CHAPTER 6

SINGLE CHIP REALISATION OF INTEGRATED FUZZY LOGIC CONTROLLER USING FPGA

6.1 INTRODUCTION TO FPGA

6.1.1 General Programmable Devices

Dynamic and ever progressing change in very large-scale integration technology (VLSI) has radically affected the design process. The life cycle of modern electronic products is sometimes shorter than its design cycle. Therefore, the need for rapid prototyping posses a greater challenge in design. In recent years, the development of application-specific integrated circuit (ASIC) technology has made it possible to integrate complex analog and digital circuits by utilising the libraries of basic circuit cells. The ASIC approach provides a rapid low-cost manufacturing solution for IC’s with special applications. For sophisticated technology which are linked to a medium-size market requirement, it is an optimal solution. However, the longer the lead-time and higher setup cost for prototyping render it inappropriate for product development in the early stage. Since the 1980’s, ASIC technology has given rise to several new specialised technologies, including mask-programmable gate array, cell-based IC (CBIC), programmable array logic (PAL) and field-programmable logic array (FPLA).

With the advancement of the various technical aspects of ASIC, three major categories have been defined: CBIC, gate array and programmable logic device (PLD). The CBIC has the longest lead-time with the highest number of gates, while the PLD allows the user to define the gate
connections, but with the less number of gates. A step above the PLD in complexity is the field-programmable gate array (FPGA). There is a little difference between an FPGA and a PLD – an FPGA is usually just larger and more complex than a PLD.

### 6.1.2 FPGA (Field Programmable Gate Array)

FPGA are arrays of logic blocks, which can be linked together to form complex logic implementations. FPGAs can be grouped into two categories:

1) Fine grained and
2) Coarse grained

Fine grained FPGAs are made up of a sea of gates or transistors or small macro cells, while coarse grained FPGAs are made up of bigger macro cells which are often made of flip flops and Look up Tables (LUTs) which make up the combinational logic functions. Inside the macro cell often switches or multiplexers are present, which allows the differing uses of the macro cell. The individual macro cells are connected together with a combination of switch matrixes and metal line matrixes, which can be implemented with, pass transistors, fuse/antifuses or multiplexes.

A FPGA is a chip which can be programmed. An IC foundry produces FPGAs with certain open connections. Design entry and simulation can be performed, using special software that creates a string of bits, describing the extra connections required to make the design – the configuration file. The configuration file can be designed by programming the chip by connecting to the personal computer. FPGAs are ideal for prototyping systems or for low-volume production.
All FPGAs have certain common key elements. All FPGAs have a regular array of basic logic cells that are configured using a programming technology. The chip inputs and outputs use special I/O logic cells which are different from the basic logic cells. A programmable interconnect scheme forms the wiring between the two types of logic cells. Finally, the designer uses custom software, tailored to each programming technology and FPGA architecture, to design and implement the programmable connections. The programming technology in an FPGA determines the types of basic logic cell and the interconnect scheme. The logic cells and interconnect scheme, in turn, determine the design of the input and output circuits as well as the programming scheme. The programming technology may or may not be permanent. Undoing the permanent programming is not possible in one-time programmable (OTP) FPGAs. Reprogrammable or erasable devices may be reused many times. Various programming technologies of FPGA are antifuse, SRAM and EPROM technologies. All FPGAs have the following key elements:

- The programming technology
- The basic logic cells
- The I/O logic cells
- Programmable interconnect
- Software to design and program the FPGA.

Xilinx uses the Static RAM programming technology. In this technology, designers can reuse chips during prototyping and a system can be manufactured using ISP. This programming technology is also useful for upgrades – a customer can be sent a new configuration file to reprogram a chip, instead of a new chip. Designers can also update or change a system on the fly in reconfigurable hardware. The problem with this technology is that, power is to be supplied constantly to the programmable ASIC as it is a volatile SRAM.
6.2 NEED FOR FPGA

Hardware implementation of fuzzy logic controllers can be achieved in a number of ways to create new products based on fuzzy logic. The most popular method of implementing a FLC is by using a general-purpose microprocessor or micro controller. Generally, the FLC is verified and implemented in software and then executed on the microprocessor. Microprocessor based fuzzy systems are more economical and flexible and work for a wide range of systems but they often have difficulty in dealing with control systems that require high processing and I/O handling speeds. ASICs are used in situations where high speed is required. Even though it runs 10 to 100 times faster than microprocessors, no changes can be made once the chip is created. Another hardware solution to solve the above problems is the use of an FPGA, which is suitable for fast implementation and quick hardware verification.

A field programmable gate array is a digital integrated circuit that can be programmed to do any type of digital function. There are two main advantages of using a FPGA over a microprocessor chip for controller:

- An FPGA has the ability to operate faster than a microprocessor chip.
- The new FPGA’s that are on the market will support hardware that is upwards of 1 million gates.

FPGAs with their static RAM technology core provide a low power consumption, high logical capacity and speed with unlimited number of programming cycles. They make possible user programmable hardware subsystems which may be easily reloaded, even “on-fly”. Another important advantage of FPGAs is their capability of parallel operation that can greatly increase the execution speed of the control algorithm. The typical FPGA
operating voltage varies between 1.8 and 3.3 Volts, while it is normal for micro controllers to run on 3.3 to 5 Volts. The FPGAs also consume less current per hour than that of the micro controllers. The result is less power consumption, which in today’s world is most expected solution in all means.

