ToiletCleaner Low-level Controller: Technical Report

Mahmoud Omar Ouali
RoboHub Eindhoven
Fontys University of Applied Sciences

Contents

Table of Content

General architecture

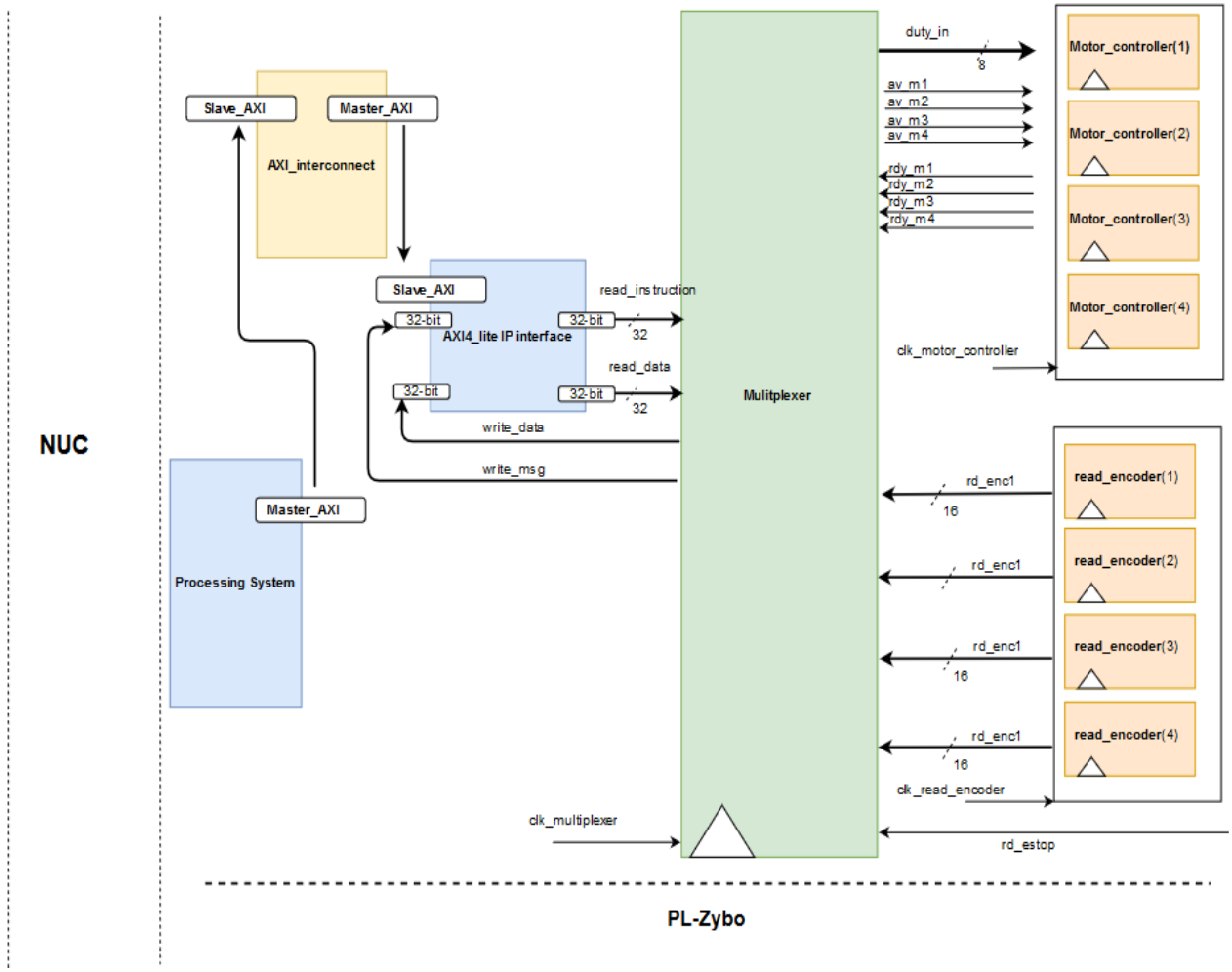


Figure 1 : The general architecture of toiletCleaner programmable logic

The low-level control logic will be responsible on controlling the motors, reading the encoders and the emergency-stop, based on the instructions and data received from the processing system. All the logic blocks implemented in the FPGA will be running in parallel, which increases the performance of the system and it's real time capabilities. The low-level design was divided into several blocks that has a specific function.

- Motor controller block ***(slave executor)*** will be responsible on controlling the PWM signal of the four motors of the AGV based on the duty received from the processing system.
- Read encoder block ***(slave executor)*** will be responsible on reading the current four encoder values of the AGV motors

- Multiplexer block *(**slave decoder***) will be responsible on decoding the processing system instruction and manage the flow of the data between the slave blocks and the master block.
- AXI4_lite IP interface *(**slave interface**)* holds four (32-bit) registers where the processing reads or writes data
    - Processing system write registers: read_instruction, read_data
    - Processing system read registers: write_data, write_msg
- Processing system *(**Master interface**)* is in instantiation of the processing system interface in the programmable logic.
- AXI_interconnect connects the AXI memory mapped processing system to the memory mapped slave AXI4_lite IP interface

Further details on the design of each block will be discussed in the following sections.

AXI4-Lite interface

Advanced extensible interface 4 (AXI4) is a high performance, synchronous and parallel communication interface between the processing system and the programmable logic, it can support multiple master and slaves. The AXI4-lite is a point-to-point and bidirectional interface between the AXI-interconnect and the AXI_lite interface block

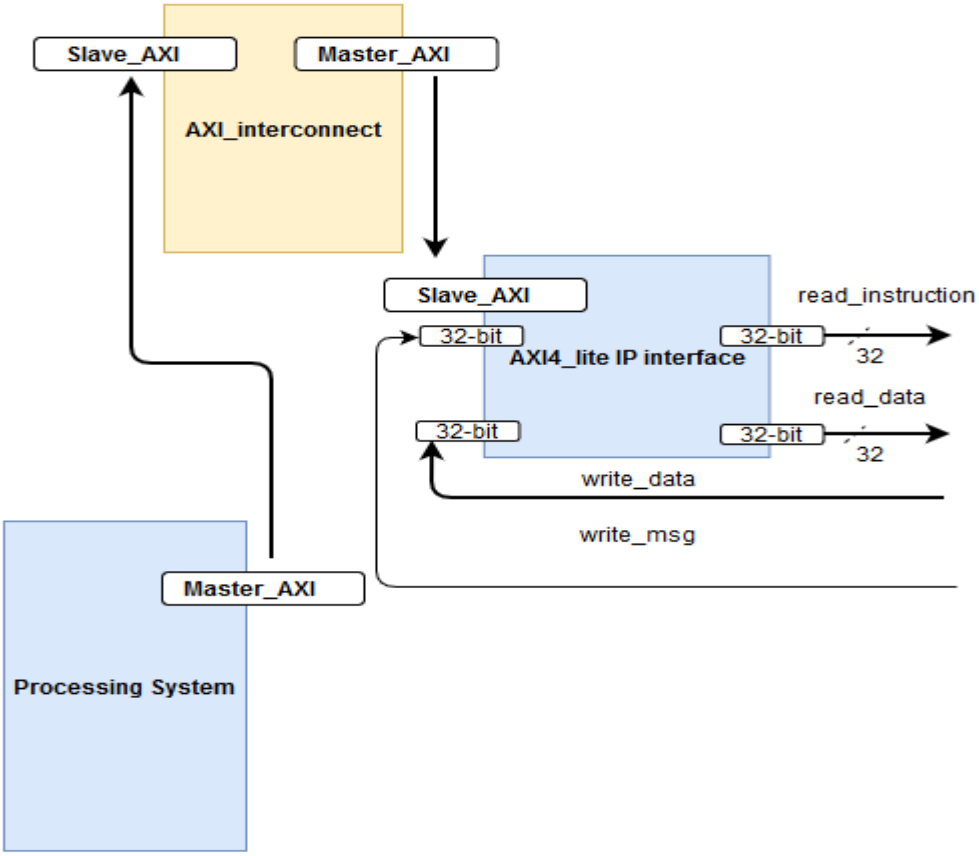


Figure 2: AXI-lite4 interface

The AXI4-lite interface consists of five channels:

- Read address
- Read data
- Write address
- Write data
- Write response

The AXI4-lite consists of two transaction:

- Read Transaction: Master (*processing system*) wants to read data, and the mechanism of reading is described below
    - Master inserts the address register on the Read address channel and indicates its ready to receive data on that address
    - The slave (*AXI4_lite IP interface*) puts the requested data the Read data channel and indicates it has sent the data
- Write Transaction: Master (*processing system*) wants to write write data, and the mechanism of writing is described below
    - Master inserts data on the Write data and the address of the data in Write address and indicated that the address and data are valid
    - The slave reads the address and the data
    - The slave writes a valid response on the Write response channel to indicates the success of the process.

