Digital Predictive Current Control of Induction Machines

Soren John Henriksen, BE(Hons 1)

March 2001

A thesis submitted to embody the research carried out to fulfil the requirements for the degree of:

Master Of Engineering
in Electrical Engineering
at The University of Newcastle
New South Wales, Australia

Declaration

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

Soren John Henriksen, BE(Hons 1)

March 2001

Acknowledgements

There are a number of people I would like to thank for their contribution to the progress of this project. Dr. Robert Betz and Dr Brian Cook offered the original concept, and supervised the course of the research.

This project required considerable hardware construction and modification. Peter McLauchlan and Tim Wylie were a great help in this regard.

Professor Graham Goodwin, and the Centre for Integrated Dynamics and Control (CIDAC) was a great help in terms of resources and expertise. My fellow postgraduates and workers at CIDAC have also been of assistance, both technically, and in the less technical aspects of postgraduate life.

Finally, I must thank my family for their perseverance through the duration of this project.

Contents

Abstract

This thesis presents the design of an induction machine current controller that may be implemented entirely in digital hardware. In the past, machine controllers have either been limited to the use of analogue circuitry, or require some form of microprocessor device. The design described in this thesis has been implemented in a single EPLD (Erasable-Programmable Logic Device), and would be suitable for implementation in an ASIC (Application Specific Integrated Circuit). The hardware current controller allows high switching frequencies with only modest external processing power.

In order to build a controller in hardware, a suitable algorithm must first be found. This thesis surveys a number of methods, and details a constant-switching frequency predictive control algorithm. Using a stationary reference frame, the control equations are sufficiently simple computationally that a hardware approach is feasible. In order to validate the performance of the controller equations, a theoretical analysis, simulation and experimental results are presented.

Although the basic controller algorithm is not new, this thesis presents a number of extensions to improve its performance and stability. This includes an effective leakage inductance estimator, and a back-emf predictor that does not incur stability penalties. This overcomes a traditional shortcoming of stationary-frame controllers. The result is a controller that does not require any machine parameters to be supplied.

Along with a DSP-based version of the controller, a design architecture is shown for a completely hardware implementation. This uses a microcoded state-machine approach to construct the controller in an efficient and flexible manner. The final controller is capable of control rates of up to 20kHz using a commonly-available logic device.

As part of the thesis, simulation and experimental results are presented to demonstrate the performance of the algorithm. These results demonstrate that the controller operates effectively in a practical environment.

Chapter 1

Introduction

1.1 Induction Machine Control

Induction motors offer a number of advantages over the alternate types of electrical machines. They have found widespread use in a wide variety of applications because of their mechanical durability and relatively low cost. Most industrial fans, pumps and compressors use this type of machine.

In the past, despite their higher cost and lower reliability, DC machines have been favoured for applications where good control over the machine is required. This is because it has been easier to control the output torque of that type of machine. In most induction machine applications, little effort has been taken to accurately control the operation of the machine. For example, fans or pumps may be operated continuously at the rated voltage.

However, the development of vector control of induction machines through the 1970s led to a new range of applications for this type of machine. By decoupling the control of the machine flux and torque, vector control provides induction machines with similar control properties to DC machines.

There is now a great interest in induction machines for servo-type applications where precise control is required. Many examples are found in robotics and manufacturing. As the price of power electronics and control equipment decreases, the benefits of improved control may also be passed onto the areas where it was previously not seen as sufficiently important.

A variety of methods have been proposed for the control of induction machines. These generally fall into one of two categories:

- Direct torque control, where torque control of the machine is provided in a single control loop.

- Cascaded control, where a torque controller is designed using a stator current control loop.

These methods each have their advantages, as described in the following chapter. The focus of this thesis is the current control loop of the cascaded control architecture.

1.2 Overview

The body of this thesis is composed of five chapters. The contents of these are:

- Chapter two is a discussion of prior work in the area of induction machine control. This includes a brief overview of torque control methods, followed by analysis of a number of proposed current control schemes.
- In chapter three, a constant switching frequency predictive current controller is described. This includes the development of the model, along with parameter estimation.
- Following the presentation of the controller, it's performance is analysed from a theoretical viewpoint. Stability analysis is performed, along with sensitivity to parameter and model error. In addition, modifications are proposed to the controller in the light of the analysis.
- Chapter five includes additional details about the implementation of the predictive controller. A significant portion of this involves the development of hardware for data acquisition and pulse-width modulation.
- The final chapter presents an alternate implementation of the controller. This version is designed to fit in a single EPLD (Erasable-Programmable Logic Device). The result is a digital predictive current controller implemented completely in hardware.

1.3 Key Contributions

The algorithm presented in this thesis was developed out of the work of Betz and Cook[3]. Beyond this, the project has involved new work in a number of different aspects. These include:

- Additional analysis of the validity of the machine model.
- Development of a numerically efficient predictive current controller and PWM generator.
- Development of a new inductance estimator.
- Theoretical analysis of the controller.
- Analysis of resistance effects.
- Back-emf prediction schemes to overcome controller limitations.

- Development of the sampling system and pulse-width modulation firmware.
- Implementation of the current, torque, speed and position controllers in software.
- Altering the control algorithm for computational efficiency.
- Design and implementation of an EPLD version of the controller.

1.4 Publications

The following publications are a direct result of the research leading to this thesis:

- R.E. Betz, B.J. Cook, and S.J. Henriksen. Digital current controller for three phase voltage source inverters. In *Proceedings of the IEEE IAS Annual Meeting*, New Orleans, October 1997.
- Soren J. Henriksen, Robert E. Betz, and Brian J. Cook. Digital hardware implementation of a current controller for im variable-speed drives. In *Proceedings of the IEEE IAS Annual Meeting*, St. Louis, October 1998.
- Soren J. Henriksen, Robert E. Betz, and Brian J. Cook. Digital hardware implementation of a current controller for im variable-speed drives. *IEEE Transactions on Industry Applications*, Vol. 35(No. 5):pp. 1021–1029, Sep 1999.
- Soren J. Henriksen, Robert E. Betz, and Brian J. Cook. Induction machine current control in digital hardware. In *Proceedings of the Australasian Universities Power Engineering Conference*, pages 557–562, Darwin, Septemper 1999.

Chapter 2

Survey of Prior Work

2.1 Introduction

This chapter investigates existing work in the area of machine control and, in particular, machine current control. There are many different types of machines, applications and controllers, but the focus of this thesis is on the medium to high performance control of induction machines using conventional three phase inverters. In order to place the review of current control in context, a brief review of the main induction machine control techniques and architectures are presented.

In this chapter, a basic overview is given of machine control architectures. This includes a brief description of some scalar and vector torque control methods. While the content of the thesis is based around the current control loop, this is typically in the context of torque control.

The focus then narrows to the current control loop, and known issues concerning this aspect are presented. Following this, a selection of popular current control methods from the literature are presented and reviewed.

2.2 Machine Control Overview

2.2.1 Scalar Control

In terms of control, a very popular approach for induction machines is the simple open-loop V/f control. Although this performs poorly from a control point of view, it is quite adequate for the majority of existing applications[8]. Its great advantage is in simplicity and robustness.

The name V/f, or $volts/frequency$ (also known as $volts/hertz$) is derived from the linear relationship between the rectifier output voltage and frequency. There is no need for a current control loop, or even the use of current measurements. Instead, the voltage and frequency are specified in open-loop based on the required speed of operation.

In an induction machine, the air-gap flux is approximately equal to the integral of

the voltage over time. For a constant flux magnitude, an increase in the voltage must be compensated by a reduction in the period of the voltage sinusoid. As a result, the V/f rule provides approximately constant machine flux over a range of operating points.

The controller is not suited to torque control, but rather an approximate speed control. The electrical rotation speed is set by the controller, and the mechanical rotation speed is governed by the amount of slip. As the load increases, the slip increases, resulting in greater torque. This is an inherent proportional feedback loop, which is stable, but suffers from poor regulation.

There are many shortcomings of this method. Apart from the variations in output shaft speed with load, the flux regulation is also only approximate. As a result, there have been many schemes[7] to address the shortcomings. However, these are not of great interest in a modern context.

Prior to the widespread adoption of vector control, a number of other scalar control methods were introduced. These controllers calculated set-points for the supply frequency and either the voltage or current. This approach generally suffered from poor transient performance on account of coupling between the control variables. For example, an attempted variation in torque would also result in a change in the machine flux. Poor modelling of this coupling necessarily leads to a sluggish controller.

2.2.2 Vector Control

The fundamental quantity to be controlled in an induction machine drive is the electrical torque. This quantity is closely related to the mechanical dynamics, which are ultimately to be controlled. In the case of a DC machine, the torque is proportional to the product of the field and armature currents. The force is produced when the current of the armature flows through the magnetic flux generated by the field winding. During normal operation, the field current is maintained as a constant, for a constant machine flux, and the armature current is varied according to the required torque. This allows the torque to be freely controlled without any associated additional dynamics.

Vector controllers provide a decoupling mechanism for induction machines that allow the torque and flux to be independently manipulated in the same manner as a DC machine. Conceptually, this can be achieved by placing the controller in a two-phase reference frame that is rotating at a particular speed. In this reference frame, one axis controls the flux, and the other the torque. The difficulty in vector control is correctly aligning the reference frame over time.

A diagram, showing the state of a vector controller, is shown in Figure 2.1. This diagram represents the state of the machine at an instant in time. The vectors indicate the directions and magnitudes of the currents and fluxes within the machine at that time. In Figure 2.1, two sets of axes are shown. One is the stationary $d-q$ axis, and the other is one rotating at the electrical rotation speed, and aligned with the rotor flux. The vector controller operates in this rotating axis, and controls the stator current components i_{ds}

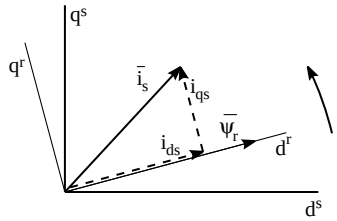


Figure 2.1: Machine flux and current vectors.

and i_{qs}

The torque equation for an induction machine may be expressed as,

$$T_e = \frac{3}{2} P_p \bar{\psi}_m \times \bar{i}_r \tag{2.1}$$

In the rotating coordinates, this cross-product is equal to the scalar product of $\psi_m i_{qs}$. Because of the alignment of the axes, the the flux is proportional to the other component of the stator current, i_{ds}. For normal operation, i_{ds} is kept constant, to maintain the machine magnetising flux, and i_{qs} is varied to control the torque.

In order to operate the controller, it is necessary to convert the rotating coordinate values into the stationary frame. The remainder of the problem is then one of current control. This is the area to be addressed in this thesis. To perform the coordinate transformation, it is necessary to know the angle of the machine rotor flux. There are two basic methods for achieving the transformation, the direct method, and the indirect method.

Direct Method

The direct method of vector control attempts to directly measure or estimate the machine flux, and use this to determine the transformation angle. While direct flux measurements are difficult physically, an estimate may instead be derived from the stator voltages and currents. At the higher speed ranges, the flux estimate may be reliably estimated from the integral of the stator voltage. However, this is subject to errors from harmonics, and is not useful at lower speeds.

Due to the difficulty in reliably obtaining an estimate of the rotor flux direction, the direct method of vector control is not commonly used. Instead, the indirect method has gained popularity.

Indirect Method

Instead of attempting to directly estimate the angle of the rotor flux, the indirect method of vector control relates the flux rotation to the slip frequency. This may be found by considering the machine equivalent circuit, as shown in Figure 2.2.

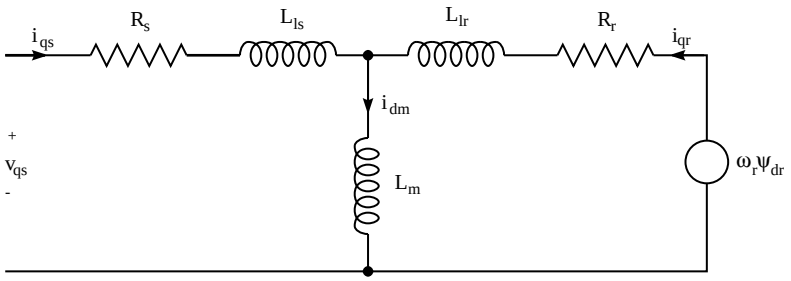


Figure 2.2: Induction machine equivalent d-q equivalent circuit. All values are referenced to the stator.

This circuit is used to find expressions for the rotor flux as a function of the stator currents. For both axes, the flux is given by,

$$\psi_r = L_m i_m + L_{lr} i_r \tag{2.2}$$
$$= L_m i_s + L_r i_r, \tag{2.3}$$

where $L_r = L_{lr} + L_m$ is the rotor self-inductance. The voltages around the rotor loop also provide the equations,

$$\frac{d\psi_{dr}}{dt} + R_r i_{dr} + \omega_{sl}\psi_{qr} = 0 \tag{2.4}$$
$$\frac{d\psi_{qr}}{dt} + R_r i_{qr} - \omega_{sl}\psi_{dr} = 0 \tag{2.5}$$

where ω_{sl} is the slip frequency. However, for the case of vector control, $\psi_{qr} = 0$, and ψ_{dr} is a constant. These equations then simplify to,

$$R_r i_{dr} = 0 \tag{2.6}$$
$$R_r i_{qr} - \omega_{sl}\psi_{dr} = 0 \tag{2.7}$$

Equation (2.3) may then be used to obtain results relating to the stator currents,

$$i_{ds} = \frac{\psi_{dr}}{L_m} \tag{2.8}$$
$$\omega_{sl} = \left(\frac{R_r}{L_m}\right)\frac{L_m i_{qs}}{\psi_{dr}} = \left(\frac{R_r}{L_m}\right)\frac{i_{qs}}{i_{ds}} \tag{2.9}$$

In addition, the torque equation from (2.1) may be expressed as,

$$T_e = \frac{3}{2}P_p\frac{L_m^2}{L_r}i_{ds}i_{qs} \tag{2.10}$$

The field oriented controller consists of these three final equations. Equations (2.8) and (2.10) are used to determine the set-point current in the rotating frame, while equa-

tion (2.9) is used to calculate the rotation of the frame with respect to the mechanical shaft speed. The use of the slip equation is the essential difference from the direct method of vector control. This estimate is used in conjunction with the shaft speed to provide the correct axis transformation into stationary coordinates.

The indirect field oriented controller has a number of requirements. These are,

- Knowledge of the shaft rotation speed. Unfortunately, this requires additional measurement hardware, and is typically provided by an incremental encoder on the shaft. As an alternative, a number of schemes have been proposed to obtain estimates of this from a soft-sensor[41], however this is still an open problem in the low speed range[28].

- The magnetising flux, ψ_m. This value is essentially a set-point, and depends on the design of the machine. For normal operation, the rated value may be used, but it may be reduced in order to gain efficiency at low torque, or to exceed the rated speed.

- The machine magnetising inductance, L_m, and rotor resistance, L_r, are required. These parameters may be estimated from tests on the machine, but they do vary according to the machine operating point. The inductance varies with the level of flux, particularly when saturation is reached, while the resistance is temperature dependent. In terms of correct operation, the ratio of the two terms, as used in the slip equation is of particular importance. If this is in error, coupling occurs between the torque and the level of magnetising flux.

This vector controller represents only a section of the machine control requirements. Outer control loops are required for the target speed or position control, and an inner loop is needed to achieve the set-point currents. The subject of this thesis is a controller for this inner loop.

2.2.3 Direct Torque Control

Direct torque control(DTC) is a strategy that was developed to remove the need for separate torque and current controllers. Instead, the cascaded control structure is replaced by a single torque controller. DTC controllers have been developed based on a number of widely differing methods This section contains an overview of three basic approaches.

Direct Self-Control

Direct Self-Control (DSC) is a direct torque control method developed by Depenbrock[14] in the mid 1980s. This method was specifically designed for high-power applications where the switching frequency had to be minimised, with a maximum rate of about 300Hz.

The basic idea of DSC is to regulate the machine flux with a feedback mechanism. The air-gap flux is estimated through the integral of the machine line-to-line voltages, on each of the three phases. Hysteresis comparators are then used to switch the output devices based on the set-point flux. A diagram of the switching scheme is shown in Figure 2.3. This figure shows the integration and switching for one phase of the inverter.

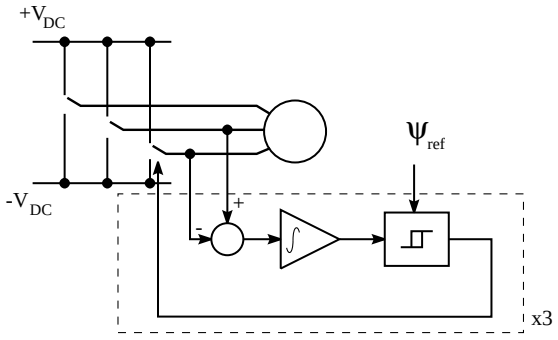


Figure 2.3: Direct self-control structure.

This is repeated for the other phases. For improved accuracy, the machine losses may be modelled. This involves subtracting the voltage across the resistance term from the terminal voltages. In particular, the low speed operation is improved by the additional modelling.

This is a very simple scheme to implement in analogue electronics. It is less suited to a digital implementation due to the added delays that would be necessary. The line voltages and fluxes over time are shown in Figure 2.4. The resulting current is not

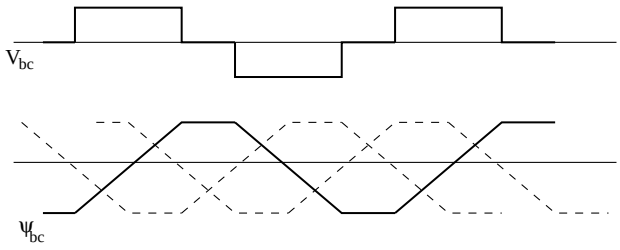


Figure 2.4: Voltage and flux waveforms for DSC.

sinusoidal, but Depenbrock claims that the harmonics are small, and unlikely to have a significant effect on the machine performance.

In this form, the controller does not control the torque or speed, but rather just the flux magnitude. Further control is added by superimposing pulse-width modulation of a zero output voltage to reduce the average voltage magnitude. For torque control, the present output torque is continuously monitored. The torque expression using the units

from [14] is,

$$T_q = \frac{3}{2}(\psi_{\alpha a} i_{\beta a} - \psi_{\beta a} i_{\alpha a}). \tag{2.11}$$

This may also be implemented in analogue hardware. Torque control is achieved by switching in the zero voltage vector when the torque exceeds the set-point by a specified hysteresis band, and then removing the zero vector when it is below the set-point.

A number of additional compensation features have been proposed to address the limitations of this controller. These include compensation for flux estimation error at lower frequencies and in the field weakening region. Overall, this design is similar in concept to the hysteresis current controllers described in Section 2.4.4. As such, it shares many of the benefits and drawbacks.

Table-Based

One of the early direct torque control strategies was proposed by Takahashi and Noguchi[44]. This approach uses a lookup table to determine the switching state. It is quite simple to implement, requiring only analogue electronics and a ROM look-up table. A diagram of the controller structure is shown in Figure 2.5.

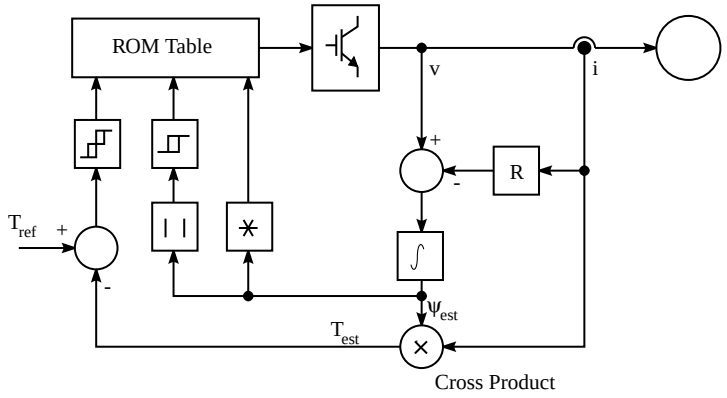


Figure 2.5: Direct torque controller.

The controller operates in a stationary 2-phase frame, with the conversions from the three-phase measurements performed by analogue adders. The machine stator flux is estimated by,

$$\psi = \int v \; dt - R_s i \tag{2.12}$$

For lower speed operation, knowledge of the winding resistance is important, as this contributes significantly to the terminal voltage.

The flux measurement is used in two ways. The angle of the flux is resolved into one of six sectors by comparison between the direct and quadrature axis flux estimates. In

addition, the flux magnitude is calculated, and compared against the desired flux in a hysteresis block. The output of this hysteresis block changes sign when the flux exceeds the normal operation bounds.

The machine torque is estimated from the cross product of the stator flux and stator current. In this case, a three level hysteresis block is used to discretise the comparison with the set-point torque. The third level allows better use of the inverter zero output voltage vector.

A number of improvements have been proposed to address the limitations of this method[24]. These provide better control of the switching frequency and ripple current. This is achieved through adding an additional controller to provide greater use of zero voltage vector. This vector is chosen subject to prediction of the rate of change of the torque. These modifications are analogous to those that have been made to hysteresis current controllers, as described in Section 2.4.4.

Constant Switching Frequency

A number of direct torque control methods have been developed that offer a constant switching frequency[17, 24, 28]. These controllers have the basic operational behaviour of a combined field-oriented controller and current controller. An early approach was described by Habetler *et al.*[17].

This method is essentially a predictive controller, where the changes in flux and torque are predicted ahead one cycle. The flux is estimated based on the integral of the terminal voltage, with optional compensation for the stator resistance voltage drop. The torque estimate is then derived from the flux and the measured current. The voltage vector for the next cycle is then calculated by solving a quadratic equation to obtain the correct flux and torque. When the inverter is incapable of supplying sufficient voltage to meet the torque demand, the basic equations have no solution. When this occurs, a table based approach is used to attempt to find the best feasible trajectory.

This type of direct torque control offers a number of advantages and disadvantages over a cascaded torque and current controller. In terms of the architecture, the main advantage is that it is possible to better handle the actuator saturation case. This is when the inverter has insufficient capacity to track the reference. However, even in the direct torque control case this involves significant complexity.

The cascaded architecture also has a number of advantages. The basic control variables, flux and torque, are closely related to the machine currents, so the control of current is a natural objective. The advantage of a separate current loop is that good current control performance may be achieved with very little knowledge of the machine parameters. This means that a fast, robust linearising loop may be achieved with current control. With the fast current tracking, the torque controller may have slower dynamics to allow for the greater parameter dependence necessary for the torque control.

A further advantage of constructing a current control loop is that the machine currents

are inherently limited by a parameter insensitive controller, reducing the possibility of an over-current condition. Both the direct torque control and cascaded approaches offer advantages, but this thesis focuses on the cascaded approach.

2.2.4 Power Converter Topology

The design of a conventional three-phase voltage source bridge converter is shown in Figure 2.6. This consists of six switching devices and associated circuitry. They are

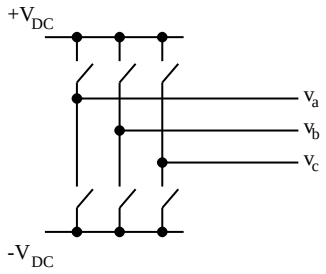


Figure 2.6: Bridge converter topology.

likely to be IGBT(Insulated Gate Bipolar Transistor) switches with free-wheeling diodes.

Although there are six switches, they are arranged in complementary pairs. At all times, there can be at most one device on in each of the three legs, otherwise a short circuit current will appear across the DC link. Normally one device is turned on, except during a switching action. While switching, it is necessary to insert a "dead-time" where neither device is turned on. This is to ensure that the first device stops conducting before the other one starts.

With three legs, there are $2^3 = 8$ possible switching states. Of these, two result in all of the outputs being at the same potential. These are the zero voltage states. The other six states provide unique non-zero output voltages.

The DC link voltage my be provided via a number of methods. The simplest is basic diode rectification of the AC power supply. The disadvantage of this approach is the distortion of the supply AC current. It also offers no return path for energy back to the supply. These shortcomings may be overcome through the use of a second bridge circuit as an active rectifier[8]. Such a rectifier may also be controlled using methods similar to the current control presented in this thesis[33].

Alternatives

An alternative to the voltage source inverter is the current source inverter. These have been less popular because of the asymmetric blocking characteristic of high frequency power devices. The advantage of a current-fed inverter is that control is easier, and regeneration possible in the popular topologies. However, it does have significant disadvantages. These include the necessity for a large inductance on the DC link, and

capacitance on both the input and output AC sides. In terms of harmonic smoothing, the inductive nature of the induction machine makes it more amenable to a voltage source inverter.

Another class of switching strategies is formed by the soft-switched converters. This style of converter is popular in switch-mode power supplies. In these converters, the switching occurs while there is zero device current, thereby reducing the switching losses. The basic approach is to use a resonant DC link (RDCL) which provides a pulsed DC link voltage and current. The zero-crossing points may then be used for switching actions, as shown in Figure 2.7.

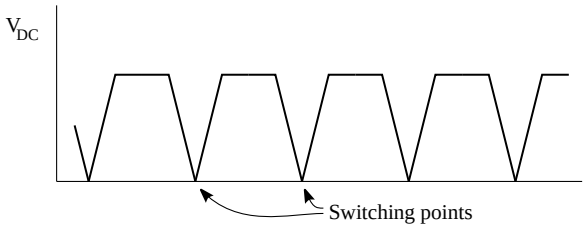


Figure 2.7: Resonant link DC voltage.

The great advantage of the resonant link converter is the reduced switching losses. This reduction improves efficiency, and reduces heat-sinking problems. The reduction of transients also improves device reliability and electromagnetic interference (EMI) properties. These attributes offer the resonant link converters a promising future[8].

The principal disadvantages of resonant link topologies are a higher harmonic content in the load, and higher peak to average ratios for the voltages and currents. The higher peaks force the devices to have higher ratings. There are also greater control restrictions, as switching events can only occur at particular times. Despite some possible advantages of resonant link approaches, this thesis concentrates on conventional voltage source converters.

2.3 Current Control Issues

2.3.1 Performance Metrics

There is a great variety of current controllers that have been developed, and each have different strengths[25]. As in most design decisions, there are trade-offs between different aspects of the performance. Some of the desirable attributes are listed here.

- Good reference tracking. The current controller should be able to control the machine current to a value that is very close to the reference specified. This may be measured in terms of steady-state amplitude and phase, as well as transient situations. Some defects, such as a linear phase error (time delay) are tolerable, as they may be overcome through prediction in the outer-loop controller.

- Good dynamics over a wide frequency and amplitude range. The purpose of adding the controller to the induction machine is likely to allow variable speed and torque operation. As such, the controller must perform well over the full range of operation.

- Well defined switching frequency. The switching devices generate thermal losses on each switching action, and must be rated for the maximum frequency of operation. A constant switching frequency is desirable, as this means that the capacity of the switching devices is well utilised across the range of operation.

- Well defined ripple current. The desired output current is typically sinusoidal, with no ripple current. However, the ripple current is a necessary consequence of a switched inverter. In most controller designs, there is a direct trade-off between the switching frequency and the current ripple. A constant switching frequency implies a varying ripple current, and a constant ripple current a varying switching frequency.

- DC link utilisation. For a given available DC link voltage, a high maximum output voltage should be achieved. This is to give the machine the widest possible range of operation within the infrastructure available.

- Machine parameter independence. Good control of the machine will require a knowledge of the dynamic parameters of the machine, such as its inductance and resistance characteristics. However, these parameters are often not well known, or are time varying. Because of this, it is desirable that these values are either estimated, or not required for operation.

- Minimal outer-loop coupling. There are benefits that may be obtained by utilising information available from the outer control loop, particularly in terms of behaviour in the saturation region. However extra coupling adds dependence on the outer controller, and propagates any errors from that loop. One example is the dependence on knowledge of the electrical rotation speed.

- Noise immunity. Inverters operate in an electrically noisy environment, and the measurements are subject to errors as a result. The controller should minimise the impact of these errors on the final output current.

- Suitability of implementation scheme. Controllers may be implemented either in analogue electronics, digital electronics, or with microprocessors. The suitability of the controller architectures varies according to the final implementation scheme. In recent times, digital schemes have become more popular due to decreasing costs, and better noise immunity. As a result, controllers suitable for digital implementation are becoming more popular.

2.3.2 Modulation Methods

In order to prevent very large converter losses, inverters can only generate a very restricted set of output voltages. The continuous range of average voltages can only be obtained by switching between the available voltage set.

The types of switching schemes may be classified as either open-loop, or closed-loop[23]. The open-loop modulation schemes provide an approximation to the desired average voltage without making use of any measurements. These typically operate with a constant switching frequency. In contrast, the closed-loop schemes typically integrate the switching strategy into the current controller. In this case, the switching events are triggered by the measured values of machine current.

An number of closed-loop schemes are described in reference to current control in Section 2.4, but this section gives a very brief overview of two popular open-loop methods.

Suboscillation Method

The suboscillation method uses a high frequency triangular carrier signal to determine the PWM switching pattern. This carrier is shared by the three phases, but apart from the common carrier, each phase is modulated separately.

The switching signal is supplied by a comparison between the carrier voltage, and the reference voltage. If the reference voltage is greater than the ramp voltage, the associated output leg is switched high, while if it is lower, the leg is switched low. This switching involves controlling both devices in the leg, and inserting appropriate dead-time between the two "on" states. The formation of the pattern is shown in Figure 2.8.

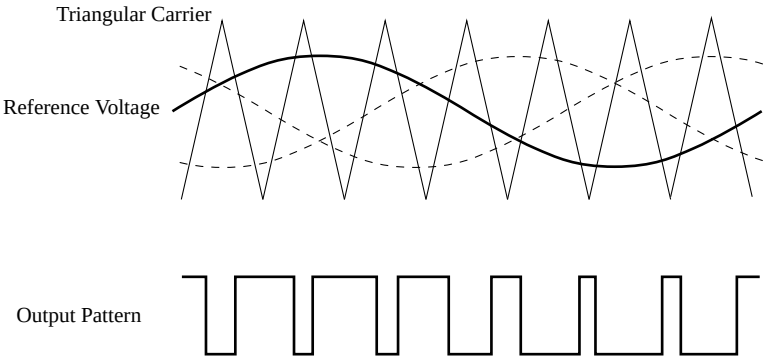


Figure 2.8: Suboscillation Method Output

This switching method was very convenient for use in analogue circuits, because it is easily implemented with analogue ramp generators and comparators. Digital techniques may also be used, but the complexity is greater. For a digital implementation, the switching times are pre-computed, and loaded into timers.

The suboscillation generates an output voltage that is linearly proportional to the reference. This may be confirmed by considering the geometry shown in Figure 2.9.

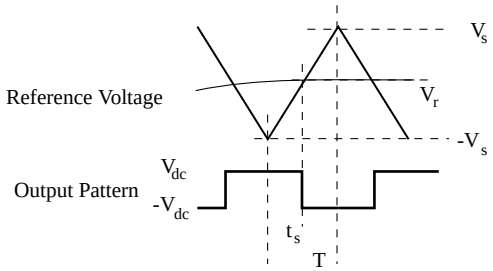


Figure 2.9: Ramp-comparison output voltage calculation.

By the triangle geometry, the switching time will be defined as,

$$\frac{t_s}{T} = \frac{V_r - (-V_s)}{V_s - (-V_s)} = \frac{V_r + V_s}{2V_s} \tag{2.13}$$

The average output voltage, V_{av} for one cycle is:

$$V_{av} = \frac{1}{T}\left(t_s V_{dc} - (T - t_s)V_{dc}\right) \tag{2.14}$$

$$= \left(\frac{2t_s}{T} - 1\right) V_{dc} \tag{2.15}$$

$$= V_r \frac{V_{dc}}{V_s} \tag{2.16}$$

Thus the average output PWM voltage is proportional to the reference voltage. In practice, there is some error in the linearity, due to switching delays, time variation in the reference, and other non-ideal phenomena. In particular, care must be taken to prevent the slope of the reference exceeding the slope of the carrier, as this can result in more switching events than expected.

One disadvantage of this approach is that it is limited to lower modulation indices. The maximum modulation index is reached when the amplitude of the reference reaches the amplitude of the carrier. This occurs at the relatively low modulation index of $\frac{\pi}{4} = 0.79$. In steady state, this problem may be overcome by adding a zero sequence voltage to distort the reference signal. This effectively moves the zero voltage point around to increase the maximum available peak voltage. However, this approach is difficult to implement in situations apart from steady-state.

Space Vector PWM

The modulation index limitation of the suboscillation method arises because each of the phases is modulated independently. The space-vector approach overcomes this through calculating the switching values of the phases as a set. This allows it to increase the maximum modulation index by optimally locating the neutral voltage with respect to the DC link potential.

The space-vector modulation technique is based on the concept of space vectors. These are single voltage or current vectors that describe the magnitudes of all of the phases. In the operation of an induction machine, these vectors will normally rotate with time, at the synchronous frequency. The construction of a voltage space vector from phase voltages is shown in Figure 2.10.

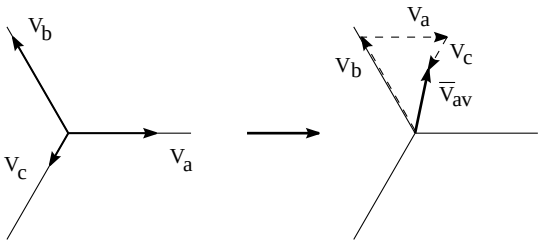


Figure 2.10: Construction of a voltage space vector.

The aim of space vector modulation is to construct the reference average voltage, $\bar{v}_{av}$ by switching between the available voltage vectors from the inverter. For the case of the standard bridge inverter, a total of seven unique voltage vectors are possible, including the zero voltage vector, as shown in Figure 2.11.

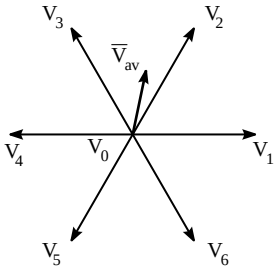


Figure 2.11: Space vectors available from the inverter.

Space vector modulation is performed across a sequence of fixed duration switching intervals. During each of these intervals, a sequence of vectors are generated such that the average voltage vector across the interval is equal to the reference. To obtain minimum current ripple, the three vectors that are adjacent to the reference are chosen to be used for the interval. This includes a zero vector, and two non-zero voltage vectors.

For the example illustrated, two possible switching sequences are shown in Figure 2.12. The specific times t_0, t_1, and t_2 are determined algebraically to obtain the correct average voltage. The patterns shown provide the same average voltage across the cycle, but the resulting ripple current will have different properties. Generally the modified pattern will have a lower current ripple for a high modulation index, while the original method will perform better for the lower indices. For the current controller that is to be presented, the standard method has the added advantage that the zero voltage vector is applied at the centre of the switching pattern. For maximum efficiency, the specific pattern can be

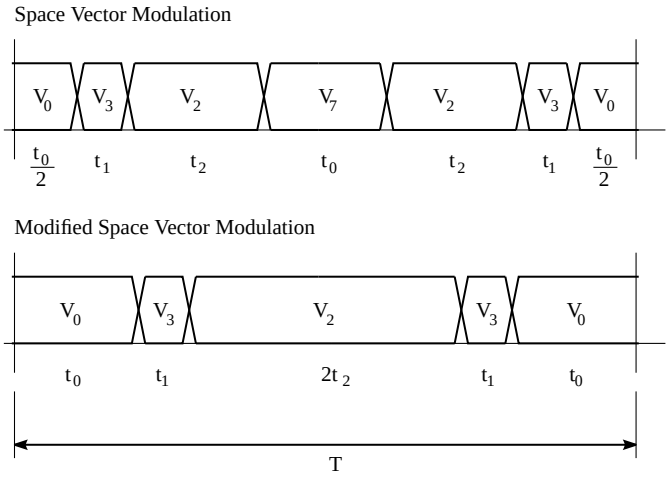


Figure 2.12: Standard and modified space vector modulation.

chosen adaptively using the reference modulation index.

For a digital implementation, the space vector modulation method has a strong advantage over the subcycle method in terms of the maximum attainable modulation index. It well utilises the inverter capacity without the added complications of the subcycle methods.

2.3.3 Inverter Dead Time

Semiconductor switching devices suffer from a range of non-ideal properties. One of these is the the turn-off delay caused by the storage effect. This is a delay that occurs between the signal to turn a device off, and the time when the device current stops flowing. This value varies according to the device itself, the current being switched and the temperature.

Under no circumstances may two devices on the same inverter leg be conducting at once, so a lock-out time is introduced between the switch-off time of one device in a leg, and the switch-on time of the other. If this delay could be exactly matched to the device's real switch-off time, there would be no impact on the performance of the inverter. However, it is not practical to estimate the delay to sufficient accuracy, so usually the worst-case plus a safety factor is allowed in the lock-out time.

The presence of a lock-out time with a safety factor means that during each switching event there will be a period where neither device in the leg will be conducting. Effectively, the output voltage is undetermined during that time, resulting in an error in the average voltage across a switching cycle.

For an inductive load, some insight can be gained into the intermediate voltage level. This may be done by considering the load to act as a current source, which will cause a current to flow through one of the free-wheeling diodes in the half-bridge. The two possible cases are shown in Figure 2.13. When the current is flowing into the inverter,

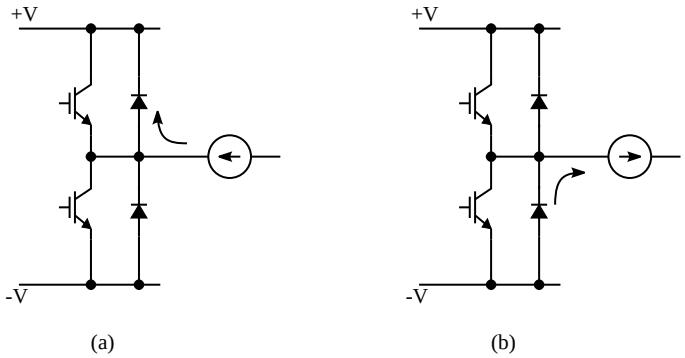


Figure 2.13: Current paths during lock-out period.

it must flow through the upper diode into the positive connection of the DC supply. Conversely, for an outward current, the other diode conducts. This means that once the active device has stopped conducting, the voltage during the lock-out time is determined by the direction of the phase current.

This knowledge may be utilised to construct a compensater for the effects of the lock-out time[23]. A significant problem is the estimation of the real turn-on and turn-off delays, particularly as they are a function of the operating point. Methods have been developed to address this[13], with the expected increase in modulator complexity.

However, the compensation schemes have problems when a switching event occurs near a zero-crossing of the current in that phase. In this case, the compensation can act in the incorrect direction, and even drive the current toward zero for a sustained period. In this situation, the worst-case error is greater with the compensation enabled than without it. This effect is evident in the results from the compensated modulators[13].

2.4 Existing Control Schemes

Many different current control schemes have been developed. These have been categorised in various ways, such as linear and non-linear[25], and optimal/non-optimal[23]. Due to the variants on each scheme, however, categorisation is difficult. In the following sections, a critique of the most important published current control techniques will be presented.

2.4.1 Stationary Frame PI Control

A basic form of induction machine current control may be obtained by applying the concepts used for DC machines. This uses a PI (Proportional-Integral) controller and a PWM modulator. The principal advantages of this method were its simplicity of implementation in analogue hardware, and the fixed switching-frequency of the switching waveforms.

The basic implementation of this method is shown in Figure 2.14. This is essentially the type of controller used on DC machines, except that in this case it is applied to

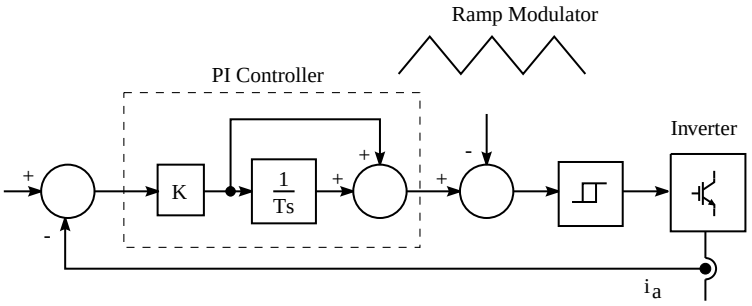


Figure 2.14: A Stationary PI Controller.

each of the three phases. It can conveniently be divided into two sections; control error calculation, and modulation of the output voltage.

To calculate the control error, the reference current is compared to the measured line current on the inverter output. This error signal is used as the input to a conventional PI regulator. The output of the PI regulator is the set-point for the average voltage to be supplied by the inverter. The PI tuning parameters may be derived by ad-hoc methods or by modelling the machine parameters.

In this controller, the ramp-comparison modulator as shown in Figure 2.15, has traditionally been used[32]. This forms the PWM switching output by an analogue comparison between the reference voltage and a generated triangular ramp waveform, as described in Section 2.3.2. This switching method is very convenient for use in analogue circuits, because it is easily implemented with analogue ramp generators and comparators.

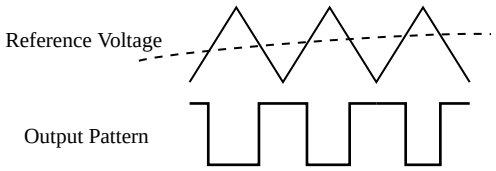


Figure 2.15: Ramp-comparison modulation.

The design of PI induction machine controllers is well known[32, 9], and quite old. The original advantage of this architecture was that it was simple to implement in analogue hardware. However, with the advent of computationally powerful and low cost digital systems, this is no longer as relevant. Analogue circuits suffered from a number of problems, such as susceptibility to noise coupling and component drift. While the static PI controller can be implemented digitally, as a digital implementation it does not have a great advantage over its competitors in simplicity.

Another problem with this scheme is that it uses three controllers when there are really only two independent current states, as $i_a + i_b + i_c = 0$. Three solutions have been used to overcome this[32]. One is to simply ignore one phase measurement, and derive the third voltage set-point from the two controlled ones. As an alternative, the

three-phase system can be transformed into an equivalent two-phase system, allowing independent control. The final option, in the case of a delta connected machine, is to synthesize a zero sequence current to decouple the three PI regulators.

These corrections detract from the simplicity of the original design, meaning that a practical system requires additional complexity. A further implementation detail would be the inclusion of anti-windup[16] on the PI regulators to prevent undesirable behaviour when control saturation is encountered.

The principal failing of stationary PI regulators for induction machine control is in performance. It has been known for some time that it inherently suffers from both amplitude and phase tracking problems[42, 40]. Performance is only acceptable when the modulation frequency is significantly higher than the harmonics present in the current and current reference signals. A value of nine times the highest significant harmonic has been suggested[46].

In addition to this, the PI controller gains need to be calculated on each machine installation. Although optimal tuning methods are available, a typical ad-hoc approach will leave the machine operating at sub-optimal performance. Performance and stability problems also occur as a result of time-variations in machine parameter values.

2.4.2 Rotating Frame PI Control

The limitations of the stationary frame PI control led to the development of PI control in rotating frames of reference[42]. The principal reason for adopting this approach was to reduce the steady-state errors inherent in the stationary frame controllers. Prior to this, the errors in the controllers had been handled using various approaches[40]. In some cases, the compensation for the behaviour of the current control was built into the torque controller. This involved significantly more complex modelling and a greater dependency on the machine and load parameters. An alternative approach was to use conventional phase lead compensation[16] in the controller. Again, this is difficult to tune when parameters are uncertain.

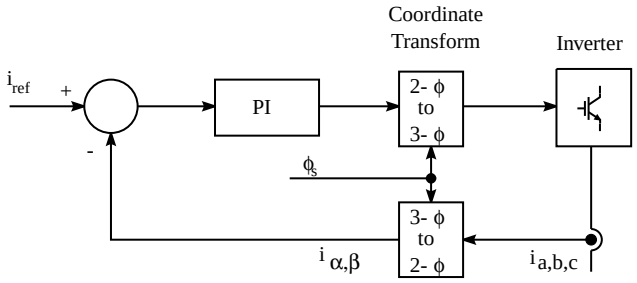


Figure 2.16: PI Regulator operating in a synchronous frame.

The basic principle is to map the measured currents from the physical stationary domain into a a synchronously rotating frame. In steady state, the reference currents

in this frame are DC quantities. Typically, the currents would also be converted to a two-phase equivalent in the frame transformation. In this case, the two phase conversion reduces complexity and avoids the problem of coupling between the phases. Once the control signal is calculated, this is transformed back into the stationary co-ordinates for PWM modulation. For an equivalent controller, the reference also requires a frame transformation, but it may instead be convenient to supply the current references in the rotating frame.

The measured currents may first be converted into a two-phase stationary frame, with the standard transformation[37]:

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix} \tag{2.17}$$

The stationary to synchronous translation may then be performed:

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \begin{bmatrix} \cos\theta_e & -\sin\theta_e \\ \sin\theta_e & \cos\theta_e \end{bmatrix} \begin{bmatrix} i_d \\ i_q \end{bmatrix}, \tag{2.18}$$

where θ_e represents the current angular displacement of the synchronous frame,

$$\theta_e = \int w_e t \; dt \tag{2.19}$$

The inverse transformations may then be used to convert the voltage back to the 3-phase quantities.

In terms of an analogue implementation, the conversion between rotating and stationary reference frames imposed significant additional complexity. Unlike the 3-phase to 2-phase conversion, multiplication was necessary, and the sin and cos unit vectors had to be generated. Although the benefits of control in the rotating frame were known in the early 1980's, the additional complexity of the transformations restricted its adoption.

Another disadvantage of this scheme over the prior stationary frame controllers was that it was more dependent on information from other controllers. It required knowledge of the current angle of the electrical rotation frame. However, this is only a small complexity problem as this information is generally available from the outer loop controllers.

Rotating Frame Control Mapped to the Stationary Frame

Following the work of Schauder and Caddy[42], Rowan and Kerkman[40] analysed the comparative performance of control in the stationary and synchronous frames. They showed that both stationary frame PI, and hysteresis controllers(§2.4.4) suffered from load dependence and steady-state error problems.

In that paper, synchronous frame regulators were proposed as the solution, and to offer superior performance over the older style regulators. The complexity issues were

also addressed by offering an alternate implementation. In this case, the rotating frame controller was re-mapped to operate in the stationary frame. This offered the performance benefits of the rotating frame without the added cost of coordinate transformations.

The resulting transformed PI controller was expressed as:

$$v_s = x_s + k\tau\left(i_s^{ref} - i_s\right) \tag{2.20}$$

$$\frac{d}{dt}x_s = k\left(i_s^{ref} - i_s\right) + j\omega_e x_s. \tag{2.21}$$

The only change from the standard PI controller is the inclusion of the $j\omega_e x_s$ term. This amounts to a compensation adjustment for the rotation of the state variable x_s over time. The resulting controller is shown in Figure 2.17.

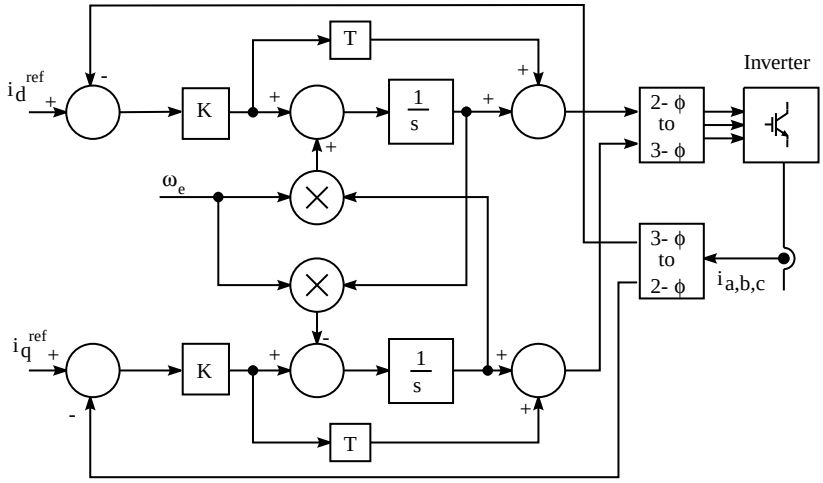


Figure 2.17: Synchronous PI controller mapped to stationary coordinates.

The advantage of performing the transformation on the controller is that rotating frame transformations are not required on the voltages and currents. Instead, only the 3-phase to 2-phase conversions are required, which are much simpler.

However, the complexity is still higher than for the simple stationary PI regulator. An additional two multipliers are required, and modified PI controllers must be used. With the minor cost in complexity comes improved steady-state regulation and load independence.

Lorenz[31] commented that the various forms of PI controller inherently use a cancellation technique. This paper states that the zero created by the controller is designed to cancel the electrical time constant of the armature. While such a cancellation works well when the parameters are exactly matched, there is a high sensitivity to parameter variations. Later work[11] shows how the pole-zero cancellation degrades as the electrical frequency increases. This degradation occurs regardless of the tuning if the synchronous frequency approaches the regulator bandwidth. The consequence of this is that there are both upper and lower bounds on the designed bandwidth.

Briz *et al.*[11] developed a modified scheme using complex vector methods to address these additional shortcomings. However, this adds a further layer of complexity to the basic PI design, and the method proposed is required to operate in the synchronous frame.

2.4.3 State Feedback

As an improvement over PI control, in either a stationary frame or synchronous frame, state-variable feedback may be implemented[25]. Lee *et al.*[29] demonstrated a state-feedback implementation in the synchronous frame. They analysed the effect of both the type of current regulator and the PWM strategy. The conclusion from their tests was that the type of error compensation had a greater impact than the choice of PWM method.

The state feedback approach involves modelling the motor in the synchronous frame as a multivariable system,

$$\dot{x} = Ax + Bu + Ed \tag{2.22}$$

$$y = Cx, \tag{2.23}$$

where

$$x = \begin{bmatrix} i_{ds} \\ i_{qs} \\ \lambda_{dr} \\ \lambda_{qr} \end{bmatrix}, \qquad u = \begin{bmatrix} v_{ds} \\ v_{qs} \end{bmatrix}. \tag{2.24}$$

A is a 4×4 matrix of machine parameters, including resistance, inductance and angular velocity components. Note that the inclusion of the velocity parameters in A renders this a non-linear model. This is unlike the constant-coefficients typically associated with state-space models. The rotor flux is considered a measurable disturbance, and included in the d term.

Within the paper, a multivariable controller was developed using pole placement. This involved developing an augmented state variable x_n including an output error state,

$$x_n = \begin{bmatrix} \dot{i}_{ds} & \dot{i}_{qs} & \tilde{y}_d & \tilde{y}_q \end{bmatrix}^T \tag{2.25}$$

where $\tilde{y}$ represents the output error $\tilde{y} = y - y_{ref}$. These additional states allow integral action in the controller, and the augmented reference will attempt to drive these additional states to zero. The state feedback is then

$$u_n = Kx_n \tag{2.26}$$

for the augmented system

$$\dot{x}_n = \hat{A}x_n + \hat{B}u_n. \tag{2.27}$$

Using pole placement, a gain controller gain matrix $K = [K_1 \; K_2]$ is found, where K_1 relates to the machine current states, and K_2 to the augmented error states. These matrices are each 2×2 in size, and depend on the machine parameters and rotational speed. As such, they are time-varying quantities.

To improve performance, additional feed-forward action is added from the reference and rotor flux disturbance. A diagram of the resulting controller is shown in Figure 2.18,

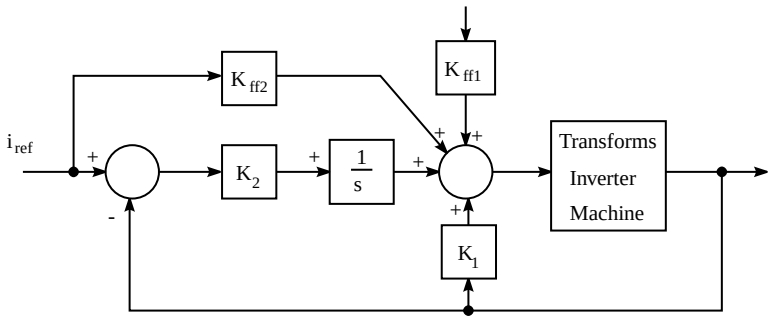


Figure 2.18: State Variable Controller.

In addition to this control action, however, the rotor flux must be estimated. In this case, a reduced order observer is used to estimate the flux based on measurements and the machine parameters.

The performance comparison presented in[29] shows that the output ripple and tracking is indeed superior to the synchronous PI controller. However, some of this gain is attributed to the use of space-vector modulation instead of the ramp-comparison method.

Overall, however, the results presented in this paper do not appear to show great improvements over the synchronous PI design. The caveat on the results are:

1. The specific results are highly dependent on the specific machine and tuning parameters used.

2. The principal advantage of the state feedback approach over the synchronous PI design is that the machine is better modelled in the controller design. However, this is only of benefit if the parameters used are well known and stable. In practice, there is no advantage in more precisely tuning a controller based on parameter values if there is substantial error in the parameter estimates.

3. In the comparison presented in [29], the greatest improvement over the PI design was in the transient response performance. In this case, the PI controller exhibited oscillation after the transient. This behaviour is consistent with the effect of wind-

up in the PI integrator, and a standard anti-windup strategy[16] should remove this defect.

4. The complexity of the state-feedback controller is much greater than that of the PI regulator.

It would appear that unless the machine parameters are very well known, that the additional complexity of the state-variable feedback approach, over the synchronous PI regulator, is difficult to justify.

2.4.4 Hysteresis Controllers

The description of hysteresis controllers in the literature dates back to the 1970's[39]. It has offered the advantage of being a very simple, robust design with good dynamic performance.

In its simplest form, each of the three phases are controlled independently. A single phase of this design in shown in Figure 2.19.

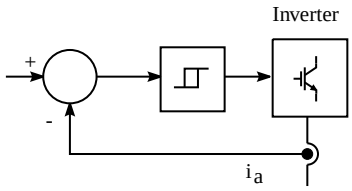


Figure 2.19: The simple hysteresis controller.

In continuous time, the measured current is compared to the reference current. The inverter is then driven in the direction that will drive the current error in that phase down in magnitude. This is really a very high gain proportional control. A hysteresis block is added to force switching instead of operating the inverter in the analogue region. The inverter also controls both devices in the leg, ensuring only one device is conducting at a time, and that appropriate dead-time is inserted.

The size of the hysteresis band controls the output current ripple and, to some extent, the switching frequency. An example of the type of output to be expected per phase is shown in Figure 2.20. This shows the regulation in one phase, ignoring the effect of the switching in the other phases.

In order to consider the combined effect of the switching in each phase, it is convenient to view a switching diagram of the current vectors[12]. The measured currents and inverter voltages are considered as vectors in a complex plane. Figure 2.21 shows an example of a reference current, i^*, a measured current i, and the error vector $\Delta i = i^* - i$.

Switching occurs when the magnitude of one of the phase currents reaches the hysteresis bounds. These bounds can be placed onto the diagram as lines perpendicular to the axis they operate in. Figure 2.22(a) shows the limits for the a-phase.

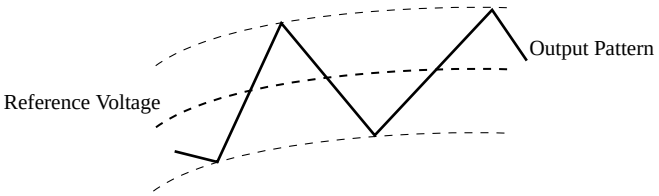


Figure 2.20: Hysteresis controlled signal

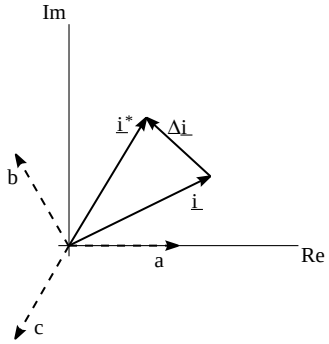


Figure 2.21: Complex current vectors for the hysteresis controller.

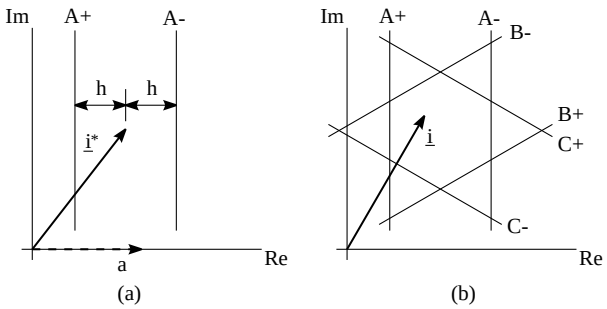


Figure 2.22: Controller switching limits.

Figure 2.22(b) includes the switching points for all three phases. The aim of the hysteresis control is to limit the output current to intersection of the regions for each phase. This corresponds to the interior of the inner hexagon.

During operation, as the current reaches a limit line, that leg of the inverter changes state. For example, if the current vector moves too far to the right in Figure 2.22(b), the "A" leg of the inverter is switched to the negative state. During normal operation, a limit cycle is reached, where the output current cycles around the interior of the hexagon.

Hysteresis Controller Behaviour

As an old control style there are many descriptions of the operating features of the hysteresis controller[12, 9, 23, 38, 35]. Holtz[23] identified four main drawbacks:

1. Because of the independent design between phases, there is no coordination to make use of the zero vectors. This causes unnecessary switching for low power outputs.
2. The tendency toward limit cycles also unnecessarily increases the switching frequency.
3. Under some conditions, the current error reaches twice the limit value before correction.
4. Sub-harmonics are generated.

The twice-limit current error occurs for a particular set of output states and current trajectories. The situation can occur when the current crosses the $A-$ boundary as shown in Figure 2.23.

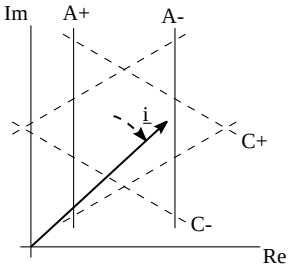


Figure 2.23: Controller error state.

If the output prior to the transition is (A+, B-, C-), the output will switch to (A-, B-, C-) as the limit is reached. However, this is the zero voltage, and in the presence of a back-emf may not be sufficient to bring the current back inside the hexagon. The current must then cross another boundary before voltage is applied.

Overall, the greatest problem with the hysteresis current controller is the unpredictability of the switching frequency. In particular, for a given implementation, the switching frequency may vary greatly depending on the load and the modulation index. Generally high and low modulation indices offer the lowest switching frequency[23]. Despite this, the hysteresis bands of the controller must be designed for the worst case switching frequency. This means that the inverter is generally not well utilised for other modulation indices.

Adjusted Switching Frequency

The primary shortcoming of the hysteresis controller may be addressed by attempting to adaptively adjust the switching frequency[6]. This is done by changing the hysteresis band limits in real time. Under normal uncontrolled operation, the frequency as a function of modulation approximately follows a curve similar to that in Figure 2.24[23].

The aim of the controllers that regulate the switching frequency is to invert this function. By suitable ripple adjustment, the output frequency may be controlled. These

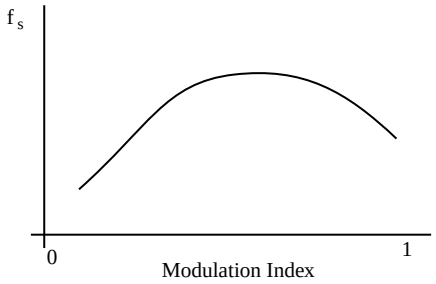


Figure 2.24: Approximate variation in switching frequency as a function of modulation index.

schemes, however, remove the primary benefits of the hysteresis controller, which are simplicity and parameter-independence.

The two basic approaches are to use either feed-forward or feedback methods. The feed-forward methods involve modelling the machine and load to calculate the size of the hysteresis band required for a particular switching frequency. As a result, the parameter dependence and calculation complexity make this approach unattractive.

The alternative is to measure the past switching frequency, and alter the hysteresis band to compensate future switching actions. This can be implemented digitally, but it is more suited to an analogue feedback loop with integration. The time constant of the update must be carefully designed. If it is too short, there will be a deterioration in the reference tracking ability of the controller. If it is too slow, it cannot perform adequate regulation. An implementation of this regulation is likely to require complex analogue hardware.

Hysteresis Control in Synchronous Coordinates

Hysteresis control has also been implemented in rotating synchronous coordinates[23]. The aim in doing this is to allow the limiting region to be rectangular in shape, as shown in Figure 2.25.

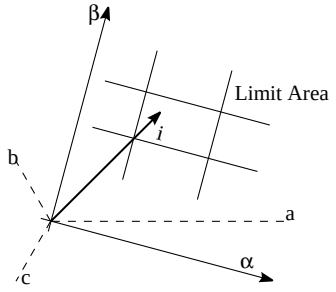


Figure 2.25: Hysteresis limits in 2-phase synchronous co-ordinates

When a synchronous frame is used in conjunction with a field-oriented torque controller, the edges of the rectangle are aligned with the rotor flux vector. This means that

greater ripple can be allowed in the flux current than in the torque current. Due to the large time constant on the rotor flux, there will not be a significant increase in the torque ripple on the machine output. Due to the larger available switching region, the switching frequency will be reduced.

In torque vector control applications, this method has advantages over the traditional hysteresis controller, but the basic hysteresis controller limitations are still present.

Coupled Hysteresis Controllers

The basic hysteresis controller has no coupling in the controller between the three controlled phases. This is the main cause of two failings of the simple method:

1. Failure to properly utilise the zero vector.
2. Excursions of up to two times the desired current ripple.

Controller schemes have been developed to address these drawbacks using coupling between the controller phases[38, 12]. Pfaff *et al.* provides a good description of the approach.

In this case, the limit detection is separated from the current error direction sensing. Comparators without hysteresis are used to determine which one of six sectors the current error is in, as shown in Figure 2.26(a). In Figure 2.26, both the star and the hexagon are centred on the reference current. The hexagon represents the maximum deviation from the reference before a switch occurs.

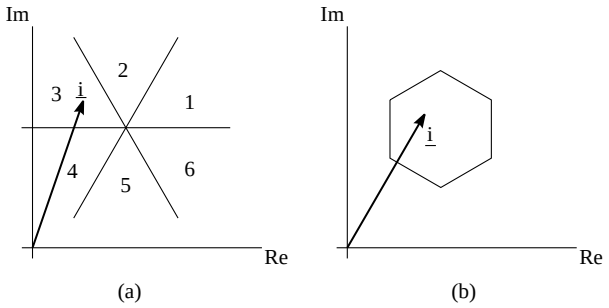


Figure 2.26: Detection functions for the coupled hysteresis controller.

The detection of the current falling outside the hexagon is the standard procedure for a hysteresis controller, but in this case, it only triggers a decision to switch, rather than a particular switching action. When the hexagon limit is exceeded alternate information is used to determine what the new output voltage should be. The sector number from Figure 2.26(a) is determined from simple comparators in each phase. This number is used in conjunction with a back-emf estimate to determine the next switching state.

When the back-emf is predicted to bring the current back into the hexagon, the zero vector is chosen. In the described implementation, a ROM lookup table was used to

determine the switching state. Additional comparators were used to detect when the edge of the hexagon, in Figure 2.26(b), was reached. When this occurred. the new switching state was applied.

This method succeeded in reducing the switching frequency of the inverter for a given current ripple[38], but significant additional estimation and prediction hardware was required to perform this. The application was also to a synchronous machine, and used control information not available in an induction machine context. Some of the more modern variants on this scheme (§ 2.4.5) achieve similar results in a more general context.

Delta Modulator

While not strictly a hysteresis controller, the Delta Modulator is closely derived from it. It could be considered to be a hysteresis controller where the output changes are forced to only occur at a periodic interval. When the controlled variable is current, these controllers are known as Current Regulated Delta Modulators, or CRΔM .

As in the case of the hysteresis controller, the CRΔM controller may be implemented independently on each phase. A diagram of the regulator for one phase is shown in Figure 2.27.

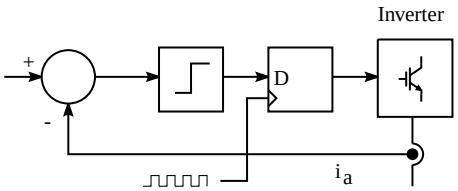


Figure 2.27: The CRΔM Controller.

The same high-gain amplifier approach is taken as the hysteresis controller, but in this case, the switching frequency is controlled by the system clock, rather than the control error bounds. This means that maximum switching frequency is better controlled, but the current ripple varies.

In order to bound the current ripple, a CRΔM controller typically needs to be designed with a relatively high clocking frequency to accommodate the largest expected $\frac{di}{dt}$. A subsequent problem is that this allows limit cycles to develop at smaller $\frac{di}{dt}$ levels. A solution to this is to retain the hysteresis block on the current error, and add the sampling to that. At low modulation indices this will increase the ripple and decrease the average switching frequency.

Overall, the delta modulator shares many of its operational characteristics with the hysteresis controller. It is simple to construct and the only tuning parameters are the clock frequency and perhaps a hysteresis band. If variations in the ripple current can be accommodated, it is quite robust to load variations[9]. Apart from current ripple, the

CRΔM scheme also suffers from strong harmonic content.

Due to the regular switching times, CRΔM ideas are suitable for soft switching in resonant link converters[8]. In this case, the switching must be synchronised with the link voltage period, and so CRΔM is well suited.

2.4.5 Predictive Controllers

A number of predictive algorithms have been proposed to combat the limitations of various control schemes. In these algorithms, a model is used to predict the behaviour of the system over a specified control horizon. Some form of optimisation is then used to select the control action. Some of the predictive schemes are basic extensions to hysteresis controllers, but the others typically involve numerical calculations based on a model of the machine. These latter controllers are suitable for digital implementation and are typically microprocessor based.

A number of authors have used predictive methods to attempt an improvement on the performance of hysteresis controllers. In this case, significant performance improvements could be expected because the hysteresis controller itself utilises little information about the system being controlled.

The main drawback is that predictive control typically carries with it a cost in controller complexity and dependence on machine parameters. So long as custom hardware and tuning is required for the implementation, there will be many applications where the limited performance requirements do not justify the additional complexity.

Predictive Hysteresis Control

The control scheme presented in Section 2.4.4 showed some elements of optimisation. This has been extended to a more general predictive scheme. Lorenz *et al.* in [32], present the architecture for this scheme.

In the same manner as the hysteresis controllers, an error boundary is constructed around the current set-point. When the measured currents cross this boundary, a new control vector is calculated. This new control vector is typically chosen to maximise the length of time before another switching action is required.

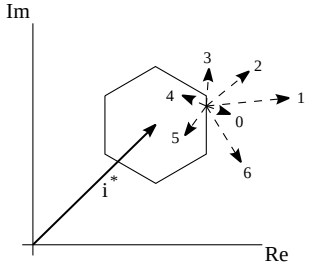


Figure 2.28: Predictive control limit boundary and estimated trajectories.

The optimisation process involves evaluating the estimated current trajectory for each of the seven possible switching states. The seven states comprise the six "on" states, together with the zero vector. According to the underlying model, an expected current vector can be calculated for each state, as shown in Figure 2.28.

An Example

A number of related methods have been used to calculate, and then select the vectors, but the method described by Nabae *et al.*[35] will now be discussed. The authors of this paper consider this to be a feedback controller, rather than a predictive controller, but this style of method is generally given the name "predictive" in the literature.

The basic control method is derived from the hysteresis controller, and uses the limit boundary depicted in Figure 2.28. When this limit is reached, the new switching state is calculated from an optimisation and prediction algorithm. The basic aim of the control is to minimise the switching frequency.

The controller derivation assumes an RLC load with equation:

$$\underline{v}(k) = L\frac{d\underline{i}}{dt} + R\underline{i} + \underline{e}_0. \tag{2.28}$$

These are vector quantities in the complex plane, with $\underline{i}$ the current, and $\underline{e}_0$ the induced voltage. When the current error is expressed as,

$$\Delta\underline{i} = \underline{i}^* - \underline{i}, \tag{2.29}$$

an approximation can be made to determine the change in the error current over time,

$$L\frac{d\Delta\underline{i}}{dt} \approx \underline{e} - \underline{v}(k). \tag{2.30}$$

Thus, the aim of the control is to choose a suitable v(k) to drive the error in the correct direction. Effectively, the hysteresis controller already does this, but the predictive controller attempts to minimise $\frac{d\Delta\underline{i}}{dt}$, subject to the constraint that it $\Delta\underline{i}$ will be decreasing.

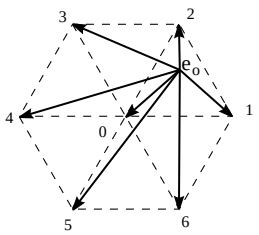


Figure 2.29: $\frac{d\underline{i}}{dt}$ vectors according to the switching pattern chosen.

Due to the limitations of the standard inverter, only the seven switching voltages are possible. These are shown in Figure 2.29 as the dotted vectors. This diagram also

shows a predicted back-emf $\underline{e}_0$. The solid vectors from the back-emf to the other vertices indicate the available $L\frac{d\underline{i}}{dt}$ vector for each of available switching states.

Under normal operation, Nabae *et al.* restrict the choice of new vectors to the three immediately surrounding the estimated back-emf. These consist of the zero vector, and the two closest active vectors, which in this case are V_1 and V_2. This is done to minimise the resulting $\frac{d\Delta\underline{i}}{dt}$ vector, hence on average increasing the time until the next switching instant.

The controller does not need to fully estimate the value of the back-emf, but instead only which of the six regions it belongs to. To aid in the detection of the region, it is assumed that the back-emf vector will lie in one of the regions adjacent to the previous switching vector. This is a reasonable assumption, as;

1. In the previous selection the vector was chosen from one of the vertices of the region containing the back-emf.

2. Given the load model, the back-emf will generally follow the direction of the applied voltage.

With a 30° coordinate transformation of the 3-phase currents, and given the above assumption, the back-emf direction may be determined via a simple comparison. The transformation used is,

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 & 0 & -1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix}, \tag{2.31}$$

The 30° transformation provides a new set of coordinate axes which are perpendicular to the possible applied voltage vectors. Being a perpendicular set, the applied voltage will be mapped to 0 in one of the three new axes. This axis is used to determine the location of the back-emf vector.

An example switching decision will now be considered. The switching decision after a $V_k = 3$ interval is depicted in Figure 2.30.

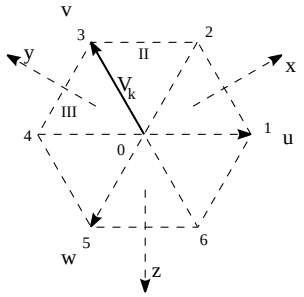


Figure 2.30: Back-emf region detection example.

By the adjacency assumption, the back-emf must be in either region II, or region III. To determine the correct region, the current in the x axis of the transformed frame is used. The component of back-emf in the x-direction may be determined by the voltage equation across the load inductance,

$$L\frac{d\Delta i_x}{dt} \approx e_x - v_x(k) = e_x. \tag{2.32}$$

The final equality is true because the x axis was chosen specifically because the previous voltage vector had a zero component in that direction. Thus, if $\frac{di_x}{dt} > 0$ across the previous switching interval, $\underline{e_k}$ is in region II, otherwise it is in region III. The same method may be applied to determine the region for other values of $\underline{V_k}$.

When the previous applied voltage vector was the zero vector ($\underline{V_k} = 0$), no prior information is known about the back-emf direction. This is not a problem, as if there is no applied voltage, the direction of $\underline{e_k}$ is easily determined by investigating the direction of $\frac{di}{dt}$. Simple sign comparisons in the rotated x, y and z phases will reveal the region.

Once the region of the back-emf is established, one of the surrounding three voltage vectors is chosen for the next switching state. This is done by selecting the one whose direction will minimise the $\Delta\underline{i}$ error.

In order to maximise transient performance, a second mode of operation is selected when the current error $\Delta\underline{i}$ is outside a second, larger error bound. In this case, no restriction is made on the vector choice, and the behaviour is more equivalent to conventional hysteresis control. The addition of this second mode provides the fast transient performance of the hysteresis controller, but with a more efficient switching behaviour in steady-state.

This controller was designed to be implemented in analogue hardware, together with a ROM for table look-up. Although at the expense of complexity, it showed a significant improvement over the standard hysteresis controller. The emphasis on using comparators in the design, however, is a short-coming. This places all of the controller gain in localised regions, causing sensitivity near the region boundaries. In particular, the back-emf estimate is chosen based on the sign of the derivative of current measurements. Measurement noise is consequently a significant threat.

The authors also addressed the switching-time variation problems of the hysteresis controller by using an adaptive feedback scheme. This integrates the deviation in past switching frequency from the desired value. The integrated value is used to adjust the current error boundaries. Although this will generally reduce the variation in switching frequency, the improvement is difficult to quantify. The harmonic spectrum is also unpredictable.

Other Hysteresis-Based Predictive Controllers

The method of Nabae *et al.* is typical of the type of extensions that have been made to hysteresis controllers. An alternative scheme[32] ignores the back-emf detection stage, and instead restricts output voltage vectors based on adjacency. After an error boundary is reached, the new vector is chosen out of either the zero vector, or one of the two vectors adjacent in angle to the previous applied one. This restriction prevents the worst-case limit cycles of the hysteresis controller and thus improves the overall performance.

An alternate approach was proposed by Malesani *el al.*[34]. The central idea of this approach is to remove interference between phases by only allowing modulation on two phases at one time, while the third is connected to one potential of the DC supply. The added advantage is that a higher modulation index is possible.

The hysteresis style predictive methods are generally best suited to analogue hardware. They have the particular advantage that the calculations are relatively straightforward to implement with analogue components. The penalty is that in the hostile environment of a drive system, analogue systems are susceptible to noise.

With the advent of wide-scale use of digital electronics in drives, the computation structure advantage of these hysteresis style current controllers is lost. Microprocessor based controllers are more suited to a short number of serial calculations, rather than the parallel structure of the hysteresis methods. The very nature of the control requires a very fast response after an error boundary is reached, so fast digital hardware is required in order to obtain equivalent performance.

For digital implementations, the double prediction method may be used. In this case, not only is prediction used once the error boundary is reached, but a prediction algorithm is used to estimate when the crossing will occur. This enables the optimal switch state to be calculated in advance.

Constant Switching Frequency Predictive Control

In the same way that predictive methods have been applied to hysteresis controllers, another class of controllers use prediction in the PI regulator style architecture(§ 2.4.1). These consist of an error regulator in conjunction with an open-loop PWM scheme.

An early description of this type of controller was proposed by Brod and Novotny[12]. This involved relatively complex modelling of the load, and the resulting controller was considered too complex to be useful. This would have been partly due to the preference for analogue electronic implementations at the time. In the same era, the same ideas were expressed by Pfaff *et al.*[38] in reference to the control of a permanent magnet machine. The early predictive controllers presented were generally complex to implement, and depended strongly on the load modelling.

In 1996, Holmes and Martin[22] described a predictive controller which is relatively simple to implement, and load independent. The strategy is similar to the one simulated by Kruker[27] in the same era. The basis of the controller is the simplified Thevenin

equivalent circuit for an induction machine. As shown in Figure 2.31, this consists of just three elements.

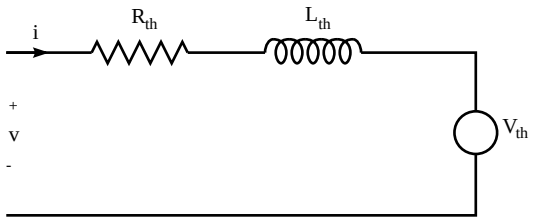


Figure 2.31: Thevenin equivalent circuit of an induction machine.

In the analysis of Holmes and Martin, the resistance term is also neglected. The analysis of this model in Section 3.2.1, and Section 4.4.2 of this thesis justifies this assumption, showing that the presence of the stator resistance has little effect on the current control.

This model was used to describe each of the three phases of the machine, but due to the interdependence of the currents, it was found only necessary to implement the controller on two out of the three phases. The alternative is to use orthogonal d-q axes, but this was dismissed as unnecessarily adding computation.

The control process consists of two stages. Firstly, the back-emf component (V_{th}) is estimated. This may be achieved by using the differential equation related to the model,

$$v = iR + L_l \frac{di}{dt} + e_b, \tag{2.33}$$

where L_l is the machine leakage inductance, and e_b the back-emf. By integrating across one control cycle of duration T, and ignoring the winding resistance, the back-emf estimate is found,

$$e_[k] = v_a v[k] - \frac{L_l}{T}(i[k] - i[k-1]). \tag{2.34}$$

Once the back-emf is found, the same equation is used to determine the voltage that must be applied to reach a current set-point. Due to the time delays incurred in calculating the control algorithm, extrapolation is used to predict-ahead the changes in the back-emf and currents. The result obtained in [22] is a more complex control update equation,

$$\begin{aligned} V^{av}[i] = &-V^{av}[i-1] + 5V^{av}[i-2] - 3V^{av}[i-3] \\ &+ \frac{L}{\Delta T}\left(3I^{ref}[i-1] - 2I^{ref}[i-2] - 6I^{meas}[i-1] + 8I^{meas}[i-2] - 3I^{meas}[i-3]\right) \end{aligned} \tag{2.35}$$

The derivation of a similar controller is shown in Chapter 3 of this thesis. The controller shown in (2.35) was trialled both in simulation and in an experimental context[22].

These showed good current tracking properties, but a real machine was not used. Instead an R-C circuit with known parameters was used for the experimental results..

Using the principles explained in Chapter 4, analysis may be performed on this control strategy. This results in the following discrete-time transfer functions for the controller and machine.

$$0 = \left(-z^3 - z^2 + 5z - 3\right) v + \frac{L^{est}}{T}\left(3z^2 - 2z\right) u + \frac{L^{est}}{T}\left(-6z^2 + 8z - 3\right) i \tag{2.36}$$

$$v = \frac{L^{mach}}{T}(z-1)i + e. \tag{2.37}$$

For the purposes of analysis, define,

$$\Delta L = \frac{L^{est}}{L^{mach}} \tag{2.38}$$

as the inductance estimate error. Solving the equations to remove the voltage terms yields,

$$\begin{aligned} 0 = {} & \frac{T}{L^{mach}}\left(-z^3 - z^2 + 5z - 3\right) e \\ & + \left(-\Delta L z^4 + (1 - \Delta L)(-6z^2 + 8z - 3)\right) i + \Delta L(3z^2 - 2z)u. \end{aligned} \tag{2.39}$$

The principles of stability analysis are discussed in Chapter 4, but it is sufficient for now to consider the magnitude of the roots of the characteristic equation. For a discrete-time transfer function system, the system is stable if all of the roots lie within the unit circle. Figure 2.32 shows the magnitude of the root locations for the controller proposed by Holmes and Martin.

