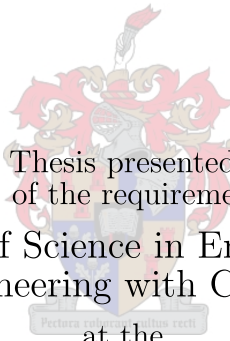
UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Integrated, FPGA Based NMR Teslameter and Power Supply for Accelerator Magnets

by

John-Philip Taylor



Thesis presented
in partial fulfilment of the requirements for the degree of

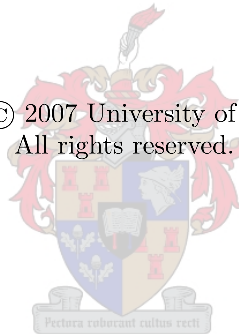
Master of Science in Engineering
(Electronic Engineering with Computer Science)
at the

University of Stellenbosch

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, 7602 Matieland, South Africa.

Supervisor: Prof H du T Mouton
Co-Supervisor: Prof T Jones

March 2007



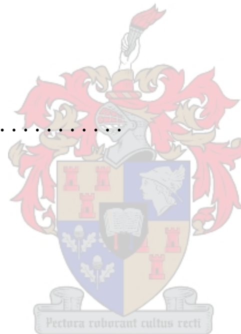
Copyright © 2007 University of Stellenbosch
All rights reserved.

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: ....
J Taylor

Date: ....



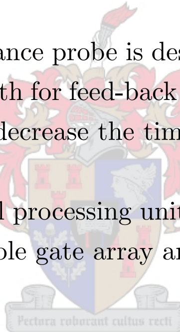
Abstract

Particle accelerators today have numerous magnets that require controlling. These include magnets for analysing, beam-path selection, focusing, etc. Also, design specifications are becoming tighter. A typical modern magnet power supply is expected to have a resolution of 16-bit and a stability of 10 ppm.

This thesis addresses two research areas. First, various aspects of high-performance accelerator magnet power supplies are investigated. An isolated dual-stage 3.5 kW converter is designed. The concept is verified through practical measurements. The control system and high-resolution pulse-width modulation are implemented within a field-programmable gate array.

Second, a nuclear-magnetic resonance probe is designed and simulated. It is intended to provide a measurement of field-strength for feed-back purposes. Some adjustments are made with existing technology in order to decrease the time between successive measurements to the order of 10 μs.

Also, the support systems (central processing unit, hardware drivers, etc.) are designed, implemented in the field-programmable gate array and tested successfully.



Opsomming

Hedendaagse partikel versnellers beskik oor talle magnete wat beheer verg. Hierdie sluit in magnete vir analise, straal-pad seleksie, fokus ens. Stelsel spesifikasies raak boonop toenemend knellend. 'n Tipiese moderne magneet kragbron is veronderstel om 16-bis resolusie en 10 ppm stabiliteit te waarborg.

Hierdie tesis adresseer twee navorsings areas: Eerstens word verskeie aspekte van hoë werkverrigting versnellersmagneet kragbronne ondersoek. 'n Geïsoleerde dubbel stadium 3.5 kW omsetter is ontwerp. Die beginsel is bevestig deur middel van praktiese metings. Die beheerstelsel en hoë-resolusie pulswydte modulator word geïmplementeer deur middel van 'n veld-programmeerbare hekskikking (FPGA).

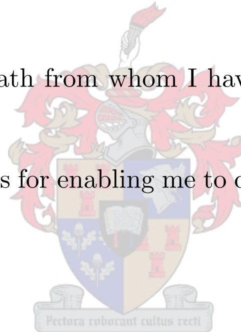
Tweedens, 'n kernmagnetiese resonansie probe is ontwerp en gesimuleer. Dit is bestem om 'n meting van veldsterkte te voorsien vir terugvoer doeleindes. Sommige verstellings is gemaak aan bestaande tegnologie ten einde die tydsverloop tussen opeenvolgende metings te verminder tot die orde van 10 μs.

Laastens, die ondersteuningstelsels (mikroverwerker, hardeware drywers ens.) is ontwerp, geïmplementeer in die FPGA en suksesvol getoets.

Acknowledgement

I would like to thank the following people:

- Prof H du T Mouton and Prof T Jones for their guidance as supervisors
- Dr C B Franklyn at NECSA for providing the idea, as well as funding, for this project
- Sandi Swanepoel, my fiancé, for proofreading and grammatical checks, as well as all the love and support in the world
- My fellow colleagues for being so supportive and available to exchange problems and ideas
- All the lecturers along my path from whom I have learnt the basics needed in order to complete this thesis
- Last but not least, my parents for enabling me to develop my full potential and pursue my dreams



Contents

Declaration ii

Abstract iii

Opsomming iv

Acknowledgement v

Contents vi

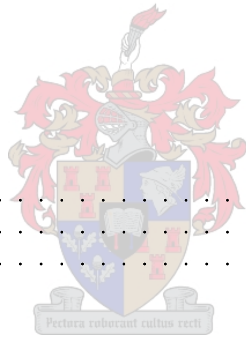
List of Figures xv

List of Tables xix

Nomenclature xxi

1 Introduction 1
1.1 Background . . . 1
1.2 Research Objectives . . . 2
1.3 Thesis Overview . . . 3

2 Existing Technology 4
2.1 Introduction . . . 4
2.2 Existing System . . . 4
2.2.1 Analysing Magnet . . . 5
2.2.2 Current Source . . . 7
2.2.3 Probe . . . 8
2.2.4 Controller . . . 9
2.3 Products on the Market . . . 10
2.3.1 Digital Teslameter . . . 10
2.3.2 Current Source . . . 10
2.4 High Performance Magnet Power Supplies . . . 10
2.4.1 Topologies . . . 11
2.4.2 Feed Forward Techniques . . . 12
2.4.3 Digital Control . . . 13
2.5 Field Measurement Techniques . . . 13
2.5.1 Calibrated Current Measurement . . . 13
2.5.2 Calibrated Hall Effect Probe . . . 14
2.5.3 Continuous Wave NMR . . . 14
2.5.4 Pulsed NMR . . . 14
2.6 Summary . . . 15



3 Design Overview **16**
3.1 Introduction 16
3.2 Block Diagrams 16
3.2.1 General Block Diagram 16
3.2.2 Mains and Power Supply 17
3.2.3 Power Converter 18
3.2.4 Probe 18
3.2.5 Controller 19
3.3 Overall Layout 20
3.3.1 Front Panel 20
3.3.2 Layout 20
3.4 Summary 21

4 Converter **23**
4.1 Introduction 23
4.2 Definitions 23
4.3 Specification 24
4.3.1 Source of Power 24
4.3.2 Power Rating and Load 24
4.3.3 Isolation 24
4.3.4 Range 24
4.3.5 Stability 25
4.3.6 Audible Noise 25
4.4 Topology 25
4.4.1 Overall 25
4.4.2 Rectifier 26
4.4.3 Stage 1 27
4.4.4 Stage 2 29
4.5 Rectifier Detail 30
4.5.1 Bus Capacitor 30
4.5.2 Waveforms 31
4.5.3 Line-Frequency Ripple 32
4.5.4 Ripple Current Through Bus Capacitor 33
4.5.5 Losses 33
4.5.6 Supply Current 33
4.6 Stage 1 Detail 34
4.6.1 Switches 34
4.6.2 Switching 34
4.6.3 Waveforms 35
4.6.4 Transformer Winding Ratio 37
4.6.5 High Frequency Rectifier 37
4.6.6 Switching Frequency 38
4.6.7 Filter 38
4.6.8 Losses 40
4.6.9 Dead Time 41
4.7 Transformer Detail 42
4.7.1 Configuration 42
4.7.2 Non-Ideal Properties 43
4.7.3 Leakage Inductance 44
4.7.4 Isolation 44
4.7.5 Thermal Analysis 46
4.8 Stage 2 Detail 47

4.8.1 Switches . . . 47
4.8.2 Switching . . . 47
4.8.3 Waveforms . . . 47
4.8.4 Switching Frequency . . . 49
4.8.5 Filter . . . 49
4.8.6 Losses . . . 50
4.8.7 Dead Time . . . 51
4.9 Heatsink . . . 51
4.9.1 Contribution of Stage 1 MOSFETS . . . 51
4.9.2 Contribution of Stage 1 Diodes . . . 52
4.9.3 Contribution of Stage 2 MOSFETS . . . 52
4.9.4 The Required Heatsink . . . 52
4.10 Physical Layout . . . 53
4.11 Summary . . . 54

5 Control System 55
5.1 Introduction . . . 55
5.2 Specification . . . 55
5.2.1 Range . . . 55
5.2.2 Bandwidth . . . 55
5.2.3 Accuracy . . . 56
5.2.4 Stability . . . 56
5.2.5 Resolution . . . 56
5.3 Block Diagrams . . . 57
5.3.1 System Diagram . . . 57
5.3.2 Stage 1 . . . 57
5.3.3 Stage 2 . . . 59
5.3.4 Estimator . . . 60
5.3.5 Feed-Forward . . . 61
5.4 System Models . . . 61
5.4.1 Stage 1 . . . 61
5.4.2 Stage 2 . . . 64
5.4.3 Integrator . . . 65
5.5 Pole Placement . . . 65
5.5.1 Stage 1 . . . 65
5.5.2 Stage 2 . . . 68
5.6 Measurements . . . 70
5.6.1 Analogue vs Digital . . . 70
5.6.2 Resolution . . . 70
5.6.3 Actual Measurements . . . 71
5.6.4 Estimated . . . 73
5.6.5 Noise . . . 73
5.6.6 Other . . . 75
5.7 Simulation . . . 75
5.7.1 Stage 1 . . . 75
5.7.2 Stage 2 . . . 76
5.7.3 Combined . . . 80
5.8 Soft-Start . . . 82
5.8.1 Rectifier . . . 82
5.8.2 Converter . . . 83
5.9 Firmware Implementation . . . 83
5.9.1 Digital Representation . . . 83