In the present work Xilinx Spartan 2S300E FPGA was used for implementation.

6.3 XILINX SPARTAN2E

6.3.1 General Overview

The Spartan™-IIE 1.8V Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The seven-member family offers densities ranging from 50,000 to 600,000 system gates, as shown in Table 6.1 (www.xilinx.com).

**Table 6.1 Spartan2E FPGA Family members**

<table>
<thead>
<tr>
<th>Device</th>
<th>Logic Cells</th>
<th>Typical System Gate Range</th>
<th>CLB Array (R x C)</th>
<th>Total CLBs</th>
<th>Maximum Available User I/O</th>
<th>Maximum Differential I/O Pairs</th>
<th>Distributed RAM Bits</th>
<th>Block RAM Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC2S500E</td>
<td>1,728</td>
<td>23,000 - 50,000</td>
<td>16 x 24</td>
<td>984</td>
<td>182</td>
<td>83</td>
<td>24,576</td>
<td>32K</td>
</tr>
<tr>
<td>XC2S100E</td>
<td>2,700</td>
<td>37,000 - 100,000</td>
<td>20 x 30</td>
<td>800</td>
<td>202</td>
<td>86</td>
<td>38,400</td>
<td>40K</td>
</tr>
<tr>
<td>XC2S150E</td>
<td>3,888</td>
<td>52,000 - 150,000</td>
<td>24 x 36</td>
<td>864</td>
<td>265</td>
<td>114</td>
<td>55,296</td>
<td>48K</td>
</tr>
<tr>
<td>XC2S200E</td>
<td>5,202</td>
<td>71,000 - 200,000</td>
<td>28 x 42</td>
<td>1,176</td>
<td>280</td>
<td>120</td>
<td>75,264</td>
<td>56K</td>
</tr>
<tr>
<td>XC2S300E</td>
<td>6,912</td>
<td>93,000 - 300,000</td>
<td>32 x 48</td>
<td>1,596</td>
<td>329</td>
<td>120</td>
<td>99,304</td>
<td>64K</td>
</tr>
<tr>
<td>XC2S400E</td>
<td>10,800</td>
<td>145,000 - 400,000</td>
<td>40 x 60</td>
<td>2,400</td>
<td>410</td>
<td>172</td>
<td>153,600</td>
<td>180K</td>
</tr>
<tr>
<td>XC2S600E</td>
<td>15,552</td>
<td>210,000 - 600,000</td>
<td>48 x 72</td>
<td>3,456</td>
<td>514</td>
<td>205</td>
<td>221,184</td>
<td>288K</td>
</tr>
</tbody>
</table>

System performance is supported beyond 200 MHz. Spartan2E devices deliver more gates, I/Os than other FPGAs by combining advanced process technology with a streamlined architecture based on the proven Virtex™-E platform. Features include block RAM (to 288K bits), distributed RAM (to 221,184 bits), 19 selectable I/O standards, and four DLLs
(Delay-Locked Loops). Fast, predictable interconnect means that successive design iterations continue to meet timing requirements.

### 6.3.2 Features

- **Second generation ASIC replacement technology**
  - Densities as high as 15,552 logic cells with up to 600,000 system gates
  - Unlimited in-system reprogrammability
  - Low in cost

- **System level features**
  - Select RAM+™ hierarchical memory:
    - 16 bits/LUT distributed RAM
    - Configurable 4K-bit true dual-port block RAM
    - Fast interfaces to external RAM
  - Fully 3.3V PCI compliant to 64 bits at 66 MHz and Card Bus compliant
  - Low-power segmented routing architecture
  - Dedicated carry logic for high-speed arithmetic
  - Efficient multiplier support
  - Cascade chain for wide-input functions
  - Abundant registers/latches with enable, set, reset
  - Four dedicated DLLs for advanced clock control
    - Eliminate clock distribution delay
    - Multiply, divide, or phase shift

- **Versatile I/O and packaging**
  - 19 high-performance interface standards
    - LVTTL, LVCMOS, HSTL, SSTL, AGP, CTT, GTL
    - LVDS and LVPECL differential I/O
- Up to 205 differential I/O pairs that can be input, output, or bidirectional

- Fully supported by powerful Xilinx ISE development system
  - Fully automatic mapping, placement, and routing
  - Integrated with design entry and verification tools
  - Extensive IP library including DSP functions and soft processors

The Spartan2E family of FPGAs has a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable Input/Output Blocks (IOBs). There are four Delay-Locked Loops (DLLs), one at each corner of the die. Two columns of block RAM lie on opposite sides of the die, between the CLBs and the IOB columns. A powerful hierarchy of versatile routing channels as shown in Figure 6.1 interconnects these functional elements (www.xilinx.com).