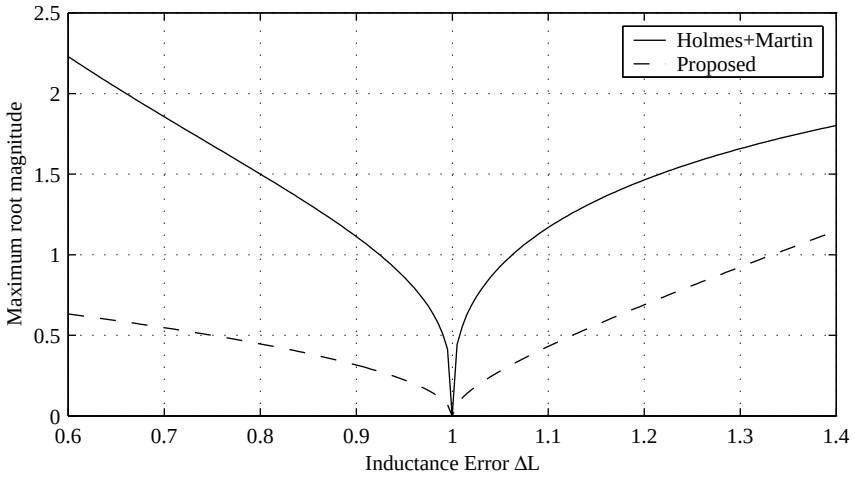


Figure 2.32: Magnitude of the transfer function poles.

The solid line indicates that an error of only 6% in the inductance estimate is sufficient to cause controller instability. The broken line describes the performance of a modified

controller, which is presented later in this thesis. The figure shows that the modifications can greatly increase the stable region.

This control technique has also successfully been applied to permanent magnet machines[43]. In this case, the stability properties were improved through using angular rotation measurements, and dead-time compensation schemes.

Stability analysis of this type of controller has been carried out as part of this project[21], and also by Malesani *et al.*[33]. The latter work was carried out in the context of an active rectifier, but reveals similar performance issues. Malesani *et al.* utilise the constant frequency nature of the PWM rectifier application to implement a filter on the back-emf estimation. An alternate approach, which is compatible with a variable-frequency operation, is presented in this thesis.

The following two chapters expand the treatment of this type of predictive controller. Some of this work has been published in the literature[4, 19, 20, 21]. A particular focus of this thesis is the suitability of this controller for implementation in direct digital hardware.

2.4.6 Artificial Intelligence Approaches

In recent years, a number of artificial intelligence techniques have become prominent. In particular, neural networks and fuzzy logic have been applied to a wide range of problems. Machine current control is not an exception, as a number of current controllers have been developed using these techniques[10, 25].

Fuzzy Logic

The origin of fuzzy logic is attributed to Lofty Zadeh in 1965[47]. The basic idea is to supplement the crisp boolean logic with a concept that is more like human thinking. Instead of numerical values, a fuzzy variable takes on values that may be expressed in natural language, such as “fast”, or “very fast”. These values are mapped to real world quantities through a membership function.

Membership functions consist of envelopes for overlapping sets. An example membership function for speed is shown in Figure 2.33. For the value noted in this figure, the

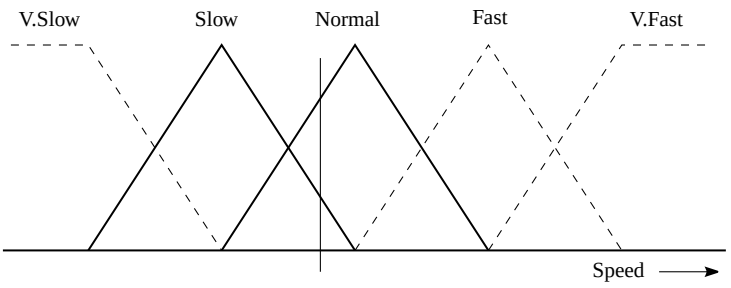


Figure 2.33: An example fuzzy membership function

two active sets are highlighted. In this case the speed is a member of both the "slow" and "normal" sets. The degree of membership of "slow" is about 0.3, and the degree of membership of "normal" is about 0.7.

Once the membership functions are evaluated for the inputs, a rule base is used to determine what response should be made. The rules are in the form of an "if-then" statement, listing the responses for various input sets. The aim of the rules is to capture the intuition of a human operator or designer.

Once the rule base has been evaluated, the response sets are transformed to output values through an inverse mapping procedure. The result is a conventional scalar quantity. Dedicated hardware exists for the evaluation of fuzzy controllers, but these actions may also be implemented in microprocessors or DSPs.

The main strength of fuzzy logic is in capturing human intuition in a problem that is difficult to model mathematically. As such, it is most suited in an outer loop supervisory capacity. Most of the successful applications of fuzzy logic have been in this role. In terms of current control, the main use has been in using fuzzy logic to tune PI controllers[25]. In this application, they have demonstrated performance improvements, but at a great expense in complexity. In the example shown in Kazmierkowski and Malesani[25], the improvement over the PI controller was restricted to a reduction in overshoot after output saturation. In this case, a conventional anti-windup strategy should perform a similar, or perhaps better role.

The performance of fuzzy controller is highly subject to the design of the fuzzy sets and membership functions, which are designed through intuition. Consequently, fuzzy logic is only a good option when conventional modelling techniques are unsuitable or difficult to apply. In the case of the current-control loop, the arithmetic model is quite well known, and the application of fuzzy logic is better suited, if at all, to the outer control loops.

Neural Networks

Neural networks are considered the most generic form of emulating human thinking[10], by attempting to simulate the function of the human brain. The normal mode of operation is as a form of fuzzy map or look-up table. The net is first trained by applying a large number of known input-to-output relationships. After training, the net may then be used to lookup the appropriate output for a supplied input.

The artificial neural network itself consists of layers of interconnected nodes, called neurons. A small example is shown in Figure 2.34. These nodes form a link of associations between the inputs and the outputs. In the simplest case, there is just input and output layers, and there must be a direct relationship between input and output. With the addition of internal "hidden" layers, more complex association patterns can be learnt.

The training of the neural network may occur entirely before the start of operation, but there are methods available for training them on-line. On-line training allows the

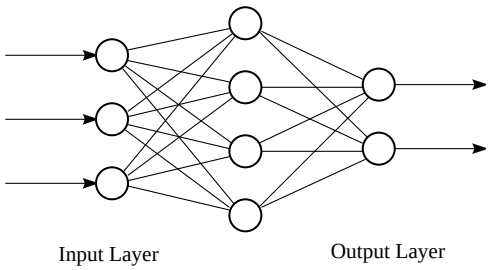


Figure 2.34: Example neural network structure.

network to adapt to changes, but suffers from the same stability problems as conventional adaptive control.

Neural networks are quite complex architectures, for both the operation itself, and the training methods. Fortunately they are produced as hardware devices, and these devices are designed to exploit the parallelism of the network architecture.

Neural networks have been used to generate PWM switching patterns in AC machine current controllers[10]. These use the current error measurements, and generate the switching times as an output. The neural network solutions appear to offer potential, but there is no evidence to suggest that they are capable of better control in this context than is achievable with algorithmic methods. The other significant problem with neural networks, which is in common with many AI techniques, is that it is very difficult to formulate any level of performance guarantee.

2.5 Conclusions

In this chapter, a number of aspects of induction machine control from the literature have been considered. There are a wide range of control strategies that have been implemented, each with their own advantages. In terms of controllers, many of the architectures were designed with a focus on an analogue implementation. With the increasing power to performance ratio of digital electronics and processors, the methods that favour digital implementation are likely to become more suitable in the future. In particular, this favours the class of predictive controllers.

The Artificial Intelligence approaches have shown to offer potential in a number of areas, although these are typically in more supervisory control roles. Due to the great difference in the philosophy behind these tools, they will not generally be considered for the remainder of the thesis.

Chapter 3

Current Control

3.1 Introduction

The basis of this thesis is the development and implementation of a predictive current controller. This chapter describes the development of controller algorithm, from modelling through to some implementation issues. The machine model is first presented, including justification for a number of simplifications in the basic model structure, comparing the tradeoffs between model complexity and accuracy.

Following this, the natural progression to a predictive control structure is made. This results in a simple control strategy, which is easy to implement, but does suffer from some limitations due to the under-modelling. In the remainder of the chapter, the extent of these limitations is investigated, together with modifications to reduce their effect.

3.2 The Predictive Controller

3.2.1 The Model

In order to design the controller, an appropriate model of the machine must be chosen. The choice of model has implications both on the performance of the controller and on the controller complexity. The full dynamic model of the induction machine, including saturation effects, would provide the most accurate representation, but the calculation complexity within the controller would be prohibitive. In particular, for low-inductance machines, and at high switching frequencies, there is only a short time available for control calculation. At a 20kHz switching rate, if half of a switching interval is allowed for the control calculation, there is only 25μs available. Unless powerful hardware is used, this is adequate for a simple control structure only.

In the following sections, the full machine model is presented, followed by a simplified representation.

The Full Dynamic Model

The full dynamic model for an induction machine consists of a set of high-order non-linear differential equations[37, 7]. The stationary frame model is expressed in Equation (3.1).

$$\begin{bmatrix} v_{ds} \\ v_{qs} \\ v_{dr} \\ v_{qr} \end{bmatrix} \begin{bmatrix} R_s + pL_s & 0 & pM & 0 \\ 0 & R_s + pL_s & 0 & pM \\ pM & -\omega_r M & R_r + pL_r & -\omega_r L_r \\ \omega_r M & pM & \omega_r L_r & R_r + pL_r \end{bmatrix} \begin{bmatrix} i_{ds} \\ i_{qs} \\ i_{dr} \\ i_{qr} \end{bmatrix} \tag{3.1}$$

where:

R_s :	stator resistance	R_r :	rotor resistance,
L_s :	stator inductance	L_r :	rotor inductance
M :	mutual inductance		
ω_r :	rotor speed, in electrical rad/s		
p :	differential operator $\frac{d}{dt}$		

The angular rotational speed ω_r is determined by the torque developed by the machine, in conjunction with the load parameters,

$$p\omega_r = \left(\frac{T_e - T_m}{J_m} \right), \tag{3.2}$$

where T_m is the external mechanical torque, and J_m the moment of inertia of the mechanical system. The electrical torque T_e is a function of the machine fluxes and currents,

$$T_e = \frac{3}{2} \left(\frac{P}{2} \right) M \left(i_{qs} i_{dr} - i_{ds} i_{qr} \right), \tag{3.3}$$

where $\frac{P}{2}$ is the number of pole pairs in the machine.

Model Reduction

In recent control strategies, this full model is rarely used for controller synthesis. Farrer and Misken proposed a simplified model in 1973[15]. This consisted of a lumped-parameter back-emf and a series leakage inductance and resistance term. The original justification for this model was under steady-state assumptions using Fourier theory.

Although commonly used in dynamic situations, little analysis of this model under transient situations appears in the open literature. The following description, based on [30], justifies model reduction in a dynamic context.

The original model in Equation (3.1) may be split into a proportional and derivative

term:

$$\tilde{v} = (pA_1 + A_2)\,\tilde{i} \tag{3.4}$$

where

$$\tilde{v} = [v_{ds}\ v_{qs}\ v_{dr}\ v_{qr}]^T, \quad \tilde{i} = [i_{ds}\ i_{qs}\ i_{dr}\ i_{qr}]^T, \tag{3.5}$$

and

$$A_1 = \begin{bmatrix} L_s & 0 & M & 0 \\ 0 & L_s & 0 & M \\ M & 0 & L_r & 0 \\ 0 & M & 0 & L_r \end{bmatrix}, \quad A_2 = \begin{bmatrix} R_s & 0 & 0 & 0 \\ 0 & R_s & 0 & 0 \\ 0 & -\omega_r M & R_r & -\omega_r L_r \\ \omega_r M & 0 & \omega_r L_r & R_r \end{bmatrix}. \tag{3.6}$$

Note that machines with squirrel cage rotors, which are the subject of consideration, will have $v_{dr} = v_{qr} = 0$. Equation 3.4 may be re-arranged into a state-space form;

$$p\tilde{i} = -A_1^{-1}A_2\tilde{i} + A_1^{-1}\tilde{v}. \tag{3.7}$$

Inspection of the A_1 matrix reveals that the direct and quadrature axis terms are decoupled, forming a block-diagonal structure in an alternate parameterisation. Thus, the matrix may be easily inverted.

$$A_1^{-1} = \frac{1}{L_s L_r - M^2} \begin{bmatrix} L_r & 0 & -M & 0 \\ 0 & L_r & 0 & -M \\ -M & 0 & L_s & 0 \\ 0 & -M & 0 & L_s \end{bmatrix}. \tag{3.8}$$

From the top row of Equation (3.4), a differential equation for the d-axis current may be found;

$$\left(L_s - \frac{M^2}{L_r}\right)\frac{di_{ds}}{dt} = V_{ds} - R_s i_{ds} + R_r \frac{M}{L_r} i_{dr} - \omega_r \left(M i_{qr} + \frac{M^2}{L_r} i_{qs}\right). \tag{3.9}$$

The equivalent equation for the quadrature axis is.

$$\left(L_s - \frac{M^2}{L_r}\right)\frac{ds_{qs}}{dt} = V_{qs} - R_s i_{qs} + R_r \frac{M}{L_r} i_{qr} + \omega_r \left(M i_{qr} + \frac{M^2}{L_r} i_{ds}\right). \tag{3.10}$$

These equations may easily be matched to a Thevenin equivalent circuit. This circuit, as shown in Figure 3.1, has behaviour modelled by,

$$v = V_{th} + R_{th} i + L_{th} \frac{di}{dt}. \tag{3.11}$$

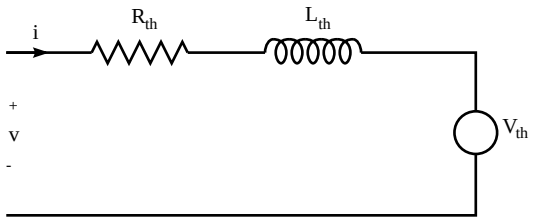


Figure 3.1: Thevenin induction machine model.

By matching the terms from (3.9) into (3.11), the following model parameters may be found for the direct axis:

$$L_{th} = L_s - \frac{M^2}{L_r} \tag{3.12}$$

$$R_{th} = R_s + R_r \left(\frac{M}{L_r}\right)\left(\frac{i_{dr}}{i_{ds}}\right) \tag{3.13}$$

$$V_{th} = \omega_r \left(M i_{qr} + \frac{M^2}{L_r} i_{qs}\right). \tag{3.14}$$

The value of L_{th} is a constant composed of a function of the inductances within the full model. A simplified approximation for the value of L_{th} may be found by parameterising with the values of leakage inductance, $L_r = L_{lr} + M$, $L_s = L_{ls} + M$, where for a normal induction machine, it is also reasonable to assume that $L_r \gg L_{lr}$ and $L_s \gg L_{ls}$.

$$L_{th} = L_{ls} + M - \frac{M^2}{L_{lr} + M} \tag{3.15}$$

$$= L_{ls} + \frac{M L_{lr}}{M + M_{LR}}. \tag{3.16}$$

This represents an equivalent circuit with L_{ls} in series with the parallel combination of M and L_{lr}. As $M \gg L_{lr}$, the effect of M in this configuration is negligible, so the Thevenin equivalent inductance is approximately the sum of the rotor and stator leakage inductances;

$$L_{th} \approx L_{ls} + L_{lr}. \tag{3.17}$$

This approximation is very accurate, and the practical limitation will not be the under-modelling, but rather the estimation of this quantity. As the inductance varies over the operating range of the machine, the exact value of this parameter of the model is subject to uncertainty.

The value of R_{th}, as expressed in (3.13) is not a constant. Instead, it varies with the ratio of rotor and stator currents as well as with temperature. As an approximation, the range of values may be established. Firstly, it may be observed that

$$\frac{M}{L_r} = \frac{M}{M + L_{lr}} \approx 1. \tag{3.18}$$

Furthermore, the ratio of currents varies between 0 and -1, depending on the load conditions. Under light load, most of the current passes though the magnetising leg, resulting in $i_r/i_s \approx 0$ Under a blocked-rotor condition, most current flows through the rotor, resulting in $i_r/i_s \approx -1$. So in practice,

$$R_s < R_{th} < R_s + R_r. \tag{3.19}$$

As in the case of the inductance, the resistances R_s and R_r also vary with the operating point of the machine. The total variation in the R_{th} parameter is a limitation of the model, but in current control applications, the effect of the variation is not great on the behaviour of the model. This allows $R_{th} \approx R_s + R_r$ to be an adequate approximation for all useful operating ranges[30]. While the estimation of the rotor resistance is important in controlling the rotor current, as in vector control, it is not so significant in describing the stator terminal conditions.

For the d-axis, the $L_{lr} \ll M$ assumption may be used to find an approximation for the Thevenin voltage;

$$V_{th} \approx \omega_r M \left(i_{qr} + i_{qs}\right) = \omega_r \psi_q. \tag{3.20}$$

This product of the rotational speed and the quadrature axis flux represents the motor's back-emf. Under normal conditions, as the machine rotates the vector describing the back-emf will rotate at synchronous speed. So in steady-state, this will be a sinewave of known period in each of the d and q axes.

The nature of the transient behaviour may be investigated by considering the variation in each of the components. Because of the relatively high value of the magnetising inductance, the magnetising current $i_{qr} + i_{qs}$ can only change slowly. Similarly, the shaft angular velocity is constrained to slow variations due to the long mechanical time constants. As the current controller response time is concerned with the dynamics associated with the leakage inductance of the machine, it is a reasonable approximation to assume that the back-emf will change slowly compared to the controller bandwidth.

The following is a summary of the simplified model parameters;

$$L_{th} \approx L_{ls} + L_{lr} \tag{3.21}$$

$$R_{th} \approx R_s + R_r \tag{3.22}$$

$$V_{thd} \approx \omega_r M \left(i_{qr} + i_{qs}\right) \quad V_{thq} \approx -\omega_r M \left(i_{dr} + i_{ds}\right). \tag{3.23}$$

Model Selection

As the desired objective is current control, the machine model need only be accurate in describing the machine phase currents. This is in contrast to a torque controller, for example, where the model must include a set of parameters defining the torque generation, such as the rotor currents.

The simplest model that provides an accurate representation of the machine currents is shown in Figure 3.2. This model consists of the leakage-inductance L_l in series with a back-emf e_x in each phase.

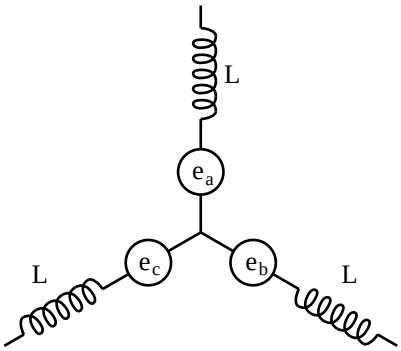


Figure 3.2: The three-phase induction machine model

While the back-emf is time-varying, for the purposes of control, it is reasonable to consider it a constant over a short time period. In the previous analysis, the back-emf was shown to be the product of the machine flux and rotational velocity, which are both slowly-changing.

The rotor and stator winding resistances are also incorporated into the lumped back-emf quantity. Better performance could be obtained by adding the extra parameters, but this also introduces significant additional complexity. Apart from the calculations involved, effective estimation of the parameters is difficult. Unless the estimation is sufficiently good, little benefit is likely to be gained by accounting for the parameters.

For the purposes of control, it is convenient to convert the three phase model in figure 3.2 to the two phase model, shown in figure 3.3. The two phase model has the advantage of having decoupled axes, allowing independent scalar calculations in each of the two dimensions. Throughout this thesis, the power-variant transformation is used between the two coordinate systems. This transformation has the advantage of maintaining the same voltage, current and impedance values across the transformation. The torque and power expressions require a $\frac{3}{2}$ scaling factor.

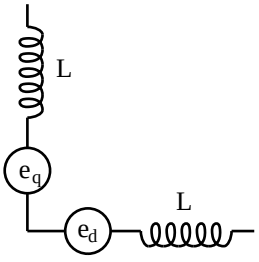


Figure 3.3: The two-phase equivalent induction machine model

Provided that there is no neutral current ($i_a + i_b + i_c = 0$), the machine can always

be represented using the two-phase model. The translation between the two coordinate systems is achieved through the expressions:

$$\begin{aligned} 3i_d &= 2i_a - i_b - i_c \\ \sqrt{3}i_q &= i_b - i_c. \end{aligned} \tag{3.24}$$

The reverse transformations are:

$$\begin{aligned} i_a &= i_d \\ 2i_b &= -i_d + \sqrt{3}i_q \\ 2i_c &= -i_d - \sqrt{3}i_q. \end{aligned} \tag{3.25}$$

It is convenient to use a scaled parameterisation of i_q, where $\hat{i}_q = \sqrt{3}i_q$ throughout the current control algorithm. In the final equations for the q axis, all of the current measurements are accompanied by a $\sqrt{3}$ multiplier. This means that the controller may be implemented without any multiplications by $\sqrt{3}$, except for the reference current.

The two-phase machine model, as depicted in Figure 3.3, may be described by a pair of two simple differential equations:

$$v_d - e_d = L_l \frac{di_d}{dt} \tag{3.26}$$

$$v_q - e_q = L_l \frac{di_q}{dt}. \tag{3.27}$$

For a constant v and e, $\frac{di}{dt}$ is a constant, so the current will be a linear segment. For the case of PWM switching, the applied voltage will consist of step changes between constant fractions of the DC link voltage. If e is assumed to be constant, the current waveform will be made up of piecewise linear segments.

The result of a current trajectory made up of linear segments is very useful because it provides an opportunity for simple control calculations. It is not necessary to evaluate any transcendental functions in order to predict the path of the current over a control cycle.

3.2.2 Output Switching Pattern

Due to the switched nature of the inverter outputs, a knowledge of the inverter switching pattern is necessary in order to effectively utilise the current measurements. Space-vector modulation is used for the PWM output in order to maximise the utilisation of the DC link, and for the simplicity of calculation in a digital implementation.

Using space-vector modulation with a double-edged switching pattern, each output is made up of three different output vectors in sequence, arranged in a symmetric pattern. One of these vectors is the zero voltage vector, which physically corresponds to all of the outputs being switched to the same DC link potential. This potential may either be the

positive or the negative rail of the DC link. The zero vector appears at the start, end and centre of the switching pattern. The two non-zero vectors appear for a calculated time between these points. An example trace of two-phase voltages and currents appear in Figure 3.4.

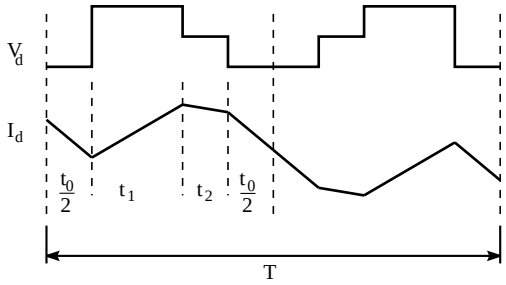


Figure 3.4: The two-phase machine voltage and current.

Measurements of the current taken at the beginning, centre, and end of the switching pattern are all easily predictable as they are equal to the currents achieved if a constant voltage, of the same average, is applied. This is because the flux, and hence the current, through the inductor is the integral over time of the voltage applied across its terminals.

Current values at other points in the cycle are heavily dependent on knowledge of the precise switching pattern. As a consequence, a complex model is needed if any instantaneous current samples, apart from those at the centre and ends, are to be used. Instantaneous measurements at these other points are also susceptible to short-term effects on the current caused by switching. Capacitive effects are likely to cause spikes in the current, resulting in noisy measurements.

For a digital controller there is a design tradeoff in the number of measurement samples to use per control interval. In this case the easiest approach is to use only two measurements in each control cycle, one at the start, and the other at the centre. The alternative is to also use current measurements made at intermediate points. The intermediate points incur the cost of a more complex model along with less reliable measurements. The measurements are less reliable as they may coincide with the instants when devices switch, whereas the PWM generator may be designed to never switch at the centre and end points.

The advantage is that reliance on only two measurements per cycle renders the controller more susceptible to noise on those particular measurements. The use of more measurements, along with filtering methods, reduces the sensitivity of the controller to each measurement.

The approach behind the controller presented in this thesis is to use a simple model and only use a small number of samples. The samples used, however, are those made during the zero voltage vectors, which are the most reliable. The exception to this is in inductance estimation, where intermediate values are also used. In this case, heavy filtering is used, so the level of noise on the measurements is less relevant.

3.2.3 Back-emf Estimation

In order to use the two-phase model shown in Figure 3.3 to determine control voltages, it is necessary to know the model back-emf. On each control interval, the back-emf must be estimated from the past behaviour of the model. The previously applied voltage and measured currents may be used to generate an estimate for e_d and e_q. The voltage-current relationship for a single phase may be found by considering the average voltage across the leakage inductance,

$$v_L(t) = L_l \frac{di}{dt} \tag{3.28}$$

$$i(t_k) = i(t_{k-1}) + \int_{t_{k-1}}^{t_k} \frac{1}{L_l} \left(v(t) - e(t)\right)\ dt \tag{3.29}$$

$$= i(t_{k-1}) + \frac{T}{L_l}\left(\frac{1}{T}\int_{t_{k-1}}^{t_k} v(t)\ dt - e\right) \text{ when } e \text{ is constant.} \tag{3.30}$$

The integral involving v is simply the average voltage across the interval, and is denoted v_k. This results in the following simplified equation relating the estimated back-emf to the measured quantities,

$$v_k - e_k = \frac{L_l}{T}\left(i_k - i_{k-1}\right) \tag{3.31}$$

$$e_k = v_k - \frac{L_l}{T}\left(i_k - i_{k-1}\right). \tag{3.32}$$

Equation (3.32) is an estimate of the back-emf across the time period between the measurements i_{k-1} and i_k. T is the length of this time interval. This measurement inherently involves differentiation, and so it is sensitive to noise on the measurements. The sensitivity is reduced with increased values for T, but this introduces a greater time delay in the measurement.

3.2.4 The Controller

The aim of the controller is to determine the inverter voltage v_{k+1} required to achieve the setpoint current u_{k+1} on each cycle. For constant frequency PWM, it is necessary to know the voltage at the start of the switching cycle. This is because the output must switch immediately from the zero vector at the start of the cycle if the maximum voltage is required. As the current measurement at the end of the previous cycle (i_k) is not available until after the next has started, this measurement cannot be used in the control calculation. Instead this value is estimated from the previous measurements of the current. Because of the symmetric switching pattern, the average applied voltage is the same over the second half of the switching cycle as over the first half. For a constant back-emf, the change in current over each of the halves of control interval is the same according to the $e{-}l$ model. Figure 3.5 shows this extrapolation by using a broken line

to denote the estimated idealised current trajectory for the second half of the control cycle.

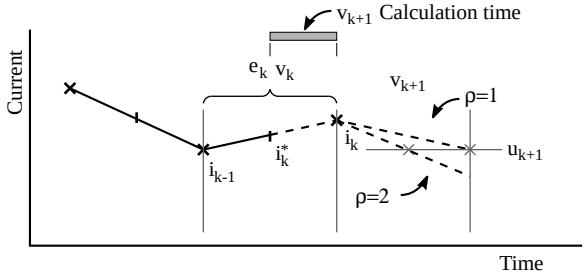


Figure 3.5: The variables used in the controller..

As some time is required to evaluate the control algorithm, only measurements up to and including the midpoint current, i_k^*, are used for calculating the inverter voltage $v_k + 1$. This leaves half of one switching cycle to perform the control calculations. If the measurement i_{k+1} were to be used, there would be no time available from the measurement to when the control has to be implemented. Due to the delay in obtaining the measurement through the data acquisition communications channels, and the control calculation time, a significant delay is necessary between the last measurement taken, and the first possible application of the control.

The current at the start of the interval is estimated by the equation,

$$\begin{aligned} i_k^{est} - i_k^* &= i_k^* - i_{k-1} \\ i_k^{est} &= 2i_k^* - i_{k-1}. \end{aligned} \tag{3.33}$$

The back-emf may be estimated from the i_{k-1} and i_k^* measurements using (3.32) to give,

$$e_k^{est} = v_k - \frac{2L_l}{T}\left(i_k^* - i_{k-1}\right). \tag{3.34}$$

The model equation (3.31) may be re-arranged to determine the voltage required for a given setpoint current of i_{k+1},

$$\begin{aligned} v_{k+1} &= e_{k+1}^{est} + \frac{L_l}{T}\left(i_{k+1} - i_k^{est}\right) \\ &= e_{k+1}^{est} + \frac{L_l}{T}\left(i_{k+1} - 2i_k^* + i_{k-1}\right). \end{aligned} \tag{3.35}$$

The value used for i_{k+1} does not always equal the input current setpoint. This is only the desired current at the end of the cycle. In order to obtain better steady-state accuracy, it could be desirable to instead try to achieve the correct average current across the cycle. In this case, the setpoint current should equal the average of the current measurements at the start and the end of the switching cycle.

Average current control may be achieved by setting $i_{k+1}^* = u_{k+1}$ where u_k is the

current setpoint. The choice between average and endpoint control may be parameterised. For this purpose, ρ is defined as:

$$T^* = \frac{T}{\rho}, \tag{3.36}$$

where T^* is the length of time from the start of the interval to when the output current is projected to match the setpoint. By appropriate choice of ρ, either average or endpoint control may be achieved,

$\rho = 1$: Endpoint control

$\rho = 2$: Average current control .

Alternate values for ρ are also valid. Values between 1 and 2 result in controllers that have behaviour between average and endpoint control. By substituting (3.36) into (3.35), with T^* in place of T, a parameterised controller equation is achieved,

$$v_{k+1} = e^{est}_{k+1} + \frac{\rho L_l}{T}\left(u_{k+1} - 2i^*_k + i_{k-1}\right). \tag{3.37}$$

Equations (3.34) and (3.37) together comprise the current controller. These may be combined to form a single control equation,

$$v_{k+1} = v_k - \frac{2L_l}{T}\left(i^*_k - i_{k-1}\right) + \frac{\rho L_l}{T}\left(u_{k+1} - 2i^*_k + i_{k-1}\right) \tag{3.38}$$

$$= v_k + \frac{L_l}{T}\left(\rho u_{k+1} - 2(\rho+1)i^*_k + (\rho+2)i_{k-1}\right). \tag{3.39}$$

Substitution may be performed for ρ to obtain the final control update equation. For endpoint control, this is:

$$v_{k+1} = v_k + \frac{L_l}{T}\left(u_{k+1} - 4i^*_k + 3i_{k-1}\right), \tag{3.40}$$

while for average current control, it is

$$v_{k+1} = v_k + \frac{2L_l}{T}\left(u_{k+1} - 3i^*_k + 2i_{k-1}\right). \tag{3.41}$$

These equations are computationally simple and quite suitable for implementation in a digital system.

3.3 Output Considerations

The control algorithm as presented calculates an average voltage to be applied over the next control cycle. This section describes the calculation of inverter switching times from the desired two-phase voltages. This involves both the two-phase to three phase transformation, along with obtaining the parameters for the space-vector modulation.

3.3.1 PWM Generation

Figure 3.6 shows the inverter topology and the voltage vectors it will generate. Each leg may be in one of two states. In the low state, the output voltage v_x is equal to the negative side of the DC input, while in the high state it is equal to the positive side. For the high state, the top device is turned on and the bottom device is turned off, and the converse is true for the low state. As each of the three legs may be in one of two states, there are $2^3 = 8$ possible vectors. The switching states required for each of the vectors is

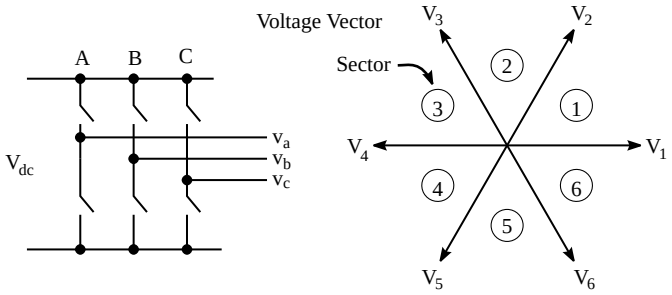


Figure 3.6: The inverter configuration and output voltage vectors achievable.

shown in Table 3.1. For each of the legs (A,B and C), a zero indicates that the bottom device is on, while a one indicates that the top device is on.

Vector	Devices (ABC)	2-Phase Model Voltage	
		v_d	v_q
V_0	000	0	0
V_1	100	V	0
V_2	110	$\frac{1}{2}V$	$\frac{\sqrt{3}}{2}V$
V_3	010	$-\frac{1}{2}V$	$\frac{\sqrt{3}}{2}V$
V_4	011	$-V$	0
V_5	001	$-\frac{1}{2}V$	$-\frac{\sqrt{3}}{2}V$
V_6	101	$\frac{1}{2}V$	$-\frac{\sqrt{3}}{2}V$
V_7	111	0	0

Table 3.1: Switching states required for each vector.

For each of the vectors, the equivalent two-phase applied voltage may be calculated by taking the d and q axis components. The results for this are shown in table 3.1. There are two zero vectors, 000 and 111. These correspond to when all of the legs of the inverter are switched to the same state. For 000, this is to the negative DC link potential, while with 111 the output is connected to the positive side. The other six unique vectors represent the voltages shown in figure 3.6, that may be applied to the machine.

Inverter voltages, apart from the six fundamentally available, may be achieved via Pulse Width Modulation. Using this method, time is divided up into periods of time equal to the switching period (T). During each interval a number of vectors are applied

to the inverter in a such a way that the average voltage vector across the interval is equal to the desired voltage.

In this implementation, a double edge modulation firing pattern was chosen, as this minimises the harmonics generated by the inverter.

The average voltage produced by the inverter across a cycle is equal to the expression,

$$V_{av} = \frac{1}{T}\int v(t)\ dt \tag{3.42}$$

$$= \frac{2t_0}{T}v_0 + \frac{2t_1}{T}v_1 + \frac{2t_2}{T}v_2 \tag{3.43}$$

under the constraint,

$$2\left(t_0 + t_1 + t_2\right) = T. \tag{3.44}$$

The vector v_0 is the zero vector, which may be implemented with either V_0 or V_7. The other two vectors, v_1 and v_2 are the vectors closest to the desired average output vector. For vectors with a phase between 0 and $\frac{\pi}{3}$, these are the vectors V_1 and V_2. This region is called sector 1. More generally, the vector $Ve^{j\phi}$ is in sector n, where $\frac{\pi}{3}(n-1) < \phi < \frac{\pi}{3}n$.

In each sector, the required switching times may be calculated by solving equations (3.43), (3.44) and the two-phase decomposition expressions in table 3.1. For sector 1 with V_1 applied for time t_1, and V_2 for time t_2, we have the the equations,

$$V_d = \frac{2t_1}{T}\left(V\right) + \frac{2t_2}{T}\left(\frac{1}{2}V\right) \tag{3.45}$$

$$V_q = \frac{2t_1}{T}\left(0\right) + \frac{2t_2}{T}\left(\frac{\sqrt{3}}{2}V\right) \tag{3.46}$$

$$T = 2\left(t_0 + t_1 + t_2\right). \tag{3.47}$$

These may be solved to determine the required switching times from the d and q axis setpoints,

$$t_1 = \frac{T}{2V}\left(V_d - \frac{V_q}{\sqrt{3}}\right) \tag{3.48}$$

$$t_2 = \frac{T}{2V}\left(2\frac{V_q}{\sqrt{3}}\right) \tag{3.49}$$

$$t_0 = \frac{T}{2} - t_1 - t_2. \tag{3.50}$$

The bounds for the valid inputs (V_d, V_q) may be determined by checking for valid values for t_0,t_1 and t_2. Negative values for any of these are not feasible as it is obviously not possible to implement such an interval. There is also an upper limit on the length of an interval, which is essentially $t_1 + t_2 < \frac{T}{2}$. This limit is implicitly enforced if (3.44) and the non-negativity constraints are met.

If the bound on T_0 is not met, the cause is that the magnitude of the vector is too great. This means that a voltage vector has been requested that cannot be achieved with the given supply voltage. The solution to this is to re-scale the vector, as described in section 3.3.2. If the non-negativity constraints for t_1 or t_2 are not met, the cause is that the wrong sector was chosen for the calculations. This result may be used to develop a set of rules which determine the correct sector from the V_d and V_q values.

By solving equations (3.48) and (3.49) for $t_1 > 0$ and $t_2 > 0$, the following relationships for sector 1 are obtained,

$$V_q > 0 \tag{3.51}$$

$$V_d > \frac{1}{\sqrt{3}} V_q. \tag{3.52}$$

For sector 2, the two adjacent vectors are V_2 and V_3. For greatest smoothness across the sector transitions, V_2 may be applied for the t_2 period, just as it was for sector 1. This leaves V_3 to be applied over t_1. For this allocation, the voltage equations for sector 2 are:

$$V_d = \frac{2t_1}{T}\left(-\frac{1}{2}V\right) + \frac{2t_2}{T}\left(\frac{1}{2}V\right) \tag{3.53}$$

$$V_q = \frac{2t_1}{T}\left(\frac{\sqrt{3}}{2}V\right) + \frac{2t_2}{T}\left(\frac{\sqrt{3}}{2}V\right) \tag{3.54}$$

$$T = 2\left(t_0 + t_1 + t_2\right). \tag{3.55}$$

The switching time solutions derived from these equations are,

$$t_1 = \frac{T}{2V}\left(-V_d + \frac{V_q}{\sqrt{3}}\right) \tag{3.56}$$

$$t_2 = \frac{T}{2V}\left(V_d + \frac{V_q}{\sqrt{3}}\right) \tag{3.57}$$

$$t_0 = \frac{T}{2} - t_1 - t_2. \tag{3.58}$$

By solving (3.56) and (3.57) for $t_1 > 0$ and $t_2 > 0$, the sector 2 conditions are obtained,

$$\begin{aligned} \frac{1}{\sqrt{3}} V_q &> V_d \\ \frac{1}{\sqrt{3}} V_q &> -V_d. \end{aligned} \tag{3.59}$$

The determination of switching times may also be achieved through vector addition. Over each interval, the overall applied average voltage is a weighted sum of the three vectors that are applied over the vector, as shown in equation (3.43). As the zero vector makes no contribution to the sum, the average is made up of the sum of the two non-zero vectors. The direction of these vectors is fixed, but the magnitude may be scaled by

altering the switching times. The switching times may be chosen by selecting the lengths necessary to achieve the correct vector sum. The geometry behind this, for sector 1, is shown in figure 3.7.

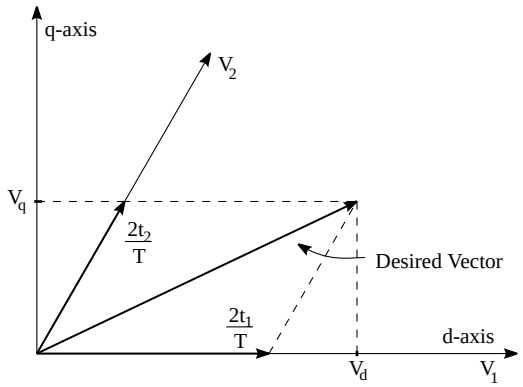


Figure 3.7: Switching times determined through vector addition.

V_1 and V_2 represent the two non-zero output voltage vectors used in sector 1. The vectors v_d and v_q are fixed, as are the directions of V_1 and V_2. The scalars t_1 and t_2 are set so that the vector sums from the two coordinate systems are equal,

$$\frac{2t_1}{T}V_1 + \frac{2t_2}{T}V_2 = V_d + V_q. \tag{3.60}$$

This may be achieved graphically in sector 1 by first setting t_2 to obtain the correct q-axis voltage, and then t_1 to obtain the correct d-axis voltage. The result is the same as that calculated above, (3.59). The switching times and conditions for each sector are given in table 3.2. These values depend on the choice of vector order so an example choice of ordering is given in the table. An order of 01277210 means that vector V_0 is applied initially for time $\frac{t_0}{2}$, followed by V_1 for t_1 and then V_2 for time t_2.

Sector	Firing Order	$\frac{2V}{T}t_1$	$\frac{2V}{T}t_2$	Conditions
1	01277210	$V_d - \frac{V_q}{\sqrt{3}}$	$2\frac{V_q}{\sqrt{3}}$	$V_d > \frac{V_q}{\sqrt{3}}; V_q > 0$
2	03277230	$-V_d + \frac{V_q}{\sqrt{3}}$	$V_d + \frac{V_q}{\sqrt{3}}$	$V_d < \frac{V_q}{\sqrt{3}}; \frac{V_q}{\sqrt{3}} > -V_d$
3	03477430	$2\frac{V_q}{\sqrt{3}}$	$-V_d - \frac{V_q}{\sqrt{3}}$	$V_d < -\frac{V_q}{\sqrt{3}}; V_q > 0$
4	05477450	$-2\frac{V_q}{\sqrt{3}}$	$-V_d + \frac{V_q}{\sqrt{3}}$	$V_d < -\frac{V_q}{\sqrt{3}}; V_q < 0$
5	05677650	$-V_d - \frac{V_q}{\sqrt{3}}$	$V_d - \frac{V_q}{\sqrt{3}}$	$V_d < -\frac{V_q}{\sqrt{3}}; \frac{V_q}{\sqrt{3}} < V_d$
6	01677610	$V_d + \frac{V_q}{\sqrt{3}}$	$-2\frac{V_q}{\sqrt{3}}$	$V_d > -\frac{V_q}{\sqrt{3}}; V_q < 0$

Table 3.2: PWM switching times and sector conditions.

As vectors may not be applied for a negative time, all of the quantities t_0, t_1 and t_2 must be positive. In the theroetical analysis, zero-duration times are feasible, but may not be practical for the PWM hardware. In the implementation described in the next chapter, a constraint that $t_0 \geq 2T_s$ is applied, where T_s is the system clock period ($0.2\mu s$ for the reference implementation).

3.3.2 Voltage Limiting

The above switching time calculations do not take into account the situations when the desired average voltage cannot be achieved by the inverter. In particular, voltages greater than the DC supply voltage cannot be applied to the inverter output. When the setpoint voltage is too large, it must be adjusted to a feasible value in order to generate the correct switching times. This adjustment may be made at a number of places. These include:

1. The torque controller,
2. Current controller,
3. PWM generation.

The earlier the correction is made, the harder it is to make. In the case of the torque controller it would be necessary to know what the maximum possible torque setpoint is without saturation. This means solving the torque and current control equations, together with the output switching equations, at the boundary of saturation. The advantage of this, however, is that the effect of the saturation will be minimised.

As computational complexity is to be minimised, the following section presents the case where the saturation is implemented at the output of the current control. Even when a more sophisticated scheme is implemented, this level may also be desirable to overcome problems due to round-off errors. If the torque controller compensates for the output limitations, subsequent round-off may cause the corrected output to still lie marginally outside the feasible region. Additional saturation at the output will have little effect on the vectors generated, but will prevent any switching logic errors.

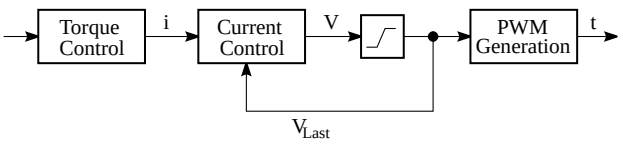


Figure 3.8: Saturation at the output of the current controller.

Figure 3.8 shows the location of the saturation in the controller. The voltage vectors produced by the current controller are subjected to saturation after the current control calculation. A feedback path also exists back into the current controller. This is needed so that the current controller knows what voltage was applied over the previous control

intervals. This is particularly important for the back-emf estimation, as this directly requires knowledge of past applied voltages.

In a single-dimensional case, saturation may be achieved by a limiter, which may be expressed as:

$$v_{out} = \begin{cases} v_{sat}; & v > v_{sat} \\ v; & -v_{sat} \leqslant v \leqslant v_{sat} \\ -v_{sat}; & v < -v_{sat} \end{cases} \tag{3.61}$$

At the output of the current controller, however, there is a two-dimensional voltage setpoint. The two voltages V_d and V_q represent the the x and y components respectively of a two-dimensional voltage vector. Limiting each of these independently can result in an output vector with quite a different direction to the originally specified vector.

Limiting to Inverter Capability

The limit of the output range may be established by determining the boundary conditions for valid switching times. As the t_1 and t_2 values have been assured non-negativity through the appropriate choice of sectors, only the value of t_0 remains. So to calculate the range of valid voltages for a given sector, the sector switching equations are solved for $t_0 \geqslant 0$. For the case of sector 1, equation (3.48) is used,

$$t_1 = \frac{T}{2V}\left(V_d - \frac{V_q}{\sqrt{3}}\right) \tag{3.62}$$

$$t_2 = \frac{T}{2V}\left(2\frac{V_q}{\sqrt{3}}\right) \tag{3.63}$$

$$\frac{T}{2} - t_1 - t_2 > 0. \tag{3.64}$$

By combining these equations a simple expression for the maximum allowable voltages may be found,

$$\frac{T}{2V}\left(V_d - \frac{V_q}{\sqrt{3}}\right) + \frac{T}{2V}\left(2\frac{V_q}{\sqrt{3}}\right) < \frac{T}{2} \tag{3.65}$$

$$V_d + \frac{V_q}{\sqrt{3}} < V. \tag{3.66}$$

This result is only valid for sector 1, and so the the equations defining that sector, equations (3.59), must also be met. Similar results may also be derived for the other possible output sectors.

The result of the voltage-limit calculations is that the voltages achievable by the inverter consist of a hexagonal area on the $d-q$ plane. This is illustrated in figure 3.8. If the voltage calculated by the current controller falls outside this region, it must be transformed back into the region before the start of the PWM cycle. A straightforward approach to this is to retain the angle of the vector, but scale the magnitude down so

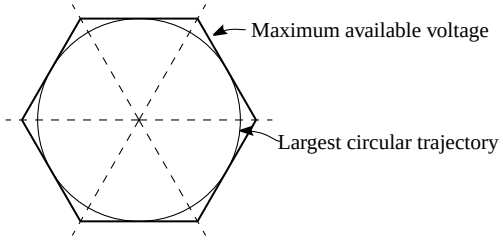


Figure 3.9: The hexagonal voltage limit for the inverter output.

that it lies on the perimeter of the valid region. This represents a vector with the same angle as the desired vector, but with the maximum available amplitude.

The required transformation is $V_d' = \gamma V_d$, and $V_q' = \gamma V_q$, for some γ. For sector 1, γ is derived directly from equation (3.66),

$$\gamma = \frac{V}{V_d + \frac{1}{\sqrt{3}} V_q}. \tag{3.67}$$

In practice, it is more efficient to work directly with the calculated switching times. After calculating t_0, t_1 and t_2, the feasibility of the times may be directly assessed from the sign of t_0. If t_0 is positive, the values are correct, otherwise γ needs to be calculated. As there is a linear function from voltage to the switching times t_1 and t_2 in each sector, γ may be determined from the calculated t_1 and t_2,

$$\gamma = \frac{T}{2\left(t_1 + t_2\right)}. \tag{3.68}$$

This equation holds true for all sectors, so the transformed switching times are:

$$t_0' = 0, \tag{3.69}$$

$$t_1' = t_1 \frac{T}{2\left(t_1 + t_2\right)}, \tag{3.70}$$

$$t_2' = t_2 \frac{T}{2\left(t_1 + t_2\right)}. \tag{3.71}$$

The complexity is further reduced to one multiplication and one division by observing that $t_1' + t_2' = \frac{T}{2}$.

Circular Path Limiting

The disadvantage of limiting the voltage to the feasible hexagon is that it is likely to result in amplitude pulsations. Under saturation conditions, an induction machine is likely to be driven by a voltage vector that is rotating with time with the maximum available amplitude. If the vector is limited to stay within the hexagon, the voltage magnitude will not only be less than desired, but have an amplitude peaking at six times

per electrical revolution. In order to prevent this, the limit must be placed at a constant magnitude. Figure 3.9 shows the largest circular path that may be achieved within the inverter limits. By limiting to this path, a constant magnitude is achieved at the expense of a small decrease in the maximum power available from the inverter.

A further difficulty involved in limiting to a circular trajectory is the level of computation involved in calculating the limit. This calculation will require square-root or trigonometric operations and so will take longer to calculate the limited value. The radius of the circle may be determined by finding the points along the hexagonal limit set whose distance from zero is the smallest. In the case of sector 1, equation (3.66) is taken as the limit path. To find the point closest to the origin, $V_d^2 + V_q^2$ is minimised for a fixed V. For sector 1, this requires,

$$d_{min}^2 = \min_{V_q} \left\{ V_q^2 + \left(V - \frac{V_q}{\sqrt{3}} \right)^2 \right\} \tag{3.72}$$

$$d_{min} = \frac{\sqrt{3}}{2} V. \tag{3.73}$$

The limiting may be performed by first finding the magnitude of the desired voltage vector,

$$|V_s| = \sqrt{V_d^2 + V_q^2}. \tag{3.74}$$

If $|V_s| < \frac{\sqrt{3}}{2} V$ then no limiting is required. Otherwise, both V_d and V_q are scaled to fit within the circle,

$$V_d' = V_d \frac{\sqrt{3}}{2} \frac{V}{|V_s|} \tag{3.75}$$

$$V_q' = V_q \frac{\sqrt{3}}{2} \frac{V}{|V_s|}. \tag{3.76}$$

As the maximum voltage magnitude under this method is always $\frac{\sqrt{3}}{2} V$, as opposed to a value varying between $\frac{\sqrt{3}}{2} V$ and V with the hexagonal limiting case, the average voltage available is approximately 7% lower.

Constant Direction Approach

More complex approaches may be applied to the limiting, to aviod torque set-point changes affecting the flux. A constant angle active voltage vector approach can be used. This technique has the advantage that it does not introduce the dq axis cross coupling present with the scaled limiting when it used in a vector controlled drive. However, the price paid for the better performance is additional computation. This can be seen from

the following expressions for the limited α's using this constraint [3]:

$$K^* = \frac{\alpha_q V - e_q}{\alpha_d V - e_d} \tag{3.77}$$

$$\alpha_{d_{lim}}[k+1] = \frac{\sqrt{3}V + K^* e_d[k] - e_q[k]}{\sqrt{3}V\left(1 + \frac{K^*}{\sqrt{3}}\right)} \tag{3.78}$$

$$\alpha_{q_{lim}}[k+1] = \frac{K^*(V - e_d[k]) + e_q[k]}{V\left(1 + \frac{K^*}{\sqrt{3}}\right)}. \tag{3.79}$$

While this alternate algorithm will offer better torque transient behaviour, it has not been implemented because of the computational complexity it would add to the controller.

3.4 Inductance Estimation

In the machine model utilised, the parameters of the machine are all bound up in one inductance quantity. This characterises the properties of the rotor and stator combined. This section describes how the value of this lumped parameter may be estimated.

The machine inductance changes slowly compared to the electrical time constant, and so for the purposes of control it may be considered a constant. However it does change with the operating point of the machine, and so there is a benefit to be gained in attempting to track these parameter variations. The provision of inductance estimation also allows the controller to be machine-independent, with no parameters required for the commissioning of the current controller.

The process of inductance estimation requires comparing the voltage across the leakage inductance to the change in current through it. The change in current may be readily determined from the machine terminal quantities, but the voltage is more difficult to obtain. The basic system equation is,

$$v(t) = L_l \frac{di}{dt} + e_b(t). \tag{3.80}$$

In this equation, the voltages and currents can both be measured, but the leakage inductance(L_l) and back emf(e_b) are unknown. Because there are two unknowns, this equation in itself is not sufficient to perform the estimation. An approximated difference version of this equation is used to estimate the back emf of the machine on each control cycle. As the back-emf varies relatively quickly with time, the available data is required for this purpose. Consequently alternate datasets are required to obtain estimates for the back-emf.

3.4.1 Quarter-Cycle Method

One approach[4], uses additional independent current measurements for the inductance estimator. The basic controller uses measurements at the start, and mid-point of a control

interval, but an inductance estimator may be designed to use a current measurement taken one quarter of the way through the control cycle.

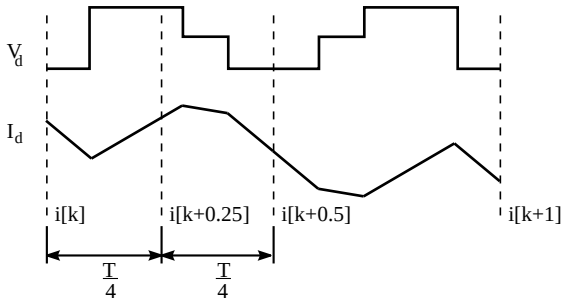


Figure 3.10: Quarter-cycle current measurement.

The quarter-cycle measurement offers an additional measurement that is independent of the values used for the calculation of the back-emf. It is also attractive for this application because the voltage applied across the first quarter of the cycle is likely to be quite different to the voltage across the second quarter. This greater variation is useful in estimating the inductance as it amounts to a greater excitation signal.

For the PWM scheme presented, a set of two-phase voltage waveforms may be found. The voltage waveform will vary according to the sector, and the resulting values are shown in Figure 3.11.

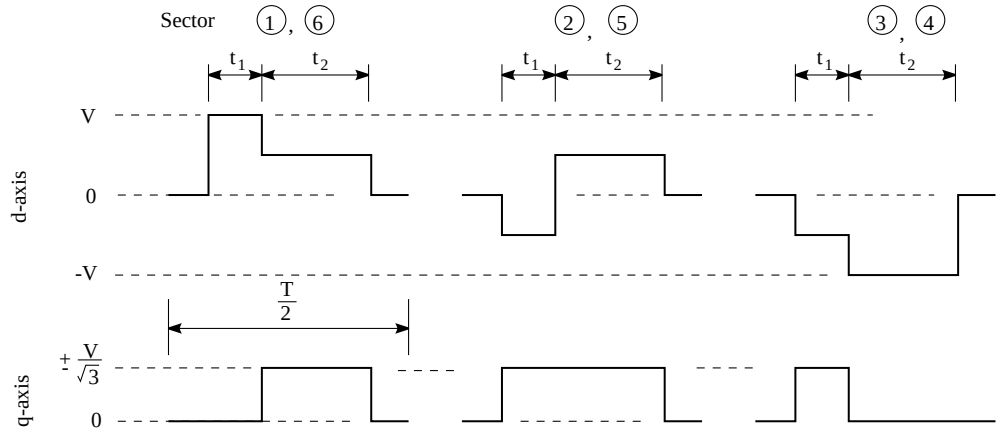


Figure 3.11: Two-phase voltage waveforms for each sector.

The upper set of traces represent the voltage pattern in the direct axis for the case $t_1 < t_2$. In the direct axis, there are five possible output voltages, $-V$, $-\frac{V}{2}$, 0, $\frac{V}{2}$ and V. On the q-axis, there are only three possible voltages, $-\frac{1}{\sqrt{3}}V$, 0 and $\frac{1}{\sqrt{3}}V$. The waveforms for sectors 1,2 and 3 are shown, while for sectors 4,5 and 6, the output is the negative of that for sectors 3,2 and 1 respectively. The switching voltage vectors were shown in Figure 3.6.

For the case of the direct axis in Sector 1, the voltage component is shown in Fig-

ure 3.12. For the purposes of using these measurements for inductance estimation, the

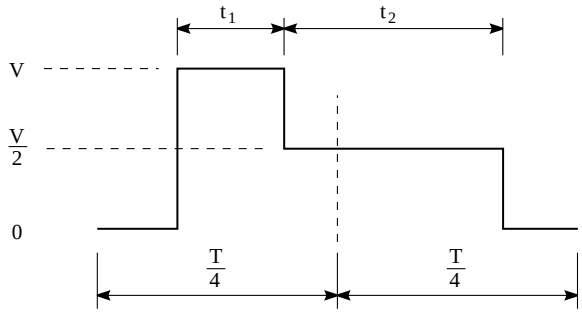


Figure 3.12: Two-phase voltage waveforms for each sector.

important quantities are the average voltage across each of the $\frac{T}{4}$ periods. In particular, the difference between these two will be used. Let V_{d1} and V_{d2} represent the average voltages across the first and second quarter of the control cycle respectively. When $t_2 > t_1$,

$$V_{d2} - V_{d1} = -\frac{V}{2}\left(\frac{4}{T}t_1\right) = -\frac{2}{T}Vt_1 \tag{3.81}$$

For the case of $t_1 > t_2$, the result is instead,

$$V_{d2} - V_{d1} = -\frac{2}{T}Vt_2. \tag{3.82}$$

By assuming that the back-emf is constant across this period, an estimate for the inductance may be obtained. Taking a first-order approximation to Equation (3.80),

$$v(t) \approx L_l \frac{i(t+\Delta T) - i(t)}{\Delta T} + e_b(t). \tag{3.83}$$

This equation is then applied across both of the first two quarter cycles and the difference in the average terminal voltage taken,

$$V_{d2} - V_{d1} \approx \frac{4L_l}{T}(i[k+0.5] - 2i[k+0.25] + i[k]). \tag{3.84}$$

By combining with the previous expressions, the estimate for L_l from the d-axis quantities is formed,

$$L_{l1,3,4,6} \approx \frac{-min\{t_1, t_2\}}{2(i[k+0.5] - 2i[k+0.25] + i[k])}. \tag{3.85}$$

This expression is not valid for Sector 2 and Sector 5. In this case, the change in the d-axis voltage component between the t_1 and t_2 intervals is double that of the other

sectors. As a result, for these sectors the estimate is,

$$L_{l2,5} \approx \frac{-min\{t_1, t_2\}}{i[k+0.5] - 2i[k+0.25] + i[k]}. \tag{3.86}$$

On the q-axis, this method cannot be used for Sector 1 and Sector 5, as the voltage waveform is symmetric across the half-cycle, leaving a zero voltage differential.

Performance of the Quarter-Cycle Method

This inductance estimator has been implemented in simulation, as well as in practical implementations. Both digital hardware and software implementations were made, but with limited success. A simulation of this inductance estimator is shown in Figure 3.13. With greater filtering, the result would be close to the true value of 10mH. For a noiseless

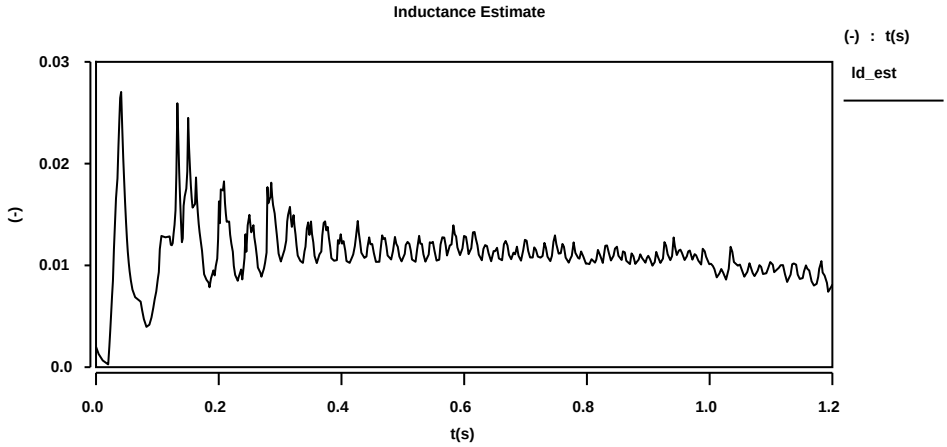


Figure 3.13: Filtered quarter-cycle inductance estimator estimate.

model, and the errors are quite high, and are mainly due to inverter dead time and quantisation errors. Under ideal conditions, this estimator performs well, as expected by the theory. Under practical conditions, there are a number of issues that reduce the effectiveness of the estimate.

Noise The most obvious problem with the estimator is that effectively a double derivative of the current measurements is needed. Differentiation increases the amplitude of noise, and differentiating a signal with noise twice greatly increases the level of noise. Provided bias is not introduced, this may be overcome by filtering the estimated value.

The division by the differentiated currents does pose the problem of bias. If the denominator of the expression is corrupted with zero-mean noise, p_{meas} from the current measurement error, the estimated value will be of the form,

$$L_l + p_{est} = \frac{K}{x + p_{meas}}. \tag{3.87}$$

Due to the non-linear division operation, p_{est} does not have a zero mean, and so the estimate is biased. If $|e_{meas}| \ll |x|$, the bias is small, but this cannot be assumed in this case. The situation is particularly poor if the noise can cause the denominator to change sign. In this case, estimates of extremely large magnitude can be generated, due to the near-zero denominator.

An alternative is to instead filter the estimate of $\frac{1}{L_l}$. If the error in the voltage generation is less than on the current, this will represent an improvement. This approach also has the advantage that measurements can be ignored if the denominator is small, without creating additional bias.

A greater improvement can be achieved through using a least-squares estimator, but at the cost of additional complexity. Such an estimator, along with a more detailed treatment of the filtering issues, is described below in reference to the alternate estimator(§3.4.3).

Measurement Location In the current controller design, the endpoint and midpoint currents were particularly chosen as this is in the centre of a period where the zero-vector is applied. As the devices can be prevented from switching at this time, the noise from switching need not affect the current measurements. This is not true for the quarter-cycle measurement, where it is difficult to prevent the measurement coinciding with a device switching instant. This increases the overall level of noise on the estimates. The coincidence of sampling and switching could be detected from the t_0 and t_1 values, but again this adds to the complexity of the estimator.

Inverter Dead-Time The presence of inverter dead-time is a significant problem for the inductance estimator. There are two reasons for this:

- Dead-time causes a delay in the entire output pattern. This causes a mis-alignment between the sampling and the true end of the interval. While the measurement times can be skewed to compensate, the uncertainty in the delay will still cause some error.

- Due to inaccuracy in the switching time there will be an error in the average voltage that is achieved.

In both of these cases, this inductance estimator is far more susceptible than the current controller itself. In the case of skew in the entire output pattern, the current controller is largely unaffected, as the difference between the measurements is still the same, unless the delay is greater than t_0. The reason for this can be seen in Figure 3.10. The derivative of the current at both of the endpoints, and the midpoint is the same. This means that a time delay will affect all of the measurements in the same way. As the current uses only the differences, it is unaffected by the skew.

The derivative at the $i[k+0.25]$ measurement is different to the endpoint, and so there will be an error in the difference. In fact, because the estimator is using the difference between the changes in the first and second quarters, the error is even greater.

The current controller may also compensate to some extent for the voltage variation due to inverter dead time. This is because of the symmetric switching pattern over the full cycle. When an error occurs on one half cycle, some compensation occurs in the other half because the switching pattern is reversed. This inductance estimator cannot benefit from this.

3.4.2 Estimation Sampling Alternatives

The deficiencies of the original inductance estimator prompted the development of an alternate estimator. In order to avoid the problems associated with the quarter-cycle measurement, two alternatives are available,

- Use a large number of measurements spread through the interval.
- Use only the end-point data.

The option of using a large number of measurements, as illustrated in Figure 3.14, was considered at length because of the suitability of digital hardware for such an implementation. While measurements will still be corrupted with noise from switching, the individual effect of the erroneous sample will be less. It would be difficult to use this

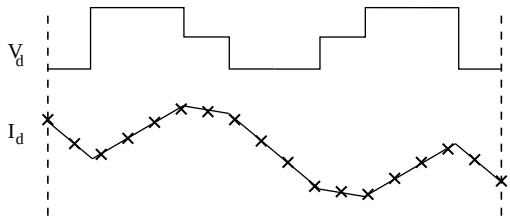


Figure 3.14: Using multiple measurements per cycle.

approach using a microprocessor shared with the controller because of the high data and interrupt rate. A parallel implementation in hardware would be very appropriate, but a suitable simple algorithm was not found for this architecture.

The alternative is to use only the measurements available from the current controller, these being the endpoint and midpoint currents. The disadvantage with this approach is that there is not as much information available, and care must be taken to ensure that the back-emf estimation and the inductance estimation are independent.

As the back-emf estimation depends on the inductance value, it is dangerous to also make the inductance estimate a function of the back-emf estimate, unless the stability of such an iterative algorithm could be proven.

3.4.3 Full-Cycle Inductance Estimator

An inductance estimation method has been developed that uses only the endpoint current measurements. The endpoint currents were chosen for convenience and so that the error due to the inverter dead-time could be minimised. Additionally, unlike the current controller, calculation delays are not of great importance because the inductance is only slowly varying.

The basic assumption for this estimator is that the derivative of the back emf only changes slowly across one cycle. This is a more accurate assumption than that of simply a slowly changing back-emf. For one axis in the two axis model, this may be written,

$$e_k - e_{k-1} \approx e_{k-1} - e_{k-2} \tag{3.88}$$

$$e_k - 2e_{k-1} + e_{k-2} \approx 0. \tag{3.89}$$

This may be related to the model currents and inductance by Equation (3.80). In this case, an Euler approximation is made for the derivative across one controller period,

$$v_k = e_k + \frac{L_l}{T}\left(i_i - i_{k-1}\right). \tag{3.90}$$

By substituting (3.80) into (3.89), the estimator is formed,

$$v_k - 2v_{k-1} + v_{k-2} = \frac{L_l^{est}}{T}\left(i_k - 3i_{k-1} + 3i_{k-2} - i_{k-3}\right). \tag{3.91}$$

This equation may be solved directly for a point estimate of the leakage inductance, but this does require a division by what is effectively a triple derivative of the current. In the presence of noise, it may not be possible to reconstruct an unbiased estimate through subsequent filtering.

Instead of point estimation, a recursive least-squares algorithm is proposed to estimate the parameter. This was noted as involving complexity in the case of the quarter-cycle method, but in this context, that is offset by the reduced complexity in not requiring the additional current samples.

Recursive Least-Squares Estimate

To apply a least-squares estimate, the model is expressed as,

$$y = Cx + v. \tag{3.92}$$

The column vectors y and x represent the measurements, and the parameter vector to be evaluated respectively. The measurement vector, y, must be larger than the number of parameters to estimate. C is a matrix, and v a noise vector that model the expected relationship between x and y.

For the inductance parameter estimation, x is a scalar, and C is a vector the same

size as y. The principle aim is to choose a parameter value for L_l, which will minimise the error in the machine current. It is therefore appropriate to estimate the reciprocal of the inductance, $\frac{1}{L_l}$. The error in y is then equal to the error in i_k if these equations are used as a predictor for current.

The least-squares variables are then,

$$y = \begin{bmatrix} i_k - 3i_{k-1} + 3i_{k-2} - i_{k-3} \\ i_{k-1} - 3i_{k-2} + 3i_{k-3} - i_{k-4} \\ i_{k-2} - 3i_{k-3} + 3i_{k-4} - i_{k-5} \\ \dots \end{bmatrix}, \quad C = T \begin{bmatrix} v_k - 2v_{k-1} + v_{k-2} \\ v_{k-1} - 2v_{k-2} + v_{k-3} \\ v_{k-2} - 2v_{k-3} + v_{k-4} \\ \dots \end{bmatrix}. \tag{3.93}$$

The standard least-squares estimation involves calculating,

$$\hat{x} = \left(C^T C\right)^{-1} C^T y. \tag{3.94}$$

This batch style calculation is not appropriate for on-line estimation. Instead, the recursive least-squares form is required. In this case, the estimation occurs for each data point that is received. The estimation algorithm uses information from the previous iteration to minimise the calculation required at each instant.

The recursive algorithms are specifically designed for the case where the estimation is to occur over an indefinitely long time. In such cases, the conventional least-squares algorithm will become "stale". This is because the estimated variable is modelled as a constant, and so time variations in the parameter are not accommodated. After a large number of data are collected, each additional measurement will have little effect on the estimation result.

In order to accommodate time-varying parameters, the variation may be explicitly modelled. This would involve the introduction of a state variable, and a model of the variation process, such as a random walk. While this addes complexity, for a basic application where the specific model of the variation is not known, the simple inclusion of a "forgetting factor" is sufficient. The forgetting factor is introduced to decrease the modelled certainty at each timestep. Effectively, it gradually reduces the impact of old estimates over time.

The form of the recursive least-squares estimation with a forgetting factor, λ, is commonly expressed as,

$$e_k = y_k - C_k \hat{x}_{k-1} \tag{3.95}$$

$$K_k = T_k C_k^T \left(W_k^{-1} + C_k T_k C_k^T\right)^{-1} \tag{3.96}$$

$$\hat{x}_k = \hat{x}_{k-1} + K_k e_k \tag{3.97}$$

$$T_{k+1} = \frac{1}{\lambda} \left(I - K_k C_k\right) T_k. \tag{3.98}$$

The forgetting factor, λ, is a number between 0 and 1. The smaller this value is, the

sooner older measurements are "forgotten". This value should be chosen based on how quickly the parameter is expected to vary. For the inductance estimation, the value varies slowly with respect to the sampling period, and so the value of λ can be rather high. For the testing, values around 0.995 were used.

These least-squares equations have provision for a weighting factor on each measurement. This expresses the relative confidence in the data for that measurement. For this case, all of the data may be treated the same, and so the weighting on the values is a constant. W may then be set to 1 for all iterations.

As only one value is being estimated, all of the quantities are scalars. Under these conditions, the equations have been simplified to,

$$U_k = \frac{T_k}{1 + T_k C_k^2} \tag{3.99}$$

$$\hat{x}_k = \hat{x}_{k-1} + C_k U_k (y_k - C_k \hat{x}_{k-1}) \tag{3.100}$$

$$T_{k+1} = \frac{1}{\lambda} U_k. \tag{3.101}$$

Each iteration requires one divide and six multiplications. This could be easily implemented on a DSP, and a direct hardware implementation would be possible. In the hardware version, careful scaling would be needed with the fixed-point quantities.

Simplified Estimator

In cases where the recursive least-squares estimate calculation is too complex, a simplified, yet inferior, estimate may be found. By re-arranging Equation (3.91), the direct form for the inductance estimate is,

$$L_l^{est} = T \frac{v_k - 2v_{k-1} + v_{k-2}}{i_k - 3i_{k-1} + 3i_{k-2} - i_{k-3}}. \tag{3.102}$$

The resulting estimate is very noisy, and must be filtered over time to be useful. A suitable filter of a quantity u_k is,

$$\hat{x}_k = \lambda \hat{x}_{k-1} + (1 - \lambda) u_k. \tag{3.103}$$

The parameter λ is equivalent to the forgetting factor above. Indeed this filter is actually a recursive least-squares estimator, for the simple case of $C_k = 1$.

As noted in the quarter-cycle estimation method above, the estimated value may be filtered either directly, or in the reciprocal form. The advantage of the reciprocal form is that it is easier to remove poorly conditioned estimates without introducing additional bias. Values with a small sum of voltages are simply ignored.

If L^{est} is filtered directly, the conditioning is based on the sum of the currents. Sums of zero must be removed to avoid a division by zero. If the rejection band is too small, the estimate will be biased high due to the bias introduced by dividing by small numbers.

In contrast, if too many measurements are rejected on the basis that the denominator is too small, the result will be biased low. While satisfactory operation can be achieved for a reasonably wide band, this dependence on a parameter can somewhat defeat the original purpose in estimating the inductance.

By filtering $1/L^{est}$, the bias issues are lessened, but a number of other problems are introduced. Firstly, the actual value required by the controller is the inductance itself, so the presence of the estimate as an inverse will require an additional division per cycle.

For the inductance value itself, in a fixed-point implementation the relative accuracy of representation increases with increasing values of L. This is because the absolute accuracy is a constant in fixed-point. This is convenient because erroneously small values of inductance are more likely to provide controller stability in the case of parameter uncertainty. As as fail-safe default, small values of inductance may be represented, but with reduced accuracy.

When the reciprocal of inductance is being used, provision for small values of L in fixed-point requires a significant increase in the word-length, or a reduction of accuracy for normal values. A further disadvantage occurs during the start-up transient. In this case, an initial value for the inductance estimate must be supplied. To ensure stability, this value should be small, say one-tenth of the true value. Now, a linear filter will have a better transient if the initial condition is one-tenth of the true value, rather than ten times the true value. In the latter case, the size of the transient in the filter is nine times as large. A comparison is illustrated in Figure 3.15. In this figure, the solid line

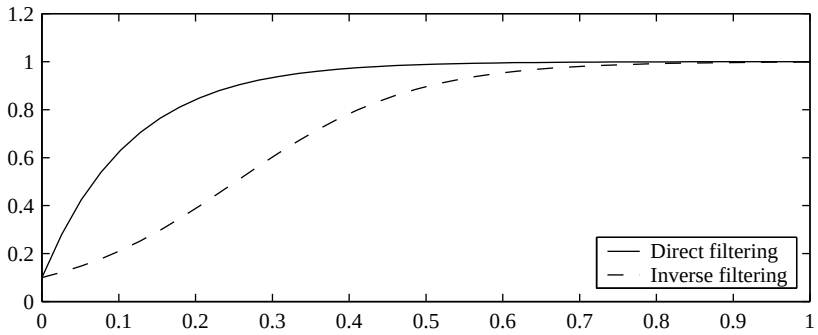


Figure 3.15: Comparison of filtering types.

represents conventional filtering, while the broken line is for the case where the inverse value is filtered. For equivalent steady-state filtering action, the initial transient from a small value is faster for the direct filtering.

This comparison shows that there is a role for each of the estimation and filtering techniques, depending on the calculation facilities. For the simplest calculation using fixed-point numbers, the direct filtering of Equation (3.102) is most appropriate, but the potential for bias is the highest. Where a wider numerical range is available, calculating and filtering the reciprocal of this quantity is a viable alternative, although the recursive

least-squares will provider a superior estimate. Note also that the least-squares requires the evaluation of only one division. For the case of a DSP implementation, where adds and multiplies are cheap, yet divisions often expensive, the least-squares estimator is likely to be the best alternative.

Performance

The performance of the full-cycle estimator is shown in Figure 3.16. This plot includes three traces, for various forms of the estimator. These estimates are all obtained from a real induction machine under test. The broken line with a sharp initial rise shows the

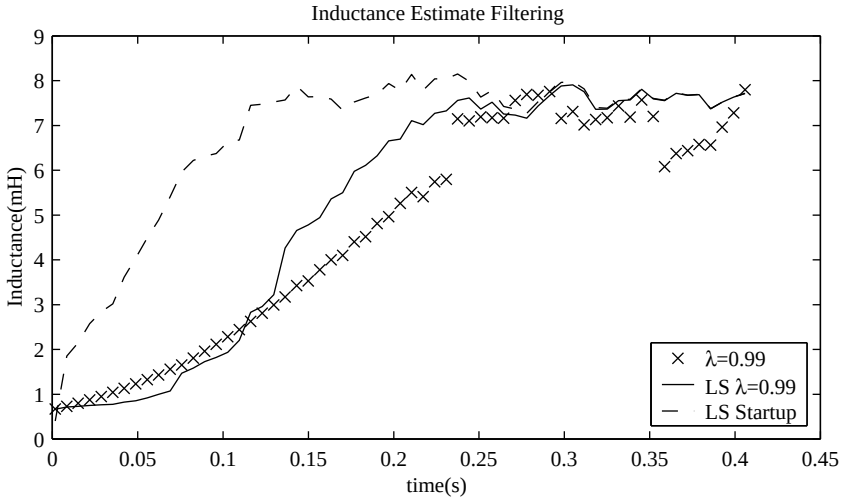


Figure 3.16: Inductance Estimation using the full cycle method.

behaviour of the basic least-squares estimator on start-up. A very small initial condition value of about 1mH was used for each test. The sharp transition toward the correct value is possible because the recursive least-squares estimator weights the data according to the certainty of the estimation. Initially, little is known of the true value, so additional weighting is given to the observed data points. Later, as more data is collected, each individual sample has less impact on the current estimate. As the estimator reaches steady-state, all of the new data has the same weighting. The size of this weighting, compared to the existing estimate, is given by the forgetting factor.

In the recursive least-squares formulation, the state variable T is a measure of the current uncertainty in the estimate. Initially this starts off high, and is reduced as additional measurements are made. As older measurements are "forgotten", this is divided by λ to increase the uncertainty in the existing value.

In the case of the solid line, the value of T in the least-squares estimate is initialised to the steady-state value from a previous test. This situation better represents the case where there is a change in the parameter value during operation. The shape of the transient under these conditions is determined by the value of λ.

This result may be directly compared to the cross points, which represent the simplified filtering method. In this example, the inverse of the inductance is filtered, to minimise the bias. The filtering constant used for the simplified estimator is the same as the forgetting factor in the least-squares estimate. As a result, the transient is a similar shape, but it does have a greater level of noise.

As the estimation equation is simply a relationship between the machine voltages, currents and the inductance, it may instead be used to predict one of the other quantities. Using the already estimated inductance, the performance of the estimator as a predictor is shown in Figure 3.17. It is obtained by re-arranging the estimator to the form,

$$i_k = 3i_{k-1} - 3i_{k-2} + i_{k-3} + \frac{T}{L_l^{est}} \left(v_k - 2v_{k-1} + v_{k-2} \right). \tag{3.104}$$

This shows the ability of the estimator to predict future values of current, based on the past currents and voltages.

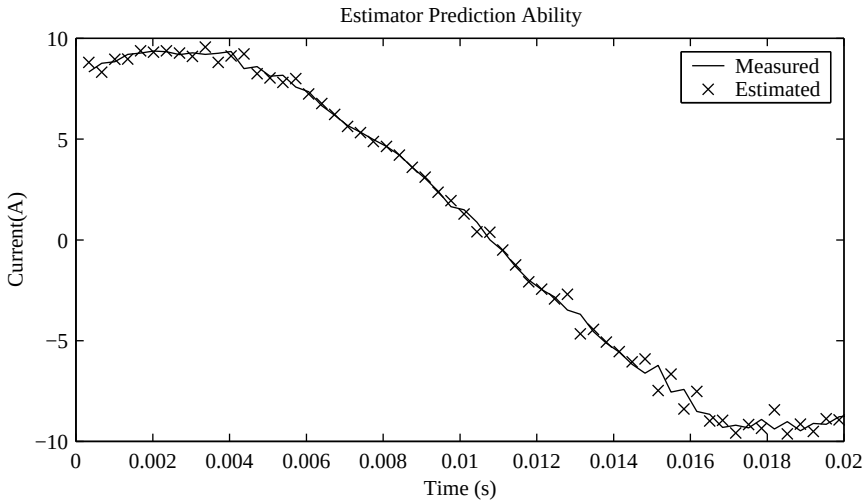


Figure 3.17: Performance of the estimator.

