (IN/OUT/Bidir)	Number of bits	Description
start_of_frame	BIDIR	1	Denotes the start of the frame transmission
identifier	IN	11	Indicates which identifier has the higher priority. Lower the value, the higher priority that specified identifier is.
Remote_transmit_request	IN	1	Must be dominant (0) in data frame, and recessive (1) in remote frame
Identifier_extension_bit	IN	1	Must be dominant (0) for 11 bit identifier

Data_length_code	IN	4	Number of bytes of data (Maximum of 8 bytes at once)
Data_field	BIDIR	32	Data to be Transmitted and received from AHB-Lite Bus and ECUs
Cycle_redundancy_check	OUT	15	Cyclic 15 bit redundancy check for error detection during transmission to AHB-Lite Bus
ACK_slot	BIDIR	1	Transmitter sends 1 and receiver can assert 0 noticing transmission has been completed
EOF (End_of Frame)	BIDIR	7	Assert 1 to indicate the end of transmission.

Table 2: AHB-Lite bus slave interface pins

Signal Name	**Type (IN/OUT/Bidir)**	**Number of bits**	**Description**
start_frame_indicaor	OUT	1	Denotes the start of the frame transmission for CAN bus
identifier_address	OUT	11	Sends identifier address to indicate which identifier will be priority.
Remote_transmit_request_indicator	OUT	1	Must be dominant (0) in data frame, and recessive (1) in remote frame
Identifier_extension_bit_indicator	OUT	1	Must be dominant (0) for 11 bit identifier
Data_length_code_indicator	OUT	8	Tells the PCU current temperature and shuts down engine if it is too hot and restricts performance when too low.
Data_field	BIDIR	64	Data to be Transmitted and Received from CAN BUS
Cycle_redundancy_check	IN	15	Cyclic 15 bit redundancy check for error detection during transmission from CAN Bus

ACK_slot	BIDIR	1	Transmitter sends 1 and receiver can assert 0 noticing transmission has been completed
EOF (End_of Frame)	BIDIR	7	Assert 1 to indicate the end of transmission.
burst_mode	IN	5	Chooses number of data bytes to be burst. (Max of 16 allowed) Ex. “0x01” => Send 1 value “0x05” +> Send 5 values
begin_burst	IN	1	Begin the burst transfer

Table 3: Miscellaneous Pin Outs

Signal Name	**Type (IN/OUT/Bidir)**	**Number of bits**	**Description**
GND	IN	1	ground
VCC	IN	1	power
n_rst	IN	1	Negedge signal that resets the operation
clk	IN	1	100MHz active high signal.

2.3 Operational Characteristics

2.3.1. Interface Module Operation

Normal usage case of the Interface module:

Commands from AHB lite bus to the CAN bus:

1. Single byte writes to activate the CAN bus to prepare to receive data from the main control unit in the module.
2. Single byte writes to clear prior unnecessary commands.
3. Transfer of command byte to bus.
4. Single word writes to tell the CAN bus to transmit the command to the sensors.
5. If not done transmitting commands to the CAN bus go back to step 3.

Sensor data from CAN bus to the AHB lite bus:

1. Single byte writes to activate the AHB bus to prepare to receive sensor data from the FIFO.
2. Single byte write to clear prior sensor data that has been used.
3. Single byte write to set the type of data transfer.
    a. If burst transfer, then burst read of data samples from the FIFO.
    b. If single transfer, read data samples from FIFO based on assigned priority from CAN bus.
4. Single word write tells the AHB bus to transmit the sensor data to the processor.
5. If not done reading sensor data from the CAN bus go back to step 3.

2.3.2. Supported AHB Lite SoC Bus Transactions

2.3.2.1 Basic Read Transfer:

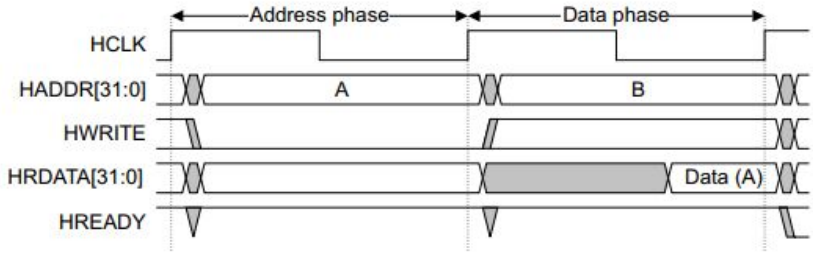


Figure 2: Basic AHB Lite bus Read Transfer timing diagram (Adapted from AHB Lite Interface Specifications Manual)

The diagram above shows a basic read transfer on the AHB bus. As can be seen, this transfer involves no waiting states. The first step is to read the address, which takes one HCLK cycle. This address can be a 32 bit value. Next, the data is used depending on the value of the HWRITE signal. When HWRITE is low, the bus is in read mode and the slave has to put the data in the read data bus. This process of reading the data might take more than a single clock cycle. The amount of time necessary to load the data is determined by the HREADY bit.

2.3.2.2 Basic Write Mode:

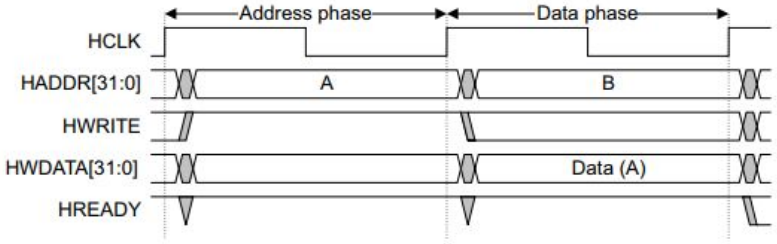


Figure 3: Basic AHB Lite bus Write Transfer timing diagram (Adapted from AHB Lite Interface Specifications Manual)

The diagram above shows a basic write transfer of the AHB bus. As can be seen in the diagram above, this transition does not involve a waiting state, just like the basic read mode. Whenever the HWRITE is high during the clock cycle, the AHB bus takes in the address of what is in HADDR. After the address has been read, on the next clock cycle, it writes the data value (32 bits) to the HWDATA. Notice that address on the HADDR can only be read when the HWRITE and HREADY are high. It is noticeable from the timing diagram that on the second clock cycle, HWRITE is both low and high. This implies that on that cycle, data of the previously acknowledged address (address A) will be written to the HWDATA during the same clock cycle (same data phase) and the data of the currently acknowledged address (address B) will be written on HWDATA during the next clock cycle.

2.3.2.3 Burst Read:

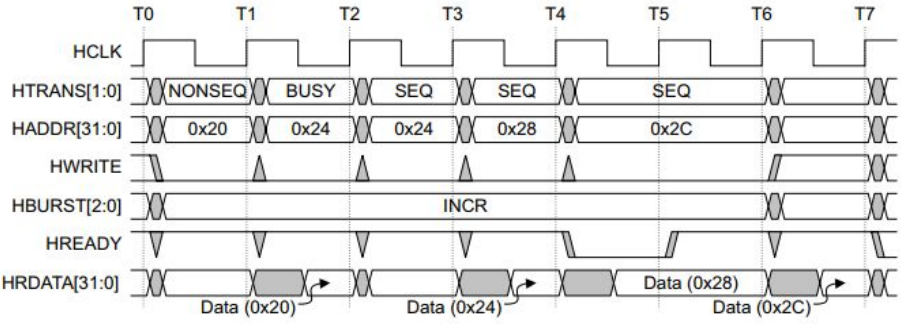


Figure 4: AHB Lite bus Burst Write Transfer timing diagram (Adapted from AHB Lite Interface Specifications Manual)

The burst read operation is to handle a bigger amount of data in a shorter period of time. This process could be helpful when reading data from more than one sensor. The diagram in figure 4 shows an example of a burst read.The 4-beat read process begins when a NONSEQ transfer is received. In Between T1 and T2, the master is incapable of working on the second beat so it enters a BUSY transfer to create a delay for the it (SEQ is asserted). After the busy state the master can move on to

perform the second beat from T2 to T3. In the next clock cycle the master works on the third beat and the slave provides the data to be read for the second beat. During T4 and T5 we can see the master working on the last beat and performing a wait operation. This wait operation is started by the slave by using the HREADY signal. Finally, the slave provides the data to be read for the third beat and in the next clock cycle does the same thing for the fourth beat.

2.3.3 CAN Bus Transactions:

2.3.3.1 CAN bus Priority Assignment:

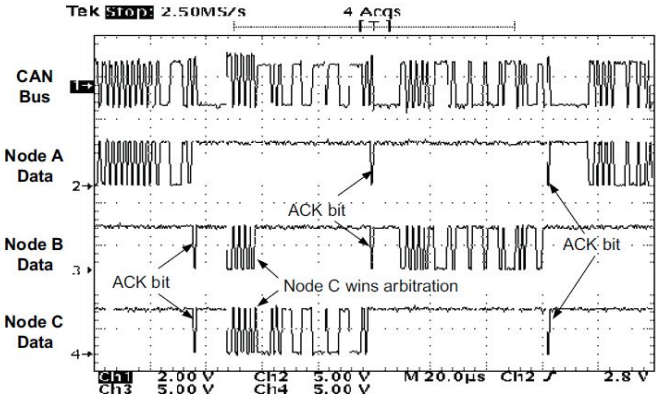


Figure 5: Basic CAN Bus Read Transfer Real-Life Timing Diagram Observed from Oscilloscope (Adapted from CAN Bus Application Report)

Can bus is defined as a communication network that links nodes, in which there are ECUs and smart sensing technology, where the bus enables nodes to communicate with one another. There are two wires that operate the CAN Bus, one being the CAN High and the other being the CAN Low wire. As can be seen above from the real-life timing diagram, as soon as the acknowledge bit has been read by the CAN bus, the nodes, or the smart sensors start to send its identifiers to determine which of the node's data will be sent to the CAN bus. In this process, whichever node has the lower identifier bit value will have the higher priority and will be able to send data to the CAN bus. Notice that the nodes that do not have the acknowledge bit stay high until the next acknowledge bit. Once all the information has been received from the nodes, the CAN bus will send out the information to the AHB Bus.

2.3.3.2 CAN bus Data Transfer

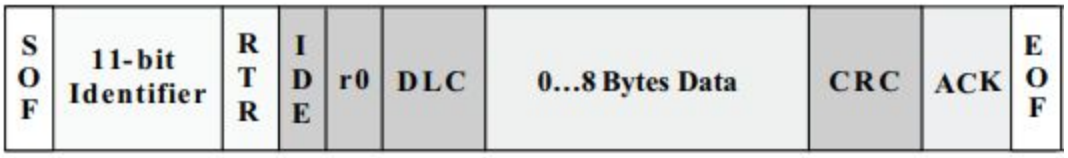


Figure 6: CAN bus data transfer diagram (Adapted from the CAN Bus Interface Specifications Manual)

The Start of Frame (SOF) bit signals the start of a transfer of data, both in read and write modes. It is followed by the 11-bit identifier which essentially assigns a priority to an incoming transmission to allow for more efficient command generation. The RTR bit (Remote Transmission Request) is used to determine if data from another sensor node is needed first. This is followed by the DLC bit (Data Length Channel) which says how many bytes of data are being transmitted. Then comes the actual data that is transmitted on the bus, which allows for a maximum of 64 bits to be transmitted. This is followed by a CRC bit that checks for overflow errors, after which the node that is receiving the data sends an acknowledge bit that indicates a successful transfer. Finally, an end of frame bit is asserted to signal the end of the transfer.

2.4 Requirements for Design:

As mentioned before, this module's purpose is to allow an SoC design to converse with the various sensors and modules present in a car. That implies, of course, that our design needs to take into consideration the requirements a car has. Since a car does not require high data transfer speeds (relative to other systems), the operation speed of this module is not our greatest concern. However adequate steps must be taken in order to ensure that the module operates at a reasonable frequency. The CAN bus, being the slower bus, will determine the transfer speed of our design. A high-speed CAN bus is capable of transferring up to 1 Mbit/s and this going to be our aim during this design. This speed is appropriate, because our CAN bus will be operating very closely to other devices. The maximum distance a CAN bus can handle with this transfer speed is 40m. This data transfer speed will put our clock at a very manageable speed of 10Mhz which is well below the maximum possible clock speed of 500Mhz that our switching transistors can handle, thus it should not be a concern. This means that the primary focus will be on making the design as small as possible in order to maximize the space available for other modules on the SoC and reduce cost. Minimizing the size of this module is key, because a modern car will have an estimate of 70 electronic control units (ECU) and components like this module will need to leave space for those and more.
The storage of data like the sensor information from the engine sensors and the commands that the microprocessor is sending will be carried out through the use of flip-flops in order to keep the design more focused on the implementation of the two bus interfaces. The amount of on-board storage the design will most likely require is 64B which accounts for the maximum 16 burst transfer bytes that the AHB can handle and the maximum 64 bits that the CAN bus can handle.
After conducting an area and timing design budgeting for our module, we have determined the approximate overall requirements for the design. The core area requirement based on the budgeting exercise was found to be 1,086,300 um^2 or approximately $1mm^2$ while the overall area requirement, including the I/O pads, comes out to be 9,734,400 um^2 or approximately $9.7mm^2$. This is considerably smaller than the 4mm x 4mm area typically seen in other designs. This shows that our

design is quite decently optimized with regards to area, since our timing constraints are fairly small. Performing a critical path timing analysis for the top 5 critical paths has shown us that the maximum delay that our design may experience is approximately 2.4ns. In order to give us some room to work with we chose a clock period of 10ns which comes up to a frequency of 100MHz. Since the CAN bus is not a very high speed interface and transmits data serially, this clock speed should be sufficient.

3 Design Implementation

3.1 Design Architecture

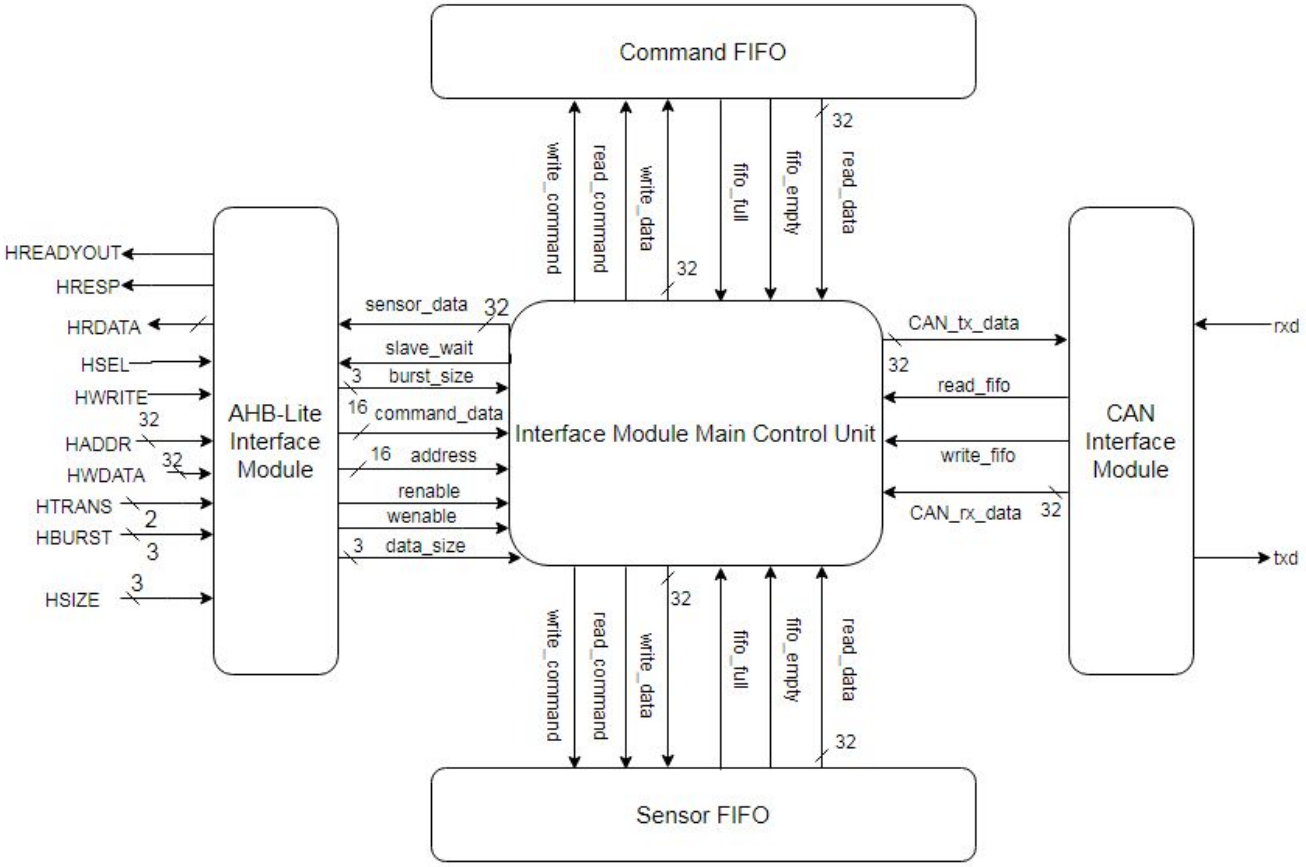
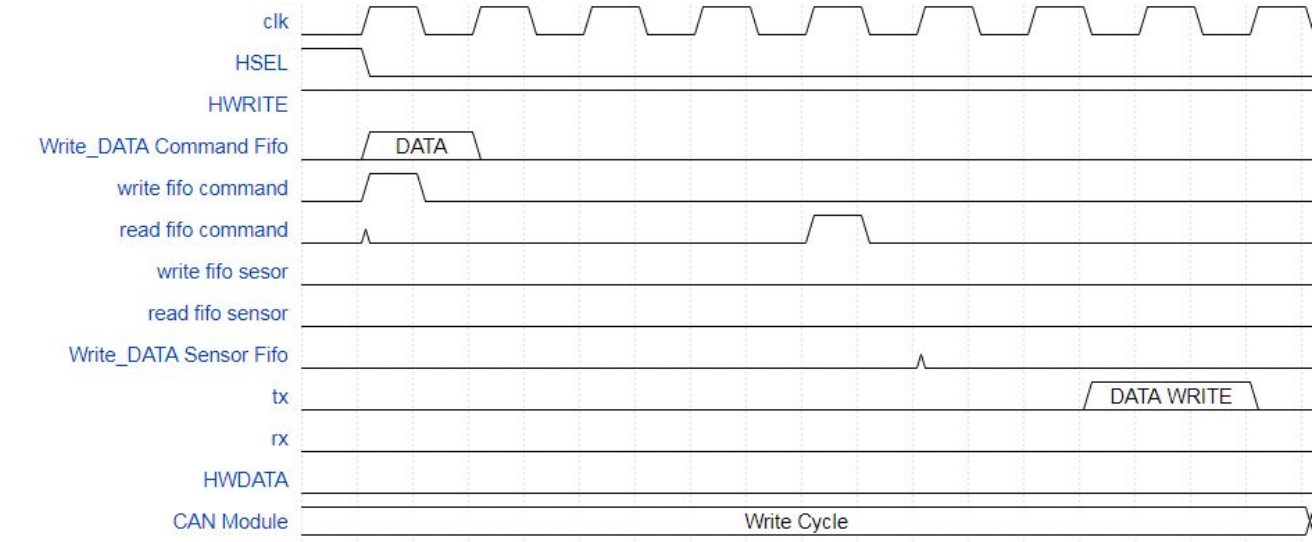


