Implementation of Space Vector Pulse Width Modulation on System on Programmable Chip

by

Vincent Luong

A project presented to Ryerson University in a requirement for the degree of

Master of Engineering in the Electrical and Computer Engineering program

Toronto, Ontario, Canada, 2010

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

Declaration

I hereby declare that I am the sole author of this thesis report.

I authorize Ryerson University to make copies of this thesis by any means, in whole or in part, for the purpose of scholarly research.

Signature:

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Richard Cheung for his guidance. His tremendous patient and constant encouragement has helped me accomplishing what I thought to be beyond my ability.

I am grateful to my wife and sons for their love and support. Because of my study, many weekend activities were cancelled.

I give thanks to my Lord Jesus for everything I have. To Him all glory belongs!

Abstract

For years, DSP has been the dominant tool in implementing gate switching for power inverter. It is a powerful and reliable technology in carrying out complex switching schemes. DSP is still expensive due to its intensive use of resource in chip fabrication. There is no flexibility in making change on hardware once a DSP chip is selected. It is also time consuming in a design development because the learning curve of the DSP is stiff. Recently, a new approach to the problem has emerged. It is called embedded system design. Basically, it is a FPGA system combined with a RISC type microprocessor. This is a robust combination that allows users to pick and choose any functional peripheral devices only as needed. Once the complete hardware platform is decided upon, the circuit is configured and down loaded to a chip. Software codes are then written to run the application. The hardware system is reconfigurable. Designers can always go back to change the hardware with ease in order to improve the performance and to meet the target cost.

This is an attempt to utilize the embedded system design also called System on Programmable Chip (SOPC) to perform Space Vector Modulation (SVM) gate switching strategy. The Altera Nios II IDE tool is selected for this task.

Table of Content

Lists of Figures

Chapter I

Introduction

1.1 Objective

The focus of this report is on the implementation of space vector modulation for induction load or motor using System on Programmable Chip (SOPC) method. The advantages of the induction motor over the DC motor will mentioned, two main pulse width modulation schemes will be discussed, different ways to apply the scheme will be examined before heading to the proposal.

1.2 Back Ground

The first induction motor was invented by Nicola Tesla in 1888. To this day, it still remains the most rugged, reliable, less expensive to build and the least maintenance required machine. In comparison with a DC motor, the DC one relies on built in permanent magnets for flux generation in order for the machine to run. The induction motor produces its flux through the supply voltages. Hence, the induction machine is lighter in term of weigh and output power ratio. In addition, the rotor of the DC machine brings about a turning force as the current carrying conductor is placed inside the magnetic field. This conductor is connected to the power source through some kind brushing mechanism. Over times, this connector gets corroded due to wear and tear that affects the performance of the motor. Therefore, periodic maintenance must be carried out to ensure its efficiency [1]. For induction motor, the supply voltages in the stator give

off a rotating field. This moving field induces voltage in the rotor. In the case of squirrel cage induction motor where the rotor windings are short circuited, the flowing current creates a rotor flux. Consequently, the rotor flux reacts with the stator flux to yield torque to the rotor. The whole process requires no external connection to the rotor. Therefore, the motor is rugged and needs no maintenance [2].

The advantages of the induction motor out weigh that of the DC motor. It was, however, not commonly used in the early days because the motor has a fixed speed which depends on the frequency of the voltage source. Recent advancement in power electronic has given the induction motor a face lift in its applications which can be found in robotics, machine tools and hybrid vehicles [3].

The general scheme for controlling the speed of the induction motor involves an AC to DC rectifying process and a DC to AC transformation procedure. A typical unit takes AC power source, rectifies it into a steady DC voltage, and then converts it back to a desired frequency AC waveform. This DC to AC sub unit is called an inverter, which is how the motor control inverter has its name. There are two main methodologies to convert DC to AC for induction motors. One method is called Sine Pulse Width Modulation (SPWM) or Voltage Frequency (VF) control. The other is Space Vector Modulation (SVM).

The principal of the sinusoidal PWM scheme is that a control sine wave at the desired frequency is compared with the triangular wave at constant amplitude. The frequency of the triangular wave set the inverter switching frequency [4]. Traditionally, VF control was popular mainly due to its uncomplicated implementation and the least on chip computation requirement. This algorithm has some major drawbacks. First of all, it

is not able to fully utilize the supply DC voltage. Less than 90% of DC supply voltage is used. Secondly, it has substantial high Total Harmonic Distortion (THD) which results in heat generated in switching devices and larger heat sink is needed. The THD is worsening when the frequency modulation index defined by the ratio of carrier and modulating frequency is not an integer. It brings about a so called non characteristic harmonics whose frequency is not a multiplication of the fundamental frequency. This poses a challenge in designing a proper filter to eliminate the unwanted frequency [5]. This method also imply an inefficient way to use the memory since three 120 degree phase shifted sine tables have to be stored in the chip memory in order to generate the necessary sine waves [6].

The SVM algorithm is based on the fact that for a balanced three phase system, the sum of vectors representing 3-phase line to neutral AC power supply sine waves is zero. Hence, these vectors can be expressed as a single space reference vector in α and β plane [7]. By controlling the amplitude and the rotating speed of this vector, the motor's torque and speed can be regulated. The SVM is an advanced and computation intensive technique which gives 15% more voltage output compared to the SPWM method [8]. It generates less THD. Now a day, this technique is widely adopted in motor speed regulation.

The main challenge of the SVM approach lays in the calculation of the angle between two vectors. The trigonometric function such as sine and cosine to define angle can be solved conventionally by interpolating from a detailed table of the known values. The end result may lack of the resolution needed for some applications. Modern day computers resolve that problem mainly by Taylor series with very high precision. That

requires large amount of hardware resources and long computing time on the computers part [9]. How fast the angles are worked out determines how well the performance of SVM. In fact, estimating the angles becomes the bottle neck of the entire process.

1.3 Methodologies

Over the years, with the progress of microelectronics and its cost reduction have made the SVM feasible in real time. There are many ways to carry out the scheme of which in general can be categorized into two three main streams: software, hardware and the combination of both. With many choices available, today designers have to juggle between the performance, cost effective and the ease of implementation in adopting the best method for their application.

Until recently, software stream has been realized by high speed microprocessor such as RISC machines and DSP. This approach is very flexible and able to accomplish complex algorithms, but the disadvantages are long development time, poor portability of programming codes and more CPU resources. DSP still is comparatively expensive.

RISC stands for Reduced Instruction Set Computer. The idea behind the creation of RISC processor is based on the observation that only a small percentage of a processor's basic instructions are used in majority of cases. Therefore, it is better to build processors where those instructions are handled efficiently on simplified and faster hardware [10]. The instructions of RISC engine are simple and fixed in size so that the implementation for this faster hardware called pipelined machines can be made easy. The concept the pipeline is borrowed from assembly line in manufacturing process. Instead of putting all the time and resource into processing one computer code, the entire operation

can be divided into smaller different stages where codes are managed continuously. The end result is that, after the initial delay, commands are completed in every cycle.

DSP are designed for extensive enumeration purposes. It is also a RISC machine with Harvard architecture. This architecture utilizes segregated, independent program and data memories with different buses so that the two memories can be accessed at the same time. It allows an instruction and an operand to be fetched from memory in one clock cycle [11]. In order to make quick calculation, all DSP processors equip with at least one fast hardware multiplier. This fast multiplier is able to bear result in a single clock cycle. Some advance DSP employ parallel execution technique called parallelism where second or more multipliers are needed. Floating point data formats are generally used in DSP algorithms and hence complex hardware are developed for this format. Since much hardware circuits are used in its construction, DSP remains the most expensive computation machine. Also, dedicated hardware requires specialized instruction set, programmers can only optimize codes in assembly level instead of high level languages as C and C++. This may prolong system development time [12].

Recent arising of the field programmable gate array or FPGA technology, has given designers a new arsenal in tackling comprehensive enumerating challenge. This is the hardware approach for SVPWM. FPGA is an array of many logic blocks that are linked by horizontal and vertical wiring channels. A logic block consists of many logic elements which embodies look up tables (LUT), a programmable flip-flop with a synchronous enable, a carry chain, and a cascade chain. The FPGA chip is configured by a specific programming code called hardware description language (HDL). A HDL programmer can "write" a hardware and down load it onto a FPGA chip which performs

as a complex combinational function or even like an application specific integrated circuit (ASIC). The hardware is reconfigurable and therefore complicated circuits can be realized and tested in very short time. In addition, instead of executing instructions sequentially as in software approach, FPGA is able to carry out different tasks simultaneously [13]. There is no need for external wiring; the system is reliable. As a result, sine and cosine function modules, PWM output waveform module [14] and even an entire SVPWM core [15] are developed. This has greatly enhanced the SVPWM performance where the sampling rate can reach as high as 40 KHz [16]. There are a few disadvantages; a new programmable language is to be learned; proficiency in finite state machine (FSM) design and handshaking protocol is a must in order to have all these hardware modules working together in correct timely manner.

1.4 Proposal

The quest for faster sampling time is propelled by Field Oriented Control (FOC) in servo motors. This is because in the heart of FOC, sit the SVPWM. To achieve dynamic performance, FOC demands high sampling rate. However, not all SVPWM applications have to be high speed. In the case of high power medium voltage (MV) drive, the switching frequency of the inverter is usually below 1 KHz [17]. The reason is that the switching devices can't handle over that frequency range. This gives rise to the proposal of this report as high sampling rate is not required. A hardware and software combined method called embedded system design is introduced.

An embedded system is a digital system with at least one processor that implements a hardware function which is part of an over all system. The Embedded

processors are typically RISC machines which are used the same way as microcontrollers. The main difference is that embedded system allows more flexibility and design customization due to its reconfigurable capability [18]. Designers are able to choose the hardware functions and merge together with microprocessors then down load the system into a FPGA chip. Assembler or high level language such as C and C++ is written to program the processor. The entire process necessitates a software tool called System on a Programmable chip or SOPC. This tool provides graphic user interface (GUI) for system components selection and generates interconnect logic automatically. SOPC outputs HDL files that define all modules of the system and HDL design file that connects all modules together for the purpose of FPGA configuration. Besides that, it offers features to write software codes and to do simulation for the prototype.

The SOPC used to realize SVPWM in this report belongs to Altera's Quatus software package. The sampling rate of this SVPWM method depends on how sophisticate the hardware components are selected or created. A process will be shown later that this method is easy to implement.

This paper is organized as the following. The next chapter demonstrates the principle of vector modulation technique. Chapter III simulates SVPWM using Mathlab and its results are served as benchmark in comparing with the real implementing results. Chapter IV executes SVPWM by SOPC mean. Chapter V concludes the report.

Chapter II

Principle of Space Vector Pulse Width Modulation

The theory presented here is base on the text book; High *power converter and AC drive* by Professor Bin Wu [19].

Voltage source inverter (VSI) is the unit that converts a DC voltage to a three-phase AC voltage with varying amplitude and frequency. The inverter is made up of six group of active switches, $S_1 \sim S_6$. Each switch has a free-wheeling diode coupled in parallel. A simplified schematic for a voltage source converter for a typical high power medium voltage is shown in Fig.2-1. Base on the DC operating voltage of the inverter, each switching group may composes two or more IGBT or GCT connected in series.

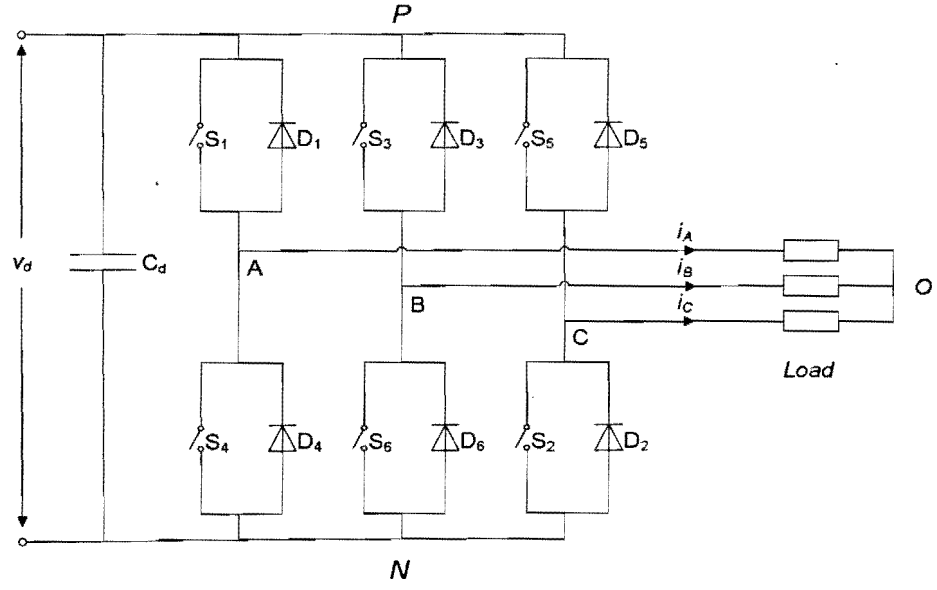


Figure 2-1 Simplified schematic of an inverter

2.1 Switching state

Before introducing the SVPWM theory, some of the terminologies are familiarized in order to have a better understanding. The operating status of the switches in each inverter branch is represented by switching states. **P** is denoted as upper switch being turned on and its inverter terminal carrying positive voltage (Vd). **O** stands for the lower switch being turned off and its inverter terminal having zero voltage. Figure 2.1-1 provides the definition of the switching state.

Switching State	**LEG A**			**LEG B**			**LEG C**		
	S_1	S_4	υ_{AN}	S_3	S_6	υ_{BN}	S_5	S_2	υ_{CN}
P	ON	OFF	V_d	ON	OFF	V_d	ON	OFF	V_d
O	OFF	ON	0	OFF	ON	0	OFF	ON	0

Figure 2.1-1 Switching State Definition

There are eight combinations of the switching states for the inverter. For instants, combination **[P O O]** corresponds to the conduction of S_1, S_6, and S_2 in legs A, B, C accordingly. Within the eight switching states, **[P P P]** and **[O O O]** are the zero states, the rest are active states. Figure 2.1-2 lists the definition of all combinations.

Space Vector		Switching State	On-state Switch	Vector Definition
Zero Vector	$\vec{V}_0$	[PPP]	S_1, S_3, S_5	$\vec{V}_0 = 0$
		[OOO]	S_4, S_6, S_2	
Active Vector	$\vec{V}_1$	[POO]	S_1, S_6, S_2	$\vec{V}_1 = \frac{2}{3} V_d e^{j0}$
	$\vec{V}_2$	[PPO]	S_1, S_3, S_2	$\vec{V}_2 = \frac{2}{3} V_d e^{j\frac{\pi}{3}}$
	$\vec{V}_3$	[OPO]	S_4, S_3, S_2	$\vec{V}_3 = \frac{2}{3} V_d e^{j\frac{2\pi}{3}}$
	$\vec{V}_4$	[OPP]	S_4, S_3, S_5	$\vec{V}_4 = \frac{2}{3} V_d e^{j\frac{3\pi}{3}}$
	$\vec{V}_5$	[OOP]	S_4, S_6, S_5	$\vec{V}_5 = \frac{2}{3} V_d e^{j\frac{4\pi}{3}}$
	$\vec{V}_6$	[POP]	S_1, S_6, S_5	$\vec{V}_6 = \frac{2}{3} V_d e^{j\frac{5\pi}{3}}$

Figure 2.1-2 All Switching State Combination

2.2 Space Vector PWM

This theory is better demonstrated in graphical way. The active and zero switching states can be represented as active and zero state vectors. A space vector diagram is shown in figure 2.2-1.

The six vectors $\vec{V}_1$ to $\vec{V}_6$ form a regular hexagon with six equal sectors (I to VI). The zero state vector, $\vec{V}_0$, lies in the centre of the hexagon. The reference voltage vector $\vec{V}_{ref}$ rotates within the hexagon at a certain speed.

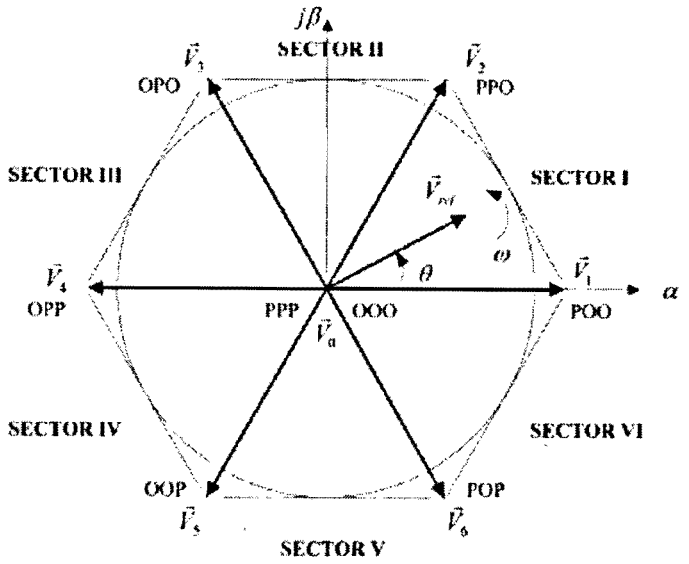


Figure 2.2-1 Space Vector Diagram

The relationship between the space vector and the switching states can be derived as the following. From the inverter shown in figure 2-1, with the assumption that the inverter having three-phase balanced load, this expression is valid,

$$V_{AO}(t) + V_{BO}(t) + V_{CO}(t) = 0 \qquad (2.2\text{-}1)$$

The state of $V_{CO}(t)$ is redundant, since it can be defined by $V_{AO}(t)$ and $V_{BO}(t)$. Also, it is convenient to transform the three-phase variable to two-phase variable.

$$\begin{bmatrix} V_\alpha(t) \\ V_\beta(t) \end{bmatrix} = \frac{2}{3}\begin{bmatrix} 1 & \frac{(-1)}{2} & \frac{(-1)}{2} \\ 0 & \frac{\sqrt{3}}{2} & \frac{(-\sqrt{3})}{2} \end{bmatrix} \begin{bmatrix} V_{AO}(t) \\ V_{BO}(t) \\ V_{CO}(t) \end{bmatrix} \qquad (2.2\text{-}2)$$

Therefore, the space vector can be expressed in α – β plane two-phase voltages.

$$\vec{V}(t) = V_\alpha(t) + jV_\beta(t) \qquad (2.2\text{-}3)$$

Substituting (2.2-1) to (2.2-3),

$$\vec{V}(t) = \frac{2}{3}\left[V_{AO}(t)e^{j0} + V_{BO}(t)e^{j\frac{2\pi}{3}} + V_{CO}(t)e^{j\frac{4\pi}{3}} \right] \qquad (2.2\text{-}4)$$

where $e^{jx} = \cos(x) + j\sin(x)$ and x = 0, 2π/3 and 4π/3.