The measured value is shown with the solid line, and the one-step ahead predictor is shown with the marks. The two traces show good agreement. This confirms the model, because given that model, the estimator will find the minimum prediction error in a sum of squares sense.

3.5 Conclusions

This chapter has presented the elements necessary for the construction of the current controller. First a model was chosen that captured the correct level of detail of the machine operation. It was shown that the approximations made from the full model were valid.

For symmetric space-vector modulation, it was found that it is suitable to use two measurements for control per output switching cycle. Using additional measurements is awkward both in terms of susceptibility to noise, and the modelling required.

Two methods were proposed for estimating the machine leakage inductance. The quarter-cycle method, was found to suffer from significant limitations, but the full-cycle method shows good performance in a practical test situation. In particular, when coupled with recursive least-squares estimation, it exhibits a fast initial transient and low steady-state noise.

Chapter 4

Controller Analysis

4.1 Introduction

In the previous chapter, a system model and a corresponding predictive controller were developed. The purpose of this chapter is to theoretically validate the controller design, and identify its strengths and weaknesses. The controller as proposed also has a number of parameters that may be altered to obtain a particular type of performance. The effect of varying these parameters is investigated.

The first section presents a theoretical analysis of the performance of the current controller. Two important aspects of the controller operation are its tracking ability, and its stability, and these are both investigated. Following this, the limitations of the basic model are analysed. The main shortcoming is found in the assumption of a constant back-emf, and a number of schemes are proposed to reduced the error associated with this.

4.2 Standard Controller Performance

For ease of notation, the definitions of the controller quantities have been varied for the analysis section. This is to be more consistent with traditional signal processing notation, and allows the same variables to be used in the transfer function form. Figure 4.1 shows the quantities used.

For the purposes of this analysis, it is assumed that the current measured in the centre of the switching interval is always the average of the currents measured at the endpoints. According to the model (Figure 3.3) this will always be the case. Due to the symmetry of the switching pattern, the average voltage applied across the first and second halves of the switching cycle will be the same. Consequently, for a constant back-emf and leakage

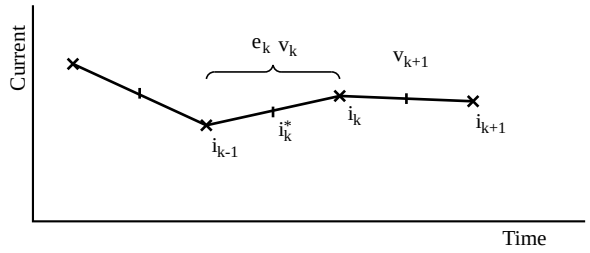


Figure 4.1: Schematic current waveform showing the allocation of subscripts to measured currents and voltages.

inductance, the change in current across each half of the interval will be the same, ie.

$$i_{k+0.5} - i_k = i_{k+1} - i_{k+0.5} \tag{4.1}$$

$$i_{k+0.5} = \frac{i_k + i_{k+1}}{2}. \tag{4.2}$$

To facilitate analysis, it is assumed that there is no prediction error when estimating the current at the start of a control interval. This is necessary to cast the controller in a standard shift operator form. By making the prediction assumption, each sample period has one voltage and one current associated with each axis of the machine.

Under these assumptions, the basic controller equations are:

$$v_{k+1} = \frac{\rho L_l \Delta L}{T} (u_{k+1} - i_k) + e^{pred}_{k+1} \tag{4.3}$$

$$e^{est}_{k+1} = v_{k+1} - \frac{L_l \Delta L}{T} (i_{k+1} - i_k). \tag{4.4}$$

In these equations, the real machine leakage inductance is L_l, while the estimated inductance is $L_l \Delta L$. ΔL is the multiplicative estimation error.

In addition to the controller equations, the back-emf prediction is taken as the estimate from the previous interval. The prediction is necessary in the controller for causality. The simple assumption of a nearly constant back-emf has shortcomings, but alternatives such as linear extrapolation are considered later. For the first analysis, the back-emf prediction is given by:

$$e^{pred}_{k+1} = e^{est}_k. \tag{4.5}$$

The behaviour of the coupled controller and machine may be found by simultaneously solving all of the associated equations. The ideal model for the machine is:

$$v_k = \frac{L_l}{T} (i_k - i_{k-1}) + e_k. \tag{4.6}$$

The combination of this model together with the controller equations describes the dynamics of the complete system.

4.2.1 System Transfer Function

In order to solve the equations, the controller equations may first be combined to obtain a single updating equation,

$$v_{k+1} = \frac{\rho L_l \Delta L}{T}\left(u_{k+1} - i_k\right) + v_k - \frac{L_l \Delta L}{T}\left(i_k - i_{k-1}\right). \tag{4.7}$$

Following this, the machine equation may be used to eliminate the voltage terms from the updating equation. Because the current is the quantity of interest, the equations should be in terms of the machine currents and not the voltages. Once the current response is found, the associated voltages could be determined from that if required.

By substituting the machine equation (4.6) into the controller equation 4.7, the system difference equation is,

$$\begin{aligned}
&\frac{L_l}{T}\left(i_{k+1} - i_k\right) + e_{k+1} = \frac{\rho L_l \Delta L}{T}\left(u_{k+1} - i_k\right) + \frac{L_l}{T}\left(i_k - i_{k-1}\right) + e_k - \frac{L_l \Delta L}{T}\left(i_k - i_{k-1}\right) \\
&\left(i_{k+1} - i_k\right) + \frac{T}{L_l} e_{k+1} = \rho \Delta L\left(u_{k+1} - i_k\right) + \left(i_k - i_{k-1}\right) + \frac{T}{L_l} e_k - \Delta L\left(i_k - i_{k-1}\right). \qquad (4.8)
\end{aligned}$$

Now collecting the current terms together:

$$\begin{aligned}
&i_{k+1} + \left(-1 + \rho \Delta L - 1 + \Delta L\right) i_k + \left(1 - \Delta L\right) i_{k-1} \\
&\quad = \rho \Delta L u_{k+1} + \frac{T}{L_l}\left(e_{k+1} - e_k\right). \qquad (4.9)
\end{aligned}$$

The forward shift operator z is now used to create a discrete-time transfer function. This is defined as,

$$z x_k = x_{k+1}, \tag{4.10}$$

for a discrete-time quantity x. After converting the system expressions to the forward shift operator, the transfer function from the reference, u, to the output current, i is,

$$\begin{aligned}
&i\left(z^2 + \left(\Delta L(\rho + 1) - 2\right)z + \left(1 - \Delta L\right)\right) \\
&\quad = z^2 \rho \Delta L u - (z - 1) z \frac{T}{L_l} e. \qquad (4.11)
\end{aligned}$$

In this case, the under-modelling of changes in the back-emf is treated by including the back-emf as a disturbance term. Unless the time-varying nature of the back-emf is being considered, this final term involving e may be ignored.

4.2.2 Stability Analysis

The stability of the controller may be determined by considering the locations of the poles of the discrete-time transfer function. The poles are the values of z that set the denominator of the transfer function to zero.

Ignoring the back-emf disturbance term, the transfer function from reference to machine current is:

$$G(z) = \frac{I(z)}{U(z)} \tag{4.12}$$

$$= \frac{z^2 \rho \Delta L}{z^2 + (\Delta L(\rho + 1) - 2)z + (1 - \Delta L)}. \tag{4.13}$$

The denominator of this expression is the characteristic equation, and the values, called the roots, for which this is zero are important for stability. The roots may be complex numbers, and if the magnitude of all of the roots is less than one, the system is stable.

Consider first the case where there is no inductance estimate error. In this case $\Delta L = 1$, and the characteristic equation is:

$$(z + \rho - 1)z = 0. \tag{4.14}$$

The roots of this equation are $1 - \rho$ and 0. The root at zero simply indicates a time delay, but dynamics may be associated with the root at $1 - \rho$. In particular, if ρ is greater than two the system will become unstable.

This shows that the average current control ($\rho = 2$) is only marginally stable, with an oscillation frequency of half the control rate. This result could be expected, as with average control, only the midpoint of the straight-line segments is constrained, and the endpoints may alternate between large positive and negative values. If the midpoint or average current is used instead of the endpoint current, this oscillation frequency is an unobservable mode, and the instability will not be apparent from those measurements. Despite this, the instability is a bad thing because it will be likely to cause a very large current ripple, constrained only by the saturation limits of the system.

In the absence of estimation errors, the endpoint controller ($\rho = 1$) offers dead-beat control over the endpoint current. At the end of the interval the reference current is reached and there are no undesirable dynamics.

The location of the closed-loop system poles may be plotted for varying inductance estimation error. This root-locus plot appears as Figure 4.2. Figure 4.2 shows a discrete-time root-locus plot for the endpoint control case ($\rho = 1$). The varied parameter is the multiplicative error in the inductance estimate. The circle indicated with a broken line represents the boundary of stability. Provided the poles remain inside this circle, the system is stable.

In this case, all of the closed loop poles are stable provided $\Delta L < 1.3$. If the estimate of L is too low, only the bandwidth is affected. In the presence of uncertainty, it is quite safe to deliberately underestimate the value of the leakage inductance.

The stability of a discrete-time system can be determined by the largest magnitude of the closed loop poles. This value is plotted as a function of the error in the inductance

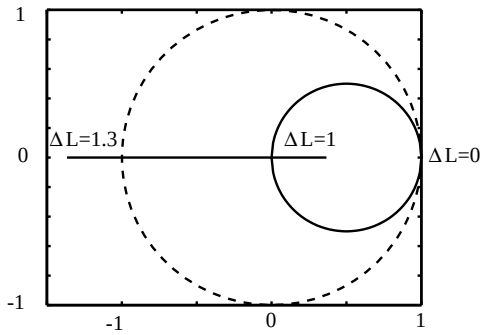


Figure 4.2: Root-locus plot for the controller.

estimate in Figure 4.3. The solid line represents the case for endpoint control, with $\rho = 1$.

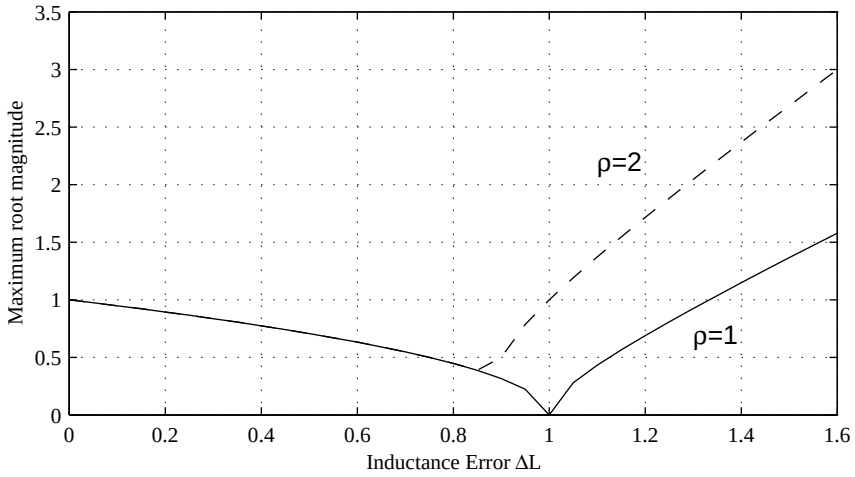


Figure 4.3: Maximum root magnitude as a function of the inductance estimation error.

This clearly shows the limits of stability for this case, where the estimate for L_l may fall anywhere between zero and over 1.3 times the the true value. However, for average current control, the pole location magnitude is shown with the broken line. In this case, estimates of L_l that are too small have the same properties, but any over-estimation will cause instability. This shows that the average current control is difficult to tune.

Endpoint vs. Average Current Control

Figure 4.4 shows how the stability boundary changes for values of ρ between 1 and 2. From this plot, it can be noted that further decreasing ρ below unity offers an even greater tolerance to errors in the leakage inductance estimate. This corresponds to the controller seeking to reach the reference at some point after the end of the control interval. There are performance penalties in doing this, and the tolerance available at $\rho = 1$ should generally be sufficient.

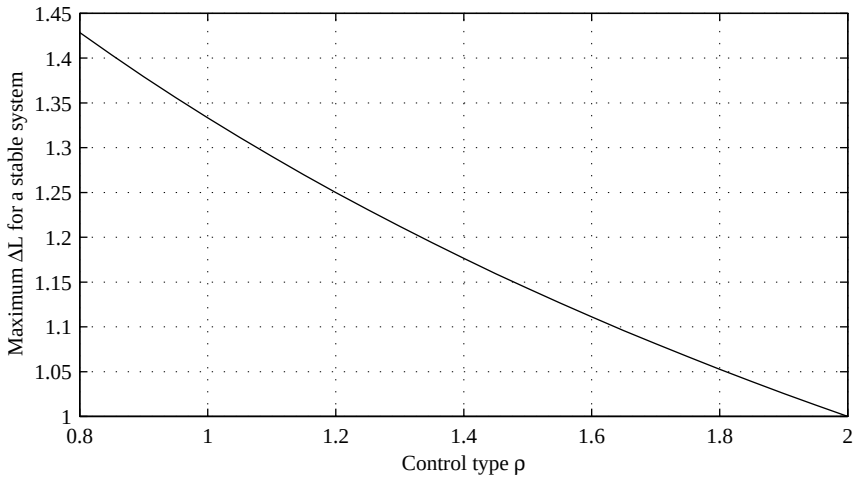


Figure 4.4: Maximum inductance estimation error for a stable system, as a function of ρ.

To obtain the tracking benefits associated with a high value of ρ, reference prediction could instead be used. This involves considering the reference to be controlling the average current across an alternate time window. This is the window centred around the control interval endpoint. The geometry is shown in Figure 4.5. With the reference

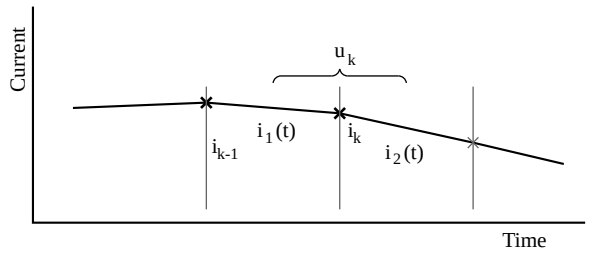


Figure 4.5: Endpoint average current window.

prediction, the aim is to control the average current across the interval designated by u_k. The average current control is obtained by simply using the endpoint controller to regulate the current at the centre of the u_k window.

The error involved in making this assumption can be found by determining the true average across the nominated window. The average u_k will be found by:

$$u_k = i_k + \frac{1}{T}\left(-\int_{-\frac{T}{2}}^{0} \frac{di_1}{dt} t \; dt + \int_{0}^{\frac{T}{2}} \frac{di_2}{dt} t \; dt\right) \tag{4.15}$$

$$= i_k + \frac{T}{4}\left(\frac{di_2}{dt} - \frac{di_1}{dt}\right) \tag{4.16}$$

$$\approx i_k + \frac{T^2}{4}\left(\frac{d^2 i_{ref}}{dt^2}\right). \tag{4.17}$$

For a reference sinusoid, $i = A\sin(\omega t)$, the second derivative is given by,

$$\frac{d^2 i}{dt} = -A\omega^2 \sin(\omega t). \tag{4.18}$$

The peak error in the average assumption is,

$$max(u_k - i_k) = \frac{1}{4} i_{peak} (\omega T)^T. \tag{4.19}$$

For 50Hz operation, and a control rate of 3kHz, this represents a maximum error of 0.14% of the peak-to-peak reference current. For most applications, this will be little more than the quantisation and noise errors in the measurement transducers. In the implemented version, this accuracy exceeds the output PWM precision. For a general-purpose controller, it would appear that the endpoint control is a satisfactory alternative to the less stable alternative of average current control.

4.2.3 Tracking Performance

The transfer function developed in Section 4.2.1 may be used to analyse the ability of the controller to track a sinusoidal reference. The steady-state frequency response is found by substituting $z := e^{j\omega\Delta}$ where ω is the frequency of interest, and Δ is the sampling period.

$$G(z)|_{z=e^{j\omega\Delta}} = \left. \frac{z^2 \rho \Delta L}{z^2 + (\Delta L(\rho+1) - 2)z + (1 - \Delta L)} \right|_{z=e^{j\omega\Delta}}. \tag{4.20}$$

The performance of a linear system is often displayed in the form of a Bode plot, which displays the output magnitude and phase change as a function of the input frequency. Figure 4.6 shows the reference to output gain for a given reference frequency of 50Hz. This is for an endpoint controller, with a control frequency of 3kHz, and for two different values of inductance estimation errors, $\Delta L = 0.5$ and $\Delta L = 1.2$. Because it is a linear model, the output is a sinusoid with the same frequency as the input. The only change is in the amplitude and phase, which is given by the magnitude and phase of $G(e^{j\omega\Delta})$.

The case of no parameter error is not shown, as both the magnitude and phase responses are perfectly flat in that case. A one time-step delay might be expected from the controller so that causality may be obtained. This does not appear in the transfer function because the reference is defined as the desired current at the end of the control interval. This was done to separate behaviour that is specific to this controller from the delays associated with all discrete-time control.

System Gain Error

Even with the errors in inductance estimation, the responses are very flat for frequencies in the usable range of up to 100Hz. The effect of the parameter estimation is now

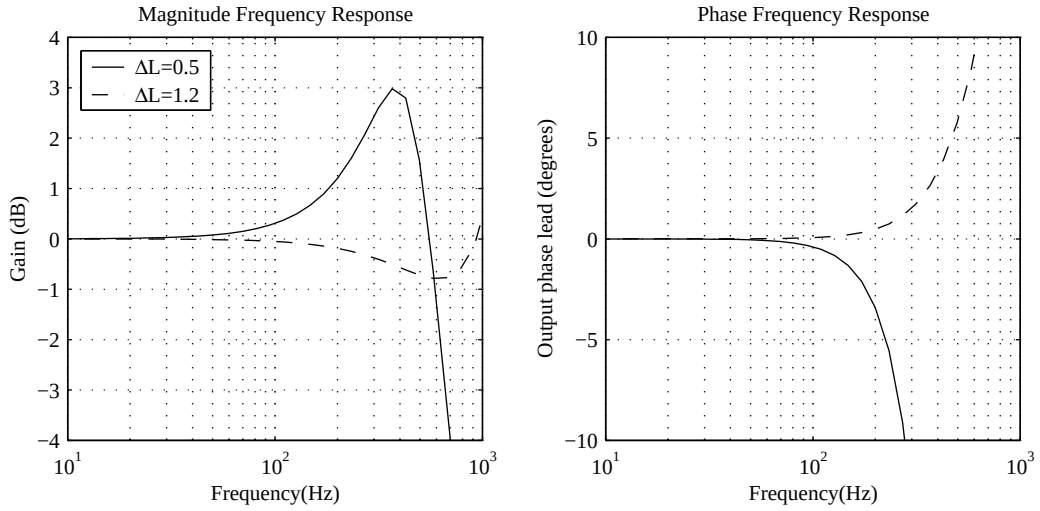


Figure 4.6: Sinusoidal magnitude frequency response of the closed-loop system.

considered for the standard operating frequency.

Figure 4.7 shows the reference to output gain for a given reference frequency of 50Hz. This represents the same endpoint controller used above, but now the magnitude is plotted over a range of inductance estimation errors. The value on the y-axis indicates the ratio of the output amplitude over the input amplitude.

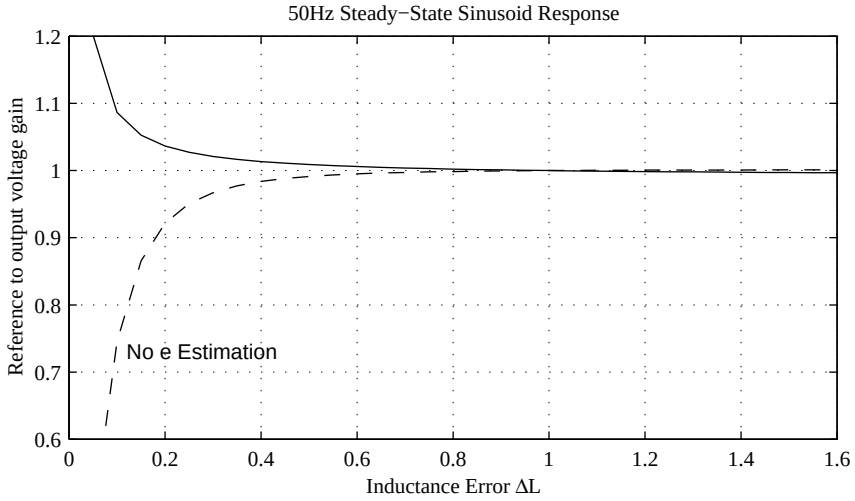


Figure 4.7: Sinusoidal magnitude response of the closed-loop system.

In this case, the controller gain behaviour is shown with the solid line. For values of ΔL near 1 (no error), the output amplitude is equal to the reference amplitude. As the estimated inductance decreases, the gain of the system increases. With the inductance estimated at only one tenth of the true value, the output gain is 1.1. For more reasonable estimates, the result is very close to the correct value.

This increased gain with a lower estimate of L_l is counter intuitive. One would expect that with a lower value of L_l, the controller would be less aggressive, knowing that fewer volts are necessary to make a given change in current. However, the additional gain is a product of the back-emf estimation stage. The controller may instead be formulated without the back-emf estimation. In this case, the controller and machine equations are:

$$v_{k+1} = \frac{\rho L_l \Delta L}{T}\left(u_{k+1} - i_k\right) + e_{k+1} \tag{4.21}$$

$$v_k = \frac{L_l}{T}\left(i_k - i_{k-1}\right) + e_k. \tag{4.22}$$

By combining these equations, the system is described by a first-order transfer function,

$$i_{k+1} + \left(\rho\Delta L - 1\right) i_k = \rho\Delta L u_{k+1} + \frac{T}{L_l}\left(e_{k+1} - e_k\right) \tag{4.23}$$

$$(z + \rho\Delta L - 1)i = z\rho\Delta L u + \frac{T}{L_l}(z-1)e. \tag{4.24}$$

The broken line in Figure 4.7 shows the performance of this system. In this case, using the true back-emf in place of the estimated one causes a greater dependence on the accuracy of the inductance parameter.

System Phase Error

By considering the phase of Equation (4.20), the phase delay between the reference and the output may be found. Figure 4.8 shows this phase error as a function of inductance estimation error. With the correct inductance, perfect phase may be obtained, but a phase lag occurs if the estimated value is too small.

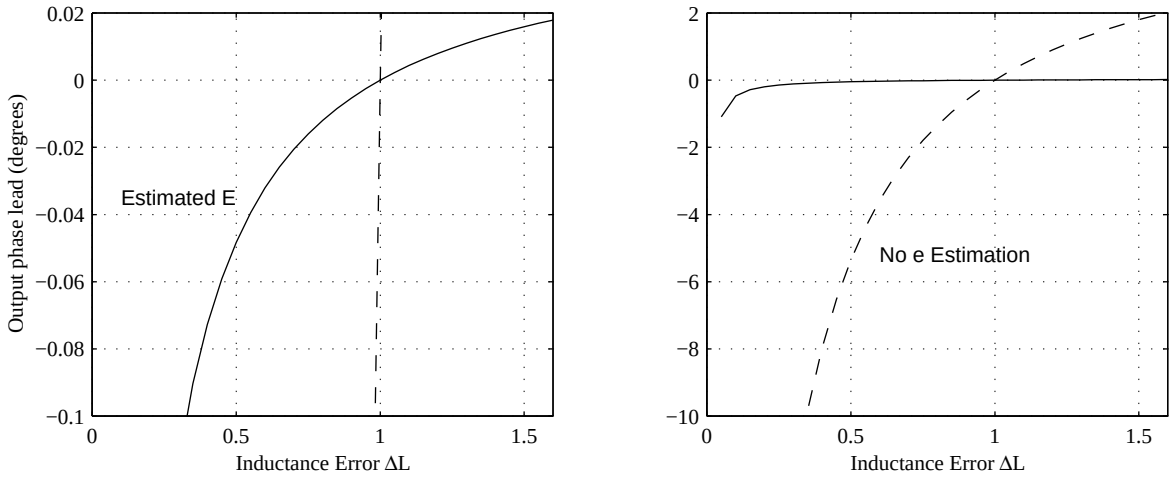


Figure 4.8: Sinusoidal phase response of the closed-loop system.

The broken line in Figure 4.8 shows the behaviour of the system when the true value of the back-emf is used in place of the estimated quantity. In this case, the lack of the

estimator leaves the controller far more susceptible to errors in the L_l parameter.

It is quite clear from this that an error in L_l causes a corresponding error in the e^{est}. This error in e^{est} serves to compensate for the L_l error when it is subsequently used for control purposes. To observe the relationship between the two estimates in the presence of error, the estimator (4.4) and the system (4.6) should be compared. Taking the difference between these equations for a given time-step,

$$e_k^{est} - e_k = v_k - v_k - \frac{L_l \Delta L}{T}(i_k - i_{k-1}) + \frac{L_l}{T}(i_k - i_{k-1}) \tag{4.25}$$

$$= (1 - \Delta L)\frac{L_l}{T}(i_k - i_{k-1}) \tag{4.26}$$

$$e_k^{est} = (1 - \Delta L)v_k + \Delta L e_k. \tag{4.27}$$

This occurs in both the d and q axes, so in the two-dimensional vector case e_k^{est} lies on the line joining v_k and e_k. Furthermore, it should be noted that if v_k and e_k were constant this translation would result in zero error.

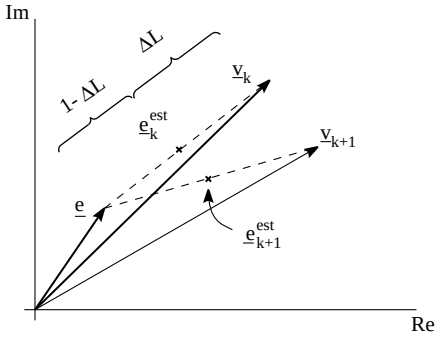


Figure 4.9: Vector diagram of translated back-emf estimate.

The only reason error occurs as a result of an inductance estimation error is that one of the controller assumptions about e^{est} is broken. This assumption is that this value should remain largely constant from one cycle to the next. The value of e_k only varies slowly, but v_k can change rapidly as a result of control action. As a result, the closer e^{est} moves to v_k, the greater the violation of this assumption.

From this analysis, the controller behaviour as a function of the error in the L_l estimate can be found. Figure 4.9 shows how $\underline{e}^{est}$ relates to the back-emf and applied voltage vectors. In this diagram, the back-emf is assumed to be constant, and the inductance estimate about half the true value. $\underline{e}_k^{est}$ is the estimated value at time-step k, while $\underline{e}_{k+1}^{est}$ is what it will be at the following time-step.

For zero-error controller operation, the value $\underline{e}_{k+1}^{est}$ should be used when calculating the control at time-step k. This value will exactly compensate for the error ΔL in the inductance. However, only $\underline{e}_k^{est}$ and $\underline{e}_k$ are available, which represent the estimated and true back-emfs respectively. Provided the change in $\underline{v}$ is relatively small, the estimated value is much better suited than the true value.

Closed-Loop Step Response

The response of the system to a step change in the reference may be found by calculating each time-step from the closed-loop transfer function (4.20). For the case of zero inductance error, dead-beat control is obtained, meaning that the final value is obtained at the end of the first control interval.

With parameter estimation errors, dynamics will appear in the output step response. The case of $\Delta L = 0.8$ is shown in Figure 4.10.

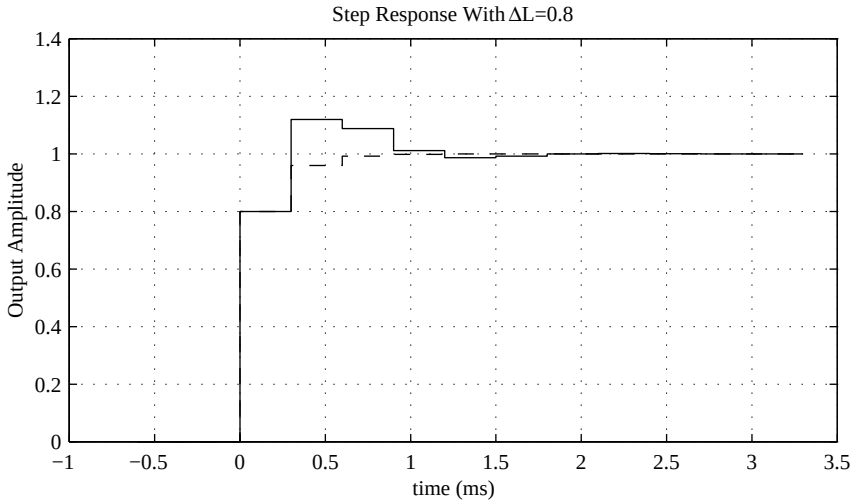


Figure 4.10: System step response with an inductance estimation error.

Again, the response obtained with the true back-emf is shown as the broken line for comparison. The value of first step can be directly related to the value of ΔL. Due to the prior steady-state conditions, the inductance error has no effect on the back-emf estimate, so both responses are the same. Because the change in current is proportional to the the leakage inductance, the relative error in the first step is the same as that in L_l.

In the case without back-emf estimation, this process continues, with the geometric response $y = 1-0.2^k$. However, after the first time-step of the step response, there will be an error in the back-emf estimate due to ΔL. As the current was less than expected, the estimator assumes that back-emf is more negative than it really is. Subsequently, there is overshoot in the second time-step as the controller attempts to compensate for this additional expected back-emf. Finally, the response settles to the correct steady-state value, as expected from the preceding analysis.

4.3 Time-Variations in Back-emf

Throughout the analysis in the previous section, the machine back-emf was considered to be a disturbance, and so did not affect the accuracy of the controller. However, in

practice the time variation in the machine back-emf does have a significant impact on the controller performance.

In terms of stability, the machine back-emf will have negligible effect on the results presented. This is because it can be considered an open-loop disturbance. While the current controller does affect the back-emf, the bandwidth of the closed loop path is very slow compared to the current controller dynamics. Consequently this issue is addressed by the outer loop controllers. The back-emf estimator does have fast dynamics, but the estimator was included in the prior analysis. This is why the effects of back-emf estimation were visible even though the true back-emf was zero for the test.

The original controller derivation included the assumption that the lumped back-emf term could be considered constant from one control interval to the next. Even though the fundamental frequency is much lower than the control rate, there may still be a significant change in machine back-emf across one control interval.

For a control frequency of f_c and a fundamental of f_f, the maximum error introduced by the assumption of a constant back-emf may be estimated by,

$$\begin{aligned}\frac{\Delta e}{e_{peak}} &= \max\ \frac{1}{f_c}\frac{d}{dt}cos(2\pi f_f t) \\ &= 2\pi\frac{f_f}{f_c}.\end{aligned} \tag{4.28}$$

With a fundamental of 50Hz and a control rate of 4kHz, this represents a maximum error of 4% of the peak-to-peak back-emf. This cannot be considered to be negligible, and merits analysis of how this error affects the system performance.

The system transfer function (4.11) derived in Section 4.2.1 includes the terms relating to the true machine back emf. This allows a transfer function to be constructed which defines how the the back-emf affects the controlled current,

$$\frac{i}{e}(z) = \frac{-(z-1)z\frac{T}{L_l}}{z^2 + (\Delta L(\rho+1) - 2)z + (1-\Delta L)}. \tag{4.29}$$

The same analysis may be performed on this transfer function as that from the reference to the output. The stability results will be identical, as these transfer functions share a common denominator. While the phase response is important in the current case, it is not directly relevant here, because no phase relationship has been assumed between the fundamental machine current and the back-emf.

Of primary interest is the magnitude frequency response. This describes the amplitude of current that is caused by a particular amplitude of back-emf. As it is a linear system, this value simply adds to the current obtained from the previous results to obtain the combined effect.

The magnitude response for a fixed fundamental electrical frequency, and varied controller parameters is shown in Figure 4.11. Unlike the the transfer functions involving

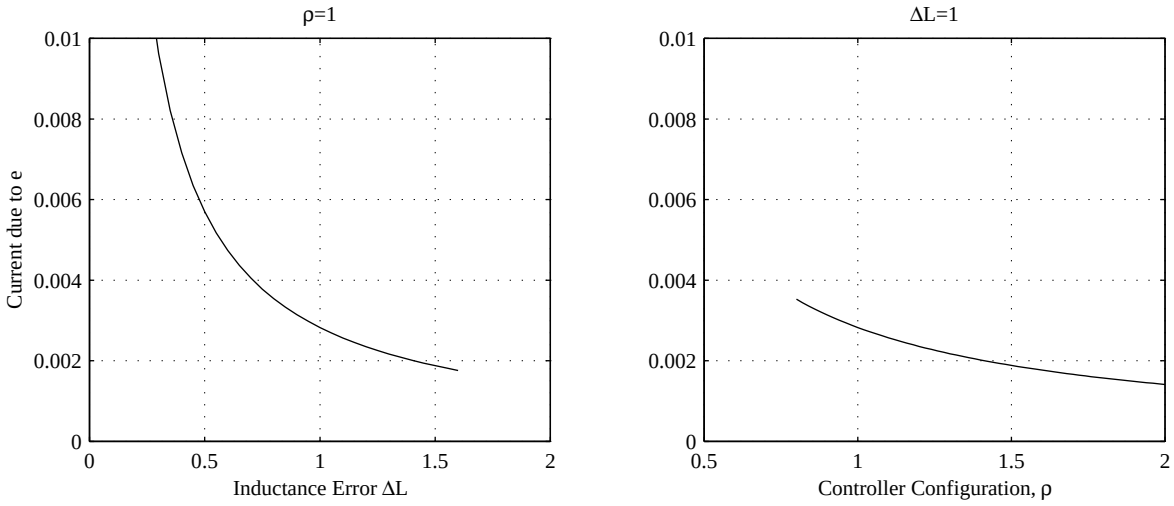


Figure 4.11: Disturbance due to back-emf error at 50Hz.

only the current, the results here depend highly on the machine parameters. The parameters used for this analysis are shown in Table 4.1.

Parameter	Value
Electrical Frequency	50Hz
Control Period	0.3ms
Leakage Inductance	10mH

Table 4.1: Test machine parameters.

With endpoint control and the correct value for inductance, the gain from the back-emf to the output is approximately 3mS. For a sinusoidal back-emf of amplitude 200V, this corresponds to a peak current error of 600mA. Under normal machine operation, this could easily represent a 10% error in the machine current.

The results of a simulation of the controller coupled to a full d-q model of the machine is shown in Figure 4.12. Here, the staircase waveform shows the current reference for a given control cycle. For correct matching, the machine current should coincide with this at the end of each interval.

The correct matching occurs toward the end of the plot window, at $t = 1.895s$. However, shortly after the peak, there is a significant error in the setpoint tracking. The peak error is about 500mA, as expected from the analysis.

The effect of the changing back-emf is the biggest shortcoming of the stationary-frame controller. However, there are a number of methods that may be used to reduce the effect of this error. These methods vary in effectiveness, side-effects and complexity. The following sections describe these.

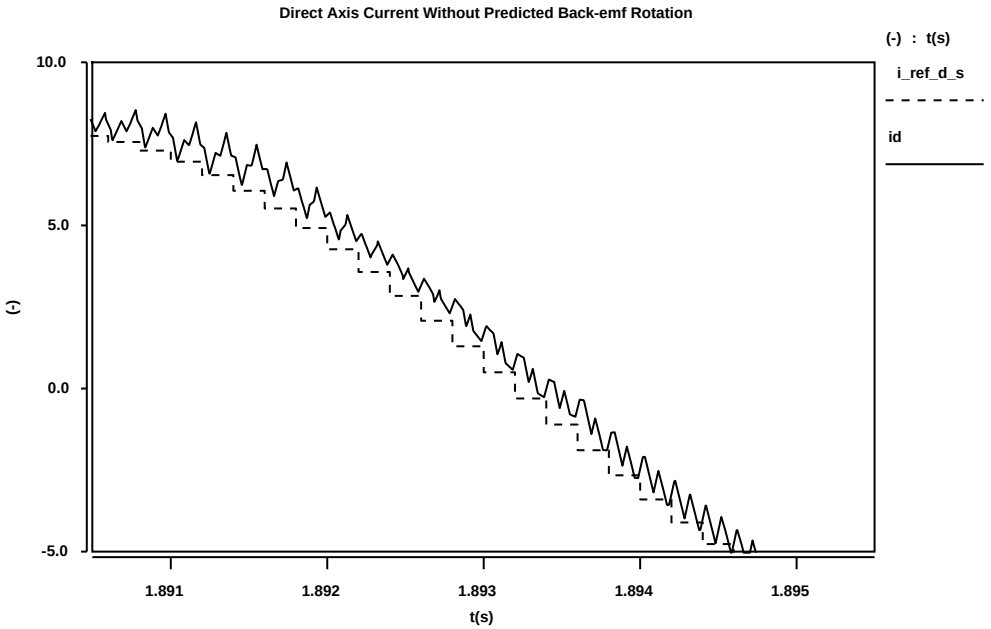


Figure 4.12: Simulation of the controller with feed-forward back-emf compensation.

4.3.1 Back-emf Extrapolation

The basic problem with the variation in the back-emf is that the estimates are delayed due to causality constraints. Essentially, the problem is with Equation (4.5), which is simply a parameter time-shift,

$$e_{k+1}^{pred} = e_k^{est}. \tag{4.30}$$

This is based on the observation that back-emf is slowly changing, and the existing interval will adequately approximate the next. The improved back-emf compensation methods involve improving the accuracy of this prediction equation.

Linear extrapolation

A closer approximation may be made by using linear extrapolation on the back-emf estimates. This assumes that the change in back-emf across the interval is a constant, rather than the back-emf itself being a constant.

$$e_{k+1}^{pred} = e_k^{est} + \left(e_k^{est} - e_{k-1}^{est}\right) \tag{4.31}$$

$$= 2e_k^{est} - e_{k-1}^{est}. \tag{4.32}$$

This relies on the second derivative with respect to the sampling period being smaller than the first. If T is the control period, the first derivative is

$$\frac{de}{dk} = \frac{d}{dt} A sin(\omega kT) \tag{4.33}$$

$$= A\omega T cos(\omega kT). \tag{4.34}$$

In this case, the relative change is ωT. For a 50Hz fundamental and a 3kHz control rate, this value is approximately 0.1. This value of 0.1 is roughly equivalent to the 10% error found in the current above. For the case of linear extrapolation, the error is caused by the change in the second derivative,

$$\frac{d^2e}{dk^2} = \frac{d^2}{dt^2} A sin(\omega kT) \tag{4.35}$$

$$= -A(\omega T)^2 sin(\omega kT). \tag{4.36}$$

This time the relative change is $(\omega T)^2$. For the 50Hz example, the error will be approximately only one tenth of that without the extrapolation.

These results suggest that linear extrapolation is a good substitute for the simpler method originally presented. One disadvantage is the additional calculation required, but in reality this overhead is relatively small. On the final implementation, for each axis, it involves storing and retrieving an additional past value of current, together with a few extra addition operations.

Controller Performance

Of greater significance is how this change affects the other controller performance attributes. This change in the emf-prediction increases the order of the system dynamics from two to three. In doing so, the stability properties and reference tracking ability is affected.

To obtain the new controller specification, Equation (4.32) may be combined with the existing controller equations (4.3) and (4.4). The system is described by the equations,

$$v_{k+1} = \frac{\rho L_l \Delta L}{T} (u_{k+1} - i_k) + 2e^{est}_k - e^{est}_{k-1} \tag{4.37}$$

$$e^{est}_{k+1} = v_{k+1} - \frac{L_l \Delta L}{T} (i_{k+1} - i_k). \tag{4.38}$$

which may be simplified into a single updating equation,

$$v_{k+1} = 2v_k - v_{k-1} + \frac{L_l \Delta L}{T} (\rho u_{k+1} - (2+\rho) i_k + 3i_{k-1} - i_{k-2}). \tag{4.39}$$

Equation (4.39) would be used to implement this controller. This is a similar, but higher order, equation to the one for the basic controller in (4.7).

The closed loop discrete-time transfer function is obtained from (4.39) together with

the machine model in (4.6). It may be shown that in forward shift operator this is,

$$\begin{aligned} & i\left(z^3 + (\Delta L(\rho+2) - 3)z^2 + 3(1-\Delta L)z + \Delta L - 1\right) \\ & = z^3\rho\Delta L u - (z-1)^2 z\frac{T}{L_l}e. \end{aligned} \tag{4.40}$$

The same analysis techniques may be applied to this controller as the original controller. In particular, the transfer function from the machine back-emf to the current reveals the effectiveness of the revised prediction scheme. This is plotted as a function of the inductance error in Figure 4.13.

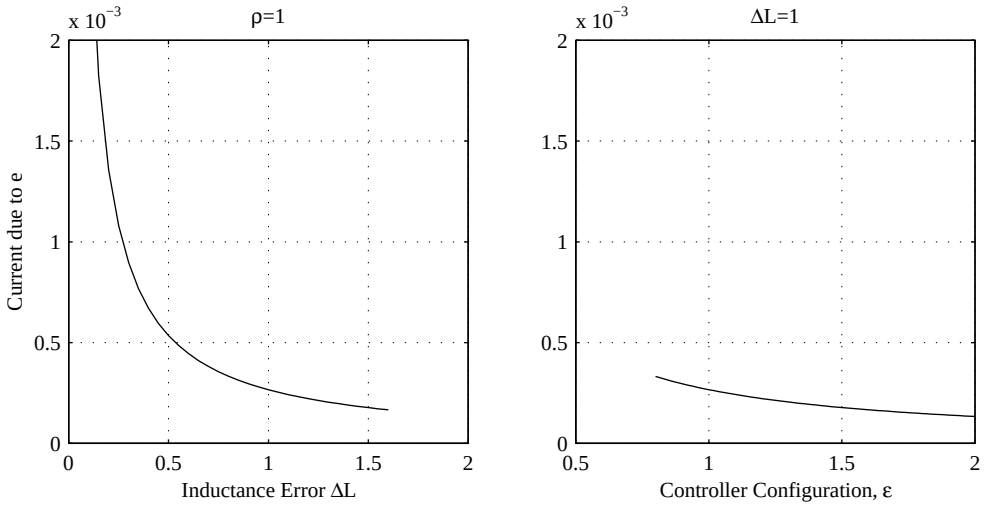


Figure 4.13: Disturbance due to back-emf error at 50Hz.

The overall graph shape is similar to the previous case, but now the gain is much lower, at about 0.25mS. For the same 200V back-emf, the resulting error in current is only 50mA. This means that the expected reduction to one tenth of the original error is realised.

The extrapolation offers much better disturbance rejection properties, but this does come at a price of system stability and parameter estimation tolerance. The extrapolation scheme involves more closely modelling the system, and so more is susceptible to problems when there is a model error.

The root-locus plot for the controller with the extrapolation included is shown in figure 4.14. Again, endpoint control is used. In this case, the available range of L_{est} for a stable system is significantly smaller. In particular, it is now necessary to ensure that the inductance estimate is at least half of the true inductance.

The magnitude of the maximum pole location magnitude as the estimation error changes is shown in Figure 4.15. The warping of the pole locations with the addition of the extra dynamics reduces the stability margin.

The system is only stable for ΔL in the range of $0.5 < \Delta L < 1.2$. The consequence

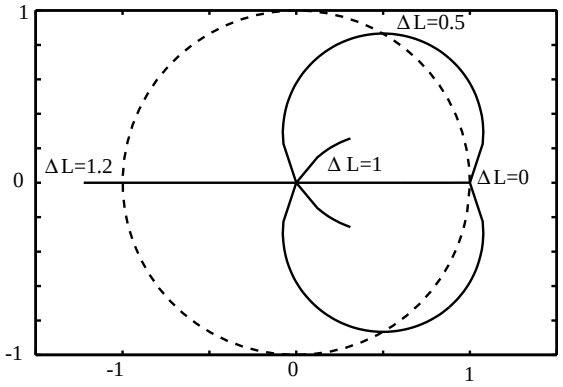


Figure 4.14: Root-locus plot for the controller with back-emf extrapolation.

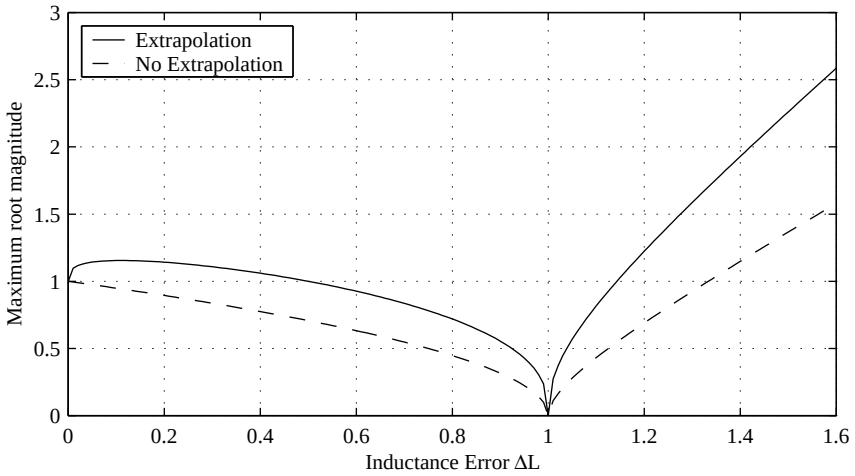


Figure 4.15: Maximum root magnitude as a function of the inductance estimation error with back-emf extrapolation.

of this is that the inductance estimate must be close to the true value, and large errors in either direction result in instability. The advantage of the controller without the extrapolation is that a low inductance estimate will be stable, even if the performance is degraded. This is particularly helpful at start-up when the inductance may not be well known. Overall, while performance may be improved by using the linear extrapolation, it is more sensitive to errors in the inductance estimate.

4.3.2 Rotation Feed-forward

The extrapolation method described in the previous section operated purely on each axis separately. Further alternatives may be found by utilising additional information from the system. In particular, it is known that the back-emf space vector will generally follow a circular trajectory at the electrical rotation frequency. This information may be used to better predict the path of the space vector over the control interval. Figure 4.16 shows

the assumed back-emf geometry.

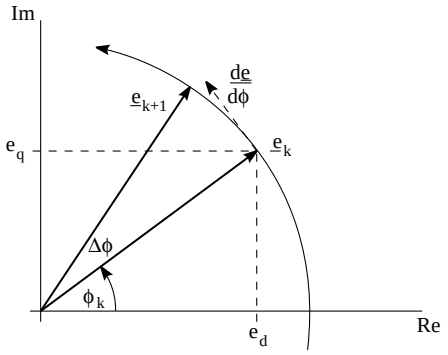


Figure 4.16: The back-emf trajectory.

The direct and quadrature axis back-emf components may be derived from the diagram to be,

$$e_{dk} = e_k \cos(\omega T k) \tag{4.41}$$

$$e_{qk} = e_k \sin(\omega T k). \tag{4.42}$$

If the value of $\Delta\phi$ is known, the value of $\underline{e}_{k+1}$ can be quite accurately predicted from $\underline{e}_k$. However, the value of $\Delta\phi$ is not well known by the current controller. While it is true that many quantities are changing at the electrical synchronous frequency, the angular frequency is not known directly by the controller. An alternative would be to assume that this quantity is known from another source. Typically the reference electrical rotation speed is well known in a field-oriented torque controller, for example. As it is the role of the torque controller to control the angle of the back-emf, it will not even be a predicted quantity, but rather a control setpoint. As such it will not suffer from the errors in forward prediction.

For the remainder of this section, it will be assumed that the projected change in angular displacement of the back-emf space vector is known. If it cannot be derived, this method is not directly suitable for the application. The modifications detailed in the following section (§4.3.3) remove the need for this information.

Implementing Feed-forward

It is assumed that the expected change in back-emf angle is known for the given control cycle,

$$\Delta\phi_k = \phi_{k+1} - \phi_k. \tag{4.43}$$

With this knowledge, a simple transformation exists to re-map the estimated back-emf coordinates by the adjustment angle. From basic trigonometry, this is,

$$\begin{bmatrix} e_{dk+1}^{pred} \\ e_{qk+1}^{pred} \end{bmatrix} = \begin{bmatrix} \cos(\Delta\phi) & -\sin(\Delta\phi) \\ \sin(\Delta\phi) & \cos(\Delta\phi) \end{bmatrix} \begin{bmatrix} e_{dk}^{est} \\ e_{qk}^{est} \end{bmatrix}. \tag{4.44}$$

This approach will indeed work. However, one of the aims of performing the control in the stationary frame is to reduce the number of these types of transformations necessary. It is possible to instead calculate a close approximation without the need for the trigonometric lookups. This is done by performing a linear approximation to (4.44).

For small angles of θ, the following approximations hold,

$$\cos(\theta) \approx 1 \tag{4.45}$$

$$\sin(\theta) \approx \theta. \tag{4.46}$$

These approximations are very close for the range range of θ under consideration. For the example of a 50Hz fundamental with a 3kHz control rate, $\Delta\phi$ will be of the order of 0.1. In this case, the approximations are accurate to less than 1%. As it is further scaled against the 10% error in the measurable quantities, this approximation is adequate.

With the approximation, the transformations in (4.44) become a simpler expression,

$$\begin{bmatrix} e_{dk+1}^{pred} \\ e_{qk+1}^{pred} \end{bmatrix} \approx \begin{bmatrix} 1 & -\Delta\phi \\ \Delta\phi & 1 \end{bmatrix} \begin{bmatrix} e_{dk}^{est} \\ e_{qk}^{est} \end{bmatrix}. \tag{4.47}$$

In this form, the only additional calculation is a multiplication of the back-emf estimates with the supplied $\Delta\phi$, and the associated additions. The resulting prediction equations differ between two axes, and may be implemented as,

$$e_{dk+1}^{pred} = e_{dk}^{est} - \Delta\phi e_{qk}^{est} \tag{4.48}$$

$$e_{qk+1}^{pred} = e_{qk}^{est} + \Delta\phi e_{dk}^{est}. \tag{4.49}$$

Performance Analysis

Analysis of the properties of this controller depends heavily on the value of $\Delta\phi$ supplied. However, it is reasonable to expect that it is a slowly-changing quantity with respect to the current controller dynamics. In this case, the stability of the controller will be unaffected, as the adjustment could be considered an open-loop disturbance.

The best approximation within the linear analysis framework is to assume that the adjustment angle is correct. This would result an accurately compensated back-emf term,

$$e_{k+1}^{pred} = e_{k}^{est} + \zeta\left(e_{k+1} - e_{k}\right). \tag{4.50}$$

The constant ζ indicates the accuracy of the supplied change in angle. This parameter

can also be used to consider the effect of a change in the magnitude of the back-emf. The changing magnitude is a less significant problem, but there will still be small changes over time. A value of $\zeta = 1$ indicates that the matching is perfect. For the case of an error in the supplied angle, ζ is defined as,

$$\zeta = \frac{\Delta\phi^{pred}}{\Delta\phi}. \tag{4.51}$$

By combining (4.50) with the controller Equations (4.3) and (4.4), the controller equation may be found,

$$\begin{aligned} &i_{k+1} + ((\rho+1)\Delta L - 2)\, i_k + (1-\Delta L)\, i_{k-1} \\ &= \rho\Delta L u_{k+1} + \frac{T}{L_l}\,(e_{k+1} - e_k)\,(1-\zeta). \end{aligned} \tag{4.52}$$

This controller behaviour is identical to the original one, except for the addition of the $(1-\zeta)$ term in the back-emf component. In the specification $\zeta = 0$ corresponds to no compensation. As expected, the expression reduces to the original equation with $\zeta = 0$.

For all values of ζ, there is no effect on the reference tracking or stability dynamics. It only has an impact on the magnitude of the error due to the back-emf. For perfect compensation, $(\zeta = 1)$, the change in back-emf has no effect on the controller output.

This controller adjustment could be considered analogous to the rotating frame PID controller operating in stationary coordinates[40], described in Section 2.4.2. It offers the steady-state tracking advantages of a rotating-frame controller, but without the requirement for the coordinate transformations. However, as with the PI controller, knowledge of the electrical rotation velocity is required.

Simulation Results

The merits of feed-forward scheme are difficult to show analytically because of the dependence on the supplied rotation parameter. An alternative is to simulate a number of trial situations to observe whether the result matches the expectations.

Figure 4.17 shows the tracking ability of the revised controller for the direct-axis current. This is equivalent to the simulation of the original controller shown in Figure 4.12.

In this case, the tracking ability is greatly improved. Perfect matching occurs for almost the entire region of the plot. The exception is an area near $t = 1.9037s$, where there is a short period of deviation. This deviation coincides with the zero-crossing of another phase and is almost certainly due to the effects of inverter dead-time.

The performance of the back-emf predictor itself can be examined by comparing the predicted value of the back-emf to the estimated value obtained in the subsequent cycle. This comparison is shown in Figure 4.18.

The predicted value of the back-emf is very close to the estimated value in the fol-

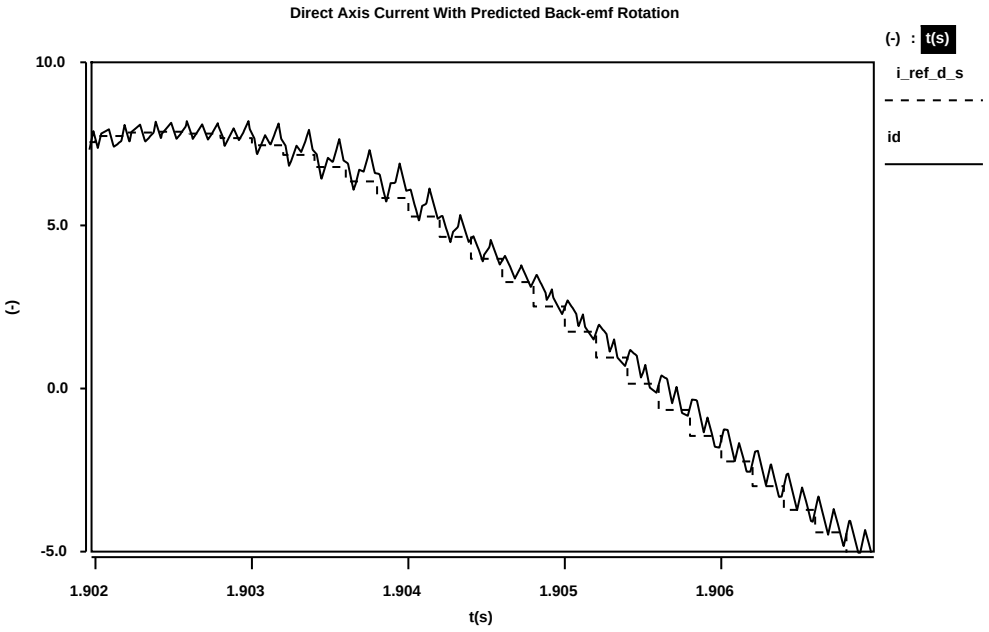


Figure 4.17: Simulation of the controller with feed-forward back-emf compensation.

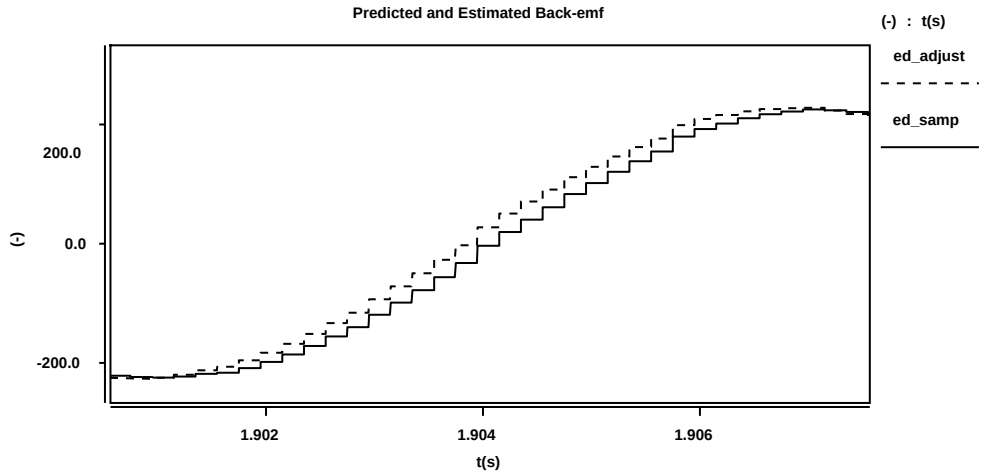


Figure 4.18: The predicted and estimated d-axis back-emf.

lowing cycle. This is the correct behaviour for the predictor. It is quite evident that the predicted value is a significantly better approximation than simply using the previous value obtained.

Figure 4.19 shows the transient response of the simulated controller and machine. This was obtained by applying a step setpoint to the torque controller. The result is a step reference for the quadrature-axis current. The simulation shows good tracking performance on the direct axis current, and there is no evidence of degradation resulting from the feed-forward mechanism. There is some coupling of the axes, but in comparison to the step size, this is reasonably small. This coupling effect is due to saturation of the

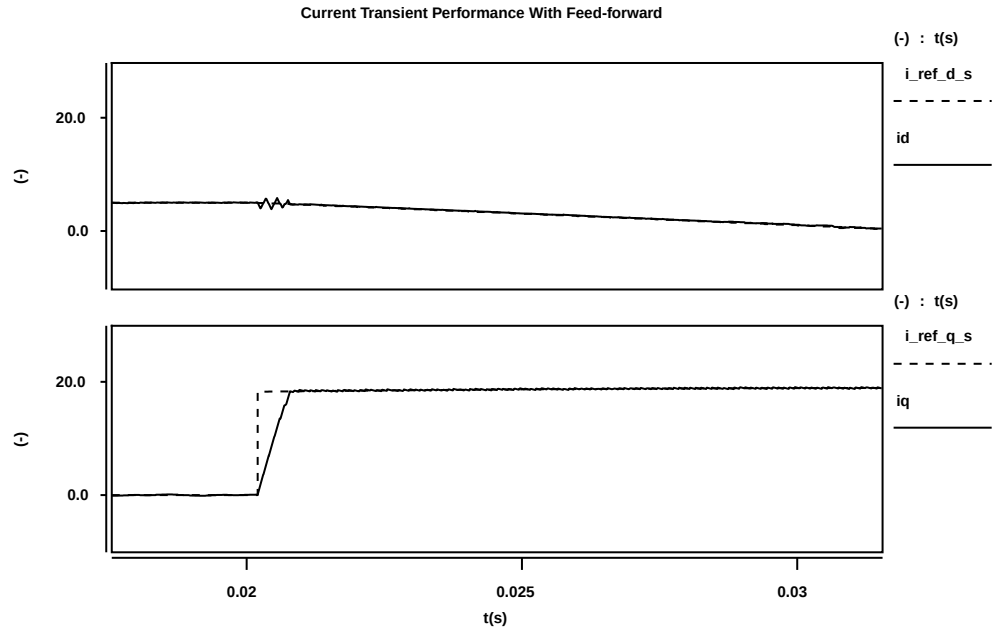


Figure 4.19: Transient response of the modified controller.

controller output, which limits the voltage that may be supplied to the machine.

Simulations were also performed with the introduction of error in the leakage inductance parameter. In this case, the inclusion of the feed-forward term does not further degrade the controller. The case for a very low controller leakage inductance parameter is shown in Figure 4.20. With the error in the leakage inductance, the feed-forward still

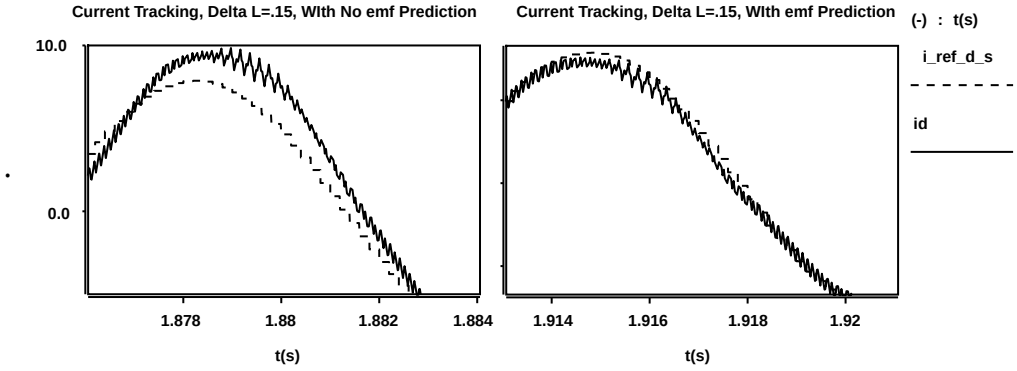


Figure 4.20: Machine current with significant inductance estimation error.

offers improvements in the tracking ability of the controller. In this case, the improvement is greater. This greater improvement is to be expected from the results shown in Figure 4.13. In that section(§4.3), it was shown that the controller is more susceptible to changes in the back-emf when $\Delta L \ll 1$. Consequently, the improvement offered by the feed-forward is greater under these circumstances.

Overall, the simulation results support the analytic expectations of the controller.

It offers the dynamics of the original controller, but with a great reduction in the error introduced by the variation in machine back-emf.

4.3.3 Observer Based Rotational Adjustment

The principal drawback of the feed-forward method described in Section 4.3.2 was that knowledge of the electrical rotation speed was required. At the cost of additional complexity, this information can instead be provided by an observer within the current controller. The observer removes the requirement for additional information to be passed to the controller. This section describes the development and analysis of such an observer.

The basic philosophy behind the observer is to use the alternate phase (direct or quadrature) to obtain derivative information. This derivative provides a coupling between the rotation of the back-emf vector and the estimated quantities. The basic objective is to use this coupling to perform linear extrapolation, but with the the angular displacement of $\underline{e}$ as an intermediate parameter. The advantage of explicitly evaluating the rotation speed is that this quantity is slowly changing, and thus can easily be filtered without phase-delay problems.

In the previous section, a predictor was developed to estimate future values for the back-emf, based on the change in back-emf angle across a control cycle. This resulted in the equations,

$$\begin{aligned} e_{dk+1}^{pred} &= e_{dk}^{est} - \Delta\phi e_{qk}^{est} \\ e_{qk+1}^{pred} &= e_{qk}^{est} + \Delta\phi e_{dk}^{est}. \end{aligned} \tag{4.53}$$

These may be re-arranged to obtain an estimate for $\Delta\phi$,

$$\Delta\phi_k^{est} = \frac{-e_{dk}^{est} + e_{dk-1}^{est}}{e_{qk}^{est}} \tag{4.54}$$

$$\Delta\phi_k^{est} = \frac{e_{qk}^{est} - e_{qk-1}^{est}}{e_{dk}^{est}}. \tag{4.55}$$

This offers two different estimates of the change in angle of the $\underline{e}$. However, due to the division operation, a good approach would be to use the equation which involves division by the larger quantity.

Errors are to be expected in this estimation, particularly as it involves a differentiation structure. However, the quantity being estimated is slowly-changing with respect to the control frequency, and so may be filtered. The filter used in the analysis is,

$$\Delta\phi_{k+1}^{filt} = (1-\epsilon)\Delta\phi_k^{filt} + \epsilon\Delta\phi_k^{est}, \tag{4.56}$$

where ϵ controls the amount of data filtering. $\epsilon = 1$ represents no filtering of the estimate. This filtered estimate may be used in place of the torque controller supplied value that

was used in the feed-forward controller. Equation (4.53) is used in the prediction stage, with the filtered value for $\Delta\phi$.

Observer Transfer Function

Unlike the basic feed-forward controller, the introduction of the observer introduces additional dynamics into the system. These dynamics may be analysed by forming a transfer function for the revised system. The primary difficulty is to find a linear model of the new estimator, preferably decoupled between the d and q axes.

From Equation (4.53), the quantity required for the predictor in the d axis is $e_{qk}^{est}\Delta\phi_k^{filt}$. Let this be defined as,

$$\gamma_k \triangleq e_{qk}^{est}\Delta\phi_k^{filt}. \tag{4.57}$$

Using (4.56),

$$\Delta\phi_k^{filt} = (1-\epsilon)\Delta\phi_{k-1}^{filt} - \epsilon\left(\frac{e_{dk}^{est} - e_{dk-1}^{est}}{e_{qk}^{est}}\right) \tag{4.58}$$

so,

$$\gamma_k = (1-\epsilon)\Delta\phi_{k-1}^{filt}e_{qk}^{est} - \epsilon\left(e_{dk}^{est} - e_{dk-1}^{est}\right) \tag{4.59}$$

$$= (1-\epsilon)\left[\gamma_{k-1} + \Delta\phi_{k-1}^{filt}\left(e_{qk}^{est} - e_{qk-1}^{est}\right)\right] - \epsilon\left(e_{dk}^{est} - e_{dk-1}^{est}\right). \tag{4.60}$$

The quantity $\Delta\phi_{k-1}^{filt}(e_{qk}^{est} - e_{qk-1}^{est})$ is difficult to manage directly, because it relates to quantities in the quadrature axis. It is also non-linear, meaning that a multi-variable analysis does not offer a solution. Instead, an approximation may be made based on the physical meaning of this quantity. By comparing Equation (4.53), $\Delta\phi e_{qk}$ is an estimate of the change in direct-axis back-emf across one control cycle. Now this is a slowly varying quantity, so the change in $\Delta\phi e_{qk}$ across one control cycle will be small. On this basis, it is reasonable to approximate the change in $\Delta\phi e_{qk}$ as being equal to its true physical value. This is,

$$\Delta\phi\left(e_{qk}^{est} - e_{qk-1}^{est}\right) \approx e_{dk} - 2e_{dk-1} + e_{dk-2}. \tag{4.61}$$

Again, because the change in γ is small compared to γ itself, it is reasonable to approximate the true value of $\Delta\phi$ with the filtered estimate, $\Delta\phi^{filt}$.

The result of these approximations is that the predictor may be expressed as,

$$\gamma_k = (1-\epsilon)\left(\gamma_{k-1} + e_k - 2e_{k-1} + e_{k-2}\right) - \epsilon\left(e_k^{est} - e_{k-1}^{est}\right). \tag{4.62}$$

This equation is convenient for analysis because all of the terms refer to the same axis. This expression is for the direct axis, but the quadrature axis is equivalent, except for

sign changes.

Expressed in the shift operator, the full set of controller and machine equations are,

$$\left[z^2 + (\epsilon - 1)z\right]\gamma = (1-\epsilon)\left(z^2 - 2z + 1\right)e - \epsilon\left(z^2 - z\right)e^{est} \tag{4.63}$$

$$ze^{pred} = e^{est} - \gamma \tag{4.64}$$

$$ze^{est} = zv - \frac{L_l \Delta L}{T}(z-1)(i+m) \tag{4.65}$$

$$z(v-e) = \frac{L_l}{T}(z-1)i \tag{4.66}$$

$$zv = ze^{pred} + \frac{\rho L_l \Delta L}{T}(zu - (i+m)). \tag{4.67}$$

The term m represents the output measurement error. This is added to allow evaluation of the susceptibility to errors and noise in measurement. It is defined as,

$$i_k^{meas} = i_k + m_k. \tag{4.68}$$

These equations may be solved for i, u, m and e to obtain the discrete-time transfer function. The result is,

$$\begin{aligned} &u\left[\rho\Delta L z^2(z+\epsilon-1)\right] + \frac{T}{L_l}e\left[-z^3 + (3-\epsilon)z^2 + (2\epsilon-3)z + (1-\epsilon)\right] \\ &\quad = i\left[z^3 + ((\Delta L - 1)(1+\epsilon) + \epsilon + \rho\Delta L - 2)\,z^2\right. \\ &\qquad \left. + ((\Delta L - 1)(-\epsilon - 2) + (\epsilon - 1)\rho\Delta L + 1 - \epsilon)\,z + (\Delta L - 1)\right] \\ &\quad + \Delta L m\left[(1+\epsilon+\rho)z^2 + (\rho(\epsilon-1) - \epsilon - 2)z + 1\right]. \end{aligned} \tag{4.69}$$

One useful observation about this transfer function is that with $\epsilon = 1$, it reduces to the transfer function in the back-emf extrapolation case. This means for that value of the parameter, both the performance and stability attributes will be the same as those of the extrapolation controller. This is not surprising, as the controllers are of the same order, and use the same information for the predictions. The advantage is that, in the case of the observer based controller, the filtering may be used to alter the stability and performance characteristics.

Stability Analysis

The primary aim of developing the observer was to improve the stability over the back-emf extrapolation case. The incorporation of the filter is aimed at decoupling the prediction dynamics from the controller dynamics.

Figure 4.21 shows the root-locus plot for the controlled system. This plot is for the nominal value of $\epsilon = 0.1$. The shape of the root locus is very similar to that of the extrapolation controller shown in Figure 4.14. The change occurs as ϵ moves away from 1, when the location of the locus is distorted.

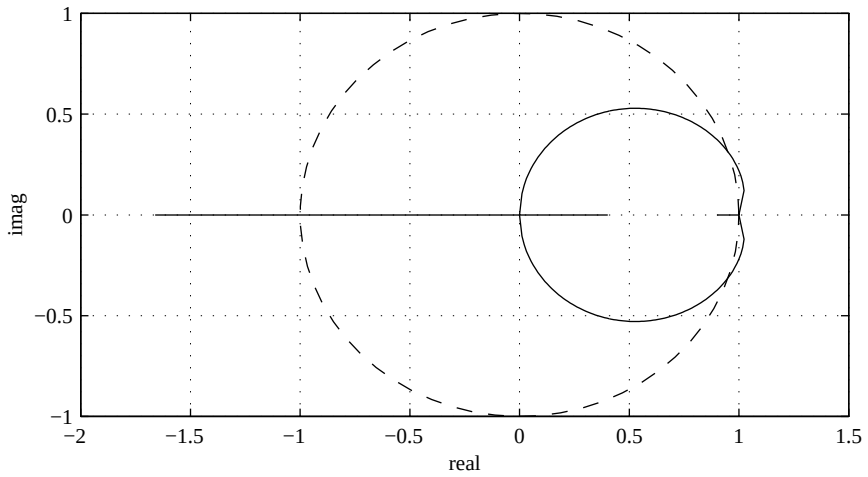


Figure 4.21: Root-locus plot for $\epsilon = 0.1$.

Figure 4.22 shows the magnitude of the maximum root as a function of the error in the leakage inductance parameter. Three cases are shown in this plot. The original controller is shown for reference, and behaviour for $\epsilon = 0.1$ and $\epsilon = 0.5$ is plotted.

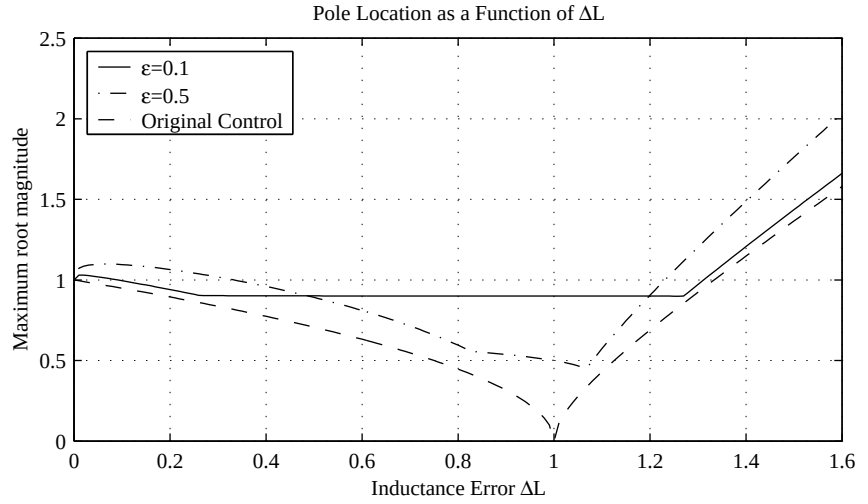


Figure 4.22: Maximum root magnitude as a function of the inductance estimation error.

It can be seen that the range of stability is less than that of the original controller but, as the ϵ parameter is reduced, the stable range approaches that of the original controller. In the case of the $\epsilon = 0.1$, the near-flat section in the centre of the plot is due to the pole located on the real axis. This pole, at approximately $(1-\epsilon)$ represents the filtering mode. Although the magnitude is high, that is necessary for a filter with a long time-constant. The filter parameters are also design parameters, and so are not subject to uncertainty, meaning that the location of this pole does not compromise the stability.

The range of controller stability is shown clearly in Figure 4.23. In this case, The

maximum and minimum values for ΔL are shown as a function of the ϵ parameter.

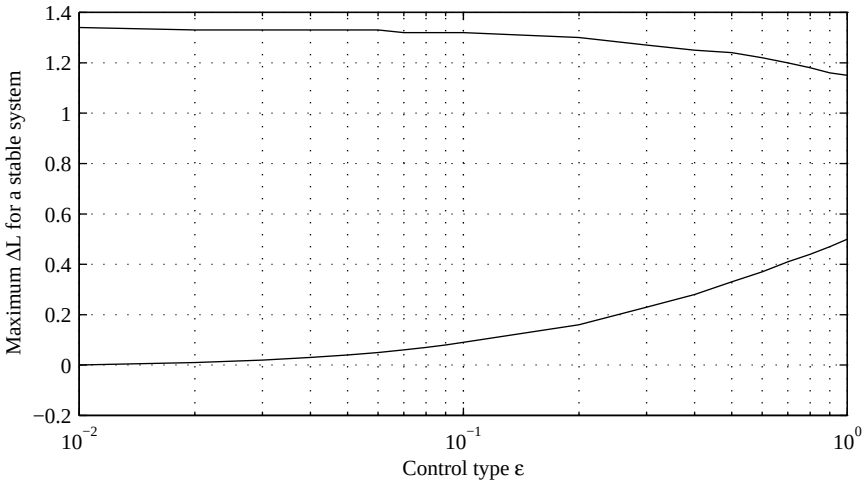


Figure 4.23: Inductance estimation error range for a stable system, as a function of ϵ, ($\rho = 1$).

For values of $\epsilon < 0.1$ the region of stability is very similar to the original controller, while for values approaching 1 it approaches that of the extrapolation controller. A criticism of the extrapolation controller was that there was no safe fall-back value that could be used for L_l when the true value is poorly known. In the original controller, a very small value of L_l^{est} could be used, and one could be confident of stability even if the performance is poor. The observer based controller recaptures this advantage in accepting very small values of $L_l \Delta L$. Even for the conservative value of $\epsilon = 0.1$, the inductance need only be known to within an order of magnitude.

Controller Performance

The magnitude frequency response of the controlled system is shown in Figure 4.24. This is for a constant reference frequency of 50Hz and a control rate of 3kHz. The output amplitude gain is shown as a function of error in L_l. At this frequency, the magnitude of the controller tracking always lies between that of the basic controller, and the one with back-emf extrapolation. The extrapolation controller has a magnitude response that is quite independent of the leakage inductance estimation at this frequency. By setting $\epsilon = 1$, this performance is achieved, and with $\epsilon \to 0$, the performance of the basic controller is achieved. The values between these extremes falls between those limits.

The phase response has different attributes, as shown in Figure 4.25. The original controller had a slightly lagging phase response for low values of ΔL, while the reverse occurs when extrapolation is used.

When using the observer, the phase is likely to be leading at this frequency and $L_l < 1$. For the case of $\epsilon = 0.1$, the phase error is high compared to the existing controllers. The

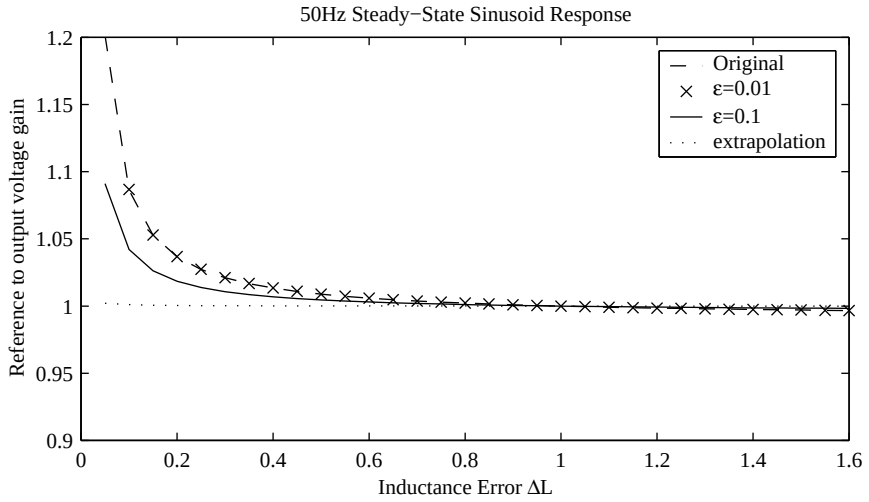


Figure 4.24: Sinusoidal magnitude response of the closed-loop system.

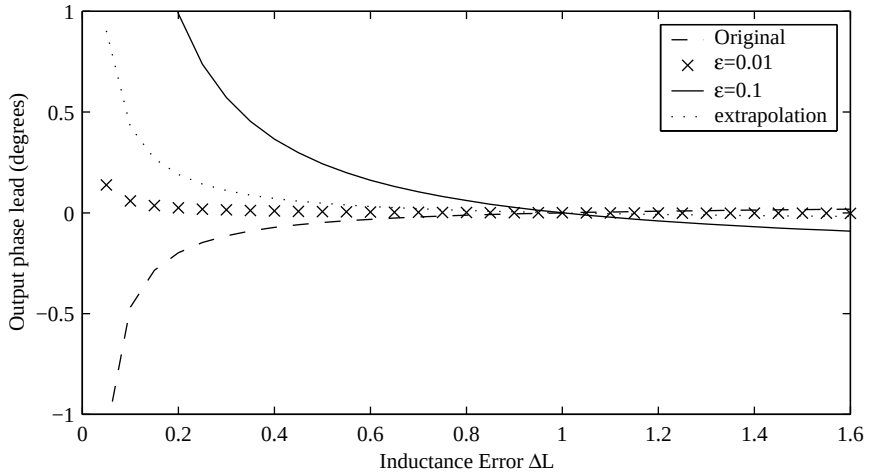


Figure 4.25: Sinusoidal phase response of the closed-loop system.

response dependence on ϵ is shown in Figure 4.26. For 50Hz operation, this shows a peak in the phase at an observer gain of 0.1.

The origin of the peak at ϵ can be seen in the frequency response plotted in Figure 4.27. The extrapolation operation produces a resonance at a high frequency. As the level of filtering is increased, the resonance is flattened out, resulting in a lower peak, but a greater spread in its influence. With $\epsilon < 0.1$, the response is quite similar to that of the original controller.