All the transactions above are controlled by a full handshake procedure. The mechanism of AXI is abstracted and synchronized by an AXI clock (50 MHz).

The architecture used in the design *Figure2* uses four registers

1) Registers where the processing system writes instruction and data. Instruction represents which action must be taken and the data represents the speed that user is aiming for.
    - *Read_instruction*: each execution will have a certain instruction as explained below

      1 => Read encoder 1
      2 => Read encoder 2
      3 => Read encoder 3
      4 => Read encoder 4
      5 => Change the speed of motor 1 to the duty specified in read_data
      6 => Change the speed of motor 2 to the duty specified in read_data
      7 => Change the speed of motor 3 to the duty specified in read_data
      8 => Change the speed of motor 4 to the duty specified in read_data
      9 => Read e-stop and write data to write_data

    - *Read_data*: The processing system writes the speed aimed by the user
2) Registers where the processing system is reading from messages, and encoder data
    - Write_data: where processing system will be reading encoder data or e-stop
    - Write_msg: Where the slaves are writing messages to the processing system about their progress (will be explained in multiplexer design)

This advantage of this architecture is economizing the AXI registers. Instead of giving each slave executor an AXI_interface, I have only implemented one slave AXI_interface that will be communicating with processing system. And it's the slave decoder role to command the slave executors based on the instructions received from the processing system.

Motor Controller(slave executor)

The PWM is a method to generate analogue signals from digital sources, the duty controls the power delivered for a certain fixed period by setting the signal high for the duty-time thus controlling the speed of the motor. The PWM controller generates a PWM signal based on the duty given by the processing system.

The motor driver board (MDD10A) used to control the AGV motors can handle a maximum PWM frequency of 20 KHz. For safety measurements the Motor Controller will deliver a maximum PWM frequency of 10 KHZ.

Digital design of Motor Controller

To ensure error-free data and synchronize between the motor controller block and the multiplexer a full handshake protocol was implemented. Ready signal indicates that the motor is ready to receive data and Available signal indicated that the multiplexer is ready to send a new duty to the motor controller.

The design of the motor controller was implemented using the Controller-Datapath architecture. The controller will be responsible with interfacing the block the multiplexer and control the flow of the Datapath. The Datapath will be responsible for storing the duty and counting the period and duty. The design will be synchronized by rising edge of a clock frequency.

To achieve a maximum period of 10 KHZ ➔ Counter of 8-bit was designed with a input clock frequency of

$$f_{input} = 2^8 \times 10 = 2.56\ MHz$$

The Clocking wizard IP from Vivado will be used as a source for digital block clocks. The clock frequency of the Motor Controller block is chosen to be 2.56 MHz as demonstrated above.

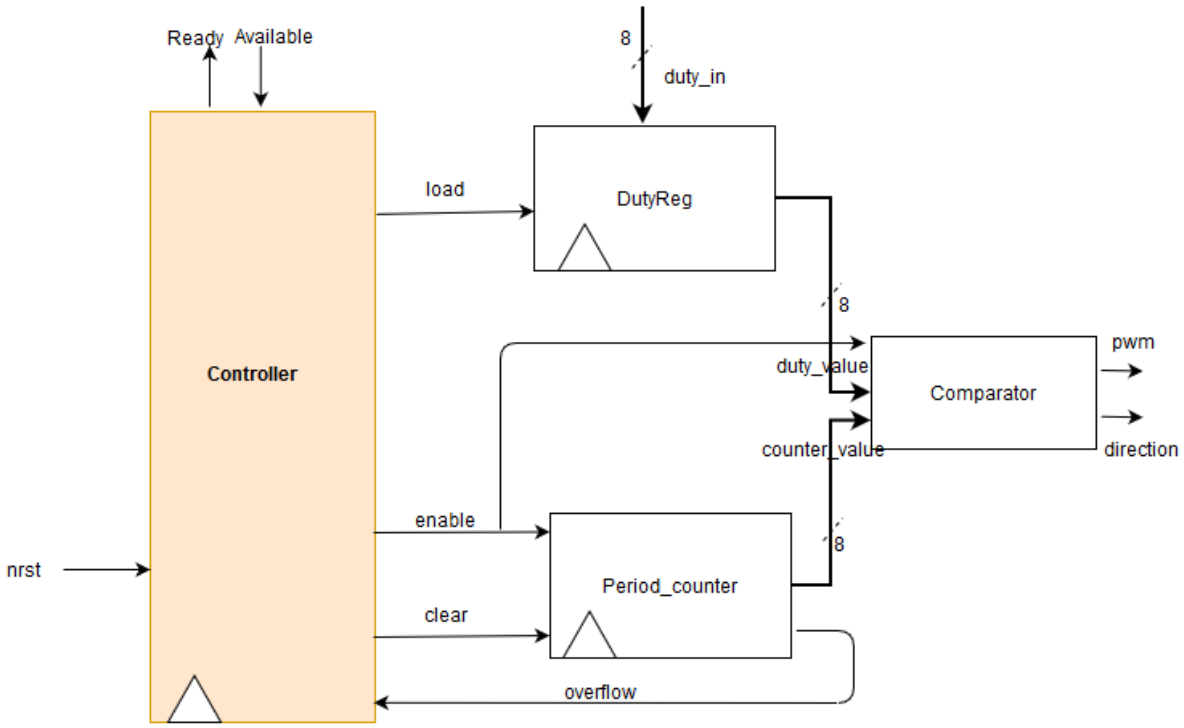


Figure 3: The digital design of motor Controller

As illustrated in *figure3,* the design was implemented using the Controller-Datapath Architecture.