Figure 7: Architecture Diagram for AHB Lite to CAN bus interface module

The intended architecture implementation is described in Figures 3 and 4. The design will consist of two modules, one each for the AHB Lite interface and one for the CAN bus interface. These modules are used to connect the design's internal data and status locations with the external SoC environment of the processor as well as the external sensors connected to the CAN bus module. Figure 7 shows the intended architecture for an AHB bus Slave interface to a CAN bus interface and also shows the intended architecture for a CAN bus interface to an AHB bus Slave interface. Each bus will have a

value storage module, one to house the commands coming in from the AHB bus and another to store the sensor input data coming in from the CAN bus. They will both be implemented using a stream register design. The main functioning of the module will be split up into two control units, with each one handling the transfer of data in one direction. This will allow for simultaneous transfer of data between both the bus modules thus increasing the efficiency of the design. These two control units, namely the Processor-to-Sensor and the Sensor-to-Processor control unit will be implemented using Finite State Machines (FSMs) with included flags to check for the progression of data between the two buses. Status signals are sent between the two buses to acknowledge the completion of a certain byte transfer and to signal a new transfer from the storage modules. Furthermore, these storage modules will have priority bits assigned to each data byte stored in order to promptly complete time sensitive tasks.



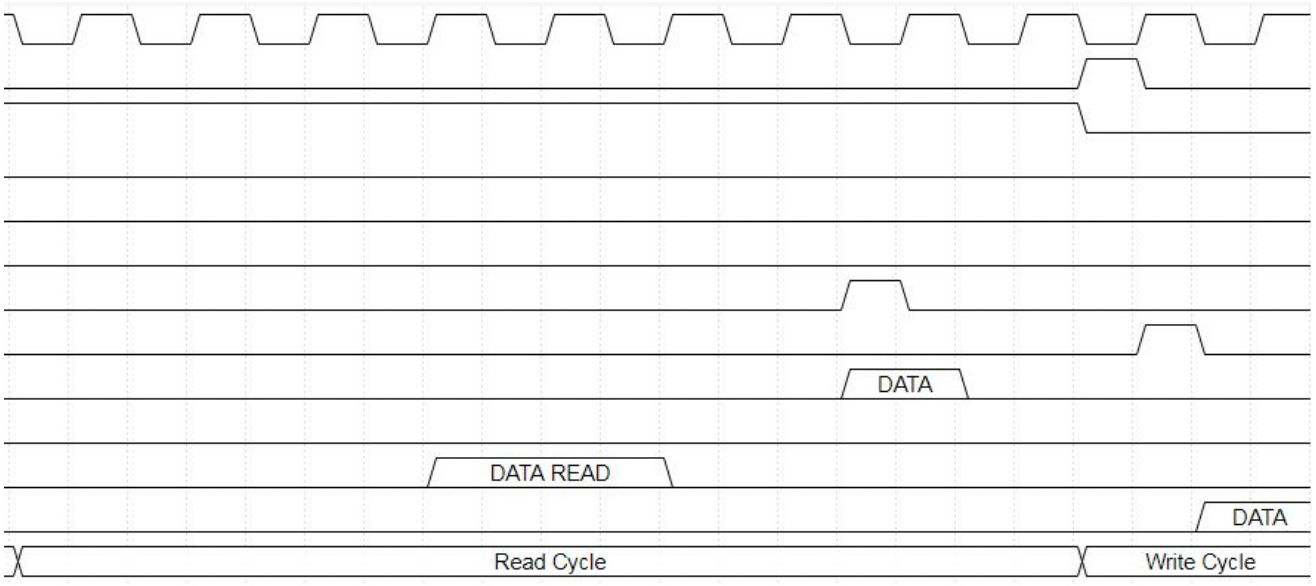


Figure 8: Basic flow of the overall design

This waveform diagram show the basic flow of data from a top down perspective. This cycle show first how data gets saved into the command fifo and send out the CAN Module, second how data gets received through the CAN module and send to the sensor fifo and third how the AHB master picks up the data in the sensor fifo. The variable CAN Module show what type of transmission the CAN bus is currently in. At the beginning, we can see how the master makes writing request by keeping the HWRITE high and raising the HSEL. This makes the command fifo read the data from the slave. Then after some time, the CAN Module sends the read command to the fifo. Later on in the write cycle, the tx register shifts out the data read by the can module. If this data is a read request. Then the CAN module goes into Read mode. During this stage the CAN module receives the data via the rx line. Then if the data is correct it will assert the write enable for the sensor fifo and give it this data. After this, the master asserts HSEL again, but this time it lowers the HWRITE signal. This signals that it will dequeue the value in the sensor fifo with a read enable fifo and the HWDATA will take the value previously in the fifo. It is important to know that CAN keeps running a writing cycle during this.

Area Estimation Table:

Table 4: Estimated Areas on each block

Name of Block	Category	Gate/FF Count	Area (um2)	Comments
AHB Master State Register	Reg. w/ Reset	3	7,200	Assuming 6 states
AHB Master Output logic	Combinational	52	39,000	Assuming Address and Data are 16bit
AHB Master Next State Logic	Combinational	8	6,000	8*500*1.5
AHB-Lite Slave Total Area	-	-	58,350	
Register for Address and Data	Reg. w/ Reset	32	76,800	Assuming 16 bits for each
RX_SR Storage Register	Reg. w/ Reset	16	38,400	16 x 1600 x 1.5
RX_SR Next State Logic	Combinational	17	12,750	17 x 500 x 1.5
RX_SR Output Logic	Combinational	20	15,000	20 x 500 x 1.5
TX_SR Storage Register	Reg. w/ Reset	16	38,400	16 x 1600 x 1.5
TX_SR Next State Logic	Combinational	17	12,750	17 x 500 x 1.5
TX_SR Output Logic	Combinational	20	15,000	20 x 500 x 1.5
Control/Check Field	Combinational	30	22,500	30 x 500 x 1.5
Acceptance Filters	Combinational	50	37,500	50 x 500 x 1.5
CAN-BUS Data	None	0	N/A	CAN-Bus data received from command FIFO
Clock Divider	Reg. w/ Reset	7	16,800	7 x 1600 x 1.5 Assuming 100 as a rollover value
Main Control Module State Reg.	Reg. w/ Reset	4	9,600	Assuming 12 States

Main Control Input Logic	Combinational	44	33,000	44*500*1.5 Assuming 7 inputs and 4 bit states
Main Control Module Output Logic	Combinational	21	15,750	21*10,500*1.5 Assuming 7 Outputs
FIFO Storage	None	-	N/A	FIFO Storage Area Budget
FIFO full Logic	Combinational	12	9,000	Check if FIFO is full
FIFO empty Logic	Combinational	6	4,500	Check if FIFO is empty
Write pointer count enable logic	Combinational	4	3,000	Enable write pointer counter
Read pointer count enable logic	Combinational	4	3,000	Enable read pointer counter
FIFO P2P Storage Registers	Reg. w/ Reset	256	614,400	Storage registers
8-input Multiplexer	Combinational	36	27,000	8-input, single output mux
Counter next state register	Reg. w/ Reset	12	28,800	Next state register for the counter
Counter output logic	Combinational	20	15,000	Output logic for the counter
Counter rollover flag logic	Combinational	16	12,000	Assuming 7 as a rollover value
Write pointer value decoder	Combinational	22	16,500	Decoding write pointer for enabling storage
Total Core Area			1,129,650	

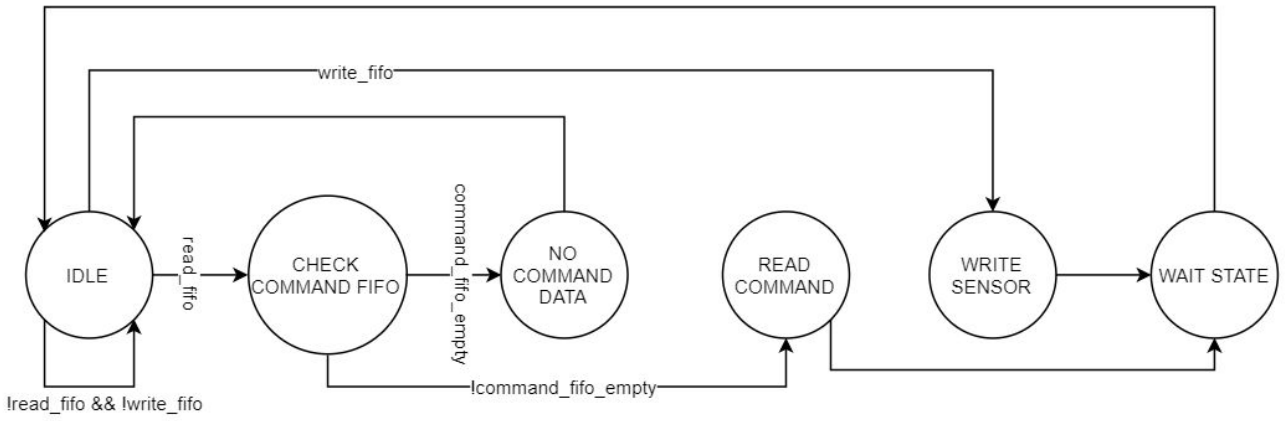
Chip Area Calculations (units in um or um2)			
Number of I/O Pads:	112		
I/O Pad Dimensions:	90	by	300
I/O Based Padframe Dimensions:	3,120	by	3,120
Core Dimensions	1,063	by	1,063
Core Based Padframe Dimensions:	1,963	by	1,963
Final Padframe Dimensions:	3,120	by	3,120
Final Chip Area:	9,734,400		

Expected Area vs Measured Area

During the design process for this project, we expected it to have an area of 9,734,400 um^2. This area was calculated using the table above. After producing a layout of our design, we calculated the area to be 9,000,000 um^2. This was really good, we improved our targeted area by almost 10 %. This is specially good considering that our target was to optimize area rather than speed. This chip needs to be as small as possible.

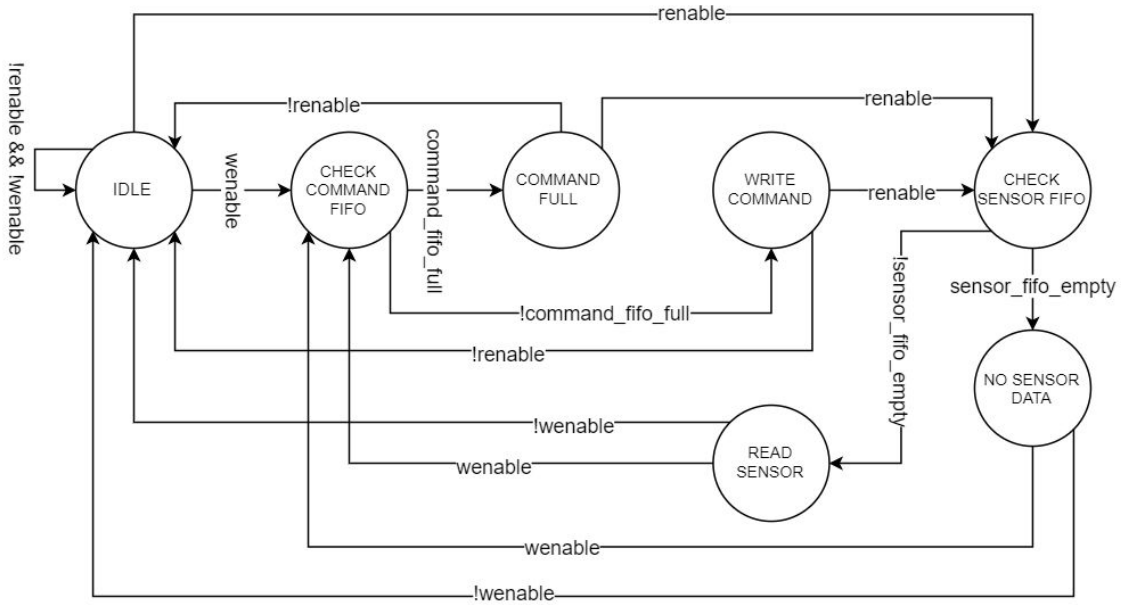
3.2 Functional Block Diagrams

3.2.1 State Machines for Sensor to Processor and Processor to Sensor Control unit



Outputs	slave_wait	command_read	tx_data	sensor_write	sensor_wdata
IDLE	1	0	0	0	0
CHECK_COMMAND_FIFO	0	0	0	0	0
NO_COMMAND_DATA	0	0	0	0	0
READ_COMMAND	1	1	command_rdata	0	0
WRITE_SENSOR	1	0	0	1	rx_data
WAIT_STATE	0	0	0	0	0

Figure 9: FSM with outputs for the Sensor to Processor Block of the Control Unit



Outputs	slave_wait	command_write	command_wdata	sensor_read	sensor_data
IDLE	0	0	0	0	0
CHECK_COMMAND_FIFO	1	0	0	0	0
CHECK_SENSOR_FIFO	1	0	0	0	0
COMMAND_FULL	1	0	{address, command_data}	0	0
WRITE_COMMAND	1	1	0	0	0
NO_SENSOR_DATA	1	0	0	0	0
READ_SENSOR	1	0	0	1	sensor_rdata

Figure 10: FSM with outputs for the Processor to Sensor Block of the Control Unit