Loading configuration data into internal static memory cells customises Spartan2E FPGAs. Stored values in these cells determine logic functions and interconnections implemented in the FPGA. Configuration data can be read from an external serial PROM (master serial mode), or written into the FPGA in slave serial, slave parallel, or Boundary Scan modes. Spartan2E devices provide system clock rates beyond 200 MHz. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan2E FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and distributed form), DLL clock drivers, programmable set and reset on all flip-flops, fast carry logic, and many other features.
Figure 6.1 Basic Spartan 2E family FPGA block diagram

Table 6.2 Spartan2E user I/O chart

<table>
<thead>
<tr>
<th>Device</th>
<th>Maximum User I/O</th>
<th>Available User I/O According to Package Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TO144</td>
</tr>
<tr>
<td>XC2S50E</td>
<td>182</td>
<td>102</td>
</tr>
<tr>
<td>XC2S100E</td>
<td>202</td>
<td>102</td>
</tr>
<tr>
<td>XC2S150E</td>
<td>265</td>
<td>-</td>
</tr>
<tr>
<td>XC2S200E</td>
<td>289</td>
<td>-</td>
</tr>
<tr>
<td>XC2S300E</td>
<td>329</td>
<td>-</td>
</tr>
<tr>
<td>XC2S400E</td>
<td>410</td>
<td>-</td>
</tr>
<tr>
<td>XC2S600E</td>
<td>514</td>
<td>-</td>
</tr>
</tbody>
</table>
The Xilinx naming system is interpreted as follows:

Example: \textbf{XC2S50E -6 PQ 208 C}

- **Device Type**
- **Speed Grade**
- **Temperature Range**
- **Number of Pins**
- **Package Type**

6.3.3 Building Blocks

The Spartan2E user-programmable gate array, shown in Figure 6.1, is composed of five major configurable elements:

- IOBs that provide the interface between the package pins and the internal logic
- CLBs that provide the functional elements for constructing most logic
- Dedicated block RAM memories of 4096 bits each
- Clock DLLs for clock-distribution delay compensation and clock domain control
- Versatile multi-level interconnect structure

From Figure 6.1, it is clear that the CLBs form the central logic structure with easy access to all support and routing structures. The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip. Values stored in static memory cells control all the configurable logic elements and interconnect resources. These values load into the memory cells on power-up, and can reload if necessary to change the function of the device.
6.3.3.1 Input/Output Block

The Spartan2E IOB, as shown in Figure 6.2 (www.xilinx.com), features inputs and outputs that support a wide variety of I/O signaling standards. These high-speed inputs and outputs are capable of supporting various state of the art memory and bus interfaces. Table 6.2 lists several of the standards, which are supported along with the required reference, output and termination voltages needed to meet the standard (www.xilinx.com). The three IOB registers function either as edge-triggered D-type flip-flops or as level-sensitive latches. Each IOB has a clock signal (CLK) shared by the three registers and independent Clock Enable (CE) signals for each register.

![Figure 6.2 Spartan2E Input/Output Block (IOB)](image)

6.3.3.2 I/O Banking

Eight I/O banks result from separating each edge of the FPGA into two banks as shown in Figure 6.3 (www.xilinx.com). Each bank has multiple VCCO pins, which must be connected to the same voltage.
6.3.3.3 Configurable Logic Block

The basic building block of the Spartan2E CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and storage element. The output from the function generator in each LC drives the CLB output or the D input of the flip-flop. Each Spartan2E CLB contains four LCs, organised in two similar slices; a single slice is shown in Figure 6.4 (www.xilinx.com). In addition to the four basic LCs, the Spartan2E CLB contains logic that combines function generators to provide functions of five or six inputs.

6.3.3.4 Look-Up Tables

Spartan2E function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16 × 1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16 × 2-bit or 32 × 1-bit synchronous RAM, or a 16 × 1-bit dual-port synchronous RAM. The Spartan2E LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-
mode data. This mode can also be used to store data in applications such as Digital Signal Processing.

Figure 6.4 Spartan2E CLB Slice (two identical slices in each CLB)

6.3.3.5 Storage Elements

Storage elements in the Spartan2E slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by function generators within the slice or directly from slice inputs, bypassing the function generators. In addition to Clock and
Clock Enable signals, each slice has synchronous set and reset signals (SR and BY). SR forces a storage element into the initialisation state specified for it in the configuration. One of the Reset signal BY forces into the opposite state. Alternatively, these signals may be configured to operate asynchronously. All control signals are independently invertible, and are shared by the two flip-flops within the slice.

6.3.3.6 Development System

The Xilinx ISE Foundation and Alliance CAE tools support Spartan2E FPGAs. The basic methodology for Spartan2E design consists of three interrelated steps: design entry, implementation, and verification. Industry-standard tools are used for design entry and simulation, while Xilinx provides proprietary architecture-specific tools for implementation. The Xilinx development system is integrated under the Xilinx Project Navigator software, providing designers with a common user interface regardless of their choice of entry and verification tools.

A unified library of standard functions supports Spartan2E FPGAs. This library contains over 400 primitives and macros, ranging from 2-input AND gates to 16-bit accumulators, and includes arithmetic functions, comparators, counters, data registers, decoders, encoders, I/O functions, latches, Boolean functions, multiplexes, shift registers, and barrel shifters.

6.3.3.7 Configuration

Configuration is the process by which the bit stream of a design, as generated by the Xilinx development software, is loaded into the internal configuration memory of the FPGA. Spartan2E devices support both serial configurations, using the master/slave serial and JTAG modes, as well as byte-wide configuration employing the Slave Parallel mode.
6.3.3.8 Configuration File and Modes

Spartan2E devices are configured by sequentially loading frames of data that have been concatenated into a configuration file. Spartan2E devices support the following four configuration modes:

- Slave Serial mode
- Master Serial mode
- Slave Parallel mode
- Boundary-scan mode

Among these four modes, the slave serial mode is used for programming the FPGA.