Pectora roborant cultus recti

5.9.2 Time-Step . . . 86
5.9.3 Reciprocal . . . 86
5.9.4 Smoother . . . 87
5.9.5 Soft-Start . . . 88
5.9.6 Timing . . . 88
5.9.7 Protection . . . 89
5.10 Summary . . . 89

6 Digital PWM 91
6.1 Introduction . . . 91
6.2 Resolution . . . 91
6.2.1 Required Resolution . . . 91
6.2.2 Practical Digital PWM . . . 92
6.2.3 Delay-Line . . . 92
6.3 Noise-Shaper . . . 94
6.3.1 Quantisation as a Noise Source . . . 94
6.3.2 Block Diagram . . . 94
6.3.3 Theory of Operation . . . 95
6.3.4 Implementation . . . 96
6.3.5 Performance . . . 97
6.4 PWM Generator . . . 99
6.4.1 Stage 1 . . . 99
6.4.2 Stage 2 . . . 100
6.5 Summary . . . 100

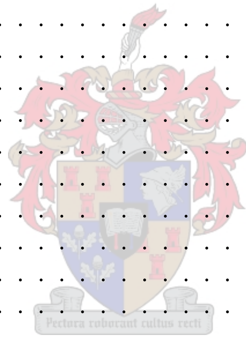
7 Probe 101
7.1 Introduction . . . 101
7.2 Phase-Locked Loop NMR . . . 101
7.2.1 Principle of Operation . . . 101
7.2.2 Phase-Locked Loop . . . 102
7.2.3 Sustaining Resonance . . . 103
7.2.4 Simulation . . . 105
7.3 Hardware Detail . . . 109
7.3.1 Modified Bloch Equations . . . 109
7.3.2 Excitation Coil . . . 110
7.3.3 Detection Coil . . . 110
7.3.4 Using a Single Coil . . . 111
7.3.5 Measurement Technique . . . 111
7.3.6 Parasitic Components . . . 113
7.4 Alternative Control Circuit . . . 115
7.4.1 DDS Operation . . . 115
7.4.2 Frequency Content . . . 116
7.4.3 Block Diagram . . . 116
7.5 Summary . . . 117

8 Controller 119
8.1 Introduction . . . 119
8.2 Hardware . . . 119
8.2.1 Block Diagram . . . 119
8.2.2 Central Processor . . . 120
8.2.3 Power Supply . . . 120
8.2.4 Start-Up Sequence . . . 122

Pectora roborant cultus recti

8.2.5 Programmer . . . 122
8.2.6 Interface Headers . . . 122
8.2.7 ADC . . . 123
8.2.8 DAC . . . 123
8.2.9 Gate Signals . . . 123
8.2.10 Delay Lines . . . 124
8.2.11 EEPROM . . . 124
8.2.12 DDS . . . 125
8.2.13 Mixer . . . 126
8.2.14 Transmission Lines . . . 126
8.2.15 FPGA I/O Pin-Count . . . 127
8.2.16 Physical Layout . . . 127
8.3 FPGA Detail . . . 128
8.3.1 Block Diagram . . . 128
8.3.2 Clock Requirements and Generation . . . 129
8.4 Peripheral Software . . . 130
8.4.1 EEPROM . . . 130
8.4.2 LCD . . . 132
8.4.3 Keypad . . . 132
8.4.4 ADC . . . 133
8.4.5 DAC . . . 133
8.4.6 RS-232 . . . 133
8.4.7 Bus . . . 135
8.5 CPU . . . 137
8.5.1 Justification . . . 137
8.5.2 Block Diagram . . . 137
8.5.3 Instruction Set . . . 141
8.5.4 Boot Loader . . . 142
8.5.5 Tests . . . 142
8.6 Program Layout . . . 142
8.6.1 Core . . . 142
8.6.2 Interface . . . 143
8.7 Summary . . . 143

9 Measurements and Results 144
9.1 Introduction . . . 144
9.2 Practical Test Setup . . . 144
9.2.1 Test Magnet . . . 144
9.2.2 Data Logger . . . 145
9.2.3 Test Conditions . . . 145
9.2.4 Photos . . . 146
9.3 Converter Results . . . 149
9.3.1 Switching Transients . . . 149
9.3.2 Without Feed-Forward . . . 150
9.3.3 With Feed-Forward . . . 153
9.4 Probe Measurements . . . 155
9.4.1 Functionality . . . 155
9.4.2 Circuit Delays . . . 155
9.4.3 Glitches . . . 156
9.5 Summary . . . 156


Pectora roborant cultus recti

10 Conclusions and Future Work **157**
10.1 Introduction . . . 157
10.2 Research Findings . . . 157
10.2.1 Integration . . . 157
10.2.2 Converter . . . 157
10.2.3 Digital Control . . . 157
10.2.4 Digital PWM . . . 158
10.2.5 Probe . . . 158
10.3 Improvements and Future Work . . . 158
10.3.1 Converter . . . 158
10.3.2 Control System . . . 159
10.3.3 PWM Generator . . . 159
10.3.4 Probe . . . 159
10.3.5 Controller . . . 160
10.4 Learning Experience . . . 160
10.5 Summary . . . 160

Bibliography **161**

A Specification **164**
A.1 Introduction . . . 164
A.2 Input From User . . . 164
A.2.1 Analogue . . . 164
A.2.2 Digital . . . 164
A.3 Actions . . . 165
A.4 Output to User . . . 165
A.4.1 Analogue . . . 165
A.4.2 Digital . . . 165
A.5 Power . . . 166
A.6 Dimensions . . . 166
A.6.1 Probe . . . 166
A.6.2 Power Supply and Controller . . . 166
A.7 Other . . . 166

B Formula Derivations **167**
B.1 Introduction . . . 167
B.2 Inductor Design . . . 167
B.3 Transformer Design . . . 169
B.4 First Order Ramp Response . . . 169
B.5 Fourier Expansions . . . 170

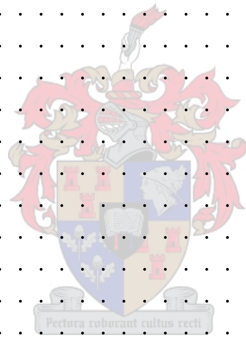
C Theory of NMR **172**
C.1 Introduction . . . 172
C.2 Basic Principle . . . 172
C.2.1 Proton Properties . . . 172
C.2.2 Precession . . . 173
C.3 Relaxation . . . 174
C.3.1 Principle . . . 174
C.3.2 Time Constants . . . 174
C.4 Precession in an RF Field . . . 175
C.4.1 Principle . . . 175
C.4.2 Bloch Equations . . . 175

Pectora roborant cultus recti

C.4.3 Rotating Frame . . . 176
C.4.4 Applying the RF Field . . . 176
C.4.5 Excitation Bandwidth . . . 177
C.5 Continuous Wave NMR . . . 177
C.5.1 Basic Principle . . . 177
C.5.2 Detecting Resonance . . . 178
C.5.3 Field Measurement . . . 179
C.6 Pulsed NMR . . . 179
C.6.1 Basic Principle . . . 179
C.6.2 Determining the Frequency . . . 179
C.6.3 Field Measurement . . . 180
C.7 Summary . . . 180

D Circuit Diagrams 181
D.1 Introduction . . . 181
D.2 Controller . . . 181
D.3 Converter . . . 181
D.4 Probe . . . 181

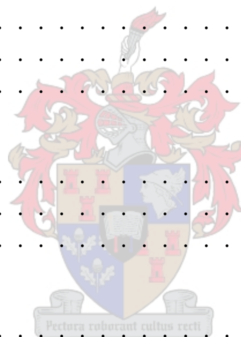
E FPGA Source Code 189
E.1 Introduction . . . 189
E.2 Interface . . . 189
E.2.1 All_Test . . . 189
E.2.2 ADC18 . . . 197
E.2.3 Adder . . . 198
E.2.4 Alternator . . . 199
E.2.5 Arith . . . 199
E.2.6 COMM . . . 201
E.2.7 Controller . . . 202
E.2.8 Counter11 . . . 203
E.2.9 Counter12 . . . 204
E.2.10 Counter14 . . . 204
E.2.11 Counter21 . . . 205
E.2.12 Counter22 . . . 205
E.2.13 Counter5 . . . 206
E.2.14 Counter8 . . . 207
E.2.15 DAC . . . 207
E.2.16 DeadTime . . . 208
E.2.17 EEPROM . . . 209
E.2.18 Interface . . . 213
E.2.19 InterfaceBus . . . 223
E.2.20 InterfaceStack . . . 226
E.2.21 Keypad . . . 228
E.2.22 Keypad_Driver . . . 229
E.2.23 Latch1 . . . 231
E.2.24 Latch8 . . . 231
E.2.25 LCD . . . 232
E.2.26 Probe . . . 234
E.2.27 PWM1 . . . 236
E.2.28 PWM2 . . . 237
E.2.29 RealTime . . . 238
E.2.30 RS232Rx . . . 239