RTL Diagram for Controller Block

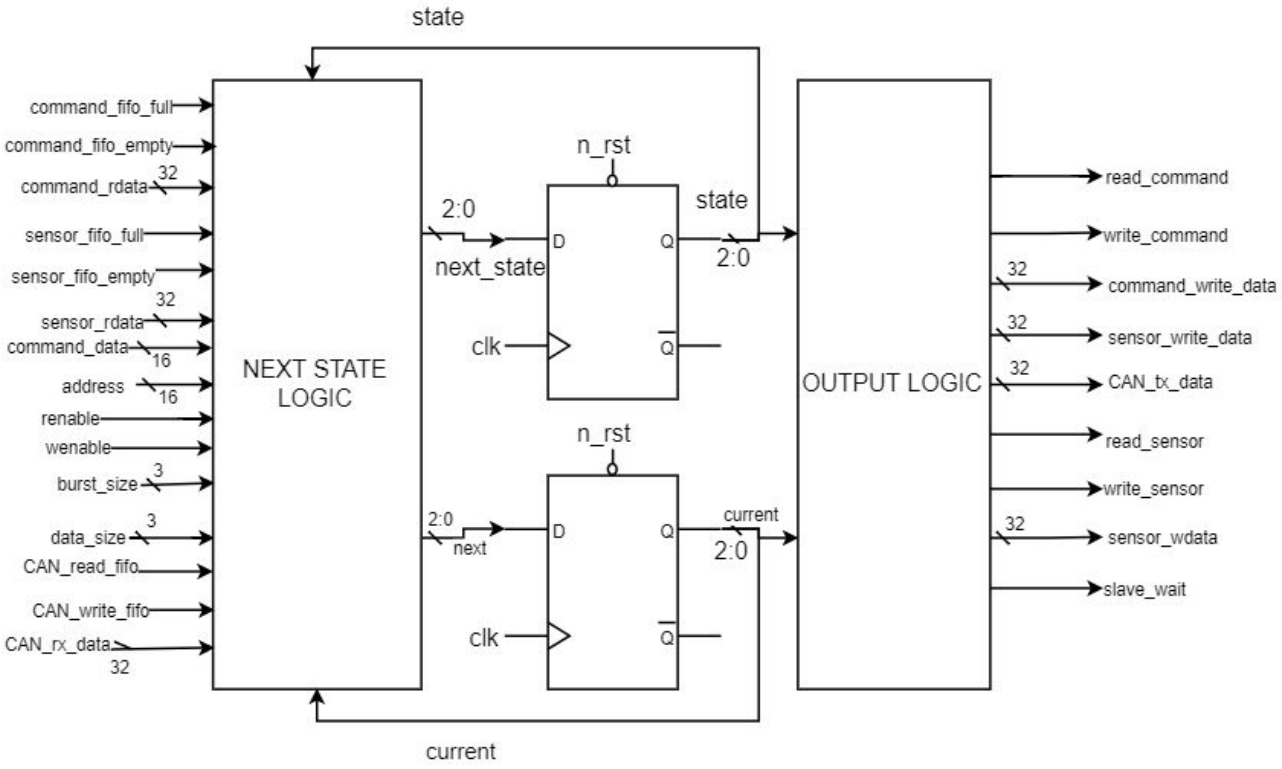


Figure 11: RTL for the Controller Block of the Interface module

This RTL diagram for the Controller Block of the Interface Module shows a state machine with 15 inputs and 9 outputs. This block is meant to control the operations of all other devices.

Area:

Since this state machine will require 14 states it will need 4 registers with resets in order accomplish this. Given the number of states and inputs, we have estimated that the number of gates required to implement the next state logic will be 44. Finally, the estimated number of gates for the output logic is 21. This number is based on the number of outputs the block has.

- Total Gates 65 → Area of 65*1.5*500 = 48,750 um^2
- Total Registers 4 → Area of 4*1600*1.5 = 9,600 um^2
- Total Area = 58,350 um^2

3.2.2 Functional Block Diagram & State Diagram for CAN bus Interface Module

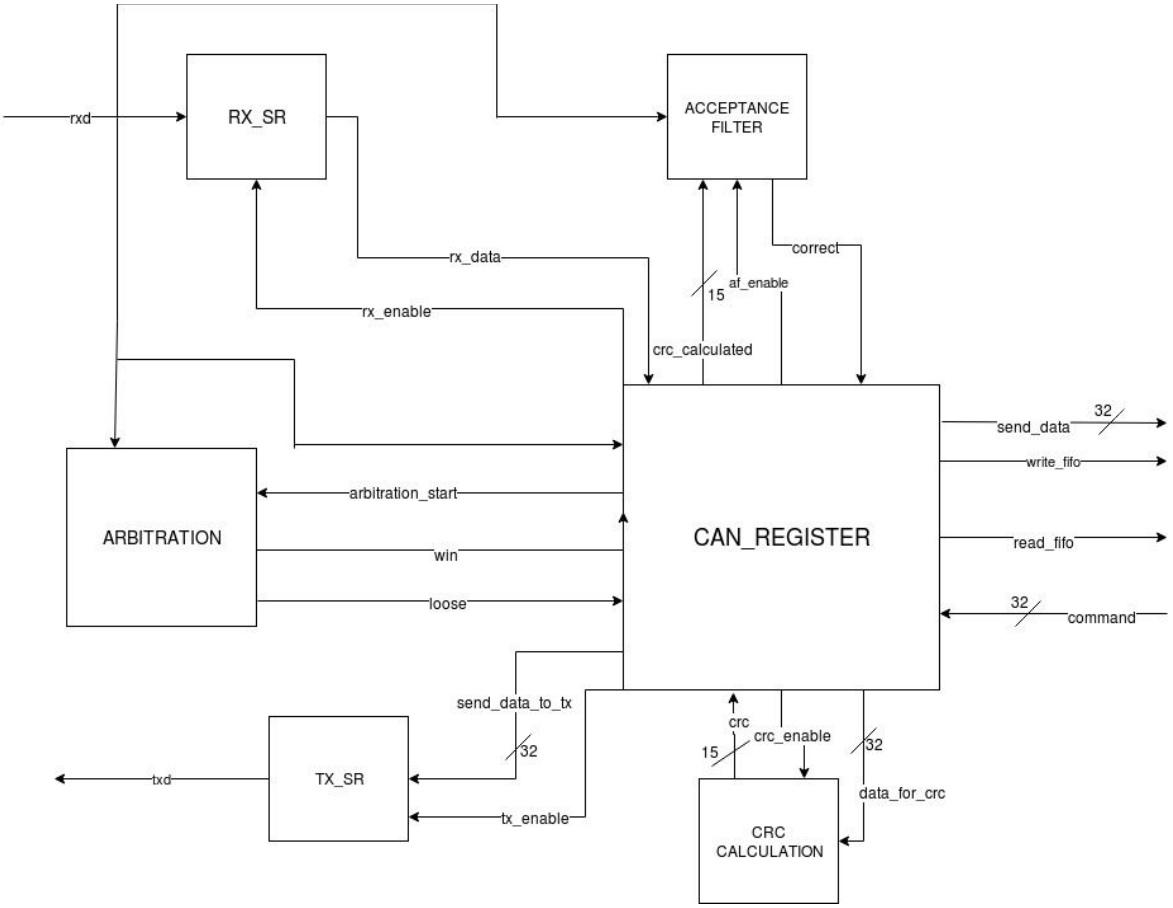
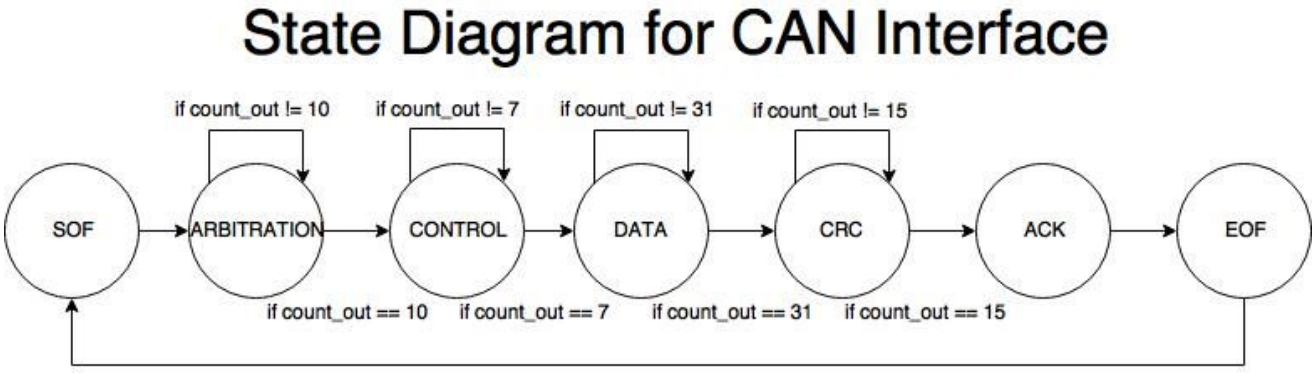


Figure 12: Specific Functionality within CAN BUS Protocol Machine



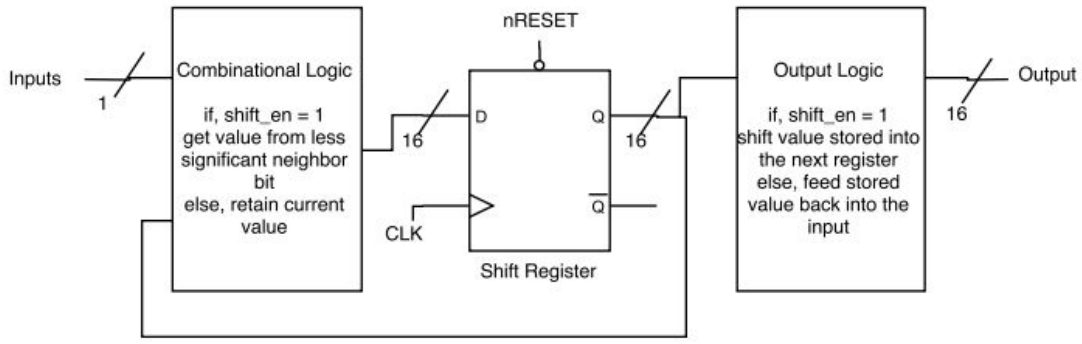
Outputs	load_enable	send_data_to_tx	arbitration_start	clear	count_enable	rollover_val
SOF	1	0	0	1	1	1
ARBITRATION	1	0	1	0	1	10
CONTROL	1	0	0	0	1	7
DATA	1	data_stored_in_	0	0	1	31

		command				
CRC	1	0	0	0	1	14
ACK	1	0	0	0	1	1
EOF	1	0	0	1	1	1

Outputs	rx_enable	tx_enable	crc_enable	read_fifo	write_fifo	af_enable
SOF	1	1	0	0	0	0
ARBITRATION	1	1	0	1	0	0
CONTROL	1	1	0	0	0	0
DATA	1	1	1	0	0	0
CRC	1	1	0	0	0	1
ACK	1	1	0	0	1	0
EOF	1	1	0	0	0	0

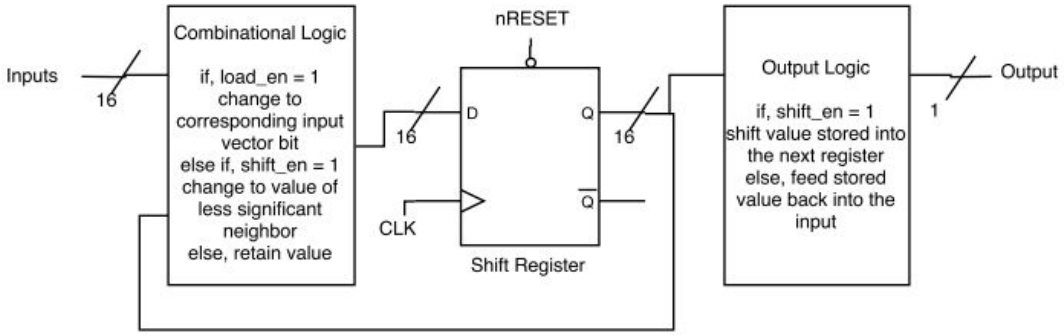
Outputs	WRITE_CAN_PH	sync	send_data_ph
SOF	0	0	0
ARBITRATION	1	0	0
CONTROL	0	0	31'b{001,'0}
DATA	0	0	0
CRC	0	0	0
ACK	0	0	0
EOF	0	0	0

Figure 13: Specific State Diagram for CAN Module Interface and its corresponding outputs



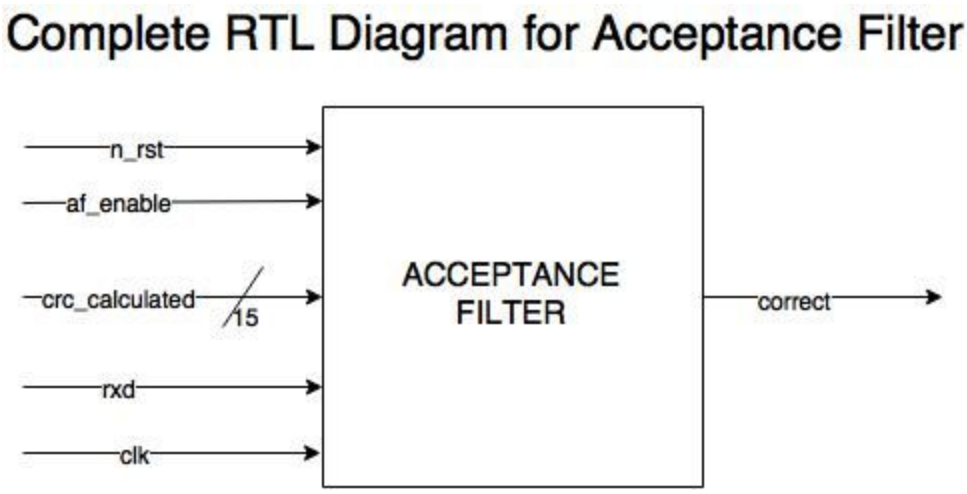
Specific Functionality within CAN BUS

Figure 14: RTL diagram for Serial-in, Parallel-out Shift Register for txd



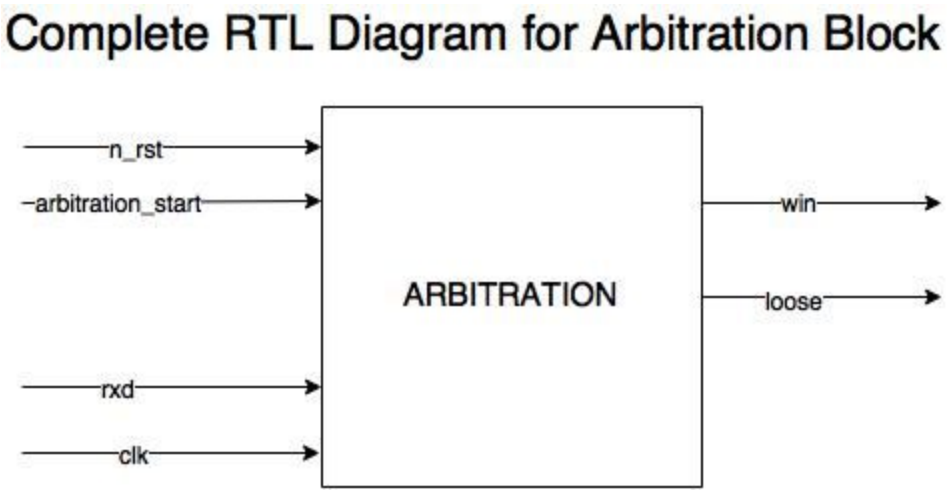
Specific Functionality within CAN BUS
Figure 15: RTL diagram for Parallel-in, Serial-out Shift Register for rxd

The block diagram above from Figure 10 represents the core of the CAN bus that receives data from Command FIFO and transmits data to Sensor FIFO. There exists total 7 states in CAN register, which are SOF, ARBITRATION, CONTROL, DATA, CRC, ACK, and EOF. The data in CAN register will always be sending out to the CAN-Bus through TX-SR and read the value of CAN-Bus from RX-SR. Whether the CAN interface will be in a read mode or write mode is determined from the ARBITRATION block. From arbitration state, the value of rxd from RX-SR is being compared with its own ID value and check whether it is read or write mode. If the value of rxd is high and the ID bit value that has been shifted out is low, then that CAN node recognizes itself that it won the arbitration and will receive the command from Command FIFO. On the contrary, if rxd is low and the shifted ID bit value is high, then that CAN node will have a signal of "loose" set high and will be sending out the data to the Sensor FIFO. Once CAN register has either win or loose signal, it will then move to Control state and see if the data that are being transferred or received have correct number of bits, in which for our design is always 32 bits. In data state, the command that has been received will be transferred to CAN-Bus if CAN register is in read mode. If the CAN register is in write mode, then it will store the 32 bit value of data from RX-SR and wait for the next state to send out the data to Sensor FIFO. At the same time in the data phase, CAN register will initiate the calculation for cyclic redundancy check (CRC) value to compute the value whether the data that has been transferred or received are correct and this will be done during the data phase. On the next state, which is CRC, the computed crc value from data phase will then be compared with the crc value received from rxd and check whether the data has been sent out correctly or received correctly. If they are identical, then there is no error and set the acknowledge bit (ACK) to 0, or else set to 1. Moving on to next state (ACK), the CAN-register will then initiate the acceptance filter module, in which decides whether or not the data should be sent to Sensor FIFO. If crc values from previous state are identical, then in read mode, nothing needs to be completed since its task is to send out the data, but set acknowledge bit to 0. This represents that there exists no error during the transmission. When in write mode, if crc values match, then send out the 32 bit data received from CAN-Bus to the Sensor FIFO and set ack bit to 0 as well. If not, then it only sets ack bit to 1, saying that there is an error during transmission. In ACK state and EOF state, it will wait for another data to be received or to be transferred.