Considering the Active switching state **[P O O]** for example, the generated inverter phase voltages are

$$V_{AO}(t) = 2\frac{V_d}{3} \;,\; V_{BO}(t) = \frac{-V_d}{3} \;,\; V_{CO}(t) = \frac{-V_d}{3} \qquad (2.2\text{-}5)$$

The corresponding space vector, $\vec{V_1}$ can be found by placing (2.2-5) to (2.2-4)

$$\vec{V}_1 = \frac{2}{3}V_d e^{j0} \qquad (2.2\text{-}6)$$

In general, all six space vectors can be represented as

$$\vec{V_k} = \frac{2}{3}V_d e^{j(k-1)\frac{\pi}{3}} \quad \text{where k = 1, 2, 3... 6}$$

The zero-space vector $\vec{V_0}$ has two switching states. These two states are redundant to each other. The redundant state is used to minimize the switching frequency and eliminate the even harmonics. The zero and active space vectors are stationary in space and hence they are called stationary vectors.

The reference vector $\vec{V}_{ref}$, in figure circulates in space with an angular velocity at

$$\omega = 2\pi f_1 \qquad (2.2\text{-}8)$$

f_1 is the fundamental frequency of the inverter output voltage. The angle between $\vec{V}_{ref}$ and α- axis in the α–β plane is obtained by

$$\theta(t) = \int_0^t \omega(t)dt + \theta(0) \qquad (2.2\text{-}9)$$

$\vec{V}_{ref}$ can be approximated by the three stationary vectors for any given length and position. These stationary vectors in turn determine the switching states of the inverter. As $\vec{V}_{ref}$ rotates one revolution in space, the inverter completes one cycle over time. The generated output voltage frequency is proportional to the angular velocity of $\vec{V}_{ref}$ and the output voltage magnitude is corresponding to the magnitude of $\vec{V}_{ref}$.

2.3 Dwell Time Calculation

Since the spinning $\vec{V}_{ref}$ can be composed by three stationary vectors, the length of these stationary vectors depends on the length of time assigned to them during the sampling period T_s. It is called the dwell time and is used to define a moving vector position at that time instant. The dwell time calculation applies the 'voltage second balancing' principle. The principle states that the product of the reference voltage $\vec{V}_{ref}$ and the sampling period T_s equal to the sum of the voltage multiplied by time interval of the selected space vectors. The sampling period T_s is always set small enough such that the reference vector is almost constant during T_s interval. Taking $\vec{V}_{ref}$ inside the sector I for example, the voltage balancing equation is

$$\vec{V}_{ref}T_s = \vec{V}_1 T_a + \vec{V}_2 T_b + \vec{V}_0 T_o \qquad (2.3\text{-}1)$$

and $\quad T_s = T_a + T_b + T_0$

T_a, T_b and T_0 are the dwell times for $\vec{V}_1$, $\vec{V}_2$ and $\vec{V}_0$ correspondingly.

The space vector in 2.3-1 can be shown as

$$\vec{V}_{ref} = \vec{V}_{ref}e^{j\theta}, \vec{V}_1 = 2\frac{V_d}{3}, \vec{V}_2 = \frac{2}{3}V_d e^{j\frac{\pi}{3}}, \vec{V}_0 = 0 \qquad (2.3\text{-}2)$$

Substituting 2.3-2 to 2.3-1 and then separate the resulting equation into real and imaginary components in the α–β plane,

Re: $\quad \vec{V}_{ref}(\cos\theta)T_s = \frac{2}{3}V_d T_a + \frac{1}{3}V_d T_b$

Im: $\quad \vec{V}_{ref}(\sin\theta)T_s = \frac{1}{\sqrt{3}}V_d T_b \qquad (2.3\text{-}3)$

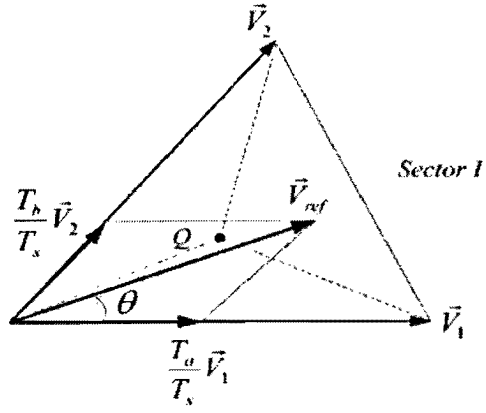


Figure 2.3-1 Dwell Time and Voltage Vectors

Solving 2.3-3 with $T_s = T_a + T_b + T_o$ yields

$$T_a = \sqrt{3}T_s \frac{V_{ref}}{V_d} \sin(\frac{\pi}{3} - \theta)$$

$$T_b = \sqrt{3}T_s \frac{V_{ref}}{V_d} \sin(\theta) \qquad (2.3\text{-}4)$$

$$T_o = T_s - T_a - T_b \qquad \text{for} \quad 0 \le \theta \le \frac{\pi}{3}$$

Figure 2.3-1 illustrates the relation between dwell time and voltage vectors.

The equations in 2.3-4 are derived when $\vec{V}_{ref}$ falls in sector I. For other sectors, $\vec{V}_{ref}$ can be located by this general angle expression

$$\theta' = \theta - (k-1)\frac{\pi}{3} \qquad \text{for} \quad 0 \le \theta < \frac{\pi}{3}$$

where k=1, 2, 3,...,6 corresponding to sectors I, II,...,VI

2.4 Modulation Index

Modulation index dictates the magnitude of the inverter's output. Equation 2.3-4 can be used to express the modulation index m_a.

$$T_a = \sqrt{3}T_s \frac{V_{ref}}{V_d} \sin(\frac{\pi}{3} - \theta)$$

$$T_b = \sqrt{3}T_s \frac{V_{ref}}{V_d} \sin(\theta)$$

$$T_o = T_s - T_a - T_b \qquad \text{for} \quad 0 \le \theta \le \frac{\pi}{3}$$

where $$m_a = \sqrt{3}\frac{V_{ref}}{V_d} \qquad (2.4\text{-}1)$$

The maximum amplitude of the reference vector $V_{ref,\max}$ lies on the vertices of the hexagon outside the circle as shown in figure 2.2-1.

$$V_{ref,\max} = \frac{2}{3} V_d * \frac{\sqrt{3}}{2} = \frac{V_d}{\sqrt{3}} \tag{2.4-2}$$

Substituting $V_{ref,\max}$ into m_a in 2.4-1 gives

$$m_a = 1 \tag{2.4-3}$$

Therefore, the modulation is within the range of

$$0 \le m \le 1$$

and the peak fundamental voltage produced by SMV is

$$\hat{V}_{\max,SMV} = V_{ref,\max} = \frac{V_d}{\sqrt{3}} \tag{2.4-4}$$

2.5 Switching Sequence

The switching sequence for any $\vec{V}_{ref}$ should follow these two guide lines in order to minimize the switching frequency.

1- Only one branch of the inverter switches change states, one being switched on and other being switched off, when one switching segment alters from one to the next.

2- As $\vec{V}_{ref}$ travels from one sector to the next, none or minimum number of switches change states. The figure 2.5-1 is an example of the switching sequence utilizing the above guide lines. The figure shows a seven segment sequence and inverter output voltage for $\vec{V}_{ref}$ in sector I. It is observed that

- Sum of the different dwell times equals to the sampling time.

- The transition from **[O O O]** to **[P O O]** involves only two switches which are S_1 (ON) and S_2 (OFF). The requirement 1 is satisfied.

- The redundant switching states of $\vec{V}_o$ are used to minimize the number of switching.

- Each switch in the inverter is turned on and off once in one sampling period T_s

Hence, the switching frequency f_{sw} is the same as the sampling frequency f_{sp} .

$$f_{sw} = f_{sp} = \frac{1}{T_s}$$

Figure 2.5-1 graphs the essence of the switching sequence rule above in sector I.

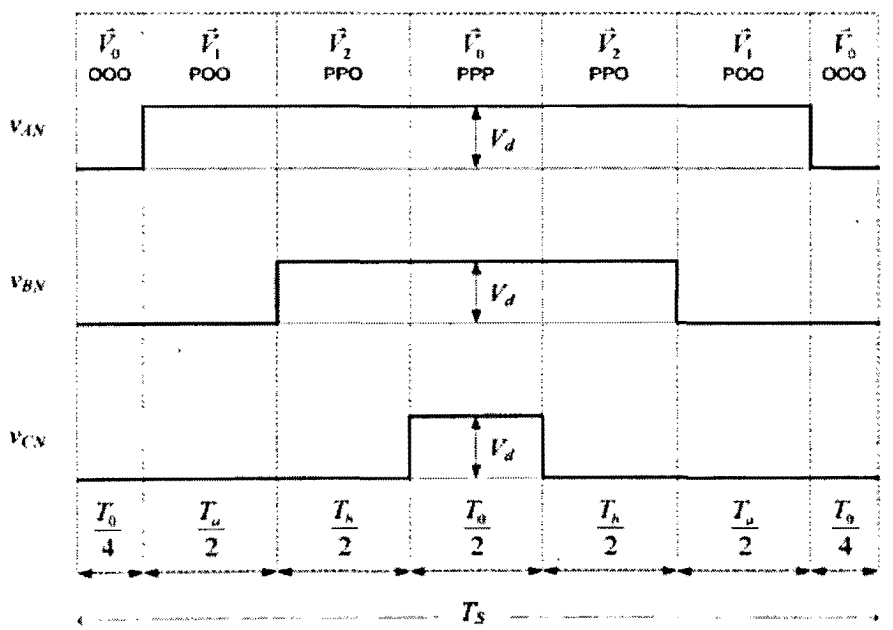


Figure 2.5-1 Switching Sequence in Sector I

Figure 2.5-2 summarizes the seven segment switching sequences of $\vec{V}_{ref}$ in all six sectors.

Sector	Switching Sequence						
I	$\vec{V}_0$	$\vec{V}_1$	$\vec{V}_2$	$\vec{V}_0$	$\vec{V}_2$	$\vec{V}_1$	$\vec{V}_0$
	OOO	POO	PPO	PPP	PPO	POO	OOO
II	$\vec{V}_0$	$\vec{V}_3$	$\vec{V}_2$	$\vec{V}_0$	$\vec{V}_2$	$\vec{V}_3$	$\vec{V}_0$
	OOO	OPO	PPO	PPP	PPO	OPO	OOO
III	$\vec{V}_0$	$\vec{V}_3$	$\vec{V}_4$	$\vec{V}_0$	$\vec{V}_4$	$\vec{V}_3$	$\vec{V}_0$
	OOO	OPO	OPP	PPP	OPP	OPO	OOO
IV	$\vec{V}_0$	$\vec{V}_5$	$\vec{V}_4$	$\vec{V}_0$	$\vec{V}_4$	$\vec{V}_5$	$\vec{V}_0$
	OOO	OOP	OPP	PPP	OPP	OOP	OOO
V	$\vec{V}_0$	$\vec{V}_5$	$\vec{V}_6$	$\vec{V}_0$	$\vec{V}_6$	$\vec{V}_5$	$\vec{V}_0$
	OOO	OOP	POP	PPP	POP	OOP	OOO
VI	$\vec{V}_0$	$\vec{V}_1$	$\vec{V}_6$	$\vec{V}_0$	$\vec{V}_6$	$\vec{V}_1$	$\vec{V}_0$
	OOO	POO	POP	PPP	POP	POO	OOO

Figure 2.5-2 Switching Sequence in All Six Sectors

Chapter III

SVPWM Matlab Simulation

It is always a good practice to simulate a design before actually carrying it out. This allows designers to see the feasibilities and the performances of their would-be products preventing costly later alternation and time delay. The software package used for this project is Matlab Simmulink.

The design specification is coming from an actual project.

3.1 Specifications

- Inverter configuration: Three phase two-level inverter
- Rated inverter output voltage: 4160V (rms fundamental line-to-line voltage)
- Rated inverter output power : 1MVA (three phase)
- DC link voltage: constant, ripple free
- Inverter load : Three-phase balanced RL load with a lagging power factor of 0.9

The inverter base parameters

- $V_B = V_R / 3 = 4160 / \sqrt{3} = 2401.8V$
- $I_B = S_R / 3V_B = (1 * 10^6) / (3 * 2401.8) = 138.8A$
- $Z_B = V_B / I_B = 2401.8 / 138.8 = 17.3\Omega$

- $\omega_B = 2\pi f_1 = 377.0 rad/s$
- Load: $R = 17.3\cos(\theta) = 15.6\Omega$

$\because \omega L = 17.3\sin(\theta) = 7.54 \qquad \therefore L = 20mH$

V_d can be defined by modulation index when $m_a = 1$.

$$\frac{V_{AB1}}{V_d} = 0.612$$

$$\therefore V_d = \frac{V_{AB1}}{0.012} = \frac{4160}{0.612} = 6797.4V$$

3.2 Simulating scheme

According to the theory of SVPWM, the position of the moving space vector at a time instant is determined by an angle formed between the vector itself and horizontal axis. The dwell time is calculated based on the angle found at the moment. The switching pattern is decided upon by the sector where the $\vec{V}_{ref}$ is located. For strategy wise, it is easier to find the sector right after the angle becomes known. The flow of the simulation logic is outlined in the figure 3.2-1.

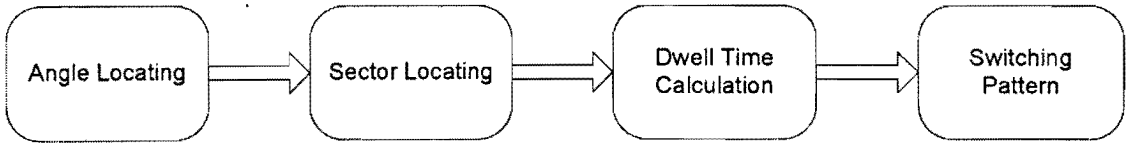
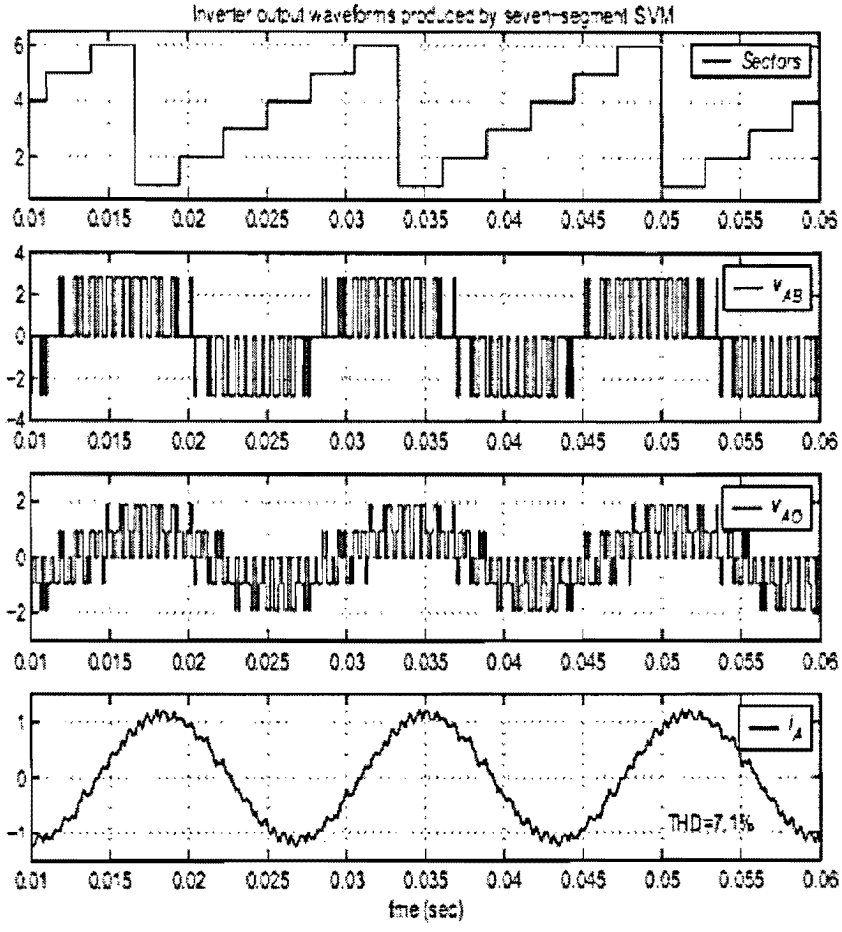


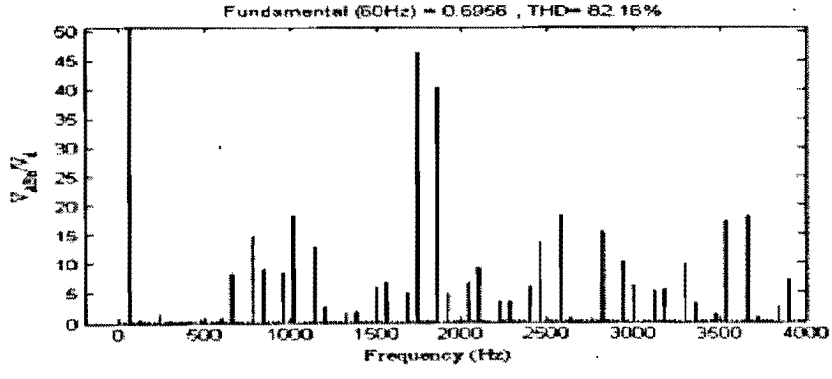
Figure 3.2-1 Simulation Logic Flow

3.3 Result

The result of the simulation is categorized by two types of graphs: one is the graphs of voltages and current waveforms; the other is the graph of harmonic spectrum.

