Bachelor's thesis

Information and Communications Technology

2021

Maria Kiigemägi

FPGA – BASED BLDC MOTOR CONTROLLER

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2021 | 37, 10

Maria Kiigemägi

FPGA – BASED BLDC MOTOR CONTROLLER

Field-Programmable Gate Arrays are a common choice for integrated circuit design that requires fast signal handling and frequent modification during the development process. An FPGA was used in this project for the exact reason of it being scalable and continuously improvable. The creation of this type of a device requires clear and concise workflow and -planning.

The objective of this thesis was to create an FPGA-based brushless direct-current(BLDC) motor controller. The work for this thesis was implemented in two parts: hardware- and code creation. Both parts were handled with the same workflow in parallel. The workflow happened in five steps: research, requirements definition , implementation, testing, and analysis.

The hardware description language used for the FPGA was VHSIC hardware description language(VHDL). The architecture was built by the author of this thesis based on research of previous implementations and the requirements defined for the controller. The hardware was based on the capabilities of the FPGA development board and the requirements of the BLDC motor used for testing. With this method, a basic BLDC motor controller was created and tested.

As a result, this thesis serves as an easy-to-understand document introducing the technologies required for a BLDC motor controller and describing the process of developing said controller. The final VHDL code was added, which can be used to implement an FPGA motor controller, and schematics which can be used to assemble the hardware for this controller.

KEYWORDS:

FPGA, project, electronics, BLDC, VHDL

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
AXI	Advanced Extensible Interface
BLDC	Brushless DC Electric Motor
CLB	Configurable Logic Block
DC	Direct Current
EMF	Electro Magnetic Field
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
IP	Intellectual Property
MAC	Medium Access Control
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
PL	Programmable Logic
PS	Processing System
PWM	Pulse Width Modulation
RAM	Random Access Memory
SoC	System on Chip
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 INTRODUCTION

In modern environment, comfort is often assessed as how many things and activities around a person are automated by machines. More often than not, this kind of automation requires mechanical energy. That means, things need to move autonomously. This mechanical force is indisputably provided to us by motors.

Motors are electromechanical devices that convert electrical energy into mechanical energy. Electricity is given to the motor at input and the motor converts this into torque, which is a rotational force. To archieve this, a motor requires some input logic. This means, the input has to be in some significant sequence for the rotational force to be produced. This logic can be provided to the motor through a controller, which is what this thesis is aiming to implement. In other words, the objective of this thesis is to create an FPGA based BLDC motor controller.

The technologies used in this thesis will be further explained in Chapter 2, going into more detail why these methods/components were chosen and how they differ from other options available. After this, in Chapter 3, the requirements for this motor controller will be introduced and ranked by importance. The implementation is described in Chapter 4, including how the device was tested, and the results of this will be further analyzed in Chapter 5. By the end of the conclusions in Chapter 6, the reader should have a comprehensive understanding of how a brushless direct-current motor controller works, how they can be designed and tested, and how to evaluate readiness for commercial use.

2 TECHNOLOGIES

In this chapter the technologies used for the work done in this thesis are introduced briefly and their working principles explained. Alternative technologies to these are also brought out for comparison, giving insight into why the technologies introduced here were chosen for this application. Going through each technology one-by-one, a clear and a visual picture of the resulting device of this thesis should start to formulate, giving better understanding of the work and resources that will go into making this BLDC controller.

2.1. Brushless Direct Current Motor

A direct current motor is an electric motor that converts direct electrical energy to mechanical energy, typically using forces created by magnetic fields for this. As the name states, DC motors use Direct Current. The two main DC motors widely used in modern days are brushed- and brushless DC motors. In recent times, brushless DC motors are becoming more and more popular due to their many advantages over a brushed DC motor. Some of these advantages are, for example, less maintenance, higher speed range, higher efficiency, reliability and accuracy. Because of the many upsides, a brushless DC motor is often more suitable and also preferred for a wide variety of applications [1].

The working prinple of a BLDC motor is similar to a brushed direct current motor. It bases on the magnetic forces of its two components: The rotor and the stator. Electricity is given to the motor at input to its coils on the stator, which carry current and create a magnetic field. The permanent magnet rotor reacts to the coils fixed on the stator being energized. As a consequence of this, the rotor will experience a force equal and opposite of the energized coil, producing torque. **Error! Reference source not found.** [2] shows the structure of a BLDC motor.

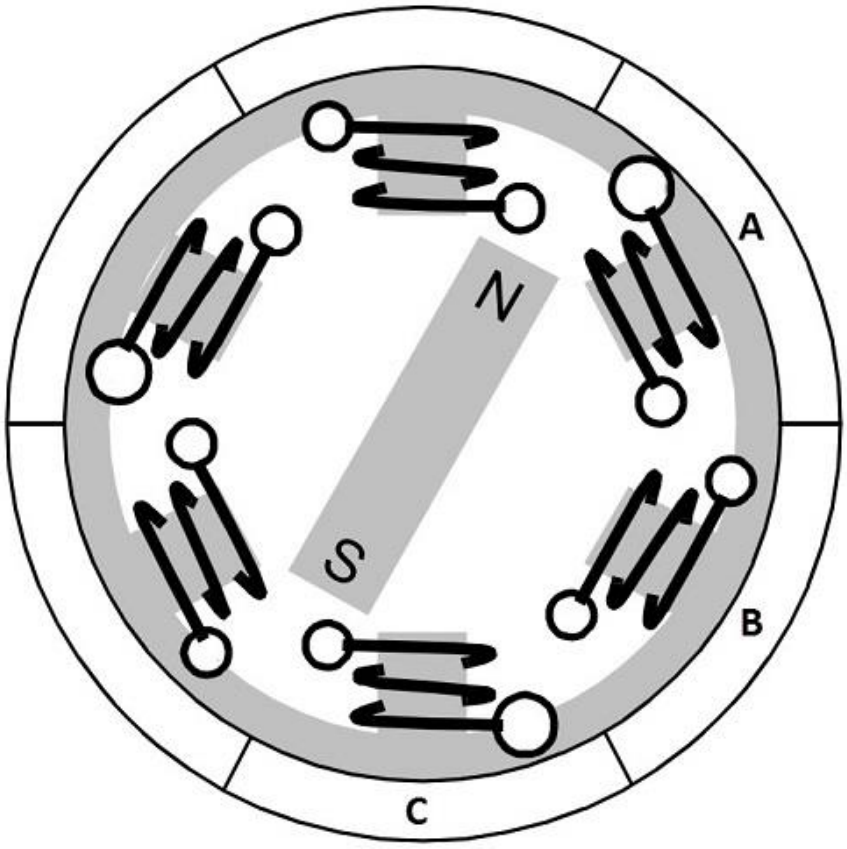

Figure 1. BLDC Motor Structure.

Conventionally, a brushless DC motor is controlled by a digital pulse-width modulation (PWM) controller. In order to rotate the motor in a stable way, the stator windings are energized in an appropriate sequence called commutational logic. This sequence and how it will be determined will be explained in more detail in Chapter 4.2. The speed of the motor can be adjusted by changing the timing of the voltages applied to the different coils of the motor. In order to determine the present position of the rotor, a hall effect sensor can be used to further improve the necessary sequence of the electromagnetic field (EMF) on the stator coils [3].

2.1.1. Hall sensors as BLDC feedback

Hall effect sensors can be used to detect the proximity, speed and displacement of a mechanical system involving magnetic fields, like a BLDC motor. Because these sensors measure the Hall effect (appearance of an electric field in solid material when it carries electric current and is placed in a magnetic field perpendicular to the current), they don’t

require contact to determine these indications, meaning they are a simple way to measure the location of moving elements. In the context of BLDC motor control, the hall effect sensors send out signals when they detect the rotor's magnetic field, indicating the current position of the rotor [4].

2.2. Field-Programmable Gate Array

Field-programmable gate arrays (FPGAs) are integrated circuits consisting of a large number of elementary building blocks. The basic cells of an FPGA are configurable logic blocks (CLBs), which are used to implement functions and macros by combining them together based on the netlist after synthesis. The CLBs contain combinatorial blocks and flip-flops, that process data according to how it is programmed. Figure 2 shows the structure of an FPGA. A typical FPGA also has Random Access Memory (RAM) built into it, and has the capability for clock conditioning. The Input and output blocks (I/O cells) of the FPGA connect the circuit with outside signals or also pass on processed signals, depending on the functionality given to them. The switch matrix is a network of connections between blocks, and contains transistors that can turn different lines on or off. For full functionality of the logic blocks to be available for the programmer, these interconnections and auxiliary circuitry take up to 80% of the silicon in a typical FPGA.

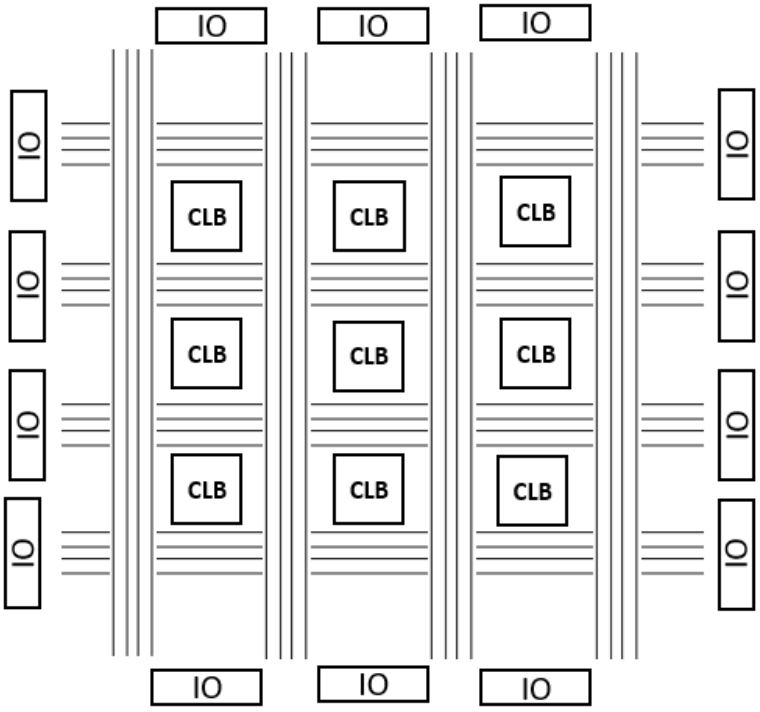

Figure 2. Basic architecture of an FPGA.