E.2.31 RS232Tx . . . 240
E.3 Controller . . . 242
E.3.1 Controller . . . 242
E.3.2 ADC18 . . . 249
E.3.3 Averager . . . 250
E.3.4 Comm . . . 251
E.3.5 Comp_Avg . . . 257
E.3.6 Control . . . 258
E.3.7 Counter11 . . . 262
E.3.8 Counter15 . . . 263
E.3.9 Counter8 . . . 263
E.3.10 DAC . . . 264
E.3.11 DeadTime . . . 265
E.3.12 Estimator . . . 265
E.3.13 Invert . . . 267
E.3.14 Keypad . . . 268
E.3.15 Keypad_Driver . . . 269
E.3.16 Mul18 . . . 272
E.3.17 PWM1 . . . 272
E.3.18 PWM2 . . . 273
E.3.19 Ramp . . . 275
E.3.20 RealTime . . . 276
E.3.21 RS232Tx . . . 277
E.3.22 Stage1 . . . 278
E.3.23 Stage2 . . . 279

F CPU Source Code 283
F.1 Introduction . . . 283
F.2 Bootloader . . . 283
F.3 Interface . . . 285

G Programmer Source Code 290
G.1 Introduction . . . 290
G.2 Graphical Interface . . . 290
G.3 Programmer . . . 292
G.3.1 Body . . . 292
G.4 Programmer Form . . . 292
G.4.1 Header . . . 292
G.4.2 Body . . . 293
G.5 Assembler . . . 301
G.5.1 Header . . . 301
G.5.2 Body . . . 302
G.6 JComm . . . 310
G.6.1 Header . . . 310
G.6.2 Body . . . 311
G.7 JFile . . . 312
G.7.1 Header . . . 312
G.7.2 Body . . . 313



H Simulation Source **317**
H.1 Introduction . . . 317
H.2 Converter . . . 317
H.2.1 Stage 1 . . . 317
H.2.2 Stage 2 . . . 319
H.3 Probe . . . 321
H.3.1 Header . . . 321
H.3.2 Body . . . 322

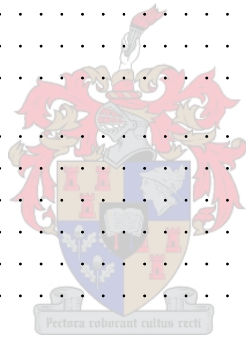
List of Figures

1.1 Van de Graaff accelerator . . . 2

2.1 Existing system block diagram . . . 4
2.2 Analysing magnet at NECSA . . . 5
2.3 Existing magnet dimensions . . . 6
2.4 Existing power supply schematic . . . 8
2.5 Existing probe block diagram . . . 8
2.6 Frequency generator . . . 9
2.7 Existing Controller . . . 10
2.8 Series SMRR . . . 11
2.9 Parallel SMRR . . . 11
2.10 Two-stage topology . . . 12
2.11 Full-bridge topology . . . 12

3.1 General block diagram . . . 16
3.2 Converter block diagram . . . 18
3.3 Probe block diagram . . . 19
3.4 Controller block diagram . . . 19
3.5 Front panel layout . . . 20
3.6 General layout . . . 21

4.1 Overall topology block diagram . . . 25
4.2 Fly-back converter . . . 27
4.3 Push-pull converter . . . 27
4.4 Isolated half-bridge converter . . . 28
4.5 Isolated full-bridge converter . . . 28
4.6 Synchronous buck converter . . . 30
4.7 DC bus waveforms . . . 31
4.8 DC bus waveforms . . . 32
4.9 Supply current waveform . . . 34
4.10 Converter Topology: Stage 1 . . . 34
4.11 Full-bridge waveforms: circuit . . . 35
4.12 Full-bridge waveforms . . . 35
4.13 Full-bridge waveforms . . . 36
4.14 Full-bridge waveforms . . . 36
4.15 Half-bridge rectifier . . . 38
4.16 Full-bridge rectifier . . . 38
4.17 Equivalent circuit of transformer . . . 43
4.18 Line voltage waveforms referenced to neutral . . . 45
4.19 Transformer voltage waveforms . . . 45


Pectora roborant cultus recti

4.20 Isolation voltage waveforms . . . 46
4.21 Converter Topology: Stage 1 . . . 47
4.22 Buck converter waveforms: circuit . . . 48
4.23 Buck converter waveforms: Switched voltage . . . 48
4.24 Buck converter waveforms . . . 48
4.25 Buck converter waveforms . . . 49
4.26 Converter layout . . . 53

5.1 Control system block diagram . . . 57
5.2 Voltage control . . . 57
5.3 Current (hysteresis) control . . . 58
5.4 Current (feedback) control . . . 58
5.5 Stage 1 control system . . . 59
5.6 Stage 2 control system . . . 59
5.7 Integrator block diagram . . . 60
5.8 Estimator block diagram . . . 60
5.9 Stage 1 circuit model . . . 62
5.10 Stage 2 circuit model . . . 64
5.11 Integrator control . . . 65
5.12 Root locus of stage 2 . . . 69
5.13 Multiple measurements . . . 71
5.14 ADC pre-filter circuit diagram . . . 73
5.15 16th order smoother bode plot . . . 74
5.16 Stage 1 ramp response . . . 76
5.17 Stage 1 step disturbance . . . 76
5.18 Stage 2 simulation: 4 H . . . 77
5.19 Stage 2 simulation: 4 H . . . 78
5.20 Stage 2 simulation: 15 H . . . 78
5.21 Stage 2 simulation: 15 H . . . 78
5.22 Stage 2 simulation: 1 H . . . 79
5.23 Stage 2 simulation: 1 H . . . 79
5.24 Converter in simulation . . . 80
5.25 Combined simulation: Step . . . 80
5.26 Combined simulation: Disturbance . . . 81
5.27 Combined simulation: Disturbance . . . 81
5.28 Combined simulation: Disturbance . . . 81
5.29 Bus charging circuit . . . 83
5.30 Signal representation: overall . . . 84
5.31 Signal representation: stage 1 . . . 84
5.32 Signal representation: stage 2 . . . 85
5.33 Signal representation: integrator . . . 85
5.34 Signal representation: estimator . . . 85
5.35 Gain block example . . . 86
5.36 Controller timing diagram . . . 89

6.1 Duty-cycle error . . . 93
6.2 Noise-shaper block diagram . . . 95
6.3 Noise-shaper block diagram . . . 95
6.4 Noise shaper bode plot . . . 96
6.5 Noise-shaper $H(z)$ implementation . . . 97
6.6 Noise-shaper implementation . . . 97
6.7 Noise-shaper results: no filtering . . . 98

Pectora roborant cultus recti

6.8 Noise-shaper results: after filtering . . . 98
6.9 PWM generator: stage 1 . . . 99
6.10 PWM state machine: stage 1 . . . 99
6.11 PWM generator: stage 2 . . . 100

7.1 Probe phase-locked loop . . . 102
7.2 Probe PLL step-response . . . 103
7.3 Locus for sustaining resonance . . . 104
7.4 Locus for a too large limit . . . 104
7.5 Locus using the wrong RF frequency . . . 105
7.6 Probe ramp response . . . 106
7.7 Probe ramp response . . . 106
7.8 Probe ramp response . . . 106
7.9 Probe long-term response: Reference and Response . . . 107
7.10 Probe long-term response: M_x and M_y . . . 107
7.11 Probe long-term response: M_z and Phase . . . 108
7.12 Probe long-term response: Continuous-wave locus . . . 108
7.13 Probe long-term response: Pulsed locus . . . 108
7.14 Probe long-term response: Modified pulsed locus . . . 109
7.15 Probe circuit diagram . . . 112
7.16 Envelope detector . . . 113
7.17 Theoretical measurement error . . . 114
7.18 Simulated measurement error . . . 115
7.19 DDS core . . . 115
7.20 Alternative probe control circuit . . . 117
7.21 Difference amplifier circuit diagram . . . 117

8.1 Control board block diagram . . . 120
8.2 Soft-switch circuit diagram . . . 121
8.3 Power distribution . . . 121
8.4 FPGA start-up sequence . . . 122
8.5 Single-ended LVPECL . . . 124
8.6 Failsafe gate signals . . . 124
8.7 DDS biasing circuit . . . 125
8.8 Simplified DDS biasing circuit . . . 125
8.9 Gilbert cell mixer [1] . . . 126
8.10 Strip-line transmission line . . . 127
8.11 FPGA block diagram . . . 129
8.12 EEPROM driver flow diagram . . . 131
8.13 Keypad connections . . . 132
8.14 CPU block diagram . . . 138

9.1 Test magnet . . . 145
9.2 Transformers . . . 146
9.3 Practical setup . . . 147
9.4 First iteration PCB . . . 148
9.5 Transformer primary voltage and current . . . 149
9.6 Start-up without feed-forward: Voltages . . . 150
9.7 Start-up without feed-forward: Currents . . . 150
9.8 Ripple without feed-forward: Voltages . . . 151
9.9 Ripple without feed-forward: Currents . . . 151
9.10 Step response without feed-forward: Voltages . . . 152

Pectora roborant cultus recti

9.11 Step response without feed-forward: Currents . . . 152
9.12 Step disturbance without feed-forward . . . 153
9.13 Feed-forward voltage reference . . . 153
9.14 Start-up with feed-forward: Voltages . . . 154
9.15 Start-up with feed-forward: Currents . . . 154
9.16 Step response with feed-forward: Voltages . . . 155
9.17 Step response with feed-forward: Currents . . . 155

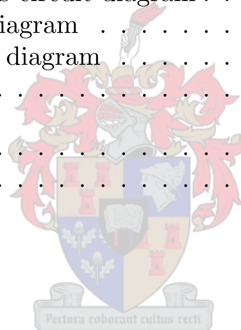
10.1 Proposed soft-start circuit . . . 159

B.1 Inductor dimensions . . . 167
B.2 Effective area of an inductor air-gap . . . 168
B.3 Single order R-L system with feedback . . . 169