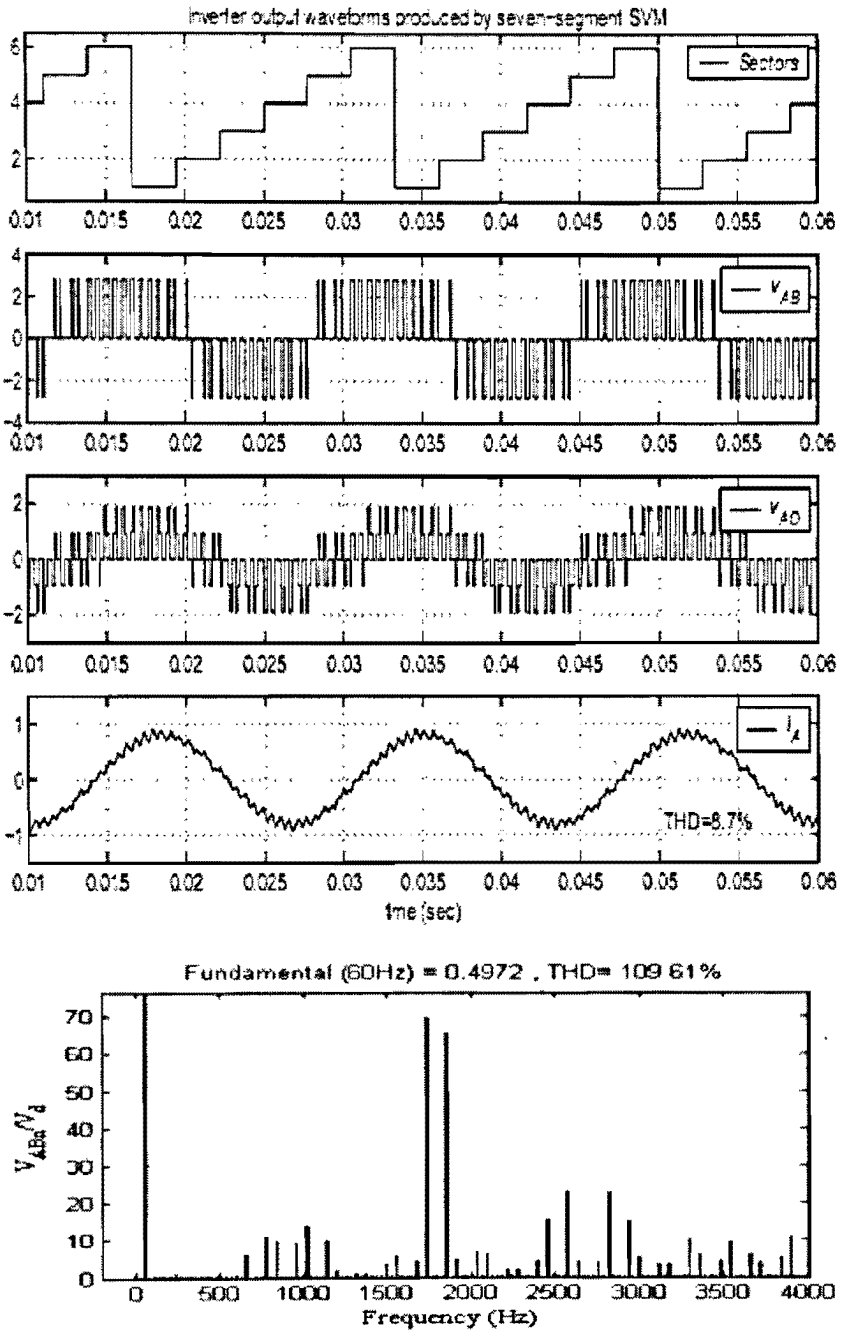


(a) Waveforms



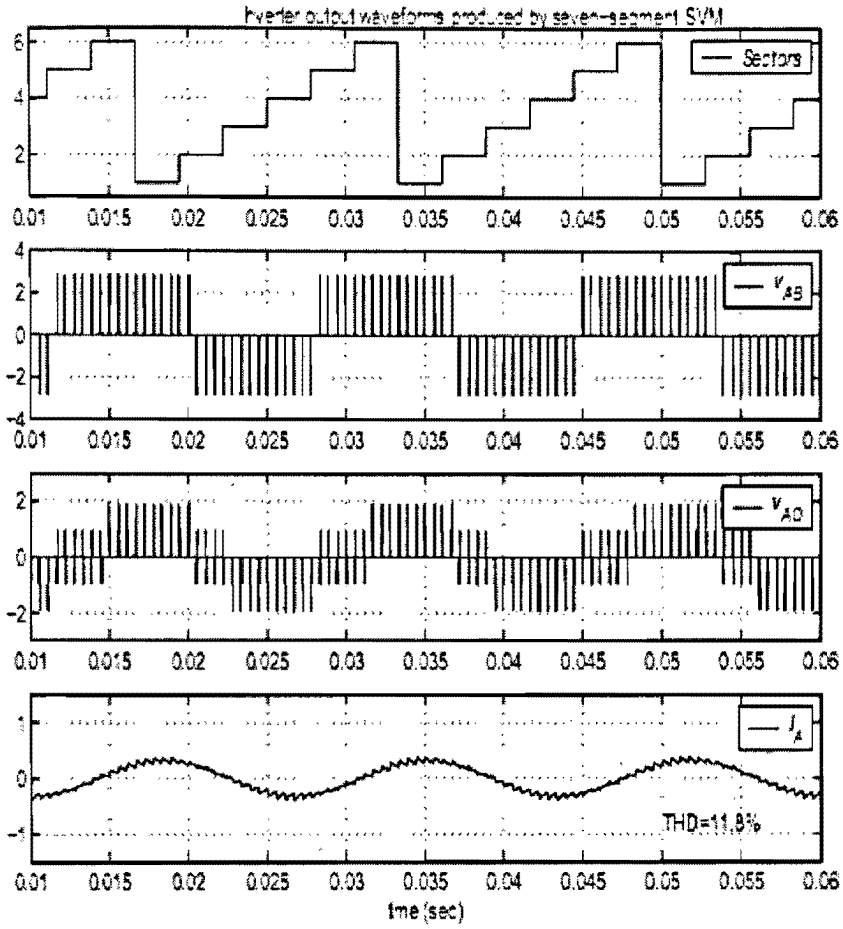
(b) Harmonic spectrum

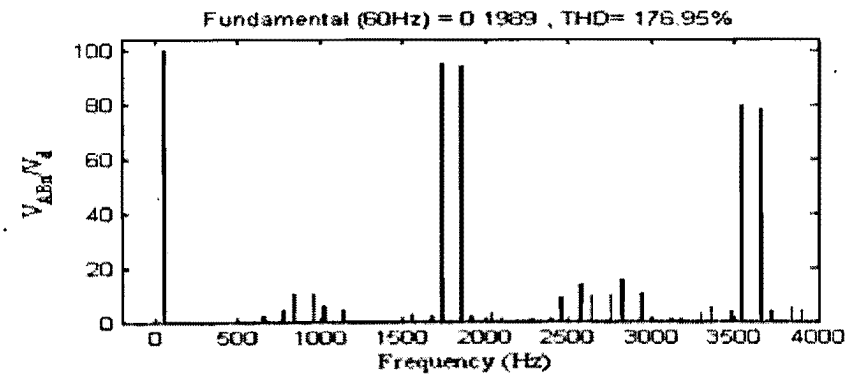
Figure 3.3-1: $f = 60Hz$, $f_{sw} = 900Hz$, $m_a = 0.7$, and $T_s = 1.11ms$



(b) Harmonic spectrum

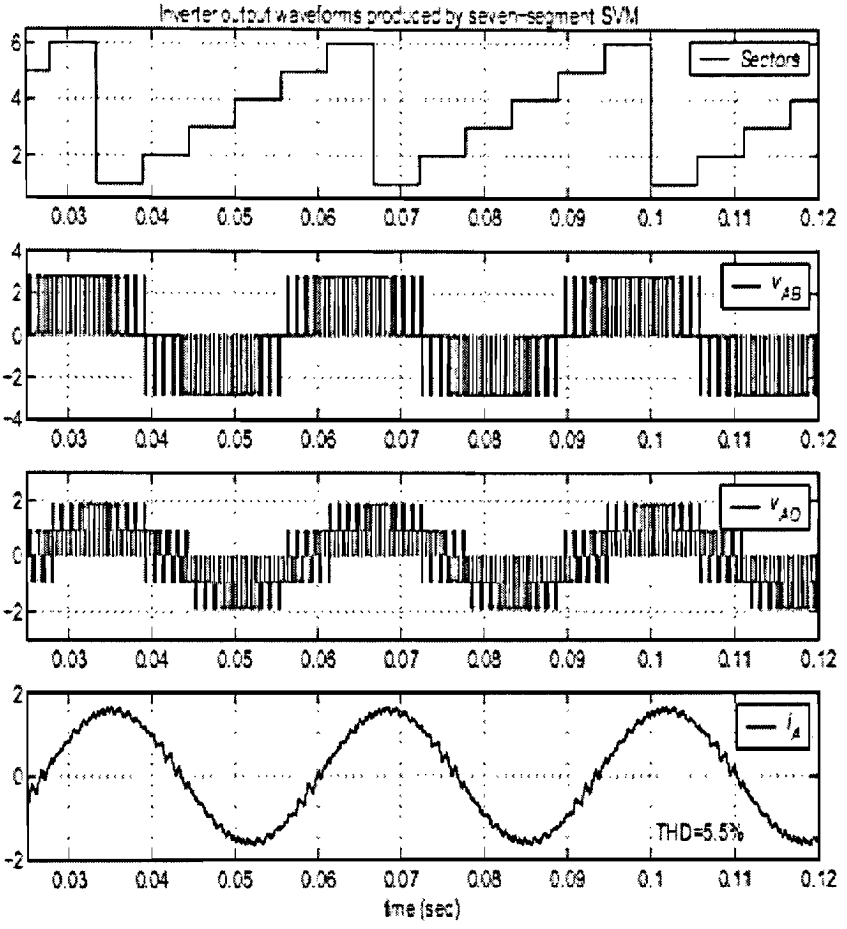
Figure 3.3-2: $f = 60Hz$, $f_{sw} = 900Hz$, $m_a = 0.5$, and $T_s = 1.11ms$



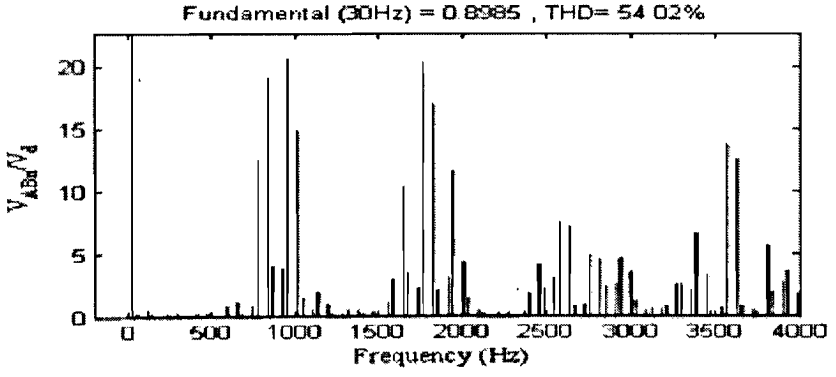


(b) Harmonic spectrum

Figure 3.3-3: $f = 60Hz$, $f_{sw} = 900Hz$, $m_a = 0.2$, and $T_s = 1.11ms$

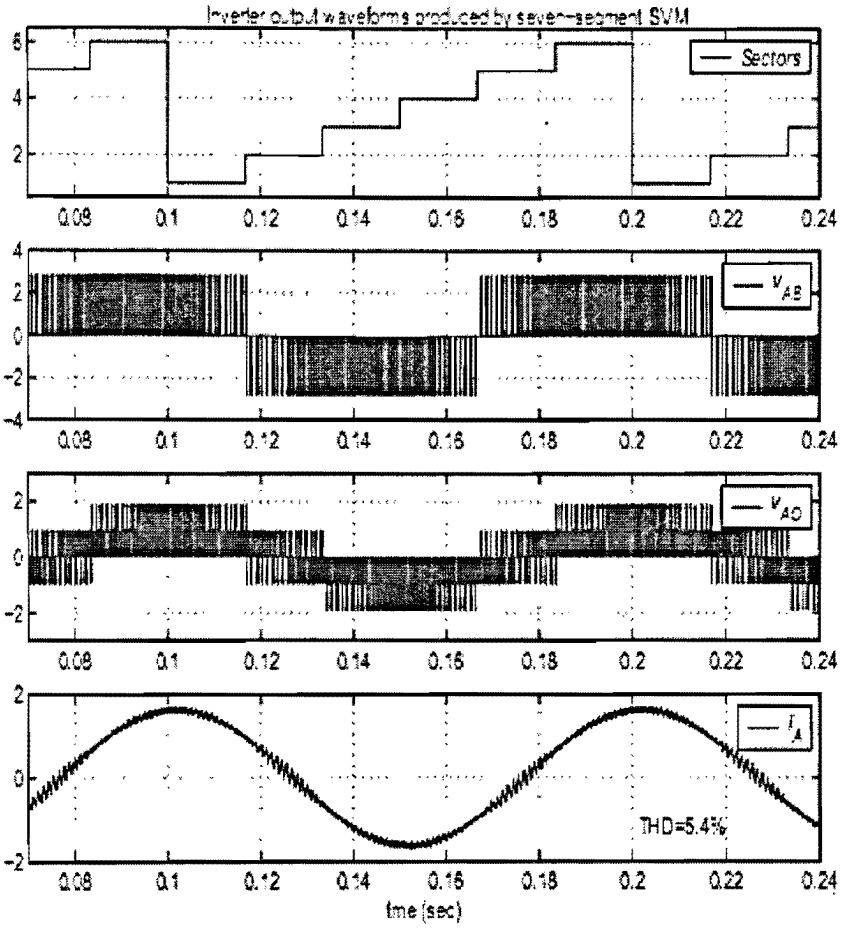


(a) Waveforms

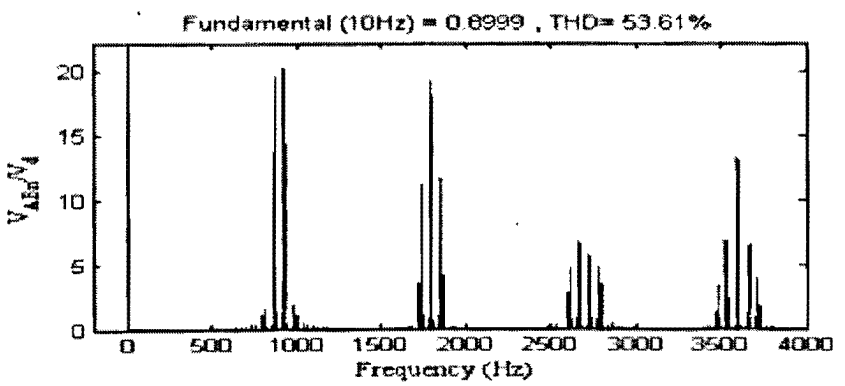


(b) Harmonic spectrum

Figure 3.3-4: $f = 30Hz$, $f_{sw} = 900Hz$, $m_a = 0.9$, and $T_s = 1.11ms$



(a) Waveforms



(b) Harmonic spectrum

Figure 3.3-5: $f = 10Hz$, $f_{sw} = 900Hz$, $m_a = 0.9$, and $T_s = 1.11ms$

Figure 3.3-1, -2, -3, -4 and -5 show simulations for the following cases A, B, C, D and E respectively. The THD result of the load current i_A at 1000Hz range is included in Figure 3.3-1.

Simulation	f_1 (Hz)	f_{sw} (Hz)	m_a	T_s (ms)	Load Current, i_A, THD%
A	60	900	0.7	1.11	7.1
B	60	900	0.5	1.11	8.7
C	60	900	0.2	1.11	11.8
D	30	900	0.9	1.11	5.5
E	10	900	0.9	1.11	5.4

Figure 3.3-1 Table of Simulation output and Its Load Current THD

3.4 Observation

The simulation result does prove that the theory works. With sampling rate remains the same through out, lower the rotating frequency yields better THD. The same is true for higher the modulation index. Higher sampling rate reduces the effect of THD. Most of the distortions occur at and over 900Hz range and are far away from the fundamental frequency. They can be easily filtered out.

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

Chapter IV

SVPWM Realization by SOPC

The best way to get familiar with Altera SOPC builder is to follow its tutorial on *Nios II hardware development* [20]. This tutorial serves as a guideline on designing a Nios II hardware system and then composing a software program to run on the system. Since the objective of this exercise is different from the thesis project, changes will be made as needed to fit the project's goal.

4.1 Target Board

The circuit board used here is the Altera DE2 Development and Education Board with Altera Cyclone II (2C35) FPGA chip on it. It has a number of switches, LEDs, LCD, and 7-segment displays either for input to the processor or visual indicator of processor activities for small project. It includes RS232 and PS2 for communication. For video and sound experiment, it provides connectors for microphone, line-in, 24-bit audio CODEC line out, video in and VGA out. It also equips USB 2.0 connectivity for host and device. For more challenging experiments, there are SRAM, SDRAM, and Flash memory chip. The board runs at frequency of 50 MHz. Figure 4.1-1 is the picture of the target board.

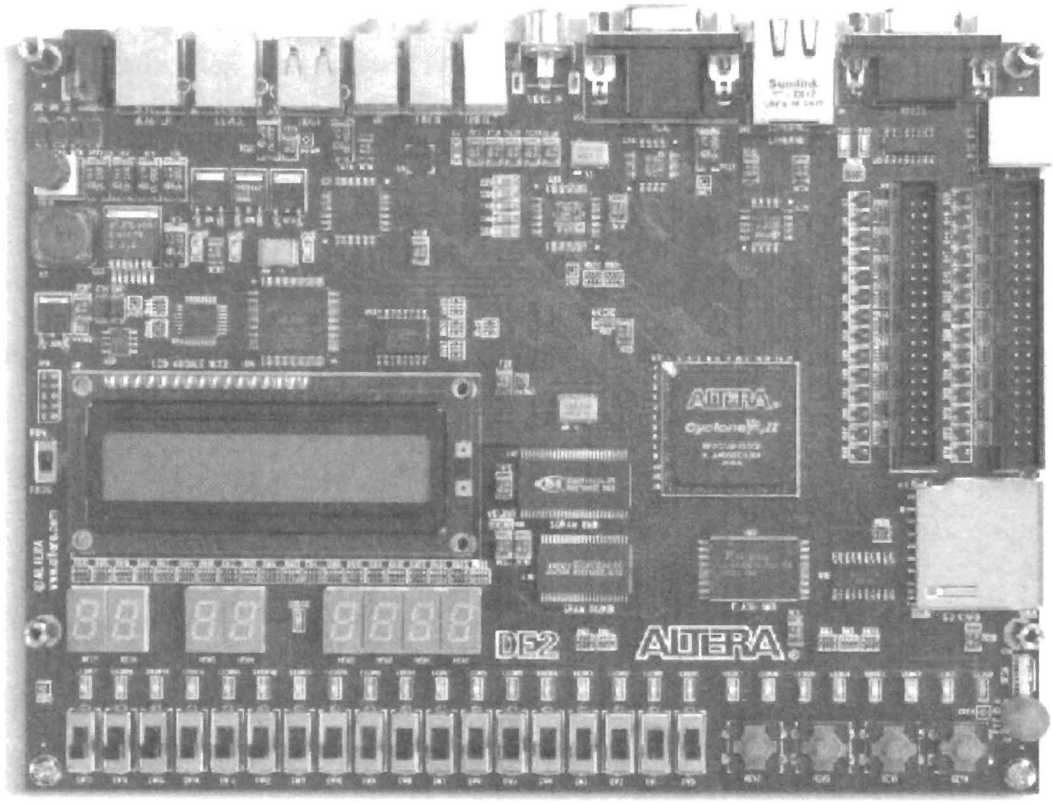


Figure 4.1-1 Target Board

4.2 Embedded Processor

The Nios II processor is a general-purpose RISC machine. It comprises of a full 32-bit instruction set, data path, and address space; 32 general purpose registers; 32 external interrupt sources; single instruction 32x32 multiplier and divider resulting in 32-bit data; floating point instruction for single precision floating point manipulation.