Overall, the errors represented by these responses are small. For all values of ϵ. with the sample machine parameters and a reasonable inductance estimate, the magnitude was within 2% of the reference, and the phase was correct to about 0.1 degrees.

The step response is also included in Figure 4.28. For $\Delta L = 0.8$ and $\epsilon = 0.1$, the step

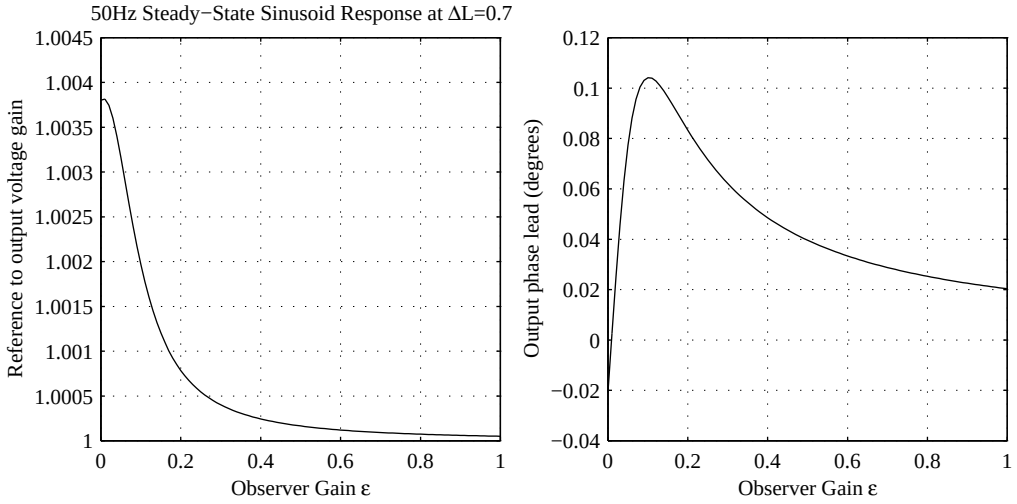


Figure 4.26: Sinusoidal phase response of the closed-loop system.

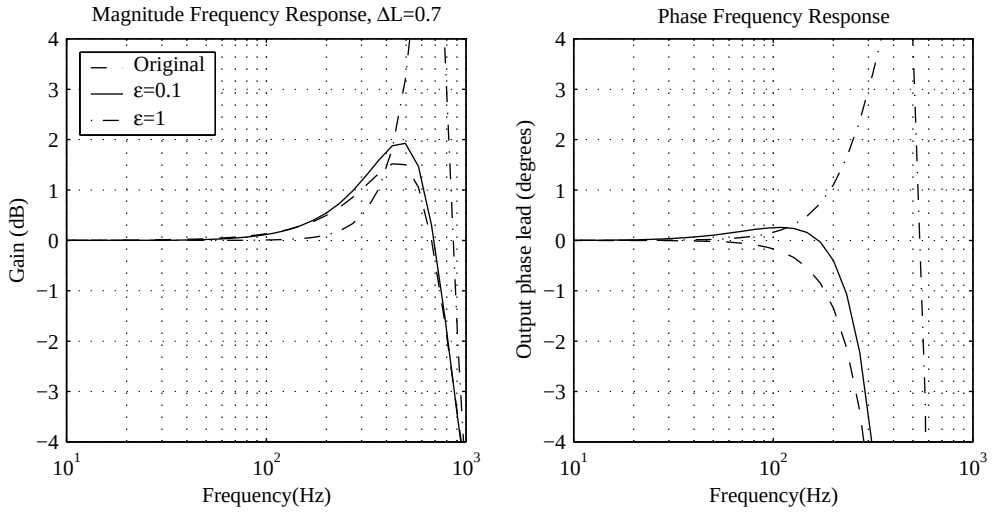


Figure 4.27: Sinusoidal phase response of the closed-loop system.

performance was similar to the basic controller.

As expected, this modified controller performs well at rejecting disturbance due to time variations in the back-emf. In this case the level of disturbance across the range of values for ϵ is almost identical to the controller with back-emf extrapolation. This response is shown in Figure 4.13, and represents a very small effect on the machine current setpoint tracking.

Sensitivity

The controller relies heavily on the measurements of the machine currents. By analysing the response of the system to the measurement error parameter, the sensitivity of the con-

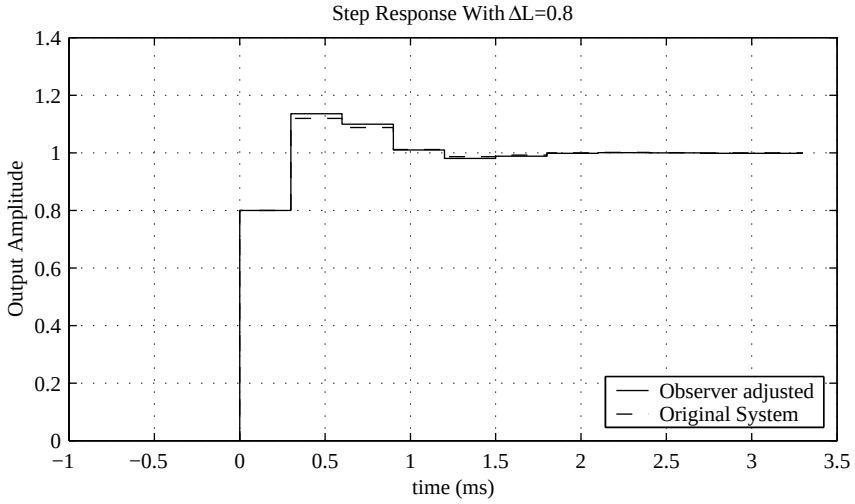


Figure 4.28: System step response with an inductance estimation error.

troller to errors in measurement may be gauged. From Equation (4.69), the measurement noise transfer function may be found,

$$\frac{i}{u} = \frac{\Delta L\left((1+\epsilon+\rho)z^2 + (\rho(\epsilon-1)-\epsilon-2)z + 1\right)}{Q(z)}, \tag{4.70}$$

where $Q(z)$ is the characteristic equation,

$$\begin{aligned} Q(z) = z^3 &+ ((\Delta L-1)(1+\epsilon)+\epsilon+\rho\Delta L-2)\,z^2 \\ &+ ((\Delta L-1)(-\epsilon-2)+(\epsilon-1)\rho\Delta L+1-\epsilon)\,z + (\Delta L-1). \end{aligned} \tag{4.71}$$

The frequency response corresponding to this transfer function is shown in Figure 4.29. Two plots are shown, for different values of leakage inductance estimate error. Three controllers are also included. The first is the original controller which has no compensation mechanism for variations in back-emf. This is the plot with the smallest sensitivity peak, and hence the greatest immunity to noise in the current measurements. The broken line represents the response of the $\epsilon = 1$, or extrapolation controller. This has a large sensitivity peak, and high frequency noise present in the measurements will be amplified greatly by the controller.

The observer based controller varies in properties between the original controller and the extrapolation one. For $\epsilon \to 0$, the sensitivity approaches that of the original controller. For the useful range of ϵ, which is less than 0.1, the sensitivity is close to the original controller. The peak in the sensitivity of this controller is about 10dB. This would be considered high in controller design, but that is necessary in order to achieve the high-bandwidth control.

The figure of 10dB sensitivity means that the controller will generate noise in the

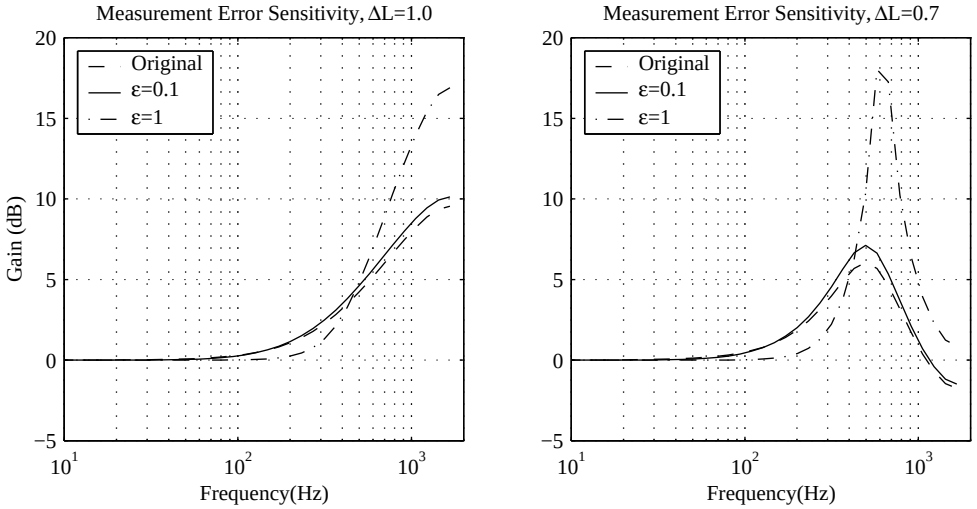


Figure 4.29: Noise sensitivity transfer function.

currents to a level of three times that present in the original measurements. Fortunately, relatively noise-free measurements are achievable and the 10dB noise sensitivity can be accommodated. The experimental part of this work has shown that a measurement signal-to-noise ratio well in excess of 40dB (100:1) is readily achievable. At this level, the 10dB of sensitivity may be accommodated, although it is accompanied with a loss of precision.

At up to 20dB of noise amplification, the extrapolation controller is difficult to accommodate. It would require very good current measurements for adequate control. In all forms of the controller, increases in the inductance estimate will increase the noise sensitivity. Increases in the controller gain, ρ, will also cause an increase in sensitivity.

To reduce the noise sensitivity, either ρ or ΔL may be reduced, although, reduction of ρ below 1 causes significant tracking phase errors. Overall, a reduction in ΔL achieves a greater reduction in sensitivity for the equivalent tracking phase error.

Inverter Dead-Time Tolerance

The presence of switching device dead time is a significant source of error in voltage-source inverters. The cause of the error is described in Section 5.4.2. This error may be considered as a disturbance to the controller, and the extent of its effect estimated.

The inverter dead-time causes an error in the average output voltage which is a function of the the sign of the phase currents. When the current in one phase of the inverter changes direction, the effect may be modelled as a step disturbance applied to the output voltage from the controller. In this case, the voltage across the leakage inductance becomes,

$$\underline{v}_l = \underline{v} + \underline{v}_{err} - \underline{e}_b. \tag{4.72}$$

Thus the effect of step on the current is equivalent to a step change in the back-emf. The back-emf is modelled as a disturbance term in Equation (4.69), so that may also be used for this purpose. Using this model, an example step response is shown in Figure 4.30. This

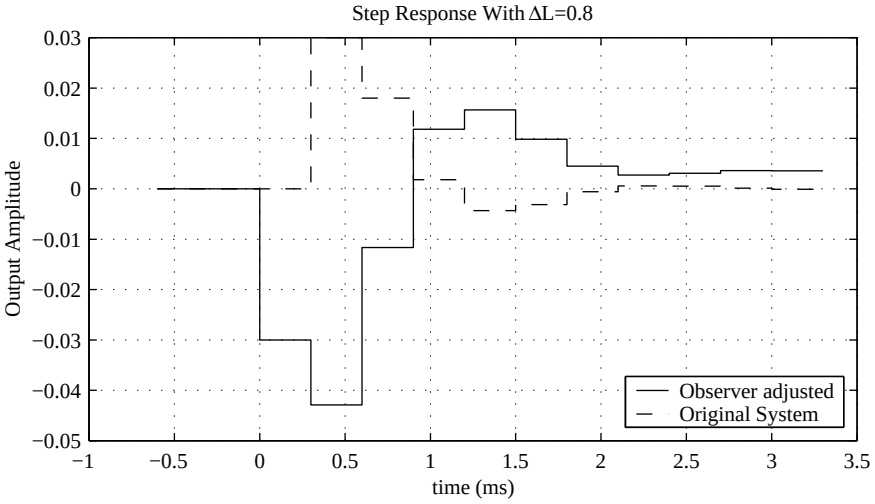


Figure 4.30: Response to a step change in back-emf.

figure shows the effect on both the original controller, and the observer based controller with $\epsilon = 0.1$. The modified controller is more susceptible to the step than the original controller, both in the size and the duration of the error.

As an example, the dead time might represent $2\mu s$ of a $300\mu s$ control interval. With a $600V$ DC link, the peak magnitude of $0.04S$ represents an error of $160mA$ for this system. This is certainly a significant deviation, but both controllers reduce the error greatly after two to three control cycles.

The observer design includes a response mode that is slowly decaying. This is due to the time-constant in the back-emf angular velocity filter. While the final settling time from this step change is very long, after 2ms the error is quite small. In the example, the error would be only 16mA at this point.

Simulation Results

The observer based controller was simulated under a number of conditions to verify the results expected from the analysis. Figure 4.31 shows the performance of the flux rotation angle estimate.

The dotted line shows the results of the raw angle estimate, while the solid line is the filtered version of it. The raw value is obviously very noisy, and unsuitable for direct usage. Note that in the case of the extrapolation controller, a value equivalent to this is used directly.

The broken line represents the angular rotation requested by the torque controller. Although there is some error between the torque controller and the filtered value, the

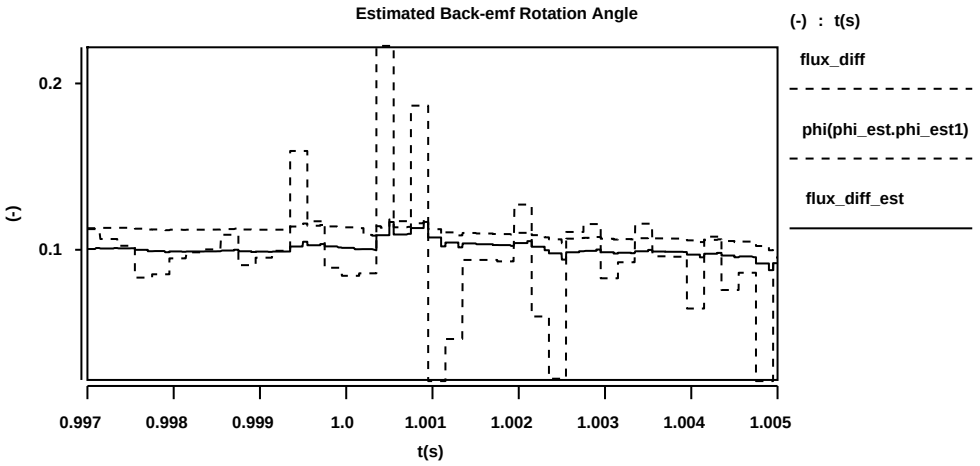


Figure 4.31: Observer based estimation of the electrical angular velocity.

result is accurate to within 10%. As this is only a correction term, this level of accuracy should be sufficient. The source of the error is likely to be a bias introduced in the axis selection process. The result may be obtained from either axis, but for numerical stability the calculation is chosen to provide the larger of the two back-emfs in the denominator. During the periods where the back-emfs in each axis have approximately the same magnitude, the measurement with a magnitude that is inflated by noise is likely to be chosen. At the expense of complexity, this error could be reduced, but it does not appear necessary for this application.

The adjusted back-emf estimate is shown in Figure 4.32. This simulation shows a sharp torque transient after accelerating the machine to rated speed. The aim was to show the fastest possible change in back-emf rotation speed. Both the direct and quadrature

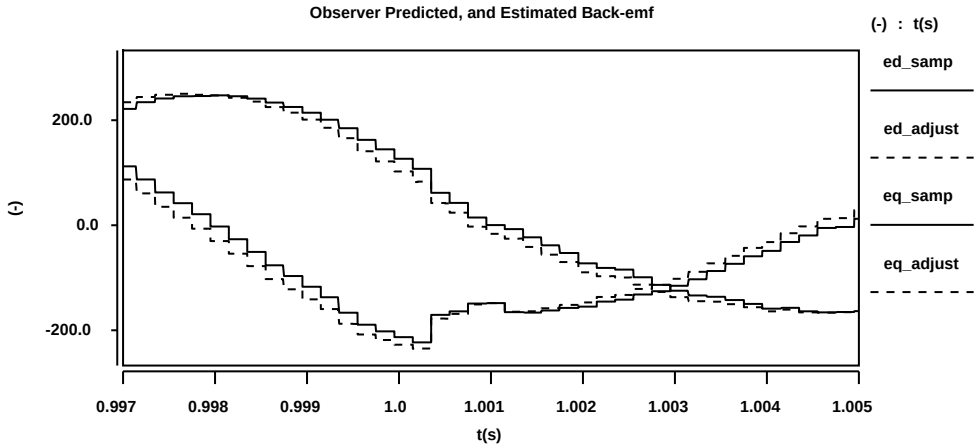


Figure 4.32: Predicted and estimated back-emf.

axis results are shown, with the quadrature axis lagging. The solid line indicates the basic back-emf estimation result. Due to causality, there is a delay in obtaining this

estimate. The dotted line shows the predicted value using the observer method. It can be seen that this generally provides a good one step-ahead prediction of the back-emf. At just after $t = 1s$, there is a glitch in the result, and this is due to the sharp transient in the current setpoint. This does not affect the performance of the machine, as the control is in saturation at that point anyhow. The error in the estimate at that point is due to a mismatch of model parameters. When a greater inductance error is introduced, the error at this point increases.

The actual current tracking performance of the controller is shown in Figure 4.33. This is for the same set of conditions as the two above simulation results. As in the

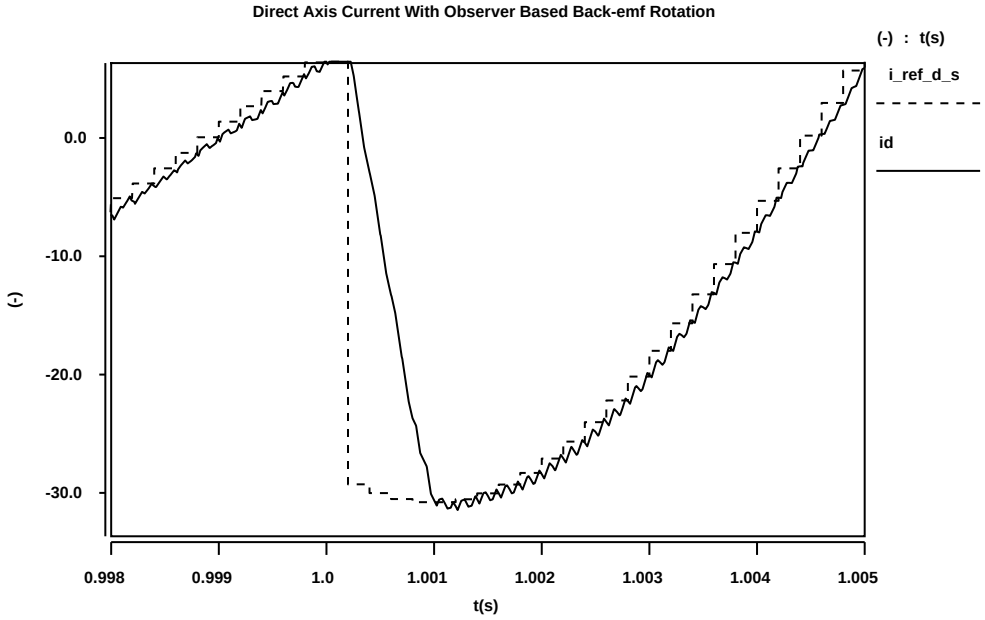


Figure 4.33: Current tracking performance with a torque transient.

case of previous simulation data, the dotted line indicates the endpoint current reference across the control interval. The real machine current should coincide with the endpoint of each reference segment, before the transition to the next reference. For most of the plot, the tracking ability of the controller is very good. During the transition itself, there is an output voltage saturation, limiting the rate of change of current.

The effect of a model parameter error is shown in Figure 4.34. In this case, there is an error in the estimated inductance, with $\Delta L = 0.7$. The main deviation in the results is the presence of overshoot after the transient. Although this is undesirable, in proportion to the size of the transient, the overshoot is quite small. After the transient, the tracking performance is still good, despite the parameter error.

Another section of the same plot is shown in Figure 4.35. This figure has been included to show the effect of inverter dead-time on the matching current. At time $t \approx 1.9135s$, a zero-crossing in current occurs in another phase. This causes a step change in the inverter output voltage error, and a subsequent error in the current. In this case, the error persists

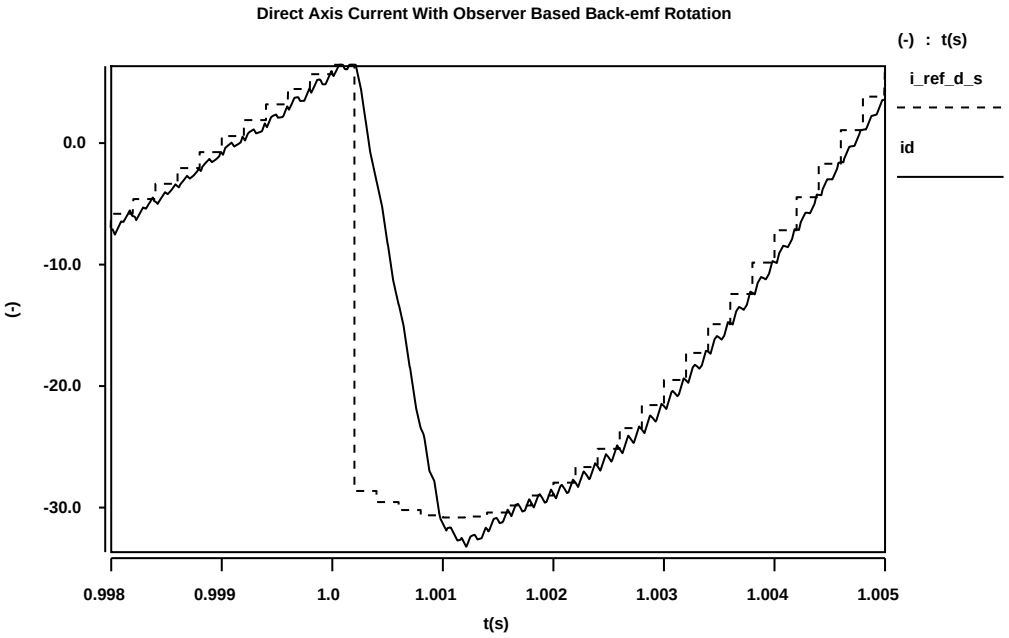


Figure 4.34: Current tracking performance with a torque transient and inductance estimation error.

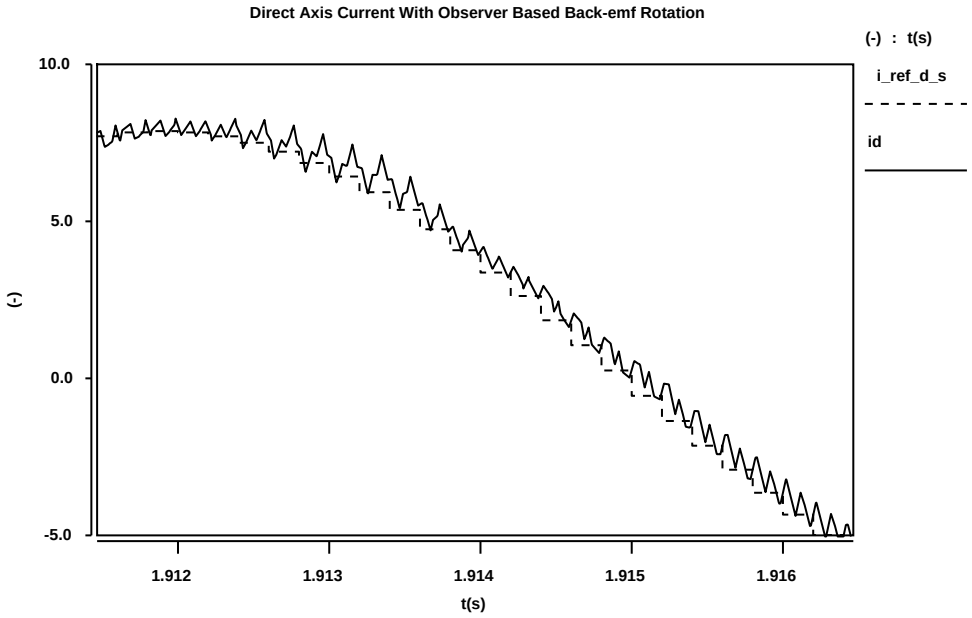


Figure 4.35: Current tracking performance with inverter dead time.

for approximately four control cycles, with a peak magnitude of about 400mA. This is consistent with the values expected from the analysis in Section 4.3.3.

4.4 Inclusion of Winding Resistance

A more complete model of the induction machine can be formed if the series winding resistance is included, as shown in Figure 4.36. In the earlier sections, the presence of

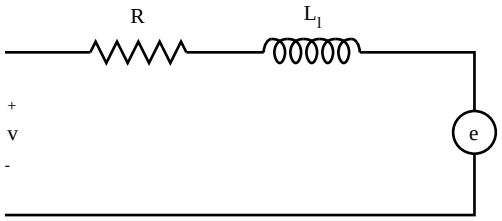


Figure 4.36: One phase of the d-q induction machine model, including the winding resistance.

this resistance was dismissed as having negligible effect. This section considers the extent of the errors introduced by neglecting to model the resistance.

Note that in the earlier sections, the effect of the resistance is not neglected simply on the basis that the value of resistance is small. There are two other important factors that combine to reduce the effect of modelling this parameter:

1. The voltage across the resistance is incorporated into the back-emf voltage parameter. For a constant current, the voltage across the resistance is a constant, and so may also be modelled as a voltage source. This means that model error only occurs as a result of the change in machine currents, and not based on the current itself.

2. Only the terminal voltages and currents are important for the purposes of control. It is the job of the outer torque controller to make calculations based on the values of internal parameters.

4.4.1 Under-modelling Errors

When resistance is included, the current is governed by the first order linear differential equation,

$$v - e_b - Ri = L_l \frac{di}{dt}. \tag{4.73}$$

This has the solution,

$$i(t) = \frac{v - e_b}{R} + \left(i(0) - \frac{v - e_b}{R} \right) e^{-\frac{R}{L_l} t}. \tag{4.74}$$

While this is a relatively simple model, it is harder to formulate a predictive controller for it than the basic two-term model. In addition, the resistance parameter must be known. It is therefore convenient to avoid the additional complication unless significant performance improvements can be made.

The initial transient response of this model is the same as that without the resistance. Figure 4.37 shows the departure over time of the two predicted responses. The initial behaviour may be considered by taking the 1st order Taylor expansion of the exponential in (4.74):

$$i(t) \approx \frac{v - e_b}{R} + \left(i(0) - \frac{v - e_b}{R} \right) \left(1 - \frac{R}{L_l} t \right) \tag{4.75}$$

$$= i(0) - \frac{R}{L_l} t \left(i(0) - \frac{v - e_b}{R} \right) \tag{4.76}$$

$$= i(0) + \left(\frac{V - e_b - i(0)R}{L_l} \right) t \tag{4.77}$$

$$= i(0) + \frac{V_L(0)}{L} t. \tag{4.78}$$

The response of the first-order approximation is identical to that of the simpler model, assuming the $i(0)R$ term forms part of the back-emf quantity.

The original model, shown in Figure 3.3, should include the voltage across the resistance within the back-emf component. As the back-emf cannot be measured directly, but is instead estimated from the measured voltages and currents, the estimator will include the iR component within the e estimate, ie,

$$e^{est} = e_b + i(0)R. \tag{4.79}$$

When the Taylor series expansion is continued beyond the first term, difference appears between the real system, and the simplified model. While the first derivatives at the point of prediction are the same, the inclusion of the resistance results in an exponential current, decaying to the value expected if the inductor were replaced with a short circuit. This is shown in Figure 4.37.

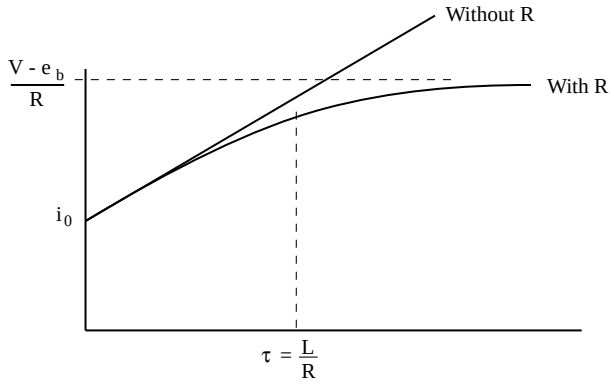


Figure 4.37: Comparison of response with and without winding resistance.

This comparison shows that the approximation will be good so long as the prediction horizon is significantly less than the the leakage time constant $\tau = \frac{L_l}{R}$. An example value

of τ for a typical induction machine is

$$\tau = \frac{10mH}{1.4\Omega} \tag{4.80}$$

$$= 7.1ms. \tag{4.81}$$

For control rates that are into the kilohertz range, this time constant is quite long. As a result, the omission of the winding resistance in the model could be expected to only have a small effect.

An approximation of the magnitude of the error may be obtained by considering the second-order term to the Taylor expansion of the model response. This is:

$$e^{-\frac{R}{L_l}t} = 1 - \frac{R}{L_l}t + \frac{1}{2}\left(\frac{R}{L_l}\right)^2 t^2 + O(t^3). \tag{4.82}$$

The difference in the resulting current between the two models is then approximated by:

$$i_e \approx \frac{1}{2}\left(\frac{e_b + i(0)R - v}{R}\right)\left(\frac{R}{L_l}\right)^2 t^2 \tag{4.83}$$

$$= -\frac{V_L(0)}{2L_l}\left(\frac{R}{L_l}\right)t^2. \tag{4.84}$$

This expression may be simplified if it is assumed that the prediction time is t_p and that there is a nominal change in current over that time of Δi_p. Using the relationship $V_L = L\frac{di}{dt}$ for the inductor, the error may be approximated:

$$i_e \approx -\frac{\Delta i_p}{2t_p}\left(\frac{R}{L_l}\right)t_p^2 \tag{4.85}$$

$$\frac{i_e}{\Delta i_p} \approx -\frac{t_p R}{2L_l}. \tag{4.86}$$

This represents the error in the final current as a ratio of the change in current over a cycle. When this value is small, the effect of including or excluding the resistance in the model is small. Again using some typical machine parameters, this value could be expected to be:

$$\left|\frac{i_e}{\Delta i_p}\right| \approx \frac{0.3ms \times 1.5}{2 \times 10mH} \tag{4.87}$$

$$= 0.02. \tag{4.88}$$

This represents a 2% error in tracking a step change in current. Compared to other errors present in the estimation, this is quite small. The sign of the error in (4.86) is also negative, indicating that the control is less aggressive in tracking than it should be. This indicates that this under-modelling does not pose a threat to the controller stability.

4.4.2 Compensation for Resistance Effects

While the effect of under-modelling the resistance is small, it can be reduced via a compensation scheme. This involves using knowledge about the shape of the exponential response, and matching it to the linear segment at the required point.

The controller operates at a fixed control rate, and nominally controls the current which is to appear at the end of the given control interval. The reference may apply to a current that is to appear before the end of the interval. This is controlled by the ρ parameter defined in Section 3.2.4. In this case, matching should instead occur at that point.

Figure 4.38 shows how the linear trajectory used in the control calculation is matched to the exponential response of the system with resistance.

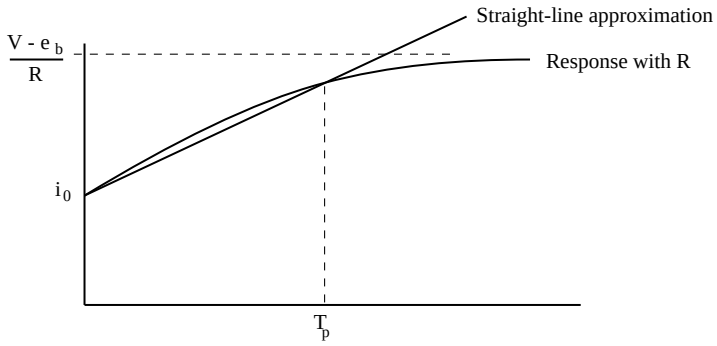


Figure 4.38: Endpoint current matching.

Even if the matching is successful, there is still an error in the average current across the cycle. However, from the preceding analysis, this is very small and will be assumed to not be significant.

In order to match the simple controller model without the winding resistance, the machine current during one control interval should have the form

$$i(t) = i(0) + \frac{v - \hat{e}}{\hat{L}} t, \tag{4.89}$$

where $\hat{e}$, and $\hat{L}$ represent parameters with the role of the machine back-emf and leakage inductance respectively. Note that these do not have to correspond directly to the machine parameters, but rather can be any constant which may be calculated from available information.

Equation (4.74), may be manipulated into this form by considering the current $i(t_p)$ at time $t = t_p$

$$i(t_p) = \frac{v - e_b}{R} + \left(i(0) - \frac{v - e_b}{R} \right) e^{-\frac{R}{L_l} t_p} \tag{4.90}$$

$$i(t_p) = \frac{v - e_b}{R} \left(1 - e^{-\frac{R}{L_l} t_p} \right) + i(0) e^{-\frac{R}{L_l} t_p} \tag{4.91}$$

$$= i(0) + \frac{v - e_b - Ri(0)}{R} \left(1 - e^{-\frac{R}{L_l} t_p} \right) \tag{4.92}$$

$$= i(0) + \frac{v - e_b - Ri(0)}{\frac{Rt_p}{1 - e^{-\frac{R}{L_l} t_p}}} t_p. \tag{4.93}$$

This result may be matched to (4.89), to obtain adjusted controller parameters,

$$\hat{L} = \frac{Rt_p}{1 - e^{-\frac{R}{L_l} t_p}} \tag{4.94}$$

$$\hat{e} = e_b + Ri(0) \tag{4.95}$$

The back-emf term is the same as that found earlier, where the voltage across the resistance is included in the lumped voltage source. The value for the adjusted inductance represents a deviation from the expected value. The modification represents the adjustment required to model the resistance. This means that in terms of the current at the end of the control interval, the model without resistance will be equivalent to the model with resistance, provided the suitable adjustment is made to the leakage inductance value used.

By taking the Taylor series expansion of the exponential in $\hat{L}$ and approximating, a more intuitive form may be found,

$$\hat{L} = \frac{Rt_p}{1 - \left(1 - \frac{R}{L_l} t_p + \frac{1}{2} \left(\frac{R}{L_l} t_p \right)^2 - \frac{1}{6} \left(\frac{R}{L_l} t_p \right)^3 + \ldots \right)} \tag{4.96}$$

$$\approx \frac{L_l}{1 - \frac{Rt_p}{2L_l}} \tag{4.97}$$

$$\approx L_l \left(1 + \frac{Rt_p}{2L_l} \right) \tag{4.98}$$

$$= L_l + \frac{1}{2} Rt_p \tag{4.99}$$

Equation (4.98) provides a multiplicative adjustment to the true machine leakage inductance. The quantity here is the same as in (4.86), showing that the change required is small. Typically, an increase in L_l of 2% would be necessary for matching.

This result is useful because it shows that the effects of the under-modelling can be represented in terms of an error in a parameter. The analysis of the effect of changes in R on controller performance and stability is then simply an extension to the analysis of the inductance.

4.5 Conclusions

From the controller analysis, the controller as described previously offers good stability and tracking performance with a suitable choice of controller parameters. In particular, the ρ controller parameter is best set to 1 to obtain the best stability attributes. Since its original formulation[3], it has been known that the average current was less stable, but this analysis shows the extent of the difference.

The main limitations of the controller were found to be the effect of rotational changes in the back-emf, and a high sensitivity to high frequency measurement noise. The sensitivity is a necessary consequence of high-bandwidth control, and the levels are acceptable for the type of signal-to noise ratios achievable in measurement.

In order to overcome the back-emf rotation errors, both the feed-forward, and observer methods offer acceptable solutions. The feed-forward method requires little additional calculation, but does mean the current controller depends on knowledge of the electrical rotation speed. At the expense of some additional complexity, the observer method removes this requirement.

Chapter 5

Controller Implementation

5.1 Introduction

A current controller design was described in Chapter 3, and then analysed in Chapter 4. This chapter describes the additional details of the controller implementation using a Digital Signal Processor (DSP).

Although one aim of this project is to implement a current controller in digital hardware, this is not the most convenient algorithm development platform. For this reason, controller hardware incorporating a DSP device was developed, allowing the controller to be specified in software. This approach combines favorable attributes of both the hardware and software implementations. Development in hardware is typically more difficult than on an equivalent software platform. This is because debugging and testing facilities are more easily implemented in software. Attributes such as controller gain may be easily modified, and measurement data recorded with the aid of a microprocessor. The result is a faster development path for control algorithms. Once the algorithm is sufficiently tested, it may be implemented directly in hardware.

5.2 System Architecture

The software implementation has been made on a dual-processor Texas Instruments TMS320C31 system. As this is a development platform only, no particular attention is applied to exploiting the specific features of the processor hardware. Instead, it is made to resemble the final hardware design in architecture. While this approach results in a potentially less efficient software architecture, it may be directly ported to the hardware implementation environment. For this reason, even though a floating point processor is available, scaling factors are included to restrict the values to appropriate fixed-point values. Advantage is taken of the non-integer values available, and is used to maintain greater precision in the calculations. However, the code developed in this fashion is easily modified to simulate integer-only calculation.

The basic structure of the test system and controller is shown in Figure 5.1. Apart

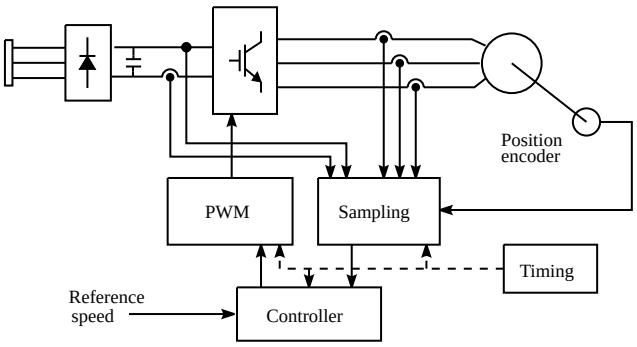


Figure 5.1: Controller desgin.

from the power electronics hardware, the controller comprises four primary sub-systems. In the previous chapters, the focus has been on the algorithmic details of the controller block. However, a functioning digital machine controller also requires the other elements. It is in these additional elements that the additional complexity of a digital controller is found. Fortunately, with modern digital devices, all of this complexity may be integrated into a single integrated circuit. The design described in this chapter uses two small EPLD (Erasable-Programmable Logic Device) devices in addition to the DSP chip to implement the control structure. In the following chapter, a harware solution will be described, where the controller and all the support logic is integrated into a single EPLD device. The larger EPLD used in the second design is capable of containing all of the logic from the software design, together with the controller itself.

The basic role of each subsystem is:

- The primary role of the sampling circuitry is to perform analogue to digital conversion of the machine currents. In addition to this basic functionality, provision is made for an error-controlled serial data link for each channel, minimising the amount of analogue signal cabling.

- The PWM module implements a space-vector modulation scheme. This includes a programmable device lock-out time, and an optional dead-time compensation scheme.

- The controller itself is implemented in software. This includes the current, torque and speed control loops.

- The entire controller system must be synchronised to achieve proper operation. The timing logic is implemented as part of the sampling system. This ensures that current samples are available at the correct locations within the switching period.

5.3 Data Acquisition

The data acquisition system is required principally to measure the phase currents of the machine. These measurements are used both in the control calculation and as an additional layer of overcurrent protection. While many devices have inbuilt overcurrent protection, there are reasons for supplying the redundancy of an external measurement. It offers more flexibility in the choice of the limit setpoint and better diagnostic information in the event of a trip. The measurement latency precludes the protection from being effective against output short-circuits, but it is quite adequate to protect against errors in the control behaviour.

A number of high speed synchronous serial channel receivers are used to acquire data from the serial A/D converters located remotely on the Hall Effect transducers. The bit rate for each channel is 5Mbs, and each data packet is 20 bits. This allows 250,000 samples per second, giving an effective bandwidth of approximately 100kHz. This matches the bandwidth of the Hall Effect transducers now in common use. The use of serial transmission has the advantage that far fewer I/O pins are required on the EPLD. Furthermore it allows simple low cost isolation of the cables from the Hall Effect devices (if required),and eliminate potentially long cables carrying analog signals.

The DC-link voltage is also measured to allow for feed-forward compensation of this quantity. This measurement also allows the controller to provide some control over the voltage. This is of relevance under regeneration conditions, where it may be desirable to dump energy. Knowledge of the link voltage may also be used to ride-through temporary supply interruptions.

5.3.1 Connection Scheme

In the test system, the controller is physically remote from the power electronics and measurement locations. This is done to assist physical access and improve the environment for the controller electronics. However, the problem of accessing current measurements is created. In order to reduce the amount of noise coupled into the signal, the signal transmission is better done digitally. As a result a serial data comminications path is needed between the analogue to digital converters and the controller processor. This implementation uses coaxial cable with opto-couplers for isolation, although optical fibre could also be used for the signal transmission.

A number of alternatives are available for synchronisation of the data packets. As shown in Figure 5.2, the three basic categories considered for this application are;

a. Asynchronous.

b. Synchronous with transmitter supplied clock.

c. Synchronous with receiver supplied clock.

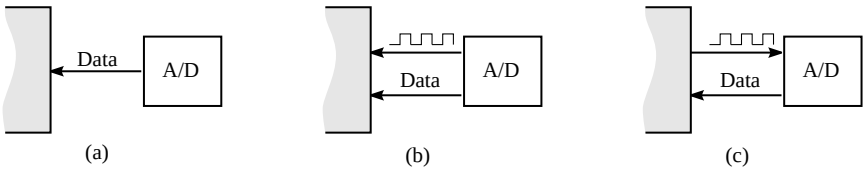


Figure 5.2: Data synchronisation.

With an asynchronous link, the clocking information is recovered at the receiver, by synchronising a local oscillator to the remote data stream. This is a common method for low-frequency data streams, but at a bit rate of 5Mbps, the design of these systems can become difficult. An alternative approach is to not only transmit the data stream, but also transmit the clock information associated with it. While this doubles the number of connections to be made, it simplifies the receiver as the clocking information is available. The third option listed has the communications clock supplied by the receiver. This simplifies the synchronisation with the receiver state machine as the whole system uses the global clock, but also introduces clock skew problems. Due to delays, particularly in the opto-couplers, the clock is skewed on reaching the transmitter. The data is further delayed on the return path, resulting in potential timing problems.

The design of the current controller relies on current samples taken at particular points in the switching cycle. As a result, it is neccessary for the controller to signal the start of each conversion. For cases (1) and (2) above, this would be done by suppling an additional connection to carry the start signal. The precision of the start timing would be limited by the local oscillator used at the transmitter. Case (3) is more suited to this form of operation, as the start signal may be encoded into the existing clock signal. It has the additional advantage that the start time may be precisely controlled with respect to the system clock.

Due to the synchronisation issues involved, and hardware available, a synchronous design was chosen, with a receiver supplied clock (case 3 above). This data acquisition structure is shown in Figure 5.3. The receiver supplies a common clock for all of the A/D converters. The converters simultaneously transmit the converted values along the data lines.

5.3.2 Communications Protocol

As shown in Figure 5.3, all of the A/D units are driven with a common clock signal. This signal also carries the start conversion timing information. The start of conversion is signalled by the supression of two cycles of the transmitted clock. The form of the resulting clock signal is shown in Figure 5.4. The receiver detects the two supressed periods, and uses this to synchronise the start of conversion, and subsequent data bits.

The synchronising circuit requires an oscillator local to the encoder. A state machine operating off the higher frequency local oscillator detects any long inactive state on the

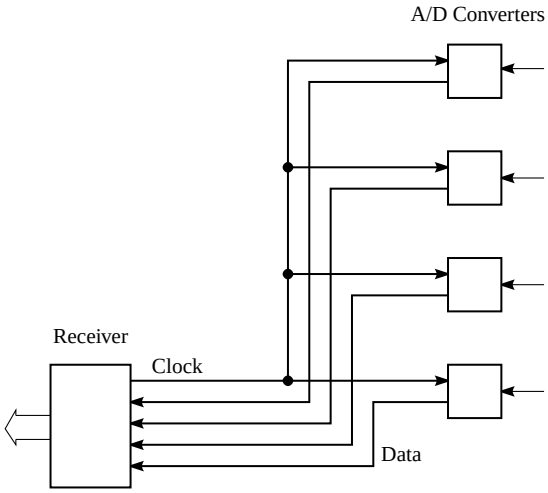


Figure 5.3: The data acquisition structure.

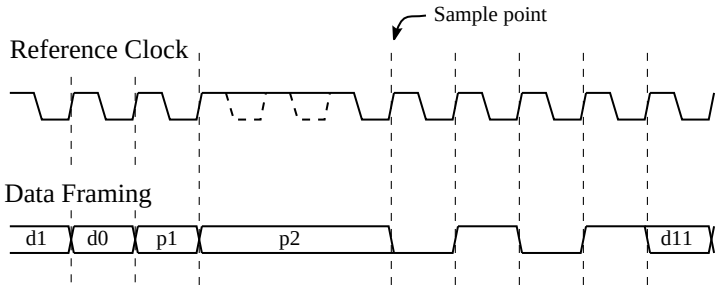


Figure 5.4: A/D converter signal format.

incoming communications clock signal.

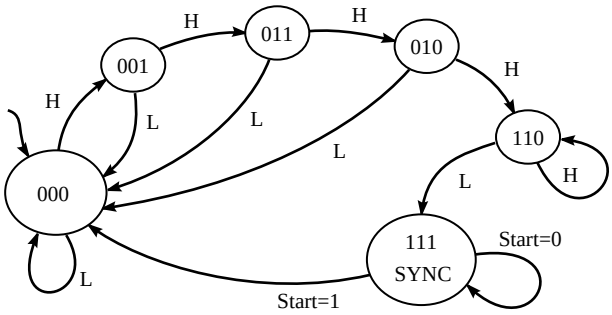


Figure 5.5: Synchronising state machine.

Figure 5.5 shows the detail of the state machine used to syncronise the transmitter to the receiver. It starts in the 000 state, and remains there while the incoming communications clock is in the low state. When the signal is high for four successive clock periods of the high frequency clock, the 110 state is reached. If it is high for a shorter time, the machine returns to the 000 state. The 110 state indicates detection of a suppressed

clock. The machine remains in this state until the clock again returns low. Once this occurs, a synchronising output is set to reset the communications state machine. Once the communcations state machine responds with a conversion start signal, the synchronising state machine returns to the reset state.

Two primary methods are used to check the link integrity. These are based on the start/stop bits in the frame, and the parity bits. The start bits are necessary due to the delay from start of conversion until the data bits are available, but they are also used to detect whether the link is operational. The receiver checks to ensure that the correct start pattern is received before accepting the data stream. Bit errors in the received data stream are also detected via the use of the parity bits.

The reliability of the link may be improved by allowing it to accomodate a small number of communication bit errors without adversely affecting the operation of the system. This may be done by incorporating error correction codes into the serial data packet. An alternate approach is to only detect communication errors, and in the case of an error, to replace the corrupted value with an estimate of the true value. In this implementation a good estimate is readily available, due to the high sampling rate. In the event that a data point is missing it may simply be replaced with the previous measurement. While this does introduce an error, it is small due to the short sample period.

An alternate estimate utilises the values measured in the other two phases. Due to the zero neutral voltage, $i_a + i_b + i_c = 0$, so only two phase measurements are required. The third phase current measurement is useful for overcurrent protection, but also offers some measurement redundancy. In the event that one of the current measurements is unavailable, it may be supplied from the other two reliable measurements.

Due to the use of a modular design in the communicaitions link, the relationship between the phase currents was not utilised. This approach would not be suitable for the DC voltage or current measurements. Instead substitution of the previous value is used for all of the measurements. Due to the low probability of bit error, the introduction of error-correction codes in the data stream is not justified. This would incur a penalty not only on the data transfer rate, but also on the transmitter and receiver complexity.

This error tolerance scheme is only suitable for isolated data bit errors. In the case of sustained corruption, or a broken data link, the drive must be tripped. The interface circuit generates a trip signal after a specified number of consecutive erroneous values. Normally the trip would be programmed to occur after receiving two errors among three consecutive received values. Each of the receivers operates independently, maintaining separate counts.

Receiver Datapath

The overall computation datapath of the receiver module is shown in Figure 5.6. The parity checking and trip limit comparison are both done sequentially, as each data bit

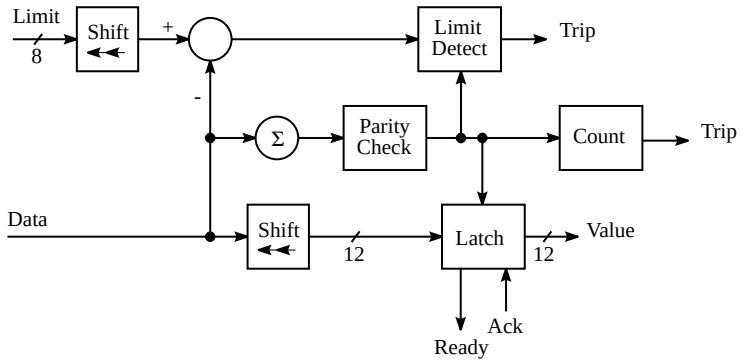


Figure 5.6: Receiver datapath structure.

arrives through the serial data channel. This was preferred over the parallel techniques in terms of both time and space resources in the logic device. Only single bit adders are required instead of full 8-bit or 12-bit adders, and the propagation delay is spread through the receiver cycle, instead of being concentrated at the receipt of the final bit. As the current limits are the same for each of the three phases, the logic for the trip-limit shift register (or multiplexer) may be shared.

Another advantage of comparing the trip limits sequentially is that the average trip latency may be reduced. This is because an overcurrent condition may be detected before all of the bits are received. However, in this implementation, trip signalling is held back until the parity is confirmed. If there is also a parity error, the overcurrent value is assumed to be due to link data corruption. Parity errors do not immediately trip the drive, and instead a counter is used to trigger a shutdown only in the case of consecutive errors.

Samples are acquired and checked for value at a high frequency, and only a subset of these are used by the controller. The timing module specifies which samples are to be loaded into the output latch for later acquisition by the controller.

5.3.3 Sample Timing and Acquisition

One requirement of the proposed controller design is that the timing of the sample acquisition is important. In particular, samples are required at the start and centre of the PWM switching pattern. This synchronisation is implemented by using common timing components.

The total cycle time, from one sample to the next, is twenty cycles of the A/D serial channel clock. While it is not necessary to do so, it is convenient to have a constant inter-sample period for all of the measurements. A futher constraint is that the controller requires samples of the currents taken at the start, one quarter, and one half of the way through the switching cycle. The resulting pattern is shown in Figure 5.7.

The use of this pattern constrains the switching period to a multiple of $80 \times T_{SC}$,

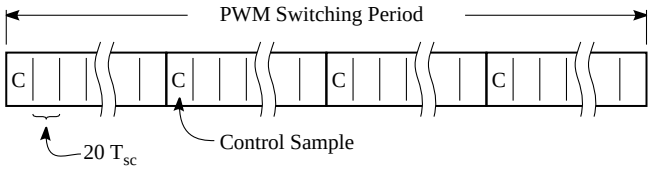


Figure 5.7: The location of the samples needed for the controller.

where T_{SC} is the period of the serial data transfer clock. For a given serial data rate of approximately 5Mbps, the switching frequency may be adjusted in increments of $16\mu s$, which would normally allow more than sufficient flexibility.

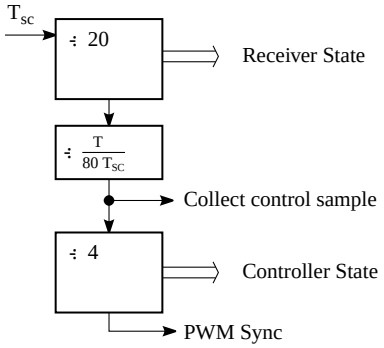


Figure 5.8: The overall timing structure for the controller.

The synchronisation requirements for the design lead to the structure shown in Figure 5.8. This consists of three cascaded counter structures. The highest frequency is at a period of T_{sc}, which is the serial communications clock. The frequency of this is either equal to, or divided down from the hardware system oscillator. This clock is supplied to the receiver state machine, which performs an implicit divide by 20 operation corresponding to each sample received. A software programmable divider is then used to count the number of samples to be collected in each quarter of a control cycle. The quarter cycle is used because the algorithm uses samples at the start, $T/4$ and $T/2$ points in the control cycle. The final division by 4 provides synchronisation information to the PWM subsystem to ensure that the control cycle is synchronised with the PWM output cycle.

5.4 PWM Generation

The PWM Generation module is designed to reduce the load on the CPU arising from switching pattern generation. Before the start of each PWM cycle, the CPU supplies switching times and sector information to the PWM module. The PWM subsystem then loads the appropriate timers and switching patterns. Programmable dead time, and a dead-time compensation strategy are implemented in the module.

The PWM generation module was designed to be used with both the software and hardware versions of the controller. The extent of the functionality was chosen based on the requirements of the hardware version. The controller module calculates the output sector and switching times, while the pattern itself is synthesised in the PWM module. The switching time calculation could be performed in this hardware, but it would require additional logic. However, the calculation time required in the controller module is quite small.

5.4.1 Switching Generator Structure

The PWM module has been implemented in an 8k series Altera EPLD for operation with the DSP based current controller. The same design has been integrated into the single 10k series Altera EPLD for the hardware controller.

The basic structure of the PWM module is shown in Figure 5.9. The switching time

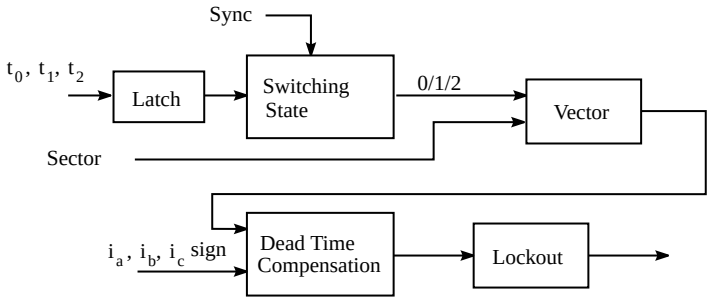


Figure 5.9: Structure of the PWM module.

section generates a state signal that describes the current state in the firing sequence. This is implemented using latches for each time period, and a down-counter to measure the duration of the periods. A state-machine provides the synchronisation and counter loading. This state machine is made more complicated by the allowance of zero-duration periods, which occur when fewer than three vectors are needed to synthesize the desired average voltage. The zero-vector is a special case of this, where no switching occurs at all.

In order to simplify the logic, both in the controller and in the PWM module, a specific choice of sector nomenclature has been devised. Instead of using the numbers 1 to 6, the sectors are represtented by a 3-bit binary string. The choice of sectors are shown in Figure 5.10.

The sector code may easily be determined simply by comparisons between the duty cycles and zero. This allocation does mean the two of the sectors are further divided up into two halves. This is not a problem, as the PWM hardware treats the two halves identically, and either may be used to specify the sector.

With the switching state determined, the output voltage states are simply a combinational function of the state and the sector. The latter is provided by the current

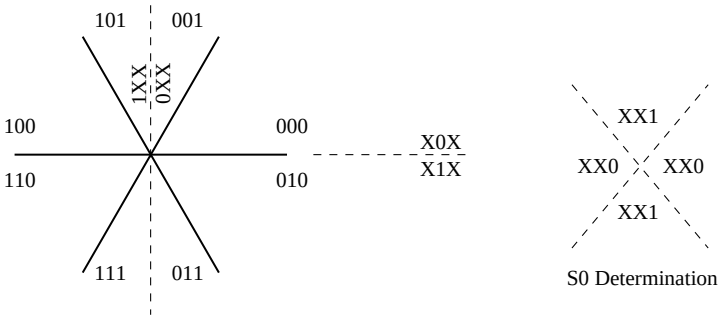


Figure 5.10: The PWM sector allocation.

controller module. The resulting vector consists of three signals, one for each phase of the inverter. A logic '1' value indicates that the corresponding leg should be switched high, and a '0' indicates it should be switched low. The remainder of the circuit relates to issues regarding the turn-on and turn-off delays of the switching devices.

5.4.2 Dead Time Issues

The issue of inverter dead-time was described in Section 2.3.3. The effect of the dead time is that delays occur on the output switching pattern. As the length of the delay varies with the direction of the current, an error occurs in the average voltage, often resulting in a distortion of the machine currents. It was found in Section 4.3.3 that this does have an effect on the proposed current controller, although the feedback control minimises the impact.

Observed Effect

The need for a factor of safety stipulates that, during an output transition of a bridge inverter, there be a significant delay from when one device switches off to when the other switches on. This safety margin worsens the output distortion because it increases the time when the output is not controlled. During the period when it is not controlled by the output devices, the output voltage is governed by the free-wheeling diodes. Depending on the direction of the current, one of these diodes will conduct. Due to the inductive nature of the load, the current will not change greatly in value during the switching period.

Based on the machine current, there are two basic cases to consider. These are for each direction of the current, and are shown in Figure 5.11. When the current is flowing into the inverter, the top diode conducts, forcing the voltage to the positive potential of the DC link. This results in the high side of the output waveform being lengthened. In the converse case, the lower diode conducts, lengthening the negative side.

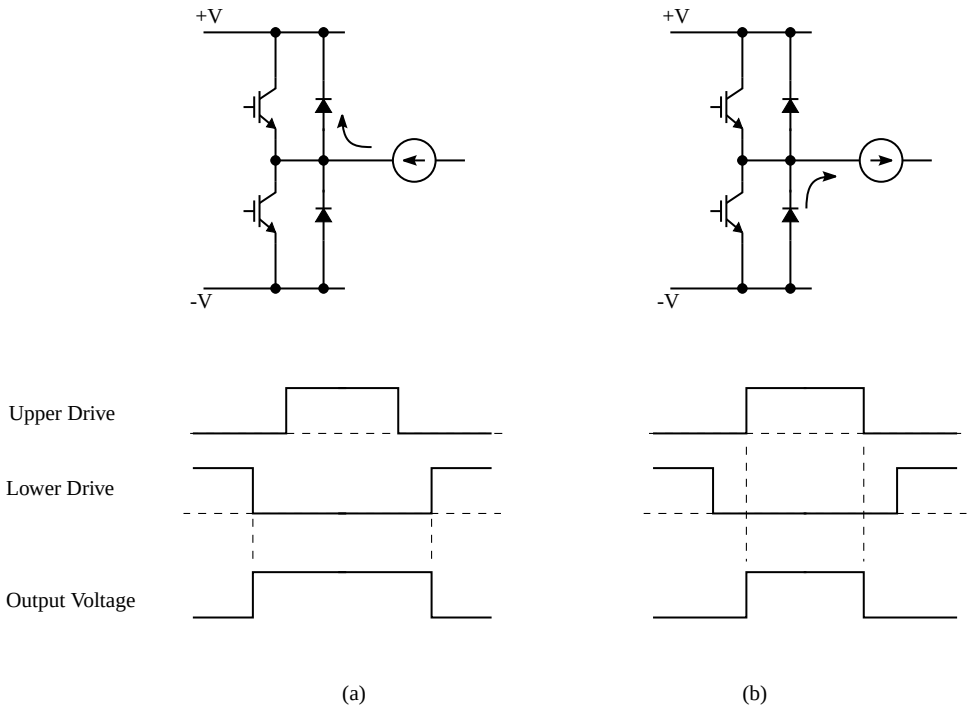


Figure 5.11: Voltage during an output transition.

Compensation Scheme

When the direction of the current is known, compensation for this behaviour is possible. The PWM circut may be modified to alter the timing based on the direction of the current at the time of the switch. The modification may be performed by inserting a delay in the switching time, based on the direction of current in that phase. Example voltage patterns are shown in Figure 5.12. The accompanying switching strategy is shown in Table 5.1.

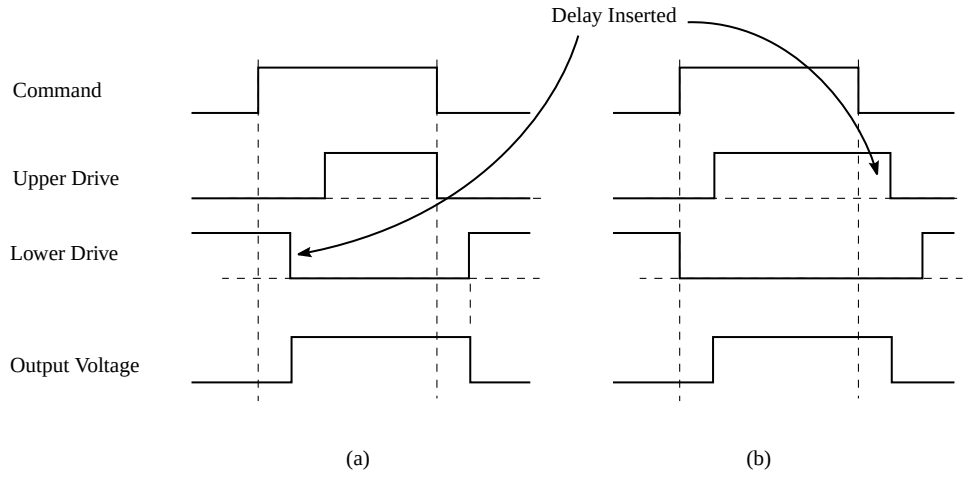


Figure 5.12: Dead-time compensation through delay insertion.

Transition	$i < 0$	$i > 0$
Low-to-High	Delay	No Delay
High-to-Low	No Delay	Delay

Table 5.1: Compensation switching strategy

Although the adjustment scheme is here described as a delay, the alternate transitions can instead be considered to be advanced. It is simply necessary that the current measuements be appropriately synchronised. In this particular implementation, the delay concept is used, to allow it to be easily be switched in and out of the design.

Note that while this scheme works well for ideal switches, as described, the switches are not ideal, and indeed that is the reason that the dead time is a problem at all. The practical considerations are that the device delays are unknown and variable. This means that the amount of compensation needed also varies. As a result, the main attribute being compensated is the factor of safety that has been inserted. As this is likely to be longer than a typical switching delay, there is merit in doing this. Much work has been done in estimating the true switching delays[13], but this was deemed too complex for this design.

The other problem with this compensation scheme is that it is difficult to model the voltage if the current is near-zero at the time of the switch. Indeed, the current may change sign between the time of measurement, and when the device actually switches. In this situation, the compensation is in the wrong direction, and there will be greater distortion than with no compensation at all. There is also a danger that the compensation will cause the current to again change sign, lengthening the duration of the problem. The nature of this problem is illustrated in Figure 5.13.

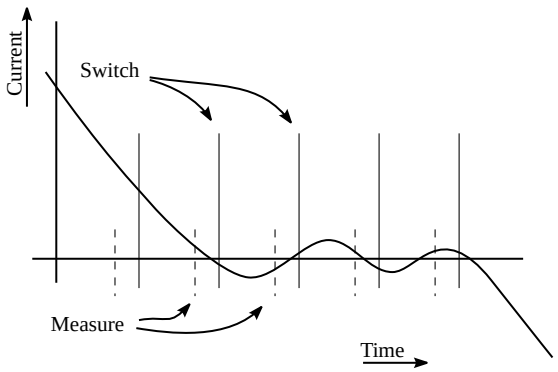


Figure 5.13: Zero-crossing problems exacerbated by dead-time compensation.

As a result of this phenomena, the compensated waveforms are likely to have "flat spots" near the zero crossings. This has been observed in trials of this system, and in the published examples[13].

Implementation Details

In order to ensure the design integrity of the final lock-out logic, that part of the circuit is kept as simple as possible. Although eventaually compiled into the same device, the dead-time compensation forms a separate logic component. This structure is shown in Figure 5.14.

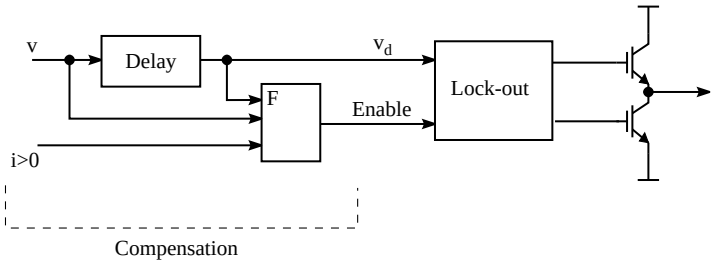


Figure 5.14: Dead-time circuit implementation.

The function block, F, may be combinatorial and is used to disable the active device in the cases where the parallel diode could instead conduct the current. When this occurs the direction of the current through the load will maintain the voltage during the period of the delay. The output devices are only disabled during the delay period, and even then only in the cases from Table 5.1 where the output pattern should should be delayed. A suitable enable function is:

$$enable = (v \oplus \bar{v}_d) + (i_{pos} \oplus v_d). \tag{5.1}$$

The final block forms the basic lock-out circuit. Complementary signals are first created. The output of each of these drive signals is forced inactive whenever the other drive signal has been active at all during the past t_l seconds, where t_l is the lock-out time.

5.5 Controller Design

5.5.1 The DSPs

The development system utilises two Texas Instruments TMS320C31 processors. These are a general-purpose DSP device, offering floating point numeric computation. Although faster versions are now available, 33Mhz parts were used in this design. The structure of the DSP system is shown in Figure 5.15.

The dual processors were not required for the design of the controller, and instead a single processor of this speed would be sufficient to compute the algorithm. The second processor is principally used as a logging facility and to run a monitor for external communications. A small amount of dual-port RAM is used for communication between the two processors.

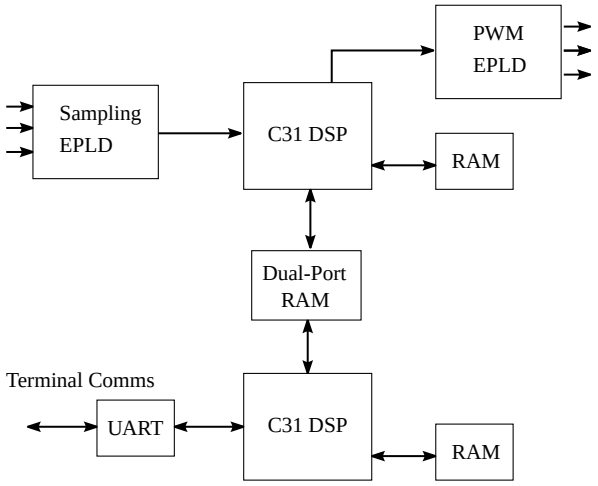


Figure 5.15: The Dual-processor arrangement.

The Monitor

The second processor was used primarily for logging and control facilities. It is useful for a number of roles:

- Enabling and disabling of the drive.
- Provide a set-point for the outer loop controller (e.g. speed).
- Choice of controller type (Speed/Position/Torque)
- Setting of parameters, such as the inductance.
- Obtaining values of controller variables.
- Provide a facility to log variables over time.

The DSP development tools offer some of these features, but they are too intrusive to use in a real-time environment. Using the second processor, all of this functionality can be achieved while the control algorithm is being executed.

The monitor communicates with a terminal emulator on a PC via an RS232 serial link. The user is presented with a simple command-line interface, which may be used to access variables and issue commands to the controller processor. In order to avoid continual re-loading, the monitor program is programmed into an EPROM. In contrast, a JTAG style interface is used to program the controller DSP in a the development environment. For this reason, the monitor is designed to static, yet flexible in operation. To aid in this, the functionality is very thin, with the command set being loaded at run-time from the other processor.

The structure of the resulting monitor is shown in Figure 5.16. The left-hand side represents the functions performed on the monitor processor. This involves parsing the input lines and tokenising the strings. There are only three valid form of input:

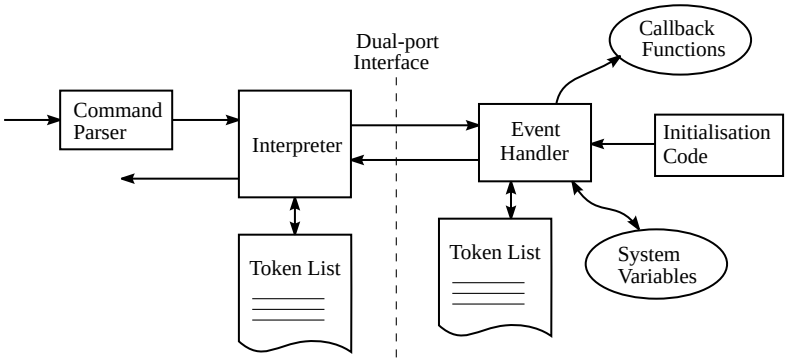


Figure 5.16: Structure of the monitor software.

- `<Command>`
- `<Variable>`
- `<Variable>=<Value>`

Provision was also added for commands to take parameters, but this was not required for the controller. The dual-port RAM between the two processors is used to hold two circular-buffers for bi-directional communications. A packet protocol is used to exchange messages.

Each time the software on the controller processor is started, the initialisation code configures the second processor, and builds matching token lists on each side of the interface. Each command and variable is given a numeric identifier, which is used to specify the target across the interface. For example, when the value of a variable is needed, the controller processor need only dereference a pointer stored in its token list, and pass the binary value back through the dual-port interface.

This dual-processor client/server monitor interface is mostly a by-product of the development system available. An equivalent monitor could readily be implemented on a single-processor if the controller is executed via interrupts.

5.5.2 EPLD Interface

The synchronous logic within the EPLDs must be interfaced to the asynchronous microprocessor bus. This is done through a general purpose interface block within the EPLD. The block provides eight inputs and outputs, each of 12 bits width. The structure of the logic is shown in Figure 5.17.

The write procedure commences with the processor writing the value into the holding register. Following this, a command word is written which specifies a write operation, and the destination latch for the value in the holding register. The command signal is synchronised to the local clock to ensure that the outputs only change synchronously

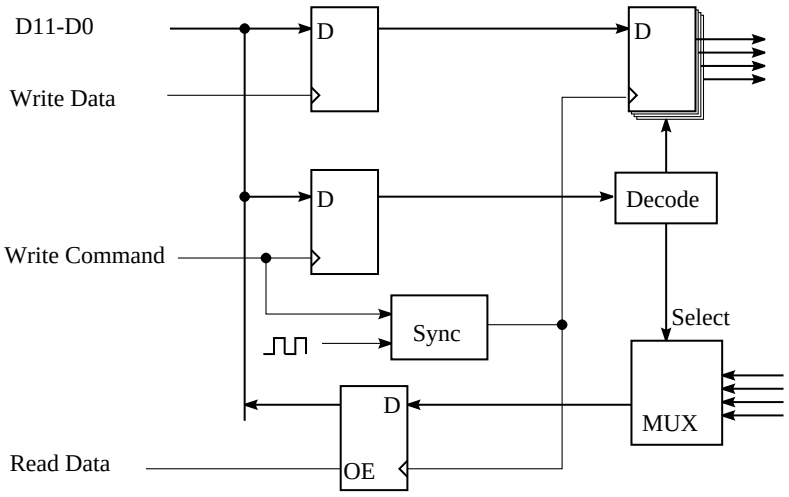


Figure 5.17: The processor interface.

with the clock signal. This is important, as it ensures that data corruption does not occur from simultaneous reading and writing to the latch.

The read procedure first involves writing a command word which specifies which value is to be read. This sets the input multiplexer to the correct value, and causes the reading hold register to be synchronously loaded with that value. Later, in a second bus access, the processor may read the value from the holding register asynchronously.

5.6 The Controller Software

The current controller itself is implemented within the single file `cc1.c`, which is included in the Appendix. This section describes the basic operation of the software controller algorithm.

5.6.1 Controller Variables

A number of variables are used during the calculation process, and the units are chosen to suit the fixed-point scaling. The most important of these are described here.

- T. This is the duration of the PWM cycle. The units of time are equal to the clock period, so this value is equal to the ratio between the system clock and the control rate. The nominal value is 1024, but it may be varied.
- $AlphaD$. The direct axis duty cycle. This is $\frac{T}{2}\alpha_d$. Useful values are in the range from $-\frac{T}{2}$ to $\frac{T}{2}$.
- $AlphaQ$. The quadrature axis duty cycle, $\frac{T}{2\sqrt{3}}\alpha_d$.
- IDx. These are direct axis currents. The forms of this are IDR for the reference, $ID0$ for the $t = 0$ measurement, $ID4$ for the $t = 0.25$ measurement and $ID2$ for

the $t = 0.5$ measurement. The specific value is three times the calibration of the A/D converter. The factor of three comes from the three to two phase conversion.

- IQx. The quadrature axis currents. In this case, the scaling is $\sqrt{3}$ instead of 3.
- *Lest*. The inductance estimate is actually an estimate of $\frac{L}{3}$, using the fixed point units. For the system implemented, [i]=10mA, [v]=0.5V, and [t]=0.26μs, so [L/3] = 39μH. This means that a real inductance of 10mH will be represented as 256.

5.6.2 The Main Loop

The controller does not use interrupts, but rather a single main loop, with polled waits for measurements. The initial entry point is the `main()` function, at the end of the code. The first section of code configures the firmware and inverter for the correct mode of operation. Following this, the second processor is initialised, with the debugging and logging variables sent across the interface. The remainder of the `main()` function deals with the logic to start and stop the controller.

The basic structure of the loop calculations is shown in Figure 5.18. The division

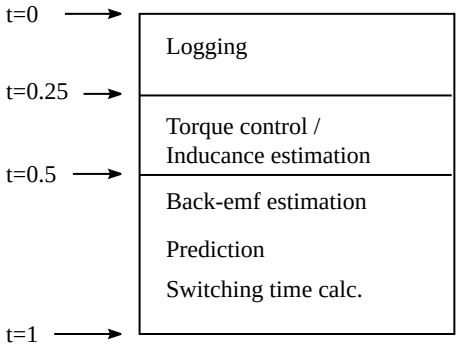


Figure 5.18: Loop structure

into two quarter-cycles and a half-cycle was governed by acquisition of current samples at the boundaries. Most of the current control algorithm operates in the second half of the cycle, after the $t = 0.5$ measurement is available.

Each iteration of the control is executed in the `CCCalc()` routine, starting at line 317. The routine starts by waiting for the start of the control interval, which is signalled by the data acquisition EPLD. The following code fragment shows the method.

```
while (ReadInPAL(5)&1);          /* wait for first measurement period*/
ThM=ReadInPAL(4)&0x3FF;          /* read position here*/
                                                                    335
ia=ReadInPAL(0);                 /* read the samples*/
ib=ReadInPAL(1);
ic=ReadInPAL(2);
```

The first line is the polled wait for the start of the interval. This ensures the correct synchronisation of the algorithm. Following this, the value is read from the position transducer. The firmware converts the grey-code into a simple 10-bit binary count. The currents themselves are read in with the final three lines.

After converting the values to the two-phase coordinates, the remainder of the first quarter-cycle is used to perform any logging or interface tasks.

5.6.3 Torque Controller

The time available in the second quarter-cycle is used to either calculate the torque control or the inductance estimate. The torque control variables are only calculated in one out of every sixteen control cycles. In the other cycles this time is instead used for the inductance estimator.

A standard indirect field-oriented controller has been used to provide torque control for the machine. The algorithm is the same as that presented in Section 2.2.2. Unlike the current controller, the variables are represented as true values, rather than scaled fixed-point values. The basic updating code is:

```
/*TORQUE CONTROLLER*/
Isy=Tr/(3*Pp/2*Lm*Lm/Lr*Isx);                                    450
if (Isy>MaxIsy) Isy=MaxIsy;
else if (Isy<−MaxIsy) Isy=−MaxIsy;

Wsl=Isy*Rr/(1.0*Lr*Isx);
if (Wsl>120) Wsl=120;                                            455
```

This is simply an implementation of the equations in Section 2.2.2, with limiting added. The limiting on I_{sy} is necessary to prevent the torque controller generating set-points that exceed the machine capability. The value for I_{sx} is supplied by the user to set the level of magnetising flux.

5.6.4 Current Control

The first part of the current control algorithm is to estimate the present back-emf. This may be integrated into a single updating equation, as shown in Equation (3.39), but in this case the back-emf is separately estimated. Although the performance is slightly lower, this allows monitoring and manipulation of the estimates. The back-emfs are estimated with the equations:

```
/*calc back-emfs*/
ed=AlphaD+LEst/VDC*1.0*(ID0−ID2);                                510
eq=AlphaQ+LEst/VDC*1.0*(IQ0−IQ2);
```

These back-emf values may then be used to obtain an estimate of the electrical rotation speed as described in Section 4.3.3. This estimate is then filtered, and used in the back-emf prediction stage. The prediction and voltages are obtained through the code:

```
ed_pred=ed−DeltaThEf*eq*3;
eq_pred=eq+DeltaThEf*ed;
                                                                    530
AlphaD=ed_pred+LEst/VDC*0.5*(IDR−2*ID2+ID0);
AlphaQ=eq_pred+LEst/VDC*0.5*(IQR−2*IQ2+IQ0);
```

The scaling of 3 on the *eq* multiplication is necessary because of the non-even scaling of the direct and quadrature axes. Although primarily used for step-ahead prediction, the predicted back-emf values may also be used in the following iteration as a filter on the back-emf estimate.

5.6.5 Space-Vector Modulation

Once the desired duty cycles have been calculated, the remainder of the code is addressed at the space-vector modulation. The first task is to determine the which sector contains the desired voltage vector. As described in Section 5.4.1, the sector may be determined using simple comparisons,

```
Sector=0;
/*****calculate the sector*****/                                    540
if (AlphaD<0) Sector|=4;
if (AlphaQ<0) Sector|=2;
if (absAQ−absAD<0) Sector|=1;
```

With the sectors determined, the next operation is to find the switching times t_0, t_1, and t_2. This is done with basic additions and subtractions,

```
if (!(Sector&1)) {      /*sectors 2 and 5   */
  T1=−AlphaD+absAQ;
  T2=AlphaD+absAQ;
}
else   {                                                            550
  if(!(Sector&4)) { /*SECTORS 1,6 */
    T1=AlphaD−absAQ;
    T2=2*absAQ;
  }
  else {   /*SECTORS 3,4 */                                        555
    T1=2*absAQ;
    T2=−AlphaD−absAQ;
```

```
    }
  }
  T0=T/2−T1−T2;                                                    560
```

The remaining task is to ensure that the switching times are feasible. The only problem that can occur is that the value of t_0 may be negative. This occurs when the voltage vector supplied is outside the range of the inverter output. The solution used here is to scale back the values of t_1 and t_2 so that they fit within the switching period. Once any scaling has been performed, the values are written out to the PWM hardware, and the loop repeated.