- The dutyReg is designed to store the duty data when given the command load.
- Period_counter is counter designed to count after each clock pulse to keep track of period when it's enabled
- Comparator is designed to switch PWM high when counter_value < duty_value and vice versa

The motor driver board is set on locked anti-phase mode

- Duty cycle < 50% ➔ Rotation clockwise
- Duty cycle> 50% ➔ Rotation anti clock wise
- Direction pin is always high when the comparator is enabled

The ASM logic of the controller

The flow chart illustrated in *figure4* represents the logic of the controller

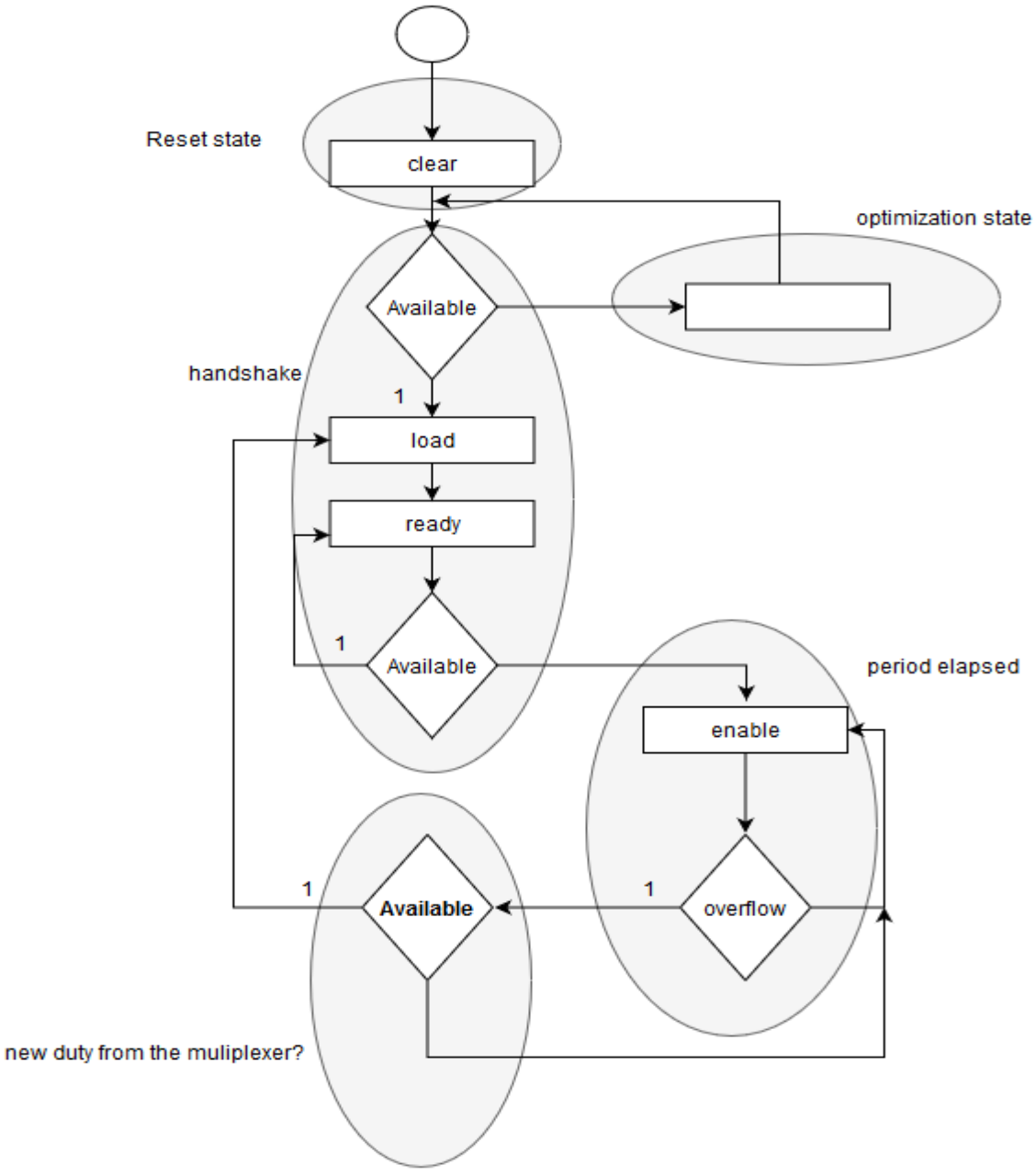


Figure 4 ASM of the controller

- Each states tales one clock pulse ➔ $T_{state} = \frac{1}{2.56} = 390\ ns$

- Reset state is starting state when the system is rested, the controller make sure to bring the Datapath to the initial state by clear command
- Optimization state was added in case no duty was available, to avoid multiple unnecessary clear commands
- The full handshake procedure to synchronize between the motor controller and the multiplexer
- After the close of the handshake, the controller Enables Command the Datapath and wait for the period to elapse to check for new speed. In case no new future speed, the controller keeps the keeps the present speed.
- The controller will not check for a new speed till time elapse ➔ $T_{elapse} = 390 * 256 = 99.88\ us$

Simulation

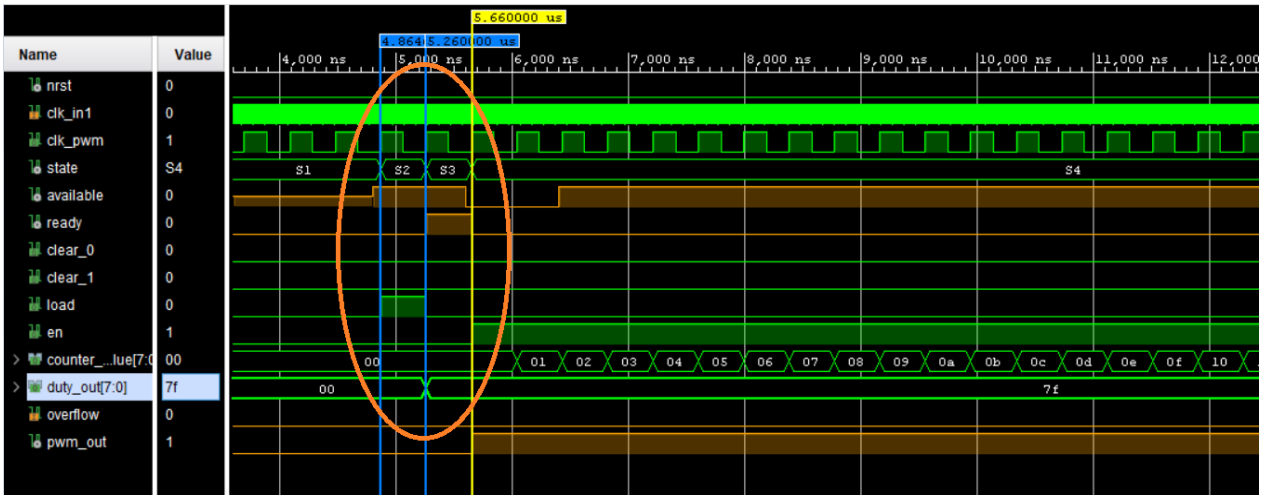


Figure 5: Simulation of handshake procedure

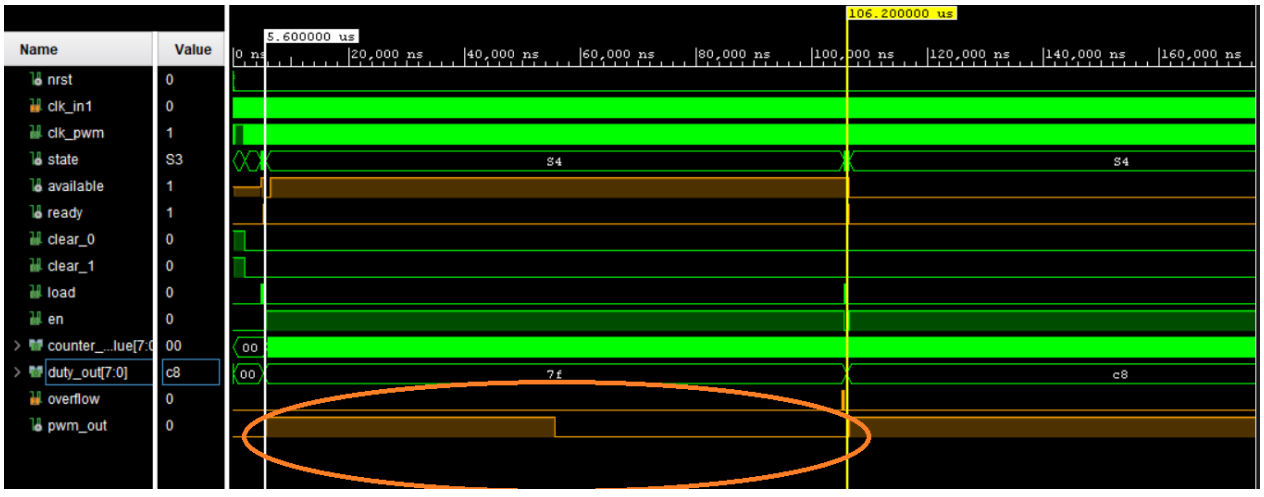


Figure 6: Simulation with a duty 50%

- As illustrated in *figure YY*, the handshake procedure is as expected with ASM.
- As illustrated in figure XY, the PWM generates a duty of 50% and at the end of the period an interrupt is generated when a duty of 128 is given
- The duration of a period is 99.89 us ➔ Maximum frequency of 10 KHz
- The simulation shows that the systems works as expected to be. The test bench files for simulation can be found in the appendix