4.3 System Development flow

The embedded system development flow consists of three forms of development: hardware design, software design and system design which includes both hardware and software. For a simpler system, all of the above procedures can be performed by one person. For a more complex system, it may request several teams of engineers to cover all those steps. Figure 4.3-1 suggests the designing flow.

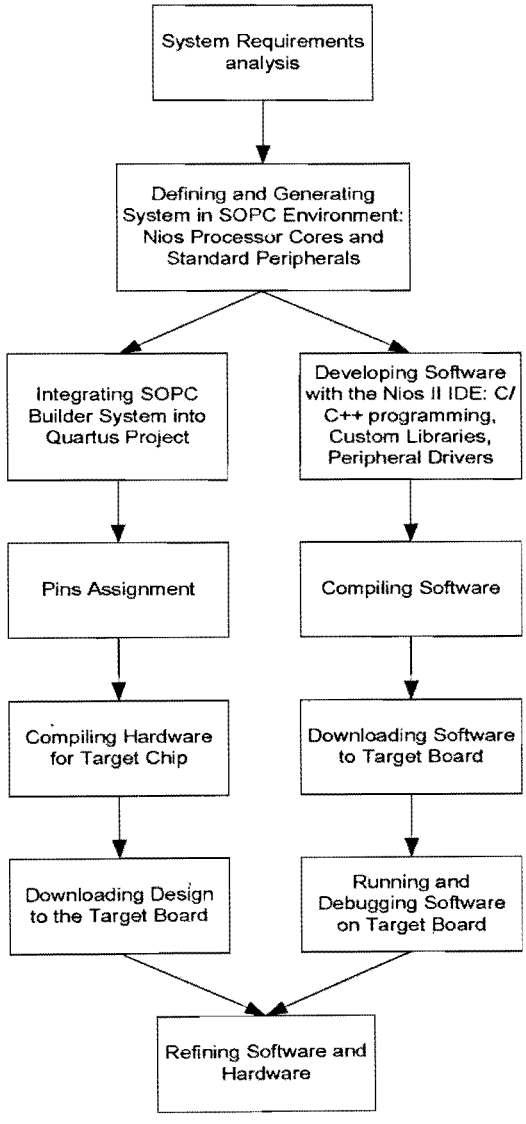


Figure 4.3-1 Embedded System Design Flow

Following the figure 4.3-1 suggestion, the requirement of the project has to be clearly defined. For SVM control scheme, calculation of the switching time and switching patterns are the predominant issues.

The switching time calculation is determined by the sine trigonometry function of an angle. This is done by mean of power series and hence a hardware multiplier is incorporated inside the processor core.

$$Sin(X) = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \ldots \qquad \text{(Equation 4.3.1)}$$

The easiest way to achieve the switching pattern is to utilize interrupt timers. This is because the switching sequences change after a certain time interval according to the figure 2.5.1.

4.3.1 Hardware Development tasks

Based on these observations, a concrete system design is finalized. This project requires a fast Nios II processor with a hardware multiplier for the power series calculation; seven interrupt timers for seven changes in switching sequences. The next step will be using SOPC builder tool to specify the processor core, memory, and other peripheral devices such as timers. The builder tool will automatically generate the interconnected logic to integrate the component in the system. In general, there are two classes of peripherals. One is the standard type such as timers, SDRAM, general purpose I/O, even a secondary processor etc provided by Nios II Embedded Design Suite. The other is the custom type like custom-made logics or the ready-made intellectual property (IP). There are two advantages; hardware implementation is faster than software; the

processor is free to execute other task in parallel while the peripheral operates on data. Ideally, a hardware sine function for SVM should be created to handle this kind of computation as hardware approach is always faster than software approach. This incurs higher development cost and longer development time. For the pilot run, the fastest and cheapest technique is used. Figure 4.3.1-1 shows the preliminary design for the SVM.

Continuing the process, pins assignment for the I/O signals can be arranged with design constraints using the Altera SOPC builder. Finally, the entire project is compiled to produce an SRAM Object File to configure the FPGA. This file is then downloaded to the FPGA on the target board through the cable called Altera USB-Blaster.

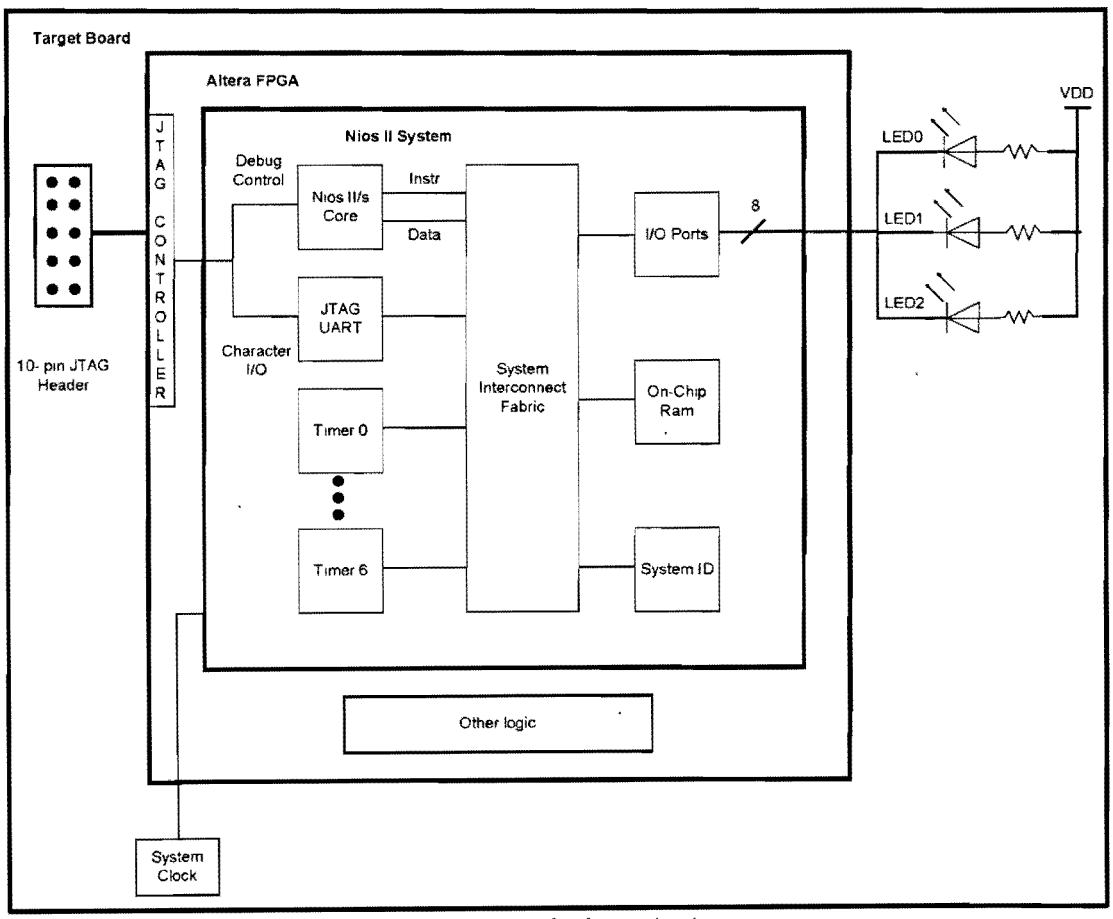


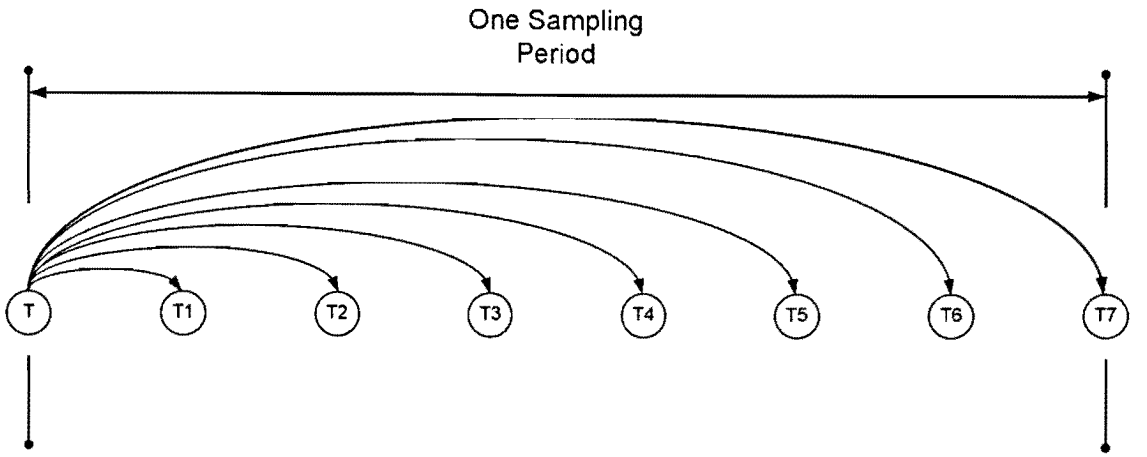
Figure 4.3.1-1 Preliminary design Block

In this layout, JTAG UART is used for downloading both for hardware and software and for debugging. General I/O ports PIO are used for switches with LEDs for visual effect. Several timer interrupts are required for setting up the switching patterns. Other logic is anticipated when the goal is not reached and custom made logic might be needed.

4.3.2 Software Development Tasks

With the help of Nios II IDE, software development tasks for Nios II processor system are performed. Designers are able to write high level computer languages like C or C++ codes for the system. To interact with low level hardware components like turning on or off switches, Altera provides components drivers and a hardware abstraction layer (HAL) to facilitate such a task. Once the application program is compiled, it can be downloaded to the target board using the same USB-Blaster cable. The IDE debugger allows users to start and stop the program, step through codes, create break points, and examine variables. This debugger also provides a way to debug the software without the present of the target hardware.

Figure 4.3.2-1 displays the software algorithm for the SVM. T to T_7 represents the timer interrupt interval. In one sampling period, values of T_o, T_a, and T_b are determined for the switching sequences.



$$\frac{T_0}{4} + \frac{T_a}{2} or \frac{T_b}{2} + \frac{T_b}{2} or \frac{T_a}{2} + \frac{T_o}{2} + \frac{T_b}{2} or \frac{T_a}{2} + \frac{T_a}{2} or \frac{T_b}{2} + \frac{T_0}{4}$$

Figure 4.3.2-1 Software Algorithm

4.3.3 Refining Hardware and Software

After running the software on the hardware, it might be that the performance does not meet the requirement. Designers can either go back to the software to make changes in algorithm or go to hardware to reconfigure the peripheral. Reconfigurable peripherals is one of the most distinguish difference between an embedded system and a fixed microcontroller. In this context, reconfigurable means that hardware features can be added or removed on a system-by-system basis to meet performance and price goal.

4.4 Creating the Design

Following are the step by step to implement the design using Nios II Development tool. This procedure is carried out closely adhering to the tutorial.

1. Install the design files for the Nios II Hardware Development Tutorial. After having the files unzipped, they are stored in a director.
2. Start the Quartus II software and open the tutorial project.
3. Create a new SOPC builder system by clicking SOPC builder on the tool menu in the Quartus II software. On the Create New System dialog box, type system name. This project uses the same name as suggested in the tutorial which is *first_nios2_system*. Figure 4.4-1 shows the SOPC builder GUI in the early state.

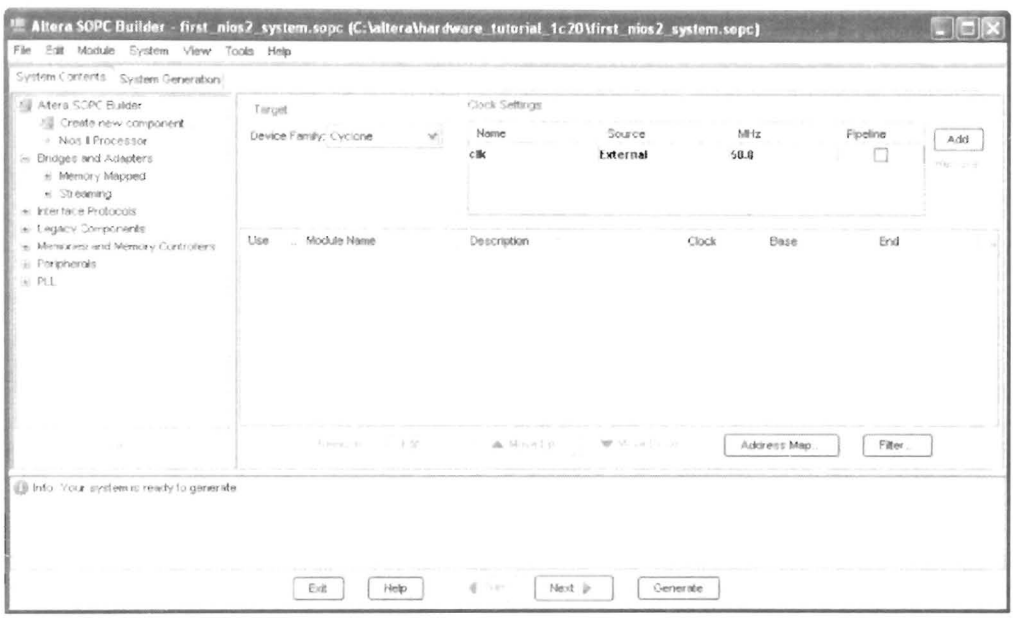


Figure 4.4-1 SOPC Builder GUI

4. Building the system in the SOPC builder. SOPC builder is used to define the hardware characteristics of the of the Nios II system, i.e. which Nios II core is selected, what peripherals to include in the system. Since this step is the most crucial of the whole process, detail contents of GUI are shown and explained. On the SOPC builder, FPGA cyclone chip is chosen, and the clock is set at 50MHz.
    - Processor requires minimum one memory to store data and instructions; the on chip memory selection is first performed. To add

the memory, on the left hand side of system contents tab, under the category **Memories and Memory Controllers/On-Chip**, user high lights the **On-Chip Memory (RAM or ROM)** then clicks **Add.** The On-Chip Memory MegaWizard interface appears. **M4K** block memory is select. Figure 4.4-2 is the On-Chip Memory interface box.

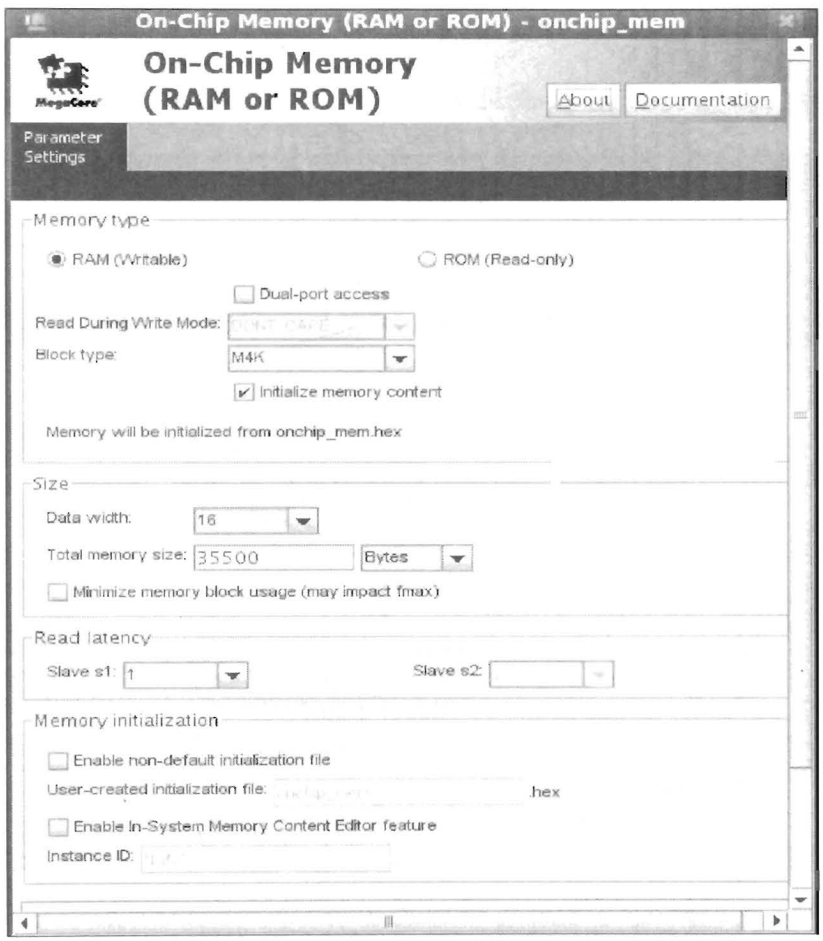


Figure 4.4-2 On-Chip Memory MegaWizard

The total memory size is 35Kbytes. This is because the actual software size is bigger than the default 20Kbytes. Floating point computation needs more memory space as well.

The Nios II core selection and its configuration is done next. There are three classes of Nios II core processor; they are economics, standard and full features. Standard core is picked in this project for the reason that it has built in hardware multiplier and divider. This hardware will accelerate the floating point manipulation. User selects **Nios II Processor** and then clicks on ADD. The Nios II processor MegaWizard appears. Figure 4.4-3 displays the Nios II core page.