C.1 Proton properties . . . 173
C.2 Precessional movement . . . 173
C.3 Continuous wave NMR bandwidth . . . 178
C.4 Pulsed NMR bandwidth . . . 180

D.1 DDS circuit diagram . . . 182
D.2 Analogue circuit diagram . . . 183
D.3 FPGA circuit diagram . . . 184
D.4 Other controller components circuit diagram . . . 185
D.5 Converter primary circuit diagram . . . 186
D.6 Converter secondary circuit diagram . . . 187
D.7 Probe circuit diagram . . . 188

G.1 Programmer screen-shot . . . 291
G.2 Programmer menus . . . 291



List of Tables

2-I Existing system signals . . . 5
2-II Analysing magnet characteristics . . . 6
2-III Estimated magnet constants . . . 7
2-IV Probe frequency allocations . . . 9

3-I General signals . . . 17
3-II Power supply minimum load specification . . . 18

4-I Rectifier comparison . . . 26
4-II Half-bridge vs full-bridge . . . 28
4-III Bus harmonic content . . . 33
4-IV Switching sequence . . . 35
4-V Stage 1 MOSFET on times . . . 41
4-VI Stage 1 MOSFET off times . . . 41
4-VII Transformer configuration properties . . . 43
4-VIII Switching sequence . . . 46
4-IX Stage 2 MOSFET on times . . . 51
4-X Stage 2 MOSFET off times . . . 51

5-I Ripple rejection using a Butterworth prototype . . . 66
5-II Ripple rejection using faster current feedback . . . 67
5-III Stage 1 gains . . . 68
5-IV Stage 2 gains . . . 69
5-V Line frequency harmonics at the load . . . 72
5-VI Stage 1 simulation circuit parameters . . . 75
5-VII Stage 1 simulation circuit parameters . . . 77
5-VIII Smoother: number of samples . . . 88

6-I Stage 1 PWM resolution . . . 92
6-II FPGA PLL configurations . . . 93

8-I FPGA I/O pin-count . . . 127
8-II FPGA clock requirements . . . 130
8-III EEPROM address allocations . . . 131
8-IV Key codes . . . 133
8-V RS-232 Instructions . . . 134
8-VI Bus address allocations . . . 136
8-VII CPU signal descriptions . . . 139
8-VIII Arithmetic unit tasks . . . 140
8-IX Arithmetic stack tasks . . . 140
8-X CPU instruction set . . . 141

9-I Circuit delay (controller side) 156
9-II Circuit delay (Probe side) 156

Nomenclature

Abbreviations

ADC = Analogue to digital converter
BNC = Bayonet Neil-Concelman
CPU = Central processing unit
LVPECL = Low-voltage positive emitter-coupled logic
DAC = Digital to analogue converter
DCCT = Direct-current current transformer
DDS = Direct digital synthesis
EMC = Electromagnetic compatibility
EMI = Electromagnetic interference
FFT = Fast Fourier transform
FPGA = Field programmable gate array
FTW = Frequency tuning word
HF = High-frequency
IC = Integrated circuit
IGBT = Isolated gate bipolar transistor
I/O = Input / output
LCD = Liquid crystal display
LF = Low- / Line frequency
LVDS = Low-voltage differential signalling
MOSFET = Metal oxide field effect transistor
NECSA = Nuclear Energy Corporation of South Africa
NMR = Nuclear magnetic resonance
PCB = Printed circuit board
RF = Radio frequency
rms = Root-mean square
SMA = Sub-miniature version A
SMRR = Switch-mode ripple regulator
SPI = Serial peripheral interface
SPICE = Simulation Program with Integrated Circuit Emphasis

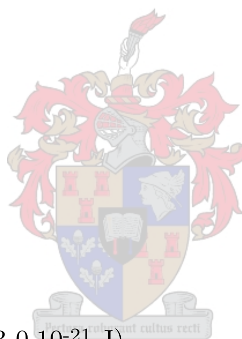
USB = Universal serial bus
USD = United States Dollar

Prefixes

f = femto = 10^{-15}
p = pico = 10^{-12}
n = nano = 10^{-9}
μ = micro = 10^{-6}
m = mili = 10^{-3}
d = deci = 10^{1}
k = kilo = 10^{3}
M = mega = 10^{6}
T = tera = 10^{9}

Units

A = Amphere
Å = Angtrom, 10^{-10} m
b = bit
B = Bel
B = Byte
C = Coulomb
°C = Degrees Celsius
eV = Electron-Volts (160.217 733 $0{\cdot}10^{-21}$ J)



F = Farad
g = Gram
H = Henry
Hz = Hertz
J = Joule
K = Kelvin
m = Meters
M = Molar concentration (mol per litre)
min = Minutes
mol = Mole
ppm = Parts per million
rad = Radians
s = Seconds
T = Tesla
V = Volt

V_p = Volt, peak value
W = Watt
° = Degrees
Ω = Ohm

Constants

These constants are taken from [2].

a	= Molecular radius	= $1.40 \cdot 10^{-10}$ m (Water)
a_0	= Proportionality constant	= $3.281\ 003 \cdot 10^{-3}$ $A \cdot m^{-1} T^{-1}$ (20°C)
b	= Dipole-dipole coupling constant	= $-217.5 \cdot 10^{3}$ Hz (Water)
c	= Speed of light	= $2.997\ 924\ 580 \cdot 10^{8}$ $m \cdot s^{-1}$
e	= Electron charge	= $1.602\ 177\ 330 \cdot 10^{-19}$ C
e	= Natural logarithmic base	= 2.718 281 828 459
h	= Planck's constant	= $6.626\ 075\ 500 \cdot 10^{-34}$ J·s
$\hbar$	= h / 2π	= $1.054\ 572\ 669 \cdot 10^{-34}$ J·s
I	= Quantum spin number	= 1/2 (Hydrogen nucleus)
j	= Imaginary constant	= $\sqrt{-1}$
k	= Boltzmann's constant	= $1.380\ 658\ 000 \cdot 10^{-23}$ $J \cdot K^{-1}$
m_p	= Proton mass	= $1.672\ 623\ 100 \cdot 10^{-27}$ kg
N_A	= Avogadro's constant	= $6.022\ 136\ 700 \cdot 10^{23}$ mol^{-1}
N_v	= Protons per unit volume	= $6.673\ 754 \cdot 10^{28}$ m^{-3} (Water at 20°C)
r	= Distance between spins	= $1.514 \cdot 10^{-10}$ m (Water)
γ	= Gyromagnetic constant	= $2.675\ 221\ 280 \cdot 10^{8}$ $T^{-1} \cdot s^{-1}$ (H^+)
ϵ_0	= Permittivity of free space	= $8.854\ 187\ 817 \cdot 10^{-12}$ $F \cdot m^{-1}$
η	= Viscosity	= $1.04 \cdot 10^{-3}$ $N \cdot s \cdot m^{-2}$ (Water at 20 °C)
λ_{Al}	= Thermal conductivity of aluminium	= 220 $W \cdot m^{-1} \cdot K^{-1}$
μ_0	= Permeability of free space	= $1.256\ 637\ 061 \cdot 10^{-6}$ $H \cdot m^{-1}$
π	= pi	= 3.141 592 653 589 793
ρ_{Cu}	= Resistivity of copper	= $16.78 \cdot 10^{-9}$ Ω·m
ρ_{Al}	= Resistivity of aluminium	= $26.53 \cdot 10^{-9}$ Ω·m
χ_w	= Susceptibility of water	= $-90 \cdot 10^{-6}$

Symbols

Ambiguous symbols will become clear within the context they are used.

A = Cross-section area
B = Flux density

C = Capacitance

D = Duty cycle

D = Rotational diffusion coefficient

d = Digital representation

E = Energy

f = Full-scale value of a digital representation

f = Frequency

f_s = Switching frequency

H = Applied magnetic field

i = Current, instantaneous value

I = Current, phasor or biasing value

J = Spectral density

l = Length

L = Inductance

m = Particle mass

m = Quantum state ($\pm frac12$ for a Hydrogen nucleus)

M = Magnetisation, Magnetic moment per unit volume

N = Number of turns

Q = Charge

r = Radius

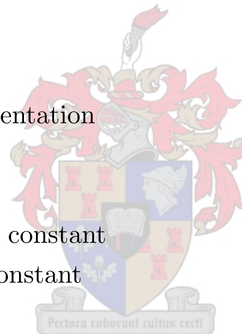
r = Real value of a digital representation

R = Resistance

T = Temperature

T_1 = Longitudinal relaxation time constant

T_2 = Transverse relaxation time constant



v = Velocity

v = Voltage, instantaneous value

V = Voltage, phasor or biasing value

Z = Impedance

β = Bandwidth

δ = Skin-depth

ϵ_{r} = Relative permittivity

μ = Magnetic moment

μ_{r} = Relative permeability

θ = Thermal resistance

τ = Time constant

τ_{c} = Correlation time

ϕ = Phase

χ = Magnetic susceptibility

ω = Angular velocity

Chapter 1

Introduction

1.1 Background

Particle accelerators today have numerous magnets that require high-precision power supplies. These include magnets for analysing, beam-path selection, focusing, etc. Additionally, design specifications for these power supplies are often tight, presenting great challenges regarding their implementation. The typical modern magnet power supply is expected to have a resolution of 16-bit and stability of 10 ppm [3]. Even the choice of electronic component will affect the performance [4].