Hardware Test

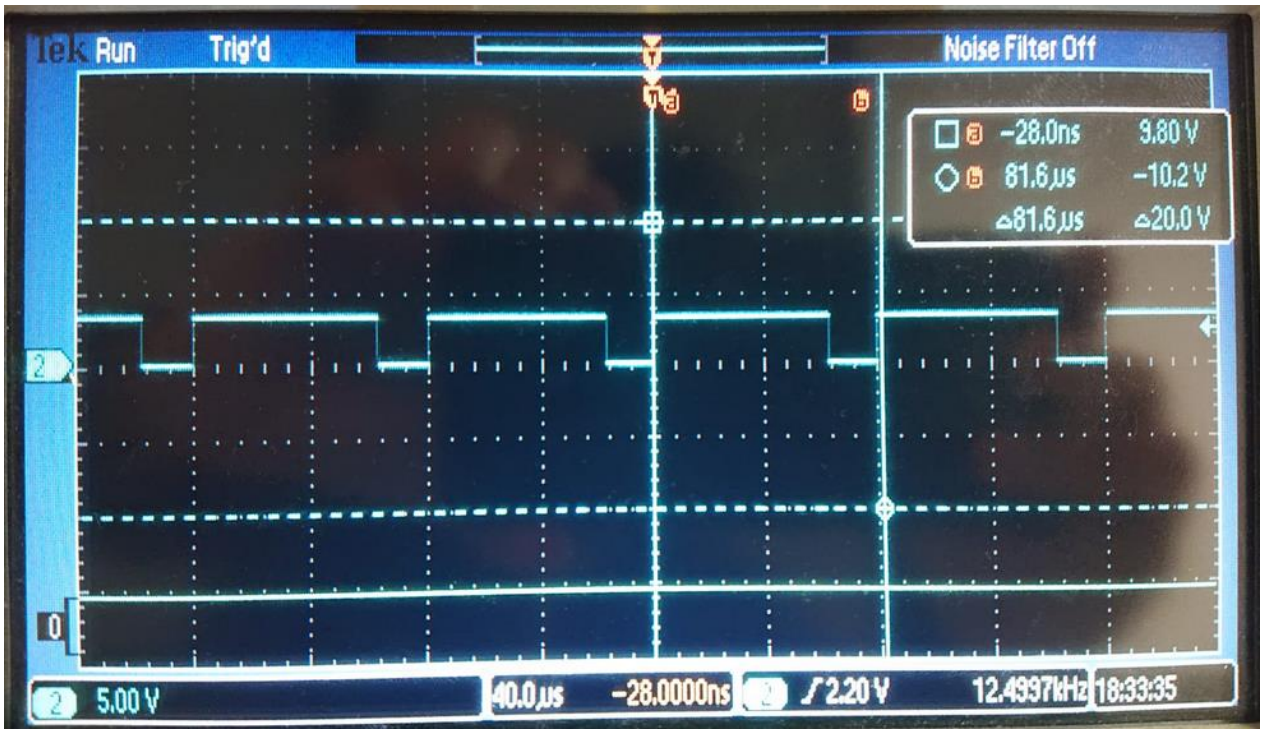


Figure 7: Test of the output PWM signal

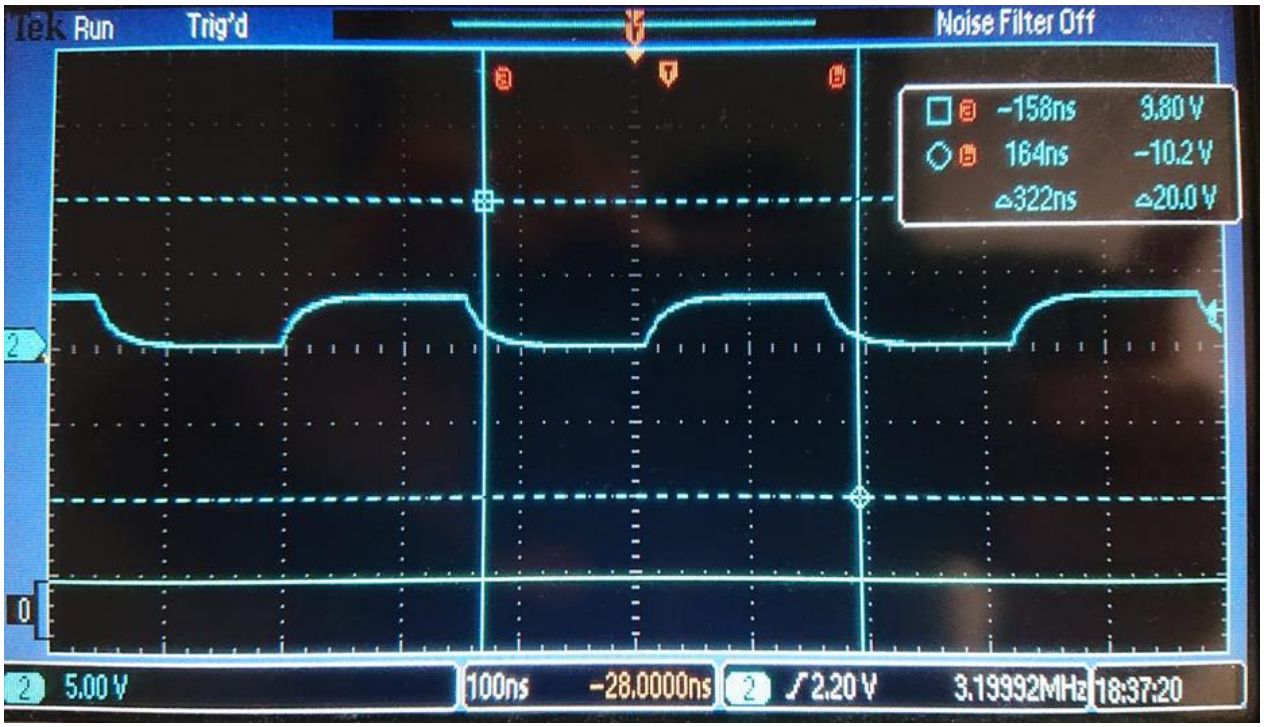


Figure 8: Output of the motor control block

- As illustrated in Figure7, the period of the clock is 81.us ➔ error of 18% is introduced
- As illustrated in Figure8, the period of the motor controller ➔ error of 18% introduced

The error is due to the Pmod port of the Zybo board, Pmod standard is used in the experiment. Pmod standard cannot handle high frequencies. An error of 18% is accepted.

Read Encoder (slave executor)

The read encoder block task is to read continuously the AGV encoder when the system is started for feedback sensing. The encoder used is a quadrature incremental encoder. The quadrature encoder generates two electrical signals (A and B) that are 90 degrees out of phase.