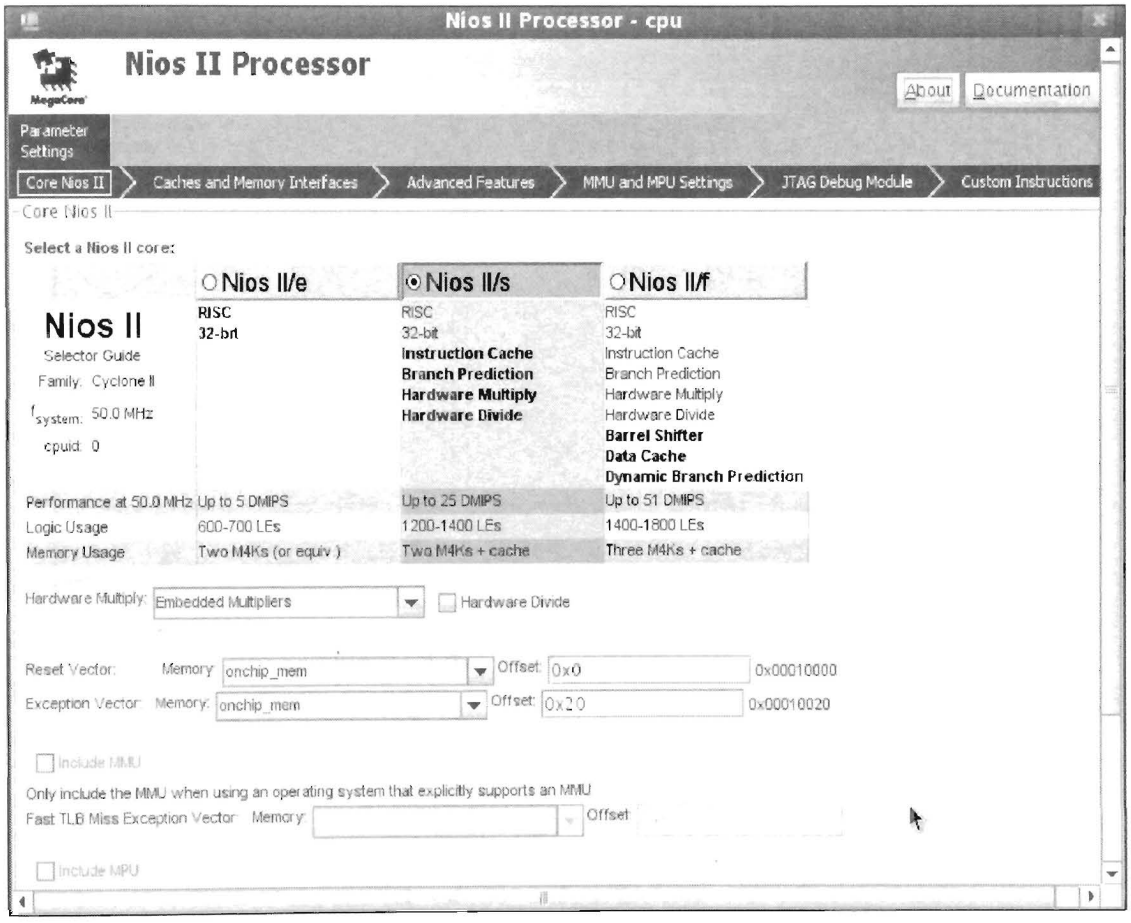


Figure 4.4-3 Nios II Processor Core GUI

Hardware divide is not adopted here since it takes up huge FPGA chip resource. The division will be done by software. Nios II processor provides custom instruction in order to use its hardware multipliers for floating point operation. To activate this feature, user clicks on *Custom Instructions* on the Nios II core GUI. Figure 4.4-4 shows the configuration wizard.

Figure 4.4-4 Floating point GUI

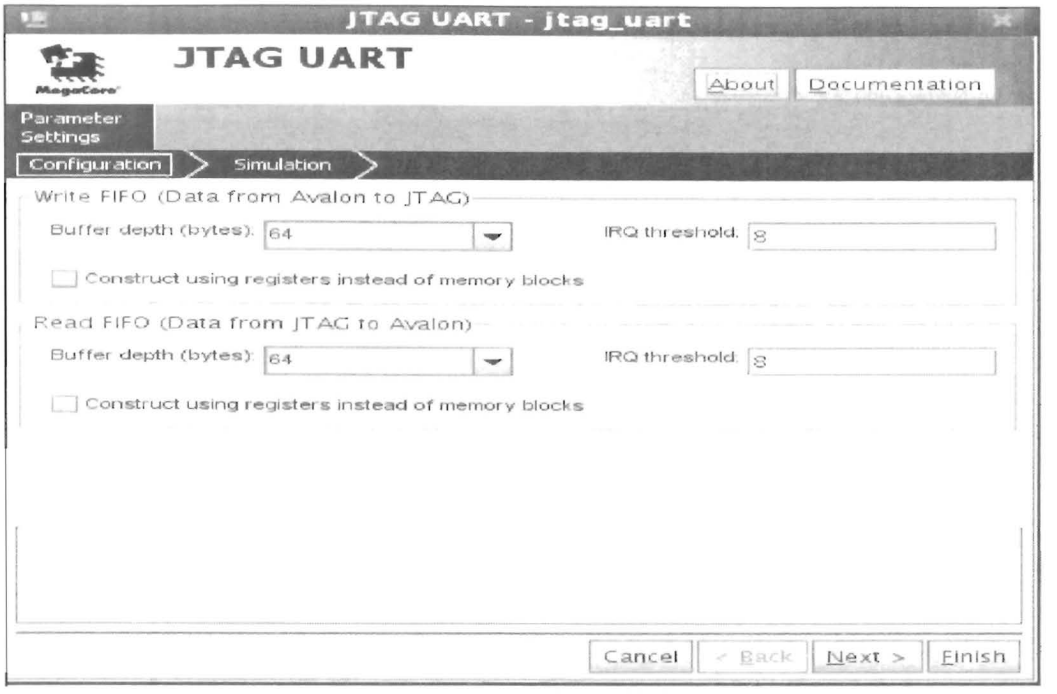


Figure 4.4-5 Serial Communication GUI

- Adding JTAG UART allows for convenient way to communicate with Nios II processor through the USB-Blaster download cable. This is very useful for software debugging purpose. On **Interface protocols/Serial**, user elects **JTAG UART** the clicks **Add.** Figure 4.4-5 indicates the setting.
- To make the programming part straight forward, seven timer interrupts are required here for seven switching pattern changes. Under **Peripheral/Microcontroller Peripheral/**, user clicks **ADD** then the Timer MegaWizard interface appears. Figure 4.4-6 displays the timer interface.

Figure 4.4-7 Interval Timer MegaWizard

Since our SVP sampling period is set at $\frac{1}{900}$ second or 1.11milisecond, the interrupt timing for each switching sequence is in the order of microsecond. With 32 bits counter size, it is more than enough to cover one sampling cycle. The timer hardware option is set at full featured, so that the timer period can be written to, read from, and started/stopped by control bits.

- General purpose I/O port or PIO is used to drive LEDs for visual effect as to confirm the program working properly. They are

actually the switch drivers in SVM scheme. Eight bits are selected.

Under **Peripherals/Microcontroller Peripherals/PIO (Parallel I/O)**, user clicks on **ADD** and the **PIO** MegaWizard appears.

Figure 4.4-8 exhibits the PIO interface menu.

Figure 4.4-8 PIO MegaWizard

- The system ID peripheral helps accidentally downloading other software for different Nios II system. Under **Peripherals/Debug and Performance**, user clicks on **System ID Peripheral** and then clicks on **ADD**. The ID Peripheral MegaWizard interface appears. There is no need to configure the ID option. Figure 4.4-9 shows the pop up interface box.

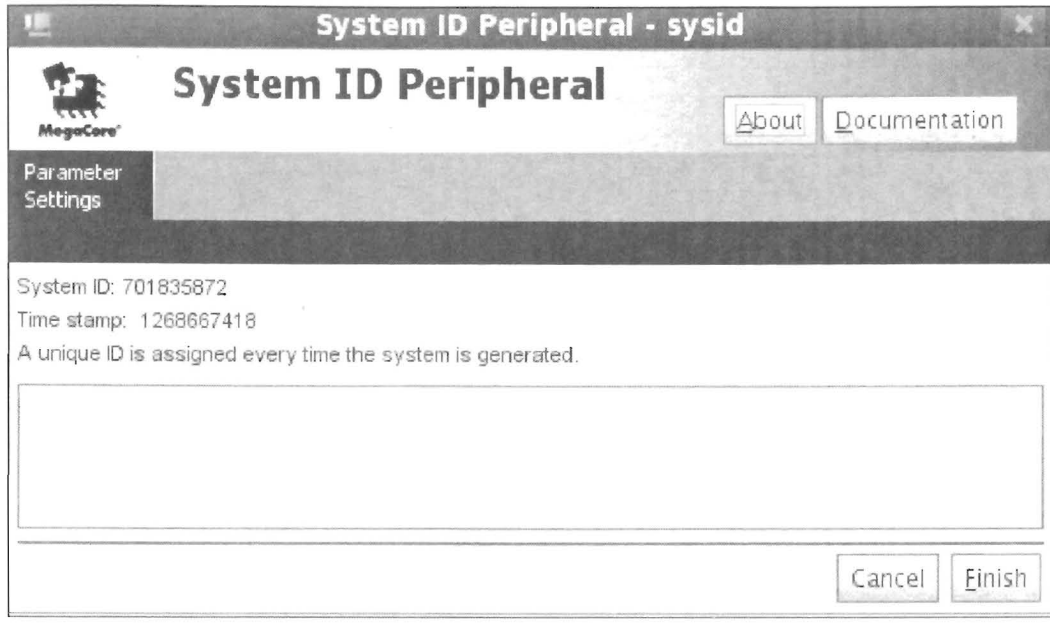


Figure 4.4-9 System ID Peripheral Interface Box

At this point, all the necessary hardware components are in place They need to be specified as to how they interact within the system. The issues can be the base addresses assigned to each component, and designating priorities for timer interrupt requests (IRQ) and the JTAG UART. SOPC builder provides **Auto-Assign Base Address** and **Auto-Assign IRQs** commands to facilitate these assignments. Since the project's timer interrupts do not happen at the same time, they can be allocated in any priority sequences as long as the JTAG UART get the lowest one. Figure 4.4-10 displays the interconnection of the system.

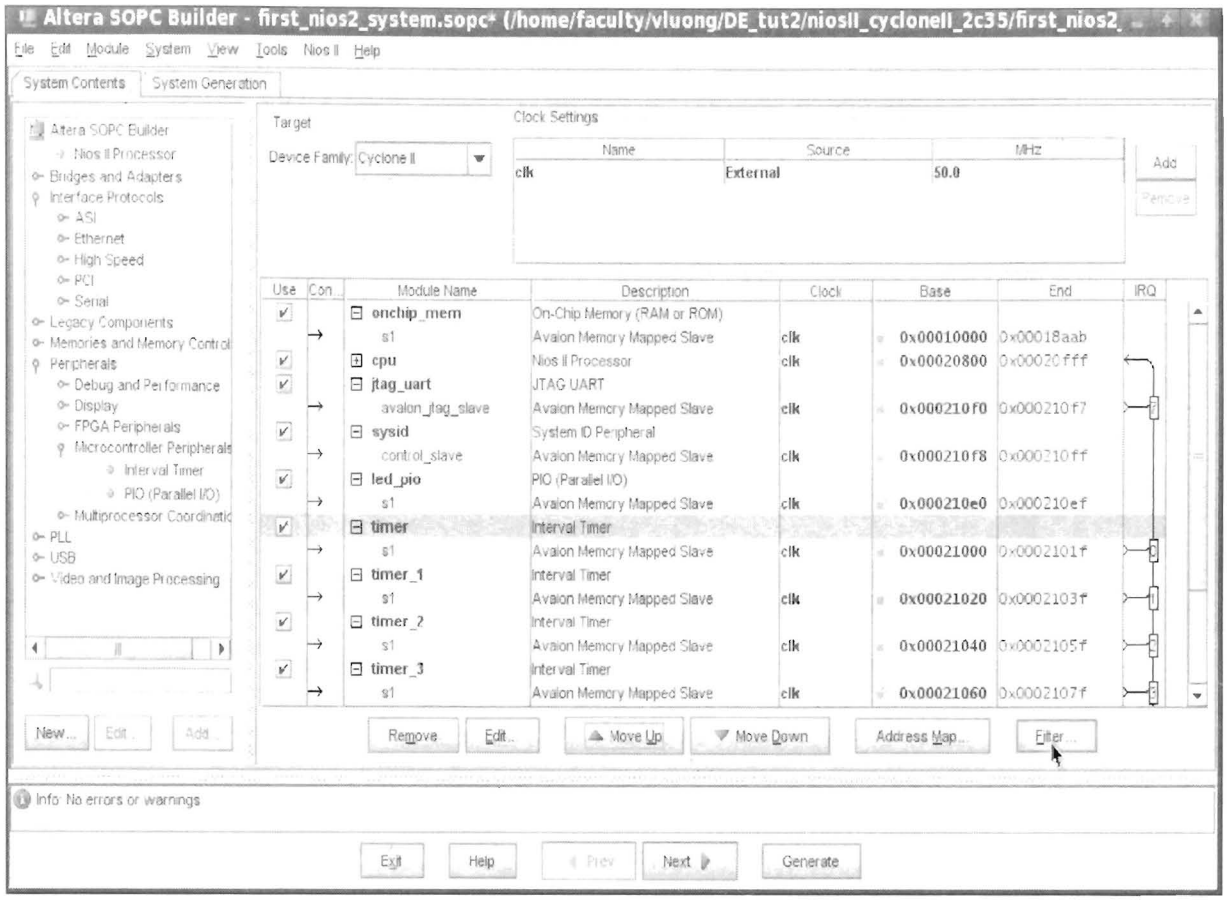


Figure 4.4-10 Contents of Complete System

The hardware can be now generated by clicking on **System Generation** tab and then clicking on **Generate** box.

5 Integrate the SOPC builder system into Quartus II project. This step involves instantiating the SOPC system module, assigning physical FPGA pins, compiling the project and verifying the timing. Figure 4.5-1 and Figure 4.5-2 each display system module and FPGA pins assignment.

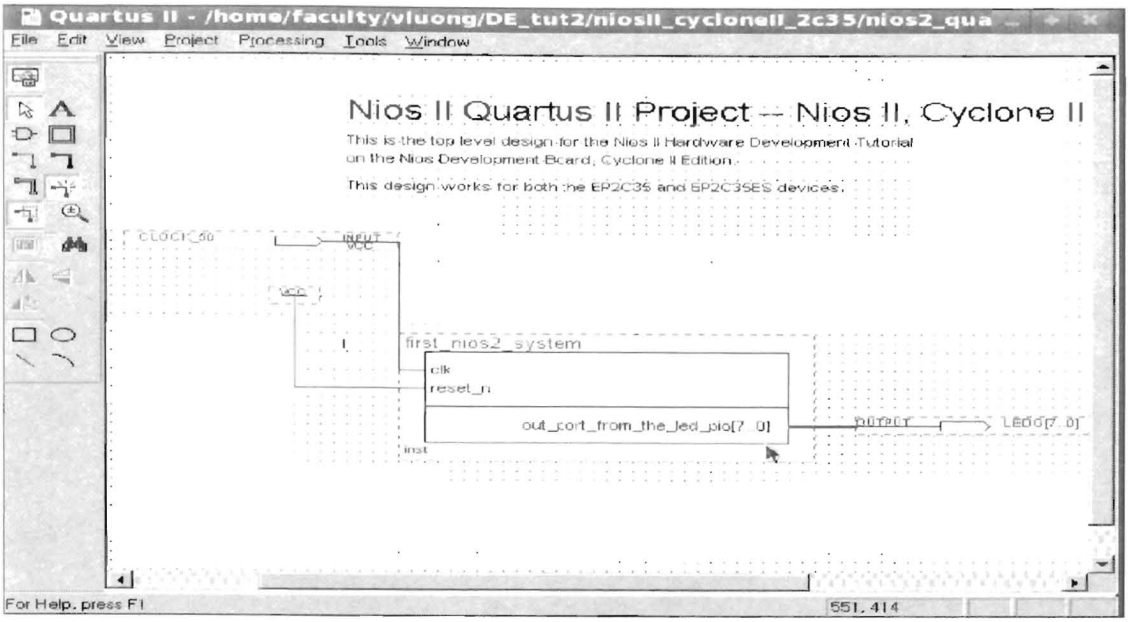


Figure 4.5-1 System Module

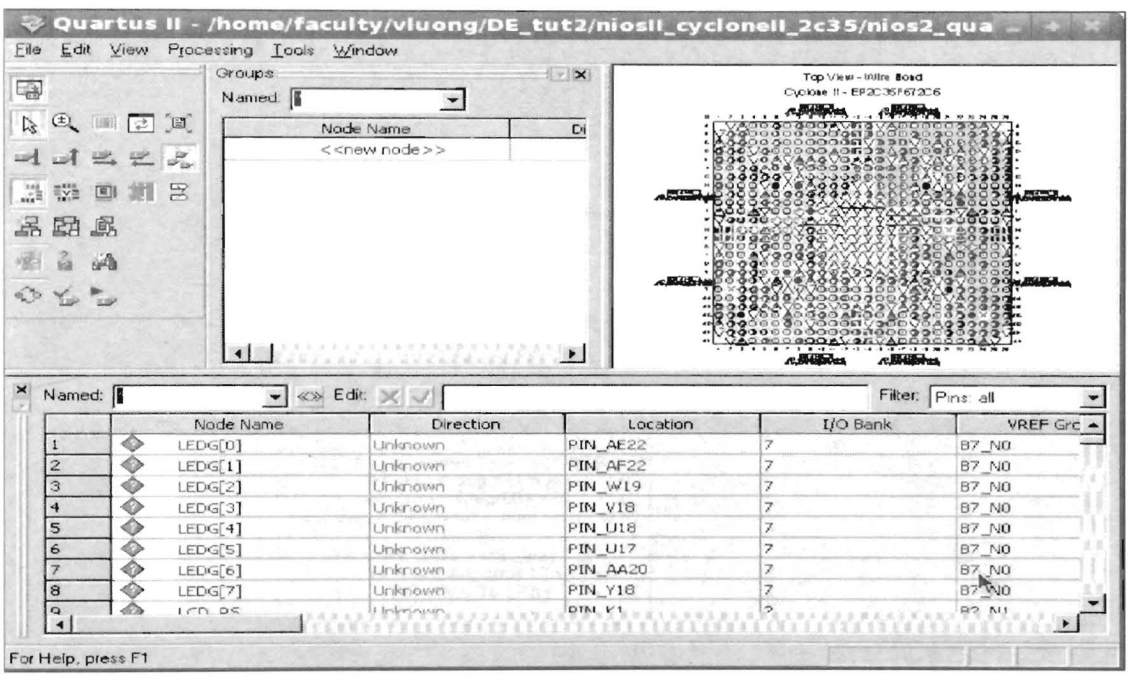


Figure 4.5-2 FPGA pin assignment Wizard

6. Download hardware design to target FPGA. This process requires USB-Blaster and following the **Programmer** GUI procedure.

7. Develop software using Nios II IDE. Figure 4.7-1 depicts the logic flow of the main software. Figure 4.7-2 shows some of the subroutines and interrupt subroutines.