5.7 Controller Performance

The performance of the controller has been evaluated using an inverter and a 7.5kW induction machine. A DC machine is connected to the output shaft to act as a load. A diagram of the test hardware arrangement is shown in Figure 5.19.

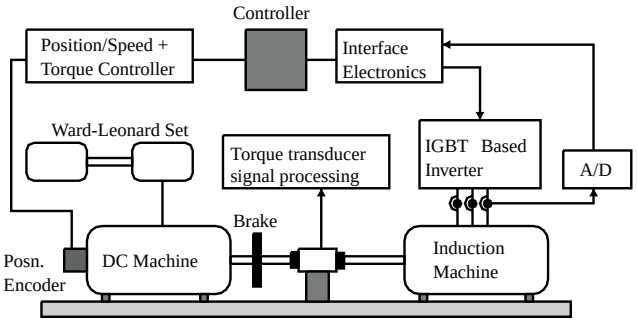


Figure 5.19: The test system hardware

A star-wound machine was used, with the parameters displayed in Table 5.2.

Parameter			Value
Rated Power	P	=	7.5kW
Core Losses	R_c	=	430Ω
Magnetising reactance	X_m	=	35.9Ω/Ph
Stator resistance	R_1	=	0.57Ω
Rotor resistance	r_2'	=	0.80Ω
Leakage Reactance	$X_1 + x_2'$	=	3.12Ω
Frequency	f	=	$50Hz$

Table 5.2: Test machine parameters.

5.7.1 Back-Emf prediction

The first stage in the current control algorithm is the estimation of the present back-emf. In Section 4.3.3, a method was also proposed to predict the back-emf for the subsequent control cycle. A plot of these two variables is shown in Figure 5.20. The

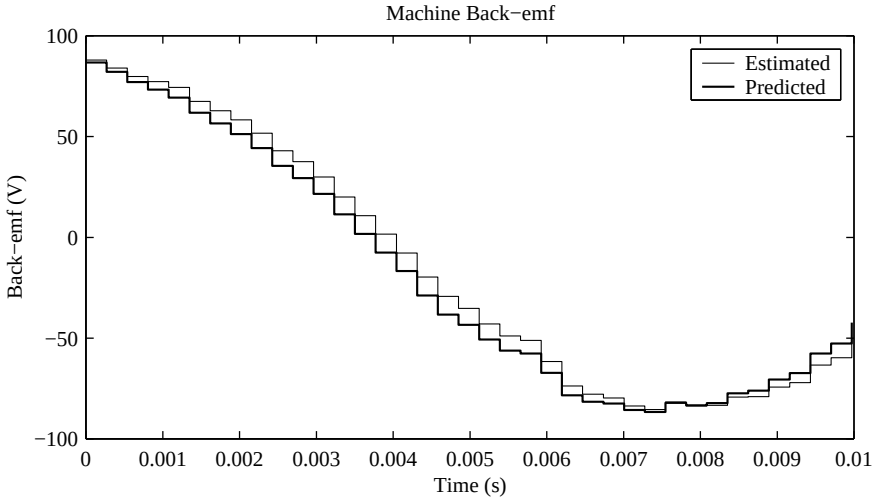


Figure 5.20: Back-emf prediction performance.

main feature of this plot is the performance of the prediction algorithm. Except in the case of disturbances, it performs well at predicting the future value, and is similar in appearance to the simulation result shown in Figure 4.32.

Under transient conditions, errors may occur in the back-emf estimate. This occurs when the inductance estimate does not match the real machine parameter exactly. In fact, in Section 4.2.3 it was shown that the resulting error in the back-emf estimate helps compensate for the error in the inductance estimate. Figure 5.21 shows a period of ripple on the back-emf that is due to a step change in the current.

The oscillations die down after a few control cycles, but they may also be significantly reduced by filtering the estimate using the predicted value from the previous cycle.

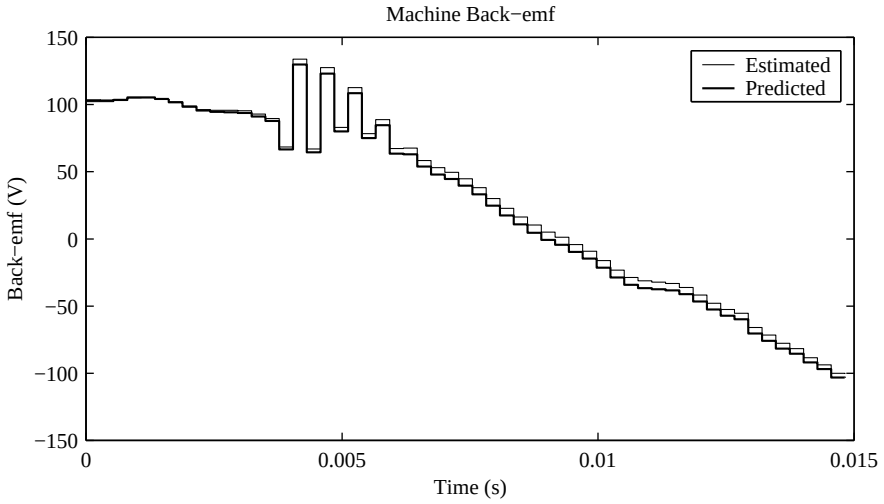


Figure 5.21: Back-emf under a set-point current transient.

5.7.2 Current Tracking

The steady-state tracking ability of the controller is shown in Figure 5.22. This shows the reference and machine current with the machine operating at full speed, and is equivalent to the system simulated in Figure 4.33. The plot data was obtained through the data logging facility integrated into the controller software. The current at the endpoint of the

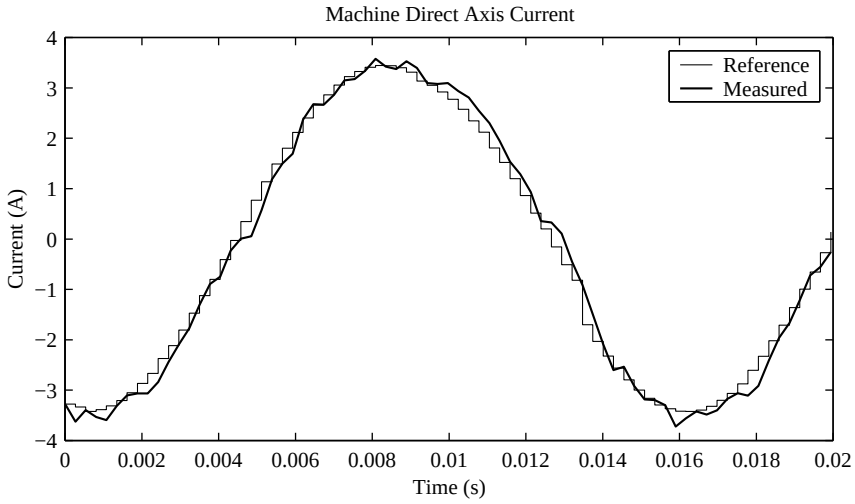


Figure 5.22: Current tracking at rated speed.

interval generally matches the reference well. Flat areas are visible near the zero-crossing, which indicate dead-time effects.

Frequency Domain

The spectrum of the machine current is shown in Figure 5.23. This shows a single dominant peak at the fundamental 50Hz. There is at least 35dB of separation between

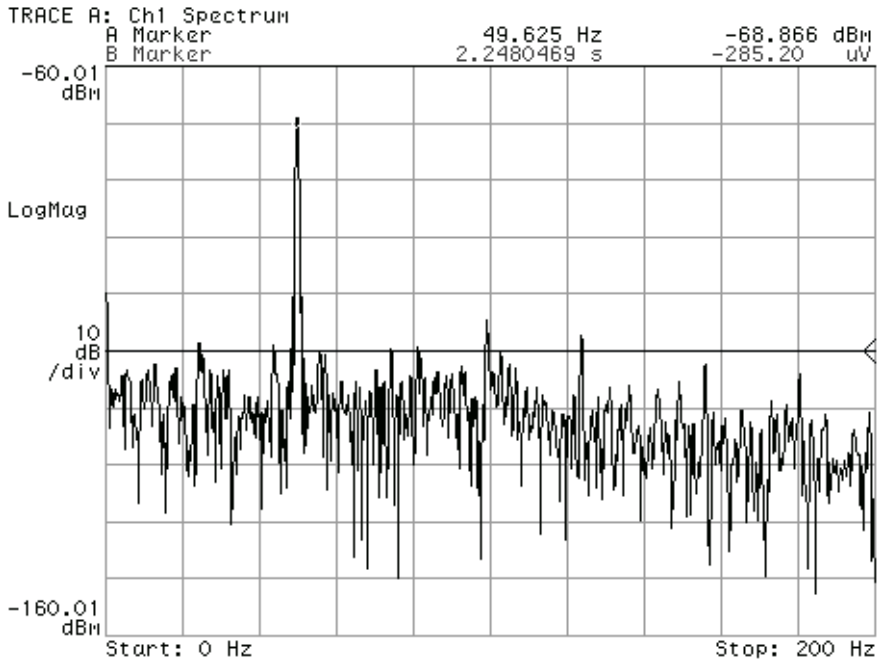


Figure 5.23: Spectrum of the output current.

the fundamental, and any other frequency component. This shows that the controller's 8dB sensitivity peak (§4.3.3) is not an obstacle to performance.

The high-frequency current spectrum is shown in Figure 5.24. This shows the expected spectral content for the space-vector PWM. The test was performed at approximately a 2.9kHz switching frequency, and the main peak is visible at twice that frequency.

Transients

The transient performance of the controller was evaluated using a step change in the torque set-point. A resulting plot of the $d-axis$ current is shown in Figure 5.25. There is a small amount of overshoot and oscillation, which is consistent with mismatches between the true and the estimated parameters. Similar behaviour was observed in the simulation shown in Figure 4.34.

A plot of a current transient from an oscilloscope is shown in Figure 5.26. This plot also shows the current ripple resulting from the PWM.

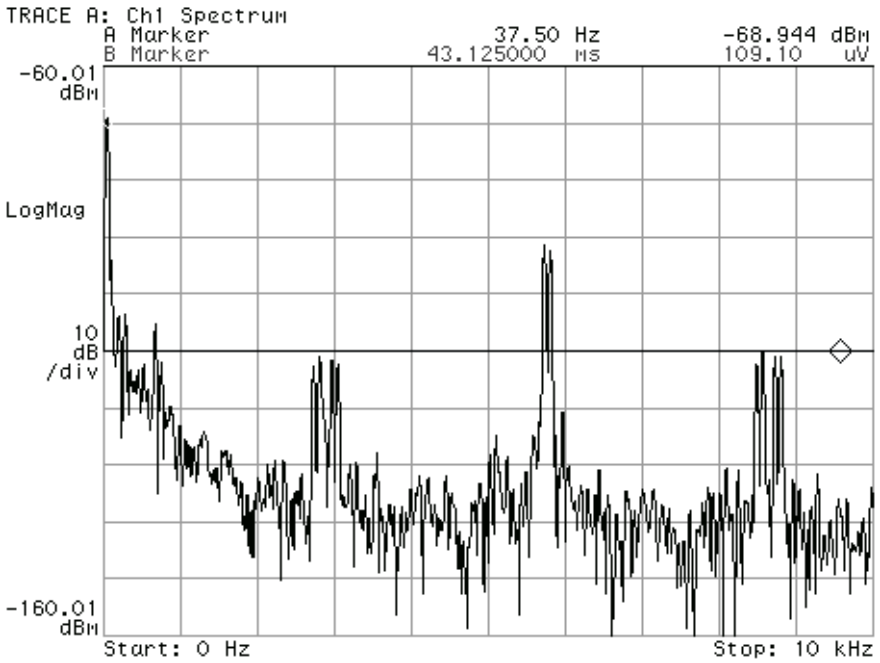


Figure 5.24: Spectral content at the switching frequency.

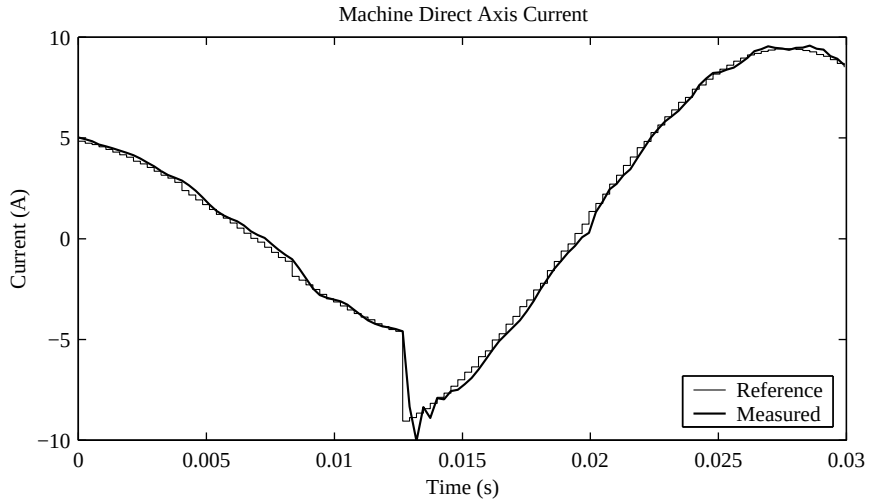


Figure 5.25: Response to a set-point transient.

5.7.3 Inductance Estimation

The results shown above were all performed with the inductance estimator enabled. The performance of the inductance estimator is difficult to assess, as the true value for a particular operating point is not known. Instead, the result may be verified through the operation of the controller. Figure 5.27 shows the ability of the estimator to track from an initially incorrect value. The final value of 9mH is close to the 10mH estimated from the no load/blocked rotor tests. Approximately 0.3s is required to adapt to the change

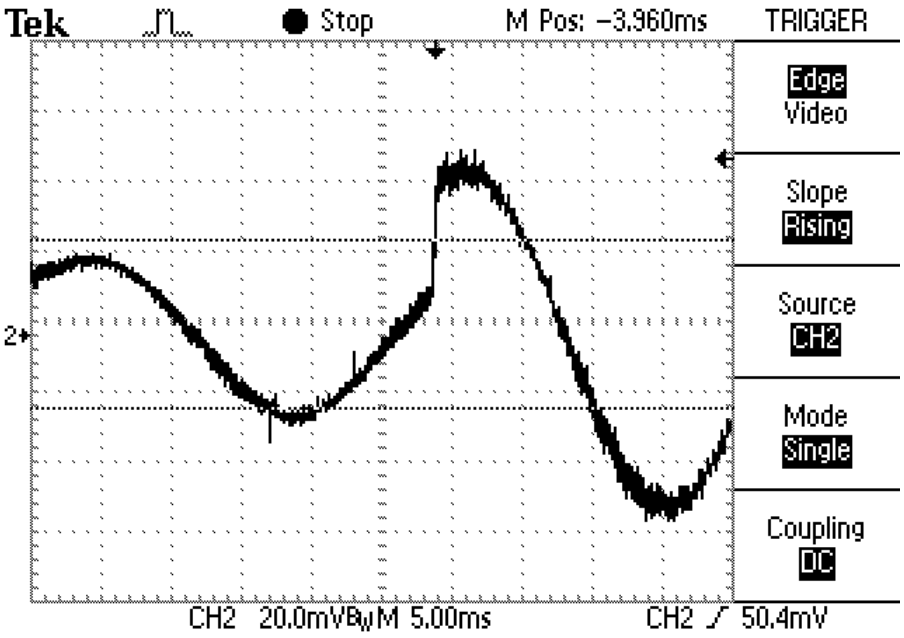
in parameter value. The estimator's forgetting factor provides a trade-off between the tracking speed and steady-state ripple.

From the stability analysis(§4.3.3), it is known that the controller becomes unstable when the inductance estimate exceeds about 1.3 times the true value. Figure 5.28 shows the performance of the controller obtained after setting the inductance parameter to 1.2 times the estimated value. The oscillations that occur indicate that the limit of stability is being reached at that point. This shows that the inductance estimate is consistent with the behaviour expected from the model. With back-emf extrapolation enabled, the expected instability at $\Delta L = 0.5$ is also observed.

5.8 Conclusions

This chapter has demonstrated an implementation of the current controller described in the previous chapters. Although the control algorithm itself is quite simple, a large amount of supporting logic and software is necessary for an operational system. Much of this logic has been implemented in a way that is suitable for integration into the hardware controller, which is presented in the following chapter.

The controller performance shows good agreement with the theoretical and simulation analysis in the previous chapters. The current controller implemented does not depend on knowledge of any machine parameters, yet offers good steady-state and tracking performance.


Tek
Stop
M Pos: -3.960ms
TRIGGER
Edge
Video
Slope
Rising
Source
CH2
Mode
Single
Coupling
DC
2+
CH2 20.0mVBwM 5.00ms
CH2 ⎍ 50.4mV

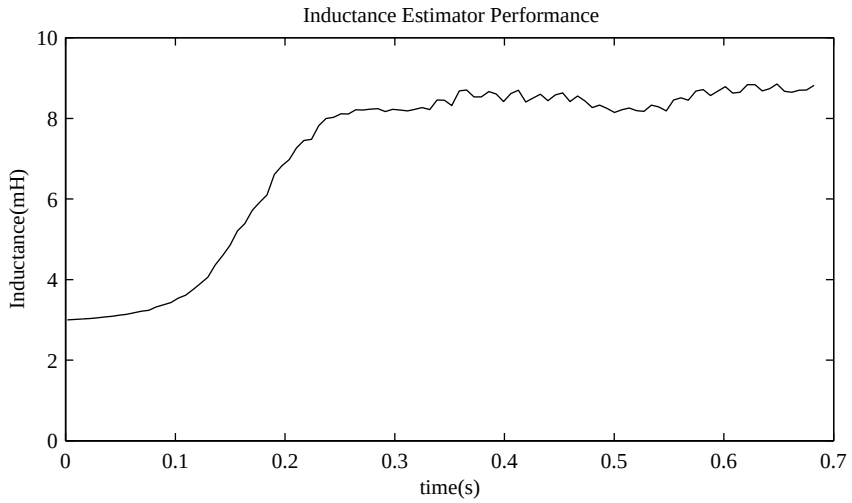


Figure 5.27: The inductance estimator in operation.

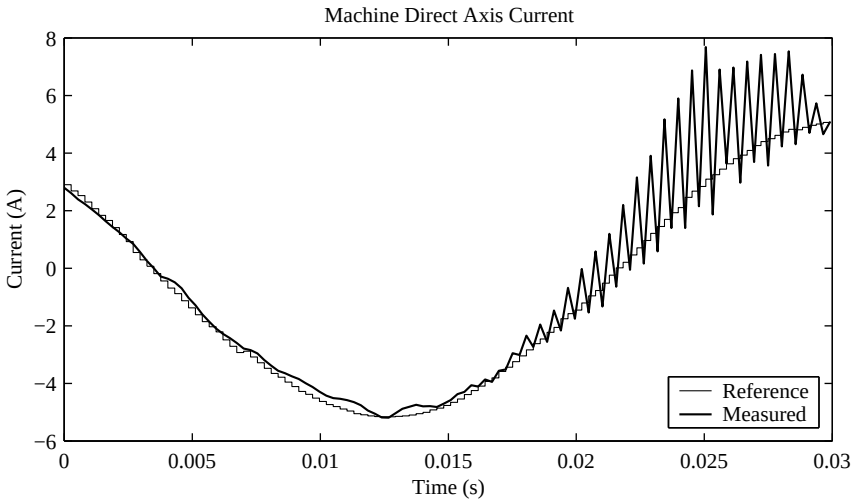


Figure 5.28: .

Chapter 6

Hardware Implementation

6.1 Introduction

One of the principal benefits of the current controller described in this thesis is its relatively simple computational structure. This allows a fast random-logic hardware implementation to be developed. The advantage of a random-logic controller is that it may be easily integrated into a single EPLD or ASIC. This reduces the circuit complexity to that of the simplest analogue controllers, bringing high-performance control to a greater range of applications.

The availability of parallel computation and specialised logic also brings with it greater computational power. This offers the potential for higher control rates. In the literature, the potential benefits of direct hardware implementation have been noted. However, some of the solutions published are limited to having little more than the modulation code in the logic device[45]. An EPLD design incorporating both a PI controller and a delta modulator has appeared, which can achieve switching rates in excess of 20kHz[26]. These hardware controllers have been concentrated more in the area of switched reluctance machines[5].

This chapter describes the hardware current controller for induction machines that has been constructed. After a brief overview of the overall concept, the control equations are reviewed for the purposes of hardware implementation. This leads to a hardware controller design, and then a description of the architecture. Some practial results are shown to verify the controller operation.

6.2 Physical Structure

An aim of the hardware controller is to maximise the on-chip integration. The connection of the current controller chip to the rest of the system is shown in Figure 6.1. The only external components are the inverter, A/D converters and the outer loop controller. Future developments could include the integration of a field-oriented torque controller into the current control chip, leaving only the application-specific controller.

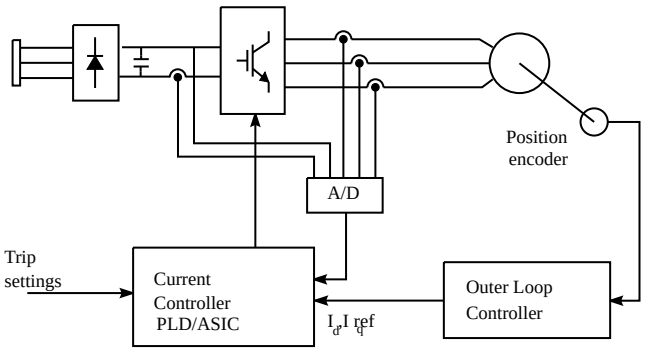


Figure 6.1: Connection of the current controller chip.

In order to achieve this level of integration, there are several other units on the chip in addition to the computational unit. These include the data acquisition system, as described in the previous chapter. The acquisition system was designed to be suitable for the hardware controller, and so requires minimal modification.

Once the current samples are received, they are compared using digital comparators with the trip levels. The drive is disabled when trip levels are reached. In the prototype, the trip levels were set by the outer loop controller, but this could easily be achieved in a variety of other ways, such as DIP switches on the controller PCB.

Another unit in the chip is the PWM generator. Again, the space-vector PWM module from the previous chapter is used. It includes programmable deadbands and deadband compensation. The chip may be programmed to operate over a large range of switching frequencies, up to approximately 20kHz with a pulse timing accuracy of 70ns. Energy dumping regeneration is supported, with integrated control for a brake chopper.

For the tests performed, the current controller EPLD was interfaced to an Intel 80186 processor. This processor was used to provide the set-point currents for the current controller. A field-oriented controller was used on that processor to provide torque control to the machine. This torque controller is similar to that implemented in the software version, and is not detailed here. A photograph of the EPLD development board, with the 80186 daughter-board attached, is shown in Figure 6.2.

6.3 Hardware Implementation issues

The basic algorithm, as developed in Chapter 3, can be manipulated to minimize the number of arithmetic operations and improve scaling of the values. This allows a simpler and more efficient hardware implementation with an add/subtract/shift ALU. This section outlines these alterations.

6.3.1 Duty Cycle

For a given control cycle, k, the following quantities are used:

Figure 6.2: Photo of the controller board.

- V: The DC link voltage
- $\alpha[k, k+1]$: The PWM duty cycle on the given axis.
- $e[k, k+1]$: The machine back-emf.
- $i[k]$: The machine current at the start of the interval.
- $i[k+0.5]$: The machine current in the middle of the interval.
- $u[k]$: The set-point current for the start of the interval.

The machine back-emf may be estimated by the expression found in (3.34):

$$e_{av}[k, k+1] \approx V\alpha[k, k+1] + \frac{2L}{T}(i[k] - i[k+0.5]). \tag{6.1}$$

This same expression is then re-arranged to calculate the duty cycle required to reach the setpoint,

$$\alpha[k, k+1] = \frac{1}{V}e_{av}[k, k+1] + \frac{\rho L}{TV}(u[k+1] - i[k]), \tag{6.2}$$

where ρ was defined in Equation (3.36), and defines endpoint or average control, with values of 1 and 2 respectively. The value $i[k]$ is needed before the start of the kth interval, and may be estimated as,

$$i[k] \approx 2i[k-0.5] - i[k-1]. \tag{6.3}$$

By combining these results, a single updating equation may be found. For endpoint control, this may be expressed as,

$$\frac{T}{2}\alpha[k, k+1] = \frac{T}{2}\alpha[k-1, k] + \frac{L}{2V}(u[k, k+1] - 4i[k-0.5] + 3i[k-1]). \tag{6.4}$$

Alternatively, with average control, this equation is,

$$\frac{T}{2}\alpha[k,k+1] = \frac{T}{2}\alpha[k-1,k] + \frac{L}{V}(u[k,k+1] - 3i[k-0.5] + 2i[k-1]). \tag{6.5}$$

The $\frac{T}{2}\alpha$ variable is a convenient representation of duty cycle because it appears in various forms in the conversion to the switching times.

Using the definitions,

$$i_x \triangleq 3i_d \tag{6.6}$$
$$i_y \triangleq \sqrt{3}iq \tag{6.7}$$
$$\alpha_x \triangleq \frac{T}{2}\alpha_d \tag{6.8}$$
$$\alpha_y \triangleq \frac{T}{2\sqrt{3}}\alpha_q. \tag{6.9}$$

The standard expressions for converting three phase to two phase currents are:

$$i_x = 2i_a - i_b - i_c \tag{6.10}$$
$$i_y = i_b - i_c. \tag{6.11}$$

To further reduce the total operations the currents used in the internal variables are i_x and i_y. Therefore, for endpoint control, the duty cycle equations become:

$$\alpha_x[k,k+1] = \alpha_x[k-1,k] + \frac{L}{6V}(3u[k,k+1] - 4i_x[k-0.5] + 3i_x[k-1]) \tag{6.12}$$
$$\alpha_y[k,k+1] = \alpha_y[k-1,k] + \frac{L}{6V}(\sqrt{3}u[k,k+1] - 4i_y[k-0.5] + 3i_y[k-1]). \tag{6.13}$$

6.3.2 Switching times

The space-vector switching time expressions originally appeared in Table 3.2. Since the duty cycles are represented internally as $\frac{T\alpha_d}{2}$ and $\frac{T\alpha_q}{2\sqrt{3}}$ only one multiply operation is required to calculate t_1 and t_2. t_0 is calculated using $t_0 = T/2 - (t_1 + t_2)$.

Using the new parameterisation, the switching time calcualtions are computationally very simple. Table 6.1 shows the revised version of the original switching times from Table 3.2.

Voltage limits must be applied to the output so that it may be physically realisable with a switching waveform. Using the above method to calculate switching times, limiting must be applied when the switching times satisfy $2(t_1 + t_2) > T$. This may be detected by examining the sign of the calculated t_0. In order to reduce the total amount of computation, all of the switching times are first calculated assuming that there are no limits to be observed. This calculation is not significantly more complex than that required for checking the limits alone.

The limit is applied by maintaining the angle of the voltage vector, but scaling the

Sector	Firing Order	t_1	t_2	Conditions
1	01277210	$\alpha_x - \alpha_y$	$2\alpha_y$	$\alpha_x > \alpha_y; \alpha_y > 0$
2	03277230	$-\alpha_x + \alpha_y$	$\alpha_x + \alpha_y$	$\alpha_x < \alpha_y; \alpha_y > -\alpha_x$
3	03477430	$2\alpha_y$	$-\alpha_x - \alpha_y$	$\alpha_x < -\alpha_y; \alpha_y > 0$
4	05477450	$-2\alpha_y$	$-\alpha_x + \alpha_y$	$\alpha_x < -\alpha_y; \alpha_y < 0$
5	05677650	$-\alpha_x - \alpha_y$	$\alpha_x - \alpha_y$	$\alpha_x < -\alpha_y; \alpha_y < \alpha_x$
6	01677610	$\alpha_x + \alpha_y$	$-2\alpha_y$	$\alpha_x > -\alpha_y; \alpha_y < 0$

Table 6.1: PWM switching times and sector conditions with the new parameterisation.

magnitude by a value γ. The required γ is:

$$\gamma = \frac{T}{2(t_1 + t_2)}. \tag{6.14}$$

The duty cycles are modified through the calculation $\alpha_{dlim} = \gamma\alpha_d$ and $\alpha_{qlim} = \gamma\alpha_q$. As these are the actual voltages applied, the limited values must be used for the back emf estimate on the next control cycle.

More complex approaches may be applied to the limiting, to aviod torque set-point changes affecting the flux. One example was described in Section 3.3.2. Due to the additional complexity and parameter dependence, this method has not yet been implemented in hardware.

6.3.3 Inductance estimation

The machine leakage inductance is estimated using the method described in Section 3.4. The values produced from this expression are low pass filtered. An efficient first order filter implementation from a hardware perspective is:

$$L_{k+1} = L_k + \frac{L_{est} - L_k}{2^n}, \tag{6.15}$$

where n determines the time constant.

6.4 Computational Architecture

The current control algorithm described above has been implemented in an Altera® FLEX10K series EPLD (Erasable Programmable Logic Device). Note that while an EPLD was used only for a prototype, an ASIC (Application Specific Integrated Circuit) would be the optimal final implementation. The overall design of the chip hardware is shown in Figure 6.3. There are three main sections used in the architecture:

1. Data acquisition: The analog currents are sampled using Hall Effect transducers and the values are converted to digital values using a serial A/D, and then transmitted to the EPLD via an isolated serial channel that incorporates error detection.

Over-current protection on each phase, together with DC-link protection, is implemented digitally in the EPLD. A 5Mbps link is used, allowing measurement at 250k samples per second. Given the machine inductance, this is fast enough to offer an additional level of protection over that offered by the drivers themselves. The primary use of the EPLD protection is to protect against overcurrents arising from poor or incorrect behaviour of the current controller. An external physical short circuit would require the intervention of the protection within the switching modules.

2. Computational unit: This consists of an ALU and its associated sequencer. The ALU is capable of 16 bit addition, subtraction and shifting. A 32 word register file is used to store intermediate values. The sequencer controls the type of operations performed in the ALU and their sequence, and is essentially microprogrammed.

3. PWM generator: This constructs the three phase switching pattern from the switching times calculated in the ALU. As an option these times can be compensated to account for inverter dead time.

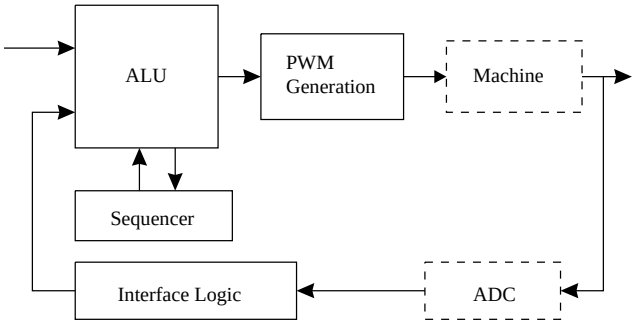


Figure 6.3: Overall block diagram of the chip.

6.4.1 Computational Structure

Concurrent and Sequential Calculation

Two basic approaches were considered in designing the computational structure of the controller hardware. One approach is to allocate a logic block for each of the calculations to be performed. For example, each multiplication calculation required for the control algorithm would have a multiplier block dedicated to it. Many of the calculations are then performed concurrently, in separate sections of the logic. This approach maximises the computational throughput, but is very demanding on the number of logic components required. Multipliers and Dividers, in particular, consume a large number of logic elements.

The second approach involves time-multiplexing the computational resources. A series of different calculations are performed sequentially using the same logic block. This

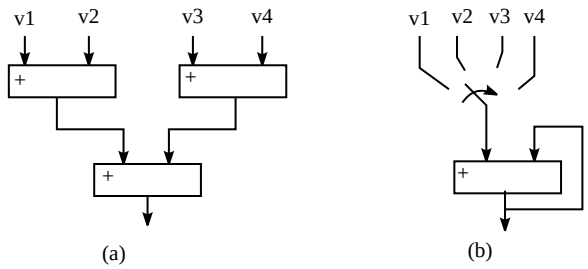


Figure 6.4: Calculation logic alternatives: (a) Random Logic, (b) State Machine

forms a trade-off between speed and required logic resources. The time multiplexing is achieved through the use of registers and multiplexers. The inputs to the computational unit are connected to a multiplexer to select the source of each calculation, while results are temporarily stored in registers.

Figure 6.4 shows a comparison between these two approaches for the simple case of adding four numbers. In the first case, (a), three two-input adder blocks are used to perform the additon. In the second case, (b), only one adder block is used, but it performs a number of separate calculations. An external state machine controls the input selection.

The random logic approach avoids the need for bus logic, multiplexers, registers and a state machine, but requires more resources for adders and multipliers. In comparison, the state machine design requires fewer on-chip resources, but sacrifices speed, especially for complex calculations. The hardware logic based solution will give increased speed of execution compared to the state machine design, but will consume more on-chip resources.

An architecture decision must be made based on the computational and logic resources and demands of the given problem. In this case, the bulk of the calculations are carried out in the state machine fashion in a single arithmetic logic unit (ALU). Those that need greater throughput, or don't easily fit into the basic architcture, are added as separate calculation units.

ALU Connection Arrangement

A state-machine sequential design philosophy has been used for the chip design. Most of the calculations for the control algorithm are performed in a single arithmetic logic unit (ALU). The system inputs and outputs are connected to the ALU via a common bus as shown in Figure 6.5. A register file is used to store the intermediate results, such as past current measurements. This arrangement allows the calculations to be performed in sequence as they would be in a microprocessor.

This design approach was chosen as it was appropriate for the speed and size requirements of the current control algorithm. Greater levels of parallel computation would only be required for control rates significantly above 20kHz. Furthermore the Altera 10K EPLD chosen for the prototype is particularly amenable to this approach as it can

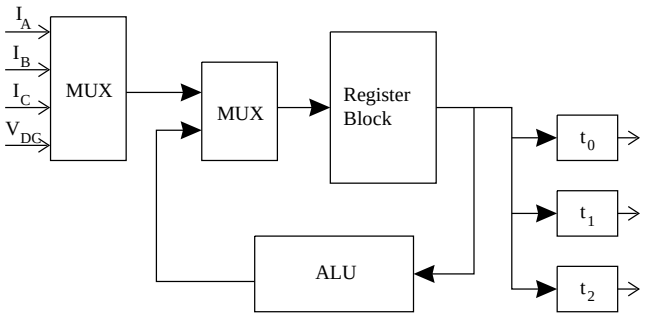


Figure 6.5: Input and output connections to the ALU

efficiently implement register files and memory required for this type of design.

From Figure 6.5, it can be seen that external input values enter the ALU via a single 16 bit wide data path. The source for this path is selected by the input multiplexers, giving a choice between external inputs or internal register values. The multiplexers are controlled by the sequencer. The register file consists of storage for 32 values, each 16 bits in width. These locations are used for storing parameters and intermediate calculation results. The output of each calculation may be latched into any of the output latches or written back into the register memory. The overall operation of this is controlled by the sequencer, which is described later, in Section 6.5.

6.4.2 ALU Structure

Computational Requirements

An aim of the controller design is to offer control of inverters with switching rates of up to 20kHz. While this could easily be acheived by calculating the control algorithm at a lower rate than the switching frequency, the resulting control would be inferior. Instead, the controller should be able to operate at the full 20kHz rate. As the current control algorithm requires measurements from the mid-point of the previous control interval, only half of each 50μs interval is available for computation. In addition, another 5μs should be allowed for data acquisition delays. This leaves 20μs to calculate the control algorithm. The design specifications are summarised in Table 6.2.

Parameter	Value
Switching Rate	20kHz
Control Rate	20kHz
Half-period Duration	25μs
I/O delays	5μs
Remaining Calculation Time	20μs

Table 6.2: Design Specifications.

An Altera® FLEX10K50 EPLD was chosen for the prototype implementation. This

decision was made primarily on the availability of development tools, cost, packaging and availability at the time of initial design. This family's use of embedded arrays was found useful for the design, as it allowed the efficient implementation of on-chip RAM and ROM resources.

The FLEX10K50 device has a rated capacity of 50,000 typical gates. Preliminary design tests indicated that this would be adequate for the controller. Initial testing also revealed that a synchronous 16-bit adder could operate at clock rates of up to 40MHz. For a more realistic circuit, the maximum frequency falls as the amount of logic between the flip-flops increases.

To allow for more complex logic without excessive pipelining, 20MHz was chosen as a target clock rate. This frequency allows 1024 clock cycles per control cycle, which means it is also convenient to use the same clock for the the output modulator. In order to use the data acquisition system detailed in the previous chapter, the sampling logic would operate at a maximum of 5MHz, which is one quarter of the master clock. This slower clock rate is not necessary for the logic, but rather for the bandwidth of the serial communications link. The resulting delay in obtaining samples is,

$$t_d = 20 \times \frac{1}{5MHz} = 80 \times \frac{1}{20MHz}. \tag{6.16}$$

After removing these 80 clock cycles from the half-interval, there remains 432 cycles for the execution of the control algorithm.

Resource Allocation

The structure of the ALU was governed by the operations required to execute the current controller algorithm. It is designed to be fast in the calculations that are performed often, while operations that are less frequently used are optimised more for logic space than speed.

An important design decision was the style of computation block to use for the various types of calcuation. Multiplication and division operations, in particular are likely to be time or resource intensive, but the simpler addition, subtraction and shifts are generally more common. Fortunately other difficult operations, like square root evaluation are not necessary for this algorithm. The approximate requirements (excluding the inductance estimation) are shown in Table 6.3. Due to its particular precision requirements, the inductance estimation uses additional hardware.

The available options for multiplication have varying demands on logic complexity and computation time. Direct multiplication logic could reasonably perform the calculation in the order of 50ns on the EPLD, but use a large proportion of its resources. Using simple shift/add techniques, the calculation time rises to the order of $1.5\mu s$.

The small number of multiply/divide operations required means that a shift/add/subtract implementation is sufficient for the application. The result of this is a design decision

Operation	Number
multiply	3
divide	3
add/subtract	60
shifts	8

Table 6.3: Operations required per iteration.

for a primarly Arithmetic Logic Unit (ALU) with add, subtract and shift functionality. This is a standard feature of a basic microprocesser design[18]. The ALU was designed to perform a 16×16 bit signed multiplication in 32 clock cycles, and a 16 bit division in 64 cycles.

Figure 6.6 shows the basic structure of the ALU implemented . Each iteration of the algorithm requires a number of arithmetic operations using this unit.

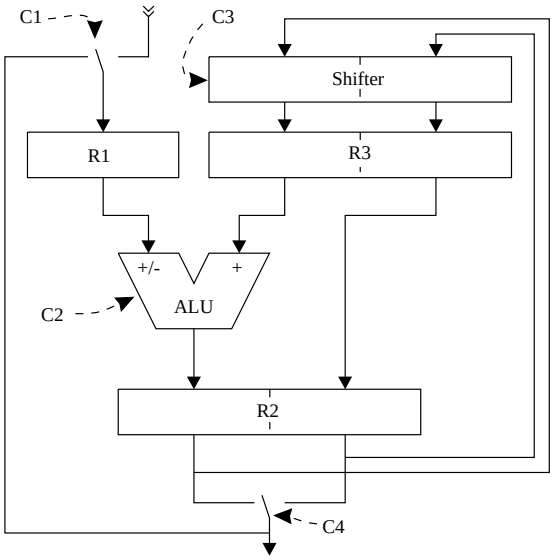


Figure 6.6: Block diagram of the ALU

Precision

The ALU was designed to handle 16-bit word lengths. Larger word lengths offer greater flexibility but incur a greater cost in logic complexity. The required precision is derived directly from that of the inputs and outputs to the systems. The data acquisition subsystem offers 12-bit samples, while the output precision is ultimately limited by the controllability of the output switching pattern. With dead times of approximatey 3μs, and control rates as low as 1kHz, a 10-bit word length offers sufficient precision.

The intermediate calculations require an accumulator with greater precision than the input data to avoid arithmetic overflow. The additional 4-bits provides this required

value range. There are also 32-bit registers to allow the manipulation of the results of the multiply and divide operations.

6.4.3 ALU Control

Figure 6.6 shows that the ALU has one input and one output. For operations requiring two parameters, such as addition, one parameter is read in first and stored in the internal registers of the ALU. On each clock cycle, a new value is read into each of the registers $R1$, $R2$ and $R3$. The source of the values is controlled by the command inputs $C1$, $C2$ and $C3$ respectively. The command inputs are defined as short binary numbers, which reflect the internal representation. These commands control the operation of the ALU. The possible commands are described in the following paragraphs:

Input Selection

The ALU command $C1$ selects the source for new calculation inputs. This essentially involves controlling the input multiplexer. To minimise resource usage, there is only one input multiplexer, so operands for binary operations have to be loaded on sequential clock cycles.

C1	Description
00	R1=External read from Addr
01	R1=Read memory location Addr
10	R1=R2 (used for a=-a and a=abs(a))
11	R1 undefined, Write memory location Addr with R2

Table 6.4: ALU Input Select Commands

$C1 = 00$ selects the external input multiplexer as the data source. This is designed to connect to the external I/O subsections, and is principally for acquiring new sample values. For example one input to the multiplexer will be connected to the direct axis current measurement from the start of the present control cycle. The actual address supplied to the multplexer does not enter the ALU module itself, instead being directly supplied by the sequencer.

The other principle data source is the system register file. This is selected with $C1 = 01$. The operation of the register file as a data source is essentially the same as the external inputs. The address in the register file is again supplied by the sequencer.

Command $C1 = 10$ was specifically added to the ALU to enable the evaluation of absolute values. It copies the value of $R2$ back into $R1$. Normally results are fed back into the addition/subtraction block via $R3$, but this does not offer negation of the result. This alternate path offers this ability.

The command $C1 = 11$ is not related to the specification of the source for $R1$. Instead, it is an instruction to write back the contents of register $R2$ into the register file. It was included in $C1$ because it would otherwise conflict with the $C1$ operation. There are two reasons for the conflict:

1. The register file is most efficiently implemented as a single-port memory block. This means that a read and write cannot occur simultaneously. Dual port memory would be possible at the expense of greater complexity. A survey of an early version of the current controller instruction sequence showed that simultaneous access was infrequently desired.

2. In the final implementation, it was convenient to only have a single bus to specify the addresses for both the register file and the external inputs. This was to minimise resources, and was again evaluated on the draft instruction sequence.

Arithmetic Operation

The $C2$ instruction controls the core arithmetic operation of the arithmetic logic unit. This is a 4-bit data field, offering the possibiliy of sixteen different operations. The first eight operations provide the basic calculations, while the second half are for more specialised functionality. Figure 6.7 shows the bit definitions for the basic operations.

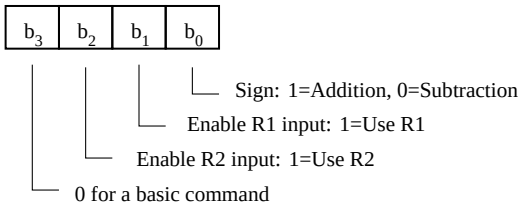


Figure 6.7: Bit definitions for basic C2 commands.

Table 6.5 shows a summary of the complete set of instructions available with $C2$. The special instructions were principally added to support efficient multiplication and division.

Shift Register Control

The shift register supplying register $R3$ allows manipulation of the 32-bit datapath section of the ALU. The left and right shifts are provided for the multiplication and division algorithms. A facility to shift right by 16-bits is also provided to enable pre-loading of the lower 16 bits of the registers $R2$ and $R3$. Table 6.6 shows a list of the available commands.

C2	Description
000x	R2=0
010x	R2=R3
0011	R2=R1
0010	R2=-R1
0111	R2=R3+R1
0110	R2=R3-R1
1000	Multiply Instruction 1
1001	Divide Instruction 1
1010	Divide Instruction 3
1011	R2=abs(R1)
11xx	Undefined

Table 6.5: ALU Calculation Selection

C3	Description
x00	R3=R2
001	R3=R2 shifted left
101	Shift+conditional add for division
x10	R3=R2 shifted right
x11	R3=R2 shifted right 16 bits

Table 6.6: ALU Shift Register Operation

C4	Description
0	Select low word of R2 for mem/ext write
1	Select high word

Table 6.7: Output Datapath Selection

Output Selection

The switch in $C4$ is to allow selection of the section of $R2$ to use for output. Normally only the high 16-bits of R2 is used in calculations, but the result of a multiplication occupies the full 32-bits. As the external datapath is only 16-bits in width, a multiplexer is required to select the required section.

6.4.4 ALU Operation

Usage Example

The operation of the ALU may be demonstrated with the simple example of $C = A + B$, where A, B, and C are locations in the register file. The actions on each rising edge of the system clock and the commands required to achieve them are:

1. Load A into R1 (C1=01). The value A would be contained in the register file at a specified address of $\&A$. This value of $\&A$ is supplied to the register file at the same time as the $C1$ command. The value of $C3$ is irrelevant as its result will not be used.

2. Load A into R2 (C2=0011). This simply involves copying the value in R1 into R2 without modification. The aim is to place the value into the datapath ready for later combination with the second operand.

3. Load B into R1 and A into R3 (C1=01 and C3=000). This operation prepares both of the operands for the calculation. This time the address of B is supplied to the register file.

4. Load $A + B$ into R2 (C2=0111). The actual arithmentic operation occurs at this stage, placing the result in R2.

5. Store $A + B$ into location C (C1=11 and C4=1). If the result is to be placed into a register, the address of that register is specified in this cycle. Note that in this case R1 cannot be loaded with a value for the next operation in the same cycle. If this is desired, an idle cycle must be inserted. This restriction incurs a small time penalty in the execution of the current control algorithm.

The ability to only supply one operand in each clock cycle turns the addition process from potentially one cycle, into a a 5 cycle procedure. For a practical set of instructions,

the penalty is not as great as this. For the current controller calculations, a number of values are typically added together, and once the calculation is started, there is only a small overhead in adding additional values into the sum.

Apart from reducing the multiplexer resources, this also simplifies the memory accessing scheme as only one value will need to be fetched at a time.

Pipelining

The ALU design, as shown in Figure 6.6 contains more registers than are required for the logic operation. Only one set of registers are required in the feedback path. The alternative ALU without these registers is shown in Figure 6.8.

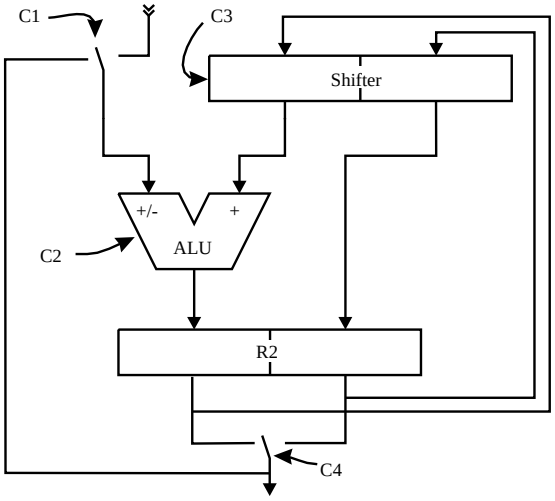


Figure 6.8: Alternative ALU without pipelining

This approach would be simpler, but suffers from greater propagation delays between logic blocks. On the test platform, cycle times of approximately 100ns were required for this configuration.

The addition of the extra register set approximately halves the maximum propagation delay between the output of a register set and the input to the next synchronous element. This reduction of propagation delay allows a doubling of the clock frequency. The cost to the instruction sequence is that each operaton requires double the number of clock cycles to complete, which in the simplest case negates the benefit.

The overall advantage of the additional register set is twofold:

1. The propagation delays throughout the synchronous design are better matched. Doubling the clock speed allows all of the other elements of the state machine to run at twice the speed, as this is one of the most complex sections. Thus, there is an advantage in keeping delays in different parts of the design to a similar levels.

2. The addition of the registers halves the time each of the computational units is active. This allows these units to be used for a subsequent calculation. In the

best case, when the reqiured operations are suited, the overall throughput may be doubled.

Dual Data Path

Instead of speeding the sequential execution, the pipelined architecture was designed for the execution of two concurrent instruction streams. This is possible through the use of two internal register sets (R2 and R1/R3). On each cycle, the contents of each register alternates between the two instruction paths. This approach offers less sequencing overhead than conventional pipelining techniques.

In the previous example, the odd steps (1,3,5) only used R1 and R3, while the even steps (2,4) only used R2. This means that the calculation throughput may be doubled by using R2 on the odd steps and R1/R3 on the even steps.

The current controller offers the opportunity for efficient utilisation of the dual data path as many of the calculations are duplicated for the two independent axes. One sequence could be used for the direct axis, and the other for the quadrature axis. In the prototype, this functionality was not used as the resources were sufficient without it. It does, however, offer opportunities for improved performance.

6.5 Sequencing Architecture

6.5.1 State Machine Implementation Issues

The sequencer generates the control signals for the ALU. On each clock cycle, a number of control signals need to be supplied. For a simple calculation sequence, this would be achieved through using a state machine. The state machine is composed of two blocks of combinatorial logic, along with a register to record the state. One block of logic calculates the next state, given the present state, and the inputs. In this case, the inputs would include the status information supplied from the ALU as well as other parts of the design. The actual binary numbers being manipulated are not directly connected to this logic. The output logic calculates the required control signals based on the present state. Figure 6.9 shows the structure of the state machine sequencer.

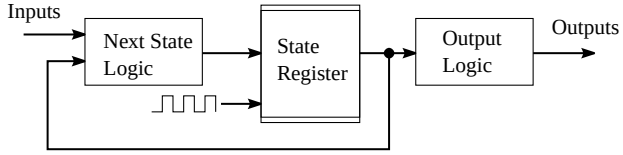


Figure 6.9: The state machine sequencer option.

At the lowest level, the state machine may be specified by discrete logic, but the state machine required for the calculation of the current control is quite large. Instead, a higher level specification is more appropriate. An attempt was made to design an

appropriate state machine using Altera's AHDL description language. This is similar in concept to VHDL, but better supported by Altera at the time of design. While this method of specification was satisfactory, the resulting design performed poorly.

The greatest problem was in propagation delays in the output-forming logic. The optimisation during design compilation was able to resolve the next state logic adequately. Techniques such as one-hot coding allow relatively simple next-state logic for designs that are largely sequential, as in this case. The outputs, however, are not a well structured function of the state. Each output depends on the value of almost all of the state variables. As a result the logic becomes slow and large as the number of states grows. State machine approaches were abandoned when it was found that the required number of states for this application was unwieldy.

6.5.2 The Microcoded State Machine

An alternative to using a large state machine is a sequencer which is based on using microcode stored in an internal ROM (Read-Only Memory). A small state machine is used to fetch the microcode words from the ROM and generate the commands for the ALU. A schematic diagram of this is shown in Figure 6.10.

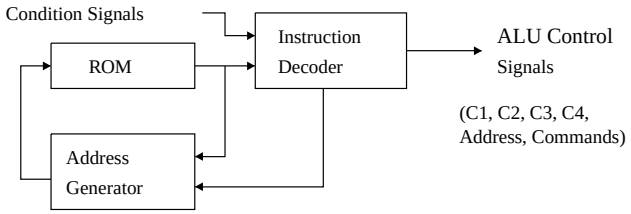


Figure 6.10: The structure of the microcode sequencer

The ROM essentially stores the information that would otherwise be contained in the state machine. In the simplest case, each successive address of the ROM would contain bit patterns to describe each control signal that needs to be defined. This quite simple approach would work for large parts of the current control algorithm, but is too limited for the complete design. Additional complexity is required to handle conditional state transitions. Some situations, such as applying limiting require different calculations to the normal set. These are handled with a form of branching, similar in concept to the JUMP instructions typically found in microprocessor instruction sets.

The ROM consists of 256 words of width 16 bits. This is quite a small structure, and may easily be incorporated into EPLDs and ASICs. In the prototype version, it is implemented within the EPLD. The address is provided by a presettable up-counter, which is controlled by the instruction decoder. Normal program execution is achieved by incrementing the counter and branches are implemented by presetting the counter to the new address. The specific design of the sequencer is shown in Figure 6.11.

This design is quite precise to ensure that the timing synchronises correctly with itself

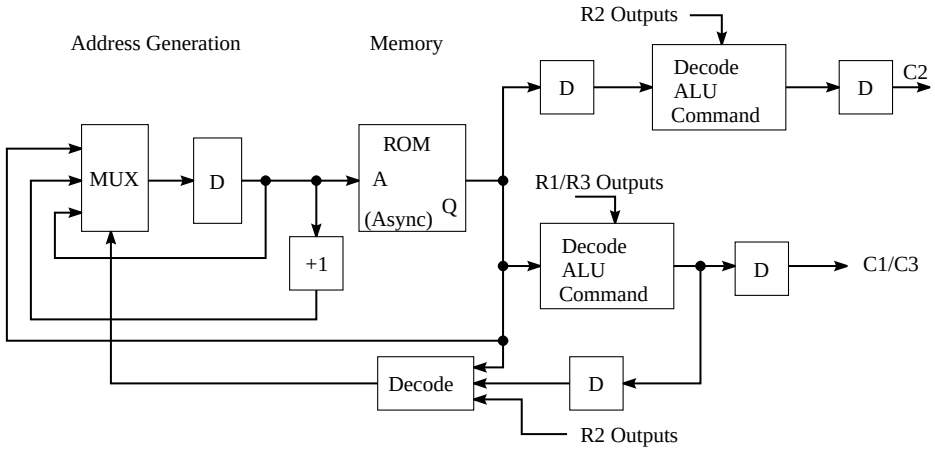


Figure 6.11: The sequencer design.

and the ALU. In particular, the two-clocks per cycle on the ALU necessitate different behaviour on odd and even clock cycles. The design could be simplified conceptually by insertion of appropriate delays on the ALU control signals so that all signals for each instruction cycle are applied simultaneously. This, however, would incur greater overall delays and increase the overhead on branching.

Figure 6.12 shows the timing relationship between various parts of the sequencer and computational engine. The overlap of sequential instructions in the pipeline spans three system clock cycles.

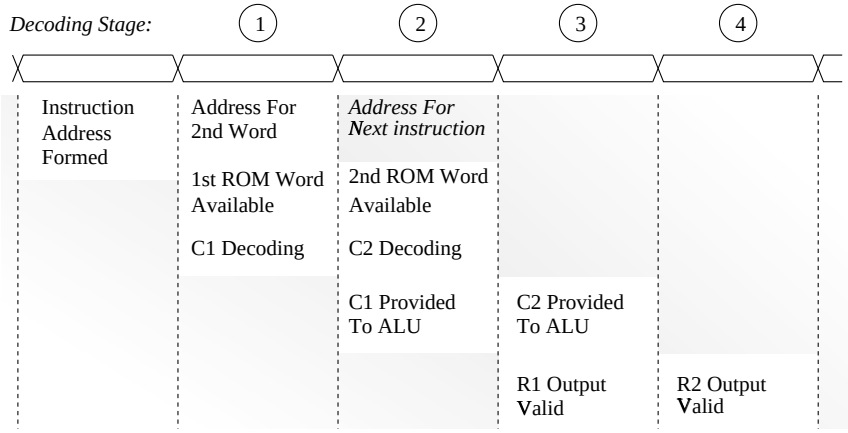


Figure 6.12: Sequencer Timing Diagram

The diagram shows that although each instruction takes five clock cycles to compm-lete, it occupies only two clock cycles in the hardware. Although not implemented, the dual data-path design of the ALU would allow an additional sequenced data path to appear at a one clock cycle delay, doubling the maximum throughput (§6.5.3).

Instruction Format

The instruction decoder reads the contents of the ROM to generate the required commands. Each microcode instruction consists of either one or two 16 bit memory words. The first word contains the primary commands for the ALU, while the second provides optional addresses and branch conditions if they are required. The format of these words is shown in Table 6.8.

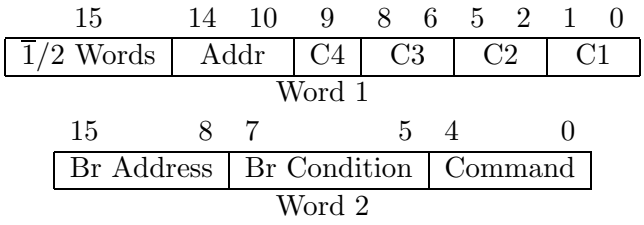


Table 6.8: Instruction Word Format

From this table, it can be seen that the microcode is largely horizontal in its design. Some narrowing has been used to reduce the instruction width to 32-bits, but there are no format fields. Instead each instruction field appears in every instruction. The role of each of the instruction fields is described in the following sections;

Word Selection: All instructions include the first word. The "$\overline{1}/2$ Words" bit indicates whether the second word is to follow. If this bit is set to zero, the second word is assumed to be zero, and the next word in the ROM is interpreted as the first word of the next instruction. This approach reduces the amount of chip resources required to represent the algorithm as the second word is only needed infrequently. The behaviour is summarised in Table 6.9.

$\overline{1}/2$ Words	Word 2	Next Instruction
0	All Fields=0	$Address + 1$
1	read from $Address + 1$	$Address + 2$

Table 6.9: Instruction length selection.

Register Addressing: The *Addr* field supplies the address for the register file and the select lines for the external input multiplexer. The sharing between these two functions is logical because only one can be used to supply $R1$ at a time. The disadvantage is that It does add an additional bottleneck, by preventing simultaneous reading of inputs and writing of results. This was a tradeoff to reduce logic size in return for a slight increase in exectuion time. The assembler (§ 6.5.3) automatically inserts delays to avoid the resource conflict.

This field is placed in the first instruction word for both frequency of use and timing reasons. For timing, the address is needed early so that it can be synchronised with a flip-flop before memory decoding takes place. Without the synchronising flip-flop, the propagation delays become too large.

ALU Commands: $C1...C4$ contain the command signals to be sent to the ALU, as described in Section 6.4.3. These bit patterns are sent directly to the ALU at the appropriate time. $C1$, $C3$ and $C4$ are applied immediatley to the ALU after one latching stage. An extra delay is inserted for the $C2$ command as this is used one cycle later. The timing constraints would have allowed $C2$ to appear in the second instruction word, but the frequency of its use make the first word a more appropiate location.

Apart from delaying the $C2$ command, the piplining of instructions is left to the instruction design stage. This means that while $C1$ and $C4$ commands are in the same instruction, they will refer to different calculated values in the ALU. It is most likely that the $C4$ command will refer to the results of the previous instruction. The assembly language and compiler developed in Section 6.5.3 helps in this pipeline design.

Branch Address The second word is included when the instruction requires branching (jumping) to another microcode location, or if an "external command" is required. The "BrAddress" field specifies the destination location of a branch if a branch is to occur. If a branch does not occur, based on the branch condition field, this field is ignored.

Branch Condition The branching logic makes use of two fields in the second instruction word. The "BrCondition" field specifies which condition is to be used for the branch decisions. The two basic conditions are "branch never" and "branch always", which result in sequential execution and unconditional branching respectively. The "branch never" condition is coded as 000 to make this the default when the second instruction word is not supplied. If this field is 000, the Branch Address field is ignored. The other conditions are similar in nature to "Branch if the ALU result is negative". Table 6.10 shows the branching conditions defined for this application.

Value	Symbol	Action
000	Br0	Don't jump
001	BrSign	Jump if result negative
010	BOverflow	Jump if the was a numerical overflow
011	BrCountNZ	Jump if internal counter not at zero
100	BrLeNZ	Jump if inductance estimator counter not zero
101	BRT0Wait	Jump if sample at T0 not available
110	BRTxWait	Jump if no samples are available
111	Br1	Always jump

Table 6.10: Defined Branch Conditions

Note that a number of these conditions are not directly related to the ALU or the sequencer. Instead they have been defined to suit the controller hardware. In this sense, the sequencer is much more tightly integrated into the application design than it would be in a microprocessor based solution.

Command The external commands are used primarily to signal the presence of output values from the ALU. For example, there are three external commands connected to the PWM module to indicate when the switching times are available on the ALU output. The five bits allocated to the "Command" field are decoded to allow up to 32 external commands to be generated. Although only one of these commands may be asserted in a single instruction, this is not a limitation as the algorithm requires only infrequent use of commands.

The command 00000 is defined as a null command, performing no action. This is compatible with single-word instructions, and allows an instruction to have a branch without a command.

Address Generation

In the first stage of instruction decoding, the value of the address for the instuction in ROM must be calculated. Once this is done, the value of the first instruction word is available on the next clock cycle. This is denoted Stage 1 in Figure 6.12. During this stage, the "$\overline{1}/2$ Words" bit is interrogated to determine whether the address should be incremented or not.

In the instruction decoding Stage 2, the address for the following instruction is determined. This involves selecting between an address increment and a new branch address from the address multiplexer. At this point, the result from the previous calculation is available in $R2$ to use to evaluate branching condition. Note that this means that branches typically act on results from the previous instruction, and not the operations in the first word of the current instruction.

6.5.3 Microcode Assembler

A simple microcode assembly language and assembler were developed to simplify the writing of the microcode. This was not an essential part of the design, but more of a refinement. Both development and debugging time are reduced considerably through using the more intuitive representation allowed with an assembler. In addition it greatly simplifies the testing of modifications of some aspects of the algorithm, for example, the inductance estimator.

It may be noted that including this step brings the system design closer to that of a microcontroller-and-software approach. Once the chip can be programmed in an assembly language, it presents many of the attributes of a microprocessor. However, there are still significant differences between this design and that of a microprocessor.

Principally, the computational structure and peripheral integration is still customised to the particular task. It should also be emphasised that a number of the controller tasks are implemented outside this central ALU structure.

The assembly language itself has been modelled to some extent off the C programming language, but with a very limited syntax. An example segment of the source code is shown in Figure 6.13:

```
//allocate an entry in the register file
int Counter=2;
//pointer to external input
constant IB=6;

  a=Counter;
  a=a*two+Counter; // a=3*counter
  a=a-IB;          // subtract from IB
  a=-a;            // load a back via R1
```

Figure 6.13: Example source code for the assembler

This example multiplies the contents of a register in the file by three and then subtracts it from the latest measurement of the B phase current.

The Grammar

The assembly code language may be specified by a context-free grammar. This is a useful method of specifying the full syntax of the language, and aids in the compiler design[1]. In this case, the grammar is also strongly related to the command sets controlling the ALU.

The complete grammar is shown in Figure 6.14, but the following sections describe each expression. This grammar type is known as the Backus-Naur From (BNF), from the work of Backus[2] and Naur[36] in the 1950's and 60's.

R1 Expressions:

$$< R1Expr > \rightarrow < MemAddress > \mid < Ext > \mid < R4Expr > \quad (6.17)$$

The $< R1Expr >$ appears up to once in each generated instruction. Its role is to define the bit pattern for the $C1$ command word. The three forms of this expression match the three out of the four valid bit patterns for the ALU $C1$ input. The fourth (memory write), is set by an alternate instruction.

The $< MemAddress >$ and $< Ext >$ fields are matched from a symbol table which is constructed from memory allocation and external declaration statements (see below). When these fields appear, the memory address in the first word is set to the associated value, as derived from the symbol table. When an $< R4Expr >$ is supplied, the address field is not specified.

$$
\begin{array}{lcl}
< R1Expr > & \rightarrow & < MemAddress > \quad | \quad < Ext > \quad | \quad < R4Expr > \\
< R3Expr > & \rightarrow & a \quad | \quad a * two \quad | \quad a/two \quad | \quad a >> sixteen \\
< R4Expr > & \rightarrow & al \quad | \quad ah
\end{array}
$$

$$
\begin{array}{lcl}
< R2Expr > & \rightarrow & < R3Expr > \\
< R2Expr > & \rightarrow & < R3Expr > + < R1Expr > \\
< R2Expr > & \rightarrow & < R3Expr > - < R1Expr > \\
< R2Expr > & \rightarrow & < R1Expr > \\
< R2Expr > & \rightarrow & - < R1Expr > \\
< R2Expr > & \rightarrow & zero \\
< R2Expr > & \rightarrow & < R3Expr > * < R1Expr > *two \\
< R2Expr > & \rightarrow & < R3Expr > / < R1Expr > /two \\
< R2Expr > & \rightarrow & abs(< R1Expr >)
\end{array}
$$

$$
\begin{array}{lcl}
< Statement > & \rightarrow & int < MemAddress > \\
< Statement > & \rightarrow & int < MemAddress >=< Number > \\
< Statement > & \rightarrow & int < MemAddress >= - < Number > \\
< Statement > & \rightarrow & constant < Ext >=< Number > \\
< Statement > & \rightarrow & a =< R2Expr > \\
< Statement > & \rightarrow & out(< R4Expr >) \\
< Statement > & \rightarrow & < MemAddress >=< R4Expr > \\
< Statement > & \rightarrow & < Ext > \\
< Statement > & \rightarrow & if < Ext > goto < Label > \\
< Statement > & \rightarrow & LABEL < Label >
\end{array}
$$

Figure 6.14: The Complete Assembly Context-Free Grammar.

R3 Expressions:

$$< R3Expr > \rightarrow a \quad | \quad a * two \quad | \quad a/two \quad | \quad a >> sixteen \tag{6.18}$$

The $< R3Expr >$ defines the value of the $C3$ command word on the ALU. This controls the shift register, which is placed between $R2$ and the $R3$. In this case, the variable a appears as a reference to the contents of the $R2$ register. This is done because at the conclusion of each instruction, generally $R2$ holds the result of the calculation. It is then convenient to think of $R2$ as an accumulator, which takes the results, and can be used as an operand via the $< R3Expr >$.

The four possible compositions of $< R3Expr >$ correspond to the four modes of operation that are possible with the ALU shift register. The three standard modes are:

1. Direct pass-through: a
2. Shift Left: $a * two$
3. Shift Right: a/two

Note that alphabetical representations are used for each symbol in order to simplify the

parsing. It also emphasises that arbitrary integers cannot be accepted as operands. The fourth shift register operation is provided to enable loading of the low-order sixteen bits of $R2$ and $R3$. This is principally required for loading the dividend of a division calculation.

R4 Expressions:

$$< R4Expr > \quad \rightarrow \quad al \quad | \quad ah \tag{6.19}$$

The $C4$ ALU command word controls the output multiplexer from $R2$. The $< R4Expr >$ specifies which half of the $R2$ accumulator should be used for output. Typically ah is used.

Addition and Subtraction:

$$\begin{aligned} < R2Expr > &\rightarrow < R3Expr > \\ < R2Expr > &\rightarrow < R3Expr > + < R1Expr > \\ < R2Expr > &\rightarrow < R3Expr > - < R1Expr > \\ < R2Expr > &\rightarrow < R1Expr > \\ < R2Expr > &\rightarrow - < R1Expr > \\ < R2Expr > &\rightarrow zero \end{aligned} \tag{6.20}$$

The compositions in (6.20) select the basic operations of the adder unit in the ALU. Because the prior $R1$ and $R3$ calculations are included in the same instruction, expressions such as $a * two - ID0$, are possible, where $ID0$ is a declared register.

Absolute Value:

$$< R2Expr > \quad \rightarrow \quad abs(< R1Expr >) \tag{6.21}$$

The absolute value instruction was added into the ALU specifically because it appeared a number of times in the current control algorithm.

Multiplication and Division:

$$\begin{aligned} < R2Expr > &\rightarrow < R3Expr > * < R1Expr > *two \\ < R2Expr > &\rightarrow < R3Expr > / < R1Expr > /two \end{aligned} \tag{6.22}$$

While they share a similar syntax to the other $R2$ expressions, multiplication and division incur a much greater implementation complexity. In each case, the assembler generates a number of instructions including a loop. See Section 6.6.2 for details of the implementation.

Memory Allocation:

$$\begin{aligned} < Statement > &\rightarrow int < MemAddress > \\ < Statement > &\rightarrow int < MemAddress >=< Number > \\ < Statement > &\rightarrow int < MemAddress >= - < Number > \end{aligned} \tag{6.23}$$

These statement compositions are used to declare memory locations. In implementing the grammar, the $< MemAddress >$ is actually identified as an unknown string, as is the number. These statements do not generate any code, but rather add to the symbol table for future use in $R1$ expressions. While an address is generated, and later used in the microcode, the user is insulated from this at an assembly level. These memory registers are always referred to by the symbol as defined here.

The optional initialiser is used to build up a table of initial values to be programmed into the EPLD. This is particularly useful for defining constants, such as $\sqrt{3}$, for use in calculations.

Constants:

$$< Statement > \quad \rightarrow \quad constant < Ext >=< Number > \tag{6.24}$$

These constants are not constants to be used in calculations, but rather as symbolic shortcuts. It is similar in concept to the C language $\#define$ pre-processor directive. The basic uses of these definitions are for:

1. External source addresses. These control the select lines of the external input multiplexer when reading from external hardware registers (for example, the sampling hardware).

2. Branch conditions. While most of the branch conditions are specified by the ALU design, these are declared symbolically in the assembler source.

3. Output commands. As the output equivalent for the external source address, these control the output command decoder.

Note that while these are simply symbolic defines, integer numbers cannot be used in their place in assembly statements. This restriction was imposed to simplify the grammar.

Basic Instruction:

$$< Statement > \quad \rightarrow \quad a =< R2Expr > \tag{6.25}$$

This is the form of the basic instruction to be converted to microcode. It often takes the form of an updating equation on the a register. For example, $a = a * two + MR_{R1}$.

Output:

$$< Statement > \quad \rightarrow \quad out(< R4Expr >) \tag{6.26}$$

The *out* instruction simply specifies which half of $R2$ is to appear on the output of the accumulator. It is usually used in conjunction with a command to indicate to external hardware to read the value.

Memory Write:

$$< Statement > \quad \rightarrow \quad < MemAddress >=< R4Expr > \tag{6.27}$$

The register file can be written to by assigning al or ah to the appropriate symbolic name. Note that for a given output instruction, this can conflict with other operations that either use the ALU output or the 5-bit register/external address field.

Command:

$$< Statement > \quad \rightarrow \quad < Ext > \tag{6.28}$$

An external command is specified by simply using it as a statement.

Branching:

$$\begin{aligned} < Statement > \quad &\rightarrow \quad if < Ext > goto < Label > \\ < Statement > \quad &\rightarrow \quad LABEL < Label > \end{aligned} \tag{6.29}$$

Branching is executed through the $LABEL$ and if statements. $LABEL$ is used to declare a label for the target of a jump. While a number of statements can usually be placed into a single microcode instruction, a $LABEL$ statement flushes the pipeline and forces a new instruction to be started. Unnecessary labels should be avoided as they are likely to incur a performance and size penalty.

The if statement inserts a branch condition and target address into the microcode instruction. The branch condition must be defined as a constant, and the target as a label in the assembly source.

The assembler uses only a single pass, so branch targets may not necessarily be defined at the time of use. If they are not defined, the instruction address is added to a symbol table for later correction. As each new label is defined, the table is scanned to correct any existing references to that label.

As already mentioned, the machine design incorporates piplining concepts. This means that multiple source instructions may be decoded and executed concurrently. While this improves performance, it creates unexpected relationships between instructions, and these are called hazards. The hazard of relevance to this design is the branch hazard.

A branch hazard arises when a branch instruction is exectued concurrently with the instruction following it. This means that even if the branch is taken, the following instruction will be at least partially executed. In order to maximise performance, pipeline branching hazard resolution is performed in the assembly source code. If protection were instead added to the assembler or hardware, very short loops could be forced to require double the existing execution time.

The pipelining causes difficulty for branching instructions because the conditions are based on the results from the previous instruction, but the branch is not taken until the

next instruction. Consider the instruction sequence in Table 6.11.

Instruction	Source code sequence
1	a = Value1;
2	if BrSign goto BranchTarget; a = Value2;
3	a = Value3

Table 6.11: Example source code.

In this case, Instruction 2 will be completely executed regardless of the result of the branch. This means that the $a = Value2$ assignment occurs even if the jump is taken. This anomaly could be resolved by reversing the order of the if statement and the assignment in the instruction source, as shown in Table 6.12. In that case, the behaviour would be the same, but the source code would better represent it.

Instruction	Source code sequence
1	a = Value1;
2	a = Value2; if BrSign goto BranchTarget;
3	a = Value3

Table 6.12: Alternate source code ordering (not used).

The code in Table 6.12 has a potentially more serious problem. In this case the confusion would occur in the specification of the branch condition. The branch condition is evaluated based on the results of the previous statement (statement 1). In this case, the branch is taken based on the value of $Value1$, not the $Value2$ which would be expected from the source code.

The instruction format necessitates that either one of these problems be present in the assembly specification, or that unnecessary performance limitations be imposed. As a result, it was chosen that the assembly format shown in Table 6.11 be used. This behaviour is more predictable because it is a common product of branch hazards in pipelined machines.

In summary, in the final design, if an $R2$ assignment statement appears immediately after a branch statement, the $R2$ statement will be executed regardless of the outcome of the branch. This includes the case of unconditional branches.

Figure 6.15 shows an example of a loop that consists of only a single instruction. In this case, it appears that the manipulation of the a register is outside the loop, but it is executed on each iteration based on the behaviour described above.

```
LABEL LCLoopD;              //multiply by 2 until count is zero
  COM_LeCDown;              //count down
  ifBrLeNZ goto LCLoopD;
  a=a*two;                  //executed inside the loop
```

Figure 6.15: A single-instruction loop

Statement Ordering

The discussion of the branch statement above introduces the concept of how the assembly statements should be ordered to represent an instruction. As multiple assembly statements may be represented in a single microcode instruction, the assembler has the task of gathering compatible statements together. This is done to minimise the number of final microcode instructions generated, subject to the constraint that the resulting behaviour is predictable and logical.

To remove ambiguity and misleading references, a standard order of statements is imposed for each output instruction. These stages are shown in Table 6.13. This shows

Stage	Description
1	Label
2	Memory or External Write
3	Commands
4	Branch
5	R2 assignment

Table 6.13: Statement encoding stages for microded instructions.

that one microcode instruction may be formed out of up to five source instructions, provided they occur in the order listed and don't conflict in resource usage.

For a single instruction, each additional statement must progress through the stages in a non-decreasing fashion. In all cases except for the label, only one statement from each stage may appear. Whenever a statement is encountered that is associated with an earlier stage than the previous one, a new output instruction is started.

Sometimes even when the specified statement order is observed it may need to be split based on resource conflicts. In particular, the register memory addressing field is used for a number of purposes. A split is required if there is memory accesses in both stages 2 and 5. In this case the split is made at the conclusion of stage 3 to ensure predictability of the stage 5 execution in the presence of a branch.

Figure 6.16 shows an example of the partitioning of a block of code based on these conditions. In this case, a total of five instructions would be required to implement the supplied statements.

```
//Some arbitrary example code
  a=ID0;
                //break on [stage 5 repeat]
  a=a*two-ID4;
                //break on [move back to stage 1]
LABEL TheLoop;
  COM_InLoop;
  a=a+IncValue;
                //break on [move back to stage 2]
  Count = a;
                //break on [memory address conflict between access
                //  to Count and DecValue]
  if BrSign goto TheLoop;
  a=a-DecValue;
```

Figure 6.16: Code sample demonstrating instruction partitioning.

Compiler Design

The art of Compiler design was not part of the core project intention, so the design emphasis was more on rapid coding and optimal output, rather than a necessarily efficient design. A basic recursive-descent parser was implemented using the grammer specified in Figure 6.14. The full C++ source listing is included in the appendix.

The compiler is a single-pass design, that iterates through the source code matching the statements to the supplied grammar. The preprocessing stage strips out the C++ style comments and splits the input into semi-colon delimited statements.

Each statement is tokenised and represented as a linked list structure of symbols in the *GetNextExpression*() function. This function is called recursively to build up the complete list from the input statement. The input is partitioned, and the symbol type determined in the *GetNextSymbol*() function. This function compares each token against an internal symbol table to determine and record the symbol type. This type is later used in matching to the grammar compositions.

Once the input is tokenised, the expressions from the grammar are recursively expanded and matched to the input expression. When a match is found, the instruction fields are set by the code generation component. The code generation section handles the collating of statements into instructions. An example of the recursive grammar expansion is shown in Figure 6.17.

For each combination of compositions, a list structure is formed and compared to the token list from the tokeniser. A left-to-right comparison is formed, and the search is pruned when a partial expansion shows that a match is not possible. This optimisation is made even though parsing speed is not an important factor due to the relatively short source files.

Additional facilities are provided to maintain addresses both in the register memory,

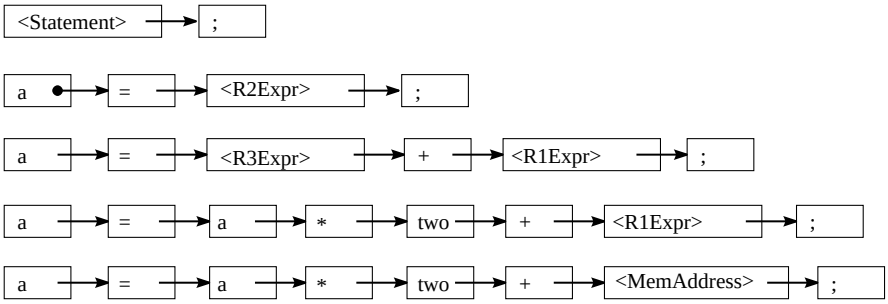


Figure 6.17: Recursive grammer token expansion example.

and program branching space. The final output is then written in a memory initialisation file format for reading by the EPLD development software.

6.6 Additional Calculation Hardware

Apart from the ALU itself and the sequencer, a number of other hardware elements are closely integrated into the computation hardware.

6.6.1 Loop Counter

A simple loop counter has been closely coupled with the sequencer. This is principally to signal the end of loops for multiplication and division. The interface to the sequencer is via the command interface, and uses boolean connections, as shown in Figure 6.18.

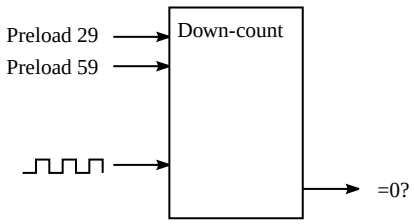


Figure 6.18: The loop counter.

At the start of the loop, the command is issued to preset the counter, and a branch instruction is used to execute the loop until the appropriate time has elapsed.

6.6.2 Multiplication

The ALU was designed specifically to accept shift-add style multiplication and division. In both cases, signed operations were chosen because of the type of calculations needed in the controller. Typically the currents or voltages being manipulated may be either positive or negative in sign.