The project described in this thesis was initiated by the Nuclear Energy Corporation of South Africa (NECSA). Their Van de Graaff accelerator facility is experiencing difficulties with their nuclear magnetic resonance (NMR) based current source controller. This controller, together with a separate current source, is used to generate a constant magnetic field in the range of 0.1 T to 1.2 T. Problems with the existing system include insufficient accuracy, long start-up times, large ripple and unwanted crystallisation of the NMR probe sample . The existing system dates circa 1970 and therefore needs a modern replacement.

A simplified diagram of the accelerator at NECSA is presented in figure 1.1. The motor turns a belt which removes electrons from the dome, charging it positively. The voltage of the dome can be charged to approximately 4 MV with respect to ground. The ion source ionises injected gas molecules (usually deuterium or helium), which are then accelerated down the beam path. The analysing magnet bends the ion-beam through 90°, according to equation 1.1.

$$E = \frac{B^2 e R^2}{2m} \tag{1.1}$$

In this equation, E is the energy of the beam (in eV), B the flux density of the magnet, e the charge of an electron, R the radius of curvature and m the mass of the particle. The equation is valid for single-charged particles.

If the energy is too high, the beam will hit the bottom detection plate. If energy is too low, the beam will hit the top plate. The resulting potential-difference on the plates can

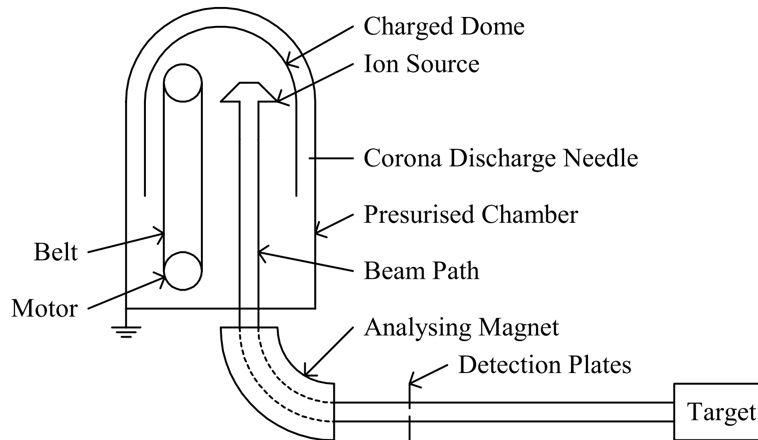


Figure 1.1: Van de Graaff accelerator

thus be measured in order to determine whether the beam energy is too high or low. A control system is in place to adjust the depth of the corona discharge needle, which in turn adjusts the voltage of the dome. The result is the ability to control the energy of the beam.

This project focuses on the power supply for the analysing magnet. By adjusting the magnetic field strength, a specific beam energy is obtained.

1.2 Research Objectives

The original research objective was to develop and implement an nuclear magnetic resonance (NMR) magnetometer and control system for the existing power supply and magnet at NECSA. It was thought that this is too small a project for a masters thesis. As a result, the objectives was extended to include the design of a highly accurate power supply.

This thesis aims to target various aspects of high-performance accelerator magnet power supplies. First the validity of implementing a truly digital controller for a high-performance power supply is investigated. The study encompasses the control system itself as well as digitally generated pulse width modulation (PWM). Two high-resolution digital PWM generation techniques are investigated. The entire digital design is implemented in a single field-programmable gate array (FPGA).

Second, it is intended that the final product will incorporate magnetic field-strength feedback as opposed to using the output current. The field strength is measured by means of NMR. The validity of this approach is examined to discover whether NMR measurements can be done fast and accurately enough to replace traditional current feedback. A thorough theoretical analysis is performed.

Third, the output range of the power supply is relatively large (over a decade) and therefore requires special attention paid to both the NMR probe and the power supply itself. There is much room for research in this area.

1.3 Thesis Overview

An examination of the existing system is performed first, along with a discussion of the existing technology available on the market and in literature. Next, an overall design is offered, including general block diagrams, signal definitions, component choices, etc.

The subsequent chapters deal with a detailed design of a stand-alone power supply and its control system. Output current feedback control is implemented so that experimentation may be performed in the absence of the NMR probe. Simulation results are provided.

A new NMR measurement technique is proposed which promises excellent results with relatively simple circuitry. Practical problems, as well as recommendations for further research into this topic, are discussed. Although very few practical results are available for the probe, extensive simulation results are included. A discussion of NMR theory is included in appendix C, together with a thorough investigation into the physics of an NMR probe.

Next, the various systems within the controller are briefly described. The discussion covers the design and implementation of a central processing unit (CPU) and various peripheral drivers. Finally, practical results are provided, along with a conclusion and recommendations.

Chapter 2

Existing Technology

2.1 Introduction

The project has been developed using the existing system at NECSA as a starting point. A thorough study of this system will follow, including an electrical model of the analysing magnet and a brief explanation as to the operation of the NMR probe and controller.

This chapter then continues with a study of relevant existing technology. Systems available on the market are mentioned and available literature is perused.

2.2 Existing System

Figure 2.1 shows the block diagram of the existing system as it is implemented at NECSA. Each block is a free-standing module.

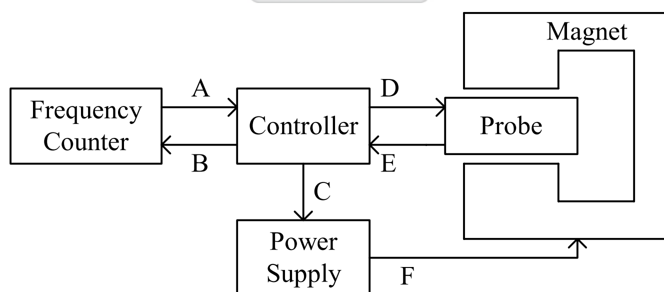


Figure 2.1: Existing system block diagram

The 'frequency counter' is an HP model 5244L high-precision frequency counter. It provides a precision frequency reference to the controller. It also measures the actual frequency of the NMR probe oscillator. Table 2-I provides a list and short description of the signal groups A through F.

Table 2-I: Existing system signals

Group	Signal	Description
A	Frequency Reference	An 1 MHz reference signal is sent to the controller
B	Frequency Out	The radio-frequency (RF) signal from the probe is amplified and sent to the frequency counter for measurement
C	Reference Voltage	A 5.6 $\text{A}\cdot\text{V}^{-1}$ reference used to set the output of the power supply.
D	Power	The probe requires ±15 V
	RF Level Set	A signal in the range 0 V to 2 V used to set the amplitude of the RF signal
	Modulation Signal	A 50 Hz, ±110 mA sinusoidal signal that passes through the modulation coil in order to generate a small field-strength deviation in the vicinity of the NMR probe
	Frequency Set	A signal in the range 3 V to 15 V used to set the frequency of the VCO in the probe. The actual frequency depends on the probe used.
E	NMR Pulses	Pulses of approximately 136 mV in amplitude which occur at the point of resonance
	RF Level	A voltage representing the peak-to-peak amplitude of the RF signal
	RF Signal	The actual RF signal
F	Magnet Current	The actual current through the magnet coils

2.2.1 Analysing Magnet

Figure 2.2 shows a photo of the analysing magnet at NECSA. For this project an electrical model of the magnet is required. Its characteristics, as revealed by the datasheet, are listed in table 2-II.

Figure 2.2: Analysing magnet at NECSA

Unfortunately, the inductance of the magnet is not given. Also, no accurate measurement of inductance could be performed on site. The inductance can, however, be mathematically

Table 2-II: Analysing magnet characteristics

Property	**Value**	**In SI units**
Core	Solid iron	
Air-gap	0.812 inches	20.6 mm
Number of turns (two coils)	1 250	1 250
Coil resistance	14.4 Ω	14.4 Ω
Rated current	15 A	15 A
Rated flux density	12 000 gauss	1.2 T
Radius of curvature	18 inches	457 mm
Field homogeneity (0.9 T)	1 part in 500 over beam path	2000 ppm
Height	94 inches	2 388 mm
Width	34.75 inches	883 mm
Length	72 inches	1 829 mm
Weight	5 000 pounds	2 268 kg
Beam-to-floor height	42 inches	1 067 mm

estimated. The formula for inductance (adapted from [5; 6] and derived in section B.2 on page 167) is given by equations 2.1 and 2.2.

$$L = n^2 \left(\frac{A_{air}\mu_{air}}{l_{air}} // \frac{A_{core}\mu_{core}}{l_{core}} \right) \tag{2.1}$$

$$B = \frac{ni}{A_{air}} \left(\frac{A_{air}\mu_{air}}{l_{air}} // \frac{A_{core}\mu_{core}}{l_{core}} \right) \tag{2.2}$$

Where:

$$x//y = \frac{x \cdot y}{x + y} \tag{2.3}$$

Approximate dimensions of the core cross-section at the air-gap are given in figure 2.3.

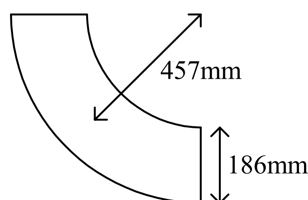


Figure 2.3: Existing magnet dimensions

The area can then be calculated using equation 2.4 where r is the radius of curvature and w the core width.

$$A = \frac{\pi}{4}\left[\left(r + \frac{w}{2}\right)^2 - \left(r - \frac{w}{2}\right)^2\right] \tag{2.4}$$

The magnetic field-lines fringe at the air-gap, effectively increasing the area. According to [6], the equivalent air-gap can be approximated by increasing the dimensions by half the air-gap length along all the edges. Estimates of the constants are listed in table 2-III.