6.4 HARDWARE DESCRIPTION LANGUAGES (HDL)

HDL is used to describe how hardware behaves. Major differences between traditional programming languages and HDL are:

1) Traditional languages are a sequential process whereas HDL is a parallel process,
2) HDL runs whereas traditional programming languages will only run if directed.

This leads to confusion as how some things may be implemented in HDL and other things cannot. For example, if the following two lines of code

\[
C = A + B; \quad (6.1) \\
A = C + D; \quad (6.2)
\]

are analyzed using a software approach and a hardware approach, then in traditional programming languages, the first line of code in equation (6.1) would execute and result in the addition of A and B and storing it into C, and
then taking C and D and adding together and storing it back in A as shown in equation (6.2). In software this is possible and is used very often. In HDL the two lines would execute at the same time causing combinational logic feedback. The most popular style of HDL is high level behavioral. This is the highest level and can model large designs. Behavioral style uses many of the same syntax that traditional programming languages use. This style of HDL is the most powerful level of HDL and can be written much faster than a schematic one.

The major drawback of traditional design methods is the manual translation of design description into a set of logical equations. This step can be entirely eliminated with HDLs. For example, most HDL tools allow the use of finite state machines for sequential systems and truth tables for combinatorial modules. Such designs can be automatically converted into HDL code that can be implemented by synthesis tools. HDL found their principal application in Programmable Logic Device (PLD) of various complexities, from simple PLDs up to complex PLDs and FPGA. There are several HDL languages in use today. The most popular ones are VHDL, Verilog and Abel.

6.5 OVERVIEW OF VHDL PROGRAMMING

A digital design can be created using schematic digital design editor that uses graphic symbols of the circuit or by using hardware description languages such as Verilog, Very High Speed Integrated Circuit hardware description language (VHDL). One of the key features of using VHDL is that it can be used to achieve all the goals for documentation, simulation, verification and synthesis of digital designs, thus saving a lot of effort and time. VHDL can be used to model a digital system at many levels of abstraction, ranging from the algorithmic level to the gate level.
The VHDL language can be regarded as an integrated amalgamation of the following languages:

Sequential language + Concurrent language + Net-list language + timing-specifications + waveform generation language => VHDL. Hence, the constructed language enables to express the concurrent or sequential behavior of a digital system with or without timing. Since this research requires reading the inputs from the A/D converter, sending the outputs to LCD and motor and determination of the motor speed which are to be performed concurrently, implementation of the fuzzy controller on a FPGA is carried out using VHDL.

6.5.1 Primary Design Unit Model Structure

Whatever the function a system performs, it must get some input data from its environment and give some data as output. The communication part of a system specification is called the interface. Without an interface, the system would be useless – just like the most popular computer in the world without a keyboard, mouse or a monitor. Thus interface is of primary importance.

The system’s interface is described in VHDL by its entity, which is the basic design unit for any system. To accomplish the desired functionality, data must undergo some transformation inside the system. This transformation of data and outputting of desired functions is handled by the system’s inner part or body, which is called architecture. However, a system can be considered as a composition of a body and an interface.

Each VHDL design unit comprises an "entity" declaration and one or more "architectures". Each architecture defines a different implementation or model of a given design unit. The entity definition defines the inputs to,
and outputs from the module, and any "generic" parameters used by the
different implementations of the module.

6.5.2 Entity Declaration Format

    entity name is
    port(port definition list); -- input/output signal ports
    generic(generic list); -- optional generic list
    end name;

Port declaration format: \textit{port\_name: mode data\_type;}

The \textit{mode} of a port defines the directions of the signals on that pin, and is one of: \textbf{in}, \textbf{out}, \textbf{buffer}, or \textbf{inout}.

6.5.3 Port Modes

- An \textbf{in} port can be read but not updated within the module, carrying information into the module (an in port cannot appear on the left hand side of a signal assignment).

- An \textbf{out} port can be updated but not read within the module, carrying information out of the module (an out port cannot appear on the right hand side of a signal assignment).

- A \textbf{buffer} port likewise carries information out of a module, but can be both updated and read within the module.

- An \textbf{inout} port is bi-directional and can be both read and updated, with multiple update sources possible.
Example

Entity fulladder is

```vhdl
port (Incr, Load, Clock: in bit;
    Carry: out bit;
    Data_Out: buffer bit_vector (7 downto 0);
    Data_In: in bit_vector (7 downto 0));
end fulladder;
```

Generics allow static information to be communicated to a block from its environment for all architectures of a design unit. These include timing information (setup, hold, delay times), part sizes, and other parameters.

Example

entity and_gate is

```vhdl
port(a,b: in bit;
    c: out bit);
    generic (gate_delay: time := 5ns);
end and_gate;
```

6.5.4 Architecture

An architecture defines one particular implementation of a design unit, at some desired level of abstraction.

```vhdl
architecture arch_name of entity_name is
    ... declarations ...
begin
    ... concurrent statements ...
end
```

The internal details of an entity are specified by an architecture body using any of the following modeling styles:
• **Behavioral Model**: No structure or technology implied. Usually written in sequential, procedural style.
• **Dataflow Model**: All data paths shown, plus all control signals.
• **Structural Model**: Interconnection of components.
• As any of the above combination

### 6.5.4.1 Dataflow Style of Modeling

In this style of modeling, the flow of data through the entity is expressed primarily using concurrent signal assignment statements.