The amount of all the connections between logic blocks is, however, what gives an FPGA its' best know quality – it is reprogrammable, meaning that the integrated circuit can be changed after it has been manufactured. FPGAs can include a hardwired Intellectual Property (IP) cores for more genereically used functions. These blocks are not reprogrammable and include high speed serial interfaces, general-purpose processors and Ethernet Medium Access Control (MAC) units [5].

The advantages of an FPGA over other processors are parallelism, flexibility and speed [5] . Programming a simpler microcontroller often takes place using a high-level programming language(such as C++ or Python), in which case the functions that can be programmed into it depend on how the hardware is controlled with its low level code. The program is executed in a sequence, so signals cannot be processed in parallel. In addition, with a classic processor, the hardware-level functionality is preprogrammed into it and cannot be changed, which means functions are predefined for the programmer on multiple levels.These kinds of wrappers are often wasteful in terms of processing time and memory usage [6].

The most common alternative to FPGA:s are ASIC:s, or Application Specific Integrated Circuits. These are devices that have been created for a specific purpose or functionality.

This functionality cannot be changed during the lifetime of the chip, which means that its digital circuitry contains permanently connected gates and flip flops in silicon. ASICs are more suitable for bulk production since they are more energy efficient and less power consuming than FPGA:s. They can also archieve higher frequencies and thereby be more efficient in some applications. This, however does not outweigh the suitability of FPGAs for prototypes and small production scale applications, since the workflow of an FPGA contains four steps: design entry, synthesis, implementation and programming. In comparison, designing an ASIC can have up to 14 steps [7].

FPGAs are programmed with hardware languages such as Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog. These are hardware descriptive languages, meaning they describe the operation of a digital circuit at 'iron level', and the logic operations take place in parallel, rather than in a sequence [5]. Programming is based on creating logical ports and gates. This is further helped by programming platforms like in the context of this thesis, Xilinx Vivado. Vivado synthesizes the programmers code, veryfing and simulating it to ensure functionality.

This combination of advantages and opportunities makes an FPGA a perfect base building block for a BLDC motor controller.

2.3. Development Board.

TUL PYNQ-Z2 is an FPGA development board designed and developed by Xilinx® for embedded systems. In addition to FPGA, the board enables the developer to take advantage of the advanced ARM processors on the board. The SoCs can be programmed in Python and/or take advantage of the imported hardware libraries [8].

The board (Figure 3 [8]) integrates HDMI Input/output, Arduino and Raspberry interfaces, 2 PMODs, user LEDs, puhs buttons and switches as well as MIC input, Ethernet and audio output.

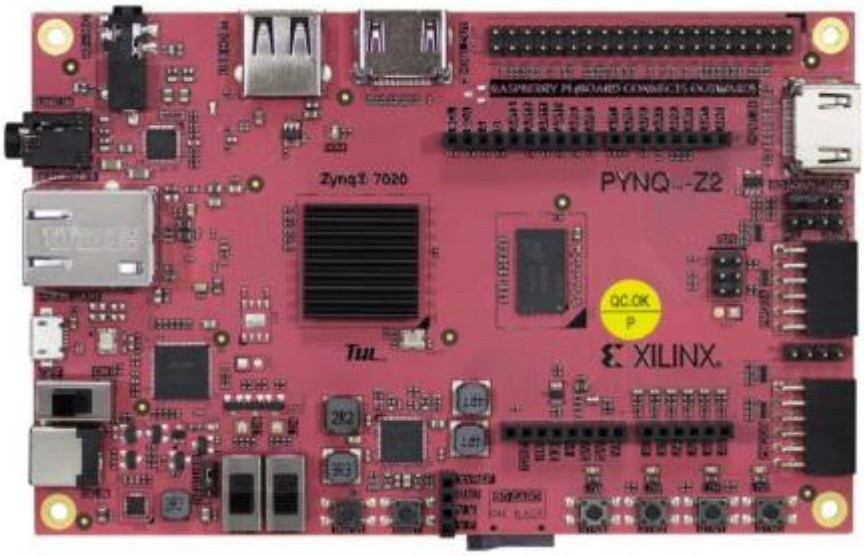

Figure 3. PYNQ-Z2 development board.

The PYNQ-Z2 board is composed of two major functional blocks: the Processing System(PS) and Programmable Logic(PL). The PS-PL interface has all the signals available for integrating the PL-based functions with the PS [8].

PYNQ-Z2 has a 50MHz oscillator at the Zynq processing system PS_CLK input, which is used for clock generation to all subsystems (**Error! Reference source not found.** [8]). This clock input allows the processor to operate at up to 650MHz via an internal clock multiplier. The board also has an external reference clock for Programmable Logic (PL), which is 125MHz [8].

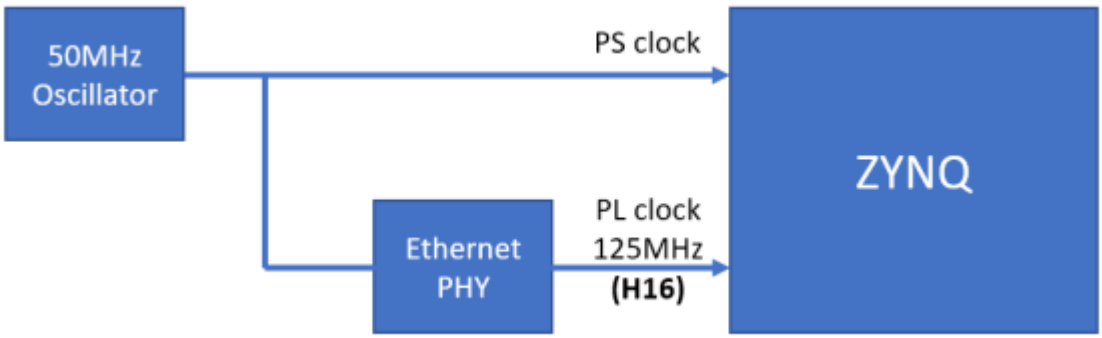

Figure 4, The PS-PL clock system diagram of PYNQ-Z2

The Processor Subsystem(PS) of the board incorporates AXI (Advanced eXtensible Interface) memory port interface and is carried with the Dual/Single Arm Cortex-A9 MPCore CPUs with Arm v7. [8].

The programmable logic (PL) of the board is equivalent to Artix-7 FPGA, which consists of 1.3M reconfigurable gates. It has 13300 logic slices, each containing four six-input look-up tables (LUTs) and 8 fip-flops. The FPGA also has 630 kB of RAM, 4 clock management tiles 220 DSP(digital signal processing) slices and on-chip analog-to-digital converter. It is programmable from JTAG, Quad-SPI flash and microSD card [8].

The PL has high range I/Os that support 1.2V to 3.3V, accessible from the 16 PMOD ports on the PYNQ-Z2. These are 2x6, 100 -mil spaced female connectors that mate with standard pin headers. Each PMOD port can provide up to 8 logic signals (up to 1A current). For configuration options the PL supports up to 5.0 Gb/s speeds and has advanced configuration options, advanced error reporting, and end-to-end CRC advanced error reporting and -features [8].

2.4. MOSFETs

MOSFETs are active electrical components that behave similarly to transistors. MOSFET is short for metal-oxide-semiconductor field-effect transistor and it is used for higher current applications, as it requires very little control current. The voltage applied to a MOSFETs gate determines its conductivity between source and drain. MOSFETs are often used for switching or amplifying signals. Because of their properties MOSFETs are nowadays more common than bipolar junction transistors. Figure 5 [9] shows the basic structure of a MOSFET.

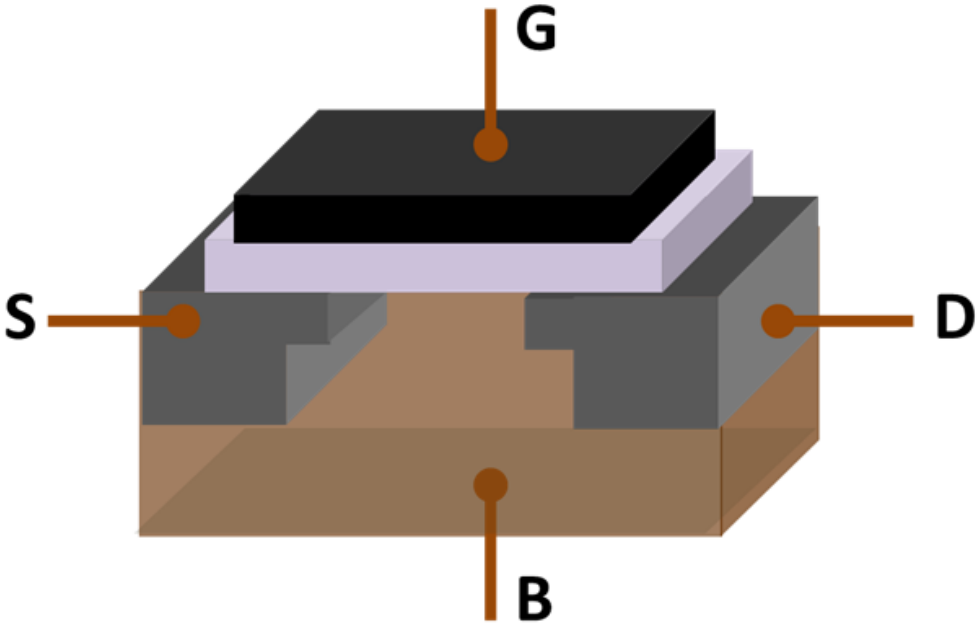

Figure 5. MOSFET structure.