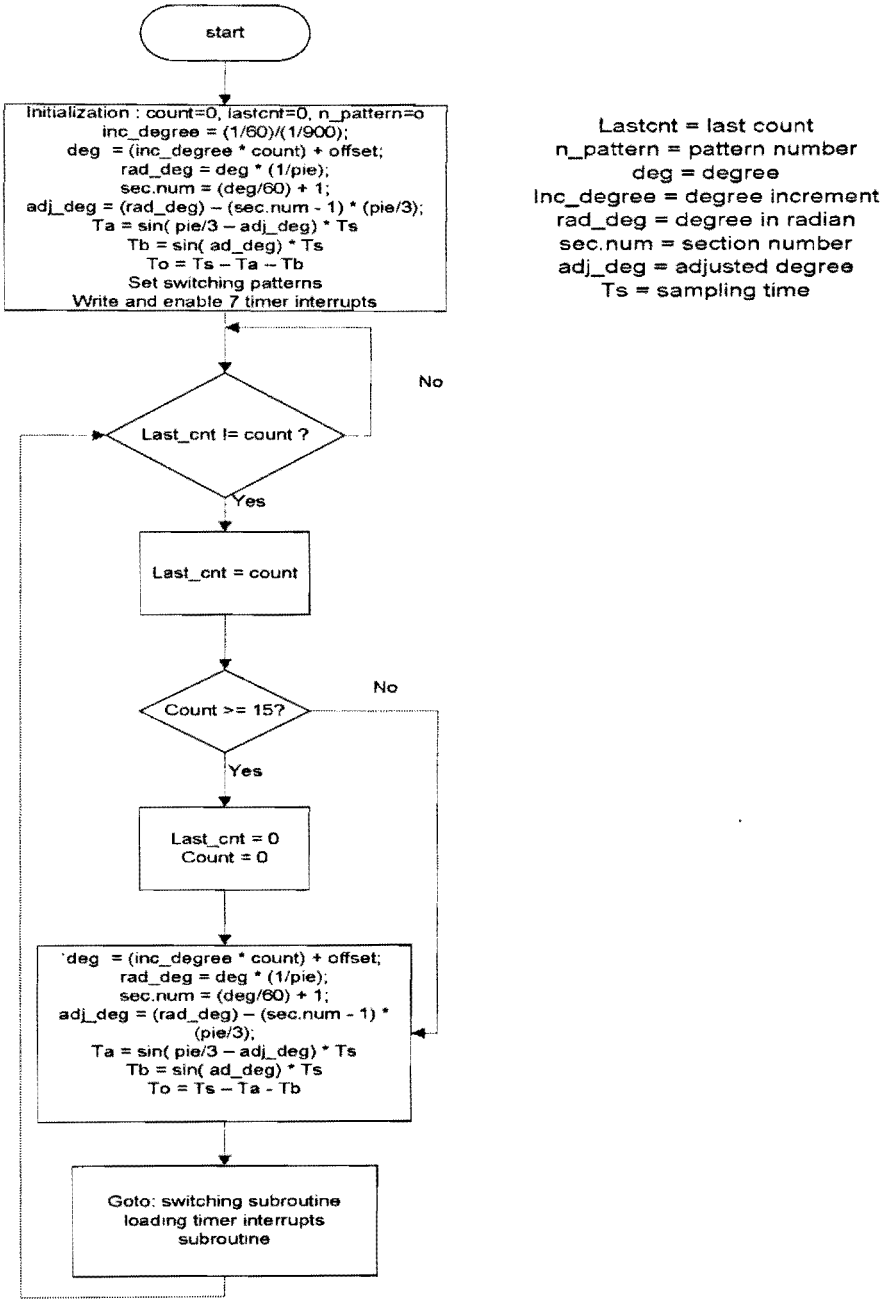


Figure 4.7-1 Software main flow chart

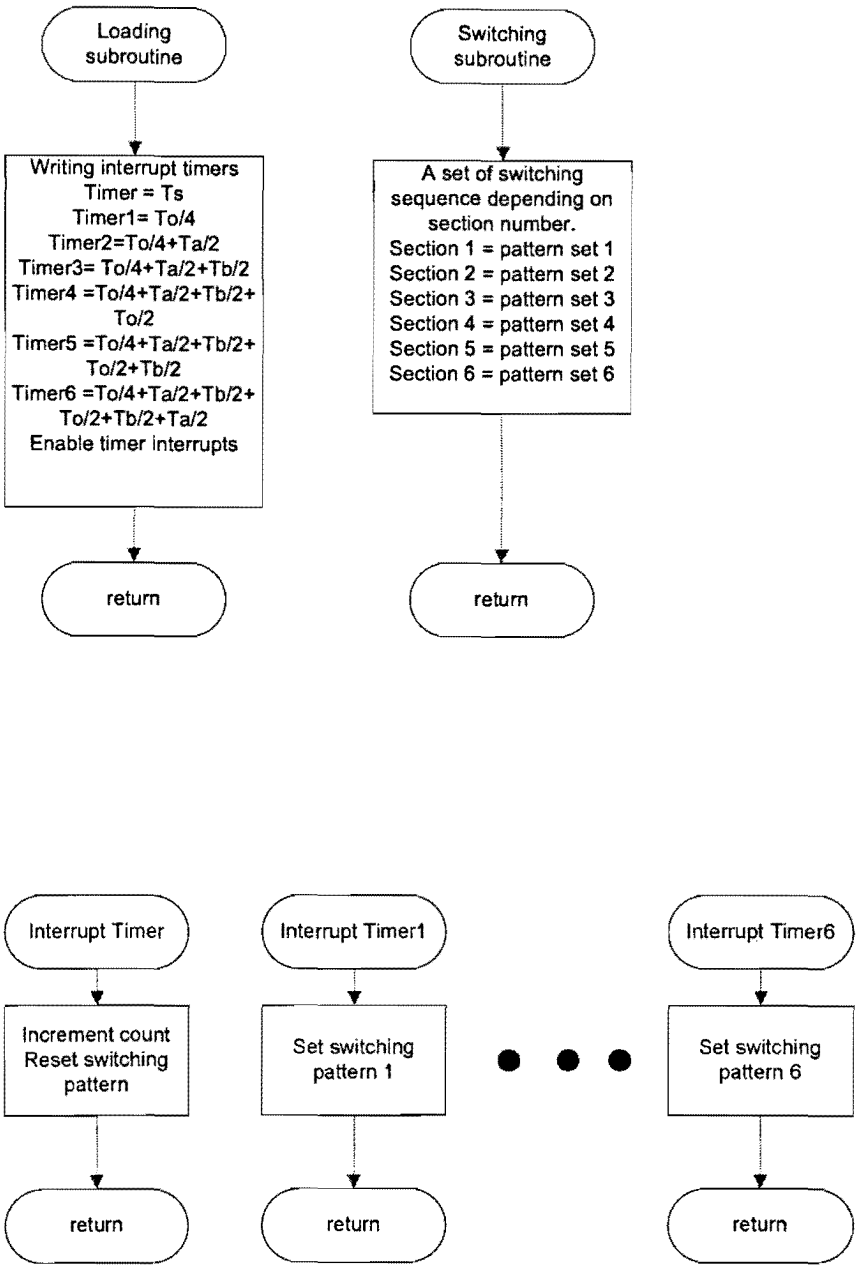


Figure 4.7-2 Flow charts of some subroutines

The detail of software implementation is described as the following. There are three parameters that the program needs to know in order to set the inverter frequency and amplitude. They are sampling frequency, modulating frequency and the modulation index. In this project the sampling frequency is set at 900Hz, the modulating frequency is at 60Hz, and the modulation index is 1.

The sampling period is

$$Sampling\_period = \frac{1}{Sampling\_frequency} = \frac{1}{900Hz} = 1.11mS \quad (4.4.7\text{-}1)$$

The number of sampling time per cycle is

$$Sampling\_time = \frac{900Hz}{60Hz} = 15 \quad (4.4.7\text{-}2)$$

The sampling angle increment in one circle is

$$Sampling\_increment = \frac{360}{15} = 24\deg ree \quad (4.4.7\text{-}3)$$

An interrupt timer at every sampling period is established. At each interrupt, a counter ranging from zero to 14 is increased by one and it will reset itself when the number reaches 15. The sampling angle at any given time is just the product of the number in the counter and the angle of the sampling increment.

$$Sampling\_angle = 24 * counter's\_number \quad (4.4.7\text{-}4)$$

When the angle is became known, the switching time constant T_a, T_b and T_o can be found but the angle has to be first converted to radian degree.

$$radian\_\deg ree(\theta) = \deg ree * \frac{\pi}{180} \quad (4.4.7\text{-}5)$$

$$T_a = \sin(\frac{\pi}{3} - \theta) \qquad \text{with} \quad m_a = 1 \quad (4.4.7\text{-}6)$$

$$T_b = \sin(\theta) \tag{4.4.7-7}$$

$$T_o = T_s - T_a - T_b \tag{4.4.7-8}$$

These time constant T_a, T_b and T_o are just the relative time as compared to the sampling period. In order to become real time, they have to be multiplied by a system clock time T_s related to the sampling time. These real times can now be used in setting the interrupt timers.

$$T_S = sampling\_period * system\_clock = \frac{1}{900Hz} * 50MHz \tag{4.4.7-9}$$

$$T_S = 55555.55$$

The angle is also used to figure out the sector number that the angle belongs to. The sector number decides the switching patterns. As indicated in the chapter II, Figure 2.5_2 of this report, each sector has its unique switching patterns.

$$Sector\_number = \text{int}(\frac{sampling\_angle}{60}) + 1 \tag{4.4.7-11}$$

Not only does the sector number affect the switching sequence, it also has effect on the order of the timing sequence. The odd sector number has this timing sequence: $\frac{T_o}{4}, \frac{T_a}{2}, \frac{T_b}{2}, \frac{T_o}{2}, \frac{T_b}{2}, \frac{T_a}{2}$ and $\frac{T_o}{4}$. The arrangement of the timing order for the even sector number is different, it is like this: $\frac{T_o}{4}, \frac{T_b}{2}, \frac{T_a}{2}, \frac{T_o}{2}, \frac{T_a}{2}, \frac{T_b}{2}$ and $\frac{T_o}{4}$. All these procedures are observed closely according to the switching rules laid out in the chapter II to ensure minimum switching loss.

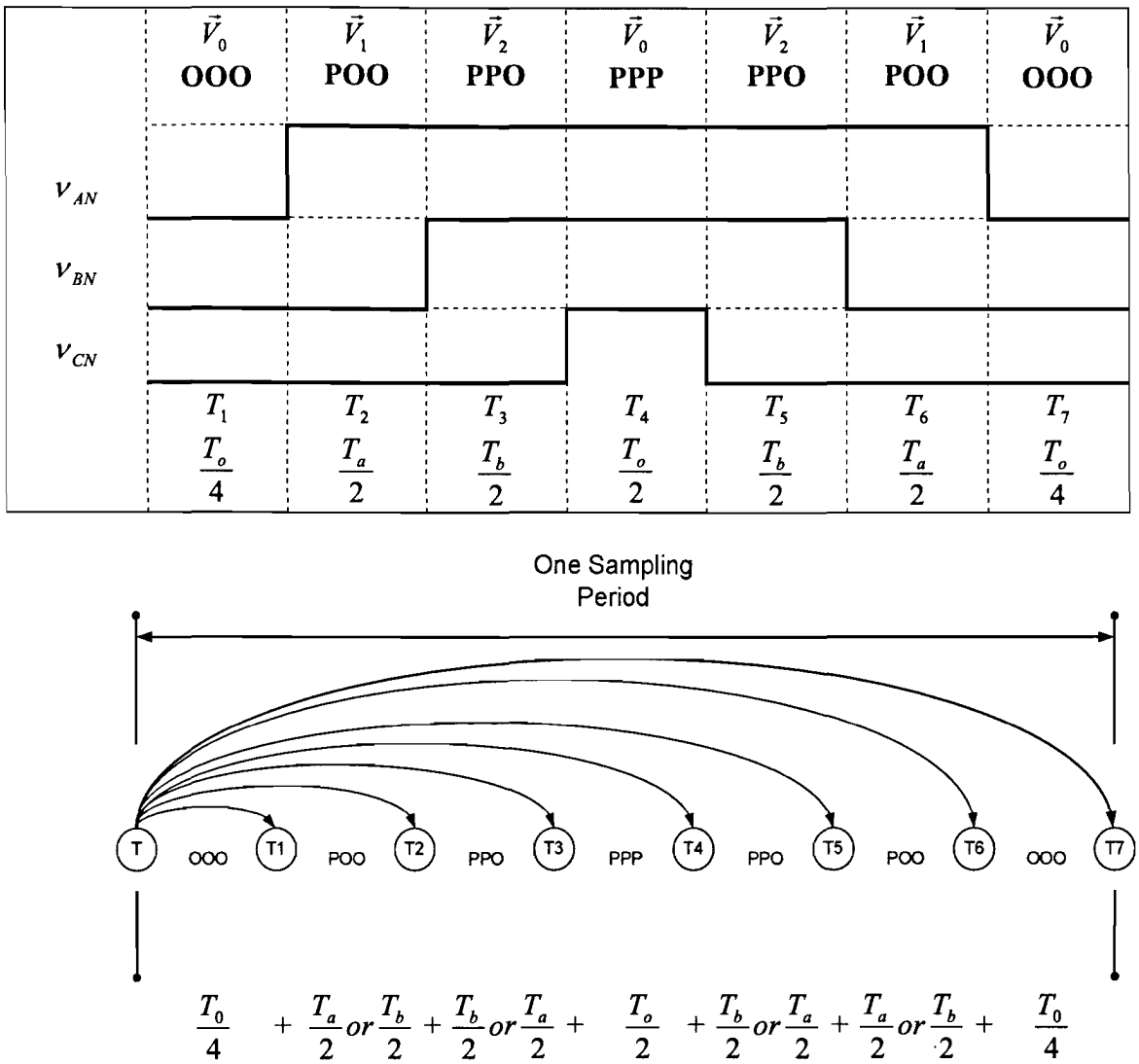


Figure 4.4.7-3 Overview of Switching Pattern, Timing Sequence and Interrupt Timers

Within one sampling period, T, there are seven interrupts are set to correspond to seven changes in switching sequence as shown in figure 4.4.7-3 where sector I is taken as an example. The length of time of each interrupt is the sum of current time and all of the previous timing. The timer 1 is the only exception when it starts from fresh. In this example, timer interrupt **T** is responsible for [OOO] switching, timer interrupt $\mathbf{T_1}$ is responsible for [POO] and

so on. The interrupt timer seven is not used here since it is redundant with the sampling time T. The contents of each timer are listed below.

$$\text{Interrupt timer T} = \frac{1}{900Hz} * T_s \qquad (4.4.7\text{-}12)$$

$$\text{Interrupt timer } T_1 = \frac{T_0}{4} * T_s \qquad (4.4.7\text{-}13)$$

$$\text{Interrupt timer } T_2 = (\frac{T_0}{4} + \frac{T_a}{2}) * T_s \qquad (4.4.7\text{-}14)$$

$$\text{Interrupt timer } T_3 = (\frac{T_0}{4} + \frac{T_a}{2} + \frac{T_b}{2}) * T_s \qquad (4.4.7\text{-}15)$$

$$\text{Interrupt timer } T_4 = (\frac{T_0}{4} + \frac{T_a}{2} + \frac{T_b}{2} + \frac{T_0}{2}) * T_s \qquad (4.4.7\text{-}16)$$

$$\text{Interrupt timer } T_5 = (\frac{T_0}{4} + \frac{T_a}{2} + \frac{T_b}{2} + \frac{T_0}{2} + \frac{T_b}{2}) * T_s \qquad (4.4.7\text{-}17)$$

$$\text{Interrupt timer } T_6 = (\frac{T_0}{4} + \frac{T_a}{2} + \frac{T_b}{2} + \frac{T_0}{2} + \frac{T_b}{2} + \frac{T_a}{2}) * T_s \qquad (4.4.7\text{-}18)$$

Once the interrupt timer of the sampling period occurs, the software program will select the switching patterns and the timing order then set off other six interrupt timers. The program then calculates the angle related timing, selects the switching pattern and timing sequence to prepare for the next sampling period.

The switching sequences are handled by interrupt timers. Usually, the number of timers available is fixed in microprocessor systems. With FPGA, users are able to instantiate as many as needed as long as the resource permits. Hardware choices ease software writing.

Armed with a hardware multiplier within the processor core, there was an attempt to avoid floating point computation. Integer arithmetic was implemented in order to speed up the calculation of the sine function.

$$Sin(\theta) = X - \frac{X^3}{3!} + \frac{X^5}{5!} \qquad (4.47\text{-}19)$$

Only three terms were used in those trials and some scaling factors are employed for the integer arithmetic method. The scale factor has to be chosen carefully so that at the fifth power term it does not exceed the capacity of the data register of the processor. The practice does not yield much higher computing speed as compared to the Nios II floating-point custom instructions. It makes calculation complicated and software codes not easy to read.

As it turns out, the Nios II floating-point custom instructions help writing the software an easy task. In this project, floating point manipulation is the major part. Instead of creating custom codes for sine or cosine functions, Nios II custom instructions have done the hard work and allow users to use simple ANSI C math functions. The Nios II processor performs single precision floating-point arithmetic operations. The time spent on code writing is much shorter, but much on chip resource is also occupied.

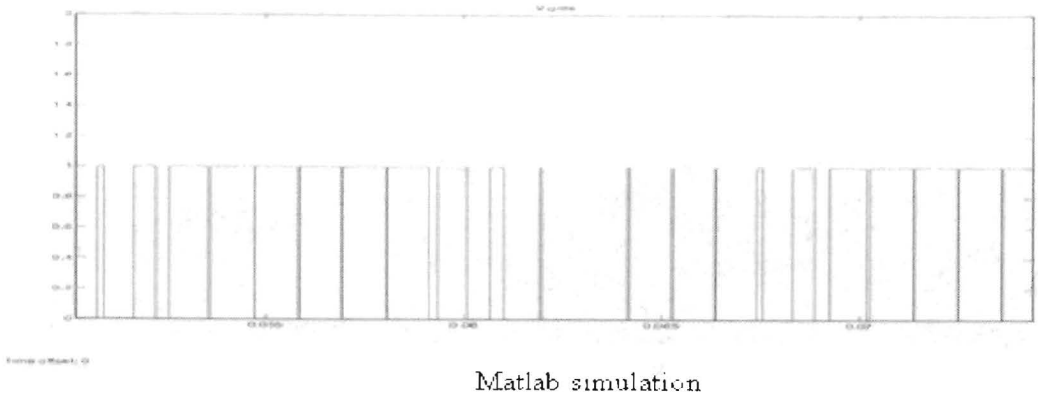
The debugging method here utilized the *printf* function of the C codes. This function transmits data from FPGA chip through UART and displayed them on the host computer screen. Timing constants such as T_a, T_b and T_o, sector number, sampling angle and switching pattern were able to be shown on the computer screen at a desired interval.