**Syntax:**

```
  Sum  <= a xor b;
  Carry <= a and b;
```

### 6.5.4.2 Behavior Style of Modeling

This type of modeling specifies the behavior of an entity as a set of statements that are executed sequentially in the specified order. These sets of sequential statements, which are specified inside a process statement, do not explicitly specify the structure of the entity but merely its functionality. A process statement is a concurrent statement that can appear within an architecture body.

**Syntax:**

```
label: process (sensitivity list)
  ... local declarations ...
begin
  ... sequential statements ...
end process label;
```
6.5.4.3 Structural Style of Modeling

In this style of modeling, an entity is described as a set of interconnected components. Instantiates (i.e. create instances of) predefined components within a design architecture. Each such component is first declared in the declaration section of that architecture, and then "instantiated" one or more times in the body of the architecture. Component instantiation statements are concurrent statements.

Syntax:

--component declaration

Component component-name [is]
[port (list of interface ports);]
end component [component name];

--component instantiation

component-label component-name [port map (association list)];

The coding is carried out at the appropriate places by using the various modeling.

6.6 IMPLEMENTATION OF FUZZY CONTROLLER IN FPGA

Hardware implementation of the fuzzy controllers can be achieved in a number of ways. In the present study, the fuzzy controller was developed in FPGA using VHDL. The fuzzy operations such as fuzzification, rule base and defuzzification were coded in VHDL and along with that the interfacing of ADC to get the crisp inputs and LCD to receive the crisp outputs was also performed. Then the output of the fuzzy controller is given to the driver unit to control the motor speed. By using optical encoder technique, the speed of the motor was sensed and given as feedback to the system.
The parameters such as acceleration, braking and state of charge were taken as analog inputs from potentiometer while terrain and gear selection were taken as digital inputs from thumb wheel switches. The design of the controller consists of an FPGA, analog-to-digital converter, potentiometer, thumb wheel switches for the inputs, motor, driver circuit and LCD for the output. A block diagram for the controller was shown in Figure 6.5.

![Block diagram of the controller with FPGA.](image)

In the present work Xilinx XC2S300E FPGA was used to implement the controller. It has been used to sense the above inputs and determine the pulse width modulation for the motor. The FPGA was interfaced with the ADC to get the digital data for the inputs acceleration, brake and state of charge of battery. For the selection of the gear and terrain, thumb wheel switches have been used. Besides this, the output of the FPGA can be seen in the Liquid Crystal Display (LCD) for the parameters like gear, motor speed, and battery status. The PWM signal output from the FPGA was taken to drive the motor through the driver interface.

### 6.7 HARDWARE FEATURES

The hardware consists of one 20 character by 4 line LCD display, potentiometers for acceleration and braking, Battery, ADC 0804, Motor, IR Sensor, Mosfet driver circuit, voltage regulator circuit and five toggle
switches. Figure 6.6 shows the hardware interface block diagram for the design work.

![Figure 6.6 Block Diagram of Hardware Interface](image)

To get the acceleration data, a 47K potentiometer was used. Likewise, another potentiometer was used to get the brake input. A lead acid battery of 12V 7.5 AH was used to give the power supply for the whole module, since the electric vehicle was a battery operated vehicle. The battery’s state of charge (i.e.) low, normal, or high can be determined by connecting the battery with the ADC to get the digital input. The ADC 0804 was a single channel, 8-bit analog to digital converter from national semiconductor. Its operating voltage was 5V DC with 100 μs conversion times. A 12V DC motor of 2400 rpm was used. Two thumb wheel switches were used for the gear selection and for the terrain selection respectively. The speed of the motor would be different for each terrain based on the other inputs. A separate battery charger circuit had been designed using bridge rectifier and LM317 voltage regulator for the offline battery charging and power supply circuit has been designed using LM 7805 voltage regulator to
convert the 12V to 5V in order to drive the Xilinx, ADC and LCD components as shown in Appendix - 1.

6.8 DATA ACQUISITION UNIT

Data acquisition unit drive ADC 0804 to get the crisp inputs (acceleration, braking & state of charge). The analog inputs were taken from potentiometers or from the battery. The analog values are to be converted into digital values before giving to the FPGA. Therefore, analog to digital converter (ADC 0804) chip is used. The following steps were carried out for data conversion by the ADC 0804 chip:

1. Make the CS = 0 and send a low to high pulse to pin WR to start the conversion.
2. Wait for the data conversion process to complete (approximately 125 us)
3. Make the CS = 0 and send a low to high pulse to pin RD to get the data out of the ADC 0804.

Figure 6.7 shows the write timing diagram for ADC 0804. Figure 6.8 shows the read timing diagram for ADC 0804. Figure 6.9 shows the way of interfacing the ADC with the Xilinx. Xilinx Xc2s300e-6pq208’s clock has the capability to operate from 1MHz to 100 MHz. For the convenience, the clock in the board is set to 20 MHz. And, a clock divider module generates the 8 KHz clock required for the ADC since the ADC conversion time is approximately 100 μs.
Figure 6.7 Write timing diagram for ADC 0804

Figure 6.8 Read timing diagram for ADC 0804

Figure 6.9 Interfacing of ADC 0804 with FPGA
The generation of the control signals for the ADC at the appropriate moments and fetch the 8-bit digital data were coded in VHDL using the finite state machine. The mealy type state machine was used. In this type of state machine, the outputs not only depend on the state of the machine but also on its input. This type of machine can be modeled using two processes in VHDL coding, among which one process that models the synchronous aspect of the finite state machine, and the other that models the combinational part. Figure 6.10 shows the state flow diagram. In this diagram, idle, delay, soc (start conversion), eoc (end of conversion) are the states, reset is the input signal and cs, rd, wr and data are the output signals. The control signals cs, rd and wr are active low signals. The 8-bit digital data are available at the data pins of the ADC at the eoc state. In this way the acceleration, braking and the state of the charge of the battery were read and given as inputs to the fuzzifier module.