Specific Functionality within CAN BUS
Figure 16: RTL diagram for Acceptance Filter in CAN-BUS Module

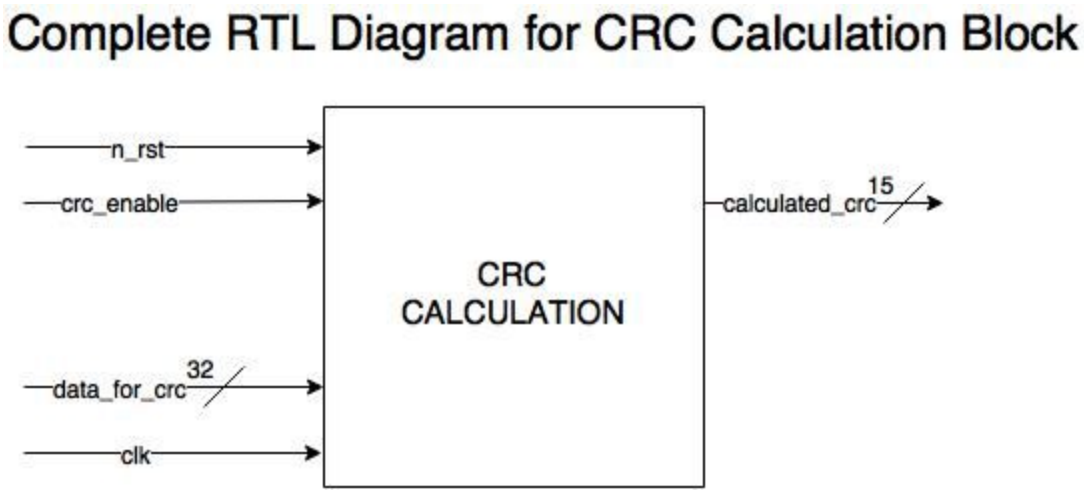
The RTL diagram shown above in Figure 14 represents the Acceptance Filter block, in which it computes whether or not the data that has been received from RX-SR or the data that has been transferred through TX-SR is correct. This acceptance filter block takes in inputs of crc_calculated, which is the calculated cyclic redundancy check value, and rxd from CAN-bus. When acceptance filter enable is on, then starting from the most significant bit of the crc_calculated value it will b compared with rxd to see if they match or not. If they match throughout the entire 15 bit of crc_calculated value, then it shows the data transmission has been successfully done and set the correct bit to 1. Or else, the correct bit is set to 0, meaning the data transmission was incorrect.



Specific Functionality within CAN BUS
Figure 17: RTL diagram for Arbitration in CAN-BUS Module

This RTL block diagram shown above in Figure 15 represents the arbitration block. The purpose of this block is to decide whether the CAN node will behave as a read mode or write mode. When arbitration_start bit is set from the CAN register, arbitration block will take in the rxd value from the CAN-Bus and starting from the

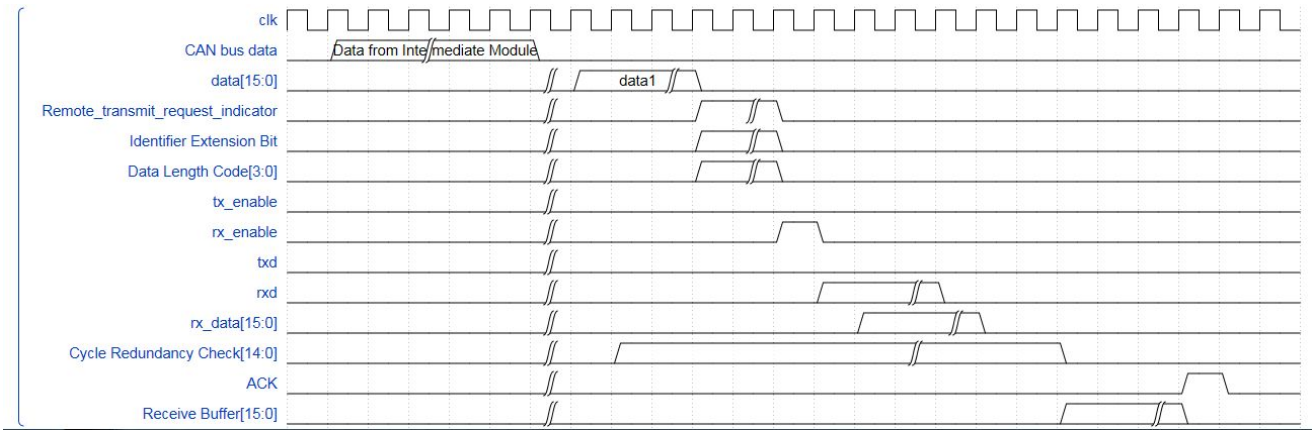
most significant bit of its own ID, it will then compare with the rxd value to determine the mode. When rxd value and ID value bit do not match and if rxd value is high, then that particular CAN node will be in a read mode since it has won the arbitration. On the contrary, while rxd value and ID value bit do not match and rxd value is low, then that particular CAN node will behave as a writing mode since it has lost the arbitration.



Specific Functionality within CAN BUS

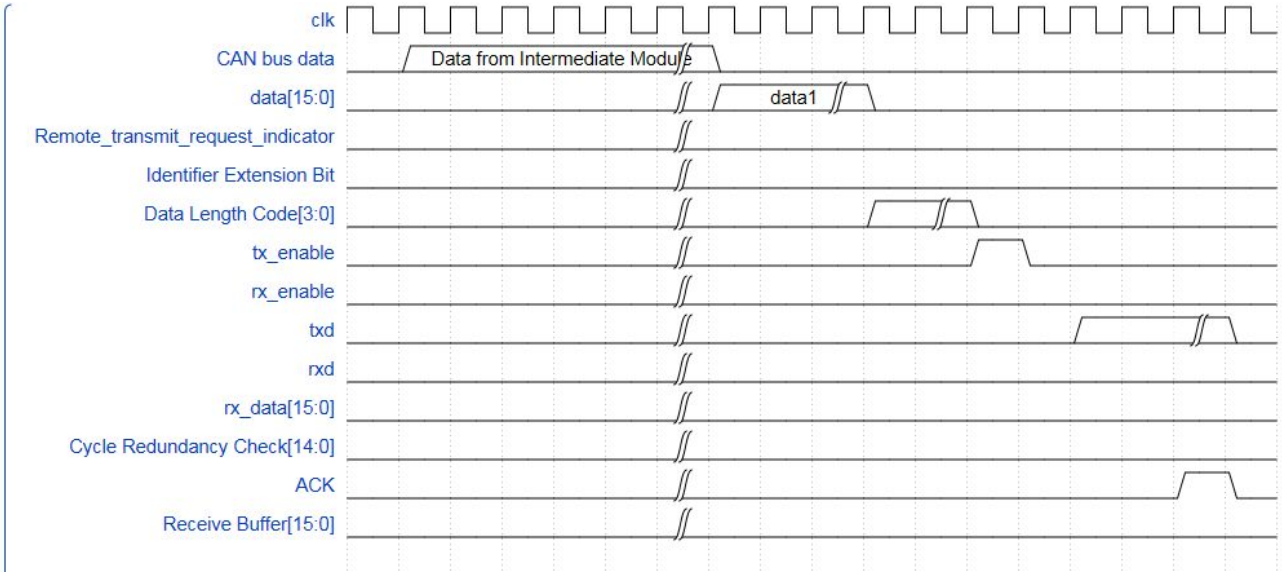
Figure 18: RTL diagram for CRC-Calculation in CAN-BUS Module

The transmitter calculates a check-sum from the transmitted bits and provides the result within the frame in the CRC field. The receivers use the same polynomial to calculate the check-sum from the bits as seen on the bus-lines. Here, the polynomial value for CAN-module is $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + x^0$. The self-calculated check-sum is compared with the received on from the Command FIFO. If it matches, the frame is regarded as correctly received and the receiving node transmits a dominant state in the ACK slot bit, overwriting the recessive state of the transmitter. In case of a mismatch, the receiving node sends an Error Frame after the ACK delimiter.



Specific Functionality within CAN BUS

Figure 19: Timing Diagram Analysis for CAN Protocol Machine when Reading Data from Transceiver



Specific Functionality within CAN BUS

Figure 20: Timing Diagram Analysis for CAN Protocol Machine when Writing Data to Transceiver

As seen above from the timing diagram, remote transmit request indicator determines whether the CAN Protocol Machine behaves as reading data from specific transceiver or writing data to the transceiver. According to how protocol machine acts, txd is asserted or rxd is being read. Furthermore, whenever the ACK bit is asserted, protocol machine realizes that it is ready for the next CAN bus data.

Area for CAN-Bus Module

RX_SR Storage Register Block = Total Flip-Flops: 16 → Area of 16*1,600*1.5 = 38,400 um^2
RX_SR Next State Logic Block = Total gates: 17 → Area of 17*500*1.5 = 12,750 um^2
RX_SR Output Logic Block = Total gates: 20 → Area of 20*500*1.5 = 15,000 um^2
TX_SR Storage Register Block = Total Flip-Flops: 16 → Area of 16*1,600*1.5 = 38,400 um^2
TX_SR Next State Logic Block = Total gate: 17 → Area of 17*500*1.5 = 12,750 um^2
TX_SR Next State Logic Block = Total gate: 20 → Area of 20*500*1.5 = 15,000 um^2
Acceptance Filter Register Block = Total Flip-Flips: 34 →Area of 34*1,600*1.5 = 81,600 um^2
CRC-Calculation Register Block = Total Flip-Flops: 15 → Area of 15*1,600*1.5 = 36,000 um^2
Arbitration Register Block = Total Flip-Flops: 13 → Area of 13*1,600*1.5 = 31,200 um^2
CAN-Register Block = Total Flip-Flops: 70 → Area of 70*1,600*1.5 = 168,000 um^2

Total Area for CAN-BUS Module = 449,100 um^2

3.2.3 Functional Block Diagram for AHB Lite interface Module

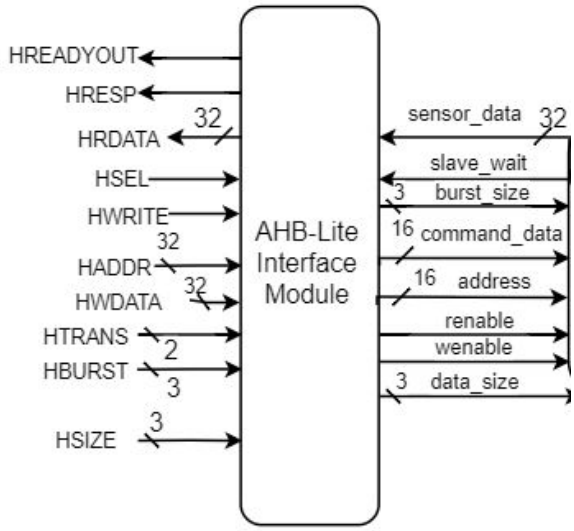
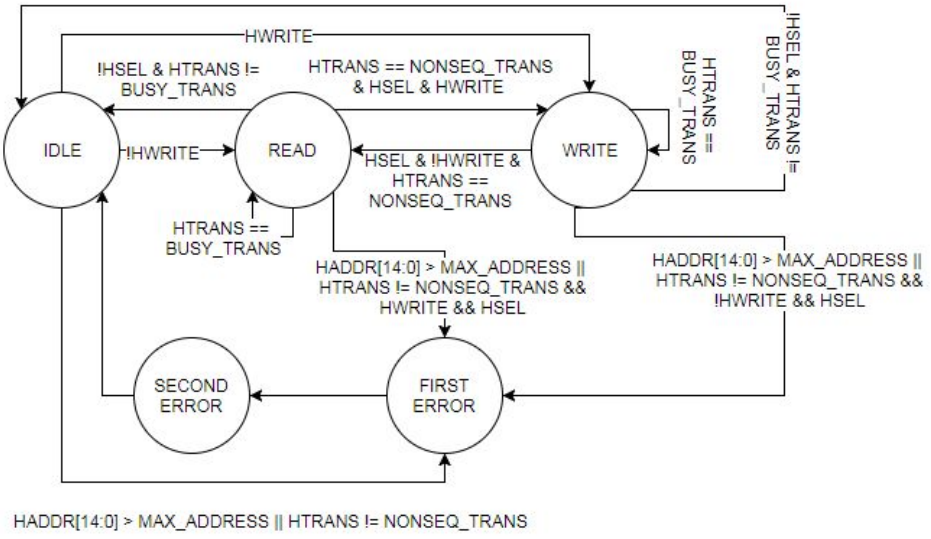


Figure 21: Architecture Diagram for AHB Lite module



Outputs	HREADYOUT	HRESP	renable	wenable
IDLE	!slave_wait	0	0	0
READ	!slave_wait	0	1	0
WRITE	!slave_wait	0	0	1
FIRST_ERROR	0	1	0	0
SECOND_ERROR	0	1	0	0

Figure 22: State machine for AHB-Lite Slave

This diagram breaks down the many signals that the slave both receives and delivers. The transfer response signals are the response that the slave sends the master. The global signals represent the clock and reset used by the module. The Address control signals on the right signal which local address the slave needs to write or read from. In this project there will only be one slave. This slave then needs to interpret this command and respond accordingly. It must inform the master of it's success in doing so, failure in doing so or if it is waiting for the data transfer. This master bus receives its instructions directly from the microcontroller. The state machine gives us a closer look at how the slave interprets the signals it receives from the master. Figure 20 shows the state machine for the slave. The functional block diagram in figure 19 also shows the amount of connections that we are going to need.

RTL Diagram of the Slave:

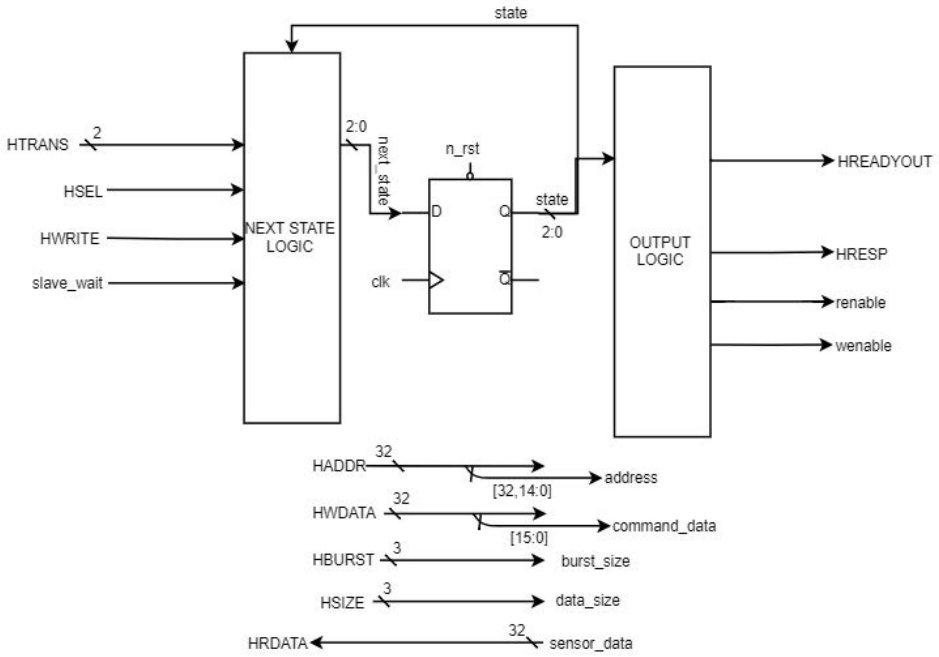


Figure 23:RTL for slave module

The slave is comprised of a simple state machine and some rerouting of signal wires. We have approximated that it will need 5 states to accomplish this purpose. The master for this slave is controlled by the Microcontroller which will be simulated via our test bench.

Area:

Given the number of states this slave module needs, it will require 3 registers with a reset. We have estimated the next state logic to require 8 logic gates, because the next state logic is not very complex. The output logic will require a total of 52 gates. This is due to the large sizes of HWRITE and HADDR.