- In case A is leading B that means that motor is rotating clockwise
- In case B is leading A that means that motor is rotating anti-clockwise
- According to the datasheet of the encoder (maxon ENX), the encoder has 1024 counts per turn. To increase the resolution I will be sampling each edge ➔ $CPT_{new} = 1024 \times 4 = 4096$

$$Res = \frac{360°}{4096} = 0.088°$$

- According to the datasheet the maximum operating frequency is $f_{op} = 4\ MHz$. Since I will be detecting the four edges ➔ $f_{op4} = 16\ MHz$.
- The system to be designed according to the Nyquist theorem needs to have a sampling frequency $f_{samp} = 2 \times f_{op4} = 32\ MHz$ ➔ The frequency chosen will 100 MHz

Digital design of the read encoder

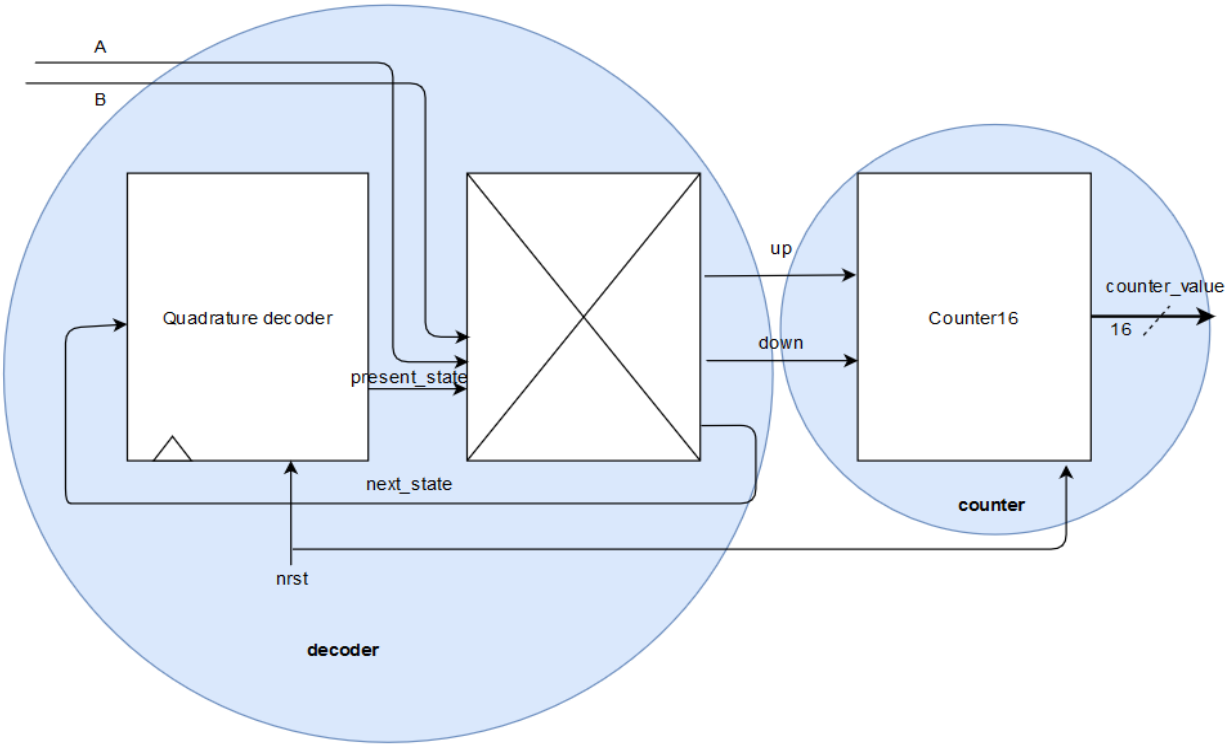


Figure 9: Read encoder digital design

The read encoder as illustrated in *figure9* is designed in two blocks, the decoder block and the counter block

- The decoder will be detecting the edges of A and B and whether A is leading or B and based on that it will be giving commands to counter to count up or count down
- The counter is designed to be 16-bit counter. The number of full rotations the counter cant detect before resetting is $N_{rot} = \frac{2^{16}}{4096} = 16\, rotations$
- The decoder is designed as a mealy machine state where the output depends on the state and the inputs A and B

The decoder logic

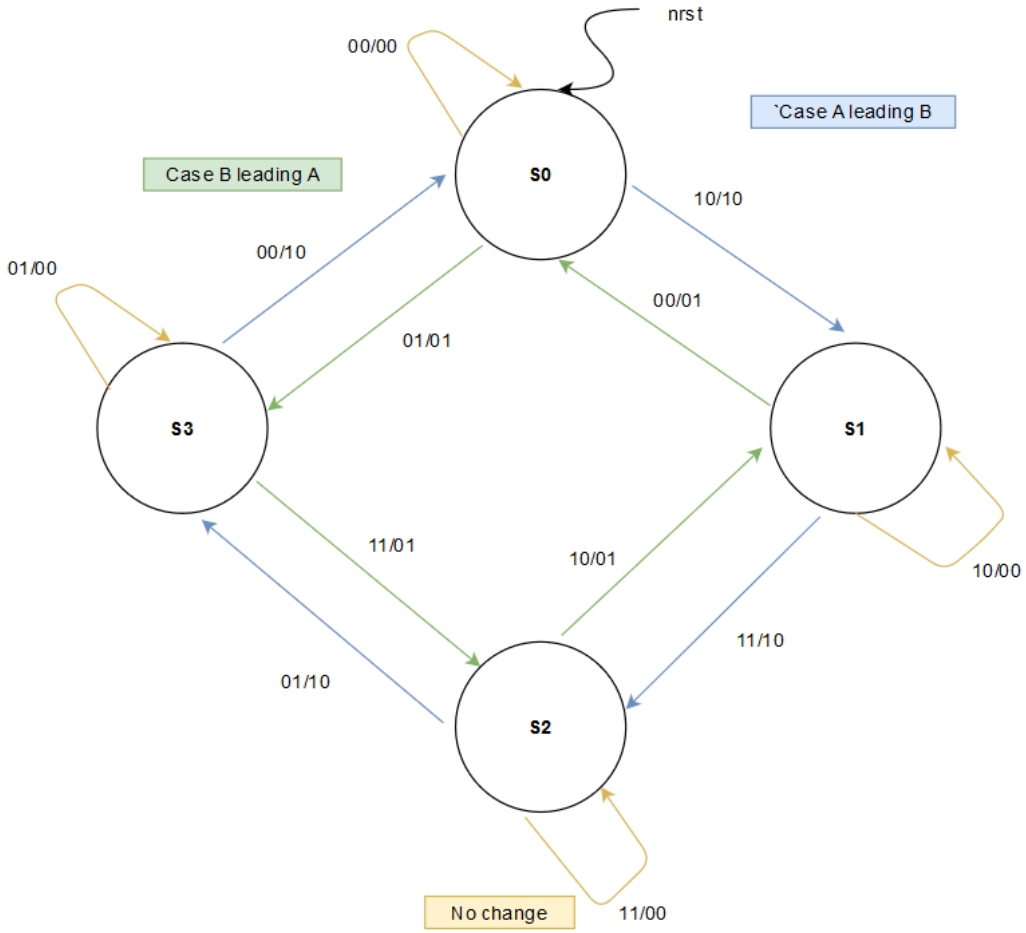


Figure 10: Decoder logic

As illustrated in Figure10

- Case A is leading B ➔ Up goes high for one clock cycle (10 ns), the counter will count up after 10ns
- Case B is leading A ➔ Down goes high for one clock cycle, the counter will count down after 10ns
- Case no change ➔ output is zero, waiting for a change to be detected.