The other two inputs such as gear and terrain were read from the thumb wheel switches. The gear selection and terrain selection were performed based on the data shown in Tables 6.3 and 6.4. The connection of Accelerator potentiometer, brake potentiometer, battery, gear switch and terrain switch with the FPGA are shown in Appendix A1.
Table 6.3  The Gear choices

<table>
<thead>
<tr>
<th></th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>GEAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Neutral</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1st</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2nd</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3rd</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4th</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Reverse</td>
</tr>
</tbody>
</table>

Table 6.4 Terrain types chosen

<table>
<thead>
<tr>
<th>Pins from switch</th>
<th>TERRAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 4 2 1</td>
<td>Smooth</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>Rough</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>Uphill</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>Downhill</td>
</tr>
</tbody>
</table>

6.9 IMPLEMENTATION OF CONTROLLER

The components of the fuzzy controller for the DC motor were implemented in VHDL. The specifications for the implementation of the FLC were, number of input variables were 6, number of output variable was 1, number of bits required to represent the input variables were 8, number of fuzzy sets for the variables vary from 3 to 6 and all the variables uses the non overlapping rectangular membership functions. The input parameters were
1) Speed 2) Acceleration 3) Braking 4) State of charge of battery 5) Gear and 6) Terrain. The output of the fuzzy system was the duty cycle of the pulse width modulation signal.

In the controller design, the speed was taken in the range of 00-FF. The range was equally divided into four linguistic variables namely very slow, slow, medium and fast. The brake parameter was taken in the range of 00-FF. The range was divided into three linguistic variables namely short, medium and high. The acceleration was taken in the range of 00-FF and their linguistic variables namely short, medium and high. The gear was taken in the range of 0-6 and classified into five linguistic variables namely Neutral, 1st gear, 2nd gear, 3rd gear, 4th gear and Reverse. The state of charge of the battery was taken in the range of 00-FF and accordingly its three linguistic variables were named as low, normal and high. The Terrain was taken in the range of 1-4 and it had been divided into four linguistic variables namely smooth, rough, uphill and downhill. The output parameter duty cycle of the pulse width modulation signal was taken in the range of 00-64 and it had been divided into four linguistic variables namely very low, low, medium and high. The implementation process was subdivided into three components namely fuzzifier, rule base and defuzzifier. The complete structure of the fuzzy controller with other blocks is shown in Figure 6.11. The function of the fuzzifier is to transform crisp inputs into fuzzy inputs. Crisp inputs for the accelerator, brake and battery are 8 –bit binary value representing the current reading and 4-bit binary value for gear and 10-bit binary value for gear and single bit stream of pulses for speed. The first step is to convert the crisp inputs to fuzzy inputs, for those comparisons of the crisp inputs with the membership function parameters of the variables respectively.

Rule evaluation is the second step of the fuzzy logic process, and determines what control action should occur in response to a given set of
input values. The rule evaluation method used here is “min-max” inferences, since it takes the minimum of the antecedents to determine rule strength and the maximum of the rule strengths for each consequent to determine fuzzy outputs.

Defuzzification is the last step in fuzzy logic process, which transforms the fuzzy outputs into crisp output based on the output membership function. In defuzzification, all significant outputs will be combined into a specific, comprehensive result to get crisp output. The most common defuzzification techniques called center of gravity or centroid method issued in the present work.

6.10 PULSE WIDTH MODULATION DRIVER UNIT

In this unit, the output from the fuzzy controller to the motor was considered. The input to the motor can be given in terms of the pulse width modulation signal. Since this system has been designed as a closed-loop, the feedback from the motor has to be taken. For getting the feedback, several devices are available. The basic function of feedback devices is to transform a physical parameter into an electrical signal for use by a motion controller. Common feedback devices are encoders for position feedback, tachometers for velocity feedback, and accelerometers for acceleration feedback.

6.10.1 Optical Encoders

Although there are various kinds of digital encoders, the most common is the optical encoder. Rotary and linear optical encoders are used frequently for motion and position sensing. A disc or a plate containing opaque and transparent segments passes between a light source (such an LED) and detector to interrupt a light beam. The disk, which is mounted on the rotating shaft, has coded patterns of opaque and transparent sectors. As the disk rotates, these patterns interrupt the light emitted, generating a digital or
pulse signal output. These signals that are generated are then fed into the controller where position information is calculated based upon the signals received.

Figure 6.11 Block diagram of the fuzzy control scheme of DC motor

6.10.2 Tachometers

For applications requiring velocity regulation, the speed can be either measured directly or derived from encoder supplied position information. For higher quality speed control, a tachometer is used which produces a voltage or current level proportional to the speed of the motor. Tachometer feedback can change instantaneously with speed change allowing faster correction and tighter regulation from a controller.