- Total Gates: 60 → Area of 52*500*1.5 = 45,000 um^2
- Total Register: 3 → Area of 3*1600*1.5 = 7,200 um^2
- Total Area: 52,200 um^2

Timing Diagrams

For a timing diagram describing the processes of the diagrams above, the process is outlined in Figure 2 and Figure 3. These Figures contain the timing diagrams for the Basic Read Mode and Basic Write Mode. This application of the AHB bus will only require one slave.

3.2.4 Functional Block Diagram for the Command and Sensor FIFOs

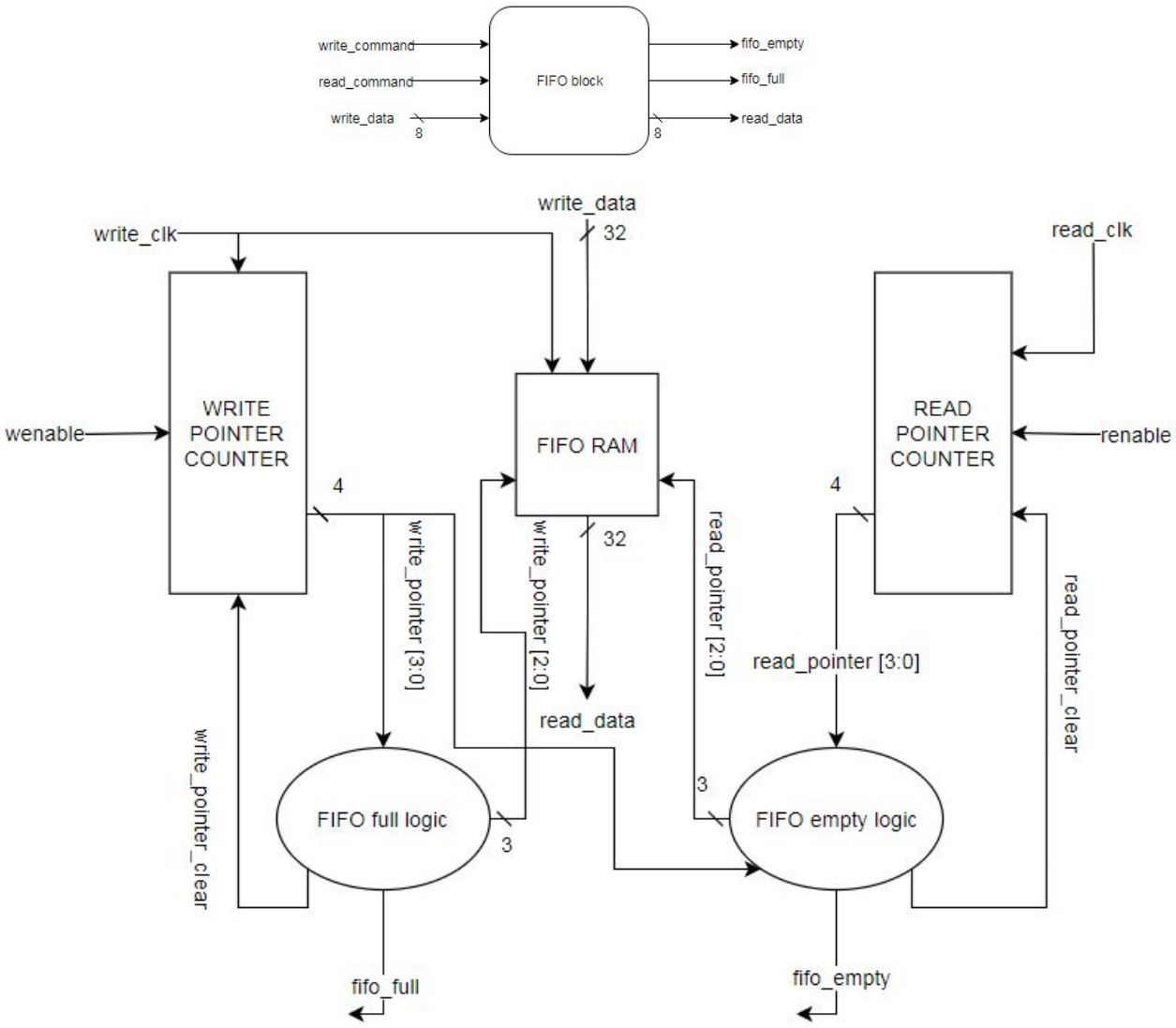


Figure 24: Top Level diagram of the FIFOs for storing Commands and Sensor Data

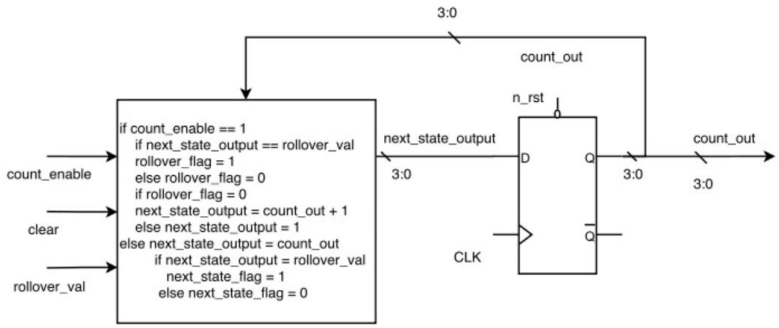


Figure 25: RTL Diagram for the counter used to increment the pointers

Timing Diagram

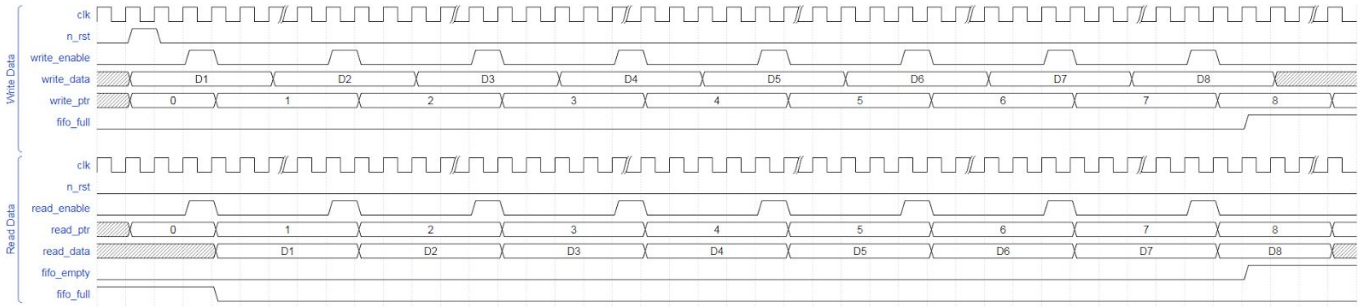


Figure 26: Fifo Read and Write timing diagrams.

In this timing diagram we can see the standard operation of a Fifo. Data gets loaded in every clock cycle that the load signal and extracted every clock cycle that the extract signal is on. In addition, the signals empty and full inform us of the current condition of the fifo. The outputs will go out of the fifo in the same order that they went in. In other words, first in, first out. Just like in our design, this diagram only features one clock. Two clocks are not necessary for this operation.

Area for a single FIFO:

FIFO full combinational block: Total gates: 6 → Area of 6*500*1.5 = 4500 um^2

FIFO empty combinational block: Total gates: 3 → Area of 3*500*1.5 = 2250 um^2

Write pointer count enable logic: Total gates: 2 → Area of 2*500*1.5 = 1500 um^2

Read pointer count enable logic: Total gates: 2 → Area of 2*500*1.5 = 1500 um^2

FIFO parallel to parallel storage registers: Total FFs: 128 → Area of 128*1600*1.5 = 307200 um^2

8-input multiplexer: Total gates: 36 → Area of 18*500*1.5 = 13500 um^2

Counter next state register: Total FFs: 6 → Area of 6*1600*1.5 = 14400 um^2

Counter output logic: Total gates: 10 → Area of 10*500*1.5 = 7500 um^2

Counter rollover flag logic: Total gates: 8 → Area of 8*500*1.5 = 6000 um^2

Write pointer value decoder: Total gates: 11 → Area of 11*500*1.5 = 8250 um^2

Total Area: 361200 um^2

Area for both the FIFOs combined: 722400 um^2

The relatively large area requirements for the FIFOs is primarily because we are storing 16 bit data values at each location. As such this will require 128 registers for each FIFO which contributes to the bulk of the design.

3.3 Design Timing Analysis

1) Main Control Unit to AHB-Lite Slave

Starting Component	Propagation Delay (ns)	Combinational Logic	Propagation Delay (ns)	Ending Component	Setup Time or Propagation Delay (ns)	Total Path Delay (ns)	Target Clock Period (ns)
Main Control Unit to AHB Slave	0.48	Output Logic Control Module, Input Logic AHB-Lite Slave	1.44	AHB-Lite Slave Next State Register	0.24	2.2	10

Figure 27: Main Control Unit to AHB-Lite Slave Timing Analysis

This a critical path that goes from the registers in the Main Control Unit to the registers in the AHB-Lite Slave. The data needs to pass through the the output logic of the Control Module and the Next State Logic for the AHB-Lite Slave. It also needs to account for the propagation delay of the registers inside the Main Control Unit.

Calculations:

- **Propagation delay**= (Flip-flop Propagation Delay)* (Wire and Capacitive Loading Delay Overhead) =0.4 * 1.2 =0.48 ns.
- **Combinational Logic:**
  - **Output Logic (Control Module)** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*3*1.2 = 0.72 ns.
  - **Next State Logic for AHB-Lite Slave** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.6*3*1.2 = 0.72 ns.
- **Setup Time for AHB-Lite Slave** = (Flip Flop Setup Time) * (Wire and Capacitive Loading Delay Overhead) = 0.2*1.2 = 0.24
-

2) Receive Shift Register from CAN-Bus to FIFO Storage register

Starting Component	Propagation Delay (ns)	Combinational Logic	Propagation Delay (ns)	Ending Component	Setup Time or Propagation Delay (ns)	Total Path Delay (ns)	Target Clock Period (ns)
Receive Shift Register from CAN-Bus	0.5	Acceptance Filter, Pointer value check	1	Sensor FIFO P2P Storage Register	0.3	1.8	10

Figure 28: Receive Shift Register from CAN-Bus to FIFO Storage register

This path goes from the register inside the series to parallel shift register in the CAN-Bus to the registers inside Sensor FIFO. This path is used when the CAN-Bus sends data to the AHB-Lite Bus. The data is first delayed by the propagation delay of the registers in the shift registers. Then, it must go through the combinational logic in the Acceptance filter and the Pointer Value check. Finally, it also needs to comply with the setup time of the registers inside the Sensor FIFO.

Calculations:

- **Propagation delay** = (Flip-flop Propagation Delay)* (Wire and Capacitive Loading Delay Overhead) =0.4 * 1.2 = 0.48 ns ~= 0.5ns
- **Combinational Logic:**

- **Output Logic (Acceptance Filter)** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*3*1.2 = 0.72 ns
- **Next State Logic for FIFO Storage** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*1*1.2 = 0.24 ns.
- 0.72ns + 0.24ns = 0.96ns ~= 1.0ns

- **Setup Time for Sensor FIFO Registers** = (Flip Flop Setup Time) * (Wire and Capacitive Loading Delay Overhead) = 0.2ns * 1.2ns = 0.24ns ~= 0.3ns

3) AHB-Lite Slave Next State Register to Command FIFO Storage

Starting Component	Propagation Delay (ns)	Combinational Logic	Propagation Delay (ns)	Ending Component	Setup Time or Propagation Delay (ns)	Total Path Delay (ns)	Target Clock Period (ns)
AHB-Lite Slave Next State Register	0.5	Output Logic block, Pointer value check	1.5	Command FIFO P2P Storage Register	0.6	2.4	10

Figure 29: AHB-Lite Slave Next State Register to Command FIFO Storage

This path goes from the register in the AHB-Lite Slave to the Registers in the Command FIFO. This path is used both during the address phase of the transmission protocol and during the write mode of the AHB-Lite Bus. First, we must account for the propagation delay caused by the registers inside the AHB-Lite Bus. This data must also go through the combinational logic blocks "Output Logic block (AHB-Lite bus)" and the Pointer value check in the Command FIFO. Finally this value must meet the setup time for the Command FIFO Registers.

Calculations:

- **Propagation delay** = (Flip-flop Propagation Delay)* (Wire and Capacitive Loading Delay Overhead) =0.4 * 1.2 = 0.48 ns ~= 0.5ns
- **Combinational Logic:**
  - **Output Logic (Pointer value check)** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*3*1.2 = 0.72 ns
  - **Next State Logic for Command FIFO Storage** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*1*1.2 = 0.24 ns.
- **Setup Time for Command FIFO Registers** = (Flip Flop Setup Time) * (Wire and Capacitive Loading Delay Overhead) = 0.2ns * 1.2ns = 0.24ns ~= 0.3ns

4) Sensor FIFO Storage Register to AHB-Lite Slave Next State Register

Starting Component	Propagation Delay (ns)	Combinational Logic	Propagation Delay (ns)	Ending Component	Setup Time or Propagation Delay (ns)	Total Path Delay (ns)	Target Clock Period (ns)
Sensor FIFO P2P Storage Register	0.5	8-input mux, Next state logic block	1.5	AHB-Lite Slave Next State Register	0.6	2.4	10

Figure 30: Sensor FIFO Storage Register to AHB-Lite Slave Next State Register

This path goes from the registers inside the Sensor FIFO storage to the registers inside the AHB-Lite Slave. This path is used when the Slave requests data from the CAN-Bus. The delay path starts with the propagation delay in the storage registers from the FIFO. Next, it must go through the 8-input mux found in the Sensor FIFO and though the Next state logic block from the AHB-Lite Slave. These two blocks are purely combinational. Finally, the value must be stable for the Setup Time of the registers in the AHB-Lite Slave.

Calculations:

- **Propagation delay** = (Flip-flop Propagation Delay)* (Wire and Capacitive Loading Delay Overhead) =0.4 * 1.2 = 0.48 ns ~= 0.5ns
- **Combinational Logic:**
  - **Output Logic (8-input mux)** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*4*1.2 = 0.96 ns
  - **Next State Logic for Command FIFO Storage** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*1*1.2 = 0.24 ns.
- **Setup Time for Command FIFO Registers** = (Flip Flop Setup Time) * (Wire and Capacitive Loading Delay Overhead) = 0.2ns * 1.2ns = 0.24ns ~= 0.3ns

5) Command FIFO Storage Register to Transmit Shift Register in CAN-Bus

Starting Component	Propagation Delay (ns)	Combinational Logic	Propagation Delay (ns)	Ending Component	Setup Time or Propagation Delay (ns)	Total Path Delay (ns)	Target Clock Period (ns)
Command FIFO P2P Storage Register	0.5	8-input mux, Control/Check field	1.5	Transmit Shift Register in CAN-bus	0.3	2.3	10

Figure 31: Command FIFO Storage Register to Transmit Shift Register in CAN-Bus

This path goes from the registers inside the Command FIFO to the Transmit Shift Register inside the CAN-bus. It is used during both the address phase of the AHB-Lite protocol and during the AHB-Lite Slave write mode. The first delay is in the propagation delay of the register for the Command FIFO. Then, we must account for the delay of the 8-input mux inside the Command FIFO and the Control/Check field inside the CAN-bus. These two are combinational blocks. The final delay is the setup time for the registers in the Transmit Shift Registers.

Calculations:

- **Propagation delay** = (Flip-flop Propagation Delay)* (Wire and Capacitive Loading Delay Overhead) = 0.4 * 1.2 = 0.48 ns ~= 0.5ns
- **Combinational Logic:**
  - **Output Logic (Command Output)** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*2*1.2 = 0.48 ns
  - **Control/Check Field** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2 *3*1.2 = 0.72 ns

- **Next State Logic for Transmit Shift Register in CAN-Bus** = (Logic Gate Propagation Delay) * (Expected Number of Logic Gates) * (Wire and Capacitive Loading Delay Overhead) = 0.2*1*1.2 = 0.24 ns.
- 0.72ns + 0.24ns + 0.48ns = 1.44ns ~= 1.5ns

- **Setup Time for Transmit Shift Register in CAN-Bus** = (Flip Flop Setup Time) * (Wire and Capacitive Loading Delay Overhead) = 0.2ns * 1.2ns = 0.24ns ~= 0.3ns

Estimated Critical Path vs Actual Critical Path

According to the estimated critical path for our design, the largest register to register time delay was estimated as 2.4 ns with target clock period for the entire interface is 10 ns. However, on the actual design, the largest critical path with target clock period of 20 ns was around 12.957 ns. Also, in the synthesis mapped version of the top level of our design, it has shown that the critical path on the mapped was 4.0 ns. The reason why actual design's critical path with pads was about 19ns due to the very first register that the signal from the I/O pads encounter. For each critical path tested, the registers that are connected to the pads do cause most of the critical path delay. Once passing the very first register that encounters connection with pads, the subsequent combinational gates until the end do have time delay of nearly close to 0 ns. This is happening to all critical paths that has been generated. Check Appendix for more information about the actual critical path of register to register timing analysis.