Simulation

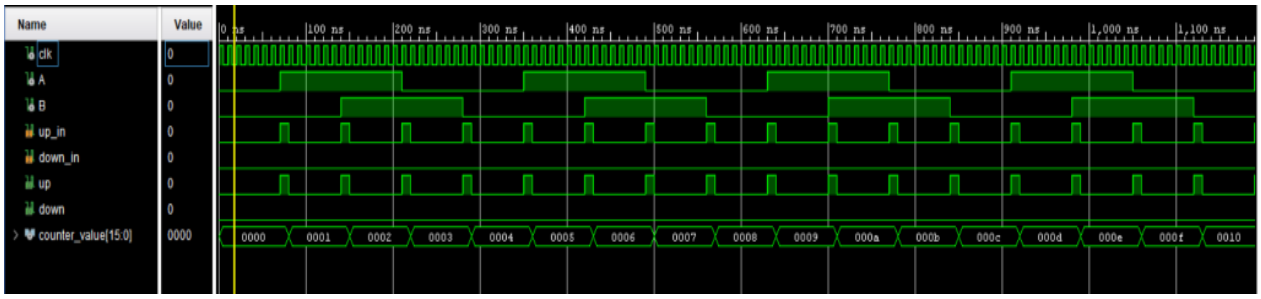


Figure 11: Simulation in case of A is leading B

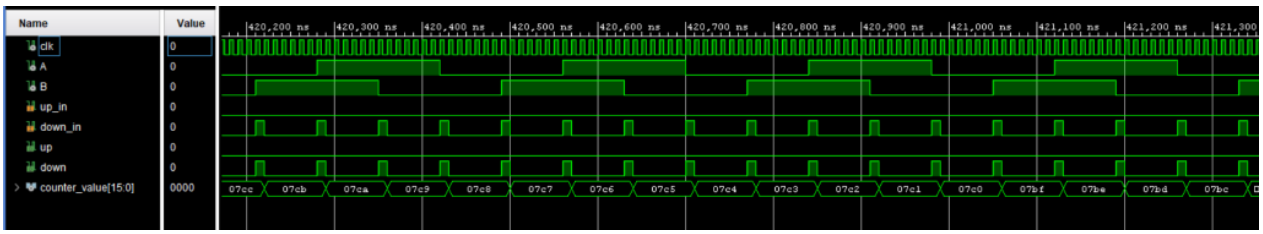


Figure 12: Simulation in case of B is leading A

- When A is leading B, the counter counts up (rotating clockwise)
- When B is leading is A, the counter counts down (rotating anti-clockwise)
- When A and B don't change, the counter latches its current value (no rotation)
- Sampling is executed on edges

Hardware Test of Encoders

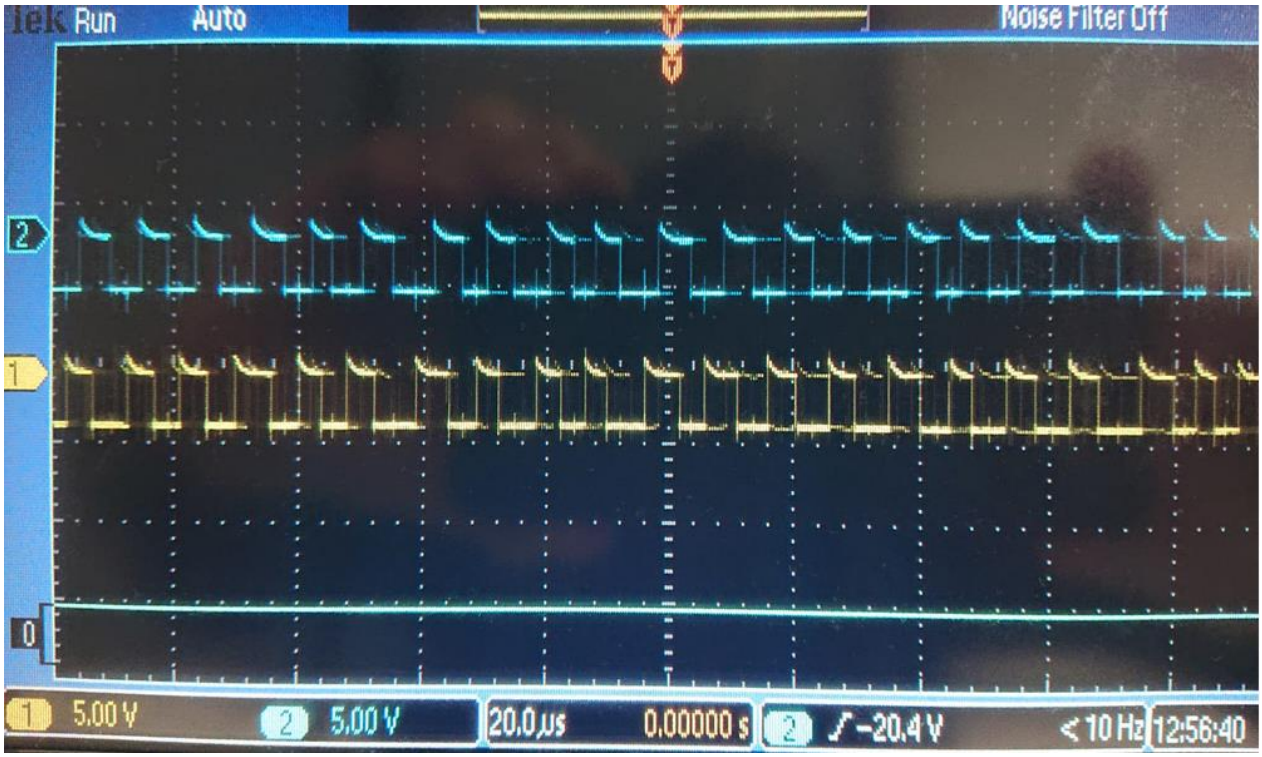


Figure 13: Hardware test of the encoders

As illustrated in *figure13* the output encoder signals are 90 degrees out of phase, the challenge is that sometimes when A is at low logic a fast peak occurs that may fool the system and trigger it as if A was high. A solution could be implementing a low pass filter digitally and filter out the noise ➔ Not implemented due to time constraints.

Multiplexer (slave decoder)

The multiplexer block plays the role of slave decoder, it decodes the instructions of the master and pass them to the slave executor blocks. The multiplexer plays a role of channel communication between the slaves and the master based on the protocol explained in AXI-4 lite interface chapter.

- The master is sending the instructions to execute a certain task
- The slaves are updating the master during their execution phase