Signed multiplication may be achieved using Booth's algorithm[18]. Although this algorithm was originally designed for performance improvements over a simple shift-add, its principal advantage is that it works on two's-complement numbers without any adjustment overhead. To use Booth's algorithm, a small addition has to be made to the ALU. This is shown in Figure 6.19. An extra bit is added to the right of the shift register.

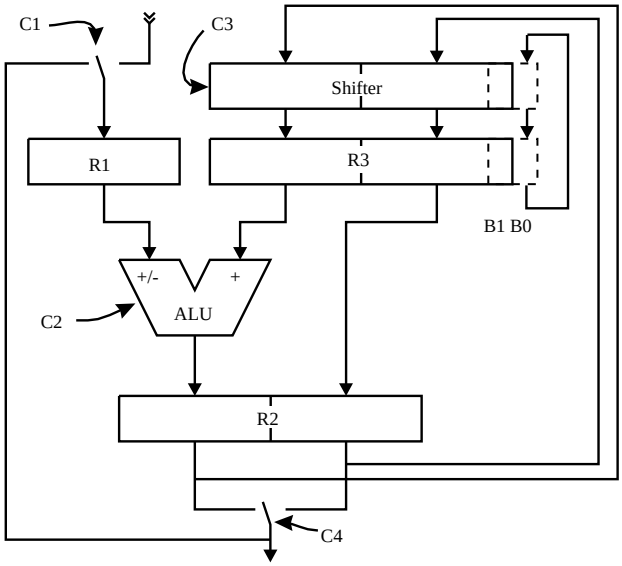


Figure 6.19: The ALU with support for Booth's algorithm.

This additional bit, and the conventional least significant bit are used in the algorithm.

The multiplication starts by loading the multiplier into the least significant bits of R3. This also clears the additional Booth bit, B0. At the same time, the multiplicand is loaded into R1, and the loop counter preset to 29. Although the process takes 32 clock cycles, the pipelining delay means the count value used is smaller.

Following this, a loop is entered, which consists of two clock cycles per iteration. The first cycle involves selecting the correct operation for the add/subtract unit. The operation to perform is chosen from the two Booth bits, B1 and B0. This choice is shown in Table 6.14.

B1 B0	Operation
0 0	R2=R3
0 1	R2=R3+R1
1 0	R2=R3-R1
1 1	R2=R3

Table 6.14: Booth algorithm operation.

In the second clock cycle, the multiplicand is reloaded into R1, and the shifter is set to shift right by one bit. The R2/R3 register set contains portions of both the product

and the multplier. The product is formed from the left, and the multiplier is shifted out to the right. At the completion of the loop, the product fills the R2 register.

6.6.3 Division

Division is a more complex operation than multiplication. Fortunately it is not required often in the control algorithm. Using the described hardware design, 64 clock cycles are needed to perform the division operation. The design of the divider is essentially a standard shift and subtract method, but with modifications to allow use with signed numbers. The main addition is a signal which compares the signs of R1 and R3. This is essentially a single XNOR gate,

$$S_{13} = R1_{MSB} \odot R3_{MSB} \tag{6.30}$$

At the start of the process, the 32-bit dividend is loaded into register R3. At the same time, register R1 is filled with the divisor. After this, a loop is entered to determine each bit of the quotient. Each iteration of the loop comprises 4 clock cycles. The details of these are:

1. If $S_{13} == 1$ (the signs of R1 and R3 are the same),
   R2=R3-R1.
   Otherwise,
   R2=R3+R1.
2. R1=Divisor, R3=R2
3. If the add/sub caused a change in sign of R2,
   If $S_{13} == 1$
   R2=R3-R1.
   Otherwise,
   R2=R3+R1. (this reverses the previous subtraction)
   Otherwise
   R2=R3
4. R1=Divisor, R3=R2 shifted left by 1 bit. The value shifted in is S_{13} from the result of step 2.

The result of this operation is a 32-bit quotient in ones-complement format. For negative numbers, this differs by one from the usual twos-complement notation, but this is not significant for the division operations to be performed for the current controller.

6.6.4 Inductance Estimator

Due to the precision and special operations required, the inductance estimator has additional hardware support outside the ALU. In order to accommodate variations in possible

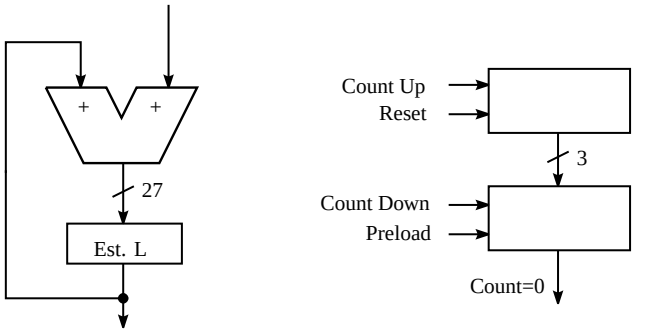
inductance values and switching frequencies, a floating point scheme was adopted. The precision needs to be higher than for normal calculations for two reasons

1. The per-unit inductance can vary from machine to machine. By considering machines of various ratings, it was estimated that a range spanning a factor of 100 is adequate. This means an additional seven bits are needed to represent the variation in the value.

2. The inductance value needs to be averaged to reduce the amount of noise on the measurement. For this case, a simple IIR filter will be considered, rather than a more sophisticated estimator, such as recursive least-squares. Adequate representation may be achieved by including an additional eight bits of precision.

Adding the 12-bits of basic precision to the above requirements results in a word-width for the inductance of 27-bits. In the implementation, a 27 bit register is used to store the value and a 3 bit counter for an exponent. The exponent is useful to aid in matching to the fixed-point ranges of the controller.

The additional hardware is shown in Figure 6.20. The high 16 bits and low 11 bits of the 27-bit register may be accessed separately, and loaded into the ALU.



(a) Mantissa Register (b) Exponent Counters

Figure 6.20: Inductance estimator hardware.

The first step in updating the inductance average is to find the difference between the newest estimate and the existing average. The estimate itself is calculated within the ALU, and the difference taken from the existing estimate. The external hardware then performs the averaging operation. The estimation is implemented using the following code fragment:

```
//calc the L/3 estimate
// = min(t1,t2)*V/2/3/abs(delta i1-delta i2)
a=ID4;                                                    135
a=a*two−ID0L;
a=a−ID2;
```

```
  a=abs(ah);
  MR_R1=ah;              //store abs \delta i_1-\delta i_2 in MR_R1
  a=a*two+MR_R1;                                                    140
  MR_R1=ah;              //3*abs(...)

  a=T1;                  //get min (T1,T2)
  a=a-T2;
  if BrSign goto HaveTx;  a=T1;                                     145
  a=T2;
LABEL HaveTx;
  a=a>>sixteen*ES_DCV*two;     //get 2TxV
  a=a/two;                     //a=TxV
  a=a-MR_R1;                   //check if the result is usable      150
  a=a+MR_R1;
  a=a/MR_R1/two;
  a=al;                  //now have TxV/2abs(...) = L/3V
                         //sub off existing estimate to give
  a=a-ES_LeH;            //amount to add to inductance estimate      155
  out(ah);
  COM_LeUpdate;          //update the inductance
```

The overflow detection code has been removed from this code for clarity (see the appendix for the complete version). Once the estimate is calculated, the high order word of the existing estimate is read, and subtracted off the new value. The command on the final line then causes the hardware to update the average based on the difference calculated.

The next task in the algorithm is to find the quotient $\frac{L}{6V}$. The main complication is that the fixed point result may be too large to fit in the 16 bits available. To overcome this, the inductance estimate is first shifted right until it is smaller than V. Once this occurs, it is safe to divide. The inductance counter keeps track of the number of shifts necessary.

```
  //— Find L/6V —
  a=ES_LeL;                    //load L/3
  a=a>>sixteen+ES_LeH;                                              195
LABEL LVLoop;                  //make sure division will not overflow
  a=a-ES_DCV;                  //div 2 until L/3<V
  if BrSign goto LVLoopEnd;
  a=a+ES_DCV;
  COM_LeCUp;                   //count num of divs                  200
  if Br1 goto LVLoop;
  a=a/two;
LABEL LVLoopEnd;
  a=a/ES_DCV/two;              //have L/6V
  L_EST=al;                                                         205
```

The $\frac{L}{6V}$ quotient is later multiplied with a summation of currents. The result of the multiply fills 32 bits, and so the result of the product is shifted to compensate for the earlier shift. For this operation, the second inductance loop counter is used to count the correct number of left shifts.

```
COM_LeCPre;                 //preload inductance counter
a=a>>sixteen*L_EST*two;     //multiply by L/3V                    260
COM_LeCDown;                //count down after branch
if BrLeNZ goto LCLoopD;     //do shifting if necessary
if Br1 goto LCLoopEndD;
```

The shifted result may then be used for further calculation.

6.7 The Current Controller

The current controller algorithm is closely based on the software version described in the previous chapter. As the units and scaling of the variables were designed to be suitable for a fixed-point implementation, these remain unchanged. The main changes have been to suit the specific hardware requirements and instruction set of the microcode machine.

6.7.1 Sample Acquisition

The control process starts with the acquisition of the current samples on the three phases. This first involves waiting until the samples are ready. This is done by polling the BrT0Wait condition. The following code block marks the start of the loop:

```
//————————————————————————
// WAIT FOR T=0
// Read In Currents at T=0                                          110
// and do 3-phase to 2-phase converstion of currents
//————————————————————————
LABEL WaitT0;
  if BrT0Wait goto WaitT0;
                                                                   115
```

Following this, the currents are read in, and the 2-phase quantities calculated. This involves simple addition and subtraction operations. The external multiplexer read is a fast process, so the currents are actually read multiple times where required.

```
//IQ0=ES_IB-ES_IC
a=ES_IB;
a=a−ES_IC;                                                         120
IQ0=ah;
```

```
//ID0=2*ES_IA-ES_IB-ES_IC;
a=ZERO;
a=a>>sixteen+ES_IA;
a=a*two−ES_IB;
a=a−ES_IC;
ID0=ah;
COM_GotCurrents;
```

The final command signals the hardware that the values of current need no longer be stored in the output latches. The assignment of zero to the accumulator in line 124 is simply taking advantage of otherwise idle time. The aim here is to clear the lower 16 bits of the accumulator for later use. The same basic procedure occurs to read the currents at $t = 0.25$ and $t = 0.5$.

6.7.2 Updating Equation

During the first half of the control cycle, the inductance estimate is calculated, as described in Section 6.6.4. In the second half, the current control algorithm is calculated. The actual control updating equation is a relatively small portion of the code. The specific operations vary according to the specific control algorithm, such as midpoint and endpoint control. For the endpoint control on the d-axis it is,

$$alphaD = alphaD + \frac{L}{6V}(IDR - 4ID2 + 3ID0). \tag{6.31}$$

The summation of currents may be implemented with the following code fragment:

```
a=IDR;
a=a+ID0;
a=a/two−ID2;
a=a+ID0;
a=a−ID2;

a=a>>sixteen*L_EST*two;        //multiply by L/3V
```

Following this, the result is shifted to compensate for the prior scaling of the inductance estimate, as described in the previous section. The product is then added to the previous duty cycle to obtain the updated duty cycle.

6.7.3 Sector Determination

After calculating the q-axis value, the calculations for the space vector PWM are performed. The first operation is to determine the sector associated with the voltage vector. This is made a relatively simple operation through an appropriate method of sector numbering. The method used in the software controller is replicated in this application.

This sector pattern is shown again in Figure 6.21. The advantage of this allocation

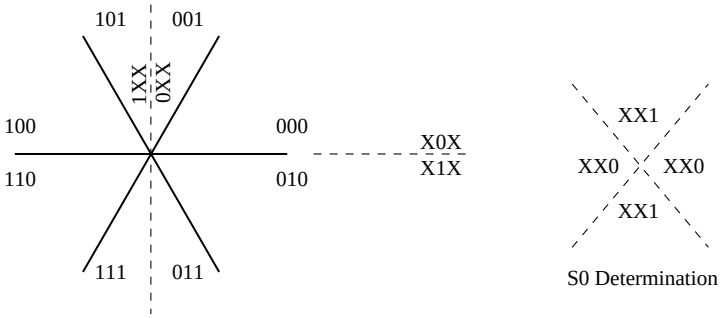


Figure 6.21: Binary sector allocation.

is that both the PWM hardware, and the calculation hardware are simplified.

Three additional flip-flops, and associated command outputs, are used to store the sector value, as shown in Figure 6.22. Each bit is loaded directly from the most significant bit of the accumulator after the appropriate value is loaded. Using the controller

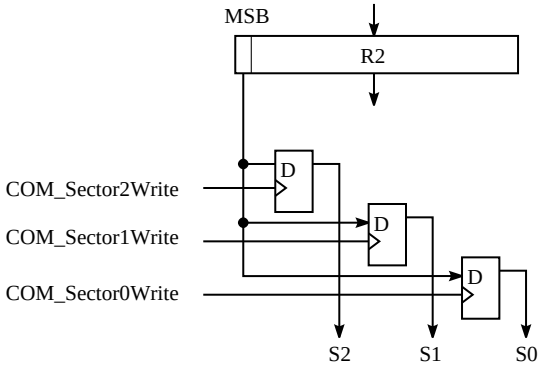


Figure 6.22: Sector determination hardware.

variables, each bit is calculated as:

- B2=MSB[AlphaD]
- B1=MSB[AlphaQ]
- B0=MSB[abs(AlphaD)-abs(AlphaQ)]

At the start of the next PWM interval, these values are supplied to the PWM hardware.

6.7.4 Switching Times

The remaining task is to determine the switching times, and then perform clipping if the calculated values are not feasible. The algorithm is again the same as that implemented in the software version. Some example code from the sector 2 and 5 section is shown below. It is entered with *AlphaD* in the accumulator.

```
//THE MIDDLE SECTORS 2,5
//- T1=-aD+abs(aQ) T2=aD+abs(aQ)                                    375
LABEL Sect25;
  a=a+ABSalphaQ;
  T2=ah;
  a=ABSalphaQ;
  a=a−alphaD;                                                      380
  T1=ah;
```

As the absolute values were already calculated, this operation is quite simple. The switching times in the other sectors are calculated in a similar manner. $T0$ is then calculated from the relationship,

$$T0 = \frac{T}{2} - T1 - T2. \tag{6.32}$$

When $T0$ is found to be negative, the duty cycle values must be clipped. This involves a scaling of both the alphas by the ratio:

$$\gamma = \frac{T}{2(T1 + T2)}. \tag{6.33}$$

Following this, the switching times are written to the PWM hardware, and the entire algorithm is repeated.

6.8 Controller Performance

The performace of the current control algorithm was demonstrated in the previous chapter with the software version. This section includes only a basic performance evaluation. As the improved back-emf prediction algorithm is not implemented, the performace of the hardware version is inferior to the software one. The inductance estimation is also less effective, as only the quarter-cycle method has been implemented.

However, despite these limitations, the hardware current controller does perform well. Figure 6.23 shows the measured current and torque for a machine with an inertial load. The upper trace on the CRO display is the signal from a current probe attached to one phase of the machine. The lower trace is the output of a torque transducer attached to the output shaft. The peak current shown is approximately 7 amps, and the torque output is about 5Nm in each direction. The first part of the plot shows the machine accelerating under a negative torque setpoint. The sharp change in measured current indicates the reversal in the torque setpoint. Unfortunately, the torque measurement system has a bandwidth limitation of 10Hz so this measurement cannot reflect the true torque transient.

Figure 6.24 was obtained from tests with a locked rotor. Figure 6.24 shows square-wave torque output and the corresponding machine current. The step change in torque

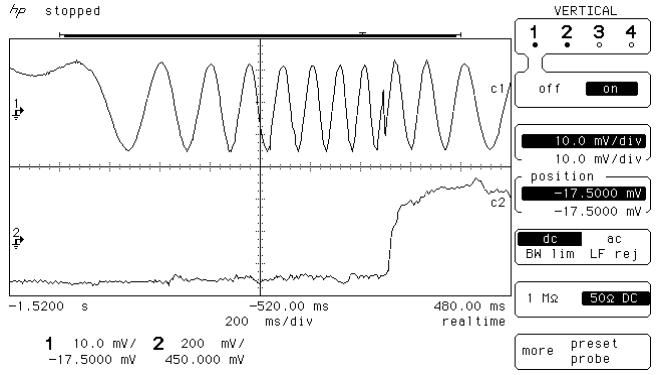


Figure 6.23: Phase current and measured torque for a demanded torque transient with a free rotor.

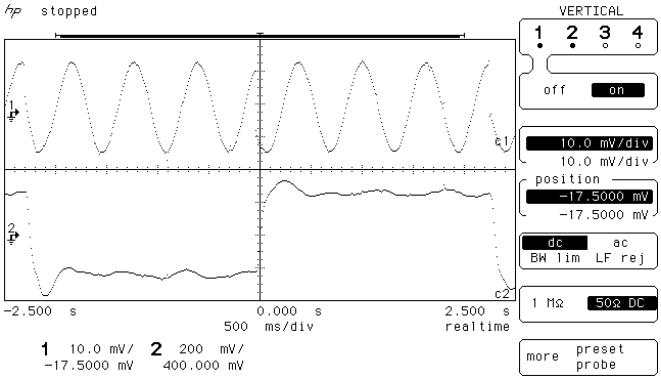


Figure 6.24: Phase current and measured torque for a demanded torque transient with a locked rotor

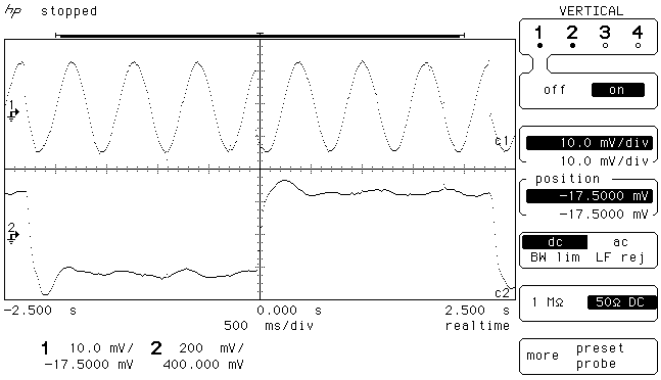


Figure 6.25: Response of the current controller to a step change in setpoint. Step is from -5A to +5A on the d-axis.

requires a step change in current. The required sharp change is visible just before the $t = 0$ axis on channel 1. The fast nature of the transient indicates the high bandwidth of the current controller in tracking the setpoint. The measured torque response is

somewhat slower. As mentioned above, the transient on the torque measurement is governed by the filtering in the transducer electronics, not the actual torque produced by the machine.

Figure 6.25 shows the response of the current controller to a step change in desired current. This shows a fast transient response and small steady-state error. With an appropriate outer loop torque controller, the electromagnetic torque bandwith will follow the bandwidth of the current, resulting in good torque performance.

The final design utilised 42% of the logic cells available in the FLEX10K50-3 device. The development tools also rated it for a clock frequency of up to 18Mhz. This is close to the original design goal of 20MHz. Faster grade parts are now cheaply available, which would allow the design to exceed the desired control rate.

6.9 Conclusions

This chapter has presented a digital hardware implementation of a current controller. It illustrates that the current control algorithm described is sufficiently simple to be implemented without the use of a microprocessor or DSP. Instead, the complete controller, together with the sampling and PWM logic, may be constructed in a readily-available EPLD device.

A number of design approaches were considered, and it was found that a microcoded state machine was the most suitable method for designing the sequencing logic. This method offers some of the advantages of a software solution, yet it still allows direct hardware integration and parallelism. To further aid the development processes, a compiler was presented that allowed the microcode instructions to be entered in a C-like description language.

Finally results have been presented that show the current controller operating a real induction machine. A low-powered external microprocessor was used for the outer-loop control, but as future work, this could also be integrated into the same hardware framework.

Chapter 7

Conclusions & Further Work

7.1 Conclusions

This thesis has demonstrated the development of a new induction machine current controller. Through appropriate choices in the machine model and controller algorithm, this controller was suitable for a digital implementation in direct hardware. Coupled with the estimation methods that have been developed, this controller may be developed as a single-chip solution that requires no knowledge of the machine parameters.

Digital controllers are inherently sampled-data systems, and it was found that two samples per PWM switching interval were adequate for effective control. This allows the controller to use models based on average voltages across an interval, instead of more complex instantaneous ones. Furthermore, it was demonstrated that the simple back-emf plus leakage inductance model was sufficient to characterise the machine's electrical behaviour.

There are a number of advantages and disadvantages of a complete hardware implementation of a controller over a hardware/software approach. The main advantages are in the price/performance ratio in production quantities, and in the level of integration. The main disadvantage is that the development process is more difficult, and many concepts need to be re-invented. However, with time, the two approaches are drawing closer. Microcontrollers are offering greater on-chip support for motor control applications, and the development facilities are improving for logic devices.

This thesis has demonstrated a controller that is suitable for both software and hardware implementations. In both cases, the necessary hardware devices are readily available. The experimental results show that the controller performs well at tracking the set-point currents without the use of supplied parameters. This provides a good basis for a torque controller, as the fastest loop in the control strategy is robust to variations in machine parameters.

7.2 Suggestions for Further Work

There are a number of areas in which this work may be extended. Some of these are:

- The newest inductance estimation and back-emf prediction schemes were not implemented in the hardware version. The addition of these elements would raise the performance of the hardware version to that of the software one.
- The hardware current controller design architecture has the capacity for further integration. A field-oriented torque controller could be included into the single-chip hardware design.
- The control of different types of power converters could be investigated. For example, the model used here is similar to that of active rectifiers. There are a large number of possibilities, including a single chip to control a combined active rectifier and induction machine drive.
- Dead time effects at very low currents is still a problem, leading to distortion of the current waveforms. New model independent techniques to alleviate these effects could be researched.

Appendix A

DSP Implementation Code

A.1 Introduction

This appendix includes schematics and listings that are relevant to the Dual TMS320C31 DSP implementation of the current controller. Due to space constraints, the hardware schematics are not included, but instead only the software and firmware described in this thesis. The first section consists of listings of the current controller software. Following this, the designs for the two Altera devices are included.

A.2 Software Description

The current controller was implemented on a dual-processor system, and so there are two separate executables. The first is for the processor executing the current controller. This uses the files:

- `cc1.c`
- `iface.c`
- `p_iface.c`

The main current controller file is `cc1.c`. The other files are concerned with the monitor interface.

The other processor executes the monitor. It shares common source code from `iface.c`, and the files are:

- `c_iface.c`
- `iface.c`
- `m_serial.c`

A.2.1 cc1.c

```
/*****************************************************************
cc1.c

The main current control program.
*****************************************************************
Scaling factors:

current transducers are calibrated for 1024=10A
note that the d-axis variable is three times this.
Voltage is calibrated to 1 unit=0.5V
Time is to 1/3.8MHz
*****************************************************************/
#include <stdlib.h>
#include <math.h>
#include "p_iface.h"
#include "iface.h"

#define IFACE /*use the interface*/


/* Output PAL Ports:
0: T0
1: T1
2: T2
3: [2..0]: sector
4: contactor and drive enables
5: [5..0] dead time [6] enable dead time compensation (+ve)*/

extern int Logging;
int LogStop=-1; /*set to +ve value to stop the log in n cycles*/
```

```
/*Output contactor bit definitions*/
#define MAIN_CON_CON 1 /*active high contactor bits*/
#define CH_CON_CON 2
#define SP1_CTRL 4
#define FAN1_CON 8
#define FAN2_CON 16
#define SP2_CTRL 32
#define SP3_CTRL 64
#define DRIVE_ENABLE 512 /*active high drive enable*/

/*output port definitions*/
#define WOP_T0 0
#define WOP_T1 1
#define WOP_T2 2
#define WOP_SECTOR 3
#define WOP_CONTACTOR 4
#define WOP_DEAD 5


/*DUAL PROCESSOR PORTS*/
#define A_WATCH_DOG 0x200000
#define A_INP_DATA_PORT 0x200040
#define A_INP_ADDR_PORT 0x200080
#define A_OUT_DATA_BASE 0x2001C0

#define A_PRIMARY_BCR 0x808064

/*VALUE CONSTANTS*/
#define Sqrt3 1.73205
#define ONEoverR3 37837ℓ
#define TWO_PII 1024

#define SW_FREQ (3.68e6/1280)
#define IF_PERIOD 16
int ITrip=0x600; /*15A*/
float MaxIsy=8;

/*********************************
Drive Control
*********************************/
int Running=0;
int Disabled=0;
int DriveStart=0;
int DriveStop=1;


/*********************************
Torque controller
*********************************/
int ICnt=0;
int Tc=0;

unsigned long int NWs;
int NIsx,NIsy;

/*Field-oriented controller tables*/
#define TAB_LEN TWO_PII
int STab[TAB_LEN];
int CTab[TAB_LEN];

/*Torque controller parameters*/
float Tr=10;        /*Nm of torque*/
float TrL=0;        /*Last torque for filtering*/
#define Pp 2
#define Lm 0.1144 /*H*/
#define Lr 0.1194 /*H*/
float Isx=5;        /*A   6.7*/
float Rr=0.7;       /*ohm 0.885*/
float Isy;          /*in amps*/
float Wsl;          /*slip in rad/sec*/
unsigned long int Ws; /*fixed point slip 2pi=65536*1024*/
unsigned long int WsS=0;/*integral of Ws*/
int Isxi,Isyi;

/*Estimates*/
int ThL;            /*last Theta*/
long int Delta,DeltaL=0; /*change in pos, last change in pos*/
long int DeltaA=0,DeltaAS=0;
long int Pos=0;
float W=0;
float X=0;

/*************Speed Control***************/
float Wset=0;       /*squarewave setpoint size*/
float Wbase=10;     /*dc offset of squarewave*/
float Wgain=3;      /*speed controller gain*/

/*************Position Control*************/
floatfloat Xset=1;  /*squarewave setpoint size*/
floatfloat J=  8.0; /*moment of inertia of load*/
float Dr=  0.7;     /*damping ratio of controller*/
float DclW=6;       /*closed loop frequency*/

int BCount=0;

float T_MAX=40;     /*max torque setpoint (Nm)*/


/*DUAL PROCESSOR IMPLEMENTATION CONSTANTS*/
#define M_PI 3.14159265
#define PERIOD 1280 /*1280*/

/*1024=10A, but setpoint is 3x for d axis and sqrt(3) times for q axis*/
int IDRl,IQRl;      /*last setpoint value, for debugging*/

/***********************************
Current Controller
***********************************/
/* phase currents are calibrated for 1024=10A, or 1unit=10mA*/
/* d-axis currents are 3* these values.   ie ID0=3i_d(0) */
/* q=axis currents are \sqrt(3)*   */
int ia,ib,ic;
float ID0L=0;       /* current measurements from last cycle */
float ID2L=0;
float ID0=0;        /* from this cycle t=0*/
float ID4=0;        /* t=0.25 */
float ID2=0;        /* t=0.5 */

float IQ0L=0;       /* current measurements from last cycle */
float IQ2L=0;
float IQ0=0;        /* from this cycle t=0*/
float IQ4=0;        /* t=0.25 */
float IQ2=0;        /* t=0.5 */

float IA0=0; /*for logging*/
float IA2=0;
float IB0=0; /*for logging*/
float IB2=0;
float IC0=0; /*for logging*/
float IC2=0;

float LEst=10*25;       /* inductance value/3 typ 200*/
/* multiply real value by 25k to do unit conversion */
/* 25k is for [v]=0.5V, [t]=1/3.8MHz [i]=10mA */
float Lk=14*25/3;   /*output from inductance estimator*/
float Ik=1;         /*conrol type (rho) average CC=2 , endpoint=1*/
float LScale=0.9;   /*multiply estimate by this to give value used*/

float AlphaD=0;     /* current duty cycles */
float AlphaQ=0;
float AlphaDL=0;    /* duty cycle from previous cycle */
float AlphaQL=0;

int t0,t1,t2;

float VDC=60*2;     /* use a fixed 60V for testing */
float VDCL=0;

int OCnt=0;         /*overrun count*/
int TripCode=0;


int useLEst=0;
float alphasum, isum, vk, Ck, yk, Pk, xhk, Tk;
float invL=1/0.999;

#define STORE_LEN 32 /*store of previous currents and voltages*/
float istore[STORE_LEN];
float astore[STORE_LEN];
int storeCtr;

float eEps=0.01;
float DeltaThEst;
float DeltaThEstk;

int idOffs=0;
int iqOffs=0;

float eFilt=0.95;

/*****************************************************************
ReadInPAL
*****************************************************************/
int ReadInPAL(int Port)
{
  int Val;
  *((int*)A_INP_ADDR_PORT)=Port|16; /*write address*/
  asm(" NOP"); asm(" NOP"); asm(" NOP"); asm(" NOP"); asm(" NOP");
  asm(" NOP"); asm(" NOP"); asm(" NOP"); asm(" NOP"); asm(" NOP");

  *((int*)A_WATCH_DOG)=1;
  *((int*)A_WATCH_DOG+1)=3;
  *((int*)A_WATCH_DOG+2)=5;
```

```
  Val=*((int*)A_INP_DATA_PORT); /*read value*/
  if (Val&1<<11)     /*if MSB is set*/
    return Val|~0xFFF; /*sign extend*/
  else
    return Val&0xFFF;
}


/*****************************************************************
WriteInPAL
*****************************************************************/
void WriteInPAL(int Port,int Value)
{
  *((int*)A_INP_DATA_PORT)=Value; /*write value*/
  asm(" NOP"); asm(" NOP"); asm(" NOP");
  asm(" NOP"); asm(" NOP");
  *((int*)A_INP_ADDR_PORT)=Port&15; /*then latch in*/
}

/*****************************************************************
WriteOutPAL
*****************************************************************/
void WriteOutPAL(int Port,int Value)
{
  *((int*)(A_OUT_DATA_BASE+Port))=Value; /*write value*/
}


/*****************************************************************
InitContactors
SetContactorBit
ClearContactorBit
*****************************************************************/
int WOP_C_Bits=0;
void InitContactors(void)
{
  WOP_C_Bits=0;      /*disable all*/

  WOP_C_Bits=512;   /*leave contactor in for testing*/

  WriteOutPAL(WOP_CONTACTOR,WOP_C_Bits);
}

void SetContactorBit(int BitPattern)
{
  WOP_C_Bits|=BitPattern;
  WriteOutPAL(WOP_CONTACTOR,WOP_C_Bits);
}

void ClearContactorBit(int BitPattern)
{
  WOP_C_Bits&=~BitPattern;
  WriteOutPAL(WOP_CONTACTOR,WOP_C_Bits);
}


/*****************************************************************
InitTab

Initialises the sine and cosine tables for the field-oriented
controller
*****************************************************************/
void InitTab(void)
{
  int i;
  for (i=0;i<TAB_LEN;i++)
    {
      STab[i]=sin(i*3.1415*2/TAB_LEN)*32767;
      CTab[i]=cos(i*3.1415*2/TAB_LEN)*32767;
    }
}


int Sector=0;
int T0=0,T1=0,T2=0;
int LLCount=0,LLDiv=0; /*inductance log counters*/


int dt=0;
int IDR,IQR;
int IDRy;


float ed,eq,e_m;   /*d, q back emf, filtered emd amplitude*/
float ed_pred, eq_pred;
float ed_last, eq_last;
float e_th,e_th_d; /*angle of back emf, change over one cycle*/


unsigned int ThM; /*measured theta*/
unsigned int ThP; /*predicted angle*/
unsigned int ThE; /*electrical angle*/
float DeltaThEf;   /*projected change in electrical angle, in rad*/

#define T PERIOD
/*****************************************************************
CCalc

Performs one iteration of the controller.
Outputs the calculated values to the PAL
*****************************************************************/
void CCCalc(float IDRo,float IQRo) /*parameters are obsolete*/
{
  float absAD,absAQ;

  float f;
  int i,j;

  /****************************************/
  /* wait for t=0 and read in the currents */
  /****************************************/
  while (ReadInPAL(5)&1);; /* wait for first measurement period*/

  ThM=ReadInPAL(4)&0x3FF;; /*read position here*/

  ia=ReadInPAL(0);   /* read the samples*/
  ib=ReadInPAL(1);
  ic=ReadInPAL(2);

  VDCL=VDC;          /*for logging only,*/
  VDC=ReadInPAL(3); /*read voltage*/
  if (VDC<20) VDC=20;
  WriteInPAL(1,1);     /*got samples*/
  WriteInPAL(1,0);

  /*do 3phase to 2 phase conversion*/
  ID0L=ID0; IQ0L=IQ0;
  IQ0=ib-ic-idOffs;
  ID0=2*ia-ib-ic-iqOffs;

  /*store past currents and voltages for the L est.*/
  storeCtr=(storeCtr+1)%STORE_LEN;
  istore[storeCtr]=ID0;
  astore[storeCtr]=AlphaD;

#ifdef IFACE
  PCheckIf();          /*poll for messages*/
  LogTime();           /*perform any logging*/
  if(LogStop>0)        /*stop logging if requested*/
    {
      LogStop--;
      if(!LogStop) lstop();
    }
#endif

  /****************************************/
  /* wait for t=0.25 and read in the currents*/
  /****************************************/
  /*the hardware supports a current sample at t=0.25, so
    we need to wait for it and get it, although with the
    new L estimator, the value is not used*/
  while (ReadInPAL(5)&2); /* wait normal*/
  WriteInPAL(1,1);     /*got samples*/
  WriteInPAL(1,0);

  /*==============================================
    Field Oriented Controller
    ==============================================*/

  /*do unwrapping of the position*/
  Delta=ThM-ThL;
  while (Delta>( TWO_PII/2)) Delta=Delta-TWO_PII;
  while (Delta<(-TWO_PII/2)) Delta=Delta+TWO_PII;
  if (abs(Delta)>TWO_PII/16) Delta=DeltaL;
  DeltaL=Delta;
  DeltaA+=Delta;     /*accumulated value for foc*/
  ThL=ThM;

  ThP=ThM+Delta;   /*predict one ahead*/


  /*accumulate slip and position changes with extra 16bits of precision*/
  /*extra shift on delta is for the two pole pairs*/
  DeltaThEf=(Ws/65536.0+Delta*2)*(2*3.1415/1024);
  if  (DeltaThEf>0.2) DeltaThEf=0.2;
  else if (DeltaThEf<-0.2) DeltaThEf=-0.2;

  WsS=(WsS+Ws+(Delta<<17))&((TWO_PII<<16)-1);
  ThE=WsS>>16;


  /*Speed estimate*/
  W=W*0.7+0.3*DeltaAS*1.0/IF_PERIOD/1024*SW_FREQ;
  //speed in Hz
  /*Position estimate*/
  X=1.0*Pos/TWO_PII;

  /*calc id ref*/
```

```
i=(1ℓ*CTab[ThE]*Isxi−1ℓ*STab[ThE]*Isyi)>>16; /* >>16 gives /2 */
IDR=i;
/*calc iq ref*/
i=(1ℓ*STab[ThE]*Isxi+1ℓ*CTab[ThE]*Isyi)>>16;
i=(1ℓ*i*ONEoverR3)>>16;
IQR=i;

/*we don't do the foc calc every cycle, on the other
  cycles, the l estimator is execed instead*/
if(++ICnt>=IF_PERIOD)
  {
    ICnt=0;
    DeltaAS=DeltaA; DeltaA=0;
    Pos+=DeltaAS;

    /*Speed estimate*/
    W=W*0.7+0.3*DeltaAS*1.0/IF_PERIOD/1024*SW_FREQ; //speed in Hz
    /*Position estimate*/
    X=1.0*Pos/TWO_PII;

    /*Setpoint oscillations*/
    if (++Tc>360) //240 /4 360
      {
        Tc=0;
        Tr=−Tr;
        Xset=−Xset;
        Wset=−Wset;
      }

    /*Speed control*/
    Tr=(Wset+Wbase−W)*Wgain; /*Speed control*/
    /*Position contol*/
    /*Xset=ReadInPAL(6)/512.0;*/
    /*Tr=(-J*(DclW*DclW*(X-Xset)+2*Dr*DclW*W)); /*XSet in revolutions*/
    /*Tr=0.4*Tr+0.6*TrL;        /*LPF Torque Setpoint*/
    TrL=Tr;

    if (Tr>T_MAX) Tr=T_MAX; else if (Tr<−T_MAX) Tr=−T_MAX;

    /*TORQUE CONTROLLER*/
    Isy=Tr/(3*Pp/2*Lm*Lm/Lr*Isx);
    if (Isy>MaxIsy) Isy=MaxIsy;
    else if (Isy<−MaxIsy) Isy=−MaxIsy;

    Wsl=Isy*Rr/(1.0*Lr*Isx); /* /Lr*/
    if (Wsl>120) Wsl=120;

    NWs=(65535.0*TWO_PII/2/M_PI)/SW_FREQ*Wsl;
    NIsx=Isx*2*3*1024/10; //10A=1024, d axis setpoint is 3× current
    NIsy=Isy*2*3*1024/10;

    Isxi=NIsx;
    Isyi=NIsy; Ws=NWs;
  }
else { /*if no FOC, the do inductance est*/

  /*********************************************************************
  Full-cycle inductance estimator
  *********************************************************************/
  if (storeCtr>2) {
    /*recursive least-squares*/
    /*alpha is offset by 1, becuase it is the future voltage*/
    alphasum=astore[storeCtr−1]−2*astore[storeCtr−2]+astore[storeCtr−3];
    isum=istore[storeCtr]−3*istore[storeCtr−1]+3*istore[storeCtr−2]−
      istore[storeCtr−3];
    Ck=alphasum*2*VDC;
    yk=isum;
    Pk=Tk/(1+Tk*Ck*Ck);
    xhk=xhk+Ck*Pk*(yk−Ck*xhk);
    Tk=Pk*invL;                    /* forgetting factor*/

    Lk=1/xhk;
    if (useLEst) LEst=Lk*LScale;
  }
}

/*=============================================*/


/*****************************************/
/* wait for t=0.5 and read in the currents*/
/*****************************************/
while (ReadInPAL(5)&2); /* wait normal*/

ia=ReadInPAL(0);   /* read the samples */
ib=ReadInPAL(1);
ic=ReadInPAL(2);
WriteInPAL(1,1);     /*got samples*/
WriteInPAL(1,0);

IA2=ia;                /*store values*/
IB2=ib;
IC2=ic;

/*do 3phase to 2 phase conversion*/
ID2L=ID2; IQ2L=IQ2;
IQ2=ib−ic−idOffs;
ID2=2*ia−ib−ic−iqOffs;

/*calc back-emfs*/
ed=AlphaD+LEst/VDC*1.0*(ID0−ID2);
eq=AlphaQ+LEst/VDC*1.0*(IQ0−IQ2);

/*calculate rotation speed estimate. est=rotation/sqrt(3)*/
if (fabs(eq)>fabs(ed)) {
  if (fabs(eq)>20) {
    DeltaThEstk=(ed_last−ed)/(3*eq);
    DeltaThEst=DeltaThEst*(1−eEps)+DeltaThEstk*eEps;
  }
}
else {
  if (fabs(ed)>20) {
    DeltaThEstk=(eq−eq_last)/ed;
    DeltaThEst=DeltaThEst*(1−eEps)+DeltaThEstk*eEps;
  }
}

/*do back-emf prediction*/
ed_pred=ed−DeltaThEf*eq*3;
eq_pred=eq+DeltaThEf*ed;

AlphaD=ed_pred+LEst/VDC*0.5*(IDR−2*ID2+ID0);
AlphaQ=eq_pred+LEst/VDC*0.5*(IQR−2*IQ2+IQ0);

absAD=fabs(AlphaD);
absAQ=fabs(AlphaQ);


/*******************Space-vector Modulation****************/
Sector=0;
/*****calculate the sector*****/
if (AlphaD<0) Sector|=4;
if (AlphaQ<0) Sector|=2;
if (absAQ−absAD<0) Sector|=1;

/*****Calculate the times*****/
if (!(Sector&1)) { /*sectors 2 and 5 */
  /*T1=-aD+abs(aQ) T2=aD+abs(aQ)*/
  T1=−AlphaD+absAQ;
  T2=AlphaD+absAQ;
}
else  {
  if(!(Sector&4)) { /*SECTORS 1,6 */
    /*T1=aD-abs(aQ)  T2=2*abs(aQ)*/
    T1=AlphaD−absAQ;
    T2=2*absAQ;
  }
  else {  /*SECTORS 3,4 */
    /* T1=2*abs(aQ) T2=-aD-abs(aQ)*/
    T1=2*absAQ;
    T2=−AlphaD−absAQ;
  }
}
T0=T/2−T1−T2;

/*****perform clipping*****/
if (T0<2) {
  f=(T/2−2)/((float)T1+T2);
  AlphaD*=f;   /* scale the alphas for next time */
  AlphaQ*=f;
  if (T1<T2)
    {
      T1=T1*f;/*(T-2)/(T1+T2);*/
      T2=T/2−T1−2;
    }
  else
    {
      T2=T2*f;/*(T-2)/(T1+T2);*/
      T1=T/2−T2−2;
    }
  T0=2;
}
/*****write the new switching times*****/
WriteOutPAL(0,T0);
WriteOutPAL(1,T1);
WriteOutPAL(2,T2);
WriteOutPAL(3,Sector);

/*****Detect Overrun*****/
if (!(ReadInPAL(5)&1)) {
  OCnt++;
}

}

/*start and stop functions*/
void DriveStartF(void) {DriveStart=1;}
void DriveStopF(void) {DriveStop=1;}
```

```
/*****************************************************************
******************************************************************
main
*****************************************************************
******************************************************************/

void main(void)
{
  int i,j,k;
  int*IDataReg=(int*)0x200040;
  int ctr=0;
  int BreakCtr=0;
  int QCnt=0;

  /*initialise the hardware*/
  InitContactors();
  /*program bus control register H/W wait states*/
  *((int*)A_PRIMARY_BCR)=0x10F0;

  /*Reg 0 should have t/80-1*/
  WriteInPAL(0,15);   /*set t/80=16 (enable compensatin 1<<6)*/
  WriteInPAL(0,15+(1<<11)+(1<<9)+(1<<10)); /*release reset*/
  WriteInPAL(1,0);    /*got samples*/

  /*Voltage calibration 1V=2 1024V=2048(full scale)*/
  WriteInPAL(2,79);   /*regen off (value/16) ie Volts/8*/
  WriteInPAL(3,88);   /*regen on*/

  WriteOutPAL(0,640); /*t0*/
  WriteOutPAL(1,0);
  WriteOutPAL(2,0);

  WriteOutPAL(5,20); /*20 dead time in (us*5)*/

  InitTab();              /*for the FOC*/


  /*Initialise FOC*/
  Isy=Tr/(3*Pp/2*Lm*Lm/Lr*Isx);
  Wsl=Isy/Lr/Isx;
  Ws=(65535.0*TWO_PII/2/M_PI)*(1280.0/4.91e6)*Wsl;
  Isxi=Isx*2*3*1024/10; //10A=1024, d axis setpoint is 3x current
  Isyi=Isy*2*3*1024/10;

  SetContactorBit(CH_CON_CON);
  SetContactorBit(MAIN_CON_CON);

  ThL=ReadInPAL(4); //initial position reading
  SetContactorBit(DRIVE_ENABLE);

  /*now add the interface data*/
#ifdef IFACE
  for (i=0;i<20;i++) /*synchronise with master cpu*/
    PIfaceInit();
  LogInit();
  PIfaceSendMessage("Slave Processor Started\r\n");
  PIfaceInit();

  PIfaceAddVariable(&OCnt,IF_INT,"overrun","Overrun detect");
  PIfaceAddVariable(&TripCode,IF_INT,"tripcode","Trip code details");
  PIfaceAddVariable(&MaxIsy,IF_FLOAT,"maxisy","max flux current setpoint");

  PIfaceAddVariable(&LEst,IF_FLOAT,"lest","Inductance value");
  PIfaceAddVariable(&Lk,IF_FLOAT,"lk","estimated inductance");
  PIfaceAddVariable(&useLEst,IF_INT,"usel","use estimated inductance");
  PIfaceAddVariable(&invL,IF_FLOAT,"invl","1/RLS forgetting factor");
  PIfaceAddVariable(&LScale,IF_FLOAT,"lscale","multiplier for L estimate");

  PIfaceAddVariable(&Wset,IF_FLOAT,"Wset","Speed variation");
  PIfaceAddVariable(&Wbase,IF_FLOAT,"Wbase","Average speed setpoint");
  PIfaceAddVariable(&Wgain,IF_FLOAT,"Wgain","Speed Controller gain");

  /*************Position Control*************/
  PIfaceAddVariable(&Xset,IF_FLOAT,"Xset","Pos control setpoint amplitude");
  PIfaceAddVariable(&J,IF_FLOAT,"J","Pos controller inertia");
  PIfaceAddVariable(&Dr,IF_FLOAT,"Dr","Pos controller damping ratio");
  PIfaceAddVariable(&DclW,IF_FLOAT,"DclW","closed loop frequency");

  PIfaceAddVariable(&Isx,IF_FLOAT,"isx","foc magnetising");
  PIfaceAddVariable(&Isy,IF_FLOAT,"isy","foc torque");
  PIfaceAddVariable(&Rr,IF_FLOAT,"rr","rotor resistance");


  PIfaceAddVariable((void*)DriveStartF,IF_FN0,"start","(re)start drive");
  PIfaceAddVariable((void*)DriveStopF,IF_FN0,"stop","stop drive");
  AddLoggingCmds();

  /*logging vars*/

  PIfaceAddVariable(&IDR,IF_INT,"IDR","current reference");
  PIfaceAddVariable(&IQR,IF_INT,"IQR","current reference");
  PIfaceAddVariable(&ID0L,IF_FLOAT,"IDa","current, t=0");
  PIfaceAddVariable(&ID4,IF_FLOAT,"IDb","current, t=0.25");
  PIfaceAddVariable(&ID2,IF_FLOAT,"IDc","current, t=0.5");
  PIfaceAddVariable(&VDCL,IF_FLOAT,"VDC","DC volts");
  PIfaceAddVariable(&AlphaD,IF_FLOAT,"AlphaD","duty cycle calculated");
  PIfaceAddVariable(&AlphaQ,IF_FLOAT,"AlphaQ","duty cycle calculated");
  PIfaceAddVariable(&ThM,IF_INT,"Pos","measured position");
  PIfaceAddVariable(&ThE,IF_INT,"EPos","electrical position");

  PIfaceAddVariable(&ed,IF_FLOAT,"ede","back-emf");
  PIfaceAddVariable(&eq,IF_FLOAT,"eqe","back-emf");
  PIfaceAddVariable(&ed_pred,IF_FLOAT,"edp","back-emf predicted");
  PIfaceAddVariable(&eq_pred,IF_FLOAT,"eqp","back-emf predicted");
  PIfaceAddVariable(&eEps,IF_FLOAT,"eeps","angle est filter coefficient");


  PIfaceAddVariable(&Tk,IF_FLOAT,"rlstk","recursive least squares Tk");
  PIfaceAddVariable(&idOffs,IF_INT,"idoffs","d axis current dc offset");
  PIfaceAddVariable(&iqOffs,IF_INT,"iqoffs","q axis current dc offset");

  PIfaceAddVariable(&eFilt,IF_FLOAT,"efilt","amount to filter e est, 0-1");
  PIfaceAddVariable(&DeltaThEst,IF_FLOAT,"deleo","delta e observer");
  PIfaceAddVariable(&DeltaThEf,IF_FLOAT,"delef","delta e foc");
#endif

  /*now calculate the offset on the transducers*/
  idOffs=0;
  iqOffs=0;

  WriteInPAL(0,15+(1<<11)+(1<<10)); /*trip reset*/
  WriteInPAL(0,15+(1<<11)+(1<<9)+(1<<10)); /*release reset*/

  for(k=0;k<16;k++) {
    while (ReadInPAL(5)&2); /* wait normal*/
    ia=ReadInPAL(0);   /* read the samples */
    ib=ReadInPAL(1);
    ic=ReadInPAL(2);
    WriteInPAL(1,1);    /*got samples*/
    WriteInPAL(1,0);

    idOffs+=ib-ic;
    iqOffs+=2*ia-ib-ic;
  }
  idOffs=idOffs/16;
  iqOffs=iqOffs/16;

  k=1;
  while(1) {
    VDCL=ReadInPAL(3);
    ThM=ReadInPAL(4)&0x3FF;

    if(Running) {
      if(ReadInPAL(4)&(2048+1024)) {
        PIfaceSendMessage("Tripped");
        Running=0;
        Disabled=0;
        WriteOutPAL(0,T/2); WriteOutPAL(1,0); WriteOutPAL(2,0);
        TripCode=ReadInPAL(7);
      }
      else if (DriveStop) {
        DriveStop=0;
        Running=0;
        Disabled=0;
        WriteOutPAL(0,T/2); WriteOutPAL(1,0); WriteOutPAL(2,0);
        ClearContactorBit(DRIVE_ENABLE);
        PIfaceSendMessage("Stopping");
      }
      else CCCalc(0,0);    /*do the current control*/
    }
    else {
#ifdef IFACE
      PCheckIf();
#endif
      if (DriveStart)    {
        DriveStart=0;
        Running=1; Disabled=0;
        AlphaD=0; AlphaQ=0;
        WriteOutPAL(0,T/2); WriteOutPAL(1,0); WriteOutPAL(2,0);

        WriteInPAL(0,15+(1<<11)+(1<<10)); /*trip reset*/
        WriteInPAL(0,15+(1<<11)+(1<<9)+(1<<10)); /*release reset*/

        ThL=ReadInPAL(4)&0x3FF; /*initial position*/
        AlphaD=0;
        AlphaQ=0;
        Ws=0; WsS=0; W=0; DeltaAS=0; DeltaA=0; ICnt=1000;
        Isxi=0; Isyi=0;
        ed_last=0; eq_last=0;
        DeltaL=0; TrL=0;

        DeltaThEst=0;
        for (k=0;k<STORE_LEN;k++) {
          istore[k]=0;
          astore[k]=0;
```

```
        }
        storeCtr=0;
        LEst=80;
        /*RLS inductance est: start with unknown state*/
        Tk=1/(1200*100/2*T)^2;
        xhk=1/40; /*initial inductance*/


        while (ReadInPAL(5)&1); /* wait for first*/
        WriteInPAL(1,1);    /*got samples*/
        WriteInPAL(1,0);
        while (ReadInPAL(5)&1); /* wait for first*/
        WriteInPAL(1,1);    /*got samples*/
        WriteInPAL(1,0);
        SetContactorBit(DRIVE_ENABLE);

        PIfaceSendMessage("Starting");
      }
    }
  }
}
```

A.2.2 c_iface.c

```
/*************************************************************
CONSOLE CODE
*************************************************************/

#include "iface.h"
#include <stdio.h>
#include <stdlib.h>
#include "m_serial.h"
#include "hwdefs.h"


/*************************************************************
Dual port details:
************
-Master writes to 7FF, slave writes to 7FE
-source of data stream writes to tail
-console is master, processor is slave
-interrupt when tail is written (ie add to queue)
-PCTail 7FE
-CPTail 7FF
-define PCHead 7FD, CPHead 7FC
*************************************************************/

#ifdef PC
extern struct TCBuff BuffPC;
extern struct TCBuff BuffCP;
#else
struct TCBuff BuffCP={DP_CP_Base,DP_CP_Head,DP_CP_Tail,256,0};
struct TCBuff BuffPC={DP_PC_Base,DP_PC_Head,DP_PC_Tail,256,0};
#endif


void CPrintVars(void);
void CPrintCmds(void);

struct TCVarData {
  char*Name; /* pointer to name & description(each 0 term)*/
  int Type;      /* the variable type */
  int Value;     /* current value */
};


struct TCCmd
{
  char*Name;
  char*Description;
  int (*Handler)(char*);
};


void CPutChar(char Ch)
{
#ifdef PC
  putc(Ch,stderr);
#else
  USendChar(Ch);
#endif
}


void CNewLine(void)
{
  CPutChar(13);
  CPutChar(10);
}


struct TCVarData CVars[IF_MAX_VARS];
int NumCVars=0;

struct TCCmd CCmds[]={
  {"vars","List Available Variables",(int(*)(char*))CPrintVars},
  {"cmds","List Available Commands",(int(*)(char*))CPrintCmds}
};
static int NumCCmds=2;


void CDisplayValue(int Num);


/*************************************************************
CHandlePacket
*************************************************************/
void CHandlePacket(int NumChars)
{
  int i,j;
  if (!NumChars) return;

  switch(IfaceGetChar(&BuffPC,0)) {

  case IF_PC_INIT:
    if (NumCVars)
    {
      for (i=0;i<NumCVars;i++) {
        free(CVars[NumCVars].Name);
        CVars[NumCVars].Name=NULL;
      }
      NumCVars=0;
    }
    break;

  case IF_PC_MESSAGE:
    CPutChar(8);                /*clear off the existing prompt*/
    for(i=1;(j=IfaceGetChar(&BuffPC,i))&&i<NumChars;i++) CPutChar(j);
    CNewLine();
        CPutChar('$');
    break;

  case IF_PC_TEXT: /*as above, but without bs or $*/
    for(i=1;(j=IfaceGetChar(&BuffPC,i))&&i<NumChars;i++) CPutChar(j);
    CNewLine();
    break;

  case IF_PC_ADDVAR:
    i=IfaceGetChar(&BuffPC,1);
    for(;NumCVars<i;NumCVars++) CVars[NumCVars].Name=NULL;
    CVars[i].Name=malloc(NumChars-3);
    CVars[i].Type=IfaceGetChar(&BuffPC,2);
    for(j=0;j<NumChars-3;j++)
      CVars[i].Name[j]=IfaceGetChar(&BuffPC,j+3);
    CVars[i].Value=0;
    /*      printf("added %s\n",CVars[NumCVars].Name);*/

    NumCVars++;
    break;

  case IF_PC_VARVAL:
    i=IfaceGetChar(&BuffPC,1);
    if (i<NumCVars) CVars[i].Value=IfaceGetWord(&BuffPC,2);
    CPutChar(8); /*back over the prompt*/
    CDisplayValue(i);
    CPutChar('$');
  }
}


int CLastCheck=0;               /* last character number checked */
/*************************************************************
CCheckIf
*************************************************************/
void CCheckIf(void)
{
  for (;CLastCheck<IfaceGetNumChar(&BuffPC);CLastCheck++)
    if (IfaceGetChar(&BuffPC,CLastCheck)==IF_DELIM) {
      CHandlePacket(CLastCheck);
      IfaceTakeChars(&BuffPC,CLastCheck+1);
      CLastCheck=0;
      break;
    }
}

int CPutString(char*String,int MaxLen)
{
  int i;
  char Ch;

  i=0;
  while((Ch=String[i]) && i<MaxLen) { /* output non-zero chars */
    CPutChar(Ch);
```

```
    i++;
  }
  return i;
}


void CDisplayValue(int Num)
{
  int j,k;
  char Buff[32];

    j=CPutString(CVars[Num].Name,24);
    /* j if offs to descript and length displayed*/
    CPutChar('='); j++;
    switch (CVars[Num].Type) {
    case IF_FLOAT:
      sprintf(Buff,"%g",CVars[Num].Value);
      break;
    case IF_INT:
      sprintf(Buff,"%i",CVars[Num].Value);
      break;
    }
    k=j+CPutString(Buff,16); /* k is the total length so far */
    if (k>=24)
      CPutChar(' ');            /* can't align, so just space */
    else
      for (;k<24;k+=8) CPutChar(9); /* otherwise tab to pos 24 */
    CPutString(CVars[Num].Name+j,80-(k>24?k:24)); /*and print description*/
    CNewLine();
}

/*****************************************************************
CPrintVars
*****************************************************************/
void CPrintVars(void)
{
  int i;

  for(i=0;i<NumCVars;i++) if ((CVars[i].Type&~7)!=IF_FN0) {
    CDisplayValue(i);
  }
}


/*****************************************************************
CPrintCmds
*****************************************************************/
void CPrintCmds(void)
{
  char Buff[32];
  int i,j,k;

  for(i=0;i<NumCVars;i++) if ((CVars[i].Type&~7)==IF_FN0) {
    j=CPutString(CVars[i].Name,24);
    /*j if offs to descript and length displayed*/
    CPutChar('('); j++;
    k=j;
    CPutChar('0'+(CVars[i].Type&7)); k++;
    CPutChar(')'); k++;
    if (k>=16)
      CPutChar(' ');  /* can't align, so just space */
    else
      for (;k<16;k+=8) CPutChar(9); /* otherwise tab to pos 16 */
    CPutString(CVars[i].Name+j,80-(k>16?k:16)); /*and print description*/
    CNewLine();
  }
}


/*****************************************************************

*****************************************************************/


/*****************************************************************
FindWhitespace

Returns the number of leading spaces in the string
*****************************************************************/
int FindWhitespace(char*Str)
{
  int i;

  for(i=0;i<80;i++) if (Str[i]!=32) return i;
  return 0;                     /* give up and return 0 */
}


/*****************************************************************
FindInteger
*****************************************************************/
int FindInteger(char*Str,int*Val)
{
  int i,j;

  *Val=0;                       /* start with zero */
  for(i=0;i<80;i++){
    j=Str[i]-'0';               /* find the digit */
    if (j>=0 && j<=9)
      *Val=*Val*10+j;           /* if valid, add to num */
    else
      return i;                 /* otherwise, we are finished */
  }
  return 0;
}

/*****************************************************************
FindFloat
*****************************************************************/
int FindFloat(char*Str,float*Val)
{
  int i,j;
  float Mult;

  *Val=0;
  for(i=0;i<80;){
    j=Str[i]-'0';               /* find the digit */
    if (j>=0 && j<=9) {
      *Val=*Val*10+j;           /* if valid, add to num */
      i++;
    }
    else
      break;
  }
  if(Str[i]!='.') return i;
  i++;                          /* go onto first decimal digit */
  Mult=0.1;
  for(;i<80;i++){
    j=Str[i]-'0';               /* find the digit */
    if (j>=0 && j<=9)
      *Val=*Val+j*Mult;         /* if valid, add to num */
    else
      return i;                 /* otherwise, we are finished */
    Mult*=0.1;
  }
  return 0;
}


/*****************************************************************
FindText
if Text and String match up to the first zero in Text, returns the
length of Text, otherwise returns 0
*****************************************************************/
int FindText(char*String,char*Text)
{
  int i;
  if (!Text || !String) return 0;

  for(i=0;i<80;i++) {           /* now compare the strings */
    if (!Text[i]) {             /* if at end, have found match */
      return i;
    }
    if (Text[i]!=String[i]) break; /* if different, no match */
  }
  return 0;
}


/*****************************************************************
FindVariable

If the string passed starts with text matching one of the variables,
VarNum is set to this variable number, and the string length of the
variable is returned.
Otherwise returns zero.
*****************************************************************/
int FindVariable(char*String,int*VarNum)
{
  int i,j;

  for(j=0;j<NumCVars;j++) { /* iterate through variables */
    if (i=(FindText(String,CVars[j].Name))) {
      *VarNum=j;
      return i;
    }
  }
  return 0;                     /* no matches found */
}


/*****************************************************************
FindCommand

If the string passed starts with text matching one of the variables,
VarNum is set to this variable number, and the string length of the
variable is returned.
Otherwise returns zero.
*****************************************************************/
```

```
int FindCommand(char*String,int*CmdNum)
{
  int i,j;

  for(j=0;j<NumCCmds;j++) { /* iterate through variables */
    if (i=(FindText(String,CCmds[j].Name))) {
      *CmdNum=j;
      return i;
    }
  }
  return 0;                    /* no matches found */
}


void PErr(char*Expect)
{
    CPutString("Expected: ",20);
    if (Expect) CPutString(Expect,20);
    CNewLine();
}


union intfloat{
  int i;
  float f;
};

int ParseString(char*String)
{
  int i,j;
  int CmdNum,ParamNum;
  /*make float-int recast go through mem */
  volatile union intfloat ival;
  float f;
  char TBuff[80];

  i=FindWhitespace(String);
  if (j=FindCommand(String+i,&CmdNum)) {
    CCmds[CmdNum].Handler(String+i+j);
  }
  else if (j=FindVariable(String+i,&CmdNum)) {
    i+=j;                      /* point i to the start of parameters */
    i+=FindWhitespace(String+i);
    switch(CVars[CmdNum].Type){

    case IF_INT:
    case IF_FLOAT:
      if(String[i++]=='=') {
        i+=FindWhitespace(String+i);
        if (CVars[CmdNum].Type==IF_INT)
          j=FindInteger(String+i,(int*)(&ival.i));
        else
          j=FindFloat(String+i,(float*)&ival.f);
        if(!j)
        {
          PErr("Value");
          return -1;
        }
        IfacePutChar(&BuffCP,IF_CP_VARSET); /*send a set msg */
        IfacePutChar(&BuffCP,CmdNum);
        IfacePutWord(&BuffCP,ival.i); /* the value */
        IfacePutChar(&BuffCP,IF_DELIM);
        IfaceSendChars(&BuffCP);

        sprintf(TBuff,"Changing var #%i to %i\n",CmdNum,ival.i);
        CPutString(TBuff,80);
        CNewLine();
      }
      else {
        IfacePutChar(&BuffCP,IF_CP_VARREQ); /*send a request */
        IfacePutChar(&BuffCP,CmdNum);
        IfacePutChar(&BuffCP,IF_DELIM);
        IfaceSendChars(&BuffCP);
      }
      break;
    case IF_FN0:
      IfacePutChar(&BuffCP,IF_CP_CMD); /*send a command message */
      IfacePutChar(&BuffCP,CmdNum);
      IfacePutChar(&BuffCP,IF_DELIM);
      IfaceSendChars(&BuffCP);
      break;

    case IF_FN1:
      if (FindInteger(String+i,&ParamNum)) {
        IfacePutChar(&BuffCP,IF_CP_CMD); /*send a command message */
        IfacePutChar(&BuffCP,CmdNum);
        IfacePutWord(&BuffCP,ParamNum); /* send the int */
        IfacePutChar(&BuffCP,IF_DELIM);
        IfaceSendChars(&BuffCP);
      }
      else PErr("Integer");
      break;

    case IF_FN_VAR:
      if (FindVariable(String+i,&ParamNum)) {
        IfacePutChar(&BuffCP,IF_CP_CMD); /*send a command message */
        IfacePutChar(&BuffCP,CmdNum);
        IfacePutWord(&BuffCP,ParamNum); /* send param number as an int */
        IfacePutChar(&BuffCP,IF_DELIM);
        IfaceSendChars(&BuffCP);
      }
      else PErr("Variable");
      break;
    }
  }
  else PErr("Command");
}

#define CMAX_INP_BUFFER_LEN 80
char CInpBuffer[CMAX_INP_BUFFER_LEN]={0};
int CInpBufferLen=0;
int LInpBufferLen=0;

void CCheckInput(void)
{
  int i;
  char*str;
  /*
  str=readline("Enter Cmd:");
  ParseString(str);
  free(str);
  */

  //return;
  // scanf("%c",&i);
#ifdef PC
  i=getchar();
  if (i==10) i=13;
#else
  if (!UTestChar()) return;    /* if no chars waiting return */
  i=UGetChar();
#endif
    // if (i==ERR) return;

  if (i==13) {
    CNewLine();
    CInpBuffer[CInpBufferLen++]=0;
    ParseString(CInpBuffer);
    LInpBufferLen=CInpBufferLen-1;
    CInpBufferLen=0;
    CPutChar('$');
  }
  else if (i==8){
    if (CInpBufferLen) { CInpBufferLen--; CPutChar(8);};
  }
  else if (i==16){ /*ctrl-p*/
    CInpBufferLen=LInpBufferLen;
    CPutString(CInpBuffer,80);
  }
  else {
    if (CInpBufferLen<CMAX_INP_BUFFER_LEN-1) {
      CInpBuffer[CInpBufferLen++]=i;
      CPutChar(i);
    }
  }
}


#ifdef PC
#else
void main(void)
{
  volatile int i;
  for (i=0;i<10000;i++); /*short delay for UART*/
  InitUART();

  CNewLine();
  CPutString("Console v1.0a Started",80);
  CNewLine();
  CPutChar('$');

  *(BuffPC.Head)=0;
  *(BuffPC.Tail)=0;
  (BuffPC.CTail)=0;

  *(BuffCP.Head)=0;
  *(BuffCP.Tail)=0;
  (BuffCP.CTail)=0;

  CPrintVars();
  CPrintCmds();

  while(1) {
    CCheckInput();
    CCheckIf();
  }
}
#endif
```

A.2.3 hwdefs.h

```
/*Dual processor interface hardware definitions*/
#define DP_BASE 0x300000
#define DP_PC_Base (void*)(DP_BASE+0)
#define DP_CP_Base (void*)(DP_BASE+256)
#define DP_PC_Tail (void*)(DP_BASE+0x7FE)
#define DP_CP_Tail (void*)(DP_BASE+0x7FF)
#define DP_PC_Head (void*)(DP_BASE+0x7FD)
#define DP_CP_Head (void*)(DP_BASE+0x7FC)
```

A.2.4 iface.h

```
/*****************************************************************
iface.h

Common interface header file
*****************************************************************/
#ifndef IFACE_H
#define IFACE_H

#define IF_MAX_VARS 64

#define IF_DELIM    0xFF      // send to indicate end of message
//Processor to console commands
#define IF_PC_NULL 0          // send a few of these before init
#define IF_PC_INIT  0x41      // init message, sent on boot
#define IF_PC_MESSAGE 0x42 // zero terminated string follows
#define IF_PC_ADDVAR 0x43 // add new variable (var,type,name,description)
#define IF_PC_VARVAL 0x44  // notify variable value (var,value)
#define IF_PC_TEXT     0x45 //zero terminated text, no $ or bs

//console to processor commands
#define IF_CP_CMD 0x61      // issue command (command#, opt params)
#define IF_CP_VARSET 0x62  // set the variable to value (variable,val)
#define IF_CP_VARREQ 0x63 // request variable value (variable)

//variable types
#define IF_INT  0x10  // specifies the type of variable (all are 32bit)
#define IF_FLOAT 0x11
#define IF_FN0  0x00            /* function with 0 parameters */
#define IF_FN1  0x01            /* function with 1 parameter */
#define IF_FN_VAR 0x02          /* function with variable name as param */
#define IF_MAX_STRLEN 40


/*-------------------------
TCBuff

Structure giving details of the dual port
circular buffer
-------------------------*/
struct TCBuff
{
  volatile char*Buffer;
  volatile int*Head;
  volatile int*Tail;
  int Len;                      /* total length of the buffer */
  int CTail;                    /* current tail pos,    *Tail only updated on the
                                   end of a packet*/
};


/*no OO in normal C, so pass a structure to each of the handling fns*/
int   IfaceGetChar(struct TCBuff *Buff,int CharNum);
int   IfaceGetWord(struct TCBuff *Buff,int CharNum);
int   IfaceGetNumChar(struct TCBuff *Buff);// number of chars in the buffer
void IfaceTakeChars(struct TCBuff *Buff,int NumChar);

int   IfacePutChar(struct TCBuff *Buff,int Char);
int   IfacePutWord(struct TCBuff*Buff,int);
int   IfacePutSpace(struct TCBuff *Buff);//amount of space left in the buffer
void IfaceSendChars(struct TCBuff*Buff);


void IfaceInit(void);
int IfaceAddVariable(void*Val,int type,char*Name,char*Description);
void IfaceSendMessage(char*Message);


#endif
```

A.2.5 iface.c

```
/*****************************************************************
iface.c

Includes interface routines common to both processors
*****************************************************************/
#include "iface.h"

/*-------------------------
Byte

de-references and masks out the top 24 bits
-------------------------*/
int Byte(volatile int*Val)
{
  return (((unsigned)*Val))&0xff;
}


/*-------------------------
IfaceMatchTail

Adjusts the private tail to match the public one
-------------------------*/
void IfaceMatchTail(struct TCBuff *Buff)
{
  Buff->CTail=Byte(Buff->Tail);
}


/*-------------------------
IfaceGetChar

//non-destructively look at a char in the buffer
-------------------------*/
int   IfaceGetChar(struct TCBuff *Buff,int CharNum)
{
  return Buff->Buffer[(Byte(Buff->Head)+CharNum)%(Buff->Len)]&0xff;
}

/*-------------------------
IFaceGetWord

non destructively look at a 32-bit word in the buffer
-------------------------*/
int   IfaceGetWord(struct TCBuff*Buff,int CharNum)
{
  return
    IfaceGetChar(Buff,CharNum)       |
    (IfaceGetChar(Buff,CharNum+1)<<7) |
    (IfaceGetChar(Buff,CharNum+2)<<14) |
    (IfaceGetChar(Buff,CharNum+3)<<21) |
    (IfaceGetChar(Buff,CharNum+4)<<28);
}

/*-------------------------
IfaceTakeChars

Remove NumChars from the buffer
-------------------------*/
void IfaceTakeChars(struct TCBuff*Buff,int NumChar)
{
  *(Buff->Head)=(Byte(Buff->Head)+NumChar)%Buff->Len;
}

/*-------------------------
IfaceGetNumChar

number of chars in the buffer
-------------------------*/
int   IfaceGetNumChar(struct TCBuff*Buff)
{
  int Num;

  Num=Byte(Buff->Tail)-Byte(Buff->Head);
  if (Num<0) Num+=Buff->Len;
  return Num;
}

/*****************************************************************/

/*-------------------------
IfacePutChar
-------------------------*/
int   IfacePutChar(struct TCBuff *Buff,int Char)
{
  while(!IfacePutSpace(Buff));   /*wait for space*/

  Buff->Buffer[Buff->CTail]=Char;
  Buff->CTail=(Buff->CTail+1)%Buff->Len;
```

```
}

/*-------------------------------------------------
IfacePutWord

32 bit words are spread over 5 bytes, 7bits in each, with the lsb first
-------------------------------------------------*/
int  IfacePutWord(struct TCBuff*Buff,int Word)
{
  int i;
  for (i=0;i<5;i++) {IfacePutChar(Buff,Word&0x7f); Word>>=7;}
}

/*-------------------------------------------------
IfaceSendChars

sends the queued message buy updating the tail pointer
-------------------------------------------------*/
void IfaceSendChars(struct TCBuff*Buff)
{
  *(Buff->Tail)=Buff->CTail;
}

/*-------------------------------------------------
IfacePutSpace

amount of space left in the buffer
-------------------------------------------------*/
int  IfacePutSpace(struct TCBuff *Buff)
{
  /*return Buff->Len-1-IfaceGetNumChar(Buff);*/
  int Space;

/*  Space=Byte(Buff->Head)-Byte(Buff->Tail)-1;*/
  Space=Byte(Buff->Head)-Buff->CTail-1;
  if (Space<0) Space+=Buff->Len;
  return Space;
}


/*****************************************************************
IfaceSendString

Writes the given string to the output stream, including null terminator
if the output buffer is full, waits for it.
*****************************************************************/
void IfaceSendString(struct TCBuff*Buff,char *String)
{
  int j;                        // count through chars
  char ch;                      // store char sent

  j=0;
  do {
    if(!IfacePutSpace(Buff)) {
      IfaceSendChars(Buff);
      while(!IfacePutSpace(Buff)) ; //wait for space
    }
    IfacePutChar(Buff,ch=String[j++]);
  } while (ch);
}
```

A.2.6 m_serial.h

```
/*****************************************************************
UART driver for the master CPU
*****************************************************************/

int InitUART(void);
void USendChar(int Val);
char UTestChar(void);
int UGetChar(void);
void USendString(char*String);
```

A.2.7 m_serial.c

```
/*****************************************************************
UART driver for the master CPU
*****************************************************************/
#include <stdio.h>

volatile char*UART_Reg=(volatile char*)0x200040;

/*UART read ports*/
#define U_SRA 1
#define U_RHRA 3
#define U_IPCR 4
#define U_ISR 5
#define U_CTU 6
#define U_CTL 7

/*UART write ports*/
#define U_MRA 0
#define U_CSRA 1
#define U_CRA 2
#define U_THRA 3
#define U_ACR 4
#define U_IMR 5
#define U_CRUR 6
#define U_CTLR 7


int InitUART(void)
{
  UART_Reg[U_CRA]=0x20;  /* receiver reset */
  UART_Reg[U_CRA]=0x1A;  /* reset mode register pointer to MR1A */
  UART_Reg[U_MRA]=0x13;  /* MR1A 8 bits, no parity */
  UART_Reg[U_MRA]=0x07;  /* MR2A one stop bit */
  UART_Reg[U_CSRA]=0xCC; /* 0xCC: 19.2k, use 0xBB for 9.6k */
  UART_Reg[U_ACR]=0x80;  /* set bit 7 for 19.2k mode */


  UART_Reg[U_CRA]=0x05;  /* enable rx and tx, 0A disables */

  /* U_SRA: bit 2: transmit ready, bit 0: receive ready*/

  return 0;
}

void USendChar(int Val)
{
  while(!(UART_Reg[U_SRA]&4)); /* wait until ready */
  UART_Reg[U_THRA]=Val;
}

char UTestChar(void)
{
  return UART_Reg[U_SRA]&1; /* return 1 if there is a char waiting */
}


int UGetChar(void)
{
  int ch;

  while(!(UART_Reg[U_SRA]&1)); /* wait until ready */
  ch=UART_Reg[U_RHRA]&0xFF;
  return ch;
}


void USendString(char*String)
{
  int i;

  for (i=0;i<80 && String[i];i++) USendChar(String[i]);
}
```

A.2.8 p_iface.h

```
/*initialisation functions*/
void PIfaceInit(void);
int PIfaceAddVariable(void*Val,int type,const char*Name,char*Description);
void LogInit(void);
void AddLoggingCmds(void);

/*Runtime functions*/
void PIfaceSendMessage(char*Message);
```

```
void PCheckIf(void);
void LogTime(void);
void lstop(void);
```

A.2.9 p_iface.c

```
#include <stdlib.h>
#include "iface.h"
#include "hwdefs.h"


struct TIVarDat
{
  void*Var;                    /* pointer to the variable */
  int Type;                    /* var type / number of params */
  const char*Name;
};

struct TIVarDat VarDat[IF_MAX_VARS];
int NumVars=0;


#ifdef PC
char CTBuffer[256];
int CTHead=240;
int CTTail=240;
struct TCBuff BuffPC={CTBuffer,&CTHead,&CTTail,256,240};
char PTBuffer[256];
int PTHead=240;
int PTTail=240;
struct TCBuff BuffCP={PTBuffer,&PTHead,&PTTail,256,240};
#else
struct TCBuff BuffCP={DP_CP_Base,DP_CP_Head,DP_CP_Tail,256,0};
struct TCBuff BuffPC={DP_PC_Base,DP_PC_Head,DP_PC_Tail,256,0};
#endif


/*****************************************************************
PIfaceInit
*****************************************************************/
void PIfaceInit(void)
{
  int i,j;
  char ch;

  IfaceMatchTail(&BuffPC);

  IfacePutChar(&BuffPC,IF_PC_INIT); // sent the init char
  IfacePutChar(&BuffPC,IF_DELIM);
  IfaceSendChars(&BuffPC);
  NumVars=0;
}


/*****************************************************************
PIfaceAddVariable
*****************************************************************/
int PIfaceAddVariable(void*Val,int type,const char*Name,char*Description)
{
  if (NumVars>=IF_MAX_VARS) return -1;

  VarDat[NumVars].Var=Val;   /* copy the pointer to the variable */
  VarDat[NumVars].Name=Name;
  IfacePutChar(&BuffPC,IF_PC_ADDVAR); /* send a notification string */
  IfacePutChar(&BuffPC,NumVars);
  IfacePutChar(&BuffPC,VarDat[NumVars].Type=type); /* keep the type */
  IfaceSendString(&BuffPC,Name);
  IfaceSendString(&BuffPC,Description);
  IfacePutChar(&BuffPC,IF_DELIM);
  IfaceSendChars(&BuffPC);
  NumVars++;

  return 0;
}


/*****************************************************************
PIfaceSendMessage
*****************************************************************/
void PIfaceSendMessage(char*Message)
{
  IfacePutChar(&BuffPC,IF_PC_MESSAGE); /* send a notification string */
  IfaceSendString(&BuffPC,Message);
  IfacePutChar(&BuffPC,IF_DELIM);
  IfaceSendChars(&BuffPC);
}

/*****************************************************************
PIfaceSendText
*****************************************************************/
void PIfaceSendText(char*Message)
{
  IfacePutChar(&BuffPC,IF_PC_TEXT); /* send a notification string */
  IfaceSendString(&BuffPC,Message);
  IfacePutChar(&BuffPC,IF_DELIM);
  IfaceSendChars(&BuffPC);
}


/*****************************************************************
PHandlePacket
*****************************************************************/
void PHandlePacket(int NumChars)
{
  int i,j;
  union {
        int (*FnPtr)(int);
        void* vptr;
  } FnPtr;

  if (!NumChars) return;

  switch(IfaceGetChar(&BuffCP,0)) {

  case IF_CP_CMD:
    i=IfaceGetChar(&BuffCP,1);
    if (i<0 || i>=NumVars) return;
    j=IfaceGetWord(&BuffCP,2);
    FnPtr.vptr=VarDat[i].Var;
    (*FnPtr.FnPtr)(j);
    break;
  case IF_CP_VARSET:
    i=IfaceGetChar(&BuffCP,1);
    if (i<0 || i>=NumVars) return;
    j=IfaceGetWord(&BuffCP,2);
    *((int*)VarDat[i].Var)=j;
    break;
  case IF_CP_VARREQ:
    i=IfaceGetChar(&BuffCP,1);
    if (i<0 || i>=NumVars) return;
    j=*((int*)VarDat[i].Var);
    IfacePutChar(&BuffPC,IF_PC_VARVAL); /* send a notification string */
    IfacePutChar(&BuffPC,i);
    IfacePutWord(&BuffPC,j);
    IfacePutChar(&BuffPC,IF_DELIM);
    IfaceSendChars(&BuffPC);
    break;
  }
}

int PLastCheck=0;              /* last character number checked */
/*****************************************************************
PCheckIf
*****************************************************************/
void PCheckIf(void)
{
  for (;PLastCheck<IfaceGetNumChar(&BuffCP);PLastCheck++)
    if (IfaceGetChar(&BuffCP,PLastCheck)==IF_DELIM) {
      PHandlePacket(PLastCheck);
      IfaceTakeChars(&BuffCP,PLastCheck+1);
      PLastCheck=0;
      break;
    }
}


/*****************************************************************
Logging section
*****************************************************************/

#define MAX_LOGGED_VARS 8
int LoggedVars[MAX_LOGGED_VARS];
int NumLogs=0;              /* number of variables to log */

int Logging;                /*indicate whether we are logging now*/
int LogNum;                 /*number of variable sets logged*/
int LogCtr;                 /*Current log pointer*/

int LogSpace;               /*number of words in the log array*/
int LogSets;                /* number of data sets (LogSpace/NumLogs) */
int*LogData;                /*pointer to the array itself*/

/*****************************************************************
LogInit

init data structures
*****************************************************************/
void LogInit(void)
{
  NumLogs=0;
  Logging=0;
  LogCtr=0;
```

```
  LogSpace=48*1024;
  do {
    LogSpace/=2;
    LogData=malloc(LogSpace);
  } while(!LogData && LogSpace);
}


/*************************************************************
lclear

clear logs
*************************************************************/
void lclear(void)
{
  NumLogs=0;
  Logging=0;
  LogCtr=0;
}


/*************************************************************
ladd

add a variable to the log list
*************************************************************/
void ladd(int LVar)
{
  char Buff[80];

  LogCtr=0;   /*clear any existing data*/
  if (NumLogs<MAX_LOGGED_VARS) {
      LoggedVars[NumLogs++]=LVar;
    sprintf(Buff,"Added Log Variable %s.",VarDat[LVar].Name);
    PIfaceSendMessage(Buff);
  }
  else {
    PIfaceSendMessage("Too Many Logged Variables");
  }

}


/*************************************************************
lstart

start logging
*************************************************************/
void lstart(void)
{
  char Buff[80];

  if(!NumLogs)
    PIfaceSendMessage("No Logged Variables");
  else if (!LogSpace)
    PIfaceSendMessage("No Memory Available");
  else {
    LogCtr=0;
    LogNum=0;
    Logging=1;
    LogSets=LogSpace/NumLogs;
    sprintf(Buff,"Logging %i sets of %i variables",LogSets,NumLogs);
    PIfaceSendMessage(Buff);
  }
}


/*************************************************************
lstop

stop logging
*************************************************************/
void lstop(void)
{
  char Buff[80];
  if(Logging){
    Logging=0;
    sprintf(Buff,"Logged %i sets, %i available.",
            LogNum,LogNum>LogSets?LogSets:LogNum);
    PIfaceSendMessage(Buff);
  }
  else {
    PIfaceSendMessage("Not Logging.");
  }
}


/*************************************************************
lvars

print the variables that are being logged
*************************************************************/
void lvars(void)
{
  int i;
  for (i=0;i<NumLogs;i++) {
    PIfaceSendMessage((char*)VarDat[LoggedVars[i]].Name);
  }
}


/*************************************************************
lprint

print the log contents
*************************************************************/
void lprint(void)
{
  int i,j,k;
  char Buff[120];

  if (!LogNum) {
    PIfaceSendMessage("No Values Logged.");
    return;
  }

  /* print the variable names */
  lvars();

  if (LogNum>LogSets)       /* have wrapped */
    i=LogCtr-LogSets;
  else
    i=0;
  if (i<0) i+=LogSets;      /* find the oldest existing record */

  do {
    for (j=0;j<NumLogs;j++) {
      switch(VarDat[LoggedVars[j]].Type) {
      case IF_INT:
        sprintf(Buff+10*j," %9.7i",LogData[i*NumLogs+j]);
        break;
      case IF_FLOAT:
        sprintf(Buff+10*j," %9.7f",LogData[i*NumLogs+j]);
        break;
      default:
        for (k=0;k<10;k++) Buff[10*i+k]=' '; Buff[10*(i+1)]=0;
      }
    }
    PIfaceSendText(Buff);
    i++;
    i%=LogSets;
  } while (i!=LogCtr);

}


/*************************************************************
LogTime

add an extra line to the log table
*************************************************************/
void LogTime(void)
{
  int i;

  if (!Logging) return;

  for (i=0;i<NumLogs;i++) {
    LogData[LogCtr*NumLogs+i]=*(int*)VarDat[LoggedVars[i]].Var;
  }
  LogCtr++;
  LogNum++;
  if (LogCtr>=LogSets) LogCtr=0;
}

/*logging commands:
lclear   clear logs
ladd     add log (clears also) (param)
lstart   start logging
lstop    stop logging
lvars    print logged vars
lprint   print logged values

functions:
LogTime - called regularly by the main loop

void PCheckIf(void);
void LogTime(void);
void AddLoggingCmds(void);
*/

/*************************************************************
AddLoggingCmds

Add the logging commands to the interface
*************************************************************/
void AddLoggingCmds(void)
```