Conclusion

Our target clock period for the module is 10 ns. As of now, our biggest register to register delay on estimation is 2.4ns. However, the actual biggest register to register delay on the layout was about 13 ns with clock period of 20 ns. This delay was unexpected since the first register hit once the signal enters from the I/O pad has a delay of about 12ns. We were unable to resolve this issue but were able to successfully generate an error free layout with the 20ns clock period.

4. Success Criteria

Fixed success criteria:

- Test benches exist for all top level components and the entire design: Successful
- Entire design synthesizes completely, without any inferred latches, timing arcs ,and, sensitivity list warnings: Successful
- Source and mapped version of the complete design behave the same for all test cases: Successful
- A complete IC layout is produced that passes all geometry and connectivity checks: Successful
- Correct area, pin count and clock rate:
    - Area : 4mm x 4mm
    - Pin Count: 112
    - Clock Period: 10ns

Design Specific Success Criteria

- Demonstrate by simulation of Verilog test benches that the complete design is able to successfully interpret data sent via the CAN bus protocol and convert it into AHB protocol. (2 Points)
- Demonstrate by simulation of Verilog test benches that the complete design is able to successfully interpret data sent via the AHB bus protocol can convert it into CAN bus protocol. (2 Points)
- Demonstrate by simulation of Verilog test benches that the complete design is able to properly determine the specific CAN transceiver based on the many modules sending data to it via the CAN bus. (2 Points)
- Demonstrate by simulation of Verilog test benches that the complete design is able to properly transmit data to the desired CAN transceiver via the CAN bus depending on data functionality. (2 Points)

Demonstration of the fixed Success Criteria:

1. All test benches can be found inside the project folder. The specifics can be found in the appendix at the end of this document.
2. Proof that we have no latches, timing arcs and sensitivity list warning can be found in the log files listed in the appendix section at the bottom. A screenshot for the overall designs log is shown below to prove this FSC as well.

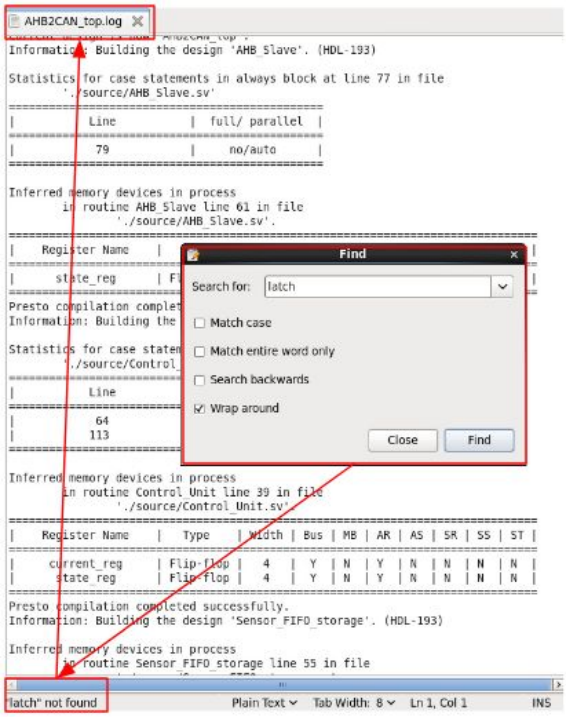


Figure 32: Check Latches

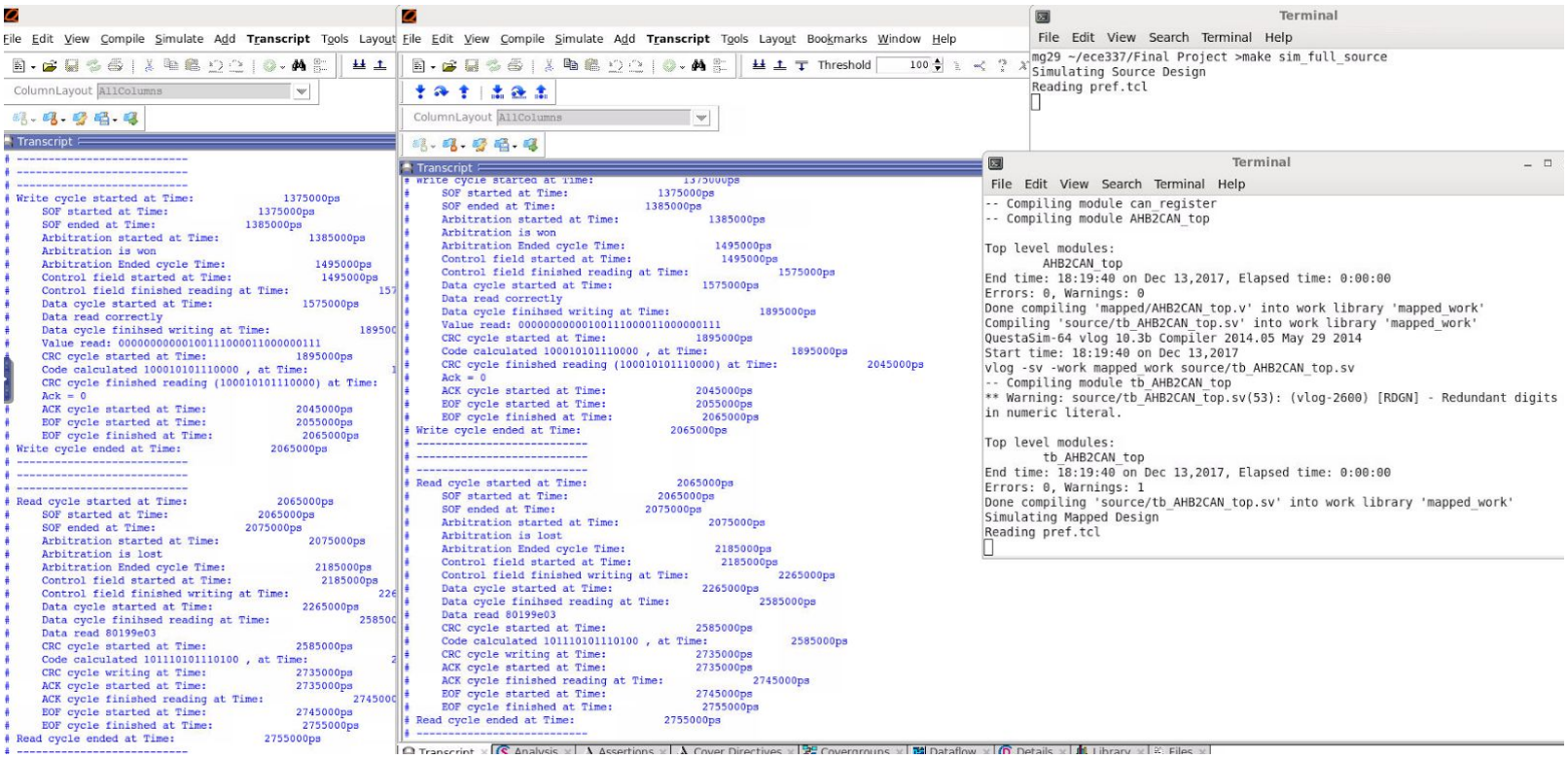
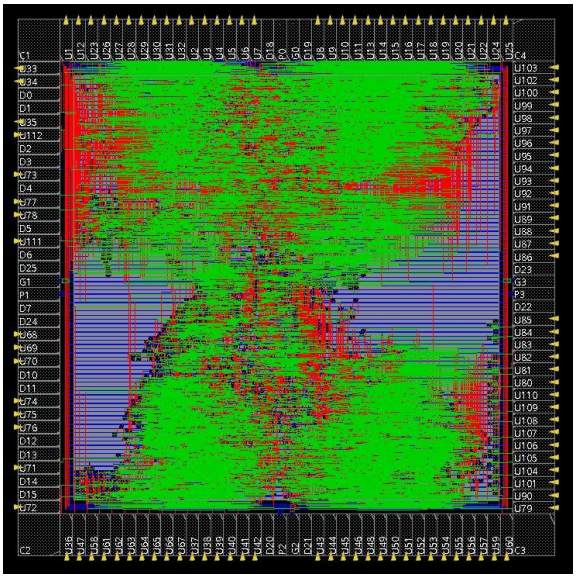


Figure 32: Source and mapped operation

3. The Picture above shows the result of running both the mapped and source versions backed to back (zoom into the picture for better resolutions). There is further proof of this in the Design verification. For further proof, both designs can be ran and compared.

4. Layout Generated for the design with no connectivity or geometry errors.

Figure 33: Layout Generated



5. a) Area Target: 4mm x 4mm.
Actual Area: 3mm x 3mm.

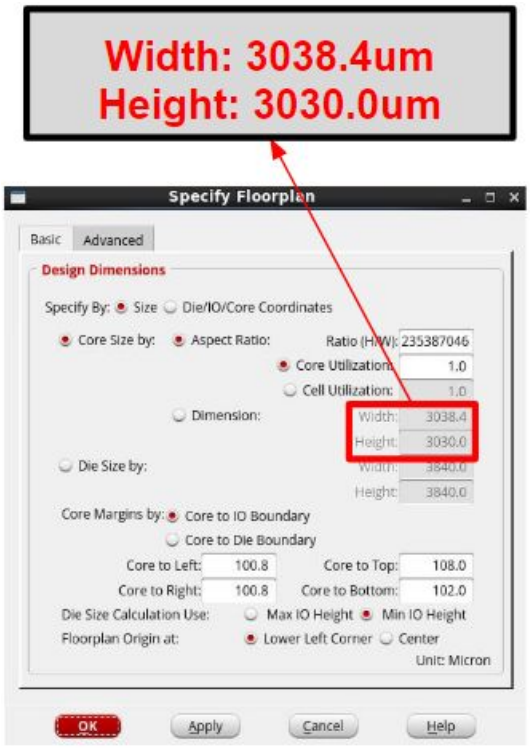


Figure 34: Calculated Layout Area

b) Target Pin Count: 112
Actual Pin Count: 112

c) Target Clock Period: 10ns based off the timing budgeting analysis.
Actual Layout Clock Period with no geometry and connectivity errors: 20ns (However still violated the critical path delay because of unresolvable reasons. The first register hit after the signal enters from the I/O pad has a 17ns delay. After that the delay is negligible.)
Synthesis Report Clock Period: 10ns. (Delay is 4ns)

Figure 35: Verification of layout :

```
############################################################
#  Generated by:      Cadence Innovus 16.12-s051_1
#  OS:                Linux x86_64(Host ID ecegrid-thin5.ecn.purdue.edu)
#  Generated on:      Wed Dec 13 16:33:59 2017
#  Design:            AHB2CAN_top
#  Command:           verifyConnectivity -type all -error 1000 -warning 50
############################################################
Verify Connectivity Report is created on Wed Dec 13 16:33:59 2017



Begin Summary
    Found no problems or warnings.
End Summary
```

```
############################################################
#  Generated by:      Cadence Innovus 16.12-s051_1
#  OS:                Linux x86_64(Host ID ecegrid-thin5.ecn.purdue.edu)
#  Generated on:      Wed Dec 13 16:34:00 2017
#  Design:            AHB2CAN_top
#  Command:           verify_drc
############################################################

No DRC violations were found
```

These images prove that we generated a layout with no geometric or connectivity errors.

5. Design Verification

This project was a success. We were capable of achieving all of our design specific success criteria for both mapped version and source version of the project. The following section is a specific breakdown of how we tested the design.

AMBA AHB-Lite protocol interfacing

- Shown in Demo: Yes
- DSSC(s) Proved: 1 & 2
- Highest Level of Design Module(s) involved:
    - Total Design/Chip
- Test bench expectations/requirements:
    - Have different test vectors/arrays for each type of sample bus transaction to be carried out
    - Have correct design response information ready for comparison purposes.
- No external or premade references are needed
- No pre/post processing is needed
- Main verification/test steps:

1. Send in data for a particular type of bus transaction in the AHB-Lite protocol (Ex. burst WRITE transaction, single WRITE transaction, READ transaction)
2. Compare design's output data with expected response values present in the response test vectors
3. If correct, repeat the above steps for all the following transaction types

The AMBA AHB-Lite protocol was tested on its own. We created a separate test bench that would check all the different transitions this module has to make. In the case of our project there was no master module present. We only had a slave module. In regular applications, the master is the microcontroller itself and in our case this was represented by the test bench.

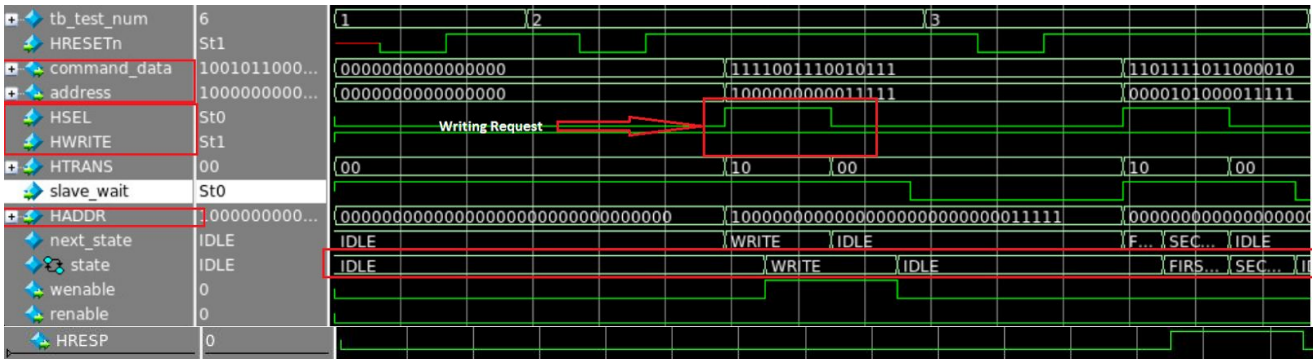


Figure 36: Test cases 1 to 3 for the AHB-Lite testbench

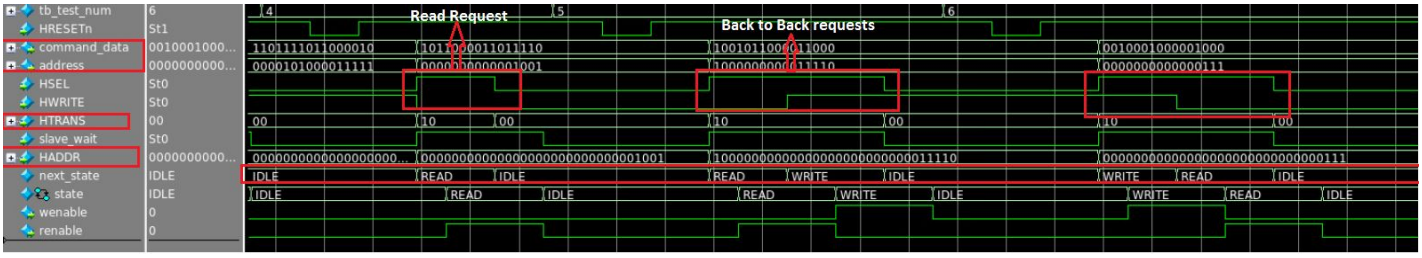


Figure 37: Test cases 4 to 6 for the AHB-Lite testbench

The timing diagrams above illustrate the functionality of the AHB-Lite bus module. As can be seen, there are only 5 states required to implement the design, namely IDLE, READ, WRITE, FIRST_ERROR and SECOND_ERROR. The two error states are simply to allow two error cycles to occur in case any condition is violated. As can be seen in the first figure in test case 2, a simple write function is taking place. The HSEL and HWRITE signals are asserted indicating that the master has selected the slave and wants to write data to it. Hence the state machine moves into the WRITE state and wenable is asserted so the data on the command_data bus gets stored into the corresponding FIFO. In test case 3, an invalid address (max allowed address is 32) is sent in on the HADDR bus, so that module goes into the first of the two error cycles from the IDLE state. The HRESP signal also

gets asserted when the module is in the error states telling the master that an error has occurred. In test cases 5 and 6, simple read and writes occur in successive cycles. Notice that the slave_wait signal is asserted whenever the module is in a state that is not IDLE indicating that it is busy and the master has to wait. The HTRANS signal (that indicates transfer type) also changes to 2'b01 which indicates a busy transfer (2'b00 indicates idle). Again in the corresponding states the renable and wenable signals are asserted to allow reading and writing. Also the HWRITE signal must be pulled low and HSEL must be pulled high for the renable signal to be asserted. This means that the master has selected the slave and wants to read data from it, meaning the data on the HRDATA bus gets read by the master when the module is in the READ state.