Digital Design of the multiplexer

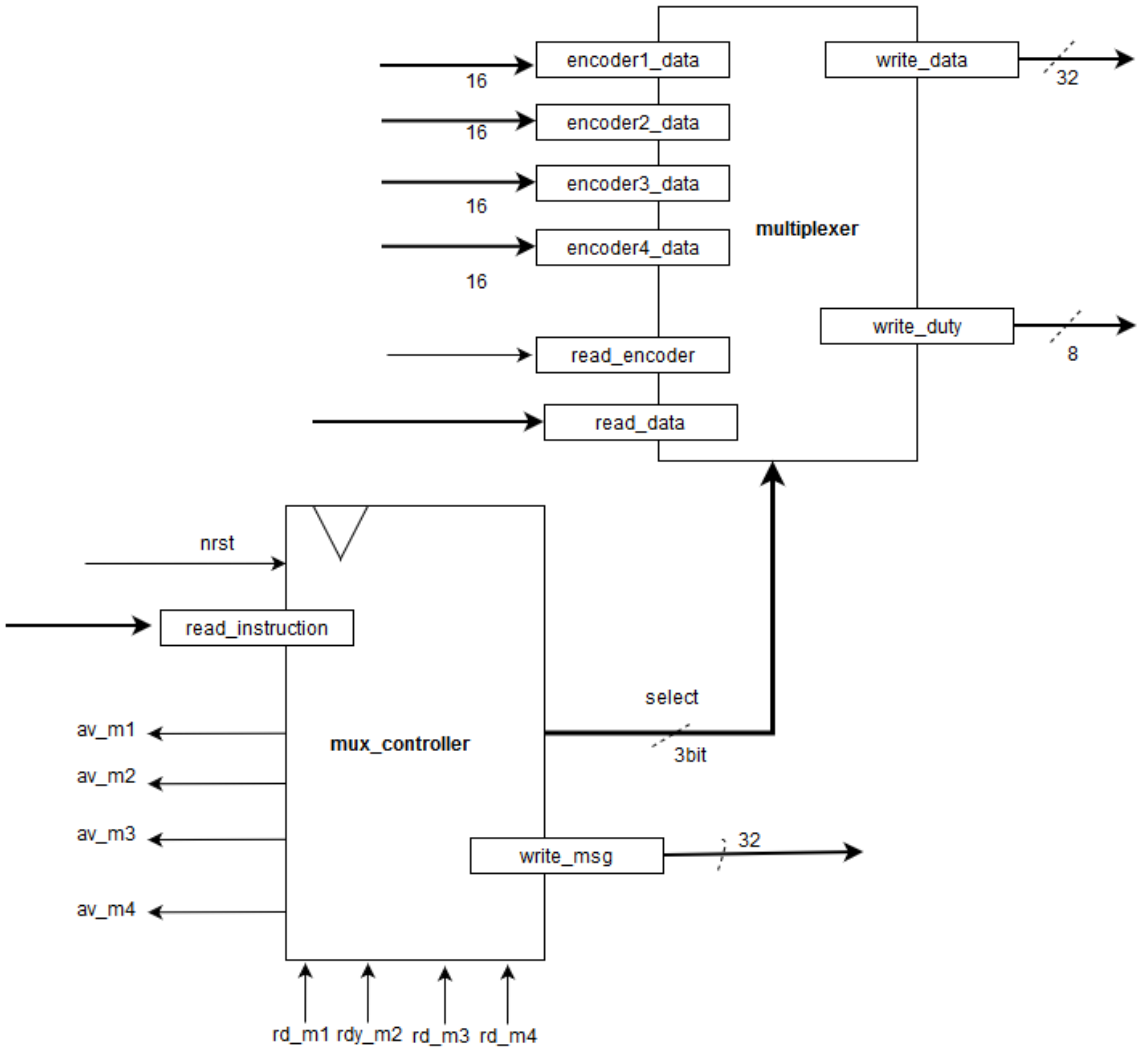


Figure 14: Digital design of the multiplexer

As illustrated in *figure14*, the multiplexer block is designed using the controller-slave architecture, based on the instruction that the mux_controller is reading it will enable the write message to flow to the right block. It could be viewed as the bridge between the processing system and slave executors block

State Logic of the mux_controller

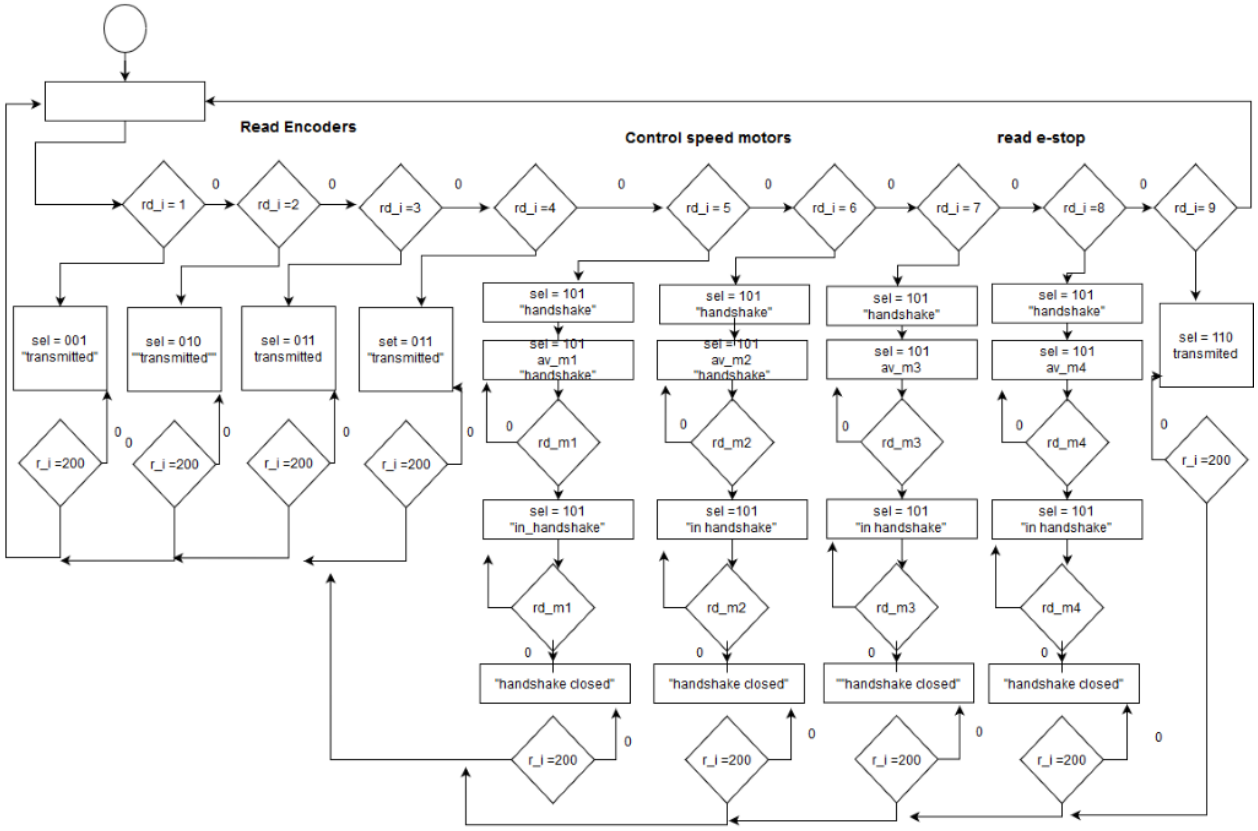


Figure 15: ASM of the mux_contorller

As illustrated in *figure15*, based on the instruction that mux_controller is reading from the processing system, the right execution will be implemented. In the process of the execution the mux_controller keeps the processing updated on the current progress by inserting messages in the write_msg register

- ***Transmitted***: To indicate that the encoder data or the e-stop was written into the write_data register

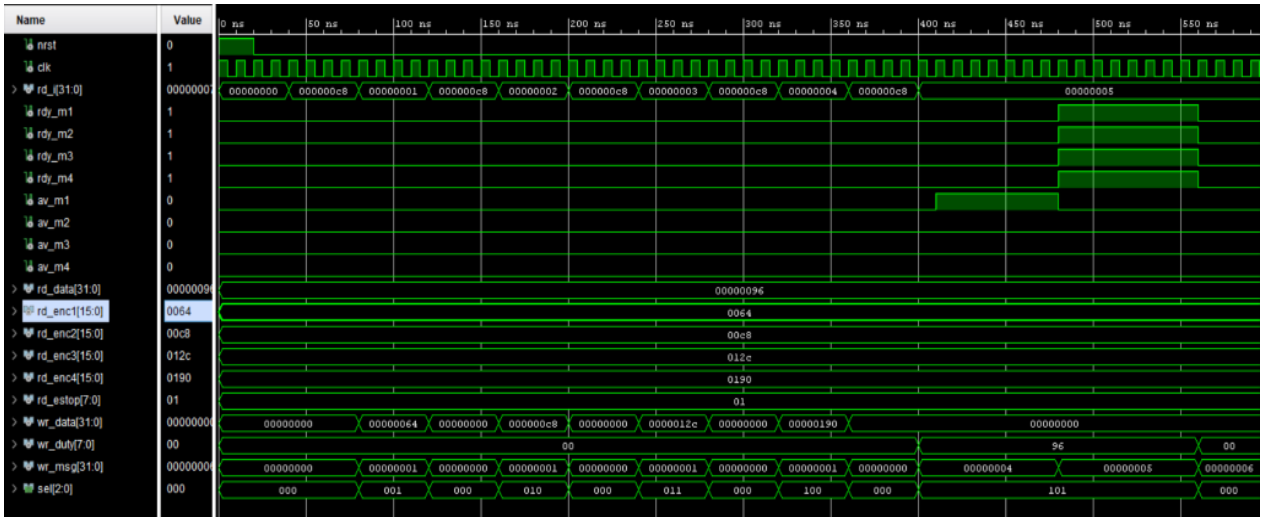
- ***Handshake***: To indicate that the multiplexer is waiting the motor controller to react to the new speed
- **In handshake:** To indicate that motor controller is the handshake process