A MOSFET has three pins – gate, drain and source. The gate is made of silicon dioxide and forms a ‘gate’ isolating the source and drain current flow. When voltage is applied to the gate (positive or negative depending on if it is an n-mos or p-mos), charge carriers will accumulate at the gate contact forming a conductive bridge between the source and drain. This principle enables one to use the smaller voltage at gate to rapidly switch off and on more powerful circuits [10].

3 REQUIREMENTS

This chapter describes what are the requirements for the resulting device. To do this, MoSCoW method is applied, which is commonly used in agile software development, when prioritization of the requirements is needed because of the approaching deadline [11]. The features have been listed in Table 1 with the priority and class of the requirement.

The chapter will start with the “must-have” features which will be the minimum of what the device needs to be considered a motor controller, after which the “should have” features describe what is generally included with a BLDC motor controller. The “could haves” are also briefly introduced as ideas for further development.

Since BLDC motor controllers are available to buy ready-made from online stores, the bare minimum requirements were chosen based on the features of the cheapest BLDC motor controllers sold on RobotShop.com [12], while extra features (could have’s) have been added according to what would be a nice-to-have extra feature for the controller according to the author of this thesis, inspired by some of the more expensive products available.

Table 1. feature set implementation prioritization using MoSCoW.

REQ #	Name	Priority	Class
1	FPGA based	MUST	Hardware
2	Three – phase motor control logic	MUST	Programmable logic
3	VHDL	MUST	Programmable logic
4	Hall sensors	SHOULD	Hardware
5	Speed control	SHOULD	Programmable logic
6	User control buttons	SHOULD	System
7	Direction control	COULD	Programmable logic
8	User control over Linux	COULD	System

3.1. Hardware requirements

This chapter will explain what the necessary hardware components for this controller are. It will also establish what the logic behind choosing these requirements is and the role that each component will have.

3.1.1. FPGA-based

The design must be based on FPGA to enable flexible reconfiguration. Because of availability and the authors prior experience with hardware description languages, PYNQ-Z2 was chosen as a development board for this project.

Because only the current and voltage output of an FPGA isn't enough to feed the motor, MOSFETs will be used to amplify the output FPGA signals and apply the necessary power for the motor.

The limiting factor for the MOSFETs was the gate. In a typical Enhancement-mode MOSFET the device is in 'OFF' state by default, meaning that no current passes from source to drain while the gate bias voltage is equal to zero. When a gate voltage is applied to the gate terminal that is greater than threshold voltage level, drain current will flow. This means that the threshold voltage has to be below or at 3.3 V for the FPGA to be able to drive it directly without extra voltage amplification [13].

The DC motor required for this would have to be a three-phase brushless DC motor, with voltage and current ratings according to the capabilities of the mosFETs chosen. Therefore, the mosfet maximum drive current determines the power level of the motor. Also, since the PYNQ-Z2 board contains a programmable logic clock of 125 MHz frequency. The fall- or rise time of the MOSFET will be the main limiting factor of the pwm frequency.

3.1.2. Hall sensors

It is possible to make a very basic BLDC controller using a simple commutation sequence ignoring the location of the rotor. However, if any extra features other than simply driving the motor at a certain speed are required, a hall effect sensor is needed.

When selecting a hall effect sensor for this type of application, the following characteristics are considered: Sensitivity, repeatability, stability-over-temperature and response time. Higher sensitivity (<60 Gauss) is to be expected for reliable results when manually installed on the motor [14].

3.2. Programmable logic requirements

The requirements for the program that the FPGA will be carrying are introduced in this chapter. The reasons behind choosing these features will also be explained.

3.2.1. VHDL

The programmable logic for this project must be written in VHDL (VHSIC Hardware Description Language). VHDL is used to program the behaviour of the programmable logic integrated into the PYNQ-Z2 board. It was chosen over Verilog, an alternative language with similar functionality, based on prior knowledge of the author of this work.

The code must contain all the necessary inputs and outputs and their behaviour to produce a BLDC controller as a result of this work. The inputs for the VHDL code will be the System Clock signal, three hall effect sensor inputs (digital logic), and four user inputs.

The user inputs will control the motor switching on/off, adjusting the speed up(higher speed), adjusting the speed down (lower speed) and system reset for the controller. The motor on/off signal will enable/disable the transistor switches, which powers the higher current circuit and motor. Speed controls will control the PWM duty cycle (Increasing the duty cycle will increase the voltage applied to the motor and decreasing will do the opposite). A system reset will reset all the VHDL components to a known logic state.

The programmable logic for this project will be implemented in three phases:

1. Implementing the basic commutation logic
2. Implementing system reset and power on, off
3. Implementing speed control with PWM

This enables the author of this work to test these features individually and makes the final work modular enough for further improvements.

3.2.2. Three-phase motor control logic

In order to ‘activate’ a BLDC motors coils in the right sequence, commutation logic is applied to it. When the correct hall sensors are displaced at 120° apart from each other, they can be used to determine the motor phases and the next commutation step for these. This system can be seen in Table 2.

Table 2. Three-phase BLDC commutation logic.

Hall sensors			**BLDC phases**		
Ha	**Hb**	**Hc**	**Phase_A**	**Phase_B**	**Phase_C**
1	1	0	OFF	Vpos	Vneg
0	1	0	Vneg	Vpos	OFF
0	1	1	Vneg	OFF	Vpos
0	0	1	OFF	Vneg	Vpos
1	0	1	Vpos	Vneg	OFF
1	0	0	Vpos	OFF	Vneg

If this system were to include a ‘reverse direction’ feature, as seen in Table 1, the commutation logic would use a different sequence, because the order of the digital logic of the Hall sensors and switching MOSFETs would be different [15].

3.3. System requirements

This chapter will explain how the whole system will work for the end-user. This means that it will show how the signals move from user input to motor output in order for the motor to work according to the user's choice.

3.3.1. User control buttons

The buttons attached to the PYNQ-Z2 board will be used as the primary user control interface for this project. The buttons and their functions are described in the system diagram are displayed in Figure 6

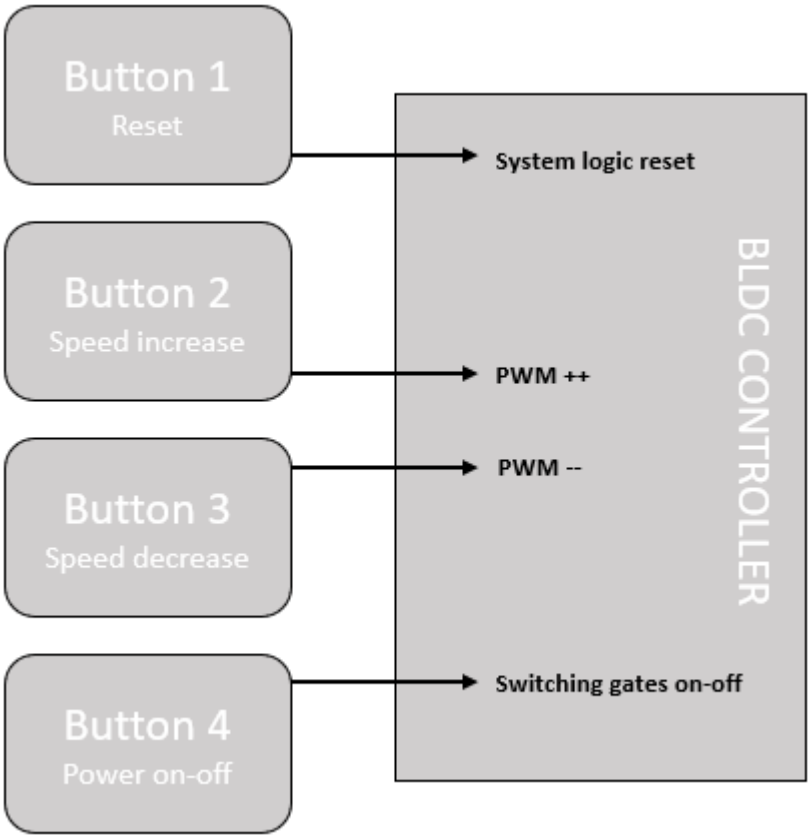

Figure 6. User buttons system diagram.

3.3.2. User control over Linux

The project could also implement user controlling the motor controller over Linux, which is supported by the ARM processor on the PYNQ-Z2 board. This will not be the primary user control interface, however and the physical buttons will have priority over these input values. As this is a “could have” feature it will not fit within the scope of this thesis and will be reintroduced briefly in Chapter 5.4. when discussing future improvements.

4 IMPLEMENTATION

This chapter goes into detail about how the requirements discussed in the previous chapter are going to be implemented. It starts with hardware components and the build structure, after which it will explain how the system operates logically including VHDL code and algorithm, and lastly the implementation is described as a whole.

4.1. Hardware

In order to develop the hardware portion of this project, the system, shown in Figure 7, was planned out as a whole and divided into four subsystems:

- Pynq-z2 wiring and connections
- Hall sensor wiring
- Switching MOSFETs
- Motor

The only motor available for the author of this thesis was an unknown motor deattached from an electric scooter (with no further information which scooter it had been). The motor had hall effect sensors attached to it, but because there was no further information to be received about this motor, it had to be tested separately, therefore the motor itself is categorized as a separate subsystem in this thesis.