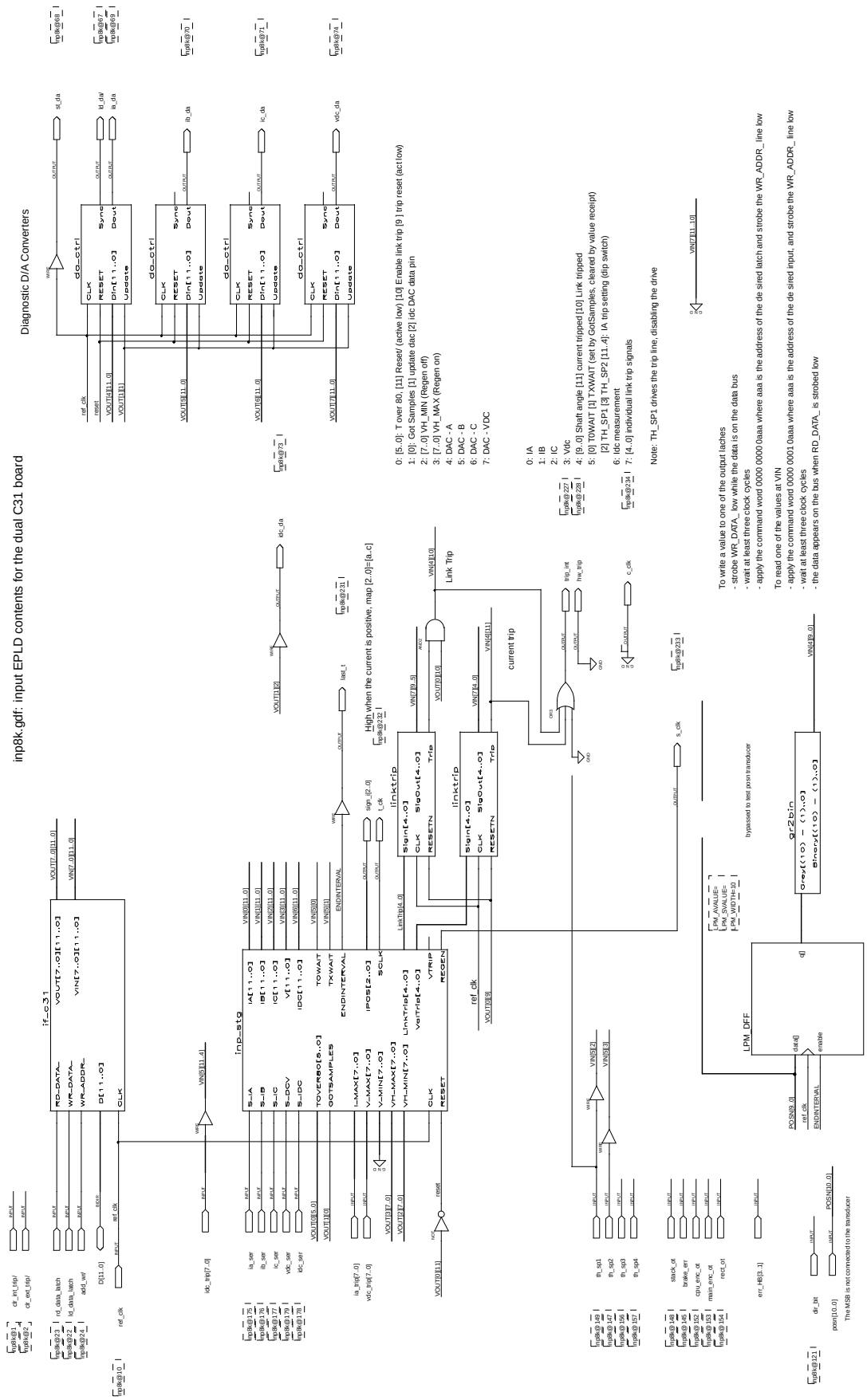
```
{
  PIfaceAddVariable((void*)lclear,IF_FN0,"lclear","Clear logs");
  PIfaceAddVariable((void*)ladd,IF_FN_VAR,"ladd" ,"Add log (clears values)");
  PIfaceAddVariable((void*)lstart,IF_FN0,"lstart","Start logging");
  PIfaceAddVariable((void*)lstop ,IF_FN0,"lstop" ,"Stop logging");
  PIfaceAddVariable((void*)lvars ,IF_FN0,"lvars" ,"Print logged vars");
  PIfaceAddVariable((void*)lprint,IF_FN0,"lprint","Print logged values");
}
```

A.3 Data Acquisition Firmware

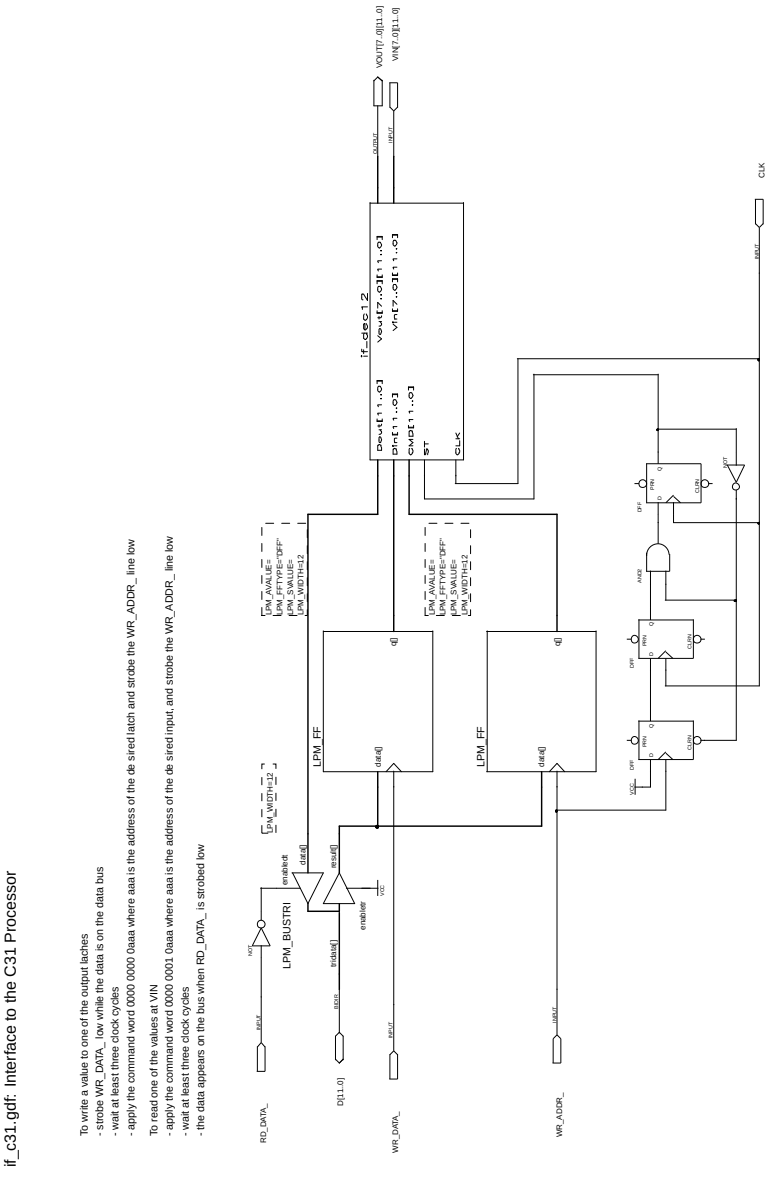
The data acquisition consists of a number of design files for the FLEX8K series Altera device. The top-level design file is called `inp8k.gdf`. The other files in the design heirarchy consist of a mixture of the graphical `.gdf` files, and the .tdf files in Altera's AHDL design language.

The following section details the modulation design, which is placed in the other 8K series EPLD. The top-level file for that design is `out_stg.gdf`. Both the data acquisition and modulation modules are also used in the hardware controller.

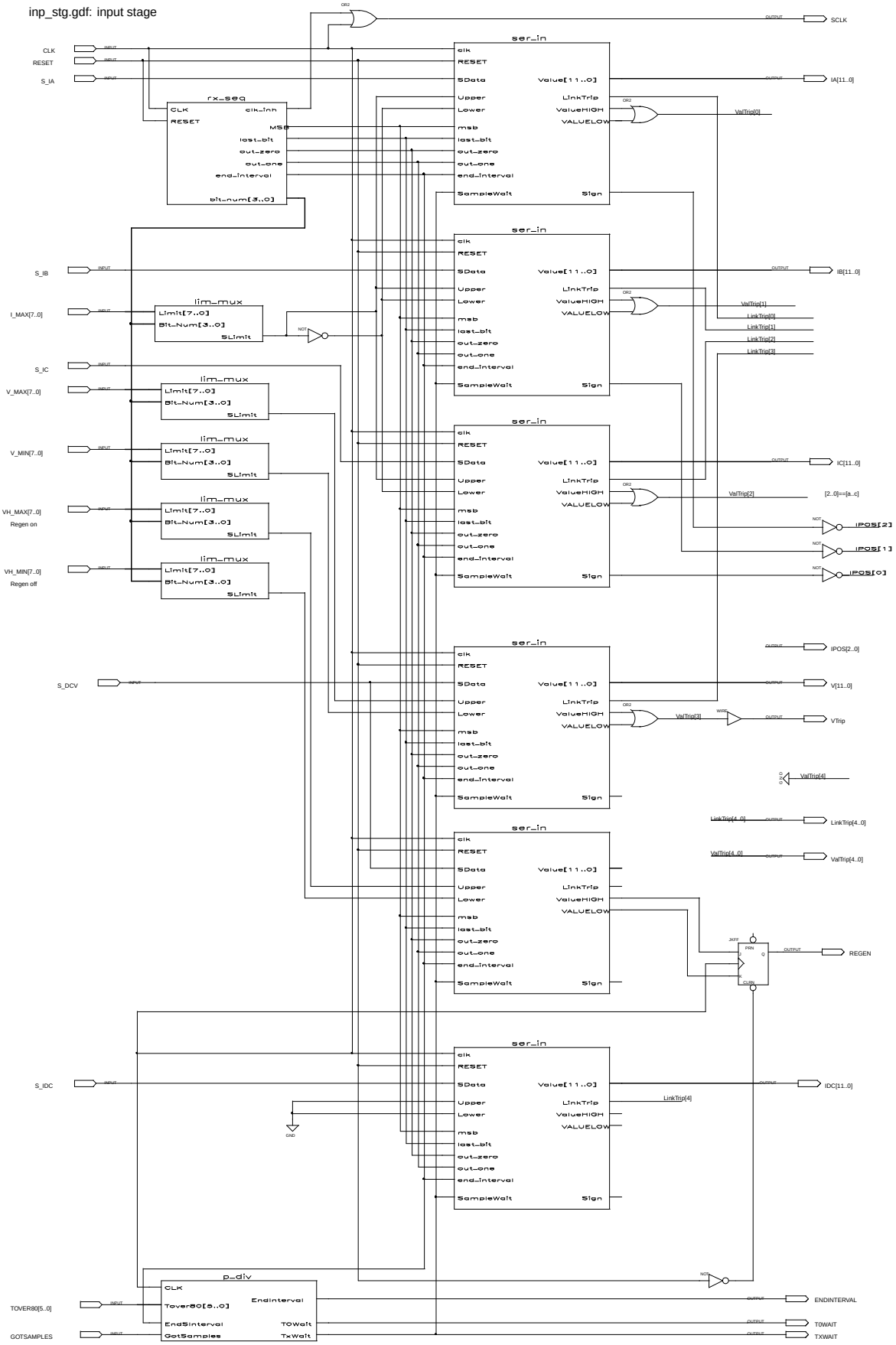
A.3.1 inp8k.gdf



A.3.2 if_c31.gdf



A.3.3 inp_stg.gdf



A.3.4 da_ctrl.tdf

```
%=============================================
da_ctrl.tdf

controls the on-board d/a converters, which are used
for debugging purposes.
=============================================%
INCLUDE "lpm_mux";


SUBDESIGN da_ctrl                                          10
(
  CLK,RESET: INPUT;
  Din[11..0]:  INPUT;
  Update:      INPUT;

  Sync,Dout:   OUTPUT;
)

VARIABLE
  dactr[3..0]:  DFF;                                       20
  Shifting:     DFF;
  ShiftingL:    DFF;          --allows lengthening of the sync pulse
  UpdateL:      DFF;
  shmux:        lpm_mux with (LPM_WIDTH=1, LPM_SIZE=12, LPM_WIDTHS=4);

BEGIN

  dactr[].clk=CLK;
  Shifting.clk=CLK;
  ShiftingL.clk=CLK;                                       30
  ShiftingL=Shifting;

  --keep state of update on last clock for edge detect
  UpdateL.clk=CLK;
  UpdateL=Update;

  --detect if it is the start of an update
  if (!shifting & Update & !UpdateL) then
    shifting=vcc;
    dactr[]=15;                 --preload bit counter      40
  else  --not the start of an update
    if (shifting & dactr[]==0) then --end of a cycle
      shifting=gnd;
    else
      shifting=shifting;
    end if;
    dactr[]=dactr[]-1;          --count down through the bits
  end if;


  shmux.data[][0]=Din[];                                   50
  shmux.sel[]=dactr[];
  Dout=shmux.result[0];

  Sync=!(Shifting);
-- Sync=!(Shifting#ShiftingL);
--it looks as if Sync has to stay low for an extra cycle after D0 to
--satisfy the hold time from clock falling edge (max 190ns)

END;                                                       60
```

A.3.5 gr2bin.tdf

```
%=============================================
gr2bin.tdf

grey code to binary converter.  For position encoder
=============================================%
CONSTANT CWIDTH=10;

SUBDESIGN GR2BIN
(
  Grey[CWIDTH-1..0]: INPUT;                                10
  Binary[CWIDTH-1..0]: OUTPUT;
)
VARIABLE
  CVAL[CWIDTH-1..0]: NODE;

BEGIN
  CVAL[CWIDTH-1]=!Grey[CWidth-1];
  FOR i IN 0 TO CWidth-2 GENERATE
    CVAL[i]=CVAL[i+1]$(!Grey[i]);
  END GENERATE;                                            20
  Binary[]=CVAL[];
END;
```

A.3.6 if_dec12.tdf

```
%=============================================
if_dec12.tdf

interface between the c31 bus and the logic.  Offers
bidirectional latching
=============================================%
INCLUDE "lpm_mux";
INCLUDE "lpm_decode";


                                                           10
%-------------------------------------
Interface decoder:
-----------

Version: 8 inputs and outputs of 12 bits width.

Command types:
0000 0aaa: 186 writes to latch aaa
0001 0aaa: 186 reads input aaa into the temporary storage latch

-------------------------------------%                    20

SUBDESIGN IF_DEC12_DEC12
(
  CLK: INPUT;

  Din[11..0]:  INPUT;
  Dout[11..0]: OUTPUT;

  CMD[11..0]: INPUT;
  ST:          INPUT;                                      30

  Vout[7..0][11..0]:   OUTPUT;
  Vin[7..0][11..0]:    INPUT;
)


VARIABLE
  OMUX:        lpm_mux with (LPM_WIDTH=12,LPM_SIZE=8,LPM_WIDTHS=3);
  OLTCH[11..0]: DFFE;
                                                           40
  IDEC:        lpm_decode with (LPM_WIDTH=3, LPM_DECODES=8);
  ILTCH[7..0][11..0]: DFFE;


BEGIN
  --186 Reads from Altera
  OMUX.sel[]=CMD[2..0];
  OMUX.data[][]=Vin[][];

  OLTCH[].clk=CLK;                                         50
  OLTCH[].ena=ST & CMD[4]; --write to latch
  OLTCH[].d=OMUX.result[];

  Dout[]=OLTCH[];

  --186 Writes to Altera
  IDEC.data[]=CMD[2..0];
  IDEC.enable=ST & !CMD[4];

  ILTCH[][].clk=CLK;                                       60
  FOR i IN 0 TO 7 GENERATE
    ILTCH[i][].d=Din[];       --watch this
    ILTCH[i][].ena=IDEC.eq[i]; --and this
  END GENERATE;

  Vout[][]=ILTCH[][];

END;
```

A.3.7 lim_mux.tdf

```
%=============================================
lim_mux

Trip limit multiplexer - converts the 8 bit parallel trip limits into
a serial data stream for the ser_in modules. The select pattern is
X011- bit 7 MSB (maps to the MSB of the input signal)
X010- bit 6
X001- bit 5
X000- bit 4
X111- bit 3
X110- bit 2
X101- bit 1
X100- bit 0 LSB
=============================================%
INCLUDE"mux";
SUBDESIGN lim_mux
(
  Limit[7..0]:   INPUT;
  Bit_Num[3..0]: INPUT;

  SLimit:        OUTPUT;
)

VARIABLE
  smux:          mux with (WIDTH=8, WIDTHS=3);

BEGIN
  smux.sel[1..0]=Bit_Num[1..0];
  smux.sel[2]=!Bit_Num[2];
  smux.data[]=Limit[];

  SLimit=smux.result;
END;
```

A.3.8 linktrip.tdf

```
%=============================================
linktrip.tdf

maintains the trip status of the serial links
=============================================%
SUBDESIGN LinkTrip
(
  SigIn[4..0]:  INPUT;
  CLK:          INPUT;
  RESETN:       INPUT;

  SigOut[4..0]: OUTPUT;
  Trip:         OUTPUT;
)

VARIABLE
  Mem[4..0]:  JKFF;

BEGIN

  Mem[].j=SigIn[];
  Mem[].k=gnd;
  Mem[].clk=CLK;
  Mem[].clrn=RESETN;

  SigOut[]=Mem[].q;
  Trip=Mem[].q!=0;

END;
```

A.3.9 p_div.tdf

```
%=============================================
p_div.tdf

The global clock divider
=============================================%
constant Tbits=5;       --1 less than the number of bits in T/80

SUBDESIGN p_div
(
  CLK:                  INPUT;

  Tover80[Tbits..0]:    INPUT;
  EndSIntervalSInterval: INPUT;     --final interval in 20-cycle sampling
  GotSamples:           INPUT;      --signal that samples have been
                                    --read in, allowing change

  EndIntervalInterval:  OUTPUT;     --true on last interval before t=0
  T0Wait,TxWait:        OUTPUT;
)

VARIABLE
  Tcount[Tbits..0]:     DFFE;       --T/80 counter
  Tzero:                DFF;        --cycle ofter end of interval marker
  Tone:                 DFF;        --end of interval marker
  Ttwo:                 DFF;        --one before
  Tthree:               DFF;        --two before
  Fcount[1..0]:         DFFE;       --four (quater) counter
  T0W,TxW:              DFF;        --wait flip-flops
BEGIN

  --T/80 counter
  Tcount[].clk=CLK;
  Tcount[].ena=EndSInterval;
  if Tzero then
    Tcount[].d=Tover80[];
  else
    Tcount[].d=Tcount[].q-1;
  end if;

  Tzero.clk=CLK;
  Tzero=(Tcount[]==0);

  Tone.clk=CLK;
  Tone=(Tcount[]==1);

  Ttwo.clk=CLK;
  Ttwo=(Tcount[]==2);

  Tthree.clk=CLK;
  Tthree=(Tcount[]==3);

  --four counter
  Fcount[].clk=CLK;
  Fcount[].ena=Tone&EndSInterval;
  Fcount[].d=Fcount[].q+1;

  --End Interval
  --MOD — moves endinterval forward 20 cycles by placing it in the Ttwo cycle.
  EndIntervalInterval=(Fcount[]==3)&Ttwo&EndSInterval;

  --Wait Flip-flops
  T0W.clk=CLK; T0Wait=T0W;
  TxW.clk=CLK; TxWait=TxW;

  if GotSamples then
    --after samples taken, set wait again
    T0W=vcc;
    TxW=vcc;
  else
    --clear wait after 1st interval
    T0W=T0W&!( (Fcount[]==0)&Tzero&EndSInterval );
    TxW=TxW&!( (Fcount[]!=3)&Tzero&EndSInterval );
  end if;
END;
```

A.3.10 rx_seq.tdf

```
%=============================================
rx_seq

Receiver Sequencer

note: bits referred to are the values on the output of the first
shift register ff
=============================================%
SUBDESIGN rx_seq
(
  CLK, RESET:           INPUT;

  clk_inh:              OUTPUT;
  MSB,last_bit:         OUTPUT; --first and last bits (last is last parity)
  out_zero, out_one:    OUTPUT; --bits should be 0 or 1 respectively (start bits)
                                --reset the parity counter on out_one
  bit_num[3..0]:        OUTPUT; --current bit, 11=sign, 0=LSB
                                --used for protection mux
  end_interval_interval: OUTPUT; --last cycle before conversion start
                                --reset parity errors on this
)

VARIABLE
  RSM: machine with states (rS1,rS2,rS3,rMSB,rOthers,rP1,rP2,rEnd,rFill1,rFill2);
```

```
  Ctr[3..0]:    DFF;
  ResetCtr:     NODE;
BEGIN
  DEFAULTS
    ResetCtr=gnd;
  END DEFAULTS;

  Ctr[].clk=CLK;
  if ResetCtr then      --count down counter unless reset
    Ctr[]=11;
  else
    Ctr[]=Ctr[]-1;
  end if;
  bit_num[]=Ctr[];

  RSM.clk=CLK;
  RSM.reset=RESET;

  case RSM is
    when rS1=>  out_one=vcc;                RSM=rS2;
    when rS2=>  out_zero=vcc;               RSM=rS3;
    when rS3=>  out_one=vcc;
                ResetCtr=vcc;               RSM=rMSB;
    when rMSB=> MSB=vcc;                    RSM=rOthers;
    when rOthers=> if Ctr[]==0 then RSM=rP1; else RSM=rOthers; end if;
    when rP1=> clk_inh=vcc;                 RSM=rP2;
    when rP2=> clk_inh=vcc; last_bit=vcc;   RSM=rEnd;
    when rEnd=> end_interval_interval=vcc;  RSM=rFill1;
    when rFill1=>                           RSM=rFill2;
    when rFill2=>                           RSM=rS1;
  end case;


END;
```

A.3.11 scompare.tdf

```
%============================================
scompare

Performs a comparison between the two signed input data streams. Inputs
A and B are signed binary numbers presented MSB first. The MSB input
is driven high while the sign bit is presented.

On the the first difference, the A_lt_B output goes
high if A is less than B.
============================================%
SUBDESIGN scompare
(
  clk:          INPUT;
  A,B,MSB:      INPUT;

  A_lt_B:       OUTPUT;
)

VARIABLE
  Leq,Llt:      DFF;    --equal and less-than flip-flops
  eq,lt:        NODE;   --input nodes to the above

BEGIN
  A_lt_B=lt;            --output on first difference

  Leq.clk=CLK; Leq.d=eq;
  Llt.clk=CLK; Llt.d=lt;

  if MSB then
    eq=A!$B;            --are equal if both the same
    lt=A&!B;            --less than, if A is negative and B positive
  else
    if Leq then         --was still equal up to last stage so compare
      eq=A!$B;
      lt=B&!A;          --less than, if A=0 and B=1
    else                --was different, so keep result
      eq=Leq;
      lt=Llt;
    end if;
  end if;
END;
```

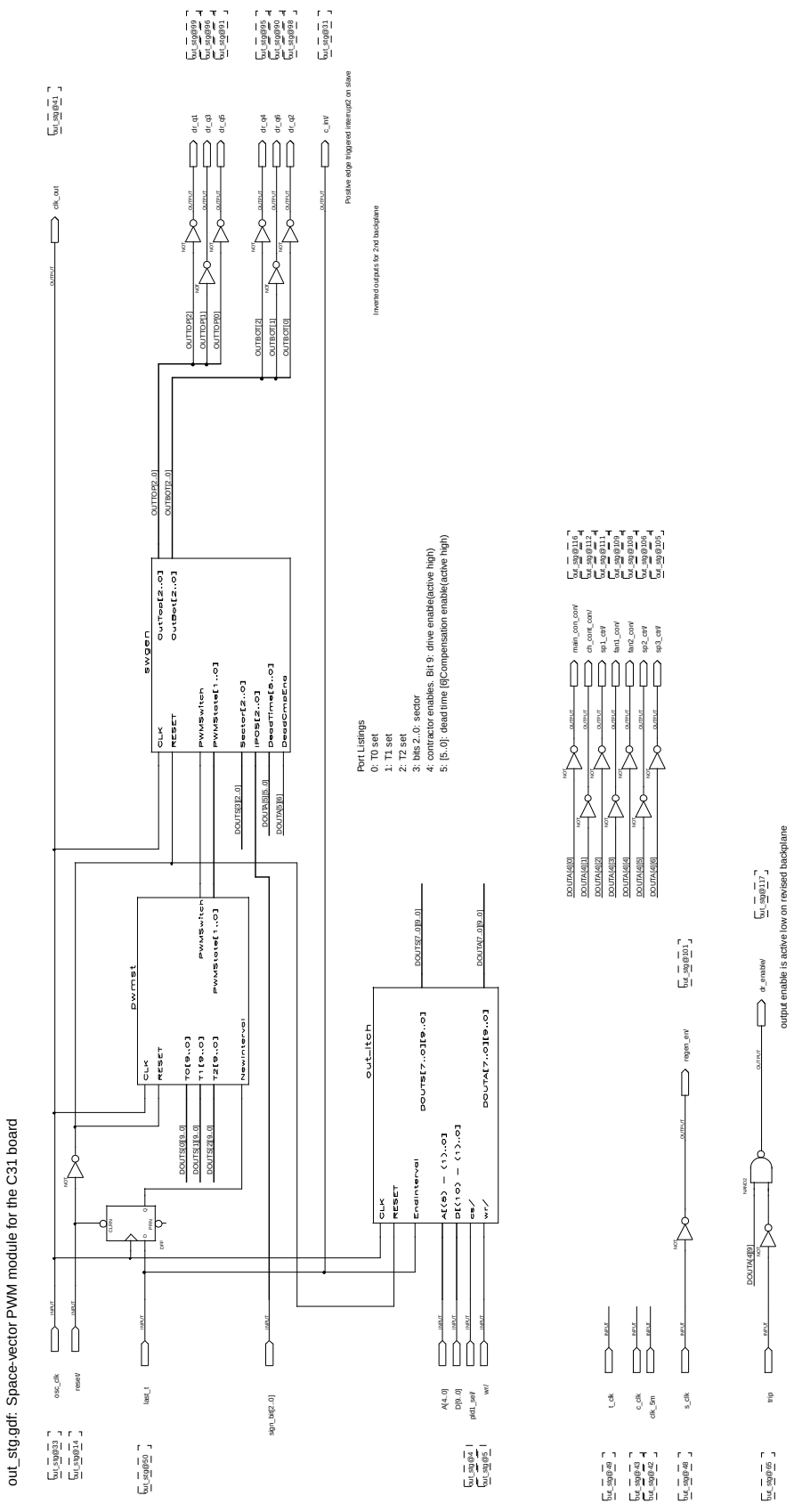
A.3.12 ser_in.tdf

```
%============================================
ser_in

Serial Input module - reads in the serial data and converts into
parallel form, also checks the parity and the trip limits

note: bits referred to are the values on the output of the first
shift register ff
============================================%
INCLUDE"scompare";

SUBDESIGN ser_in
(
  CLK,RESET:            INPUT;
  SData:                INPUT;

  Upper,Lower:          INPUT; --upper and lower trip limits presented serially

  MSB,last_bit:         INPUT; --first and last bits (last is last parity)
  out_zero, out_one:    INPUT; --bits should be 0 or 1 respectively (start bits)
  end_interval_interval: INPUT; --last cycle before conversion start

  SampleWait:           INPUT; --update output register when this is true

  Value[11..0]:         OUTPUT; --output value
  LinkTrip:             OUTPUT; --trip flags
  ValueHigh,ValueLow:   OUTPUT; --value trips
  Sign:                 OUTPUT; --sign of current sample
)

VARIABLE
  SReg[14..0]:          DFF;
  CmpU,CmpL:            scompare;
  FrameError:           JKFF;
  Parity:               JKFF;

  CData:                NODE;

  ErrorCnt[1..0]:       DFF;
  IncErrorCnt:          NODE;
  Reg[11..0]:           DFFE;
  LoadReg:              NODE;
  SignL:                DFFE;

BEGIN
  DEFAULTS
    FrameError.j=gnd; FrameError.k=gnd;
    LoadReg=gnd;
    IncErrorCnt=gnd;
    LinkTrip=gnd;

    ValueHigh=gnd;
    ValueLow=gnd;
--  ValueTrip=gnd;
  END DEFAULTS;

  --Sign
  SignL.clk=CLK;
  SignL.d=CData;
  SignL.ena=MSB;
  Sign=SignL;

  --shift register
  SReg[].clk=CLK;
  SReg[14..1].d=SReg[13..0].q;
  SReg[0].d=SData;
  CData=SReg[0];

  --Output register
  Reg[].clk=CLK;
  Reg[11..0].d=SReg[13..2].q;       --was [14..3] for some reason?
  Reg[].ena=LoadReg;
  Value[]=Reg[];

  --Error Counter
  ErrorCnt[].clk=CLK;
  ErrorCnt[].clrn=!Reset;
  if ErrorCnt[]==3 then
    LinkTrip=vcc;
    ErrorCnt[]=0;
  else
    if IncErrorCnt then
      ErrorCnt[]=ErrorCnt[]+1;
    else
      ErrorCnt[]=ErrorCnt[];
    end if;
  end if;

  --comparators: A is the trip level, B is the incoming value
```

```
−−this allows reasonably symmetric limits if ones complement is used for the
−−lower trip level.
CmpU.clk=CLK;
CmpU.A=Upper;
CmpU.B=CData;
CmpU.MSB=MSB;

CmpL.clk=CLK;
CmpL.A=Lower;
CmpL.B=CData;                                                    100
CmpL.MSB=MSB;

−−Framing
FrameError.clk=CLK;
if end_interval_interval then FrameError.k=vcc; end if; −−clear error on new interval
if (out_zero&CData)#(out_one&!CData) then FrameError.j=vcc; end if;
                                                  −−set if wrong framing

−−Parity
Parity.clk=CLK;                                                  110
if out_one then                                   −−clear parity on start bits
  Parity.k=vcc;
  Parity.j=gnd;
else
  Parity.k=CData;
  Parity.j=CData;
end if;

−−Handle the last bit
if last_bit then                                                 120
  if FrameError#Parity!$CData then                −−if error in frame
    IncErrorCnt=vcc;                              −−increase error counter
  else
    if SampleWait then LoadReg=vcc; end if; −−if waiting for samples store val
    −−check if the value was ok;
    ValueHigh=CmpU.A_lt_B;
    ValueLow =!CmpL.A_lt_B;
−−    if CmpU.A_lt_B#!CmpL.A_lt_B then ValueTrip=vcc; end if;
  end if;
end if;                                                          130

END;
```

A.4 Modulation Firmware

A.4.1 out_stg.gdf



A.4.2 dead.tdf

```
%============================================
dead.tdf

Forms complementary drives, inserts lockout and optional
dead-time compensation
============================================%
INCLUDE "DEAD_TME";
INCLUDE "DEAD_CMP";


SUBDESIGN Dead
(
  CLK, RESET: INPUT;
  IPos[2..0]:    INPUT;
  DEV_IN[2..0]: INPUT;

  DEAD_TIME[5..0]:  INPUT;
  DeadCmpEna:       INPUT;

  OutTop[2..0]: OUTPUT;
  OutBot[2..0]: OUTPUT;
)

VARIABLE

cmp[2..0]:      DEAD_CMP;
tme[2..0]:      DEAD_TME;

BEGIN
  cmp[].clk=CLK;
  cmp[].reset=RESET;
  cmp[].DEV_IN=DEV_IN[];
  cmp[].I_POS=IPOS[];
  cmp[].DEAD_TIME[]=DEAD_TIME[];

  tme[].clk=CLK;
  tme[].reset=RESET;
  IF DeadCmpEna THEN        -- dead time compensation enabled
    tme[].DEV_REQ=cmp[].DEV_REQ;
    tme[].TOP_DIS=cmp[].TOP_DIS;
    tme[].BOT_DIS=cmp[].BOT_DIS;
  ELSE
    tme[].DEV_REQ=DEV_IN[]; -- compensation disabled
    tme[].TOP_DIS=GND;
    tme[].BOT_DIS=GND;
  END IF;
  tme[].DEAD_TIME[]=DEAD_TIME[];


  OutTop[]=tme[].OutTop;
  OutBot[]=tme[].OutBot;
END;
```

A.4.3 dead_cmp.tdf

```
%============================================
dead_cmp

dead-time compensation
============================================%

SUBDESIGN dead_cmp
(
  CLK, RESET: INPUT;

  DEV_IN:     INPUT;        --the desired device output
  I_POS:      INPUT;        --sign of current - active if positive

  DEAD_TIME[5..0]: INPUT; --the length of deadtime in clock cycles

  DEV_REQ:            OUTPUT; --the desired device output
  TOP_DIS, BOT_DIS: OUTPUT; --top and bottom disable signals


)


VARIABLE
  DEV_L:      DFF;

  LH_SM:      machine with states (LH_IDLE, LH_DELAY, LH_DEAD);
  LH_CTR[5..0]: DFF;
  LH_CtrLoad: NODE;

  HL_SM:      machine with states (HL_IDLE, HL_DELAY, HL_DEAD);
  HL_CTR[5..0]: DFF;
  HL_CtrLoad: NODE;

  DEV_OUT:    DFFE;


BEGIN
  DEFAULTS
    LH_CtrLoad=gnd;
    HL_CtrLoad=gnd;
    DEV_OUT.ena=gnd;
    BOT_DIS=gnd;
    TOP_DIS=gnd;
  END DEFAULTS;

  --last device input
  DEV_L.clk=CLK;
  DEV_L.d=DEV_IN;

  --counter load
  LH_CTR[].clk=CLK; HL_CTR[].clk=clk;
  if LH_CtrLoad then LH_CTR[]=DEAD_TIME[]; else LH_CTR[]=LH_CTR[]-1; end if;
  if HL_CtrLoad then HL_CTR[]=DEAD_TIME[]; else HL_CTR[]=HL_CTR[]-1; end if;

  --device output
  DEV_OUT.clk=CLK;
  DEV_REQ=DEV_OUT;
  --for the two following transitions, disable the active device immediately
  if  I_POS & (LH_SM==LH_DELAY) then BOT_DIS=vcc; end if;
  if !I_POS & (HL_SM==HL_DELAY) then TOP_DIS=vcc; end if;

  LH_SM.clk=CLK;   HL_SM.clk=CLK;
  LH_SM.reset=RESET; HL_SM.reset=RESET;

  --Low to High transition
  if DEV_IN&!DEV_L then --rising edge of device drive signal
    LH_SM=LH_DELAY;
    LH_CtrLoad=VCC;
  end if;
  if LH_SM==LH_DELAY & LH_CTR[]==1 then --end of delay
    DEV_OUT.d=vcc;
    DEV_OUT.ena=vcc;
    LH_SM=LH_IDLE;
  end if;

  --High to Low transitions
  if DEV_L&!DEV_IN then --falling edge
    HL_SM=HL_DELAY;
    HL_CtrLoad=VCC;
  end if;
  if HL_SM==HL_DELAY & HL_CTR[]==1 then --end of delay
    DEV_OUT.d=gnd;
    DEV_OUT.ena=vcc;
    HL_SM=HL_IDLE;
  end if;


END;
```

A.4.4 dead_tme.tdf

```
%============================================
dead_tme

Insert lock-out for a single half-bridge
============================================%

SUBDESIGN dead_tme
(
  CLK, RESET:       INPUT;

  DEV_REQ:          INPUT; --the desired device output
  TOP_DIS, BOT_DIS: INPUT; --top and bottom disable signals

  DEAD_TIME[5..0]: INPUT; --the length of deadtime in clock cycles

  OutTop, OutBot:   OUTPUT;
)


VARIABLE
  USED_TOP,USED_BOT: DFF;
  DEAD_CTR[5..0]:   DFF;
  DEAD_CTR_LOAD:  NODE;

  OUT_TOP_L,OUT_BOT_L: DFF; --output latches
  NEXT_TOP,NEXT_BOT: NODE;
```

```
BEGIN
  OUT_TOP_L.clk=CLK;      --output latch signals
  OUT_TOP_L.d=NEXT_TOP;
  OutTop=OUT_TOP_L;

  OUT_BOT_L.clk=CLK;
  OUT_BOT_L.d=NEXT_BOT;
  OutBot=OUT_BOT_L;

  NEXT_TOP= DEV_REQ&!(USED_BOT#OUT_BOT_L#TOP_DIS);
  NEXT_BOT=!DEV_REQ&!(USED_TOP#OUT_TOP_L#BOT_DIS);

  DEAD_CTR_LOAD=OUT_TOP_L#OUT_BOT_L#RESET;
  DEAD_CTR[].clk=CLK;
  if DEAD_CTR_LOAD then
    DEAD_CTR[]=DEAD_TIME[];
  else
    DEAD_CTR[]=DEAD_CTR[]-1;
  end if;

  USED_TOP.clk=CLK;
-- USED_TOP.prn=RESET;
  if OUT_TOP_L then
    USED_TOP.d=vcc;
  elsif DEAD_CTR[]==2 then --count down to 2 to allow for delays
    USED_TOP.d=gnd;
  else
    USED_TOP.d=USED_TOP.q;
  end if;

  USED_BOT.clk=CLK;
-- USED_BOT.prn=RESET;
  if OUT_BOT_L then
    USED_BOT.d=vcc;
  elsif DEAD_CTR[]==2 then --count down to 2 to allow for delays
    USED_BOT.d=gnd;
  else
    USED_BOT.d=USED_BOT.q;
  end if;


END;
```

A.4.5 out_ltch.tdf

```
%=============================================
out_ltch.tdf

latches in values from the C31 bus
=============================================%
TITLE "PWM Output stage latch";

INCLUDE "lpm_ff";
INCLUDE "lpm_decode";

CONSTANT A_WIDTH=5;
CONSTANT A_USED=3;
CONSTANT OUTPUTS=8;
CONSTANT D_WIDTH=10;

SUBDESIGN out_ltch
(
  CLK, RESET:   INPUT;
  A[A_WIDTH-1..0]:INPUT;
  D[D_WIDTH-1..0]: INPUT;
  cs/:          INPUT;
  wr/:          INPUT;

  EndIntervalInterval: INPUT;

  --Asynchronous outputs
  DOUTA[OUTPUTS-1..0][D_WIDTH-1..0]: OUTPUT;
  --outputs synchronous to interval
  DOUTS[OUTPUTS-1..0][D_WIDTH-1..0]: OUTPUT;
)


VARIABLE
  ILATCH[OUTPUTS-1..0]: LPM_FF with (LPM_WIDTH=D_WIDTH);
  OLATCH[OUTPUTS-1..0]: LPM_FF with (LPM_WIDTH=D_WIDTH);

BEGIN


  ILATCH[].DATA[]=D[];
  ILATCH[].ENABLE=lpm_decode (.data[]=A[], .enable=!cs/)
   WITH (LPM_WIDTH=A_WIDTH, LPM_DECODES=OUTPUTS);
  ILATCH[].CLOCK=wr/;
  ILATCH[].ACLR=RESET;

  OLATCH[].DATA[]=ILATCH[].Q[];
  OLATCH[].ENABLE=EndInterval;
  OLATCH[].CLOCK=CLK;
  OLATCH[].ACLR=RESET;

  DOUTA[][]=ILATCH[].q[];
  DOUTS[][]=OLATCH[].q[];


% DOUT[][]=lpm_ff
  (
    .data[]=ILATCH[].q[],
    .enable=EndInterval,
    .clock=CLK,
    .aclr=RESET
  )
  with
  (
    LPM_WIDTH=D_WIDTH
  );
%

END;
```

A.4.6 pwmst.tdf

```
%=============================================
generates a sequence of states corresponding to the output PWM
vectors.  This does not allow for dead time and doesnt calcuate
the actual vector (done in swgen.tdf).
=============================================%
TITLE "PWM State Generator";

INCLUDE "pwm_comm";

--Counter preload multiplexer settings
CONSTANT SelT0= 0;
CONSTANT SelT1=2;
CONSTANT SelT2=3;
CONSTANT SelT0over2 =1;

% in pwm_comm
-Output States
CONSTANT ST0A=0;
CONSTANT ST0B=1;
CONSTANT ST1=2;
CONSTANT ST2=3;
%

SUBDESIGN pwmst
(
  CLK, RESET: INPUT;
  T0[9..0], T1[9..0], T2[9..0]: INPUT;
  NewInterval:  INPUT;

  PWMSwitch, PWMState[1..0]: OUTPUT;
)

VARIABLE

  --state machine
  P1: machine with states (T0E,T0S,T1S,T2S,T0M,T2E,T1E);

  --nodes to implement an OR function on the inputs
  T1NZ, T2NZ:       NODE;

  --time delay counter
  PCNT[9..0]:       DFF;
  PCNTsrc[1..0]:    NODE;
  PCNTload:         NODE;
  EndCountCount:    NODE;


BEGIN
  DEFAULTS
    PCNTLoad=GND;       --dont preload conter unless specified
  END DEFAULTS;

  --Outputs
  if P1==T0E then
    PWMSwitch=NewInterval;
  else
    PWMSwitch=EndCount;
  end if;
  --on all states except T0E, the end of the state is denoted by EndCount, but
  -- on T0E, it waits for NewInterval in order to ensure synchonisation
```

```
--state machine
P1.clk = CLK;
P1.reset = RESET;

--time delay counter
PCNT[].clk=CLK;
if PCNTload then
  case PCNTsrc[] is
    when SelT0 => PCNT[].d = T0[];
    when SelT1 => PCNT[].d = T1[];
    when SelT2 => PCNT[].d = T2[];
    when SelT0over2 => PCNT[8..0].d = T0[9..1]; PCNT[9].d=GND;
  end case;
else
  PCNT[].d = PCNT[].q-1;
end if;
if PCNT[].q==1 then EndCountCount=vcc; else Endcountcount=gnd; end if;

--Tx=0 detectors
if T1[]==0 then T1NZ=gnd; else T1NZ=vcc; end if;
if T2[]==0 then T2NZ=gnd; else T2NZ=vcc; end if;
case (P1) is
  when T0E =>
    --wait in this state for the NewInterval signal to indicate the
    --start of a new 1024 clock interval
    if NewInterval then
      PCNTLoad=VCC;              --load the new T0/2
      PCNTsrc[]=SelT0over2;
      P1=T0S;
    end if;
    PWMState[]=ST0A;


    --in each of the following states, if the state counter has reached the
    --end of count, a branch is made to the next state in the sequence.
    --If the next is zero in length, it is skipped and instead the following
    --one is used.
    when T0S =>
      if (!T1NZ & !T2NZ) then P1=T0E;
        --if both T1 and T2 are zero, then go straight to the end of the
        --control interval, as no switching is required.
      else

      if EndCountCount then
        PCNTLoad=VCC;
        if T1NZ then
          PCNTsrc[]=SelT1; P1=T1S;
        else
          PCNTsrc[]=SelT2; P1=T2S;
        end if;
      end if;
      PWMState[]=ST0A;

      end if;


    when T1S =>
      if EndCountCount then
        PCNTLoad=VCC;
        if T2NZ then
          PCNTsrc[]=SelT2; P1=T2S;
        else
          PCNTsrc[]=SelT0; P1=T0M;
        end if;
      end if;
      PWMState[]=ST1;

    when T2S =>
      if EndCountCount then
        PCNTLoad=VCC;
        PCNTsrc[]=SelT0; P1=T0M;        --T0 is not zero, so must go there
      end if;
      PWMState[]=ST2;

    when T0M =>
      if EndCountCount then
        PCNTLoad=VCC;
        if T2NZ then
          PCNTsrc[]=SelT2; P1=T2E;
        else
          PCNTsrc[]=SelT1; P1=T1E;
        end if;
      end if;
      PWMState[]=ST0B;

    when T2E =>
      if EndCountCount then
        PCNTLoad=VCC;
        if T1NZ then
          PCNTsrc[]=SelT1; P1=T1E;
        else
          PCNTsrc[]=SelT0over2; P1=T0E;
        end if;
      end if;
      PWMState[]=ST2;

    when T1E =>
      if EndCountCount then
        PCNTLoad=VCC;
        PCNTsrc[]=SelT0over2; P1=T0E;
      end if;
      PWMState[]=ST1;

  end case;
END;
```

A.4.7 swgen.tdf

```
%=============================================
Calculates the switching vectors based on the state in the sequence and
allows for dead time.
=============================================%
TITLE "Switching Generator";


INCLUDE "pwm_comm";
INCLUDE "dead";

-- Space vector firing allocations for phases "ABC"
CONSTANT V0=B"000";
CONSTANT V1=B"100";
CONSTANT V2=B"110";
CONSTANT V3=B"010";
CONSTANT V4=B"011";
CONSTANT V5=B"001";
CONSTANT V6=B"101";
CONSTANT V7=B"111";


SUBDESIGN swgen
(
  CLK, RESET:       INPUT;
  PWMSwitch:        INPUT; --1 to indicate a change in the PWMState after the
                           --next clock
  PWMState[1..0]:   INPUT; --the state in the pwm sequence
  Sector[2..0]:     INPUT; --current operation sector - defn in pwm_comm.inc
                           --these two together give the vector.

  IPOS[2..0]:       INPUT; --1 if current is +ve for each A,B,C
  DeadTime[5..0]:   INPUT; --clock counts for dead time
  DeadCmpEna:       INPUT; --when high, enables the dead time compensation.

  OutTop[2..0]:     OUTPUT; --output device drive signals — active high
  OutBot[2..0]:     OUTPUT;
)


VARIABLE
  CVector[2..0]:    NODE;
  DT:               Dead;

BEGIN

  --determine the current vector
  --this is interpreting the Firing order column in Table 1
  case (PWMState[]) is
    when ST0A => CVector[]=V0;
    when ST0B => CVector[]=V7;
    when ST1 =>
      case (Sector[]) is
        when Sect1, Sect6 => CVector[]=V1;
        when Sect2, Sect3 => CVector[]=V3;
        when Sect4, Sect5 => CVector[]=V5;
      end case;
    when ST2 =>
      case (Sector[]) is
        when Sect1, Sect2 => CVector[]=V2;
        when Sect3, Sect4 => CVector[]=V4;
        when Sect5, Sect6 => CVector[]=V6;
      end case;
  end case;

  dt.clk=clk;
  dt.reset=reset;

  dt.IPos[]=IPos[];
  dt.DEV_IN[]=CVector[];
  dt.DEAD_TIME[]=DeadTime[];
  dt.DeadCmpEna=DeadCmpEna;

  OutTop[]=dt.OutTop[];
  OutBot[]=dt.OutBot[];
END;
```

Appendix B

Hardware Implementation Details

B.1 Introduction

The hardware design shares some of the design elements with the software based controller. The core difference is that the software program `cc1.c` has been replaced with an ALU, a sequencer and the control algorithm written in microcode. In this chapter, the additional Altera firmware design is first presented. Following this, the control algorithm in microcode is presented. Finally, the code to convert this to the internal representation is shown.

B.2 Altera Design Files

The top-level design file for the hardware implementation is `if_vsd.gdf`. A number of files are common with the TMS320C31 design which has already been detailed. The following files are part of this design, but are included in the previous chapter:

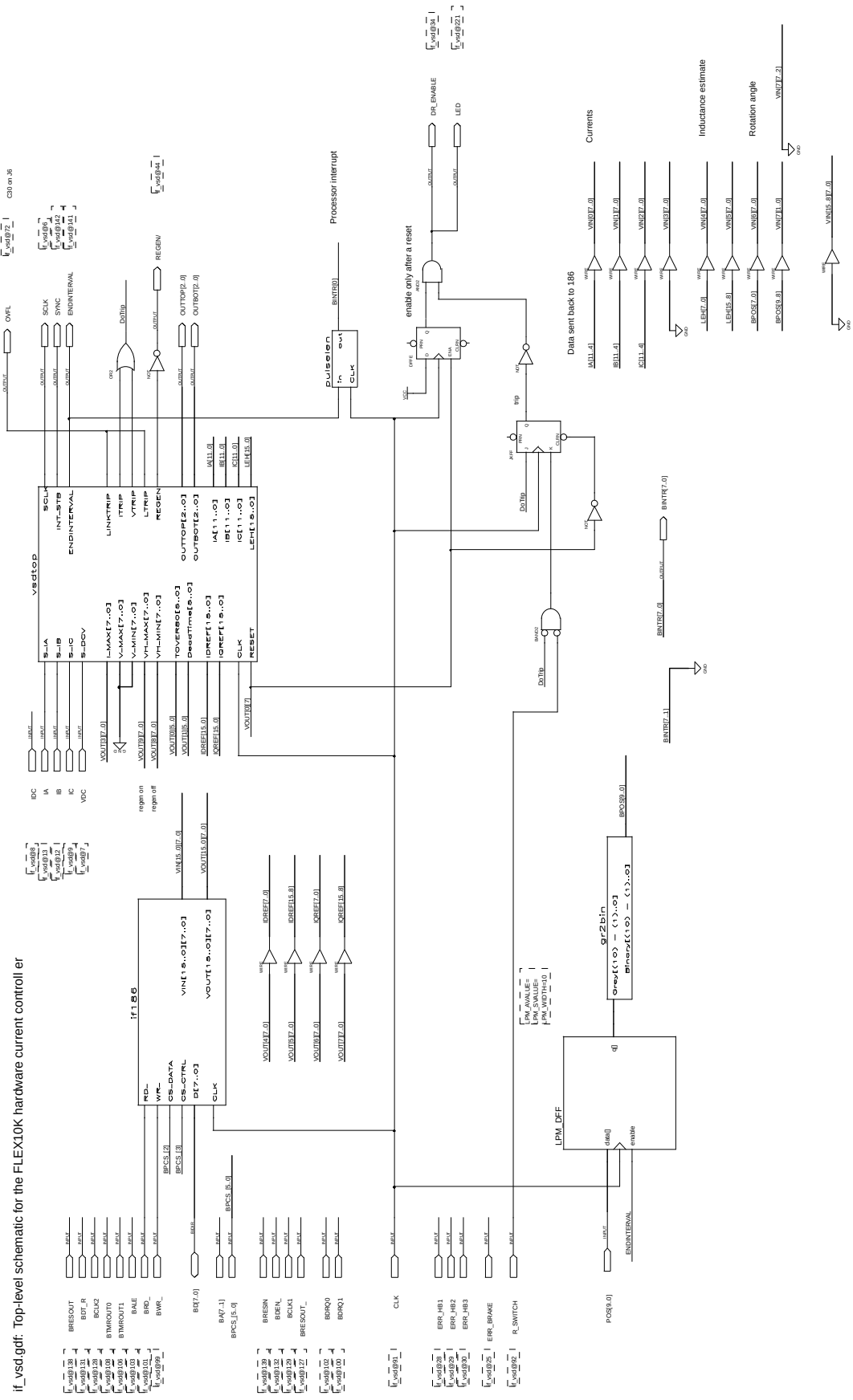
Input Module

- gr2bin
- lim_mux
- p_div
- rx_seq
- scompare
- ser_in

PWM Module

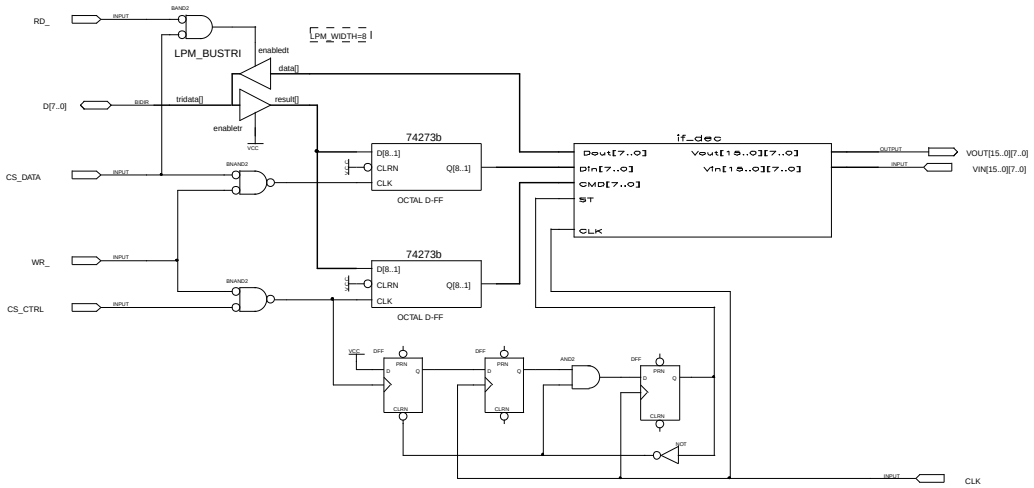
- dead
- dead_cmp
- dead_tme
- pwmst
- swgen

B.2.1 if_vsd.gdf

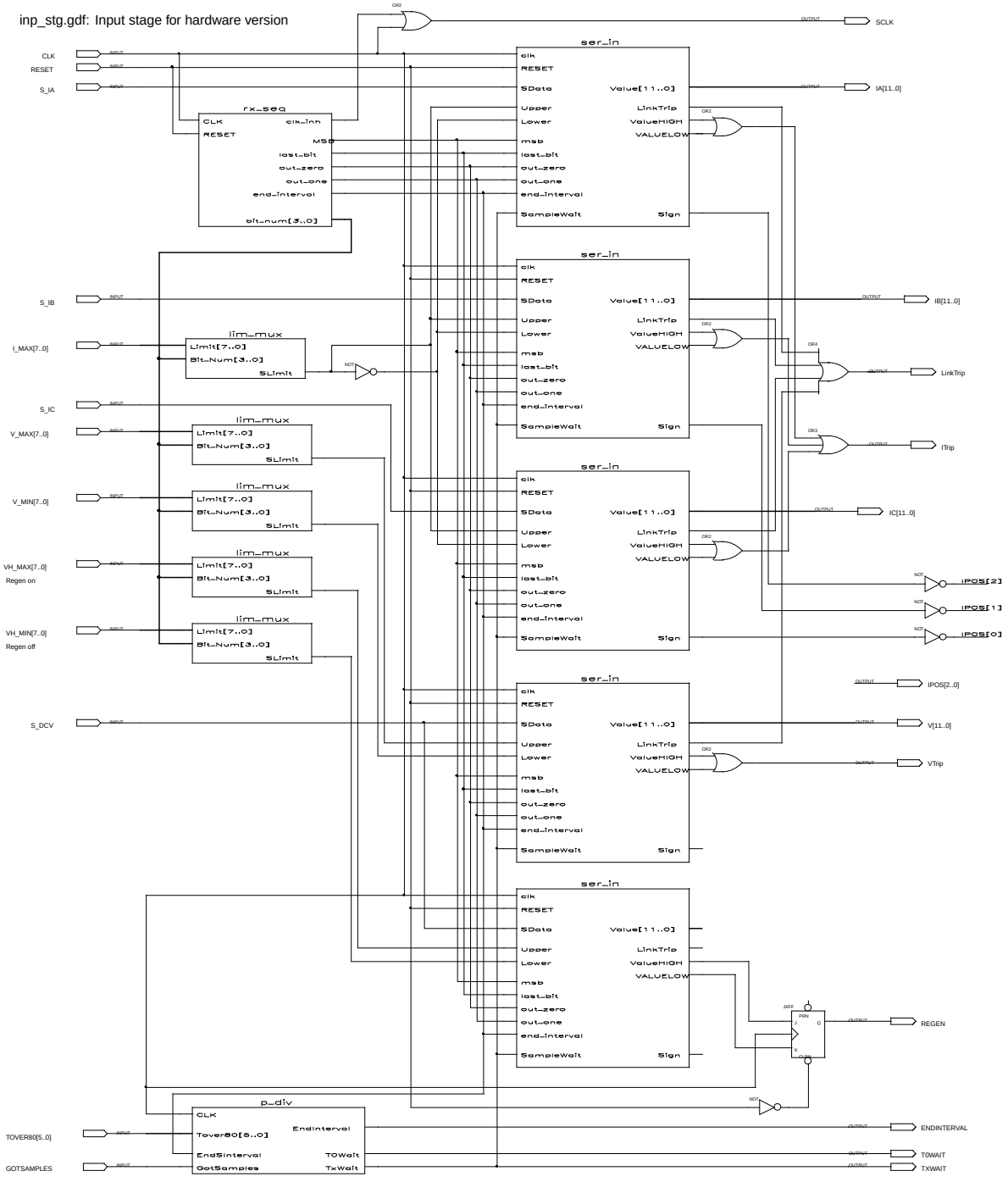


B.2.2 if186.gdf

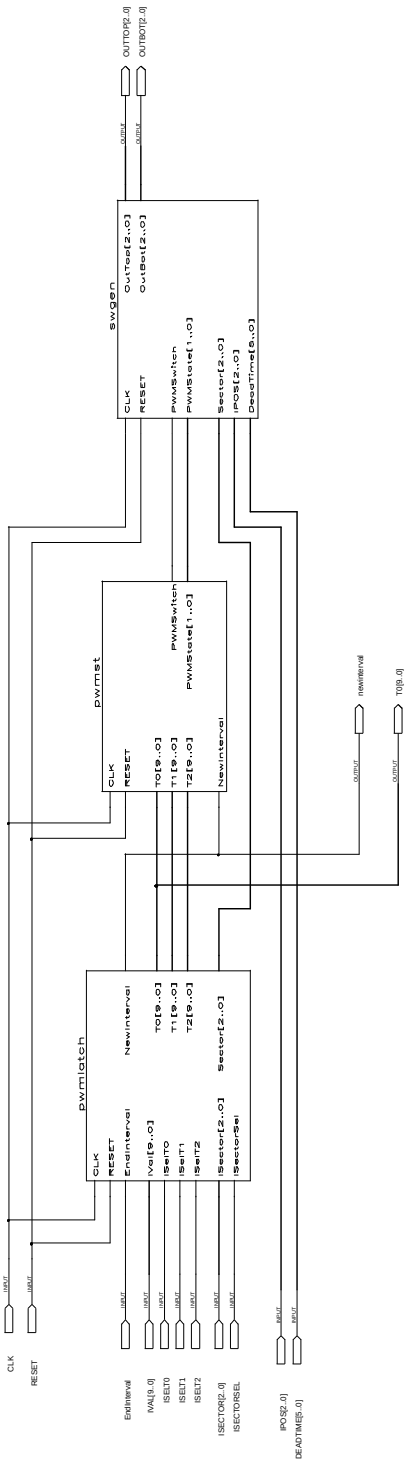
if186.gdf: interface to the 186 processor



B.2.3 inp_stg.gdf



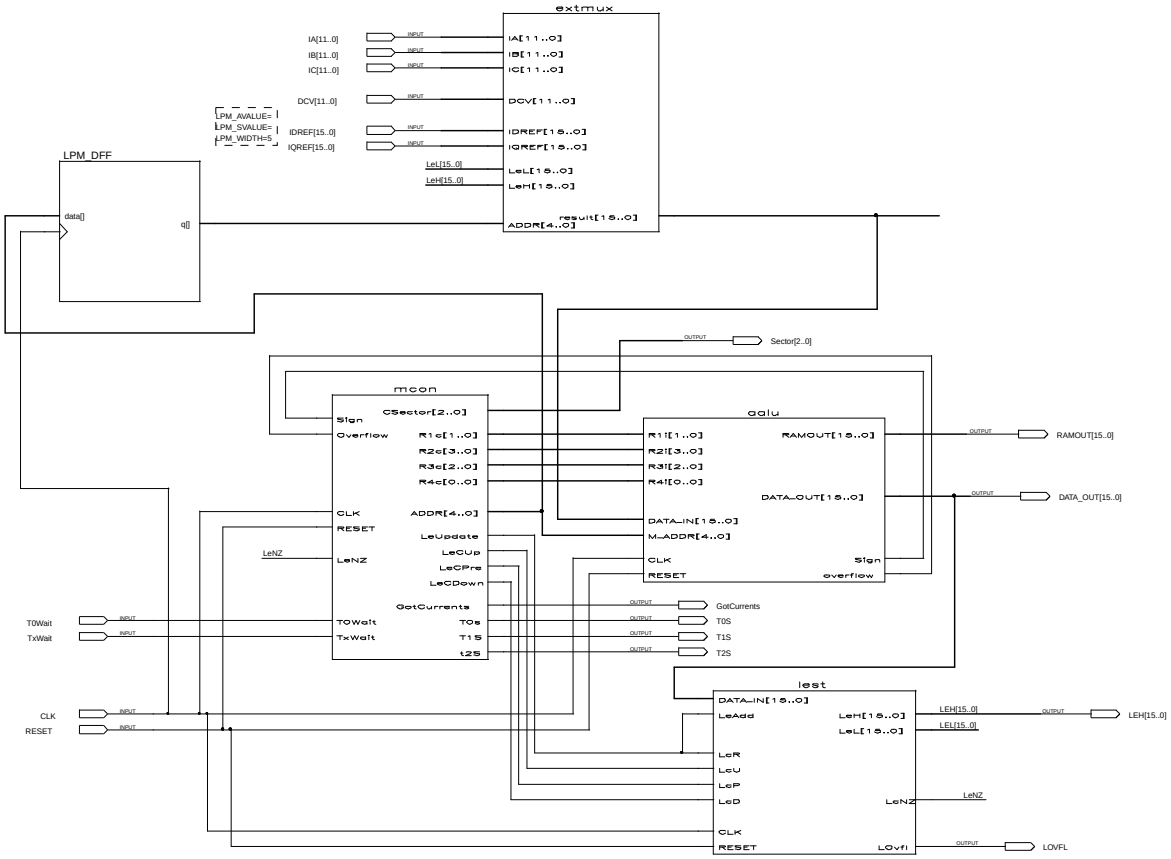
B.2.4 pwm.gdf



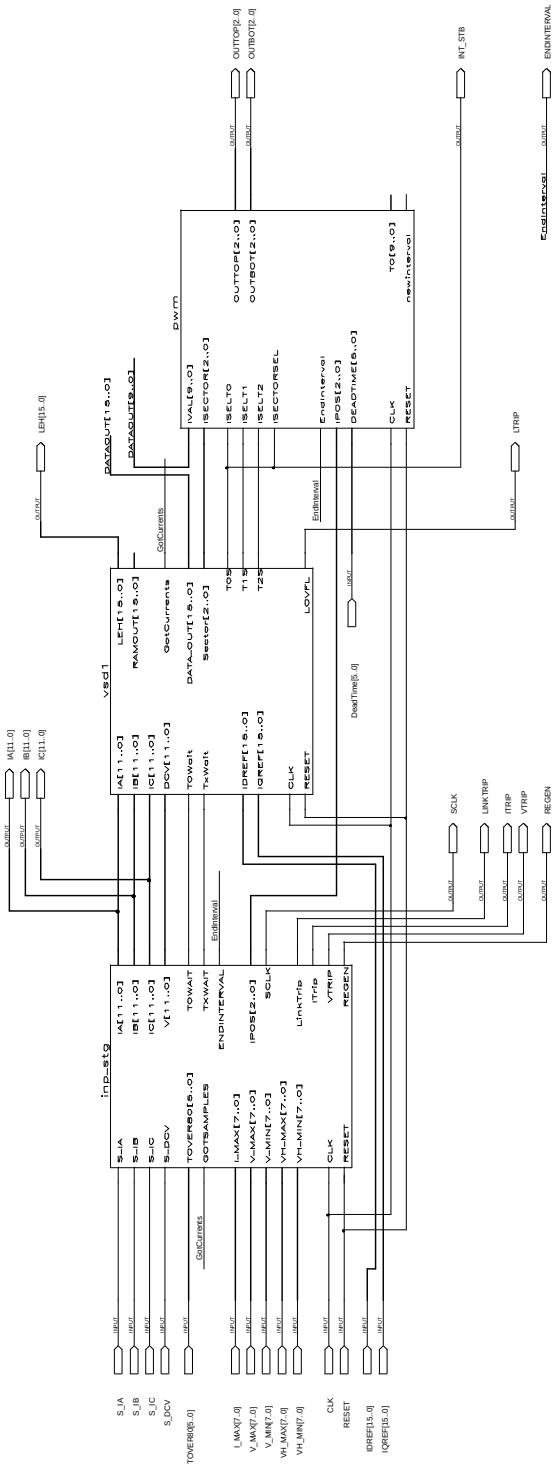
pwm.gdf: PWM generator

B.2.5 vsd1.gdf

vsd1.gdf: The ALU and Sequencer



B.2.6 vsdtop.gdf



vsdtop.gdf: The controller itself, without the interface

B.2.7 aalu.tdf

```
%============================================
aalu.tdf

The ahdl alu design
============================================%

TITLE "AHDL ALU";

INCLUDE "pwm_comm";
INCLUDE "lpm_ram_dq";
INCLUDE "lpm_add_sub";
INCLUDE "lpm_mux";
INCLUDE "aalu_h";


SUBDESIGN aalu
(
  CLK, RESET: INPUT;
  R1i[1..0],R2i[3..0],R3i[2..0],R4i[0..0]:    INPUT;
  DATA_IN[15..0]:    INPUT;
  M_ADDR[4..0]:      INPUT;
  DATA_OUT[15..0]:   OUTPUT;
  Sign, Overflow:    OUTPUT; --valid in R2 cycle

  RAMOUT[15..0]: OUTPUT;
)

VARIABLE

  ADDR[4..0]: DFF;
  R1[1..0],R2[3..0],R3[2..0],R4[0..0]: DFF;

  RG1[15..0], RG2[31..0], RG3[31..0]: DFF;
  InputSign:                          NODE;
  AddSrc1[15..0],AddSrc3[15..0]:      NODE;
  AddRes[15..0]:                      NODE;
  SHI[3..0][31..0],SHO[31..0]:        NODE;
  Booth[1..0]:                        NODE;
  SignDiff, SignDiffL, SignChg:       NODE;
  SignL:                              DFF;
  MUL1_R2[3..0]:     NODE;
  DIV1_R2[3..0]:     NODE;
  DIV3_R2[3..0]:     NODE;

  OverflowL:         DFF;

  MEM: lpm_ram_dq WITH (LPM_WIDTH=16, LPM_WIDTHAD=5,
       LPM_NUMWORDS=32, LPM_FILE="regs.mif",
       LPM_INDATA="UNREGISTERED",
       LPM_ADDRESS_CONTROL="REGISTERED",
       LPM_OUTDATA="UNREGISTERED" );
  ADDER: lpm_add_sub WITH (LPM_WIDTH=16,
       LPM_REPRESENTATION="SIGNED");
  SMUX: lpm_mux with(LPM_WIDTH=32, LPM_WIDTHS=2, LPM_SIZE=4);

BEGIN

  RAMOUT[15..0]=MEM.q[];


  -- multiplication node defines ----------------------------
  case (Booth[]) is
    when B"00" => MUL1_R2[]=R3_LD; -- just shift
    when B"11" => MUL1_R2[]=R3_LD; -- just shift
    when B"01" => MUL1_R2[]=R1R3_ADD; -- add and shift
    when B"10" => MUL1_R2[]=R1R3_SUB; -- sub and shift
  end case;

  -- division node defines ----------------------------------
  if (SignDiff) then
    DIV1_R2[]=R1R3_ADD;
  else
    DIV1_R2[]=R1R3_SUB;
  end if;

  if (SignChg) then
    DIV3_R2[]=DIV1_R2[];
  else
    DIV3_R2[]=R3_LD;
  end if;
%
  if (SignDiffl) then
    DIV4_R3[]=R2_SHL0;
  else
    DIV4_R3[]=R2_SHL1; -shift
  end if;
%


  --Generate Command Inputs--------------------------------

  --multiplex RXi[] for dual-processor
  R1[].clk=CLK; R2[].clk=CLK; R3[].clk=CLK; R4[].clk=CLK;
  ADDR[].clk=CLK; ADDR[].d=M_ADDR[];

  R1[].d=R1i[];
  --NOTE!!! REGISTER MEMORY USES R1i[]

  if !R2i[3] then
    R2[].d=R2i[];
  else
    case R2i[] is
      when R1R3_MUL1=> R2[].d=MUL1_R2[];
      when R1R3_DIV1=> R2[].d=DIV1_R2[];
      when R1R3_DIV3=> R2[].d=DIV3_R2[];
      when R1_ABS=> if InputSign then
        R2[].d=R1_LDN;
      else
        R2[].d=R1_LD;
      end if;
    end case;
  end if;

  R3[1..0].d=R3i[1..0];
  R3[2]=R3i[2]&!SignDiffl;   --assume that a shift in of 1 command means
                             --a divide operation
  R4[].d=R4i[];


  --R1 Inputs
  RG1[].clk=CLK;
  case R1[] is
    when MEM_RD => RG1[].d=MEM.q[];  InputSign=MEM.q[15];
    when EXT_RD => RG1[].d=DATA_IN[]; InputSign=DATA_IN[15];
    when R2_RD => RG1[].d=DATA_OUT[]; InputSign=DATA_OUT[15];
    when MEM_WR => RG1[].d=DATA_OUT[]; InputSign=DATA_OUT[15];
  end case;

  --Adder Inputs;
  AddSrc3[15..0]=RG3[31..16] & R2[2];
  AddSrc1[15..0]=RG1[15.. 0] & R2[1];

-- if R2[2] then AddSrc3[15..0]=RG3[31..16]; else AddSrc3[]=0; end if;
-- if R2[1] then AddSrc1[15..0]=RG1[15..0 ]; else AddSrc1[]=0; end if;

  --R2 Inputs (Adder);
  RG2[].clk=CLK;
%
  if R2[0] then                 -add
    AddRes[]=AddSrc1[]+AddSrc3[];
    Overflow=( AddSrc1[15]& AddSrc3[15]&!AddRes[15]) #
             (!AddSrc1[15]&!AddSrc3[15]& AddRes[15]);
  else                          -subtract
    AddRes[]=AddSrc1[]-AddSrc3[];
    Overflow=( AddSrc1[15]&!AddSrc3[15]&!AddRes[15]) #
             (!AddSrc1[15]& AddSrc3[15]& AddRes[15]);
  end if;
%
  ADDER.dataa[]=AddSrc3[];
  ADDER.datab[]=AddSrc1[];
  ADDER.add_sub=R2[0];
  ADDER.cin=!R2[0];


  OverflowL.d=ADDER.overflow;
  OverflowL.clk=CLK;
  Overflow=OverflowL.q;

-- Overflow=GND;
  AddRes[]=ADDER.result[];

  RG2[31..16].d=AddRes[];
  RG2[15..0].d=RG3[15..0];

  --Shifter
  SHI[0][]=RG2[].q;
  SHI[1][31..1]=RG2[30..0]; SHI[1][0]=R3[2];
  SHI[2][30..0]=RG2[31..1]; SHI[2][31]=RG2[31];
  SHI[3][15..0]=RG2[31..16]; --do sign extension
    SHI[3][31..16]=0; --RG2[31]; -better not to sign extend for multiply
  SMUX.data[][]=SHI[][];
  SMUX.sel[]=R3[1..0];
  SHO[]=SMUX.result[];


  --Booth multiplication bits - value available on R2 cycle.
  Booth[1]=SHO[0];
  if R3[]==R2_SHR then Booth[0]=RG2[0]; else Booth[0]=GND; end if;

  --division sign detect bits
  SignDiff=InputSign $ RG2[31].q;   --look-ahead xor of R1 and R3
                                    --valid in previous R2 cycle
  SignDiffL=RG1[15].q $ RG3[31].q; --valid in R3 cycle

  SignL.d=RG3[31];                  --Store previous sign of R3
```

```
  SignL.clk=CLK;
  SignChg=SignL.q $ RG2[31].q;      --true if sign od R3 has changed
                                    --due to last Add, valid in R2

  --Output
  if R4[0] then                                                          190
    DATA_OUT[]=RG2[31..16];
  else
    DATA_OUT[]=RG2[15..0];
  end if;
  Sign=RG2[31];                     --sign valid in R2 cycle

  --Register Memory
  MEM.data[]=DATA_OUT[];
  if R1i[]==MEM_WR then MEM.we=VCC; else MEM.we=GND; end if;
  MEM.address[]=M_ADDR[];                                                200
  MEM.inclock=CLK;

  --R3 Inputs
  RG3[].clk=CLK;
  RG3[].d=SHO[];

END;
```

B.2.8 extmux.tdf

```
%=============================================
extmux.tdf

the multiplexer to select external source data for the
ALU.
=============================================%
TITLE "External Multiplexer";

INCLUDE "lpm_mux";                                                       10

constant ES_IA=0;
constant ES_IB=1;
constant ES_IC=2;
constant ES_DCV=3;
constant ES_IDREF=4;
constant ES_IQREF=5;
constant ES_R4=6;
--constant ES_T=7;
--constant ES_Tminus2=8;
constant ES_DCVover4=9;                                                  20

constant ES_LeL=10;
constant ES_LeH=11;

SUBDESIGN extmux
(
  ADDR[4..0]:          INPUT;
  IA[11..0],IB[11..0],IC[11..0],DCV[11..0]: INPUT;
  IDREF[15..0], IQREF[15..0]: INPUT;                                     30

  LeH[15..0], LeL[15..0]: INPUT;

  Result[15..0]:       OUTPUT;
)

VARIABLE

  ISel: lpm_mux with(LPM_WIDTH=16, LPM_WIDTHS=4, LPM_SIZE=16);

BEGIN                                                                    40
  ISel.sel[]=Addr[3..0];

  ISel.data[ES_IA][11..0]=IA[]; ISel.data[ES_IA][15..12]=IA[11];
  ISel.data[ES_IB][11..0]=IB[]; ISel.data[ES_IB][15..12]=IB[11];
  ISel.data[ES_IC][11..0]=IC[]; ISel.data[ES_IC][15..12]=IC[11];

  --PATCH FOR LOW VOLTAGE FIX — VOLTAGE IS ALWAYS 1024
  ISel.data[ES_DCV][]=1024;

  ISel.data[ES_DCVover4][9..0]=DCV[11..2];                               50
  ISel.data[ES_DCVover4][15..10]=GND;

  ISel.data[ES_IDREF][]=IDREF[];
  ISel.data[ES_IQREF][]=IQREF[];

  ISel.data[ES_LeL][]=LeL[];
  ISel.data[ES_LeH][]=LeH[];

  Result[]=ISel.result[];
END;                                                                     60
```