Table 2-III: Estimated magnet constants

Constant	**Value**
n	1 250
A_{air}	0.158 m^2
μ_{air}	$1 \cdot \mu_0$
l_{air}	20.6 mm
A_{core}	0.139 m^2
μ_{core}	$10\,000 \cdot \mu_0$
l_{core}	2 m

These estimates result in an inductance of 15 H. As a test for validity, the rated current should result in the rated flux density. This can be calculated by equation 2.2, yielding 1.13 T at 15 A. In the core the flux density is 1.29 T at 15 A. Within the beam-path the flux density is somewhere between these two extremes and therefore consistent with the datasheet value of 1.2 T.

The effect of capacitive coupling between windings and the eddy-currents in the solid-iron core is negligible at low frequencies (<10 Hz). Also, high-frequency voltage ripple would manifest itself as an increase in eddy-currents in the core, rather than as an increase in the magnetic flux ripple. The power supply load is thus modelled as a 14.4 Ω resistor in series with a 15 H inductor.

2.2.2 Current Source

The existing power supply is a physically large, free-standing power supply. It operates by using a servo motor to adjust a variac so that the output current follows a given voltage reference. It has a transfer constant of 5.6 $A \cdot V^{-1}$. A much-simplified schematic is given in figure 2.4.

Its bandwidth and settling time was not measured.

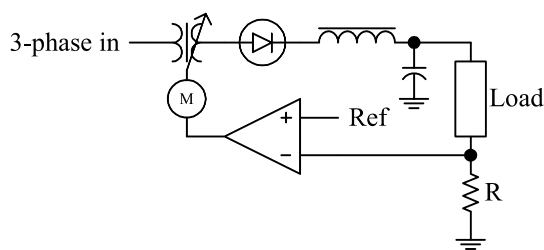


Figure 2.4: Existing power supply schematic

2.2.3 Probe

The existing system uses an NMR technique known as continuous wave NMR (explained in section C.5 on page 177) to probe the field strength. This technique is not as accurate as pulsed NMR, but requires less complex electronics. Figure 2.5 shows the block diagram of the probe [7].

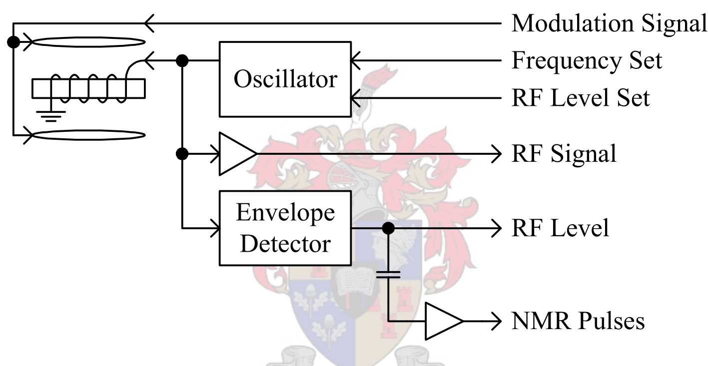


Figure 2.5: Existing probe block diagram

The change in Q-factor of the NMR coil (at resonance) is relatively small. Due to this fact, a marginal oscillator has been implemented. This type of oscillator is very sensitive to changes in inductor characteristics (such as Q-factor). In this system, NMR resonance will lead to amplitude changes in the RF signal.

The change in amplitude is still very small. Also, amplitude drift may occur due to environmental factors (especially temperature), making DC measurement impractical. To solve these problems, the magnetic field in the vicinity of the NMR coil is modulated by a superimposed 50 Hz sinusoidal field. This field is nominally 0.4% of the main field. The effect of this technique is to take the coil in and out of resonance repetitively, thus producing pulses. These pulses are easily amplified by AC-coupled amplifiers and completely compensate for any environmental factors.

The addition of $Fe(NO_3)_3$ decreases the time it takes for resonance to be established or ceased. The concentration of the solution is 0.01 M. This enables the sample to be taken in

and out of resonance at a greater rate.

The frequency is adjusted by changing the biasing voltage on a varactor diode. This method unfortunately gives the oscillator a narrow range of operating frequencies. As a result, three different NMR coils are used, depending on the frequency needed. These are listed in table 2-IV.

Table 2-IV: Probe frequency allocations

Probe	L [μH]	f [MHz]	B [mT]
I	1.6	7.8–14.3	183–336
II	0.5	14–25.6	329–601
III	0.15	25.5–46.4	599–1 090

2.2.4 Controller

The magnetic field strength is controlled by adjusting the power supply reference in an analogue control loop. Using thumbwheels, the frequency of the probe oscillator is set. A phase-locked loop frequency multiplier, shown in figure 2.6, is then used to ensure an RF frequency accurate to the nearest 1 kHz. The RF signal amplitude is set by means of a knob on the front panel. An analogue meter shows the actual RF amplitude.

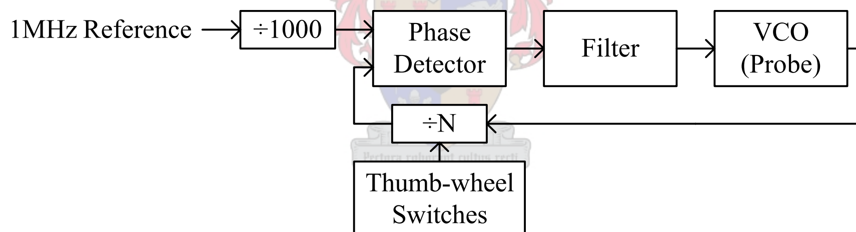


Figure 2.6: Frequency generator

Next, these same thumbwheels are used to generate a bias voltage for the power supply reference. The control system works around this bias point.

In the absence of NMR pulses, the power supply reference passes through a Schmitt-trigger. This signal is then integrated in order to provide an oscillating signal around the bias point. The effect is a search for the resonance point.

When NMR pulses are present, however, the modulation signal is ‘sampled’ by a sample-and-hold circuit on every NMR pulse. This in turn provides an error signal, which then passes through the integrator. The controller guides the resonance point towards the position where the modulation signal passes through zero. A block diagram of the entire controller is presented in figure 2.7.

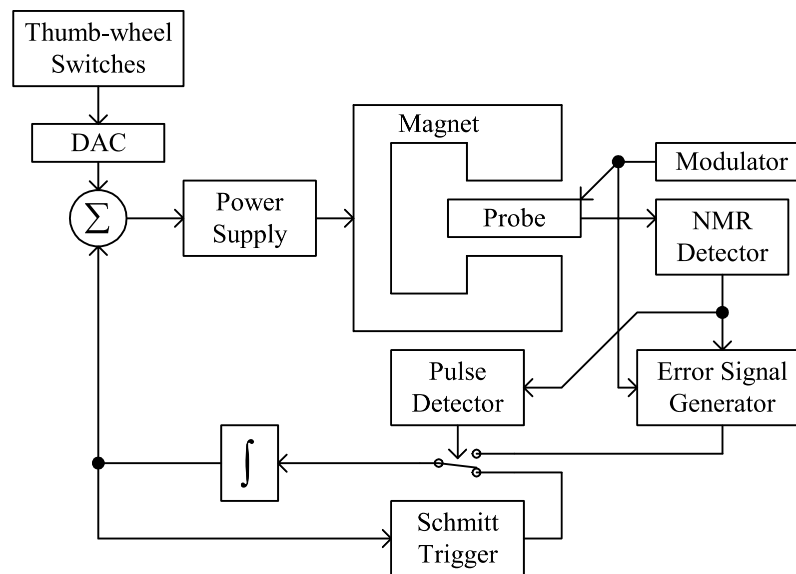


Figure 2.7: Existing Controller

2.3 Products on the Market

2.3.1 Digital Teslameter

The DTM-151 from GMW Associates (www.gmw.com) is a digital hall-probe based teslameter providing 20-bit resolution, 0.01% accuracy and 10 Hz measurement rate. It costs USD 3 670 and has an RS-232 interface.

The PT-2025, also from GMW associates, is a digital NMR based teslameter with 1 μT resolution (24-bit), 5 ppm accuracy, 1 Hz measurement rate (or 10 Hz at reduced resolution) and 1% per second tracking rate. It costs USD 17 580 and also has an RS-232 interface.

2.3.2 Current Source

Most current sources on the market are just that – current sources. The computer provides a current reference and the supply follows it. This implies that there is always a need for external field measurement equipment and a feedback loop to ensure that the field is correct. Normally these current sources are stable enough to have a person (operator) act as the feedback loop. The operator may increase or decrease the current reference manually until the teslameter shows the correct field strength.

2.4 High Performance Magnet Power Supplies

This section deals with recent papers published on the subject of magnet power supplies, specifically for accelerator applications.

2.4.1 Topologies

High-current power supplies can often not be implemented using fast-switching converters. They resort to using phase-controlled rectifiers [8], which produce large low-frequency harmonics. The solution is to use a switch-mode ripple regulator (SMRR) [8–10]. An SMRR is a fast-switching MOSFET or IGBT-based converter (normally in full-bridge configuration) that is either in series or in parallel with the load. These topologies are presented in figures 2.8 and 2.9.