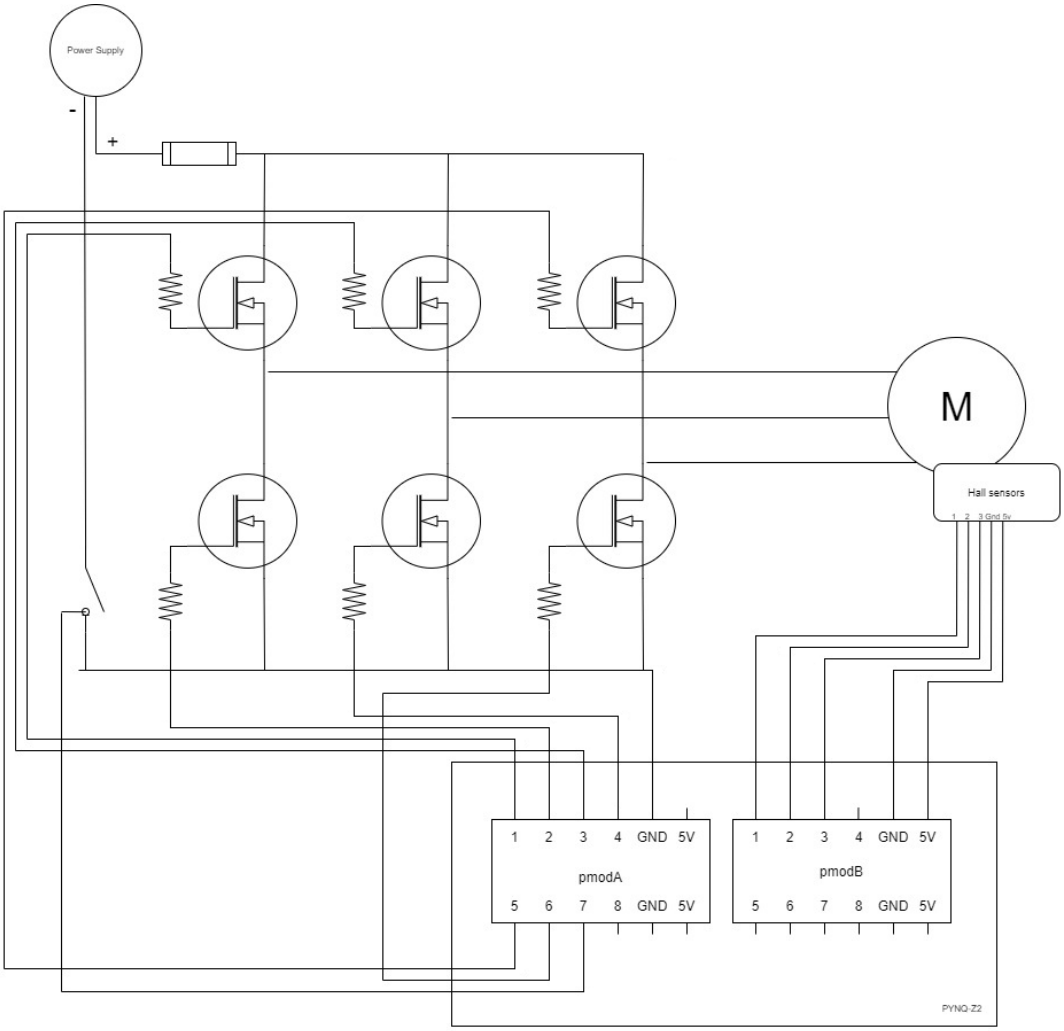

Figure 7. Generic system diagram.

4.1.1. Motor

As stated in Chapter 4.1. , because of availability, the motor used for testing the code and circuitry for this work was an unknown. This meant that it had to be tested and if possible, researched before it could be used to test this application.

The motor had been extracted from an electric scooter and the serial number written on the exterior was HHJ25NT36V190930533. Unfortunately this provided no datasheet when researched, but included the number 36V, which, at the very least, gave some idea as to what would be the maximum voltage to drive this motor with. 4.1.1.

4.1.2. Pynq-Z2 wiring and connections

The PYNQ Zedboard includes two attached PMOD interfaces. These are 2x6 connectors that provide 3.3V signals, two ground ports and 8 logic signals, as shown in Figure 8 [8]. When a pin is assigned as ‘input’, it will be grounded and a signal of 3.3 Volts will be read as ‘high’. When a pin is assigned as input, it will be 0 or 3.3 volts according to the output the program is sending out at the time.

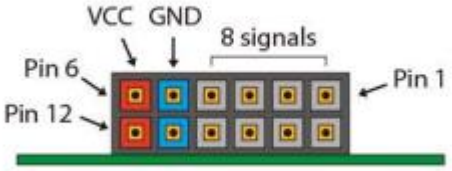

Figure 8. PMOD Port.

4.1.3. Hall sensor wiring

The hall sensors attached to the motor could be tested without turning the whole system on. The motor wires included a 5-header pin socket for hall sensors. The red wire was assumed to be Vcc (positive DC), typically 3.3 – 5 V for hall sensors, black wire as gnd and the three colorful wires as hall sensor outputs, color-coded the same as the three similarly colored motor phase wires. These assumptions are confirmed by the markings and positions of the sensors on the actual motor, shown in Figure 9.

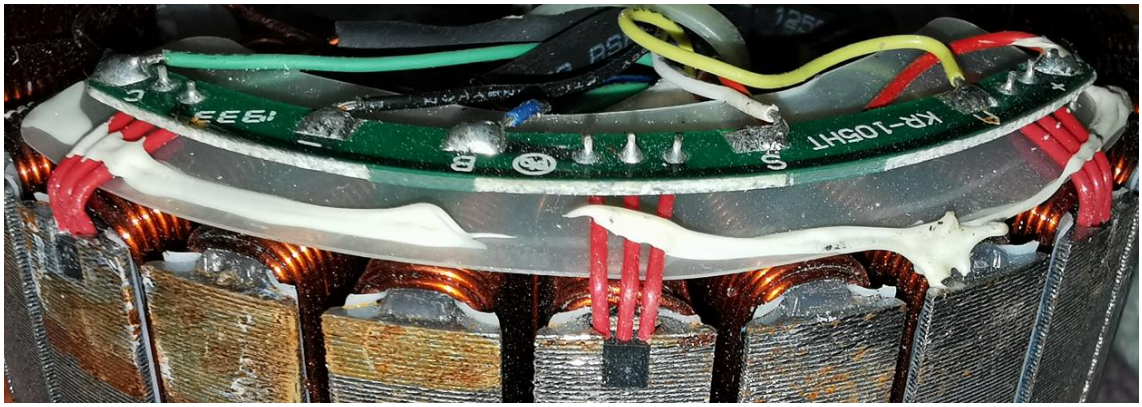

Figure 9. BLDC motor hall sensors.

These sensors were tested on breadboard by attaching the outputs to LED's. When the motor was rotated, the hall sensors sent signals to LED's, lighting them up when hall sensor was active.

What was discovered during the initial breadboard test, however, was that the hall sensors attached to this motor were 'active low', meaning the active output would pull the voltage to GND. This can be solved on the PYNQ-Z2 by pulling the PMODoutput ports to high, and using the sensor output to pull them low. This had to be adjusted in the code as well (swapping the signals 'low' to 'high' and vice versa).

This connection type was also tested with the PMOD port by setting up the circuitry (Figure 10) and attaching the PMODinputs to the existing LEDs on PYNQ-Z2. Custom VHDL code was written for this test.

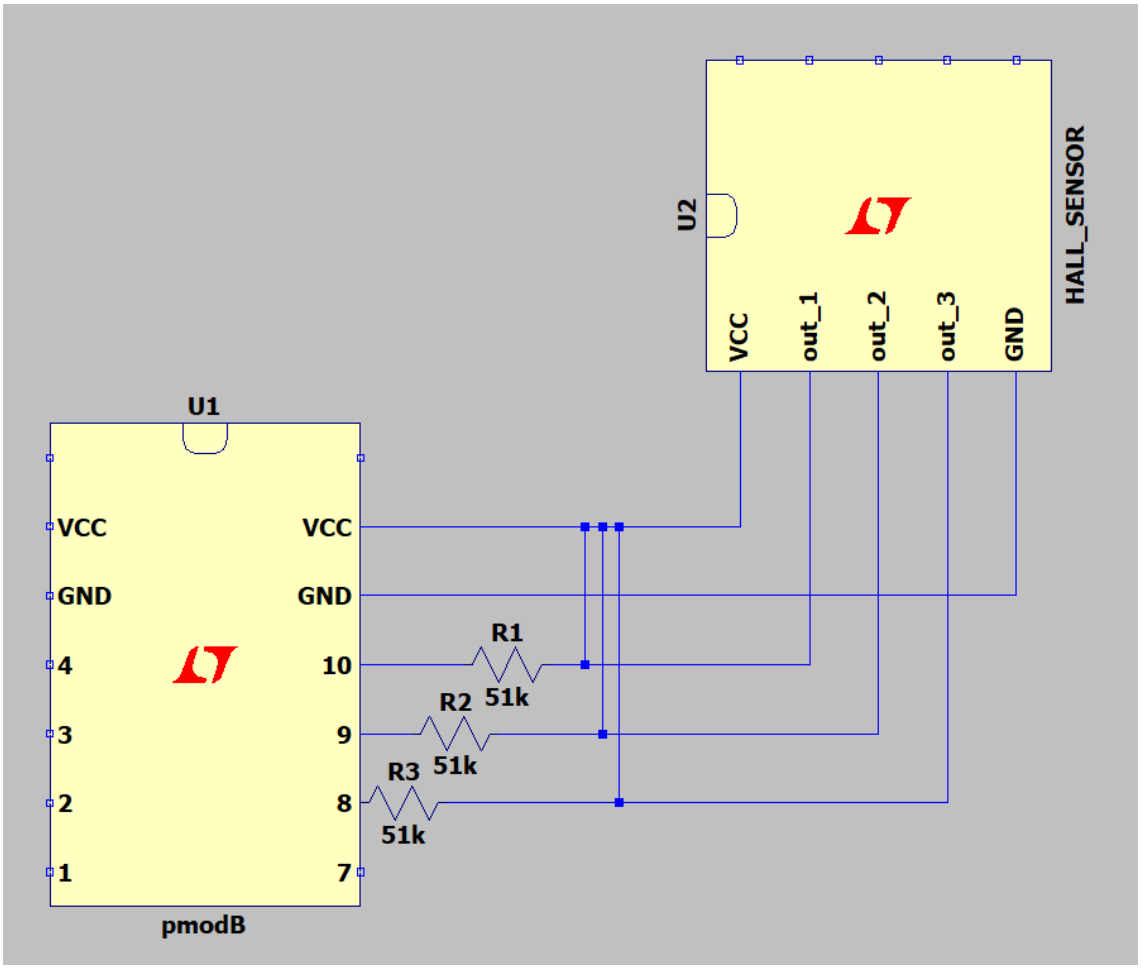

Figure 10. Hall sensors circuit diagram.