C code:

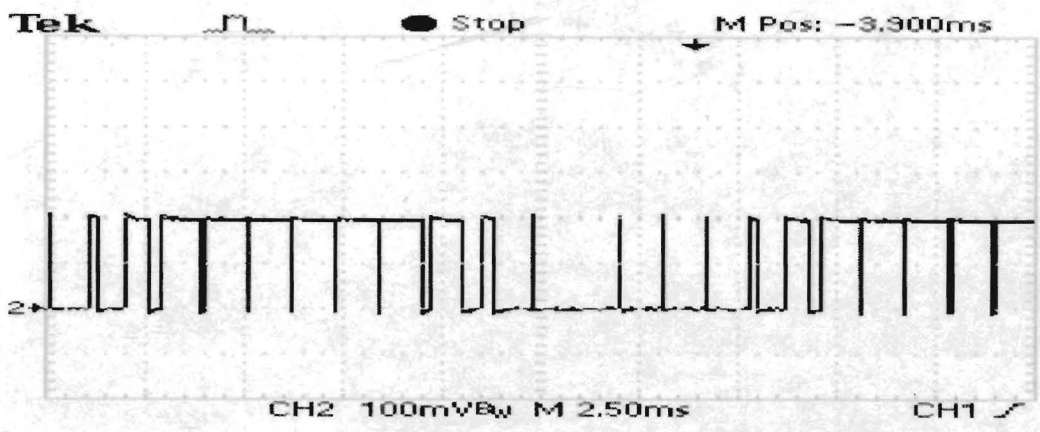
Printf("%d %d %d %d %u %u %u\n", sec.num, deg, pattern,T_0/4, T_a/2, T_b/2);

Sampling time was first changed to1.11Sec as to allow enough time for the data to be transferred and displayed. A complete cycle, 360 degree, of data were collected and compared to hand calculations. When the numbers were correctly matched, the sampling rate was put back to 1.11 mSec and the switching action was captured by an oscilloscope. The switching waveform was then weighed against the switching waveform of the Mathlab.

4.5 Design Result



Matlab simulation



SOPC Hardware output

Figure 4.5-1 Wave forms of software simulation and hardware output

The design result is right on its target. The pulse width modulation output of the hardware is identical to that of the Mathlab simulation. Since the SVM inverter design is intended for high power system (~3800V), it would be impractical to build a complete system just to show the validity of SOPC method. Hence, the switching waveform of Mathlab is used as a benchmark to verify the switching outcome of SOPC design. Figure

4.5-1 illustrates both result of the theory and practicality. The inverter sampling rate is set at 1/900 second, rotating at 60 Hz frequency and the amplitude modulation index is one. Figure 4.5-2 shows current, voltage, sector number, switching waveforms of the Matlab simulation at above setting.

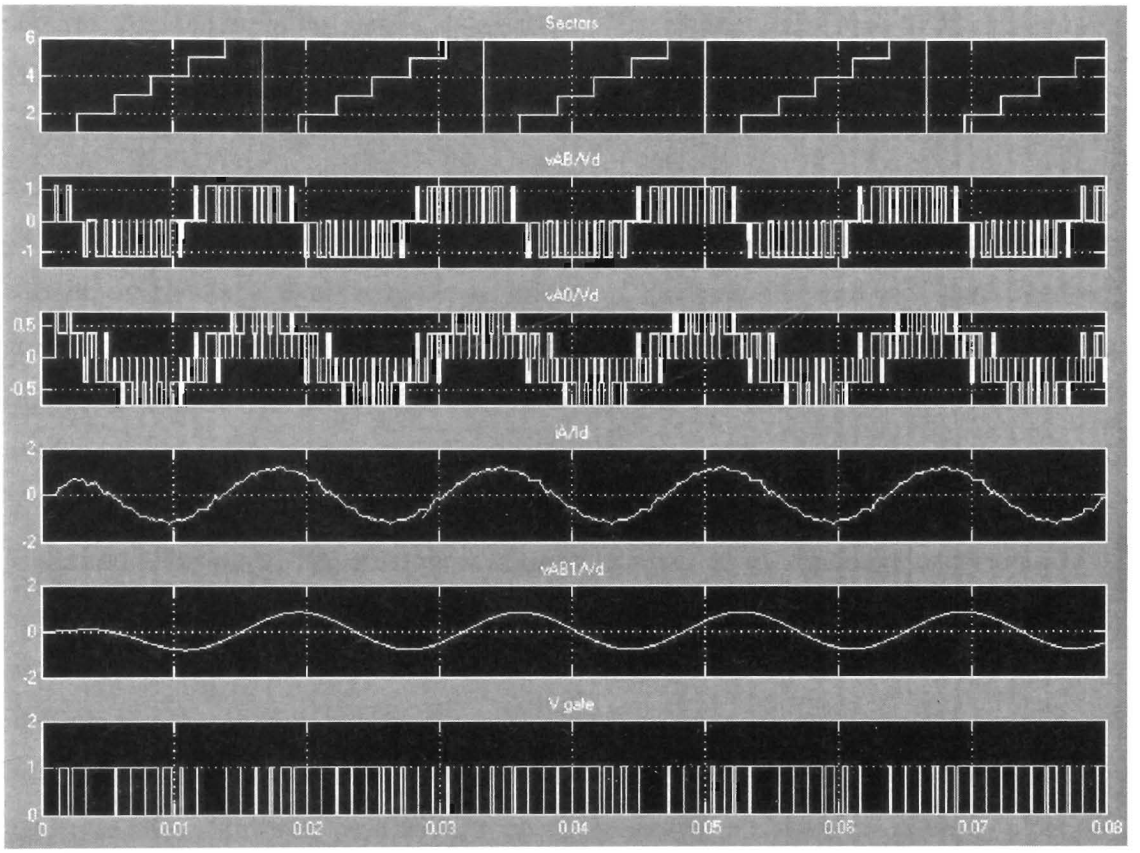


Figure 4.5-2 Matlab simulation waveforms

4.6 Future Work: Multiple Processors System

In this paper, all hardware resources and software effort are concentrated on realization of Space Vector Modulation (SVM). The processor spends most of its time computing power series of the two trigonometry sine functions and outputting switching patterns. At each sampling period, calculation of the timing vectors has to be carried out. That means no pre-calculated values or table is stored in the memory as to boost up the sampling speed. The highest true sampling rate obtained is close to 1000Hz with 50MHz system clock. Since sampling rate of the SVM is the dominant issue, there is little attention paid for controlling and safely running of the motor. The subjects such as over current, over voltage, over temperature protection, blanking time and user input interface are left out. To amend these shorting comings, a multiprocessor system within a SOPC is suggested. Basically, it is a system which incorporates two or more microprocessors working together to perform one or more related tasks.

Altera SOPC builder package allows users to add as many processors to a system as desired effortlessly. The arranging and connecting of hardware components are no longer an issue in building multiprocessor system. The challenge now lies in writing the software for the processors so that they do not conflict with one another. To prevent multiple processors from interfering with each other, hardware peripherals to coordinate effective operating of the processors are included in the Embedded Design Suite. The hardware allows different processors to claim ownership of a share resource for a period of time and to coordinate data exchange in a single resource such as memory. They are named hardware mutex core and hardware mailbox core accordingly.

A preliminary design for the open loop motor control system is shown below in figure 4.6-1. The first processor takes care of SVM and switching functions. The second processor looks after the safety operating of the machine and interfaces with users for input speed; this information is passed through the shared memory to the processor 1 which will decides either to stop, run or change speed of the motor. There is an added feature for this configuration. When the first processor detects no change in motor speed,

higher sampling rate can be obtained by pre-calculating all the timing vectors in one cycle and their values are stored in the memory. The next time around, the processor 1 just reads from memory and generates switching sequences without going through the long computing process. Higher sampling rate yields cleaner waveform and reduces THD.

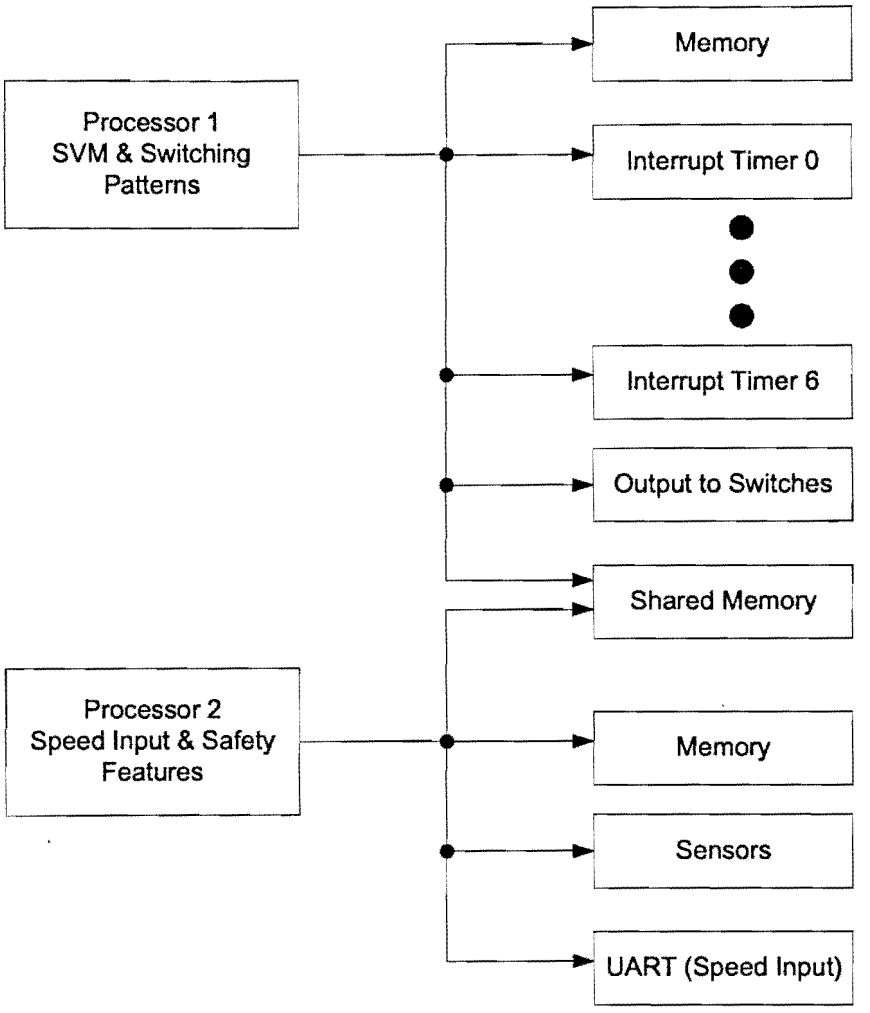


Figure 4.6-1 Multiple Processors in Open Loop Control System

A closed loop servo motor control like the FOC can also be carried out in the same fashion. The field oriented control unit can be comprised of more than one processors

operating together. Each processor acts as a functional unit, thereby, data are processed concurrently. The performance of the overall system is faster as a result.

The Altera Nios II development software is capable of debugging simultaneously a multiple processors design. Debug for all processors can run at the same time and is able to pause and restart each processor independently. Break points can be placed individually anywhere in a processor. Once a breakpoint is hit, it does not halt the debug procedures of other processors. This debugging capability facilitates the development of a multiple processors system.

Chapter 5

Conclusion

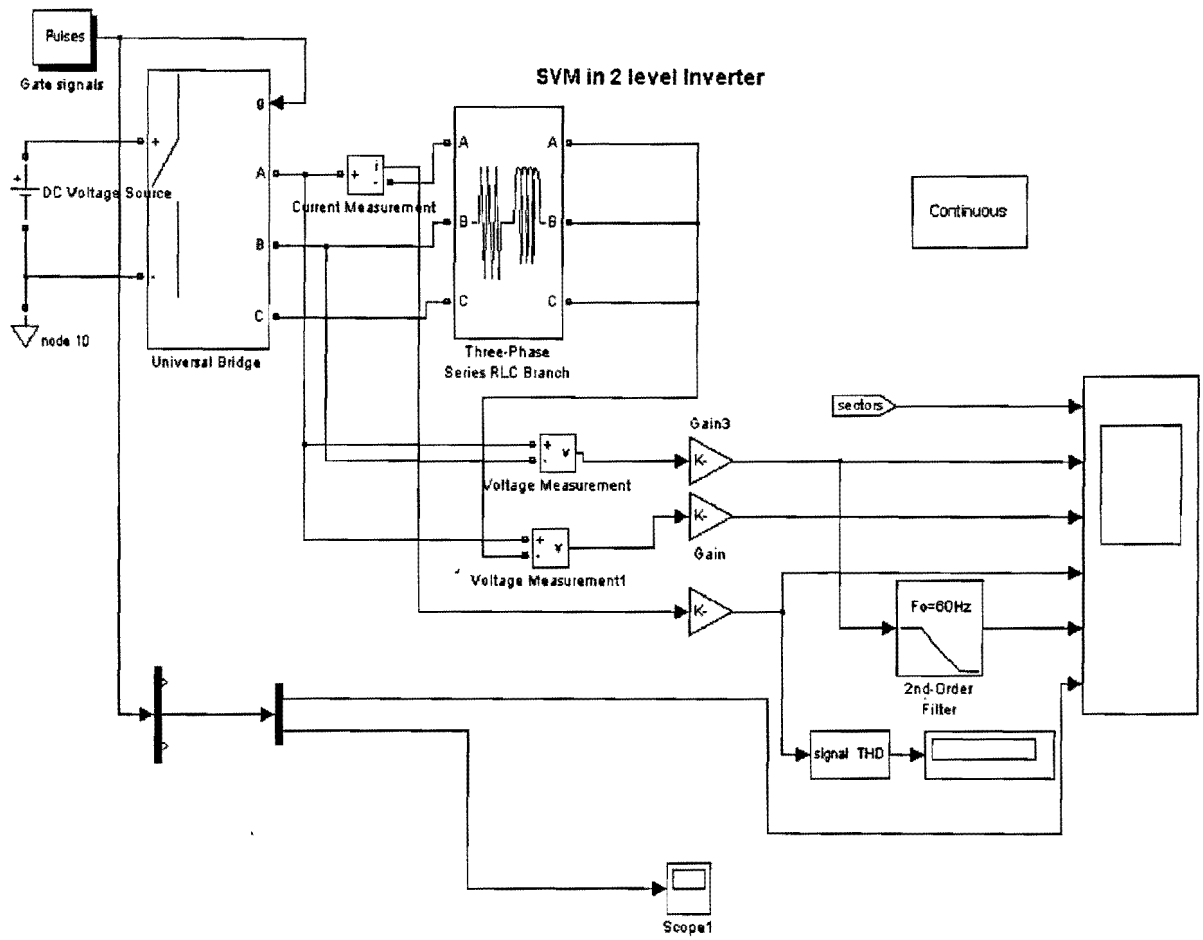
Space Vector Modulation is a superior method in converting DC voltage to AC voltage. It is a known fact that SVM has a lower THD, less switching loss, and more efficient usage of DC power supply as comparing to all other modulation schemes. The major disadvantage of this method is that it requires intensive computation. For this reason, DSP and high end microprocessor are often chosen to implement SVM. They are expensive because of complicated computer architecture is employed and hence huge resources are used to fabricate these chips. There is lack of flexibility in DSP or microprocessor technique. Once a DSP chip is elected, there is no easy turning back to make the hardware change without replacing with other DSPs. This may result in longer development period or an under utilization design. The recent emerging FPGA technology has given rise to a new approach to carry out SVM. It is called embedded system design or System On Programmable Chip SOPC. The system is consisted of a fast RISC microprocessor in a FPGA chip. Since FPGA can be configured into any logics or functions, coupling this with a processor makes it a robust combination. The processor is acting like a tasks distributor that it out sources the works to the hardware peripherals formed by FPGA. The work done by hardware is always faster than the software for

hardware works concurrently and software works sequentially. That also frees up the processor to operate on other task and the system becomes truly multitasking. Software coding gets easier as well as in the case of floating-point arithmetic. The hardware added is not limited only to peripherals; a multiple processors can be put together as well. This feature makes the hardware peripherals more intelligent. The over all system performance is increased as a consequence. Unlike DSP or microprocessor structure which has fixed peripherals, the hardware in the embedded system is reconfigurable. Designers can easily reiterate the hardware by adding or deleting functions in order to meet performance or price goal.

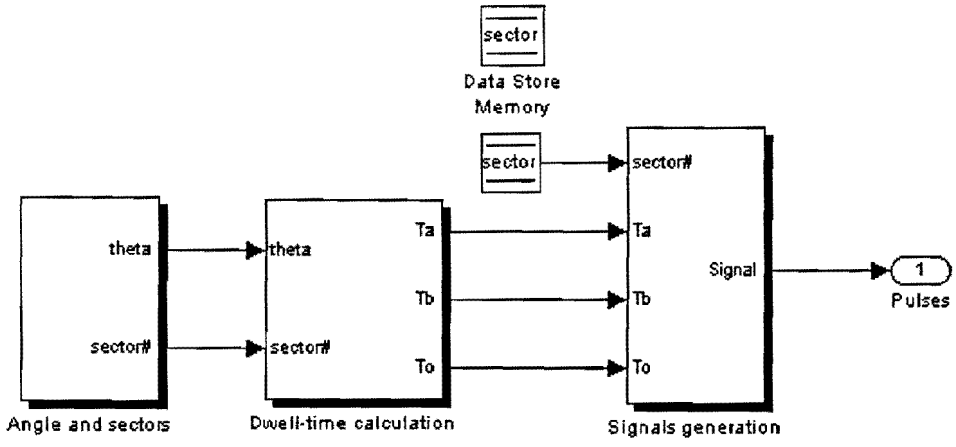
The tools for this project are Altera DE2 development board and Altera Quatus II design suite. It is a successful implementation of SVM in SOPC. The output switching waveform is the same as the Mathlab simulation. There is two-fold in the purpose of the Mathlab exercise. One is to observe how the theory works. The other is served as a benchmark to verify the practical outcome. It is a long journey to go from proving the theory of the Space Vector Modulation to practically working hardware design. In reality, it is the same procedure that any serious engineering design has to go through. This practice not only detects any design flaws at early stage. It also helps foreseeing problems may emerge later on. An idea is stemmed from the same reasons; SOPC can be served as hardware simulation for the design.