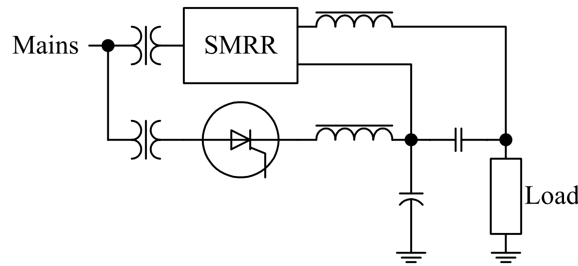


Figure 2.8: Series SMRR

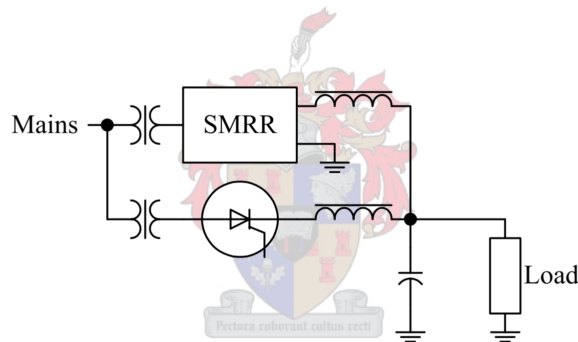


Figure 2.9: Parallel SMRR

The advantage of this topology is immediately evident. A phase-controlled rectifier is a high-power, slow-reacting converter. The SMRR is a medium-power, fast-reacting converter. The combination of these two in the above topology yields a high-power, fast-reacting converter – two qualities very difficult to combine otherwise.

Since the load is inductive, it is sensitive to current ripple and non-sensitive to voltage ripple. The series SMRR minimises voltage ripple while the parallel SMRR minimises current ripple. Series SMRR is thus more popular.

A useful method used in medium-power supplies is to follow a phase-controlled rectifier with a buck converter [10]. The phase-controlled rectifier can then be used to soft-charge the bus capacitors at start-up, after which the buck converter takes over. This buck converter is then followed by a full-bridge. The topology is depicted in figure 2.10.

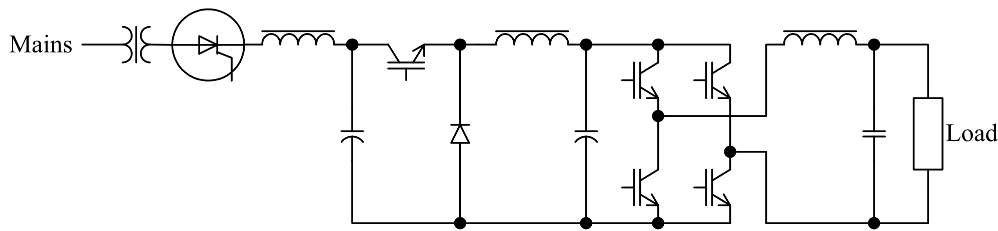


Figure 2.10: Two-stage topology

The buck converter is controlled with a voltage feedback loop and the half-bridge with current feedback. The advantages of this circuit include greater controllability of small output currents as well as bi-directional output.

The topologies discussed thus far all require a large line-frequency transformer. It is, however, advantageous to use a smaller high-frequency transformer instead. A popular topology that makes use of a high-frequency transformer is the isolated full-bridge [11]. This design is shown in figure 2.11. The final choice of topology is further discussed in section 4.4 on page 25.

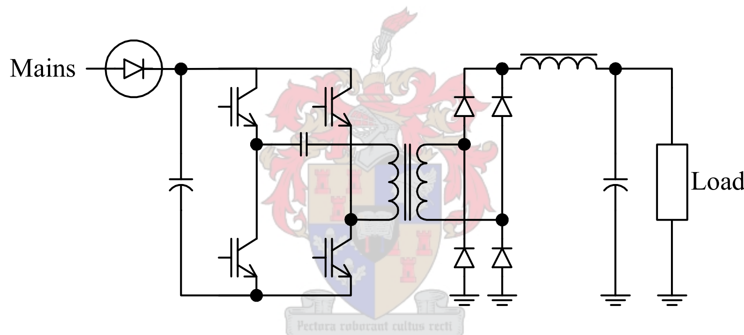


Figure 2.11: Full-bridge topology

2.4.2 Feed Forward Techniques

Feed-forward control is a popular technique used to eliminate output line-frequency harmonics. Any feed-back control loop has the disadvantage that a significant change in output has to arise before the control system rectifies it. Feed-forward techniques anticipate this effect and perform corrections before a disturbance is perceived on the output. Among the many implementations of feed-forward, three were considered.

The first feed-forward strategy to be considered is the instance of its implementation within a typical control loop (consisting of a fast voltage inner loop and a slow current outer loop). The supply voltage is measured and included into the calculation of switching times [3].

The second feed-forward scheme is to make the saw-tooth (the signal used to generate PWM) amplitude proportional to the bus voltage. This ensures a higher duty-cycle for lower bus-voltages and vice versa [12].

The third method uses an SMRR [9] (figure 2.8). The line voltage and output of the phase-controlled rectifier are both considered in the calculation of the SMRR duty-cycle. The output of the line-frequency filter is predicted and then compensated for. The fast dynamic response of an SMRR makes this an attractive option for large power supplies.

2.4.3 Digital Control

Digital control has numerous advantages – superior noise immunity, predictability, high accuracy and ease of modification, to name a few. A digital interface means that many power supplies can be connected to a single digital network. Also, measurement noise can be filtered out. One great disadvantage, however, is digitisation error. It is for this reason that the PWM is usually generated with analogue circuitry [11].

Analogue PWM is generated by comparing a reference signal to a saw-tooth. This saw-tooth is often noisy and results in PWM jitter. With new technology (explained further in chapter 5) it is possible to digitally create PWM with high enough resolution (>18-bit) for digitisation errors to become negligible. The two techniques considered in this thesis are the delay-line and the noise-shaper [13].

Many digital control schemes exist, of which the most powerful is full state feedback [3; 14]. Full state feedback may be implemented with analogue circuitry, but this is very complicated. It is much simpler to use a digital implementation. An estimator [3] may be used in order to obtain the value of states that are not directly measurable.

2.5 Field Measurement Techniques

2.5.1 Calibrated Current Measurement

The measurement most often used in controlling magnet power supplies is the output current. This current may be measured using a current-sensing resistor or with a so called direct-current current-transformer (DCCT). A DCCT is a hall-probe based current transducer, such as a LEM module.

Calculating the magnetic field strength using this current measurement is always slightly inaccurate and therefore needs calibration. The true magnetic field is measured using an NMR technique and the current measurement is then adjusted to match the actual field-strength. This calibration has to be performed quite often, as current measurement drifts with time, temperature and other environmental factors. The measurement itself is also noisy. Advantages of this method include high-bandwidth measurements (>10 kHz) and low cost.

2.5.2 Calibrated Hall Effect Probe

A better alternative to current measurement is using a hall-effect probe [15], which measures the field-strength directly. For larger flux densities (>300 mT), however, the hall-probe measurement is non-linear. It drifts with temperature (600 ppm per °C) and is prone to being noisy. Regular calibration is once again required [16]. As with calibrated current measurement, this method provides high bandwidth.

2.5.3 Continuous Wave NMR

Frequency measurement and creation by means of a crystal oscillator is highly immune to noise and environmental factors. Since NMR flux density measurement techniques are based upon the frequency of resonance, it provides a highly accurate measurement. Continuous wave NMR [17] works on the principle that a coil undergoing resonance experiences a change in impedance. The energy absorbed by the NMR sample (usually water) manifests itself as a series resistance added to the inductance of the NMR coil.

A small radio-frequency (RF) signal is applied over the NMR coil. The frequency at which resonance occurs is then determined, but the response is too small to distinguish it continuously. To solve this problem, the sample is taken in and out of resonance repeatedly. This is performed be sweeping either the local field-strength or the frequency. The field-strength is then determined by using the highly linear relationship between the resonance frequency and flux density. The constant involved (the gyromagnetic ratio γ) has a value of $2.675\ 221\ 280 \cdot 10^{8}\ \mathrm{s}^{-1} \cdot \mathrm{T}^{-1}$ for a hydrogen nucleus. An accuracy of 100 ppm and 50 Hz bandwidth is easily obtainable.

Higher bandwidth measurements are possible by adding specific impurities to the water [18]. The effect is to decrease the time-constant of relaxation (explained in chapter 8). A bandwidth of 1 kHz is obtainable using this technique.

2.5.4 Pulsed NMR

Pulsed NMR is a very popular alternative to continuous-wave NMR [16; 18–20]. Pulsed NMR uses a large RF pulse to induce an oscillation. This oscillation is then detected by the NMR coil. The frequency is determined either by means of frequency-counting the detected signal or by calculation of the fast Fourier transform (FFT).

An initial accurate measure of field-strength must be attained in order to calculate the frequency of the pulse. A pulse of the wrong frequency will not induce an oscillation. This measurement may be found by continuous wave NMR, calibrated hall-probe or calibrated current measurements. A phase-locked loop can then be used to keep track of the resonant frequency needed for the next pulse [20; 21].

As with continuous-wave NMR, faster update times (≈1 kHz) are made possible by adding impurities to the water sample [18]. The effect of these additions unfortunately result in decreased signal strength. Therefore the detection amplifier must have a smaller bandwidth in order to detect the signal.

The L-C filter used within the amplifier may be tuned by means of tuning diodes [19]. The result is an amplifier that has a narrow bandwidth, but an adjustable centre frequency. The centre frequency can therefore be set to coincide with the Lamour frequency (see section C.2.2 on page 173) of the sample.

Pulsed NMR has superior accuracy (one part in 10^8 are possible [18; 22]). Drift and noise do not pose much of a problem.

2.6 Summary

Investigation into the existing system provided insight into the solution to the problem. The same basic principles will be used. These include using NMR as a field-strength measurement strategy, an output range of 1 A to 15 A, a resolution better than 20 μT, etc. The load for the power supply may be modelled as a 14.4 Ω resistor in series with a 15 H inductor.

Various topologies are available. The most popular in the power range similar to this project are the buck and full-bridge converters. The isolation transformer is more often a line-frequency transformer, although some examples do use a high-frequency one.