4.1.4. Switching MOSFETs

The most critical subsystem of this project were the switching MOSFETs, which are essential to driving the three phases of the BLDC motor. Because of this, the subsystem for the MOSFETs was tested several times with various methods to assure that all the connections work correctly. The circuit diagram for the switching MOSFETs is pictured in **Error! Reference source not found.**.

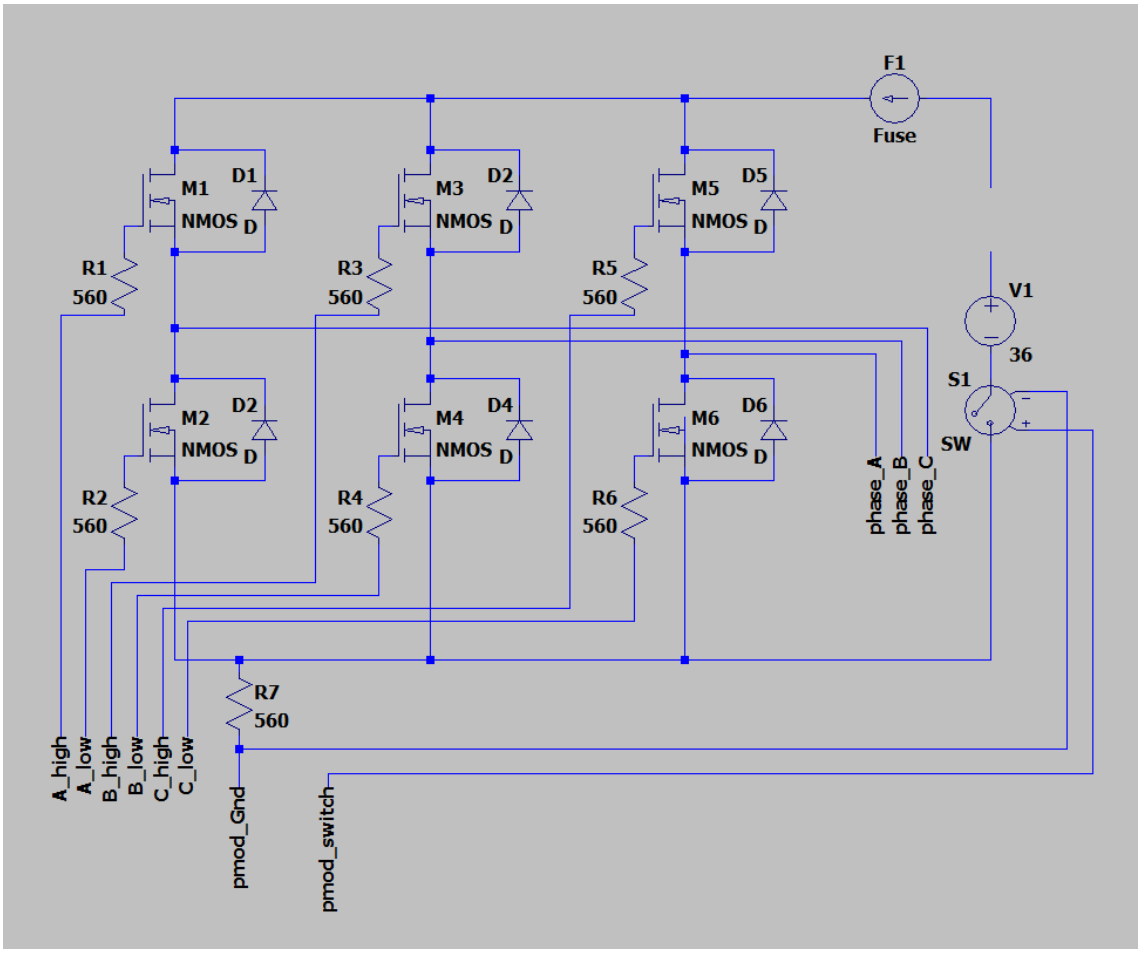

Figure 11. Switching MOSFETs circuit diagram

The switching MOSFET circuit includes a 10 A fuse right at the positive power source of the circuit, a switch connected to the pynq-z2 switch signal, the PMOD inputs at the gates of MOSFETS, which are passed through 560Ω resistors and the motor phase outputs. The MOSFET used for this application is International Rectifier's IRL34NPbF n-mos. According to the datasheet [15]. These have a logic-level gate drive (gate-source

threshold at 1-2 V), And a drain-source voltage limit of 60V, with continuos drain current of 30 A, which are all suitable for this implementation.

This circuit was soldered on a model plate and tested first by attaching LED's to where outputs were. The input was given from a separate 3.3V voltage source by manually connecting and disconnecting the wires.

Once the initial test was done and the connections worked with the LED test, the MOSFET gate inputs were connected to the PYNQ-Z2 PMODA connectors. This was also done in the correct sequence so that the planned outputs matched the correct MOSFETs.

Only when the inputs passed the first two tests, was the motor attached to the MOSFET switching circuit. Initially, a testing code was used, which only applied the commutation sequence in a known frequency to the motor, with no hall sensors used. When this test passed, the circuit was considered ready for testing with the whole system connected.

4.2. Embedded software

As previously stated in Section 3.2.1. , the code for this project was written modularly. The hardware description was written as an architecture of six components, as **Error! Reference source not found.** below shows. The general workflow of creating a VHDL 'component' was planning out the behaviour of the component, planning the I/Os of said component, writing the code in Vivado, writing a testbench for the component in Vivado and simulating the behaviour with the testbench. The last three steps were repeated until the behaviour worked as was intended. This sort of testing with different patterns is called black box testing, which is one of the more common techniques for testing for VHDL [17].

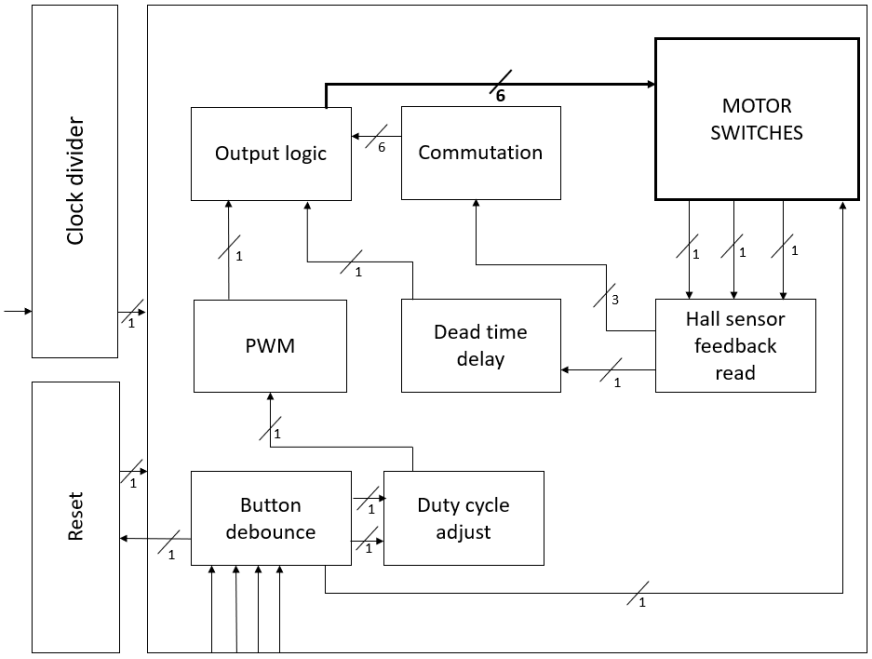

Figure 12. VHDL code structure.

Each component was created and tested separately, with its own inuputs and outputs. Then, a testbench was written. A testbench in Vivado means that the component is treated as a separate entity, which enables the coder to generate input-output signals in the testbench. These simulated signals are hard-coded into behaving as expected by the coder, and a timeline of simulation is created, enabling the coder to monitor the behaviour of the top-level structure (the component itself).

In the subchapters of this chapter, the implementation, behaviour and testing of each of these components is explained separately. An overview and the behaviour of the whole architecture along with the hardware is handled in Chapter 4.3.

4.2.1. Clock Divider

The whole system is clocked with the system clock of the FPGA on the PYNQ-Z2 board, which is a 125 MHz oscillator. This makes system clock time period to be 8 nanoseconds. As this was by far too small time frame in this project, the system clock was divided in the clock divider component. This division can be adjusted with a generic variable, which in the current application is the desired output frequency of the divider.

The clock divider testbench was a simple behaviour model, where a 125 MHz clock signal was sent to the component. The behavour of the clock was observed and measured at different clock frequencies. In Figure 13Figure 1 , this behavour was tested when clock frequency is adjusted to 1kHz, three clock cycles equal to 3 ms.

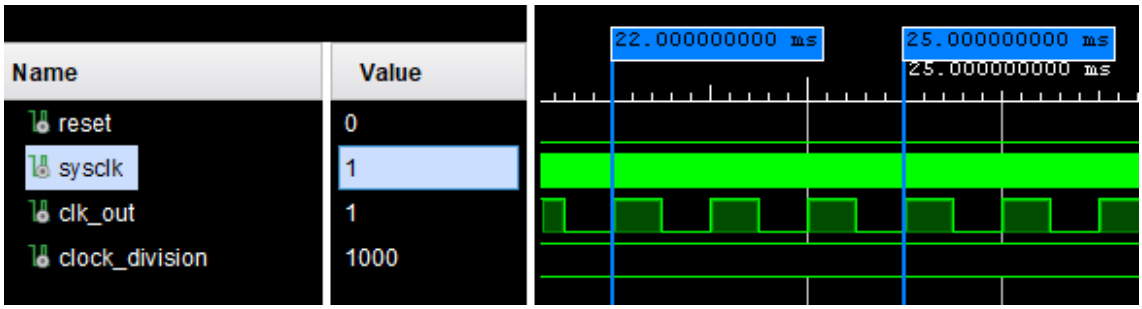

Figure 13. Clock divider simulation.