Appendix A

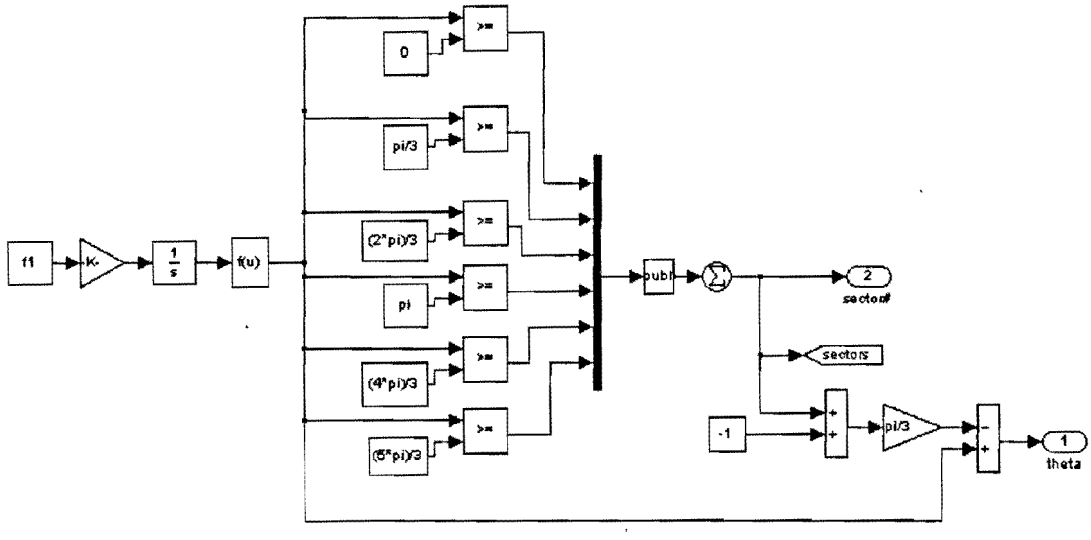
Mathlab Simulink Block Diagrams



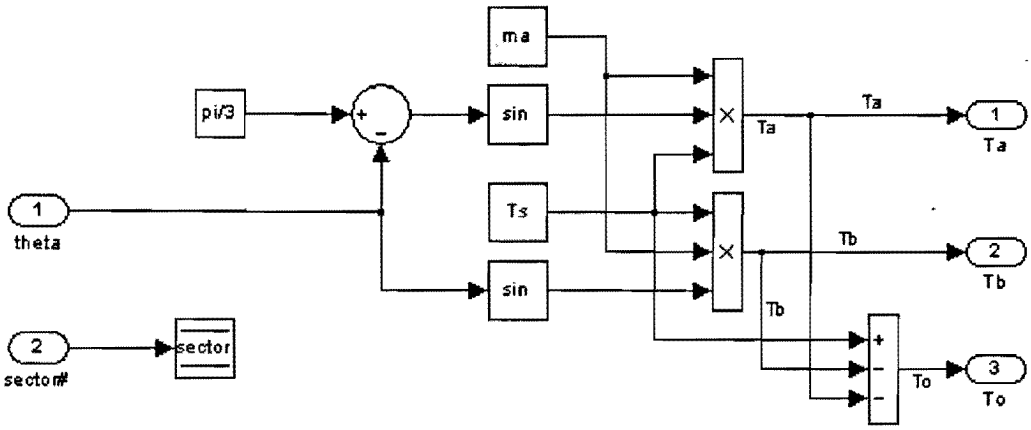
SVM overview in Matlab Simulink



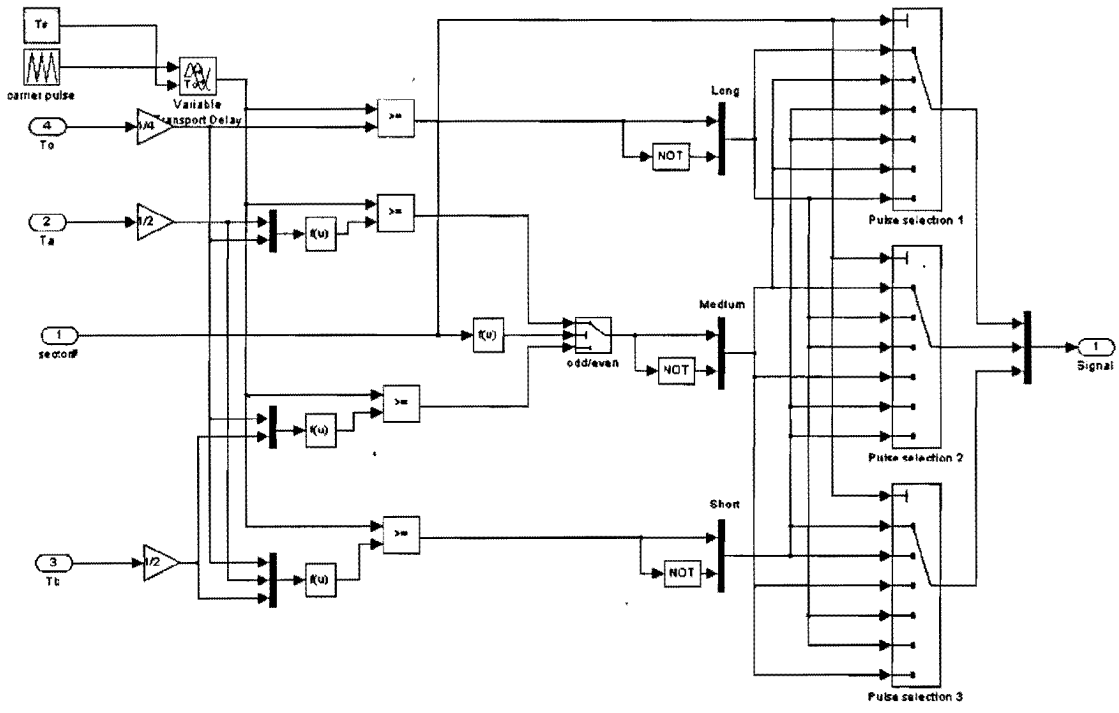
Pulses-Gate signal block overview



Angle and sectors block diagram



Dell time calculation diagram



Signal generation block diagram

Appendix B

Software Codes in C Programming Language

```
#include "count_binary.h"
#include "sys/alt_irq.h"
#include "system.h"
#include "unistd.h"
#include "math.h"
#include "float.h"
#include "altera_avalon_timer_regs.h"


//typedef unsigned int REGISTER;
//typedef unsigned int BITS;
// union TIMER_STATUS_REG {
//   REGISTER data;
//    struct {
//        BITS TO      : 1;
//        BITS RUN     : 1;
//        BITS unused  : 30;
//    } bits;
//};

//union TIMER_CONTROL_REG {
//    REGISTER data;
//    struct {
//        BITS ITO       :1;
//        BITS CONT      :1;
//        BITS START     :1;
//        BITS STOP      :1;
//        BITS unused    :28;
//    } bits;
//};

//struct TIMER_REGS {
//    union TIMER_STATUS_REG status;
//    union TIMER_CONTROL_REG control;
//    REGISTER periodl;
//    REGISTER periodh;
//   REGISTER snapl;
//    REGISTER snaph;
//};
//struct TIMER_REGS *TIMER = (struct TIMER_REGS *)(TIMER_BASE
|0x00021020);

static char lastcnt;
```

```
/* A "loop counter" variable. */

static alt_u8 count;
float rad = 0.0174532f;
float pi = 3.141592f;
float Ts = 55555.55f; //416666666.66f;    //55555.5555;//555555555.5;
float pi_by_3 = 1.04719f;
float rad_deg;
float adj_deg;
float adj_angl;

unsigned long int Ta;
unsigned long int Tb;
unsigned long int To;
unsigned long int adj_To_1;
unsigned long int adj_To_2;
unsigned long int adj_Ta;
unsigned long int adj_Tb;
unsigned short int c_deg;
char pattern_0;
char pattern_1_5;
char pattern_2_4;
char pattern_3;
char pattern;
char n_pattern;
char section;
char flag;
//char red_flag;
static char int_cnt;
char cnt;
unsigned short int  deg;

struct b_type {
    unsigned last:1;
    int: 7;
};

union sec {
    char num;
    struct b_type bits;
} sec;

union duration_type {
    unsigned short int d[2];
    unsigned long int l_durtn;
} duration;

void handle_timer_interrupts () {
    count++;
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0);
////    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 8);
}

void handle_timer1_interrupts () {
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_1_BASE, 0);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_1_BASE, 8);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_1_5);
```

```
    n_pattern = pattern_1_5;
}

void handle_timer2_interrupts () {
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_2_BASE, 0);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_2_BASE, 8);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_2_4);
      n_pattern = pattern_2_4;
}

void handle_timer3_interrupts () {
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_3_BASE, 0);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_3_BASE, 8);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_3);
    n_pattern = pattern_3;
}

void handle_timer4_interrupts () {
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_4_BASE, 0);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_4_BASE, 8);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_2_4);
    n_pattern = pattern_2_4;
}

void handle_timer5_interrupts () {
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_5_BASE, 0);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_5_BASE, 8);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_1_5);
    n_pattern = pattern_1_5;
}

void handle_timer6_interrupts () {
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_6_BASE, 0);
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_6_BASE, 8);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_0);
    n_pattern = pattern_0;
}



static void initial_message()
{
    printf("\n\n**************************\n");
    printf("* Hello from Nios II!    *\n");
    printf("* Counting from 00 to ff *\n");
    printf("**************************\n");
}



static void switching()
{
    switch(section)
    {
```

```
        case 0x1:
            pattern_0 = 0x0;
            pattern_1_5 = 0x4;
            pattern_2_4 = 0x6;
            pattern_3 = 0x7;
           break;

        case 0x2:
            pattern_0 = 0x0;
            pattern_1_5 = 0x2;
            pattern_2_4 = 0x6;
            pattern_3 = 0x7;
           break;

        case 0x3:
            pattern_0 = 0x0;
            pattern_1_5 = 0x2;
            pattern_2_4 = 0x3;
            pattern_3 = 0x7;
           break;

        case 0x4:
            pattern_0 = 0x0;
            pattern_1_5 = 0x1;
            pattern_2_4 = 0x3;
            pattern_3 = 0x7;
           break;

        case 0x5:
            pattern_0 = 0x0;
            pattern_1_5 = 0x1;
            pattern_2_4 = 0x5;
            pattern_3 = 0x7;
           break;

        case 0x6:
            pattern_0 = 0x0;
            pattern_1_5 = 0x4;
            pattern_2_4 = 0x5;
            pattern_3 = 0x7;
           break;

    }
}

static void load_1()
{
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_1_BASE, duration.d[0]);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_1_BASE, duration.d[1]);
}

static void load_2()
{
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_2_BASE, duration.d[0]);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_2_BASE, duration.d[1]);
}
```

```
static void load_3()
{
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_3_BASE, duration.d[0]);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_3_BASE, duration.d[1]);
}

static void load_4()
{
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_4_BASE, duration.d[0]);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_4_BASE, duration.d[1]);
}

static void load_5()
{
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_5_BASE, duration.d[0]);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_5_BASE, duration.d[1]);
}

static void load_6()
{
    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_6_BASE, duration.d[0]);
    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_6_BASE, duration.d[1]);
}

//static void load_7()
//{
//    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_BASE, duration.d[0]);
//    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_BASE, duration.d[1]);
//}

static void fire()
{
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_1_BASE, 5);

    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_2_BASE, 5);

    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_3_BASE, 5);

    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_4_BASE, 5);

    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_5_BASE, 5);

    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_6_BASE, 5);


 //   IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 5);

}

static void loading()
{
    duration.l_durtn = 0;
    // Segment 1 //
    duration.l_durtn = adj_To_1;
    load_1();

    if (flag != 0)    //  It is odd section //
```

```
        {

            // Segment 2 //
               duration.l_durtn = duration.l_durtn + adj_Ta;
               load_2();
            // Segment 3 //
               duration.l_durtn = duration.l_durtn + adj_Tb;
               load_3();
            // Segment 4 //
               duration.l_durtn = duration.l_durtn + adj_To_2;
               load_4();
            // Segment 5 //
               duration.l_durtn = duration.l_durtn + adj_Tb;
               load_5();
            // Segment 6 //
               duration.l_durtn = duration.l_durtn + adj_Ta;
               load_6();

          }
    else                                    // It is even section //
        {

            // Segment 2 //
               duration.l_durtn = duration.l_durtn + adj_Tb;
               load_2();
            // Segment 3 //
               duration.l_durtn = duration.l_durtn + adj_Ta;
               load_3();
            // Segment 4 //
               duration.l_durtn = duration.l_durtn + adj_To_2;
               load_4();
            // Segment 5 //
               duration.l_durtn = duration.l_durtn + adj_Ta;
               load_5();
            // Segment 6 //
               duration.l_durtn = duration.l_durtn + adj_Tb;
               load_6();

        }
        fire();
}



int main(void)
{


//    FILE * lcd;
//unsigned long int sine_fcn();
    count = 0;
    lastcnt = 1;
    sec.num = 0;
    deg = 0;
    int_cnt = 1;
```

```
//    int_cnt = 0;


/* Initial message to output. */

    initial_message();
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0x0c);

    alt_irq_register(TIMER_IRQ, NULL, (void*)handle_timer_interrupts);

//duration.l_durtn= 0x2faf080;
duration.l_durtn= (unsigned long int)Ts;

IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_BASE, duration.d[0]);
IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_BASE, duration.d[1]);

IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 7);
alt_irq_register(TIMER_IRQ, NULL, (void*)handle_timer_interrupts);

/* initialization */

alt_irq_register(TIMER_1_IRQ, NULL, (void*)handle_timer1_interrupts);
alt_irq_register(TIMER_2_IRQ, NULL, (void*)handle_timer2_interrupts);
alt_irq_register(TIMER_3_IRQ, NULL, (void*)handle_timer3_interrupts);
alt_irq_register(TIMER_4_IRQ, NULL, (void*)handle_timer4_interrupts);
alt_irq_register(TIMER_5_IRQ, NULL, (void*)handle_timer5_interrupts);
alt_irq_register(TIMER_6_IRQ, NULL, (void*)handle_timer6_interrupts);


 //          IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_0);
           n_pattern = 0;
           cnt = 0;
           //red_flag = 0;
           deg = (24 *count) + 4;
           rad_deg = deg * rad;

           sec.num = (deg/60) + 1;

           adj_deg = rad_deg - (sec.num -1)*pi_by_3;
           adj_angl = pi_by_3 - adj_deg;

           Ta = (unsigned long int)(sin(adj_angl)*Ts);
           Tb = (unsigned long int)(sin(adj_deg)*Ts);
           To = (unsigned long int)Ts - Ta - Tb ;

           adj_To_2 = To >> 1;
           adj_To_1 = To >> 2;
           adj_Ta = Ta >> 1;
           adj_Tb = Tb >> 1;

           section = sec.num;
           flag = (char)(sec.bits.last);
           switching();
           loading();
```

```
            printf("%d %d %d %d %u %u %u\n",sec.num,flag,
deg,pattern,adj_To_2, adj_Ta, adj_Tb);

 //           IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_0);
            n_pattern = 0;


            //TIMER -> periodl = dur_0_6.d[0];
            //TIMER -> periodh = dur_0_6.d[1];
            //TIMER -> status.bits.TO = 0;
            //TIMER -> control.bits.ITO = 1;
            //TIMER->control.bits.START = 1;


/* Continue 0-ff counting loop. */

    while( 1 )
    {

     if (lastcnt != count)
        {
            lastcnt = count;
            if (count >= 15)
            {
               count = 0;
               lastcnt =0;
            }

            IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, pattern_0);
            n_pattern = 0;

           switching();
           loading();

            deg = (24 *count) + 4;
            rad_deg = (float)deg * rad;

            sec.num = (deg/60) + 1;

            adj_deg = rad_deg -(float) (sec.num -1)*pi_by_3;
            adj_angl = pi_by_3 - adj_deg;

            Ta = (unsigned long int)(sin(adj_angl)*Ts);
            Tb = (unsigned long int)(sin(adj_deg)*Ts);
            To = (unsigned long int)Ts - Ta - Tb ;

            adj_To_2 = To >> 1;
            adj_To_1 = To >> 2;
            adj_Ta = Ta >> 1;
            adj_Tb = Tb >> 1;


            c_deg = deg;
            section = sec.num;
            flag = (char)(sec.bits.last);

            //switching();
```

```
            //loading();

   //          count_all( lcd );

         //printf("%d %d %d %d %u %u %u\n",sec.num,flag, deg,pattern,
adj_To_2, adj_Ta, adj_Tb);

        }

    }
    return 0;
}
```

References

[1]Ned Mohan, "Electric Drives, An Integrative Approach", MNPERE Minneapolis, 2001, pp7-1.

[2]Harold W.Gingrich, "Electrical Machinery, Transformers, and Controls", Prentice-Hall, Inc., 1979, pp231.

[3]Ned Mohan, "Electric Drives, An Integrative Approach", MNPERE Minneapolis, 2001, pp11-1.

[4]Mohan, Undeland, Robbins, "Power Electronics-Converters, Applications and Design", John Wiley & Sons, INC., 2003, pp203.

[5]Bin Wu, "High-Power Converters and AC Drives", John Wiley and Sons, INC., 2006, pp 6.3.

[6]Microchip, "VF Control of 3-Phase Induction Motor Using Space Vector Modulation", AN955, 2005, pp1.

[7]Hein Willi Van Der Broeck, Hans-Christoph Skudelny and Georg Viktor Stanke, "Analysis and Realization of a Pulsewidth Modulator Based on Voltage Space Vector", IEEE Transaction on Industry Applications, Vol. 24, No. 1, January/February 1988, pp143.

[8]D.Rathnakumar, J.Lakshmana Perumal and T.Srinivasan, "A new Software Implementation of Space Vector PWM", Proceedings IEEE SoutheastCon, 2005, pp131.

[9]Clive "Max" Maxfield, "The Design Warrior's Guide to FPGAs", Newnes, 2004, pp95, pp229.

[10]Michel Cosnard, Denis Trystram, "Parallel Algorithms and Architectures", International Thomson Computer Press, 1995, pp22.

[11] Douglas Gard, "Digital Signal Processor Architecture", US patent 5954811, 1999, pp20.

[12]Theerayod Wiangtong and Prasoot Dechsuwan, "Unified Motor Controller Based on Space Vector Modulation Technique", IEEE International Symposium on Circuit and System, 2006, pp5635.

[13]Rui Wu, Donghua Chen and Shaojun Xie, "A Three-Dimentional Space Vector Modulation Algorithm in A-B-C Coordinate Implemented By a FPGA", IEEE 32nd Annual conference of Industrial Electronics Society, 2005, pp1071.

[14]Su Chen and Gera Joos, "Symmetrical SVPWM Pattern Generator Using Field Programmable Gate Array Implementation", 17th Annual IEEE Applied Power Electronics Conference and Exposition, 2002, pp1004.

[15]Guijie Yang, Pinzhi Zhao and Zhaoyong Zhou, "The Design of SVPWM IP Core Based on FPGA", The 2008 IEEE Conference on Embedded Software ans System Symposia, 2008, pp191.

[16] Zhaoyong Zhou and Tiecai Li, "Design of a Universal Space Vector PWM Controller Based on FPGA", IEEE 19th Annual IEEE Applied Power Electronics Conference and Exposition, 2004, pp1968.

[17]Bin Wu, "High-Power Converters and AC Drives", John Wiley and Sons, INC., 2006, pp 6.1.

[18]Zainalabedin Navabi, PH.D., "Embedded Core Design with FPGAs", McGraw-Hill, 2007, pp217, pp391.

[19]Bin Wu, "High-Power Converters and AC Drives", John Wiley and Sons, INC., 2006, pp6.1-pp6.13.

[20]Altera Corporation, "Nios II Hardware Development Tutorial", Nios II Applications, 2007, pp1-1 –pp1-41.