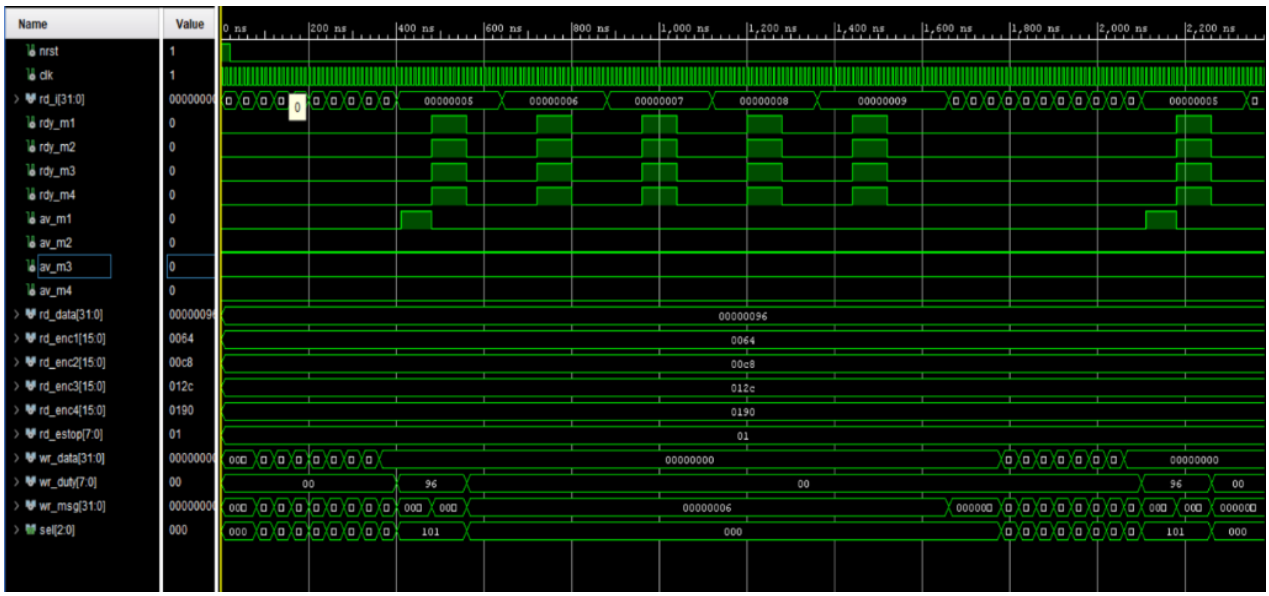
- Handshake closed: To indicate that the new speed was passed to the motor controller

The multiplexer is synchronized to clock frequency of 100 MHz , the simulation of the multiplexer is given in the appendix A

Appendix

Appendix A: Simulation of the multiplexer





Appendix B : Simulation of the Total System

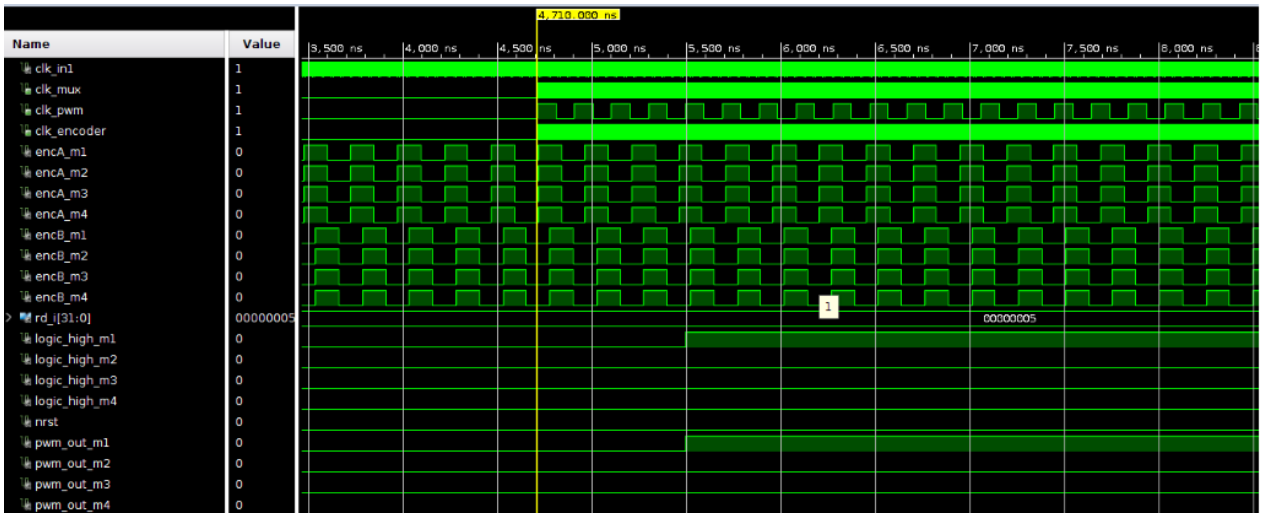


Figure 16: Simulation of the total system when received instruction 5

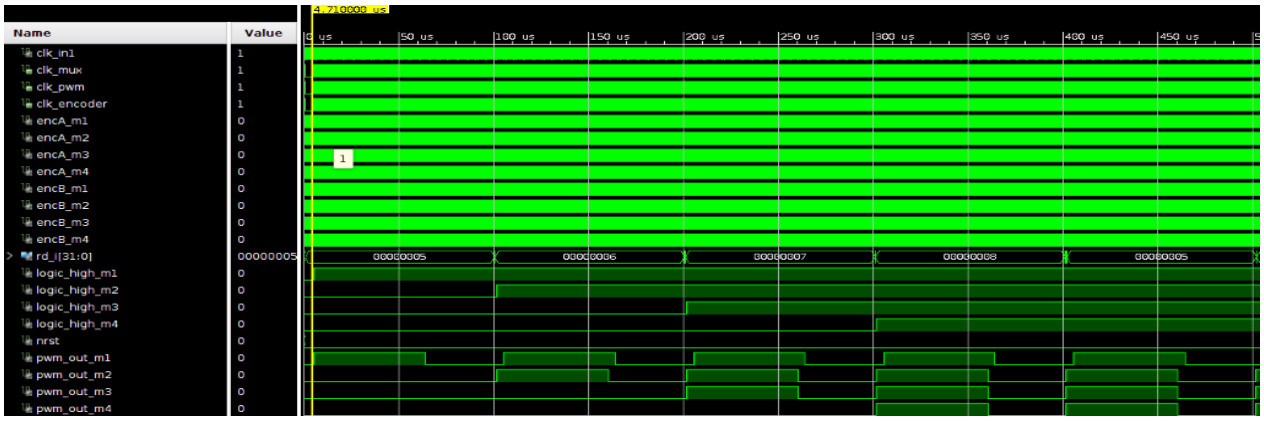


Figure 17: The simulation of the total system when received instruction 6&7&8

When the processing system sends instruction 5, the PWM of motor 1 is generated and the logic pin is set high

A. When the processing system sends instruction 6, the PWM signal of motor 2 is generated and the logic pin is set high
B. When the processing system sends instruction 7, the PWM signal of motor 3 is generated and the logic pin is set high
C. When the processing system sends instruction 6, the PWM signal of motor 4 is generated and the logic pin is set high