4.2.2. Button Debounce

Because the implementation required user buttons as a way to control the system behaviour, button debouncing was needed. A button debounce is basically a counter which startswhen the system gets the first signal. If the signal stays consistent for the duration of the counter, the signal is read and passed as a valid button press. This is implemented, because physical buttons are unreliable and can tremble (or “bounce” as seen from Figure 14. Button "Bounce" [15]) around the logic threshold during archieving the contact, causing the signal to be unreliable during that [15].

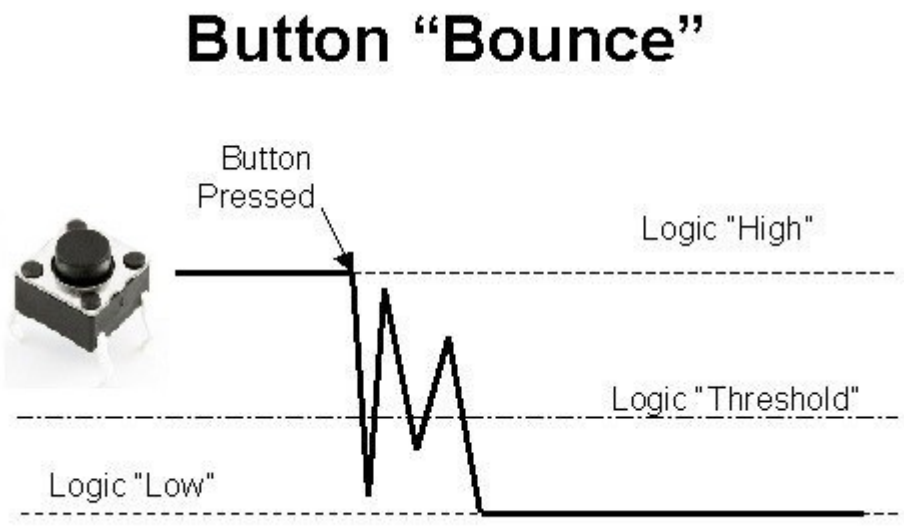

Figure 14. Button "Bounce".

Four buttons were implemented in the system – the reset button, speed up button, speed down button and a power on/off button. The reset button behaves as a reset for the whole system. Each component has a reset input which zeroes all the counters and outputs, bringing the logic to a known state.

The speed of the motor is controlled by a PWM components duty cycle. There is a set of 5 registers implemented in the system which contain pre-existing duty cycle values. The value of currently selected register is sent to the PWM module. Cycling through these values is done with the speed up and speed down button signals.

The power on/off button simply sends a signal to the switching mosfets to shut down and this behaviour is handled in the hardware. This means, the logic of the FPGA will continue running, but the switching mosfets will be turned off until the power on/off switch is pressed.

This behavour was tested in a testbench by sending simulated button pulses with active high time of longer and shorter than the debounce delay variable.

4.2.3. PWM

The speed control of the motor is done with pulse width modulation. This component has a generic variable called pwm divider, which behaves in the same manner as the clock

divider. To achieve, for example, 1 kHz PWM frequency, system clock is divided by 125000 (original system clock frequency is 125 MHz). After this, the signal is divided by the duty cycle value. The PWM component sets its output signal to a high value while the counter is counting until duty cycle value and low until the end of the counter (whole clock cycle, in this example, 1 ms). Using a different clock division enables the PWM component to run at customizeable frequencies for optimal pulse width modulation for the motor.

PWM was tested by creating a testbench, which received clock signals of different values (this was to test the PWM's clock divider) while also changing the duty cycle of the PWM during the process.

4.2.4. Hall sensor feedback

The hall sensor feedback gets the three-bit input directly from the motor hall effect sensors. This component reads these signals and saves them in a register, which it compares to the next received signals. When this comparison indicates a change in signals, a flag is raised for the dead time delay counter to start. The new value is saved in the register and the new signal is also passed to the commutation logic. This component was tested very similarly to the commutation component handled in Chapter 4.2.6.

4.2.5. Dead time delay

The dead time delay component is necessary for the FPGA to not switch the transistor logic immediately. As stated in Section 4.1. the timing of the commutation logic switch is dependent on the MOSFETs used for this application. IRLZ32PbF has a Rise time of 100 ns and a fall time of 29 ns [15]. In the application a 5 clock cycles dead time delay was used, as the timing and efficiency weren't this critical for the first prototype.

The dead time delay gets a signal when the hall effect sensor state has changed and starts counting. In this implementation the dead time delay counts to 5, giving the mosfets 5 microseconds to empty and close the gate. The delay timer sends out a pulse staying in high state from when it starts counting until it finishes counting. This system will let the output logic know, when to turn the commutation signals off and back on to the new state.

4.2.6. Commutation

One 3-bit logic vector runs into the commutation component. This is the current state of the hall sensors. The commutation logic applied according to the hall sensor feedback directly applies to TableTable 2. Three-phase BLDC commutation logic. The output of this component is a 6-bit bus marking the states of the 6 switch transistor gates. This behavor was checked in the testbench by sending it all the different possible hall sensor states and seeing if the sequence applied correctly to each one.

4.2.7. Output logic

The output logic portion of this architecture reads the commutation logic signal and the dead time delay signals and the PWM signal and joins them together in a comprehensive logic to send to the switching transistors as a 6-bit data bus.

When the hall sensor feedback reading changes, the commutation is changed immediately. The output logic, however, reads the dead time delay signal first and turns the outputs to zero until the dead time counter flag is raised. When the flag is lowered, the output logic switches the output to the new commutation state.

When the commutation logic is sent to the switching transistors, only the high side drivers get the PWM applied to them. This is called a trapezoidal control and is common for more simple BLDC motor controllers, as speed can be adjusted quite straightforward and no extra complications are introduced [16].

Because this portion of the code was not written as a separate component, the testing of output logic was more difficult. Writing a testbench for this was impossible without including all the signals and components of the overall architecture. Because it is very difficult to test the behaviour like this, a separate HDL code was created for this portion. This allowed to test the output logic with different components added each time the previous one proved to be working correctly.

4.3. Overview

All the VHDL components were joined together to an architecture, which, again was passed through a testbench to check the behaviour of all the subsystems together. This was monitored with signal timings as close to reality as possible (button signal timings, hall sensor timings). When this testbench gave results that would be applicable in the physical world (see Figure 15), the code was tested with the hardware.

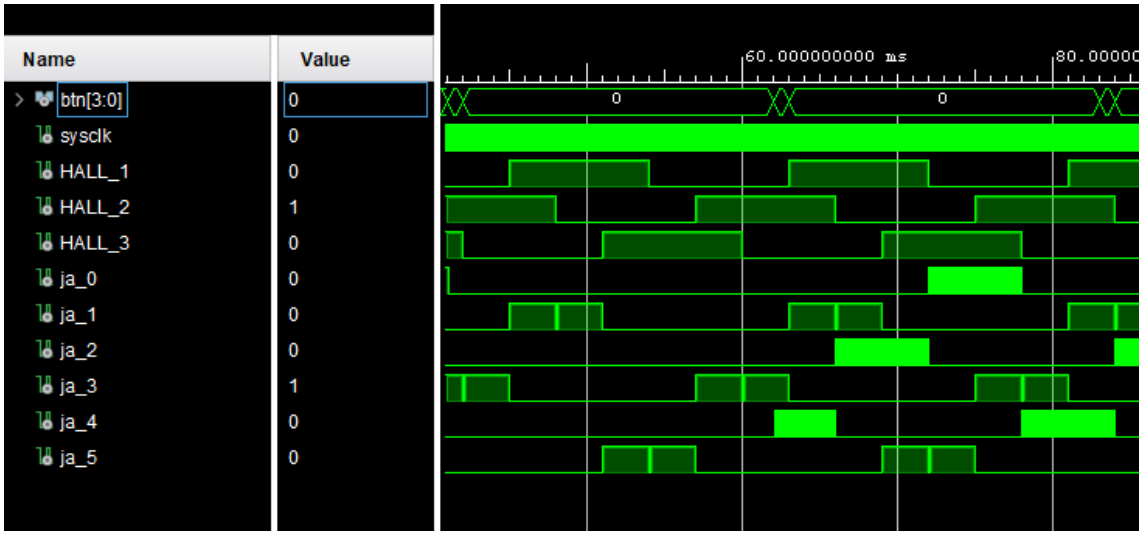

Figure 15. Architecture testbench result.

Along with all the hardware and code, the final test included measuring the voltages applied to the motor with an oscilloscope. All the different PWM levels were measured to see, how the speed change interaction works. The results and analysis of the final testing phase will be handled in Chapter 5.

5 RESULTS AND ANALYSIS

5.1. Implementation

With the final testing it was proven that the FPGA logic works as intended at different frequencies. The motor moved in the desired direction with the hall effect sensor placement affecting the output logic on the MOSFET gate. Figure 16 shows that theoutput logic meets all the requirements – no same phase low and high signal are active at the same time, and the dead time delay is behaving correctly.

Figure 16. Logic output of the motor driver.

The behaviour was also measured with different PWM levels. Figure 17 shows the desired behaviour with PWM duty cycle at 50%. As intended, only the high side of the phases are PWM'ed and the dead time delay remains correct.

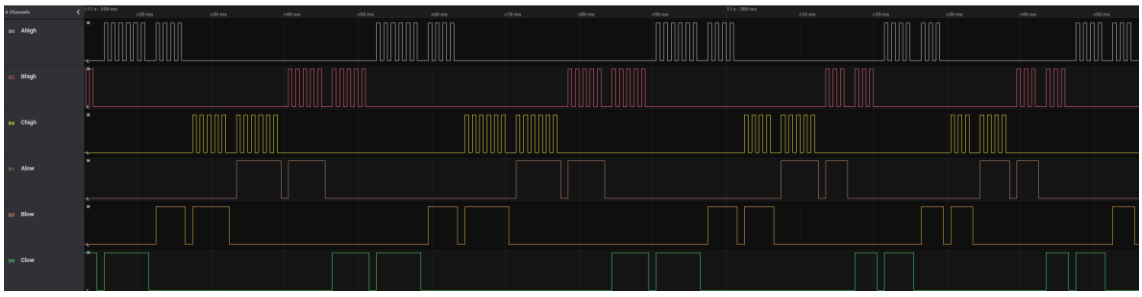

Figure 17. Output to MOSFET gates with PWM.