B.2.9 if_dec.tdf

```
%=============================================
if_dec.tdf

interface between the 186 bus and the logic.  Offers
bidirectional latching
=============================================%

INCLUDE "lpm_mux";
INCLUDE "lpm_decode";
                                                                         10

%-----------------------------------------
Command types:
0000 0aaa: 186 writes to latch aaa
0001 0aaa: 186 reads input aaa into the temporary storage latch

-----------------------------------------%

SUBDESIGN IF_DEC_DEC
(                                                                        20
  CLK: INPUT;

  Din[7..0]:   INPUT;
  Dout[7..0]:  OUTPUT;

  CMD[7..0]:   INPUT;
  ST:          INPUT;

  Vout[15..0][7..0]:   OUTPUT;
  Vin[15..0][7..0]:    INPUT;                                            30
)


VARIABLE
  OMUX:        lpm_mux with (LPM_WIDTH=8,LPM_SIZE=16,
                             LPM_WIDTHS=4);
  OLTCH[7..0]: DFFE;

  IDEC:        lpm_decode with (LPM_WIDTH=4, LPM_DECODES=16);
  ILTCH[15..0][7..0]: DFFE;                                              40


BEGIN
  --186 Reads from Altera
  OMUX.sel[]=CMD[3..0];
  OMUX.data[][]=Vin[][];

  OLTCH[].clk=CLK;
  OLTCH[].ena=ST & CMD[4]; --write to latch
  OLTCH[].d=OMUX.result[];                                               50

  Dout[]=OLTCH[];

  --186 Writes to Altera
  IDEC.data[]=CMD[3..0];
  IDEC.enable=ST & !CMD[4];

  ILTCH[][].clk=CLK;
  FOR i IN 0 TO 15 GENERATE
    ILTCH[i][].d=Din[];        --watch this                              60
    ILTCH[i][].ena=IDEC.eq[i]; --and this
  END GENERATE;

  Vout[][]=ILTCH[][];

END;
```

B.2.10 lest.tdf

```
%============================================
lest.tdf

Inductance estimator logic
============================================%
TITLE "Inductance Estimator";

INCLUDE "lpm_add_sub";

SUBDESIGN Lest
(
  CLK, RESET: INPUT;

  DATA_IN[15..0]:        INPUT;
  LeAdd:                 INPUT;
  LeH[15..0],LeL[15..0]: OUTPUT;
  LOvfl:                 OUTPUT;

  LcR, LcU, LcP, LcD:    INPUT;
  LcNZ:                  OUTPUT;
)


VARIABLE
  Lc_upc[2..0], Lc_dnc[2..0]:  DFF;  --Up and down inductance counters;

  InLatch[15..0]:      DFF;
  L_Latch[26..0]:      DFF;
  DoAdd:               DFF;
  Adder: lpm_add_sub with
       (LPM_WIDTH=27, LPM_DIRECTION="ADD",
       LPM_REPRESENTATION="UNSIGNED");
BEGIN

  --input latch
  InLatch[].clk=CLK;
  InLatch[]=DATA_IN[];
  --InLatch[].ena=LeAdd;

  --ADD Pipeline latch
  DoAdd.clk=CLK;
  DoAdd=LeAdd;      --do add on cycle after preload

  --ADDER STUFF
  Adder.cin=gnd;
  Adder.dataa[]=L_Latch[];
  Adder.datab[15..0]=InLatch[]; Adder.datab[26..16]=InLatch[15];
  LOvfl=Adder.cout & DoAdd;

  --Inductance latch
  if RESET then
    L_Latch[11..0]=0; L_Latch[26..12]=200; --initial L
  elsif DoAdd then
    L_Latch[]=Adder.result[];
  else
    L_Latch[]=L_Latch[];
  end if;
  L_Latch[].clk=CLK;
  --L_Latch[].ena=DoAdd;
  LeH[15]=gnd;              LeH[14..0]=L_Latch[26..12];
  LeL[15..4]=L_Latch[11..0];  LeL[3..0]=0;


  --Up inductance counter
  Lc_upc[].clk=CLK;
  if LcR then
    Lc_upc[]=0;
  elsif LcU then
    Lc_upc[]=Lc_upc[]+1;
  else
    Lc_upc[]=Lc_upc[];
  end if;

  --Down inductance counter
  Lc_dnc[].clk=CLK;
  if LcP then
    Lc_dnc[]=Lc_upc[];
  elsif LcD then
    Lc_dnc[]=Lc_dnc[]-1;
  else
    Lc_dnc[]=Lc_dnc[];
  end if;
  --Output
  LcNZ=Lc_dnc[0] # Lc_dnc[1] # Lc_dnc[2];

END;
```

B.2.11 mcon.tdf

```
%============================================
mcon.tdf

The microcode sequencer. Reads values from the microcode
ROM, and generates commands
============================================%
TITLE "Microcode Controller";

INCLUDE "lpm_rom";
INCLUDE "lpm_mux";
INCLUDE "aalu_h.inc";

CONSTANT Br0=0;
CONSTANT BrSign=1;
CONSTANT BrOverflow=2;
CONSTANT BrCountNZ=3;
CONSTANT BrLeNZ=4;
constant BrT0Wait=5;
constant BrTxWait=6;
CONSTANT Br1=7;

constant COM_preload29=1;
constant COM_preload59=2;

constant COM_T0s=4;
constant COM_T1s=5;
constant COM_T2s=6;
constant COM_GotCurrents=7;

constant COM_Sector2Write=8;
constant COM_Sector1Write=9;
constant COM_Sector0Write=10;

constant COM_ClearOverflow=11;

constant COM_LeUpdate=12;
constant COM_LeCUp=13;
constant COM_LeCPre=14;
constant COM_LeCDown=15;

--constant COM_T1Pre1=12;
--constant COM_T1Pre2=13;
--constant COM_T2Pre1=14;
--constant COM_T2Pre2=15;


SUBDESIGN mcon
(
  CLK, RESET: INPUT;
  Sign, Overflow:      INPUT;

  LeNZ:INPUT;
  R1c[1..0],R2c[3..0],R3c[2..0],R4c[0..0]:   OUTPUT;
  ADDR[4..0]:          OUTPUT;

  CSector[2..0]: OUTPUT;
  LeUpdate,LeCUp,LeCPre,LeCDown: OUTPUT;

  T0Wait, TxWait:      INPUT;
  GotCurrents:         OUTPUT;
  T0s,T1s,T2s:         OUTPUT;
)

VARIABLE
  SM1: machine with states (Word1,Word1Pre,Word1_2,Word2);
  ROM: lpm_rom WITH (LPM_WIDTH=16,LPM_WIDTHAD=9,
               LPM_FILE="ccprog.mif",
               LPM_ADDRESS_CONTROL="unregistered",
               LPM_OUTDATA="unregistered");

  R2comm[3..0]:        DFF;
  IAddr[8..0]:         NODE;
  AddrLatch[8..0]:     DFF;
  BrCond: lpm_mux with(LPM_WIDTH=1, LPM_WIDTHS=3, LPM_SIZE=8);

  count[5..0]:         DFF;
  load_count:          NODE;
  preload[5..0]:       NODE;
  Sector[2..0]:        DFFE;

  Ovfl:                DFFE;
  NewOvfl:             NODE;
  ResetOvfl:           NODE;

  --prepared instructions
  --R1cP[1..0],R2cP[3..0],R3cP[2..0],R4cP[0..0]:    DFF;

  --UsePrepared:              NODE;

BEGIN
```

```
DEFAULTS
  preload[]=B"XXXXXX";
  load_count=0;
  Sector[].ena=GND;

  LeUpdate=gnd; LeCUp=gnd; LeCPre=gnd; LeCDown=gnd;
  T0s=gnd; T1s=gnd; T2s=gnd;
  GotCurrents=GND;

  Ovfl.ena=GND;
  ResetOvfl=GND;
END DEFAULTS;


-- DEBUGGING
CSector[]=Sector[];


-- counter stuff ---------------------------------
count[].clk=CLK;
if load_count then
  count[].d = preload[];
else
  count[].d = count[].q-1;
end if;

-- Sector Stuff ----------------------------------
Sector[].clk=CLK;
Sector[].d=Sign;

-- State machine
SM1.clk=CLK;
SM1.reset=RESET;

--Memory latch
AddrLatch[].d=IAddr[];
AddrLatch[].clk=CLK;
AddrLatch[].clrn=!RESET;

--Memory
ROM.MEMENAB=VCC;
ROM.address[]=AddrLatch[]; --IAddr[];
--ROM.outclock=CLK;

--R2 output
R2comm[].clk=CLK;
R2c[]=R2comm[].q;


--Branch Conditions
BrCond.sel[]=ROM.q[6..4];
BrCond.data[Br0][0]=GND;                    --no jump condition
BrCond.data[BrSign][0]=Sign;
BrCond.data[BrOverflow][0]=NewOvfl;
BrCond.data[BrCountNZ][0]=(count[]!=0);
BrCond.data[BrLeNZ][0]=LeNZ;
BrCond.data[BrT0Wait][0]=T0Wait;
BrCond.data[BrTxWait][0]=TxWait;
BrCond.data[Br1][0]=VCC;

--Prepared Instructions
-- R1cP[1..0].clk=CLK;
-- R2cP[3..0].clk=CLK;
-- R3cP[2..0].clk=CLK;
-- R4cP[0..0].clk=CLK;


Addr[]=ROM.q[14..10];

--Overflow
Ovfl.clk=CLK;
NewOvfl=(Ovfl#Overflow)&!ResetOvfl;
Ovfl.d=NewOvfl;

--state machine
case SM1 is
  when Word1 =>   --first word
    R1c[]=ROM.q[1..0];
    R2comm[].d=ROM.q[5..2];
    R3c[]=ROM.q[8..6];
    R4c[]=ROM.q[9];

    IAddr[]=AddrLatch[].q+1; --read next address
    if ROM.q[15] then SM1=Word2; else SM1=Word1_2; end if;


  when Word1_2 =>          --2nd clock of 1 word sequence
    Ovfl.ena=VCC;          --latch in the overflow
    IAddr[]=AddrLatch[].q; --retain the same address
    SM1=Word1;

  when Word2 =>
    Ovfl.ena =VCC;         --latch in the overflow
    if BrCond.result[] then
      IAddr[]=ROM.q[15..7]; --branch
    else
      IAddr[]=AddrLatch[].q+1;    --else increment
    end if;
    --SM1=Word1; is the default
    case ROM.q[3..0] is
      when COM_preload29 => preload[]=29; load_count=VCC;
      when COM_preload59 => preload[]=59; load_count=VCC;
      when COM_Sector2Write => Sector[2].ena=VCC;
      when COM_Sector1Write => Sector[1].ena=VCC;
      when COM_Sector0Write => Sector[0].ena=VCC;

      when COM_ClearOverflow => ResetOvfl=VCC;

      when COM_LeUpdate => LeUpdate=VCC;
      when COM_LeCUp => LeCUp=VCC;
      when COM_LeCPre => LeCPre=VCC;
      when COM_LeCDown => LeCDown=VCC;

      when COM_GotCurrents=>GotCurrents=VCC;
      when COM_T0s=>T0s=vcc;
      when COM_T1s=>T1s=vcc;
      when COM_T2s=>T2s=vcc;


      end case;

    SM1=Word1;

  end case;
END;
```

B.2.12 pulselen.tdf

```
%=============================================
pulselen.tdf

simple pulse lengthenging.
output goes high for 8 cycles when the input goes high
=============================================%


subdesign PulseLen
(
  CLK, in:      INPUT;
  out:          OUTPUT;
)


VARIABLE
  Ctr[2..0]:    DFF;
BEGIN
  Ctr[].clk=CLK;

  if in then
    Ctr[]=7;
  else
    if Ctr[]==0 then
      Ctr[]=0;
    else
      Ctr[]=Ctr[]-1;
    end if;
  end if;

  out=!(Ctr[]==0) # in;
END;
```

B.2.13 pwmlatch.tdf

```
%=============================================
pwmlatch.tdf

Interface between the ALU and PWM generator. Takes the Tx values off a
common bus and latches them for the next PWM cycle. There are two sets of
latches so that the values can be held across the whole cycle.
=============================================%
TITLE "PWM value latch";

INCLUDE "pwm_comm";

SUBDESIGN pwmlatch
(
  CLK, RESET:                INPUT;
  IVal[9..0]:                INPUT;
  %IAddr[IAddrLen..0], ISel: INPUT;%
  ISelT0,ISelT1,ISelT2:      INPUT;        --active high selects
  ISector[2..0], ISectorSel: INPUT;
  EndIntervalInterval:       INPUT;

  NewInterval: OUTPUT;
  T0[9..0], T1[9..0], T2[9..0], Sector [2..0]: OUTPUT;
)


VARIABLE
  Interval: DFF;        --flip-flop to delay the EndInterval signal
                        --by one clock
  L1Sect[2..0], L2Sect[2..0]:          DFFE;
  L1T[2..0][9..0], L2T[2..0][9..0]:    DFFE;

BEGIN
  DEFAULTS
    L1T[][].ena=GND;
    L1Sect[].ena=GND;
  END DEFAULTS;

  --NewInterval logic
  Interval.clk=CLK;
  Interval.clrn=!RESET;
  Interval.d=EndInterval;
  NewInterval=Interval.q;

  --INPUT LATCHES
  L1Sect[].clk=CLK;
  L1Sect[].ena=ISectorSel;
  L1Sect[].d=ISector[];

  L1T[][].clk=CLK;
  L1T[0][].d=IVal[]; L1T[1][].d=IVal[]; L1T[2][].d=IVal[];
%
  if ISel then case IAddr[] is
    when IA_T0 => L1T[0][].ena=VCC;
    when IA_T1 => L1T[1][].ena=VCC;
    when IA_T2 => L1T[2][].ena=VCC;
  end case; end if;
%
  if ISelT0 then L1T[0][].ena=VCC; end if;
  if ISelT1 then L1T[1][].ena=VCC; end if;
  if ISelT2 then L1T[2][].ena=VCC; end if;

  --OUTPUT LATAHES
  L2Sect[].clk=CLK;
  L2Sect[].ena=EndInterval;
  L2Sect[].d=L1Sect[].q;

  L2T[][].clk=CLK;
  L2T[][].ena=EndInterval;
  L2T[][].d=L1T[][].q;

  --OUTPUTS
  Sector[]=L2Sect[].q;
  T0[]=L2T[0][].q; T1[]=L2T[1][].q; T2[]=L2T[2][].q;
END;
```

B.3 Microcode Source Code

The control algorithm itself is described in a microcode. This section contains that microcode in a source format. The complier in the next section is used to convert the source into the final binary code.

B.3.1 ccprog.vmc

```
//=============================================
// ccprog.vmc
//
// Source code for the variable speed drive current controller.
//=============================================


//------------
//External Sources
//------------
constant ES_IA=0;
constant ES_IB=1;
constant ES_IC=2;
constant ES_DCV=3;
constant ES_IDREF=4;
constant ES_IQREF=5;
constant ES_R4=6;
//constant ES_T=7;
//constant ES_Tminus2=8;
constant ES_DCVover4=9;

constant ES_LeL=10;
constant ES_LeH=11;

//------------
//MEMORY ADDRESSES
//------------
//direct axis currents
int ID0L=0;             // t=-1
int ID2L=0;             // t=-0.5
int ID0=0;              // t= 0
int ID4=0;              // t= 0.25
int ID2=0;              // t= 0.5
//quadrature axis currents
int IQ0L=0;             // t=-1
int IQ2L=0;             // t=-0.5
int IQ0=0;              // t=0
int IQ2=0;              // t=0.5
//reference currents
int IDR=0;              //direct*3
int IQR=0;              //quadrature*sqrt(3)

int L_EST=873;          //inductance estimate -
                        //L/6V*2^(16-k) where k is stored in
                        //the inductance counter
int alphaD=0;
int alphaQ=0;
int alphaDL=0;
int alphaQL=0;
int ABSalphaD;
int ABSalphaQ;
int T0;                 //PWM times
int T1=453;
int T2=789;
int MR_R1;              //GP reg
int MaxPreAlpha=16383;
int Tminus2=638;        // T/2-2   510
int T=640;              // T/2     512
int vtwo=2;             // 2
int Root3over4=28378; //used to calc reference
                        //iq*sqrt(3) (not implemented)

//------------
//Branch Conditions
//------------
constant Br0=0;
constant BrSign=1;
constant BrOverflow=2;
constant BrCountNZ=3;
constant BrLeNZ=4;
constant BrT0Wait=5;
constant BrTxWait=6;
constant Br1=7;

//------------
//Commands
```

```
//————————
//commands 1 and 2 are reserved for multiply
// and divide counter preload
constant COM_test0=4;
constant COM_test1=5;
constant COM_test2=6;

constant COM_GotCurrents=7; //!!!

constant COM_Sector2Write=8;
constant COM_Sector1Write=9;
constant COM_Sector0Write=10;

constant COM_ClearOverflow=11;

constant COM_LeUpdate=12;
constant COM_LeCUp=13;
constant COM_LeCPre=14;
constant COM_LeCDown=15;


//Note: units of T are not explicitly specified.  The only
//points where this is important are the initial values of
//the inductance and in the values of T, Tminus2 etc.  The
//largest value of T usable with this arithmetic is 2048 —
//to prevent overflow.


  //————————————————————
  // WAIT FOR T=0
  // Read In Currents at T=0
  // and do 3-phase to 2-phase converstion of currents
  //————————————————————
LABEL WaitT0;
  if BrT0Wait goto WaitT0;
  a=ID0; ID0L=ah; //store last ID0 as ID0L
  a=IQ0; IQ0L=ah; //store last IQ0 as IQ0L

  //IQ0=ES_IB-ES_IC
  //————————
  a=ES_IB;
  a=a−ES_IC;
  IQ0=ah;

  //ID0=2*ES_IA-ES_IB-ES_IC;
  //————————————
  a=ZERO;
  a=a>>sixteen+ES_IA;
  a=a*two−ES_IB;
  a=a−ES_IC;
  ID0=ah;
  COM_GotCurrents;

  //calc the L/3 estimate
  // = min(t1,t2)*V/2/3/abs(delta i1-delta i2)
  //————————————————————
  //any overflow will be trapped at the end
  //and will cause the inductance estimate
  //to be ignored.
  a=ID4;
  a=a*two−ID0L;
  a=a−ID2;
  a=abs(ah);
  MR_R1=ah;          //store abs \delta i_1-\delta i_2 in MR_R1
  a=a*two+MR_R1;
  MR_R1=ah;          //3*abs(...)

  a=T1;              //get min (T1,T2)
  a=a−T2;
  if BrSign goto HaveTx; a=T1;
  a=T2;
LABEL HaveTx;
  a=a>>sixteen*ES_DCV*two; //get 2TxV
  a=a/two;                 //a=TxV
  a=a−MR_R1;               //check if the result is usable
  if BrSign goto LeSizeOK; a=a+MR_R1;
  if Br1 goto LUpdateOvfl;
LABEL LeSizeOK;
  a=a/MR_R1/two;

  if BrOverflow goto LUpdateOvfl;
                     //dont update if there was an overflow
                     //sub instr cannot ovfl as it is +ve - +ve
  a=al;              //now have TxV/2abs(...) = L/3V

  if Br1 goto LUpdate; //sub off existing estimate to give
   a=a−ES_LeH;       //amount to add to inductance estimate

LABEL LUpdateOvfl;
  COM_ClearOverflow;
  a=ZERO;            //do an update of zero to reset the counter
                     //in the event of an overflow.
LABEL LUpdate;
  out(ah);
  COM_LeUpdate;      //update the inductance

  //————————————————————
  // WAIT FOR T=0.25
  // Read In Currents at T=0.25
  // and do 3-phase to 2-phase converstion of currents
  // (Direct axis only)
  //————————————————————
  //ID4=2*ES_IA-ES_IB-ES_IC;
  //————————————
  a=ZERO;
LABEL WaitT4;
  if BrTxWait goto WaitT4;
  a=a>>sixteen+ES_IA;
  a=a*two−ES_IB;
  a=a−ES_IC;
  ID4=ah;
  COM_GotCurrents;

  //Find L/6V
  //————
  a=ES_LeL;          //load L/3
  a=a>>sixteen+ES_LeH;

  //to make the alphas bigger for more precision,
  //could add COM_LeCUp commands here

LABEL LVLoop;        //make sure division will not overflow
  a=a−ES_DCV;
  if BrSign goto LVLoopEnd;
  a=a+ES_DCV;
  COM_LeCUp;
  if Br1 goto LVLoop;
  a=a/two;

LABEL LVLoopEnd;
  a=a/ES_DCV/two;    //have L/6V
  L_EST=al;


  a=ID2; ID2L=ah; //store last ID2 as ID2L
  a=IQ2; IQ2L=ah; //store last IQ2 as IQ2L

  //————————————————————
  // WAIT FOR T=0.5
  // Read In Currents at T=0.5
  // and do 3-phase to 2-phase converstion of currents
  //————————————————————
LABEL WaitT2;
  if BrTxWait goto WaitT2;
  a=ES_IDREF; IDR=ah;
  a=ES_IQREF; IQR=ah;

  //IQ2=ES_IB-ES_IC
  //————————
  a=ES_IB;
  a=a−ES_IC;
  IQ2=ah;

  //ID2=2*ES_IA-ES_IB-ES_IC;
  //————————————
  a=ZERO;
  a=a>>sixteen+ES_IA;
  a=a*two−ES_IB;
  a=a−ES_IC;
  ID2=ah;
  COM_GotCurrents;

  //————————————————
  // CALCULATE DIRECT AXIS ALPHA
  //————————————————
  //alphaD=2*alphaD(L)-alphaDL(L)+
  //   2*Lover6V*(IDR-4*ID2+3*ID0+ID2L-ID0L)
  //alphaDL=alphaD(L)
  a=−ah;
  a=a*two+ID0;
  a=a*two+ID0;
  a=a+ID2;
  a=a−ID0;
  a=a+IDR;
  ABSalphaD=ah;      //temp store this result so that the
                     //sign of it may be used if there is
                     //an overflow in calculating alpha

  COM_LeCPre;        //preload inductance counter

  a=a>>sixteen*L_EST*two; //multiply by L/3V

  COM_LeCDown;       //count down after branch
  if BrLeNZ goto LCLoopD; //do shifting if necessary
  if Br1 goto LCLoopEndD;


LABEL AD_MagError;
```

```
//-------------------------------------------
  //Magnitude Error in product
  //so just give the maximium allowable magnitude instead of the
  //multiplication result
  COM_ClearOverflow;
  a=ABSalphaD;
  if BrSign goto AD_Store;   a=-MaxPreAlpha;
  if Br1    goto AD_Store;   a= MaxPreAlpha;
//-------------------------------------------

  //loop to do 2^n adjustment of the product
LABEL LCLoopD;              //multiply by two until count is zero
  COM_LeCDown;              //this loop must be one instruction!!!
  if BrLeNZ goto LCLoopD;
  a=a*two;

LABEL LCLoopEndD;

  //combine with previous alphas
  a=a-alphaD;               //sub off alpha(t-2)
  MR_R1=ah;                 //store result
  a=ZERO;
  a=a>>sixteen+alphaD;      //before updating alphaDL
  alphaDL=ah;
  a=a*two+MR_R1;            //then add 2*last

LABEL AD_Store;
  //Write the new desired alphaD
  alphaD=ah;
  COM_Sector2Write;
  if  BrOverflow goto AD_MagError; //if there was overflow, fix it

  a=abs(ah);                //store the absolute value of alphaD
  ABSalphaD=ah;

LABEL CalcAQ;
  //-----------------
  // CALCULATE QUADRATURE AXIS ALPHA
  //-----------------
  //alphaQ=2*alphaQ(L)-alphaQL(L)+
  //   L/3V*(IQR-4*IQ2+3*IQ0+IQ2L-IQ0L)
  //alphaQL=alphaQ(L);
  a=ZERO;                   //purge lower 16 bits
  a=a>>sixteen-IQ2;
  a=a*two+IQ0;
  a=a*two+IQ0;
  a=a+IQ2;
  a=a-IQ0;
  a=a+IQR;
  ABSalphaQ=ah;             //temp store this result so that the
                            //sign of it may be used if there is
                            //an overflow in calculating alpha

  COM_LeCPre;               //preload inductance counter

  a=a>>sixteen*L_EST*two; //multiply by 2L/3V

  COM_LeCDown;              //count down after branch
  if BrLeNZ goto LCLoopQ;   //do shifting if necessary
  if Br1 goto LCLoopEndQ;


LABEL AQ_MagError;
//-------------------------------------------
  //Magnitude Error in product
  //so just give the maximium allowable magnitude instead of the
  //multiplication result
  COM_ClearOverflow;
  a=ABSalphaQ;
  if BrSign goto AQ_Store;   a=-MaxPreAlpha;
  if Br1    goto AQ_Store;   a= MaxPreAlpha;
//-------------------------------------------

  //loop to do 2^n adjustment of the product
LABEL LCLoopQ;              //multiply by two until count is zero
  COM_LeCDown;              //this loop must be one instruction!!!
  if BrLeNZ goto LCLoopQ;
  a=a*two;

LABEL LCLoopEndQ;

  //combine with previous alphas
  a=a-alphaQ;               //sub off alpha(t-2)
  MR_R1=ah;                 //store result
  a=ZERO;
  a=a>>sixteen+alphaQ;      //before updating alphaQL
  alphaQL=ah;
  a=a*two+MR_R1;            //then add 2*last

LABEL AQ_Store;
  //Write the new desired alphaQ
  alphaQ=ah;
  COM_Sector1Write;
  if  BrOverflow goto AQ_MagError; //if there was overflow, fix it
  a=abs(ah);                //store the absolute value of alphaQ
  ABSalphaQ=ah;
  a=a-ABSalphaD;            //get abs(aQ)-abs(aD)
  COM_Sector0Write;

//-------------------------------------------
// Calculate T1,T2 based on the sector
//-------------------------------------------
  if BrSign goto Sect1346;
  a=alphaD;     //load aD for the comparison in Sect1346

//THE MIDDLE SECTORS 2,5
//- T1=-aD+abs(aQ) T2=aD+abs(aQ)
LABEL Sect25;
  a=a+ABSalphaQ;
  T2=ah;
  a=ABSalphaQ;
  a=a-alphaD;
  T1=ah;
  if Br1 goto FindT0;

LABEL Sect1346;
  //check sign of aD loaded after delayed branch
  if BrSign goto Sect34;
  a=ABSalphaQ;

//THE RIGHT SECTORS 1,6
//- T1=aD-abs(aQ) T2=2*abs(aQ)
LABEL Sect16;
  a=a*two;
  T2=ah;
  a=alphaD;
  a=a-ABSalphaQ;
  T1=ah;
  if Br1 goto FindT0;

//THE LEFT SECTORS 3,4
//- T1=2*abs(aQ) T2=-aD-abs(aQ)
LABEL Sect34;
  a=a*two;
  T1=ah;
  a=-alphaD;
  a=a-ABSalphaQ;
  T2=ah;

//-------------------------------------------
LABEL FindT0;
//-------------------------------------------
  a=Tminus2;
  a=a-T1;
  a=a-T2;
  if BrSign goto DoClip;
  a=a+vtwo;
  T0=ah;
  if Br1 goto OutputTx;     //need T0 in ah
  a=a;

LABEL DoClip;
//-------------------------------------------
//T1+T2 is too large - clipping required
//-------------------------------------------
  //T0=2^16*(t1+t2)/T/2;
  a=T1;                     //MESSY
  a=a+T2;
  T0=ah;
  a=Tminus2;
  a=a/T0/two;
  T0=al;

//  a=a>>16*alphaD*two;     //other method
//  a=a/Tminus2/two;
//  alphaD=al;

  //now scale the alphas by this value
  a=al;
  a=a>>sixteen*alphaD*two;
  alphaD=ah;
  COM_Sector2Write;

  a=abs(ah);                //store absolute value also
  ABSalphaD=ah;
  a=T0;
  a=a>>sixteen*alphaQ*two;
  alphaQ=ah;
  COM_Sector1Write;

    //do last bit of sector determination by
    //storing sign of abs(aD)-abs(aQ)
    //in sector[0]
  a=abs(ah);
  ABSalphaQ=ah;
  a=a-ABSalphaD;
  COM_Sector0Write;
```

```
//sector is re-determined as I don't know that
//the mul and div routines will
//always give the same sector - but they may...

//-------------------------------------------------
// Calculate Clipped T1,T2 based on the sector
//-------------------------------------------------
  if BrSign goto cSect1346;
  a=alphaD;              //load aD for the comparison in Sect1346

LABEL cSect25;
  if Br1 goto HaveT2;    //aD laoded
  a=a+ABSalphaQ;

LABEL cSect1346;
  if BrSign goto cSect34;  //check sign of aD
  a=ABSalphaQ;

LABEL cSect16;
  if Br1 goto HaveT2;    //abs(aQ) loaded
  a=a+ABSalphaQ;

LABEL cSect34;           //abs(aQ) loaded
  a=-ah;
  if Br1 goto HaveT2;
  a=a-alphaD;

LABEL HaveT2;
  T2=ah;
  if BrSign goto T2isZero;
  a=-ah;
  a=a+Tminus2;
  if BrSign goto T1isZero;
  T1=ah;
  if Br1 goto cFindT0;

LABEL T2isZero;
  a=ZERO;
  T2=ah;
  a=Tminus2;
  T1=ah;
  if Br1 goto cFindT0;

LABEL T1isZero;
  a=ZERO;
  T1=ah;
  a=Tminus2;
  T2=ah;

LABEL cFindT0;
  a=vtwo;
  T0=ah;

LABEL OutputTx; //assume T0 is in ah
  out(ah);
  COM_test0;
  a=T1;
  out(ah);
  COM_test1;
  a=T2;
  out(ah);
  COM_test2;

  if Br1 goto WaitT0;
  a=T0;

  END;
```

B.4 The Microcode Compiler

B.4.1 vsdc.cpp

```
/*=================================================
vsdc1_u.cpp

updated microcode compiler
=================================================*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

//Stage Defines
#define ST_Label      0
#define ST_Write      1
#define ST_Command    2
#define ST_Branch     3
#define ST_R2Expr     4

//note: if an R2 expression follows a branch, that R2 expression will
//be executed regardless of whether the branch is taken or not

#define NumFields 8
//Instruction field codes
#define IF_R1  0
#define IF_R2  1
#define IF_R3  2
#define IF_R4  3
#define IF_Addr 4
#define IF_BrAddr     5
#define IF_BrCond     6
#define IF_Command    7

#define CONSTANt int

#define COM_preload29 1      //prelaod the counter to 31
#define COM_preload59 2

#define BR_countnz    3

FILE *FIn,*FOut,*FReg;


//R1 CONSTANTS
CONSTANt MEM_RD= 1;//B"01";
CONSTANt EXT_RD= 0;//B"00";
CONSTANt MEM_WR= 3;//B"11";
CONSTANt R2_RD=  2;

//R2 CONSTANTS
CONSTANt R3_LD=  4;//B"010X";
CONSTANt R1_LD=  3;//B"0011";
CONSTANt R1_LDN= 2;//B"0010";
CONSTANt R1R3_ADD= 7;//B"0111";
CONSTANt R1R3_SUB= 6;//B"0110";
CONSTANt ZERO_LD= 0;//B"000X";

CONSTANt R1R3_MUL1= 8;//B"1000";
CONSTANt R1R3_DIV1= 9;//B"1001";
CONSTANt R1R3_DIV3= 10;//B"1010";
CONSTANt R1_ABS= 11;

//R3 CONSTANTS
CONSTANt R2_LD=  0;//B"X00";
CONSTANt R2_SHL0= 1;//B"001";
CONSTANt R2_SHL1= 5;//B"101";          //internal use only
CONSTANt R2_SHR= 2;//B"X10";
CONSTANt R2_SHW= 3;//B"X11";
CONSTANt R2_DIV= R2_SHL1;      //treat shift in of 1 as a divide op

//R4 CONSTANTS
//CONSTANt MEM_WRH=B"11";
//CONSTANt MEM_WRL=B"10";
CONSTANt OUTH= 1;//B"1";
CONSTANt OUTL= 0;//B"0";


/*=================================================
Register Initialisation
=================================================*/
#define REG_LEN 32
int RegInit[REG_LEN];


void InitReg(int Reg,int Val)
{
  if (Reg<0 || Reg>=REG_LEN)
    printf("Register Number out of range\n");
  else
    RegInit[Reg]=Val;
}

void WriteRegs(void)
{
  int i;
  fprintf(FReg,"WIDTH=16;\nDEPTH=32;\nADDRESS_RADIX=HEX;\
```

```
\nDATA_RADIX=HEX;\n");
  fprintf(FReg,"\nCONTENT BEGIN\n");

  for (i=0;i<REG_LEN;i++)
    fprintf(FReg,"%2.2X : %4.4X;\n",i,RegInit[i]);
  fprintf(FReg,"\nEND;\n");


}


int Bound(int Val,int Min,int Max)
{
  if (Val>Max) printf("<Warning> Constant out of Range\n");
  return (Val>=Min && Val<=Max);
}

/*=============================================
Code generation
=============================================*/
int Field[NumFields];
int LastStage=0;
int AddressCounter=0;
long Word1;
long Output[512];

/*****************************************************************
WriteWord

Writes the next instruction word to the output stream
*****************************************************************/
int WriteWord(long Word)
{
  printf("<<%2.2X : %4.41X>>;\n\n",AddressCounter,Word);
  Output[AddressCounter++]=Word;

  return 0;
}


/*****************************************************************
ShipInstruction

Converts the field information into instructions and outputs them
*****************************************************************/
int ShipInstruction(void)
{
  long Word1=0;
  long Word2=0;

  Word1+=(Bound(Field[IF_Addr],0,31))?Field[IF_Addr]:0;
  Word1<<=1;
  Word1+=(Bound(Field[IF_R4],0,1))?Field[IF_R4]:1;
  Word1<<=3;
  Word1+=(Bound(Field[IF_R3],0,7))?Field[IF_R3]:R2_LD;
  Word1<<=4;
  Word1+=(Bound(Field[IF_R2],0,15))?Field[IF_R2]:R3_LD;
  Word1<<=2;
  Word1+=(Bound(Field[IF_R1],0,3))?Field[IF_R1]:EXT_RD;

  if (Field[IF_BrCond]<0 && Field[IF_Command]<0)
  {           //one word only
    WriteWord(Word1);
  }
  else
  {           //two words
    WriteWord(Word1|0x8000);
    Word2=0;
    Word2+=(Bound(Field[IF_BrAddr],0,511))?Field[IF_BrAddr]:0;
    Word2<<=3;
    Word2+=(Bound(Field[IF_BrCond],0,7))?Field[IF_BrCond]:0;
    Word2<<=4;
    Word2+=(Bound(Field[IF_Command],0,15))?Field[IF_Command]:0;
    WriteWord(Word2);
  }

  //reset fields
  LastStage=0;
  for (int i=0;i<NumFields;i++) Field[i]=-1;

  return 0;
}

/*****************************************************************
SetField

Sets the value of one of the instruction fields.  Stage indicates the
instruction stage.  If this is less than the previous field add, a new
instruction is started and the previous one is output.  This is so that
instructions are added in the correct order.

Num is the field number and val is the new value.
If there is a field conflict (eg 2 instructions using memory at once),
the command is split, outputing two commands.
*****************************************************************/
int SetField(int Stage,int Num,int Val)
{
  //if an earlier stage in the sequence, write out the last instr
  if (Stage<LastStage) ShipInstruction();
  LastStage=Stage;

  if (Num>=0)
  {
    if (Field[Num]>=0) //check if already initilised
                       //if so, extend
    {
      //send off the R1 and address operations in the first instruction
      //and copy existing branching, R2 and R3 stuff into next instruction
      //keep R2 by shuffling back into R3

      int ExR2,ExR3,ExBrA,ExBrC;
      ExR2 =Field[IF_R2 ]; Field[IF_R2    ]=R3_LD;
      ExR3 =Field[IF_R3 ]; Field[IF_R3    ]=R2_LD;
      ExBrA=Field[IF_BrAddr]; Field[IF_BrAddr]=-1;
      ExBrC=Field[IF_BrCond]; Field[IF_BrCond]=-1;
      ShipInstruction();          //send out the first instruction

      Field[IF_R2    ]=ExR2;
      Field[IF_R3    ]=ExR3;
      Field[IF_BrAddr]=ExBrA;
      Field[IF_BrCond]=ExBrC;
    }
    Field[Num]=Val;
  }
  return 0;
}


/*****************************************************************
GetMemAdder

Returns the current address count - valid only immediately after a
SetField(ST_Label,-1,0); or a SetField(ST_Branch,IF_BrAddr)
*****************************************************************/
int GetMemAddr(void) {return AddressCounter;}


/*****************************************************************
EndStage

Called to indicate that no more field entries of this stage are acceptable
in this command
*****************************************************************/
void EndStage(int Stage) {LastStage=Stage+1;}


/*****************************************************************
InitialiseOutput

Prepares the fields and the output file
*****************************************************************/
void InitialiseOutput(void)
{
  int i;

  //initialise fields to empty
  for (i=0;i<NumFields;i++) Field[i]=-1;
  AddressCounter=0;
  LastStage=0;


  fprintf(FOut,"WIDTH=16;\nDEPTH=512;\nADDRESS_RADIX=HEX;\
\nDATA_RADIX=HEX;\n");
  fprintf(FOut,"\nCONTENT BEGIN\n");
}


/*****************************************************************
CloseOutput

Ships out the last instruction and closes the MIF file
*****************************************************************/
void CloseOutput(void)
{
  int i;

  SetField(0,-1,0);    //stage zero command to force any outpt

  for (i=0;i<AddressCounter;i++)
    fprintf(FOut,"%2.2X : %4.41X;\n",i,Output[i]);
  fprintf(FOut,"\nEND;\n");
}


/*****************************************************************
ChangeMemRef

Changes the memory reference at location InstrLoc to point to BrLoc
```

```
*************************************************************/
void ChangeMemRef(int InstrLoc,int BrLoc)
{
  Output[InstrLoc+1]&=0x7F;
  Output[InstrLoc+1]|=BrLoc<<7;
}


struct LabelInfo
{
  char Text[64];
  int Location;
  LabelInfo*Next;
};

LabelInfo*LabRefs=0;

/*************************************************************
ResolveLabel

Scans the list of labels Refs and if any match the Text, the pointer in the
instruction is changed to Loc.

Returns the new list of labels, with the resolved labels deleted
*************************************************************/
LabelInfo*ResolveLabel(LabelInfo*Refs,char*Text,int Loc)
{
  LabelInfo*RetPtr;

  if (!Refs) return Refs;        //if no list, return nothing
  Refs->Next=ResolveLabel(Refs->Next,Text,Loc);
                                 //reslove rest of list
  if (!strcmp(Text,Refs->Text))
  {
    printf("Resolved <%s> referred at [%i] is at [%i]\n",
           Text,Refs->Location,Loc);
    ChangeMemRef(Refs->Location,Loc);
    RetPtr=Refs->Next;
    delete Refs;
    return RetPtr;
  }
  return Refs;
}

/*************************************************************
PrintUnresolved

Prints the list of unresolved labels
*************************************************************/
void PrintUnresolved(LabelInfo*Refs)
{
  if (!Refs) return;    //if no list, return nothing

  printf("Unknown <%s> referred at [%i]\n",
         Refs->Text,Refs->Location);
  PrintUnresolved(Refs->Next);
}


/*************************************************************
UnknownLabel

Adds a label to the unresolved list.
*************************************************************/
void UnknownLabel(char*Text,int Location)
{
  LabelInfo*Next;

  Next=LabRefs;
  LabRefs=new LabelInfo;
  if (!LabRefs)
  {
    printf("ERROR: Cannot allocate memory");
    exit(1);
  }
  LabRefs->Next=Next;
  LabRefs->Location=Location;
  strcpy(LabRefs->Text,Text);

}


/*===============================================
NODE CLASS

Forms a base class for a linked list
===============================================*/

class Node
{
  Node*NextN;

  public:
    Node(Node*N): NextN(N) { }

    Node*Next(void) {return NextN;};
    void Next(Node*N) {NextN=N;};
};

/*===============================================
SYMBOL CLASS
===============================================*/

class Symbol:public Node
{
  int SType;
  int SValue;
  char SText[32];

  public:

  Symbol(int Type,int Value,char*Text,Symbol*Next); //:Node(Next);
  Symbol(Symbol*S,Symbol*Next);
  int Type(void) {return SType;}
  int Value(void) {return SValue;}
  char*Text(void) {return SText;}

  Symbol*Next(void) {return (Symbol*)(Node::Next());}
  void Next(Symbol*S) {Node::Next(S);};

  virtual int Parse(Symbol*Expr);


};

/*-----------------------------------------------
Node Functions
-----------------------------------------------*/

/*************************************************************
DeleteNode

Deletes up to Num nodes off the head of a linked list. Returns the pointer
to the rest of the list
*************************************************************/
Node*DeleteNode(Node*N,int Num)
{
  Node*Nt;

  if (!N || !Num) return 0;              //stop if at end
  Nt=DeleteNode(N->Next(),Num-1);        //delete rest
  delete N;                              //delete this node

  return Nt;                             //return pointer to rest
}

/*-----------------------------------------------
Symbol Functions
-----------------------------------------------*/

/*************************************************************
Constructor
*************************************************************/
Symbol::Symbol(int Type,int Value,char*Text,Symbol*Next):Node(Next)
{
  SType=Type;
  SValue=Value;

  if (Text)                      //if string text was passed
  {
    strncpy(SText,Text,31);      //copy the string text
    SText[31]=0;
  }
  else
    SText[0]=0;                  //otherwise have empty string
}

/*************************************************************
copy constructor

copies existing symbol, and then links to the specified next node
*************************************************************/
Symbol::Symbol(Symbol*S,Symbol*NextS):Node(NextS)
{
  *this=*S;
  Next(NextS);
}

/*************************************************************
Parse

Tries to match the passed symbol list to this type of symbol.  The passed
expression comes from the command line. This function is overridden in
```

```
classes that represent more complex grammar elements.

Returns 0 if the expression did not match this expression type.
*****************************************************************/
int Symbol::Parse(Symbol*Expr)
{
  int Match;

  if (!Expr) return 0;                    //no match if no expression
  if (Type()!=Expr->Type()) return 0; //no match to this symbol
  if (!Next())
    Match=1;                              //last in chain, so there is match
  else
    Match=Next()->Parse(Expr->Next()); //otherwise check rest of chain

  if (Match)                              //if it matched, copy the details
  {                                       //into this symbol data
    SValue=Expr->Value();
    strcpy(SText,Expr->Text());
  }
  return Match;
}


/*****************************************************************
Print Expression

Provides an ascii represenation of the parsed expression.  Used in error
reporting
*****************************************************************/
void PrintExpression(Symbol*S)
{
  if (S)
  {
    printf("[%c,%i,%s]",S->Type(),S->Value(),S->Text());
    PrintExpression(S->Next());
  }
  else printf("\n");
}


/*****************************************************************
Symbol defines

Following are the integer typenames for the allowed vocabulary of symbols
S_unknown are typically labels.
*****************************************************************/

#define IDoffset 'a'
#define S_unknown 'A'

#define S_memloc IDoffset+0
#define S_extloc  IDoffset+1
#define S_memdecl IDoffset+2
#define S_extdecl IDoffset+3//declare both external memory and ext commands
#define S_if      IDoffset+4
#define S_brd     IDoffset+5
#define S_R2      IDoffset+6
#define S_R2H     IDoffset+7
#define S_R2L     IDoffset+8
#define S_eof     IDoffset+9
#define S_one     IDoffset+10
#define S_sixteen IDoffset+11
#define S_zero    IDoffset+12
#define S_out     IDoffset+13
#define S_labdecl IDoffset+14
#define S_label   IDoffset+15
#define S_two     IDoffset+16
#define S_R3      IDoffset+17
#define S_abs     IDoffset+18
//#define Tcommode 5  //change to command mode


//To simplify the grammar, numeric operands, such as the 2 and 16 used
//for shift operations, are spelt out.      This also makes it plain that
//these numbers are special and that other integers cannot be used.

//Here the symbol structure is used to store the list of all possible
//symbols, and their associated text.    This is not the conventional
//use of the symbol structure.

Symbol*Table= new Symbol(S_memdecl,0,"int",
              new Symbol(S_extdecl,0,"constant",
              new Symbol(S_if,0,"if",
              new Symbol(S_brd,0,"goto",
              new Symbol(S_R2,0,"a",
              new Symbol(S_R2H,0,"ah",
              new Symbol(S_R2L,0,"al",
              new Symbol(S_eof,0,"END",
              new Symbol(S_one,0,"one",
              new Symbol(S_sixteen,0,"sixteen",
              new Symbol(S_zero,0,"ZERO",
              new Symbol(S_out,0,"out",
              new Symbol(S_labdecl,0,"LABEL",
              new Symbol(S_two,0,"two",
              new Symbol(S_R3,0,"R3",
              new Symbol(S_abs,0,"abs",
              0 ))))))))))))))))));

int Nmemloc=0;        //number of memory locations defined

/*****************************************************************
AddTableEntry

Adds an entry to the dynamic symbol table
*****************************************************************/
void AddTableEntry(int ID,int Value,char*Label)
{
  Table=new Symbol(ID,Value,Label,Table);
}
void AddTableEntry(Symbol*S)
{
  Table=new Symbol(S->Type(),S->Value(),S->Text(),Table);
}


/*****************************************************************
AddMemoryLocation

Memory locations are added to the symbol table with the location value
as a parameter
*****************************************************************/
int AddMemoryLocation(char*Label)
{
  AddTableEntry(S_memloc,Nmemloc,Label);
  return Nmemloc++;          // global counter
}


/*****************************************************************
FindTableEntry

Find the symbol in the table with the text of Label
*****************************************************************/
//recursive helper function to span the table
Symbol*FindEntry(char*Label,Symbol*List)
{
  if (!List) return 0;
  if (!strcmp(Label,List->Text())) return List;
  return FindEntry(Label,(Symbol*)List->Next());
}

Symbol*FindTableEntry(char*Label)
{
  return FindEntry(Label,Table);
}


int NextChar=32;              // remembers fetched char
int LineNo=1;                 // global line number counter

/*****************************************************************
GetNextChar

Returns the next pre-processed character
comments are removed
Line numbers are counted
*****************************************************************/
int GetNextChar(void)
{
  int ThisChar;

  ThisChar=NextChar;
  NextChar=fgetc(FIn);

  //Filter out line comments
  if ((ThisChar=='/' && NextChar=='/') ||
      (ThisChar=='-' && NextChar=='-'))
  {
    while (NextChar!=10 && NextChar!=EOF) NextChar=fgetc(FIn);
    ThisChar=' ';
  }

  if (ThisChar==10) LineNo++;

  return ThisChar;
}


/*****************************************************************
GetLineNo

returns the line number in the current main file
*****************************************************************/
int GetLineNo(void)
{
  return LineNo;
```

```
}


/*============================================
R1Expr Class

============================================*/
class R1Expr:public Symbol
{
  int Stage;
  Symbol*LastExpr;
  public:

  R1Expr(int EStage,Symbol*Next):Stage(EStage), Symbol(0,0,0,Next) { }
  virtual int Parse(Symbol*Expr);
  void ParseLastAgain(void);
};


/*============================================
R4Expr Class

============================================*/
class R4Expr:public Symbol
{
  public:

  R4Expr(Symbol*Next):Symbol(0,0,0,Next) { }
  virtual int Parse(Symbol*Expr);
};


/*************************************************************
R1Expr::Parse

Parse an R1 Expression
*************************************************************/
int R1Expr::Parse(Symbol*Expr)
{
  Symbol*TestExpr;

  LastExpr=Expr;       //kludge to allow multiply to request the r1 status
                       //to be written again

  //-------------
  //Check for memory read
  //-------------

  //the test expression is linked into the Next() node, because *this is
  //likely to be a test expression itself, with a greater structure imposed
  //by the subsequent nodes.
  TestExpr= new Symbol(S_memloc,0,0, Next() );
  if (TestExpr->Parse(Expr))
  {
    SetField(Stage,IF_R1,MEM_RD);
    SetField(Stage,IF_Addr,TestExpr->Value());
    printf("Read <Mem>%s\n",TestExpr->Text() );

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);


  //-------------
  //Check for External read
  //-------------

  TestExpr= new Symbol(S_extloc,0,0, Next() );
  if (TestExpr->Parse(Expr))
  {
    SetField(Stage,IF_R1,EXT_RD);
    SetField(Stage,IF_Addr,TestExpr->Value());
    printf("Read Ext <%s>\n",TestExpr->Text() );

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);

  //-------------
  //Check for R2 read
  //-------------

  TestExpr= new R4Expr( Next() );
  if (TestExpr->Parse(Expr))
  {
    SetField(Stage,IF_R1,R2_RD);
    printf("Read R4");

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);
  return 0;
}

/*************************************************************
R1Expr::ParseLastAgain

kludge to allow multiply to request the r1 status to be written again
Re-scans the R1 expression from last time
*************************************************************/
void R1Expr::ParseLastAgain(void)
{
  Symbol*Nxt=Next();
  Next(0);             //making this the end of the list means that it
                       //will successfully parse only the R1 expression
                       //this relys on Symbol declaring a success if
                       //it is the last element in the expresssion list
  Parse(LastExpr);

  Next(Nxt);
}


/*============================================
R3Expr Class

============================================*/
class R3Expr:public Symbol
{
  public:

  R3Expr(Symbol*Next):Symbol(0,0,0,Next) { }
  virtual int Parse(Symbol*Expr);
};


/*************************************************************
R3Expr::Parse

Parse an R3 Expression
*************************************************************/
int R3Expr::Parse(Symbol*Expr)
{
  Symbol*TestExpr;


  //-------------
  //Check for R3=R2
  //-------------

  TestExpr= new Symbol(S_R2,0,0, Next() );
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R3,R2_LD);
    printf("R3=R2\n");

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);


  //-------------
  //Check for R3=R2*two
  //-------------

  TestExpr= new Symbol(S_R2,0,0,
              new Symbol('*',0,0,
              new Symbol(S_two,0,0,
            Next() )));
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R3,R2_SHL0);
    printf("R3=R2<<1\n",TestExpr->Text() );

    DeleteNode(TestExpr,3);
    return 1;
  }
  DeleteNode(TestExpr,3);


  //-------------
  //Check for R3=R2/two
  //-------------

  TestExpr= new Symbol(S_R2,0,0,
              new Symbol('/',0,0,
              new Symbol(S_two,0,0,
            Next() )));
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R3,R2_SHR);
    printf("R3=R2>>1\n",TestExpr->Text() );
```

```
    DeleteNode(TestExpr,3);
    return 1;
  }
  DeleteNode(TestExpr,3);


  //————————————
  //Check for R3=R2>>16
  //————————————

  TestExpr= new Symbol(S_R2,0,0,
                new Symbol('>',0,0,
                new Symbol('>',0,0,
                new Symbol(S_sixteen,0,0,
              Next() ))));
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R3,R2_SHW);
    printf("R3=R2>>16\n",TestExpr->Text() );

    DeleteNode(TestExpr,4);
    return 1;
  }
  DeleteNode(TestExpr,4);


  //————————————
  //Check for R3="R3"
  //————————————
  TestExpr= new Symbol(S_R3,0,0,
              Next() );
  if (TestExpr->Parse(Expr))
  {
    printf("Reading R3 directly\n");

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);

  return 0;
}


/*============================================
R2Expr Class

============================================*/
class R2Expr:public Symbol
{
  public:

  R2Expr(Symbol*Next):Symbol(0,0,0,Next) { }
  virtual int Parse(Symbol*Expr);
};


/*****************************************************************
R2Expr::Parse

Parse an R2 Expression
*****************************************************************/
int R2Expr::Parse(Symbol*Expr)
{
  Symbol*TestExpr;
  int Match;
  int i;

  Symbol*S1,*S2;

  //————————————
  //Check for R3
  //————————————
  TestExpr= new R3Expr(Next());
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R2,R3_LD);
    printf("R2=R3\n");

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);

  //————————————
  //Check for R1
  //————————————
  TestExpr= new R1Expr(ST_R2Expr,Next() );

  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R2,R1_LD);
    printf("R2=R1\n");
    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);


  //————————————
  //Check for R3+R1
  //————————————
  TestExpr= new R3Expr(
              new Symbol('+',0,0,
              new R1Expr(ST_R2Expr,
              Next() )));

  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R2,R1R3_ADD);
    printf("R2=R3+R1\n");

    DeleteNode(TestExpr,3);
    return 1;
  }
  DeleteNode(TestExpr,3);


  //————————————
  //Check for R3-R1
  //————————————
  TestExpr= new R3Expr(
              new Symbol('-',0,0,
              new R1Expr(ST_R2Expr,
              Next() )));

  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R2,R1R3_SUB);
    printf("R2=R3-R1\n");

    DeleteNode(TestExpr,3);
    return 1;
  }
  DeleteNode(TestExpr,3);

  //————————————
  //Check for -R1
  //————————————
  TestExpr= new Symbol('-',0,0,
              new R1Expr(ST_R2Expr,
              Next() ));

  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R2,R1_LDN);
    printf("R2=-R1\n");

    DeleteNode(TestExpr,2);
    return 1;
  }
  DeleteNode(TestExpr,2);

  //————————————
  //Check for 0
  //————————————
  TestExpr= new Symbol(S_zero,0,0, Next() );

  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R2,ZERO_LD);
    printf("R2=0\n");

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);


  //————————————————————
  //Check for Multiply
  //————————————————————
  R1Expr*R1E;
  TestExpr= new R3Expr(
              new Symbol('*',0,0,
          R1E=new R1Expr(ST_R2Expr,
              new Symbol('*',0,0,
              new Symbol(S_two,0,0,
              Next() )))));

  if (TestExpr->Parse(Expr))
  {

    //1st instr
    SetField(ST_R2Expr,IF_Command,COM_preload29);
//    SetField(ST_R2Expr,IF_R3,R2_SHW);
    SetField(ST_R2Expr,IF_R2,R1R3_MUL1);
```

```
//     EndStage(ST_R2Expr);
    SetField(ST_Label,-1,0); //force new instruction
    i=GetMemAddr();

    //2nd instr
    SetField(ST_Branch,IF_BrCond,BR_countnz);
    SetField(ST_Branch,IF_BrAddr,i);
//     TestExpr->Parse(Expr);   //set R1expr again
    R1E->ParseLastAgain();
    SetField(ST_R2Expr,IF_R3,R2_SHR);
    SetField(ST_R2Expr,IF_R2,R1R3_MUL1);
    printf("Multiply\n");

    DeleteNode(TestExpr,5);
    return 1;
  }
  DeleteNode(TestExpr,5);


  //----------------------
  //Check for Divide
  //----------------------
  TestExpr= new R3Expr(
              new Symbol('/',0,0,
          R1E=new R1Expr(ST_R2Expr,
              new Symbol('/',0,0,
              new Symbol(S_two,0,0,
              Next() )));

  if (TestExpr->Parse(Expr))
  {

    //1st instr
    SetField(ST_R2Expr,IF_Command,COM_preload59);
    SetField(ST_R2Expr,IF_R2,R1R3_DIV1);

    //2nd instr
    SetField(ST_Label,-1,0); //force new instruction
    i=GetMemAddr();
    SetField(ST_R2Expr,IF_R3,R2_LD);
    R1E->ParseLastAgain();
    SetField(ST_R2Expr,IF_R2,R1R3_DIV3);

    //3rd instr
    SetField(ST_Label,-1,0); //force new instruction
    SetField(ST_Branch,IF_BrCond,BR_countnz);
    SetField(ST_Branch,IF_BrAddr,i);
    SetField(ST_R2Expr,IF_R3,R2_DIV);
    R1E->ParseLastAgain();
    SetField(ST_R2Expr,IF_R2,R1R3_DIV1);

    //4th instr - same as 2nd
    SetField(ST_Label,-1,0); //force new instruction
    SetField(ST_R2Expr,IF_R3,R2_LD);
    R1E->ParseLastAgain();
    SetField(ST_R2Expr,IF_R2,R1R3_DIV3);

    //5th instr
    SetField(ST_Label,-1,0); //force new instruction
    SetField(ST_R2Expr,IF_R3,R2_DIV);
    //allow R2 instructions to occur now the result may be accessed in
    //this instruction by the R3 expression "R3"
    printf("Divide\n");

    DeleteNode(TestExpr,5);
    return 1;
  }
  DeleteNode(TestExpr,5);


  //----------------
  //Check for abs R1
  //----------------
  TestExpr= new Symbol(S_abs,0,0,
              new Symbol('(',0,0,
              new R1Expr(ST_R2Expr,
              new Symbol(')',0,0,
              Next() ));

  if (TestExpr->Parse(Expr))
  {
    SetField(ST_R2Expr,IF_R2,R1_ABS);
    printf("R2=abs(R1)\n");

    DeleteNode(TestExpr,4);
    return 1;
  }
  DeleteNode(TestExpr,4);


  return 0;
}

/*************************************************************
R4Expr::Parse

Parse an R4 Expression
*************************************************************/
int R4Expr::Parse(Symbol*Expr)
{
  Symbol*TestExpr;


  //----------------
  //Check for out low
  //----------------

  TestExpr= new Symbol(S_R2L,0,0, Next() );
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_Write,IF_R4,OUTL);
    printf("Low Output\n");

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);

  //----------------
  //Check for out high
  //----------------

  TestExpr= new Symbol(S_R2H,0,0, Next() );
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_Write,IF_R4,OUTH);
    printf("High Output\n");

    DeleteNode(TestExpr,1);
    return 1;
  }
  DeleteNode(TestExpr,1);

  return 0;
}


/*=============================================
Statement Class

=============================================*/
class Statement:public Symbol
{
  public:

  Statement(Symbol*Next):Symbol(0,0,0,Next) { }
  virtual int Parse(Symbol*Expr);
};

/*Statement::Statement(Symbol*Next):Symbol(0,0,0,Next)
{
} */

int Statement::Parse(Symbol*Expr)
{
  Symbol*TestExpr;
  int Match;
  int i,j;

  Symbol*S1,*S2;

  //----------------
  //Check for memory declare
  //----------------

  TestExpr= new Symbol(S_memdecl,0,0,
            S1=new Symbol(S_unknown,0,0,
              Next() ));

  if (TestExpr->Parse(Expr))
  {
    printf("Memory Allocated to %s\n",TestExpr->Next()->Text() );
    AddMemoryLocation(TestExpr->Next()->Text());

    DeleteNode(TestExpr,2);
    return 1;
  }
  DeleteNode(TestExpr,2);


  //----------------
  //Check for init mem declare
  //----------------
  TestExpr= new Symbol(S_memdecl,0,0,
            S1=new Symbol(S_unknown,0,0,
              new Symbol('=',0,0,
            S2=new Symbol(S_unknown,0,0,
```

```
            Next() ))));

if (TestExpr->Parse(Expr))
{
  i=atoi(S2->Text());
  printf("Memory Allocated to %s, Init=%i\n",S1->Text(),i);
  j=AddMemoryLocation(TestExpr->Next()->Text());
  InitReg(j,i);

  DeleteNode(TestExpr,4);
  return 1;
}
DeleteNode(TestExpr,4);


//-----------------
//Check for negative init mem declare
//-----------------
TestExpr= new Symbol(S_memdecl,0,0,
          S1=new Symbol(S_unknown,0,0,
             new Symbol('=',0,0,
             new Symbol('-',0,0,
          S2=new Symbol(S_unknown,0,0,
             Next() )))));

if (TestExpr->Parse(Expr))
{
  i=-atoi(S2->Text());
  printf("Memory Allocated to %s, Init=%i\n",S1->Text(),i);
  j=AddMemoryLocation(TestExpr->Next()->Text());
  InitReg(j,i);

  DeleteNode(TestExpr,5);
  return 1;
}
DeleteNode(TestExpr,5);


//-----------------
//Check for External declare
//-----------------
TestExpr= new Symbol(S_extdecl,0,0,
          S1=new Symbol(S_unknown,0,0,
             new Symbol('=',0,0,
          S2=new Symbol(S_unknown,0,0,
             Next() ))));

if (TestExpr->Parse(Expr))
{
  i=atoi(S2->Text());
  printf("External <%i> Allocated to %s\n",i,S1->Text() );
  AddTableEntry(S_extloc,i,S1->Text());

  DeleteNode(TestExpr,4);
  return 1;
}
DeleteNode(TestExpr,4);


//-----------------
//Check for R2=R2Expr
//-----------
TestExpr= new Symbol(S_R2,0,0,
          S1=new Symbol('=',0,0,
          S2=new R2Expr(
             Next() )));

if (TestExpr->Parse(Expr))
{
  i=atoi(S2->Text());
  EndStage(ST_R2Expr);
  printf("R2=R2Expr\n");
  AddTableEntry(S_extloc,i,S1->Text());

  DeleteNode(TestExpr,3);
  return 1;
}
DeleteNode(TestExpr,3);

//-----------------
//Check for output value
//-----------------
TestExpr= new Symbol(S_out,0,0,
          S1=new Symbol('(',0,0,
          S2=new R4Expr(
             new Symbol(')',0,0,
             Next() ))));

if (TestExpr->Parse(Expr))
{
  //dont end stage EndStage(ST_Write);
  printf("Output\n");

  DeleteNode(TestExpr,4);
  return 1;
}
DeleteNode(TestExpr,4);

//-----------------
//Check for memory write
//-----------------
TestExpr= new Symbol(S_memloc,0,0,
          new Symbol('=',0,0,
          new R4Expr(
          Next() )));

if (TestExpr->Parse(Expr))
{
  //dont end stage EndStage(ST_Write);
  SetField(ST_Write,IF_R1,MEM_WR);
  SetField(ST_Write,IF_Addr,TestExpr->Value());
  printf("Mem write\n");

  DeleteNode(TestExpr,3);
  return 1;
}
DeleteNode(TestExpr,3);

//-----------------
//Check for command
//-----------------
TestExpr= new Symbol(S_extloc,0,0,
          Next() );

if (TestExpr->Parse(Expr))
{
  SetField(ST_Command,IF_Command,TestExpr->Value());
  EndStage(ST_Command);
  printf("Command <%s>\n",TestExpr->Text());

  DeleteNode(TestExpr,1);
  return 1;
}
DeleteNode(TestExpr,1);


//-----------------
//Check for UnDefined label
//-----------------
TestExpr= new Symbol(S_labdecl,0,0,
          S1=new Symbol(S_unknown,0,0,
             Next() ));

if (TestExpr->Parse(Expr))
{
  //dont end stage EndStage(ST_Label);
  SetField(ST_Label,-1,0); //force start of instruction
  AddTableEntry(S_label,GetMemAddr(),S1->Text());
  LabRefs=ResolveLabel(LabRefs,S1->Text(),GetMemAddr());
  printf("Address Defined <%s=%i>\n",S1->Text(),GetMemAddr());

  DeleteNode(TestExpr,2);
  return 1;
}
DeleteNode(TestExpr,2);


//-----------------
//Check for Branch to Defined label
//-----------------
TestExpr= new Symbol(S_if,0,0,
          S1=new Symbol(S_extloc,0,0,
             new Symbol(S_brd,0,0,
          S2=new Symbol(S_label,0,0,
             Next() ))));

if (TestExpr->Parse(Expr))
{
  SetField(ST_Branch,IF_BrCond,S1->Value());
  SetField(ST_Branch,IF_BrAddr,S2->Value());
  EndStage(ST_Branch);
  printf("If <%s> Branch to <%s=%i>\n",S1->Text(),S2->Text(),S2->Value());

  DeleteNode(TestExpr,4);
  return 1;
}
DeleteNode(TestExpr,4);


//-----------------
//Check for Branch to Undefined label
//-----------------
TestExpr= new Symbol(S_if,0,0,
          S1=new Symbol(S_extloc,0,0,
             new Symbol(S_brd,0,0,
          S2=new Symbol(S_unknown,0,0,
             Next() ))));
```

```
  if (TestExpr->Parse(Expr))
  {
    SetField(ST_Branch,IF_BrCond,S1->Value());
    UnknownLabel(S2->Text(),GetMemAddr());
    SetField(ST_Branch,IF_BrAddr,0);
    EndStage(ST_Branch);
    printf("If <%s> Branch to <%s=???>\n",S1->Text(),S2->Text());

    DeleteNode(TestExpr,4);
    return 1;
  }
  DeleteNode(TestExpr,4);
  printf("Error in Line %i:",GetLineNo());
  PrintExpression(Expr);
  return 0;
}


char SymBuff[80];
int Next=32;            //holds next char to be processed
                        //initialise with whitespace
/*****************************************************************
GetNextSymbol

Returns a pointer to a string containg the next symbol in the command line.
For alphanumeric strings, the first character in the string is A, followed
by the string.   For other symbols, the first character is the symbol.
*****************************************************************/
Symbol*GetNextSymbol(void)
{
  Symbol*NewSymbol;
  Symbol*TableSymbol;
  int SymLen=0;


  while (Next>=0 && Next<=' ') Next=GetNextChar(); //skip whitespace
  if (Next==EOF || Next<0) return 0; //check for end of file

  //check for alphanumeric string
  if ((Next>='a' && Next<='z') || Next=='_' ||
      (Next>='A' && Next<='Z') ||
      (Next>='0' && Next<='9'))
  {
    SymLen=0;
    do
    {
      SymBuff[SymLen++]=Next;
      Next=GetNextChar();
    } while ((Next>='a' && Next<='z') || Next=='_' ||
             (Next>='A' && Next<='z') ||
             (Next>='0' && Next<='9'));
    SymBuff[SymLen++]=0;

    //check if it is a symbol in the table
    //if it is, replace the info with the symbol info
    if ( (TableSymbol=FindTableEntry(SymBuff))!=0 )
      NewSymbol=new Symbol(TableSymbol,0);
    else
      NewSymbol=new Symbol(S_unknown,0,SymBuff,0);
  }
  else
  {
    NewSymbol= new Symbol(Next,0,0,0);
    Next=GetNextChar();
  }


  return NewSymbol;
}


Symbol*GetNextExpression(void)
{
  Symbol*S;

  S=GetNextSymbol();
  //keep collecting symbols into the list until end of line or file
  if (S && S->Type()!=';' && S->Type()!=S_eof)
    S->Next(GetNextExpression());
  return S;
}


/*****************************************************************
Statement loop

*****************************************************************/
int ReadStatements(void)
{
  int Res;
  Symbol*Expr=new Statement(new Symbol(';',0,0,0));
  Symbol*Line;

  Line=GetNextExpression();
  while (Line->Type()!=S_eof)
  {
    if (!Line)
    {
      printf("Unexpected end of file\n");
      return 1;
    }

    Res=Expr->Parse(Line); //parse the line
    DeleteNode(Line,-1);   //then delete it
    if (!Res) return 1;    //if error, then exit

    Line=GetNextExpression(); //get the next line
  }

  return 0;
}


int main(int argc,char*argv[])
{
  char FName[64];

  printf("-------------------------------------------------------\n");
  printf("VSD-1 Assembler\n");
  printf("-------------------------------------------------------\n");

  //If no filename, print out usage prompt
  if (argc<2)
  {
    printf("Format:\n%s Input_File [Register_Init_File]\n",argv[0]);
    return 1;
  }

  strcpy(FName,argv[1]);
  strcat(FName,".VMC");
  //attempt to open file
  if (!(FIn=fopen(FName,"rb")))
  {
    printf("Cannot Open %s For Input\n",FName);
    return 1;
  }
  NextChar=fgetc(FIn);


  strcpy(FName,argv[1]);
  strcat(FName,".MIF");
  //attempt to open file
  if (!(FOut=fopen(FName,"wt")))
  {
    printf("Cannot Open %s For Output\n",FName);
    return 1;
  }

  InitialiseOutput();
  ReadStatements();
  CloseOutput();

  printf("Unresolved References:\n");
  if (LabRefs)
    PrintUnresolved(LabRefs);
  else
    printf("None.\n");


  if (argc>2)
  {
    strcpy(FName,argv[2]);
    strcat(FName,".MIF");
    printf("Writing Register Initialisation to %s\n",FName);
    printf("%i of 32 Registers used",Nmemloc);
    //attempt to open file
    if (!(FReg=fopen(FName,"wt")))
    {
      printf("Cannot Open %s For Output\n",FName);
      return 1;
    }

    WriteRegs();
    fclose(FReg);
  }
  else
    printf("No register initialisation file specified.\n");


  fclose(FIn); fclose(FOut);
  //FOut=fopen("out.ser","wb");
  return 0;

}
```

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. W.H. Freeman and Company, New York, 1992.

[2] J. W. Backus. The fortran automatic coding system. In *Proceedings AFIPS Western Joint Compter Conference*, pages pp 188–198, Baltimore, 1957. Spartan Books.

[3] R.E. Betz and B.J. Cook. A digital current controller for three phase voltage source inverters. Technical Report EE9702, Department of Electrical and Computer Engineering, The University of Newcastle, Australia, January 1997. Available at http://www.ee.newcastle.edu.au/users/staff/reb/Betz.html.

[4] R.E. Betz, B.J. Cook, and S.J. Henriksen. Digital current controller for three phase voltage source inverters. In *Proceedings of the IEEE IAS Annual Meeting*, New Orleans, October 1997.

[5] F. Blaabjerg, P. C. Kjaer, L. Christensen P. O. Rasmussen, S. Hansen, and J.R. Kristoffersen. Fast digital current control in switched reluctance motor drive without current feedback filters. In *European Conference on Power Electronics and Applications*, pages pp. 3625–3630, Trondheim, 1997.

[6] B. K. Bose. An adaptive hysteresis-band current control technique of a voltage-fed PWM inverter for machine drive system. *IEEE Transactions on Industrial Electronics*, vol. 37(no. 5):pp. 402–408, Oct 1990.

[7] Bimal K. Bose. *Power Electronics and AC Drives*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[8] Bimal K. Bose. Power electronics and motion control - technology status and recent trends. *IEEE Transactions on Industry Applications*, Vol. 29(No. 5):pp. 902–909, Sep 1993.

[9] Bimal K. Bose, editor. *Power Electronics and Variable Frequency Drives*. IEEE Press, Piscataway, NJ, 1997.

[10] Bimal K. Bose. Fuzzy logic and neural networks in power electronics and drives. *IEEE Industry Applications Magazine*, Vol. 6(No. 3):pp. 57–63, May 2000.

[11] F. Briz, M. W. Degner, and R. D. Lorenz. Analysis and design of current regulators using complex vectors. In *Proceedings of the IEEE IAS-97 Annual Meeting*, pages pp. 1504–1511, New Orleans, Oct 1997.

[12] David M. Brod and Donald W. Novotny. Current control of vsi-pwm inverters. *IEEE Transactions on Industry Applications*, Vol. 21(4):562–570, May 1985.

[13] Jong-Woo Choi and Seung-Ki Sul. Inverter output voltage synthesis using novel dead time compensation. *IEEE Transactions on Power Electronics*, Vol. 11(No. 2):pp. 221–227, Mar 1996.

[14] M. Depenbrock. Direct self-control (dsc) of inverter-fed induction machine. *IEEE Transactions on Power Electronics*, Vol. 3(No. 4):pp. 420–429, Oct 1988.

[15] W. Farrer and J.D. Miskin. Quasi-sine-wave fully regenerative invertor. *Proceedings of the IEE*, Vol. 120(No. 9):pp. 969–976, September 1973.

[16] Gene F. Franklin, J. David Powell, and Abbas Emmami-Naeini. *Feedback Control of Dynamic Systems*. Addison-Wesley Publishing Company, third edition, 1994.

[17] Thomas G. Habetler, Francesco Profumo, Michele Pastorelli, and Leon M. Tolbert. Direct torque control of induction machines using space vector modulation. *IEEE Transactions on Industry Applications*, Vol. 28(No. 5):pp. 1045–1053, Sep 1992.

[18] John L. Hennessy and David A. Patterson. *Computer Organization and Design*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1994.

[19] Soren J. Henriksen, Robert E. Betz, and Brian J. Cook. Digital hardware implementation of a current controller for im variable-speed drives. In *Proceedings of the IEEE IAS Annual Meeting*, St. Louis, October 1998.

[20] Soren J. Henriksen, Robert E. Betz, and Brian J. Cook. Digital hardware implementation of a current controller for im variable-speed drives. *IEEE Transactions on Industry Applications*, Vol. 35(No. 5):pp. 1021–1029, Sep 1999.

[21] Soren J. Henriksen, Robert E. Betz, and Brian J. Cook. Induction machine current control in digital hardware. In *Proceedings of the Australasian Universities Power Engineering Conference*, pages 557–562, Darwin, Septemper 1999.

[22] D.G. Holmes and D.A. Martin. Implementation of a direct digital predictive current controller for single and three phase voltage source inverters. In *Proceedings of the IEEE IAS-96 Annual Meeting*, pages pp. 906–913, San Diego, Oct 1996.

[23] Joachim Holtz. Pulsewidth modulation for electronic power conversion. *Proceedings of the IEEE*, Vol. 82(No. 8):pp. 1194–1214, Aug 1994.

[24] Jun-Koo Kang and Seung-Ki Sul. New direct torque control of induction motor for minimum torque ripple and constant switching frequency. *IEEE Transactions on Industry Applications*, Vol. 35(No. 5):pp. 1076–1082, Sep 1999.

[25] Marian P. Kazmierkowski and Luigi Malesani. Current control techniques for three-phase voltage-source pwm converters: A survey. *TIE*, Vol. 45(No. 5):pp. 691–703, Oct 1998.

[26] P. C. Kjaer, C. Cossar, and T. J. E. Miller. Very high bandwidth digital current controller for high-performance motor drives. In *6th International Conference of Power Electronics and Variable Speed Drives*, pages pp. 185–190, Nottingham, 1996.

[27] O. Kukrer. Discrete-time current control of voltage-fed three-phase pwm inveters. In *IEEE Transactions of Power Electronics*, volume Vol. 11, pages pp. 460–469, Mar 1996.

[28] Cristian Lascu, Ion Boldea, and Frede Blaabjerg. A modified direct torque control fo induction motor sensorless drive. *IEEE Transactions on Industry Applications*, Vol. 36(No. 1):pp. 122–130, Jan 2000.

[29] Dong-Choon Lee, Seung-Ki Sul, and Min-Ho Park. High performance current regulator for a field-orientted controlled induction motor drive. *IEEE Transactions on Industry Applications*, Vol. 30(No. 5):1247–1257, Sep 1994.

[30] S. Lithgow. Notes on induction machine modelling. Technical report, Department of Electrical and Computer Engineering, The University of Newcastle, Australia, 2000.

[31] Robert D. Lorenz and Donald B. Lawson. Performance of feedforward current regulators for field-oriented induction machine controllers. *IEEE Transactions on Industry Applications*, Vol. 23(No. 4):pp. 597–602, Jul 1987.

[32] Robert D. Lorenz, Thomas A. Lipo, and Donald W. Novotny. Motion control with induction motors. *Proceedings of the IEEE*, Vol. 82(No. 8):pp. 1215–1240, Aug 1994.

[33] Luigi Malesani, Paolo Mattavelli, and Simone Buso. Robust dead-beat current control for pwm rectifiers and active filters. *IEEE Transactions on Industry Applications*, Vol. 35(No. 3):pp. 613–620, May 1999.

[34] Luigi Malesani, Paolo Tenti, Elana Gaio, and Roberto Piovan. Improved current control technique of vsi pwm inverters with constant modulation frequency and extended voltage range. *IEEE Transactions on Industry Applications*, Vol. 27(No. 2):pp. 365–369, Mar 1991.

[35] Akira Nabae, Satoshi Ogasawara, and Hirofumi Akagi. A novel control scheme for current-controlled pwm inverters. *IEEE Transactions on Industry Applications*, Vol. 22(No. 4):pp. 678–690, Jul 1986.

[36] P. Naur, editor. *Revised report on the algorithmic language Algol 60.* Comm. ACM 6:1, 1963. pp. 1-17.

[37] D. O'Kelly and S. Simmons. *Introduction to Generalized Electrical Machine Theory.* McGraw-Hill Publishing Company, London, 1968.

[38] Gerhard Pfaff, Alois Weschta, and Albert F. Wick. Design and experimental results of a brushless ac servo drive. *IEEE Transactions on Industry Applications*, Vol. 20(Num. 4):pp. 814–821, July 1984.

[39] A. B. Plunkett. A current-controlled pwm transister inverter drive. In *Conference Record of the 14th Annual Meeting, IEEE Industry Applications Society*, pages pp. 785–892, 1979.

[40] Timothy M. Rowan and Russell J. Kerkman. A new synchronous current regulator and an analysis of current-regulated pwm inverters. *IEEE Transactions on Industry Applications*, Vol. 22(No. 4):pp. 678–690, Jul 1986.

[41] C. Schauder. Adaptive speed identification for vector control of induction motors without rotational transducers. In *Proceedings of the IEEE IAS Annual Meeting*, pages pp. 493–499, Oct 1991.

[42] C. D. Schauder and R. Caddy. Current control of voltage-source inverters for fast four-quadrant drive performance. *IEEE Transactions on Industry Applications*, Vol. 18(No. 2):pp. 163–171, Mar 1982.

[43] Lothar Springob and Joachim Holtz. High-bandwidth current control for torque-ripple compensation in pm synchronous machines. *IEEE Transactions on Industrial Electronics*, Vol. 45(No. 5):pp. 713–721, Oct 1998.

[44] Isao Takahashi and Toshihiko Noguchi. A new quick-response and high-efficiency control strategy of an induction motor. *IEEE Transactions on Industry Applications*, Vol. 22(No. 5):pp. 820–827, Sep 1986.

[45] Ying-Yu Tzou and Hau-Jean Hsu. Fpga realization of space-vector pwm control ic for three-phase pwm inverters. *IEEE Transactions on Power Electronics*, Vol. 12(No. 6):pp. 953–963, Nov 1997.

[46] H. W. van der Broeck, H. Ch. Skudelny, and G. Stanke. Analysis and realization of a pulse width modulator based on voltage space vectors. *IEEE Transactions on Industry Applications*, Vol. 24(No. 4):pp. 124–150, Jan 1988.

[47] L. A. Zadeh. Fuzzy sets. *Informat. Contr.*, Vol 8:pp 338–353, 1965.