Among the above-mentioned ways to measure the speed of the motor, in the present investigation the optical encoder technique is used. The crisp output from the defuzzifier is given to the pulse width generator to
generate the pulse width modulation (PWM) signal to drive the motor as shown in Figure 6.12. The motor was connected to the FPGA through the driver circuit, which is shown in Appendix A1. Actually, this PWM signal was given as a gate signal to the mosfet in the drive unit, which in turn drives the motor depending upon the duty cycle of the signal. A totem pole transistor is used before the mosfet to decide whether the mosfet should be turned on or off. Also the direction of rotation (i.e.) clockwise or anticlockwise direction is decided by the control signal, which is generated from the FPGA. When the gear was chosen to be reverse, then the control signal was set at 0 to make the motor to rotate in the anticlockwise direction. Otherwise, this control signal set to 1 for all other gears to make the motor to rotate in clockwise direction. For achieving this operation, a relay coil with DPDT (Double Pole Double Throw switch) was used. For 100 % duty cycle, the motor runs at its maximum speed and for 10 %, it runs at its lower speed.

![Figure 6.12 Optical Encoder](image)

The pulse width modulation (pwm) signal was generated with reference to the external clock signal. Since the motor was running at a speed of 100 Hz, the pwm signal was generated for this frequency. The on and off period of the pwm signal was varied with reference to the generated clock frequency. These feedback signals from the motor that are generated are then
fed into the controller where pulses are counted based on the signals received and the speed is determined internally to decide the speed for the successive moments for the motor.

6.11 DISPLAY UNIT

The function of the display unit was to display the terrain type, gear chosen, accelerator status, braking status, battery charge status and speed of the motor in Liquid Crystal Display (LCD). In order to display any information in LCD, first it has to be initialised and then the data will be sent. The main thing to consider is that the information to be fed into LCD should be of ASCII value. So the LCD module has to change the decimal value to ASCII or std_logic_vector to ASCII respectively and also three control signals such as Enable, Register Select and Read/Write were generated from the FPGA and given to LCD. Then ASCII values were passed to LCD through 8 – bit data lines. Figure 6.13 shows the interfacing of LCD with FPGA and Figure 6.14 shows the timing diagram for LCD.

![Figure 6.13 Interfacing of LCD with FPGA](image)

Figure 6.13 Interfacing of LCD with FPGA
20 * 4 LCD display was selected. The physical connection of LCD with FPAGA is shown in Appendix A1. To display the information in any row, the appropriate row address had been selected and the cursor had been moved accordingly. The cursor address for this LCD is given below.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>81</td>
<td>82</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>93</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>C1</td>
<td>C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>95</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>D5</td>
<td>D6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following steps had been carried out for the initialisation process, and wait for a required period of time after the execution of each step.

1) Send 38h to LCD for function set
2) Send 0Eh for display on and cursor on
3) Send 01h for clear display
4) Send 1Ch for shift the entire display to the right
5) Send 02h for return home
Then, by sending the row address to LCD the required information would display. The required clock for this module had been generated internally with reference to the external clock.

6.12 RESULTS

All the modules such as data acquisition unit, fuzzy unit, PWM unit and LCD unit are integrated and compiled under XST synthesiser. First, the file would get synthesised and generate the HDL synthesis report which gives the information about components such as number of registers, adders/subtractors and comparators etc used. Then the UCF file was created, which contains the port map information of input/output signals to the FPGA. Next the file was subjected to translate, map, place and route. From the mapping report generated, the number of slices, LUTs, IOBs, GCLKs utilised by the project and timing information such as pin delay are listed. Finally, the generated bit file was ready to fuse into Xilinx FPGA for execution. The synthesis, mapping, placement and routing report generated for this project were shown below:

6.12.1 HDL Synthesis Report

<table>
<thead>
<tr>
<th>Top Level Output File Name</th>
<th>controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimisation Criterion</td>
<td>Speed</td>
</tr>
<tr>
<td>Target Technology</td>
<td>spartan2e</td>
</tr>
<tr>
<td>Macro Statistics</td>
<td></td>
</tr>
<tr>
<td># Registers</td>
<td>161</td>
</tr>
<tr>
<td>31-bit register</td>
<td>1</td>
</tr>
<tr>
<td>7-bit register</td>
<td>1</td>
</tr>
<tr>
<td>32-bit register</td>
<td>11</td>
</tr>
<tr>
<td>1-bit register</td>
<td>148</td>
</tr>
<tr>
<td># Adders/Subtractors</td>
<td>17</td>
</tr>
</tbody>
</table>
31-bit adder : 1
18-bit adder : 1
32-bit subtractor : 7
32-bit adder : 7
7-bit adder : 1

# Multipliers : 1
32x12-bit multiplier : 1

# Comparators : 58
32-bit comparator greater : 55
32-bit comparator less : 3

Design Statistics
# IOs : 59

### 6.12.2 Timing Summary

Speed Grade : -6
Minimum period : 18.726 ns (Max Frequency: 53.402MHz)
Minimum input arrival time before clock : 18.452 ns
Maximum output required time after clock: 9.914 ns
Maximum combinational path delay : 9.871 ns

### 6.12.3 Mapping Report

Design Summary
Number of Slices : 1,513 out of 3,072 49 %
Total Number Slice Registers : 483 out of 6,144 7 %
Number used as Flip Flops : 444
Number used as Latches : 39
Total Number 4 input LUTs : 2,648 out of 6,144 43 %
Number used as LUTs: 2,388
Number used as a route-thru: 260
Number of bonded IOBs: 58 out of 142 40 %
Number of GCLKs: 1 out of 4 25 %
Total equivalent gate count for design: 24,585