This proves that the implementation requirements were satisfied and the controller works as was originally intended. The behaviour of the motor also met the expectations.

5.2. Performance analysis

The result of this thesis was a simple FPGA driver for a BLDC motor, which was capable of the most basic functions of the motor, like switching it on and off, controlling its motion with the hall sensor feedback and changing the speed with a pwm.

It should be further mentioned, that the timings of the controller were very inefficient. For example, the dead time delay counter was set to 5 (clock cycles). Because the clock divider was set to 1MHz this equals to a dead time delay of 5 microseconds. As was earlier stated in Section 4.2.5. , the minimum delay could be 129 ns. This is one example of small improvements that could be done to the code once the system is in futher development.

5.3. Usability

At its current state, the motor controller is still in the process of development. As it is lacking many important features a simple BLDC controller would need. It makes the motor spin and enables the user to change the speed via button. In reality, however, the controller needs to be able to communicate with a microcontroller.

In addition, there are no specifications what types of motors the controller can handle. In BLDC motors, it is very important to specify, what timings does the motor require, i.e what the rpm is. The PWM frequency should depend on the motortype and/or be adjustable by the user.

At its current state it does perform the initial requirements set for it. In the perspective of end user, however, the result is very niche. There are very little real world applications that this controller system could be used for. With this said, with the controller being custom made, the motor controllers best use case is for the thesis author’s personal projects as of now.

5.4. Improvement discussion

The most urgent improvement points for the controller would be hardware testing and improving the compability of the components. More research and documentation would be needed to use the controller within a bigger-scoped project.

Thanks to the modularity of the VHDL code, improvements can be made to the system and features added to the controller. Some examples include acceleration and deceleration algorithms, SPI interface implementation, current control system etc.

As mentioned in Chapter 3.3.2. , the whole system could also be controlled over another microcontrol using, for example, Linux. This would improve the system with more reliable signals than physical button ones.

6 CONCLUSION

The aim of this thesis was to create an FPGA-based BLDC motor controller. This controller was designed to be able to switch the motor on and off, and change its speed. The whole system was to rely on the feedback of hall effect sensors attached to the motor. The initial objectives of the thesis were met to satisfaction. The FPGA logic works as intended with the signals behaving as expected, and the motor moves, with its speed depending on the PWM levels sent to it through the switching MOSFETS.

The FPGA development board used for this output the signals through a PMOD port, which controlled the switching MOSFETS assembled for testing of this project. These MOSFETS controlled the three phases of the motor, which directly sent back hall effect sensor feedback to the FPGA PMOD port.

The implementation of this work leaves room for many improvements and extra features for the controller. The VHDL code is modular and extra features can be added to it. The hardware portion of this project is not documented at adequate level and needs further research, testing and improvement.

With the results of this thesis, a solid base for a more advanced FPGA-based BLDC motor controller has been achieved, with a documentation that gives good knowledge of how to reproduce the work done by this far.

REFERENCES

[1] P. Yedamanle, “Brushless DC (BLDC) Motor Fundamentals”, AN885, Microchip Technology Inc., 2003, http://ww1.microchip.com/downloads/en/AppNotes/00885a.pdf Referred to 15.08.2021

[2] Image from https://www.electronicshub.org/wp-content/uploads/2019/09/BLDC-Working-1.jpg Referred to 24.9.2021

[3] A. Sathyan , N. Milivojevic,Y. Lee , M. Krishnamurthy and A. Emadi, "An FPGA-Based Novel Digital PWM Control Scheme for BLDC Motor Drives", in IEEE Transactions on Industrial Electronics, vol. 56, no. 8, pp. 3040-3049, Aug. 2009, doi: 10.1109/TIE.2009.2022067.

[4] D. Collins, 2017: “FAQ: What are Hall effect sensors and what is their role in dc motors?”, Motion Control Tips, 11.01.2017 https://www.motioncontroltips.com/faq-what-are-hall-effect-sensors-and-what-is-their-role-in-dc-motors/ Referred to 16.08.2021

[5] J. Serrano 2008: Introduction to FPGA design. CAS – CERN Accelerator School: Digital Signal Processing, pp.231-247. 22.04.2008 https://cds.cern.ch/record/1100537/files/p231.pdf Referred to 15.08.2021

[6] FPGA Vs Microcontroller – Which is better for your needs. OurPCB https://www.ourpcb.com/fpga-vs-microcontroller.html Referred to 16.06.2021

[7] P. Pdv Sai 2020: ASIC vs FPGA. Signoff-scribe 26.02.2020. http://www.signoffsemi.com/asic-vs-fpga/ Referred to 15.08.2021

[8] 2018: “PYNQ-Z2 Reference Manual v1.0”, Technology Unlimited https://dpoauwgwqsy2x.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf Referred to 14.09.2021

[9] B. Ohare: “MOSFET Structure.” Wikipedia. Image from: https://en.wikipedia.org/wiki/MOSFET#/media/File:MOSFET_Structure.png Referred to 24.09.2021

[10] “A Complete Guide to MOSFET Transistors”, RS Components Ltd. https://uk.rs-online.com/web/generalDisplay.html?id=ideas-and-advice/mosfet-guide Referred to 24.9.2021

[11] MoSCow Method. Wikipedia. https://en.wikipedia.org/wiki/MoSCoW_method Referred to 13.09.2021

[12] RobotShop webstore https://www.robotshop.com/eu/en/brushless-dc-motor-controllers.html Referred to 14.09.2021

[13] AspenCore: The MOSFET, ElectronicsTutorials: https://www.electronics-tutorials.ws/transistor/tran_6.html Referred to 19.10.2021

[14] "How to select Hall-effect sensors for brushless dc motors", Honeywell International Inc. 12.2012: https://www.mouser.com/pdfdocs/Selecting-Hall-Effect-for-DC-Brushless-Motors.pdf Referred to 14.09.2021

[15] G. Mihalache and A. Ioan, "FPGA Implementation of BLDC Motor Driver with Hall Sensor Feedback," 2018 International Conference and Exposition on Electrical And Power Engineering (EPE), 2018, pp. 0624-0629, doi: 10.1109/ICEPE.2018.8559886.

[16] IRLZ34NPbF datasheet, International Recrifier: https://www.infineon.com/cms/en/product/power/mosfet/n-channel/irlz34n/ Referred to 5.11.2021

[17] A. Andrews, A. O'Fallon, T.Chen, "A Rule-Based Software Testing Method for VHDL Models.", IFIP VLSI-SoC 2003, IFIP WG 10.5 International Conference on Very Large Scale Integration of System-on-Chip. Referred to 10.11.21

[18] Embedds, "Software Debouncing of buttons", https://embedds.com/software-debouncing-of-buttons/ Referred to 8.11.2021

[19] P. Wadhwani, D. Joshi 22.09.2021: " Trapezoidal & Sinusoidal: Two BLDC Motor Controls", https://www.bacancytechnology.com/blog/trapezoidal-and-sinusoidal-bldc-motors Referred to 5.11.2021

1 VHDL CODE

main.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity main is
    Port ( sysclk : in std_logic;
           btn: in std_logic_vector(3 downto 0);
           jb_1, jb_2, jb_3: in std_logic;
           ja_0, ja_1, ja_2, ja_3, ja_4, ja_5, ja_6: out std_logic
           );
end main;


architecture Behavioral of main is

    constant CLOCK_FREQUENCY : integer
            := X; -- Insert the system base clock frequency here
    constant DEBOUNCE_DELAY: integer
            :=  X; -- Insert debounce delay here
    constant STATE_CHANGE_DELAY: integer
            := X; -- Insert state change delay here

    signal clk, reset : std_logic:= '0';
    signal bus_out : std_logic_vector(5 downto 0):= "000000";
    signal power_btn, reset_signal, speed_up, speed_down :
                                            std_logic := '0';
    signal pwm_duty_cycle: std_logic_vector(3 downto 0):= "0000";
    signal pwm_out: std_logic:= '0';
    signal state_change_flag : std_logic:= '0';
    signal hall_state : std_logic_vector(2 downto 0):= "000";
    signal state_change_timer_flag: std_logic:= '0';
    signal a_h, a_l, b_h, b_l, c_h, c_l: std_logic:= '0';
    signal power_on: std_logic:= '0';

    component clock_divider
           Generic( CLOCK_FREQUENCY:    integer );
           Port(    sysclk, reset:     in std_logic;
                    clock_out:         out std_logic );
    end component;

    component read_feedback
           Port (   input_1, input_2, input_3: in std_logic;
                    reset, clk: in std_logic;
                    state_change_flag: out std_logic;
                    hall_state: out std_logic_vector(2 downto 0));
    end component;

    component timer
           Generic( timer_total: integer );
           Port (   reset, clk: in std_logic;
                    ENA: in std_logic;
                    timer_out: out std_logic);
    end component;
```

```
    component commutation
          Port(  clk, reset:       in std_logic;
                 hall_state:       in std_logic_vector(2 downto 0);
                 bus_out:          out std_logic_vector(5 downto 0));
    end component;

    component button_debounce
          Generic( debounce_delay:    integer );
          Port(    clk, reset, btn:   in std_logic;
                   btn_out:           out std_logic );
    end component;

    component speed_select
          Port   (  clk:                    in std_logic;
                    reset:                  in std_logic;
                    speed_up, speed_down:   in std_logic;
                    duty_cycle:     out std_logic_vector(3 downto 0));
    end component;

    component pwm_module
          Generic( PWM_DIVIDER_DIVISION: integer );
          Port (
                   PWM_DUTY_CYCLE: in std_logic_vector(3 downto 0);
                   sysclk, reset: in std_logic;
                   pwm_out : out std_logic );
    end component;

begin

    CLK_DIV:           clock_divider
    generic map ( CLOCK_FREQUENCY => CLOCK_FREQUENCY )
    port map    ( sysclk => sysclk, reset => reset, clock_out => clk);