Correct enqueuing and dequeuing of data in the Command FIFO (Based on control Module)

- Shown in Demo: Only if Top level testing does not show this.
- DSSC(s) Proved: 4
- Highest Level of Design Module(s) involved:
    - Main Control Unit
    - Command FIFO
- Test bench Expectations / Requirements:
    - Have dedicated test bench for command FIFO and Control Unit
    - Test uses dummy data to confirm proper read enable and write enable signals from the Control Unit and proper queuing/dequeuing from the fifo.
    - Test bench must emulate the behaviour of CAN module and AHB Lite master.
    - Test bench will compare the values stored and outputted by the FIFO with the values expected to be stored and outputted from it.
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Start with an empty FIFO
    2. Check status of the simulated AHB Lite Master to determine when the control module must assert write enable.
    3. Send multiple 16 bit data strings
    4. Check if the FIFO is storing them in the correct order.
    5. Repeat steps 1 to 4 until the fifo is full.
    6. Wait for the simulated CAN Bus to say that it is ready
    7. Main control Unit sends read enable signal
    8. Check if the output of the FIFO is correct
    9. Repeat steps 6 to 8 until FIFO is empty
    10. Repeat the entire process until correct operation is guaranteed.

The correct enqueuing and dequeuing of values from the command fifo was first tested on its own. By creating a separate testbench we assured that the fifo was working as expected. This meant that

the pointers inside the fifo where increasing and decreasing correctly, the fifo full and fifo empty flags where both asserted at the right time and most importantly the data read and the data written where correct. . After that we tested it in along the entire design. By having our master send commands whenever it wanted the fifo would have to enqueue the values given to it. This was covered during the writing phase of the protocol. Since the CAN bus is slow, the fifo can get full rather quickly, so had to ensure that the HREADY flag was being de-asserted when the fifo was full. The next part of the test included using the CAN bus. Our read enable signal comes out of the can bus when it wins arbitration. This part of the test had to deal with the fifo empty. In case the fifo were empty, the can bus should not attempt to read data from it. Finally, by using both the waveforms generated and the $display() function in our test bench we were able to determine that the data was correct. This was very helpful while testing the mapped version of the code, since we could compare the correct results of the source version to the outputs of the mapped version. This method was successful and the mapped version behaved as expected.

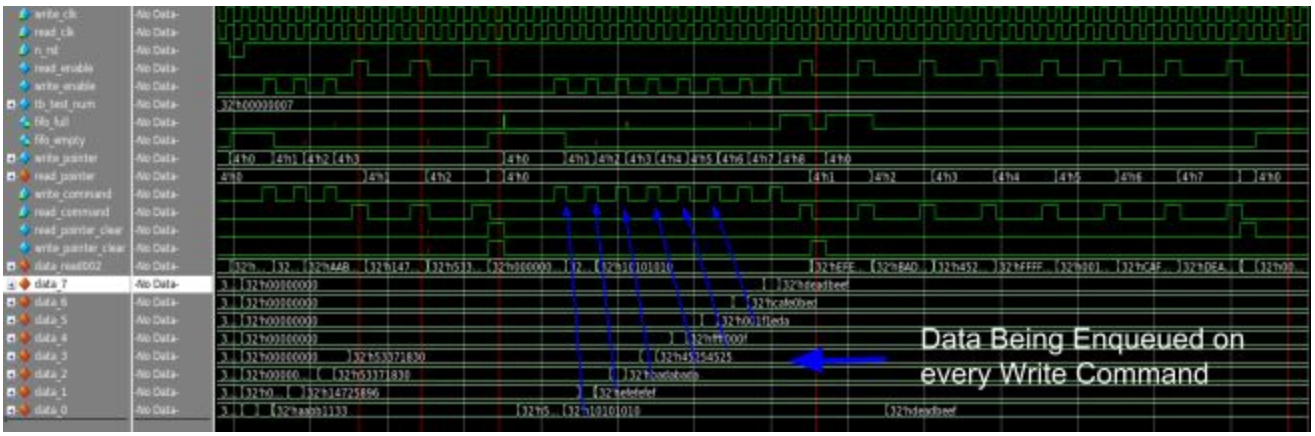


Figure 38: Data being enqueued in the command FIFO

In this image we can see how after the the AHB master makes several writing requests by asserting HSEL and HWRITE from the AHB master, which leads to the write_command signal being asserted. When this signal is asserted, the fifo stores the data in the write_data port, adds 1 to the write pointer and initializes the new value that the write pointer is pointing to. During this test the first read_command also enters the fifo, which leads to the fifo queueing the first value enqueued, lowers the write pointer and re-asserts the signal fifo_empty.

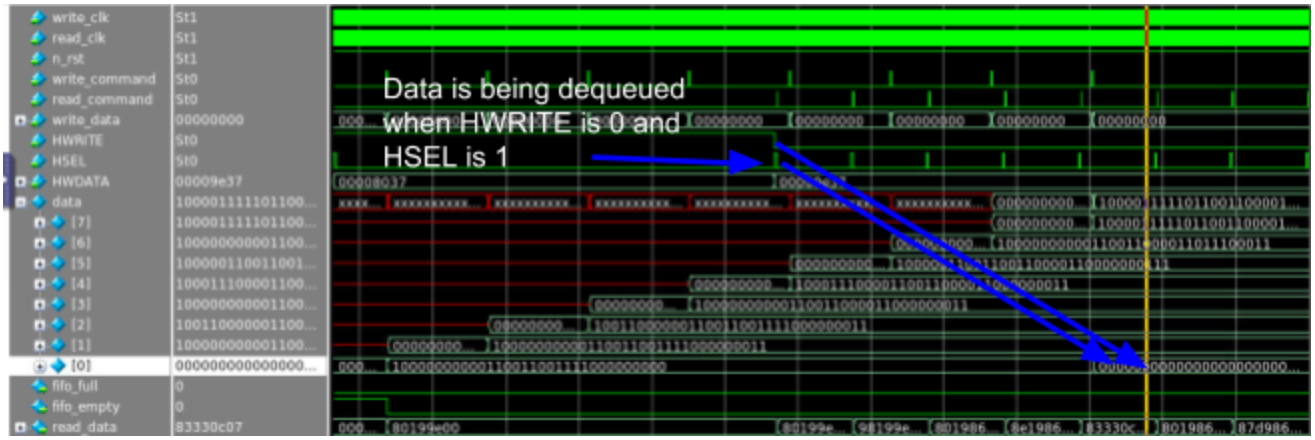


Figure 39: Data being dequeued from the command FIFO

This image shows two cycles where the can bus wins arbitration and as a result it sends a read command to the command fifo. This read signal causes the read_pointer to move and point to a different value. We can see that the pointer starts of as being 0. On the first read command it increases to 1 and the read_data value changes to be the value in the address 1. The values don't get reset until a the AHB master attempts to write a new command on it. We can see the same behaviour on the second write command, where the read_data value changes alongside the read_pointer. In this waveform we can also see the win signal asserted by the can module, which goes along the read read enable signal for the fifo.

Correct queueing and dequeuing of data in the Sensor FIFO (Based on control Module)

- Shown in Demo: Only if Top level testing does not show this.
- DSSC(s) Proved: 3
- Highest Level of Design Module(s) involved:
    - Main Control Unit
    - Command FIFO
- Test bench Expectations / Requirements:
    - Have dedicated test bench for sensor FIFO and Control Unit
    - Test uses dummy data to confirm proper read enable and write enable signals from the Control Unit and proper queuing/dequeuing from the sensor FIFO.
    - Test bench must emulate the behaviour of CAN module and AHB Lite master.
    - Test bench will compare the values stored and outputted from the sensor FIFO with the values expected to be stored and outputted from it. These values are the outputs of the simulated CAN module.
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Start with and empty FIFO

2. Check status of the simulated CAN Bus Module to determine when the control module must assert write enable.
3. Send multiple 16 bit data strings from the CAN Bus Module.
4. Check if the FIFO is storing them in the correct order.
5. Repeat steps 1 to 4 until the fifo is full.
6. Wait for the simulated AHB Bus to say that it is ready to receive data
7. Main control Unit sends read enable signal
8. Check if the output of the FIFO is correct
9. Repeat steps 6 to 8 until FIFO is empty
10. Repeat the entire process until correct operation is guaranteed.

The initial stages for this test were very similar to the previous test. We started off by assuring correct functionality of the fifo. Then we tested the sensor fifo inside of the top level design. For this to work we had to assure that the command fifo, control module and CAN bus module were all working as intended. The tests started of by sending the can bus many read requests, this lead to it attempting to write data on the command fifo. This signal is asserted when the crc value received by the can. It is the inverse of the ACK bit. After assuring that the values sent by the CAN were being correctly enqueued and following the restrictions regarding the fifo full. We had to to tell the sensor fifo to now dequeue those values. We accomplished this by sending reading requests to the AHB slave. We also had to make sure that the requests did not come through when the fifo was empty. After each request we would check if the values enqueued were coming out in the order of first in first out. We did this by checking at the waveforms and the outputs of the $display function in our test bench. Using the waveforms were were able to determine that the outputs from the $display functions were correct. This was very helpful while testing the mapped version of the code, since we could compare the correct results of the source version to the outputs of the mapped version. This method was successful and the mapped version behaved as expected.

Figure 40: Data being enqueued to the Sensor FIFO

This FIFO works just like the one previously discussed. The main difference between them is the context of their operation. The sensor fifo relies on a fully correct can protocol. The write command in this image is asserted during the EOF stage of the CAN bus protocol, but the value gets determined by the ACK value. If the can bus acks, then the write enable will go high. Just like with the previous fifo, the write pointer goes up and the value gets stored.

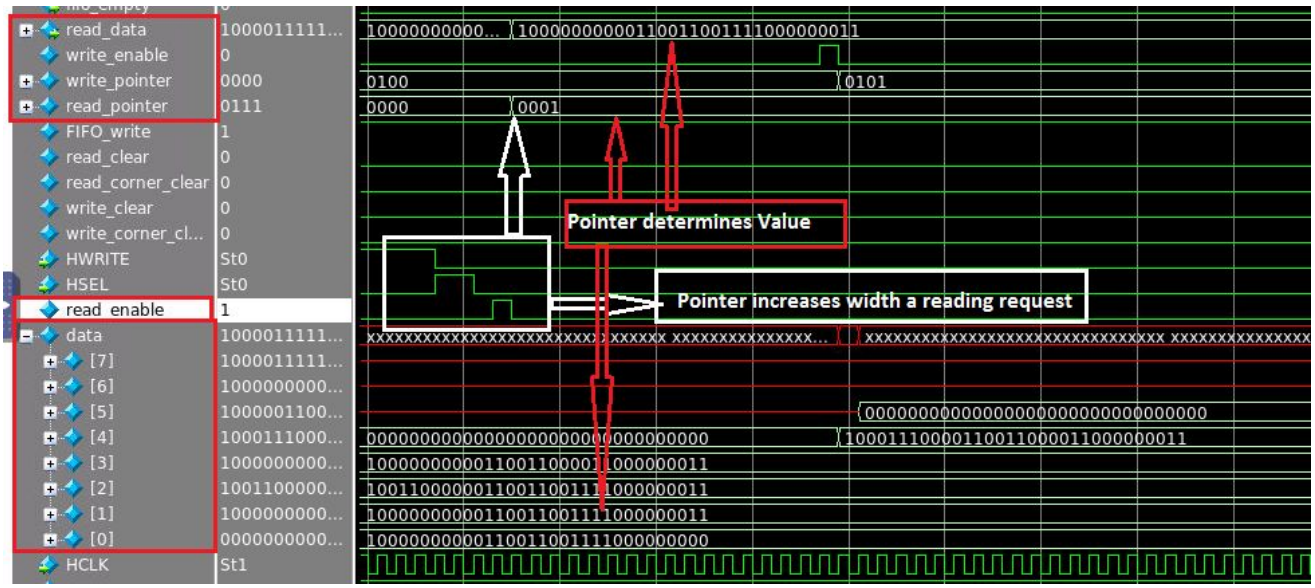


Figure 41: Sensor FIFO de-queueing data

This image shows how the sensor fifo de-queues data. This happens when the AHB-Lite Master keeps the HWRITE signal low and asserts the HSEL signal. This causes the read pointer to increment and the read_data changes its value to the one the read_pointer is pointing to. If we look at the last 2 images we can see that the values are correct and follow the order of first in first out.

Correct command Sent from Command FIFO to CAN Module

- Shown in Demo: Yes
- DSSC(s) Proved: 1 & 3
- Highest Level of Design Module(s) involved:
    - Can-Bus Interface Module
    - Command FIFO
- Test bench Expectations / Requirements:
    - Have dedicated test bench for Can-Bus interface Module and Command FIFO
    - Test uses dummy data to send command from command FIFO to Can-Bus interface module

    - Compare the command sent from command FIFO to Can-Bus interface module and verify all commands necessary to operate Can-Bus interface module are received.
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Send command to the Can-Bus interface module
    2. Check and validate if the data sent from command FIFO matches with the data received in Can-Bus interface module
    3. Within Can-Bus interface module, verify if the commands sent satisfy to operate module.
    4. Repeat step 1-3 for multiple commands

This test was performed after the previous two tests mentioned above. This had to be done in this order to assure that the command fifo would have correct values in store. The first thing we had to test was the arbitration module inside the can bus. We had to make sure that it was producing the read signal for the command fifo when arbitration was won. This was tested separately first and the results were positive. After confirming that the read fifo signal was coming in at the correct times we had to check weather the data was being stored correctly in the can module for later use in the DATA phase. This was done by observing the waveforms and which data value was to come out of the fifo first and comparing it to the value stored by the CAN module. During the DATA phase this data is to be shifted out MSB first. To test for the correctness of this data and to assure that the CAN protocol was running at the correct time. We created function that would continuously run and simulate the CAN bus behaviour but in reverse. By reverse I mean that if our CAN module was writing, then our test bench function was reading and vice-versa. This function would display all important data related to the CAN bus protocol, including which state it is in at any given time at what important values it has received or send. By both looking at the values being shifted out and the values and the values received by the test bench and displayed by the $display function we were capable of determining if the data was being shifted out correct. To minimize the time spent testing we created sort of a “fifo” inside the testbench. This was two function, one that would first enqueue the values send by the AHB bus and the second one would dequeue them in a first in first out order. This first function was called whenever the AHB lite module made a request and the second one was called during the dataphase of our test bench. Then the testbench would compare the value dequeued with the value received from the CAN bus. By using the waveforms to assure that the outputs from the testbench where correct we were also able to cut down the time spend testing the mapped version. Since we knew that the detailed outputs from the testbench where reliable we used them to confirm that the mapped version was correct. During this data phase we also had to calculate the CRC. Like all the other modules, we started of by testing this on its own. After confirming with sources that the long division method and CRC polynomial were correct it was only a matter of syncing it so that it would work during the data

phase. The testbench would also calculate the CRC code and state in the console if the codes matched.

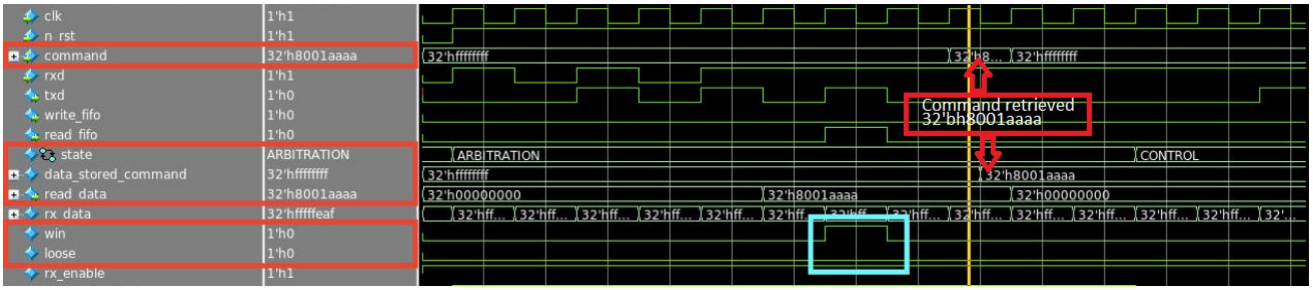


Figure 42: Correct Command Sent from Command FIFO to CAN Interface

This image above from Figure 35 shows that the data that is in Command FIFO has successfully transmitted its data to CAN Interface and those data match each other. This transmission happens during the arbitration state in the CAN Interface that when the CAN node wins the arbitration, then it requires the data from Command FIFO since that specific CAN module will behave as a read mode. As can be seen above from the picture, the 32-bits command input is being stored in data_stored_command for entire cycle since CAN module won the arbitration.