6.12.4 Place and Route

The Delay Summary Report:
The Score for this design is: 294
The Number of signals not completely routed for this design is: 0
The Average Connection Delay for this design is: 1.779 ns
The Maximum Pin Delay is: 9.450 ns
The Average Connection Delay on the 10 Worst Nets is: 5.808 ns

6.12.5 Timing summary:

Timing errors: 0
Score: 0
Constraints cover 1138305152 paths, 2711 nets, and 8596 connections
Design statistics:
Minimum period: 28.323 ns (Maximum frequency: 35.307 MHz)
Maximum net delay: 0.707 ns

The motor was tested for various terrains and by altering gear conditions respectively. Initially, the system was in the neutral gear, smooth terrain type, with the available battery status, no acceleration and brake. To start the system, 1st gear was chosen and the motor ran at the minimum speed then the gear was changed for increasing the speed. Also, the vehicle could switch over from one terrain to another terrain by setting the terrain thumb
wheel switch position appropriately. Likewise, the gear switching was also carried through out moving the switch position. The module was checked for various input conditions. Some of the cases were listed below.

**CASE 1**

Input Conditions:
- Condition of terrain - smooth
- State of charge in battery - high
- Acceleration - nil
- Brake force applied - nil
- Gear position - 1st and
- Speed of the vehicle - very less

for the above input parameters, the fuzzy module performed the fuzzy operation and the output duty cycle was obtained. The output was 10 % and pulse width modulation signal generated was given to the motor as shown in Figure 6.15.

![Figure 6.15 PWM signal for 10 % duty cycle](image)

**CASE 2**

Input Conditions:
- Condition of terrain - smooth
- State of charge in battery - high
- Acceleration - nil
- Brake force applied - medium
- Gear position - 4th and
- Speed of the vehicle - high
for the above input parameters, the output duty cycle was 50 % and pulse width modulation signal generated for this output was given to the motor as shown in Figure 6.16.

![Figure 6.16 PWM signal for 50 % duty cycle](image1)

CASE 3

Input Conditions:
- Condition of terrain - rough
- State of charge in battery - normal
- Acceleration - high
- Brake force applied - nil
- Gear position - 4th
- Speed of the vehicle - high

the output duty cycle for this case was 80 % and pwm signal generated for this output was given to the motor as shown in Figure 6.17.

![Figure 6.17 PWM signal for 80 % duty cycle](image2)
CASE 4

Input Conditions:
- Condition of terrain - downhill
- State of charge in battery - normal
- Acceleration - nil
- Brake force applied - nil
- Gear position - 3rd and
- Speed of the vehicle - medium

For the above input conditions, the output duty cycle was 25% and PWM signal generated for this output was given to the motor as shown in Figure 6.18.

![Figure 6.18 PWM signal for 25 % duty cycle](image)

CASE 5

Input Conditions:
- Condition of terrain - smooth
- State of charge in battery - high
- Acceleration - nil
- Brake force applied - nil
- Gear position - 4th and
- Speed of the vehicle - high

Then the output duty cycle was 100% and pulse width modulation signal generated for this output was given to the motor. The feedback from the
motor was measured using a Cathode Ray Oscilloscope (CRO) as shown in Figure 6.19. From the figure, it can be inferred that it took 28 ms to complete one revolution. Now in this condition, the motor was running at a speed of 2120 rpm.

![Figure 6.19 pulses from the motor for 100 % duty cycle input](image)

**Figure 6.19 pulses from the motor for 100 % duty cycle input**

**CASE 6**

Input Conditions:
- Condition of terrain - smooth
- State of charge in battery - high
- Acceleration - nil
- Brake force applied - nil
- Gear position - 4th and
- Speed of the vehicle - high

the output duty cycle was 65 % and pulse width modulation signal generated for this output was given to the motor. The feedback from the motor was measured using cathode ray oscilloscope (CRO) as shown in Figure 6.20.
From the Figure, it was inferred that it took 41.4ms to complete one revolution. Now in this condition, the motor was running at the speed of 1450 rpm.

![Figure 6.20 pulses from the motor for 65 % duty cycle input](image)

By using this controller, the motor was controlled to go at the maximum speed of 2120 rpm for smooth terrain, 1450 rpm for rough terrain, 200 rpm for uphill terrain and 500 rpm for downhill terrain. Table 6.5 shows the experimental results of the DC motor.

**Table 6.5 Experimental Results for the DC motor**

<table>
<thead>
<tr>
<th>Duty cycle (%)</th>
<th>Motor in RPM</th>
<th>Period in milliseconds</th>
<th>Terrain</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 %</td>
<td>2120</td>
<td>28</td>
<td>Smooth</td>
</tr>
<tr>
<td>70 %</td>
<td>1450</td>
<td>39</td>
<td>Rough</td>
</tr>
<tr>
<td>50 %</td>
<td>200</td>
<td>70</td>
<td>Uphill</td>
</tr>
<tr>
<td>30 %</td>
<td>500</td>
<td>80</td>
<td>Downhill</td>
</tr>
</tbody>
</table>

The realisation of single chip has been done for the electric vehicle, which is the future trend in the automobile industry.