    READ_HALL:         read_feedback
    port map    ( input_1 => jb_1, input_2 => jb_2, input_3 => jb_3,
        reset=>reset, clk=>clk, state_change_flag=>state_change_flag,
        hall_state=> hall_state);

    DELAY_TIMER:       timer
    generic map ( timer_total => STATE_CHANGE_DELAY)
    port map    ( reset => reset, clk => sysclk,
        ENA => state_change_flag,
        timer_out=> state_change_timer_flag);

    COM_LOGIC:         commutation
    port map    ( clk => clk, reset => reset,
        hall_state => hall_state, bus_out => bus_out);

    BUTTON_0:          button_debounce
    generic map ( debounce_delay => DEBOUNCE_DELAY )
    port map    ( clk => clk, reset => reset, btn => btn(0),
        btn_out => power_btn);

    BUTTON_1:          button_debounce
    generic map ( debounce_delay => DEBOUNCE_DELAY )
    port map    ( clk => clk, reset => reset, btn => btn(1),
        btn_out => speed_up);
```

```
    BUTTON_2:           button_debounce
    generic map ( debounce_delay => DEBOUNCE_DELAY )
    port map    ( clk => clk, reset => reset, btn => btn(2),
        btn_out => speed_down);

    SPD_SLCT:           speed_select
    port map    ( clk => clk, reset => reset, speed_up => speed_up,
        speed_down=> speed_down, duty_cycle=>pwm_duty_cycle);

    BUTTON_3:           button_debounce
    generic map ( debounce_delay => DEBOUNCE_DELAY )
    port map    ( clk => clk, reset => reset, btn => btn(3),
        btn_out => reset_signal);

    PWM_SIGNAL:         pwm_module
    generic map ( PWM_DIVIDER_DIVISION=>CLOCK_FREQUENCY )
    port map    ( sysclk=> sysclk, reset=> reset,
        PWM_DUTY_CYCLE => pwm_duty_cycle, pwm_out => pwm_out);

    OUTPUT_PROC: process(clk) begin
        if rising_edge(clk) then
            if state_change_timer_flag = '1' then
                a_h <= '0';
                a_l <= '0';
                b_h <= '0';
                b_l <= '0';
                c_h <= '0';
                c_l <= '0';
            else
                a_h <= bus_out(0);
                a_l <= bus_out(1);
                b_h <= bus_out(2);
                b_l <= bus_out(3);
                c_h <= bus_out(4);
                c_l <= bus_out(5);
            end if;
            if power_btn = '1' then
                power_on <= not power_on;
            end if;
        end if;
    end process;
    ja_0 <= a_h when pwm_out = '1' else '0';
    ja_1 <= a_l;
    ja_2 <= b_h when pwm_out = '1' else '0';
    ja_3 <= b_l;
    ja_4 <= c_h when pwm_out = '1' else '0';
    ja_5 <= c_l;
    ja_6 <= power_on;
end Behavioral;
```

pwm.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pwm_module is
  Generic(
  PWM_DIVIDER_DIVISION: integer );
  Port (
  PWM_DUTY_CYCLE: in std_logic_vector(3 downto 0);
  sysclk, reset: in std_logic;
  pwm_out : out std_logic);
end pwm_module;

architecture Behavioral of pwm_module is

signal PWM_DIVIDER: integer:= 125000000 / PWM_DIVIDER_DIVISION;
signal PWM_TRIGGER: integer:= PWM_DIVIDER / 2;
signal duty_cycle: integer:= to_integer(unsigned(PWM_DUTY_CYCLE));
signal count: integer:= 1;
signal tmp: std_logic:= '0';

begin
process(sysclk, reset) begin

if(reset = '1') then
    count<= 1;
    tmp<= '0';
elsif(sysclk'event and sysclk='1') then
    count<=count+1;
    if(count = PWM_DIVIDER) then
        count<= 1;
    elsif (count < PWM_TRIGGER) then
        tmp<= '1';
    elsif (count >= PWM_TRIGGER) then
        tmp<= '0';
    end if;
    if(duty_cycle = 0) then
        PWM_TRIGGER<= 0;
    else
        PWM_TRIGGER<=  PWM_DIVIDER / duty_cycle;
    end if;
end if;


end process;
pwm_out<= tmp;
duty_cycle<= to_integer(unsigned(PWM_DUTY_CYCLE));
end Behavioral;
```

clock_divider.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity clock_divider is
    Generic     (
                CLOCK_FREQUENCY: integer
                );
    Port        (
                sysclk, reset:  in std_logic;
                clock_out:      out std_logic
                );
end clock_divider;

architecture Behavioral of clock_divider is

signal CLOCK_DIVIDER_DIVISION: integer:= 125000000 / CLOCK_FREQUENCY;
signal CLOCK_DIVIDER_HALF: integer:= CLOCK_DIVIDER_DIVISION / 2;
signal count: integer:= 1;
signal tmp: std_logic:= '0';

begin
process(sysclk, reset) begin


if(reset = '1') then
    count<= 1;
    tmp<= '0';
elsif(sysclk'event and sysclk='1') then
    count<=count+1;
    if(count = CLOCK_DIVIDER_DIVISION) then
        count<= 1;
        tmp<='1';
    elsif (count = CLOCK_DIVIDER_HALF) then
        tmp<= '0';
    end if;
end if;
clock_out<= tmp;
end process;
end Behavioral;
```

read_feedback.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity read_feedback is
  Port ( input_1, input_2, input_3: in std_logic;
         reset, clk: in std_logic;
         state_change_flag: out std_logic;
         hall_state: out std_logic_vector(2 downto 0));
end read_feedback;

architecture Behavioral of read_feedback is

Signal stored_state: std_logic_vector(2 downto 0):= "000";
Signal new_state: std_logic_vector(2 downto 0);

begin

READ_AND_STORE_SIGNALS: process(clk, reset) begin

if (reset = '1') then
    state_change_flag <= '0';
    hall_state<= (others => '0');
    stored_state<= (others=> '0');
elsif rising_edge(clk) then
    new_state(0)<= not input_1;
    new_state(1)<= not input_2;
    new_state(2)<= not input_3;
    if(stored_state /= new_state) then
        state_change_flag<= '1';
        hall_state<=new_state;
        stored_state<=new_state;
    else
        state_change_flag<='0';
    end if;
end if;
end process;
end Behavioral;
```

speed_select.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;

entity speed_select is
    Port        (
                clk:                    in std_logic;
                reset:                  in std_logic;
                speed_up, speed_down:   in std_logic;
                duty_cycle:           out std_logic_vector(3 downto 0)
                );
end speed_select;

architecture Behavioral of speed_select is

type pwm_values is array (0 to 4) of std_logic_vector(3 downto 0);
signal pwm_value: pwm_values:= ("0000", "1000", "0100", "0010",
"0001");
signal current_value: integer range 0 to 4;
signal buf: integer:= 0;
signal up, down: std_logic;

begin

process(clk, reset) begin

    if reset = '1' then
        current_value<= 0;
    elsif rising_edge(clk) then
        if speed_up = '1' and up = '0' and speed_down = '0' then
            if (current_value = 4) then
                current_value<= 0;
            else
                current_value<=current_value+1;
            end if;
        elsif speed_down = '1' and down = '0' and speed_up = '0' then
            if (current_value <= 0) then
                current_value <= 4;
            else
                current_value<=current_value-1;
            end if;
        else
            current_value<= current_value;
        end if;
        up<= speed_up;
        down<= speed_down;
    end if;
end process;
duty_cycle<=pwm_value(current_value);
end Behavioral;
```

button_debounce.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity button_debounce is
    Generic     (
                debounce_delay: integer
                );
    Port        (
                clk, reset, btn:  in std_logic;
                btn_out:      out std_logic
                );
end button_debounce;

architecture Behavioral of button_debounce is

signal count: integer:= 0;
signal tmp: std_logic:= '0';

begin
process(clk, reset) begin

if(reset = '1') then
    count<= 1;
    tmp<= '0';
elsif(clk'event and clk='1') then
    if(btn = '0') then
        count<= 0;
        tmp<= '0';
    else
        if (count = debounce_delay) then
            count<= count + 1;
            tmp<='1';
        else
            count<=count+1;
            tmp<='0';
        end if;
    end if;
end if;
btn_out <= tmp;
end process;
end Behavioral;
```

delay_timer.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity timer is
  Generic(  timer_total: integer );
  Port (    reset, clk: in std_logic;
            ENA: in std_logic;
            timer_out: out std_logic);
end timer;

architecture Behavioral of timer is

Signal timer_count: integer:= 1;
Signal timer_flag: std_logic:= '0';

begin

TIMER: process(clk, reset) begin

if (reset = '1') then
    timer_count<= 1;
    timer_flag<= '0';
elsif rising_edge(clk) then
    if ENA = '1'  then
        timer_flag <= '1';
    elsif timer_flag = '1' then
        timer_count <= timer_count + 1;
        if timer_count = timer_total then
            timer_flag<= '0';
            timer_count<= 1;
        end if;
    end if;
end if;
end process;
timer_out <= timer_flag;
end Behavioral;
```

commutation.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity commutation is
    Port ( clk, reset   : in std_logic;
           hall_state : in std_logic_vector(2 downto 0);
           bus_out      : out std_logic_vector(5 downto 0)
           );
end commutation;

architecture Behavioral of commutation is

Constant delay_time: integer:= 10;

Type Commutation_state is(p_1, p_2, p_3, p_4, p_5, p_6, p_off);
Signal State, State_next : Commutation_state;
Signal output: std_logic_vector(5 downto 0):= "000000";
Signal delay_time_count: integer:= 0;

begin

COM_PROCESS: process(clk, reset) begin
if reset = '1' then
    output<="000000";
elsif rising_edge(clk) then
    if hall_state = "001" then
        output<="001001";
    elsif hall_state = "011" then
        output<="100001";
    elsif hall_state = "010" then
        output<="100100";
    elsif hall_state = "110" then
        output<="000110";
    elsif hall_state = "100" then
        output<="010010";
    elsif hall_state = "101" then
        output<="011000";
    else
        output <= "000000";
    end if;
end if;
bus_out<= output;
end process;
end Behavioral;
```