Correct data Sent from Can-Module to Sensor FIFO

- Shown in Demo: Yes
- DSSC(s) Proved: 1 & 3
- Highest Level of Design Module(s) involved:
    - Can-Bus Interface Module
    - Sensor FIFO
- Test bench Expectations / Requirements:
    - Have dedicated test bench for Can-Bus interface module and Sensor FIFO
    - Test uses dummy data to send command from Can-Bus interface module to sensor FIFO
    - Compare the data sent from sensor FIFO to Can-Bus interface module and verify data in sensor FIFO and data in Can-Bus interface module match
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Send data to the sensor FIFO
    2. Check and validate if the data sent from sensor FIFO matches with the data received in Can-Bus interface module
    3. Repeat step 1-2 for multiple data until sensor FIFO is full

To complete this test, we first had to test the arbitration module. We had to make sure that when it lost it didn't produce a read enable signal for the fifo. This would lead to data being lost. Also, we had to make sure that the CAN module would lose arbitration whenever the other nodes, in this case our test bench, wanted to write to it. After checking the arbitration module on its own and confirming its functionality, we went on test the testbench. In our design, the first bit of the 32 bits being sent represents whether or not the master is attempting to receive data from the slave. So when the testbench received a first bit value of 0, it should know that in the next cycle it should be in writing mode and win arbitration. Also, after writing a value, it should go back to reading mode on the next cycle. We tested the behaviour on its own by using the information displayed on each stage on the can bus cycle. To assure the CAN module and testbench were in sync, a state variable was created in testbench that represented showcased the state the machine was currently in. After assuring they were in sync, we tested them in the top level module. All previous test were necessary for this. By using the waveforms, we assured that the data was read correctly during the arbitration phase, which lead to the CAN module correctly losing arbitration and not asserting a read signal. Next up, we tested the data phase. This test was not very complicated since we had already established that the data was being sent and received correctly. We only had to compare the outputs of the $display function with the waveforms and the value that we told the testbench to send. During this data phase we also had to calculate the CRC. Like all the other modules, we started of by testing this on its own. After confirming with sources that the long division method and CRC polynomial were correct it was only a matter of syncing it so that it would work during the data phase. The testbench would also calculate the CRC code and state in the console if the codes matched. This part is crucial to the operation, because it let's us know that the data has not been corrupted. If the CRC calculated and send by the testbench matched the CRC calculated by the CAN module, then this module would send out a write enable to the sensor fifo. After that we checked that value enqueued the sensor fifo. After making sure that the waveforms in the source function matched the outputs to the console from the test bench we were able to use these outputs to the console to check the functionality of the mapped version. The results were correct.

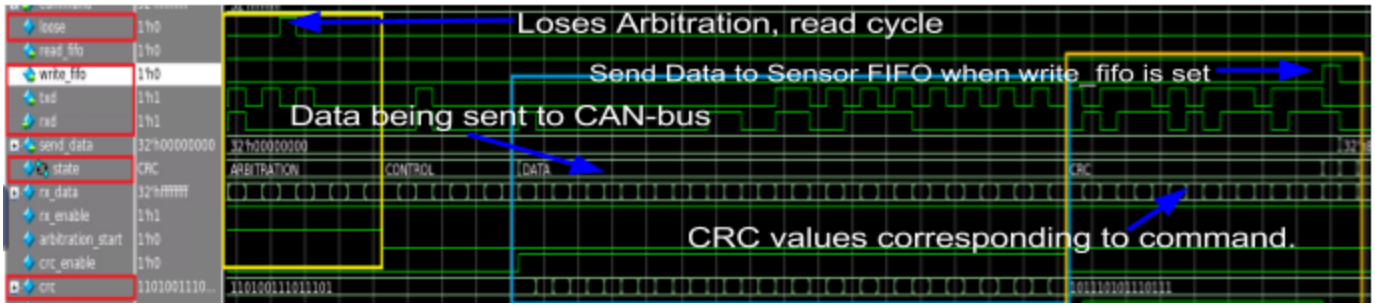


Figure 43: Correct Data Sent from CAN interface to Sensor FIFO

The image above in Figure 36 shows the waveform of the result that the CAN interface has successfully sent data to the Sensor FIFO. As can be seen above from the picture, the CAN node has

lost the arbitration, in which CAN node behaves as write mode. Once CAN module acts as write mode to Sensor FIFO, it then retrieves the data from the CAN-Bus through rx-sr during the data phase. On the CRC stage, CAN module then checks whether the cyclic redundancy check values match with what it has just received from CAN-Bus and then set write-fifo bit to 1 if the data transmission was correct, had no error, and notify Sensor FIFO to be prepared to receive new incoming data.

Correctness of CAN Module transferring to and receiving Data from Can-Bus

- Shown in Demo: Yes
- DSSC(s) Proved: 1 & 3
- Highest Level of Design Module(s) involved:
    - Can-Bus Interface Module
    - Command FIFO
- Test bench Expectations / Requirements:
    - Have dedicated test bench for Can-Bus interface Module and Command FIFO
    - Test uses dummy data to send data from one Can-Bus interface module to another interface module.
    - Compare the data that has been sent from one Can-Bus interface module with data that has been received from another Can-Bus interface module.
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Receive commands from sensor FIFO block for both Can-Bus interface modules
    2. Determine which one of two is writing data to Can-Bus or reading data to Can-Bus
    3. Can Bus interface module with writing data to Can-Bus write data to Can-Bus, and another module read data.
    4. Compare if the data read from Can-Bus matches with the data that has been transmitted.
    5. Repeat step 1-4 for multiple data values.

For this test we created a function that would follow the functionality of the can bus. This test bench acts as another node in the CAN bus and follows all the stages of the transmission that our actual CAN module follows. The first thing we had to check was the arbitration Both the simulated node and the actual node had to send their respective arbitrations values and both had to know whether they won or lost. To make this transmission work we had to make sure that both the testbench and the CAN module were working in sync. To do this, we first created a statetype variable inside the testbench that would track which stage the testbench was in. This allowed us to compare the value to the one from the CAN Module. Alongside this variable, we also printed several messages with the important data being received and sent by the testbench. This printed values also included their respectives times. After making sure that the processes were parallel we started checking the actual

values. Some stages parts of the design had already been tested separately, so after syncing the testbench and CAN module we had no issues with them. Among these modules are the CRC generator, Arbitration Module and the Acceptance filter. For the arbitration phase, the testbench will print weather or not it lost and where it lost. For the data phase, the testbench will save the 32-bit value sent by the can and then compare it to the one it was expecting. It know which value to expect, because those values get saved in the testbench when they are sent to the AHB slave. A correct data phase or incorrect data phase message will appear. This is the same of the crc cycle. The testbench will print the crc value it calculated and will compare to the one calculated and sent by the CAN Module. If the first bit of the data sent is a 0, then the testbench know that in the next cycle it must win. At this point it receives a higher arbitration value and wins arbitration. We checked this by making sure the read_enable signal for the command fifo stayed low during a read cycle for the CAN Module. All of the following stages are similar to the writing mode, but in reverse. Finally, we had to make sure that the CRC generated by CAN module matched the one the testbench sends out. If it did, then a write_enable signal would be asserted for the sensor value and we could compare the value enqueued to the value the testbench sent.

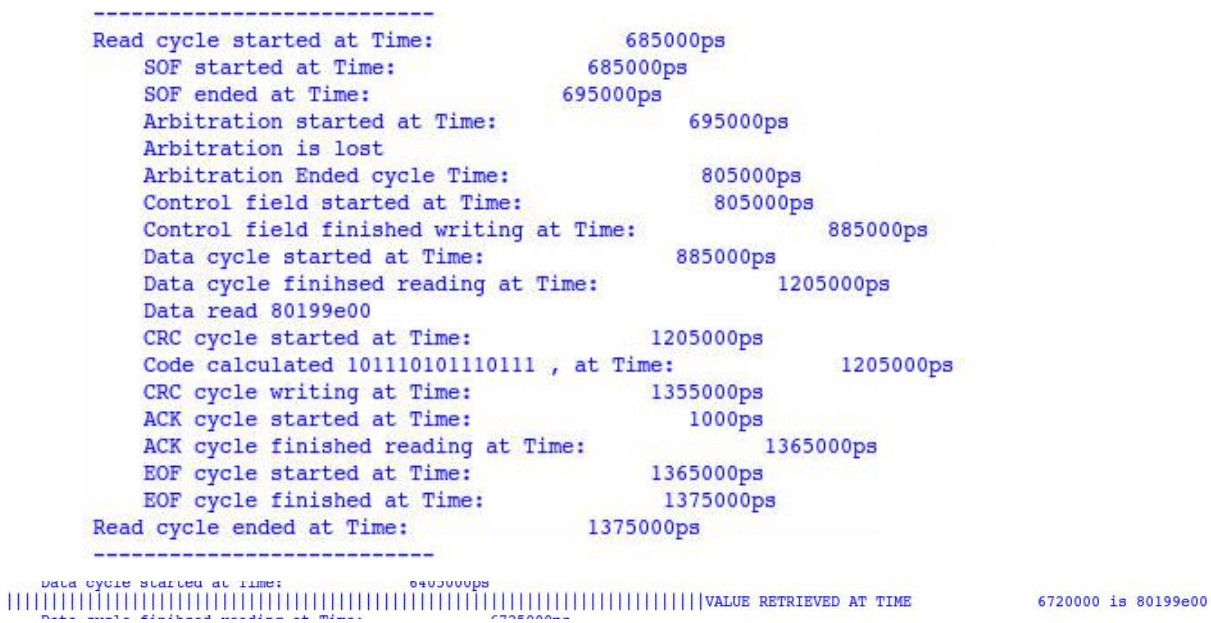


Figure 44: Correct Data Sent from CAN-Bus (Test Bench) and received from CAN-Module

The picture above in Figure 37 shows the test bench result that the correct number of bits and correct information of data have successfully been transmitted from CAN-Bus to CAN module. As can be seen above from the picture, during the CAN module write cycle, the data that has been read (command) is '80199e00'. Later in the write cycle of CAN module, when it is receiving data from CAN bus, it should be reading the same data command value. Therefore in 6720000 ps after the test bench has been initiated, the CAN interface has received the same value that was sent from CAN Bus.

```
# Write cycle started at Time:                    0ps
#      SOF started at Time:                    0ps
#      SOF ended at Time:                  5000ps
#      Arbitration started at Time:                 5000ps
#      Arbitration is won
#      Arbitration Ended cycle Time:               115000ps
#      Control field started at Time:               115000ps
#      Control field finished reading at Time:                195000ps
#      Data cycle started at Time:               195000ps
#      Data read correctly
#      Data cycle finihsed writing at Time:                515000ps
#      Value read: 00000000000000011010101010101010
#      CRC cycle started at Time:                515000ps
#      Code calculated 110100111011101 , at Time:                 515000ps
#      CRC cycle finished reading (110100111011101) at Time:                665000ps
#      Ack = 0
#      ACK cycle started at Time:                665000ps
#      EOF cycle started at Time:                675000ps
#      EOF cycle finished at Time:                685000ps
# Write cycle ended at Time:                685000ps
```

Figure 45: Correct Data send From CAN Module to CAN Node (Test Bench)

The previous image shows the outputs of our test bench. This output belongs to the first write cycle in the cycle the CAN module goes through. The first key value we can see is the arbitration. As you can see, since the is a writing cycle, the arbitration was won at time 5000ps. The next value is the data that was send. The test bench lets us know that it received the value it was expecting and prints it on the screen. After that the test bench will calculated the CRC code based on the data it received and compare it with the CRC code it received. Since both of these values are equal, the testbench will respond with an Ack bit of 0. This proves that the data is being correctly transferred from the CAN Module to the CAN Bus.

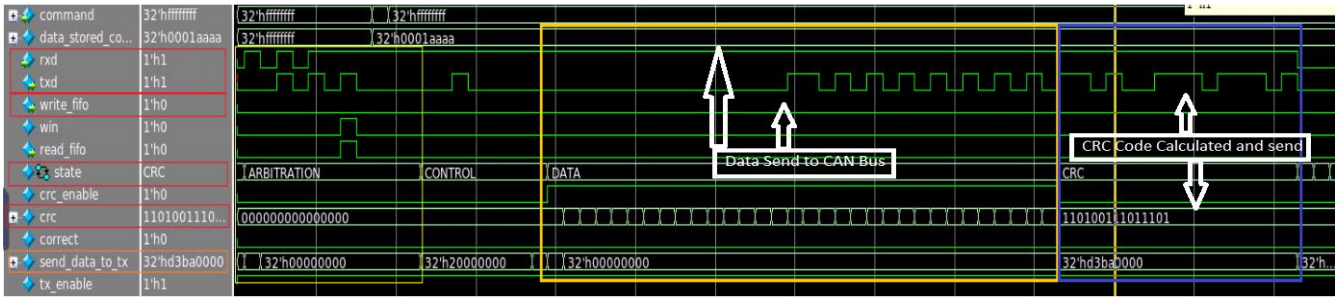


Figure 46: Writing Cycle CAN Module

This waveform represents the outputs of figure 38. As we can see, after winning arbitration, the CAN module stores the fifo's value in the data_stored_command register and prepares to send it during the data phase. This data phase will shift out the values stored in said register most significant bit first.

During this state, the CRC code will also be calculated. After the data phase ends, the CRC code pointed to in the picture will be send. Please note that all of these values match the picture above. This assured us that the communication with the CAN bus was correct.

6. Actual Division of Tasks

- **Vaibhav Ramachandran:** Command and Sensor FIFO, AHB-Lite Slave module, Main Control Unit and their respective test benches, layout generation.
- **Miguel Kulisic:** Top level test bench, Arbitration module test bench, CRC module test bench. Worked with Sang Hun Kim to test the CAN bus module.
- **Sang Hun Kim:** CAN Module and respective tests.

7. Appendix:

Account and directory where all files are located: mg43/ASIC_Design/ECE337_Project

- **Design Verilog Code Module**
    - **Top level Structural Verilog code:** source/AHB2CAN_top.sv
    - **AHB Slave:** source/AHB_Slave.sv
    - **Can Module control unit:** source/can_register.sv
    - **Command Fifo:** source/Command_FIFO_storage.sv
    - **Sensor Fifo:** source/Sensor_FIFO_storage.sv
    - **Main Control Unit:** source/Control_Unit
    - **Acceptance_filter:** source/acceptance_filter.sv
    - **Arbitration module:** source/arbitration.sv
    - **CRC Module:** source/can_crc.sv
    - **Counter:** source/flex_counter.sv
    - **Parallel to Series shift register:** source/flex_pts_sr.sv
    - **Series to Parallel shift register:** source/flex_spt_sr.sv
    - **RX module:** source/rx_sr.sv
    - **TX module:** source/tx_sr.sv
- **Test Benches:**
    - **Test bench acceptance filter:** Testbenches/tb_acceptance_filter.sv
    - **Test bench top level:** Testbenches/ tb_AHB2CAN_top.sv
    - **Test bench AHB Slave:** Testbenches/tb_AHB_Slave.sv
    - **Test bench arbitration module:** Testbenches/tb_arbitration.sv
    - **Test bench CRC module:** Testbenches/tb_can_crc.sv
    - **Test bench Command Fifo:** Testbenches/tb_Command_FIFO_storage.sv
    - **Test bench Sensor Fifo:** Testbenches/tb_Sensor_FIFO_storage.sv
- **Docs:**
    - **AHB data sheet:** Docs/ahb_data_sheet.pdf

    - **Final Presentation:** Docs/final_presentation.ppt
    - **Final Report:** Docs/final_report.pdf
    - **CRC Code:** Docs/crc.pdf
- **Mapped files:**
    - **Top level module:** mapped/AHBCAN_top.v
- **Reports:**
    - **Design top level report:** Synthesis Reports/AHB2CAN_top.rep
- **Log Files:**
    - **Design Top Level Log File:** Logs/AHB2CAN_top.log
- **Timing Reports:**
    - **Critical Path Timing Analysis Report:** Layout/Timing Reports/AHB2CAN_top_postRoute_all.tarpt
- **Connectivity Reports:**
    - **Connectivity check:** Layout/Connectivity Reports/AHB2CAN_top.conn.rpt
    - **Geometry check:** Layout/Connectivity Reports/AHB2CAN_top.geom.rpt
- **Scripts:**
    - **Makefile script:** makefile
    - **Full_run.tcl:** Layout/full_run.tcl
    - **Innovus.io:** Layout/innovus.io
    - **Init.tcl:** Layout/init.tcl