A dual-stage topology is also popular. An example is a phase-controlled rectifier followed by full-bridge or SMRR. The advantage of dual-stage topologies is there large dynamic range and superior ripple rejection properties.

Digital control is prevalent, although the PWM generator is usually analogue. There are techniques to improve the digital PWM to an acceptable resolution – for instance, using a delay-line or noise-shaper to improve digital PWM.

The field-strength can be determined in many ways. Power-supplies on the market commonly make use of a calibrated current measurement. This may be supported by an external hall-effect probe or NMR magnetometer. The existing system uses NMR to measure the field and includes a current-controlled power supply in the control loop.

Chapter 3

Design Overview

3.1 Introduction

This chapter provides a preliminary design of the entire system. An effective method of planning is to divide the system into smaller, manageable modules. This process was performed for the project and is described in this chapter. Each module, or block, is discussed briefly.

3.2 Block Diagrams

3.2.1 General Block Diagram

An overall block diagram of the entire system is presented in figure 3.1. The converter, power supply, controller and front panel are separate modules contained within a single 19″ rack casing. The probe and magnet are situated apart from the controller, outside the casing. Table 3-I lists each signal group in more detail.

Pectora roborant cultus recti

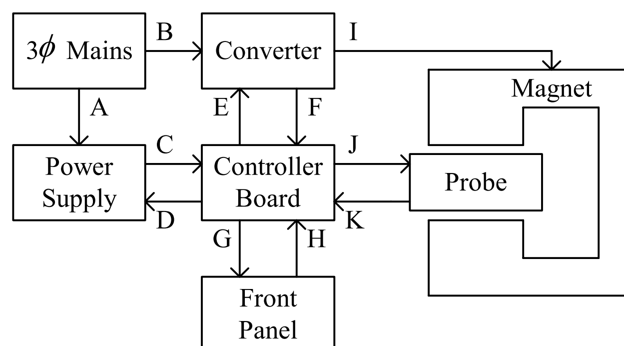


Figure 3.1: General block diagram

Table 3-I: General signals

Group	Signal	Description
A	Mains	Single phase – Live and Neutral
B	Mains	3-phase
C	Power	-12 V; -5 V; Ground; 3.3 V; 5 V; 12 V
	Power Good	5 V implies that the power voltages are within acceptable range
		0 V implies that one or more power voltages are out of range
D	Power Supply On	0 V switches the power supply on
		5 V switches the power supply off
E	Power On	5 V switches the 3-phase mains on by means of triacs,
		0 V switches the 3-phase mains off
	Switching Signals	The output from the PWM signal generators – 4x single-ended LVPECL signals that drive the MOSFETS
F	Measurements	4x analogue measurements in the range 0 V to 5 V
G	LED	5x LEDs: Power; Field High; Field Low; Field Locked; Error
	LCD	LCD Signals: 3x Power; 2x Control; 8x Data
	RS-232	To the DB-9 RS-232 port
	Keypad Probe	4x Keypad probe signals (Y)
H	Keypad	4x Keypad return signals (X)
	Power On	The power on button
	Power Off	The power off button
	Reset	The reset button
I	Magnet Current	The actual magnet current
J	Power	The probe requires 12 V
	Excite	An RF signal sent to the probe to drive the excitation coil – superimposed on the Power signal
K	Detect	An RF signal returning from the probe

Pectora roborant cultus recti

3.2.2 Mains and Power Supply

A 400 V (line-line), 13 kVA, 4-wire 3-phase connection is available for use in this project. Triacs are used as a mains switch due to their small size and lack of moving parts.

One of the phases is used to power the small-signal electronics. On the primary side of the magnet power supply, a small 2 VA, 50 Hz, 230:12 transformer is used to drive the MOSFET-driver circuitry. For the secondary side and the control board, an ATX computer power supply is used. This has the advantage of providing various voltages (-12 V; -5 V; ground; 3.3 V; 5 V; 12 V) as well as ‘power-on’ remote control and a ‘power-good’ signal. It also has an automatic shut-down feature when the output voltages or currents fall outside specifications, thus providing protection for sensitive components.

The only disadvantage is the large quiescent currents needed for the power supply to remain regulated and ‘on’. Table 3-II shows the minimum currents required for the WIN 250PS-SPC power supply from Wintech. The resulting total minimum power is 99.54 W. The total power drawn from a typical FPGA controller board is is the order of 10 W.

Table 3-II: Power supply minimum load specification

Output voltage	Minimum current
-12 V	120 mA
-5 V	120 mA
3.3 V	3 A
5 V (Standby)	400 mA
5 V	8 A
12 V	3.8 A

The total loss is therefore approximately 90 W. This is less than 3% of the converter rating and thus ignored. The minimum currents are obtained by using resistors.

3.2.3 Power Converter

A full description of the power converter design is presented in chapter 4. The need for two stages is clear. The first stage provides isolation and a variable bus-voltage for the second stage. The topologies in mind for the first stage include the half-bridge and full-bridge. The second stage provides a means of implementing feed-forward control. Provision has been made for a synchronous buck converter. This implies a total of four gate signals. The converter is presented in figure 3.2.

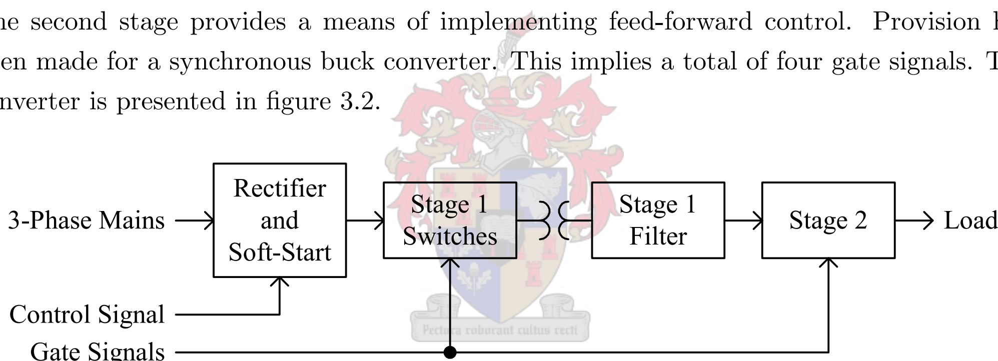


Figure 3.2: Converter block diagram

3.2.4 Probe

The probe is an NMR-based probe. The controller generates an RF signal of a specific frequency and sends it to the probe over 50 Ω co-axial line. The probe then amplifies this signal and drives the excitation coil with it. Also included in the probe is a detection circuit. It consists of a detection coil and amplifier. The detected signal is sent back to the controller over a separate 50Ω co-axial line. A block diagram is presented in figure 3.3.

Power is delivered to the circuit as a 12 V bias voltage on the excitation signal. This reduces the number of cables needed. BNC connectors are used.

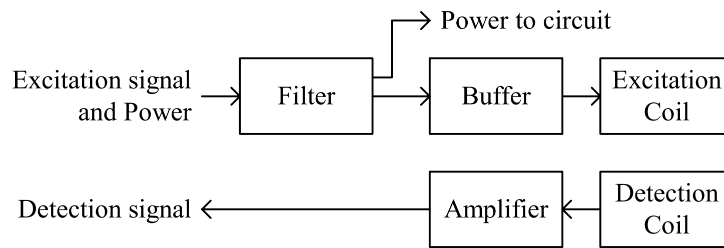


Figure 3.3: Probe block diagram

3.2.5 Controller

The controller consists of many components, most of which are implemented digitally. A block diagram is given in figure 3.4.

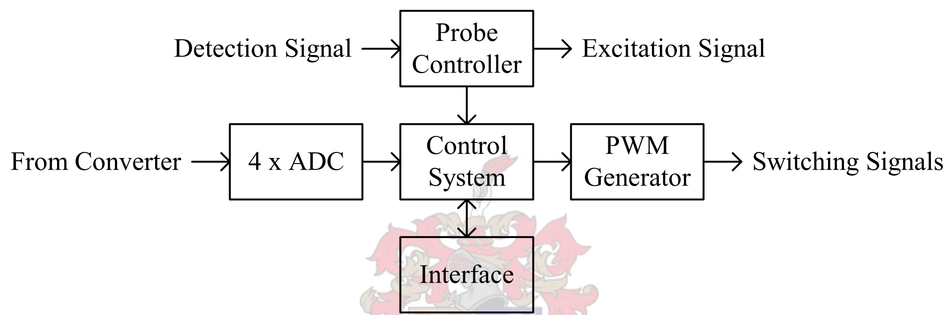


Figure 3.4: Controller block diagram

Pectora roborant cultus recti

The probe controller includes both an excitation pulse generator and detection circuitry. It also includes the components required to measure the field-strength. This measured field-strength is presented to the control system for further processing.

The four signals to be converted are discussed in section 5.6.3 on page 71. They are the first filter inductor current, the load current, the first stage output voltage and the load voltage.

The control system is implemented digitally, and therefore requires analogue to digital converters. ADCs typically have an analogue input in the range of 0 to 5 V. All analogue measurements (both voltage and current) are thus scaled to fit within that range prior to being sent to the control board. Provision is made for four signals, although it is not clear at this stage which measurements to take. Ribbon-cable is used for this connection with every second wire connected to ground.

The control system sends the desired duty-cycles to the PWM generator, which then generates the switching signals.

The interface hardware is discussed in section 3.3.1. The unit may be fully computer-controlled or free-standing. It is intended to implement a menu-driven interface.

3.3 Overall Layout

3.3.1 Front Panel

Figure 3.5 shows the design for the front panel of this project. First, there should be a power button. In this case, separate 'on' and 'off' buttons are used. There is also a 'reset' button.

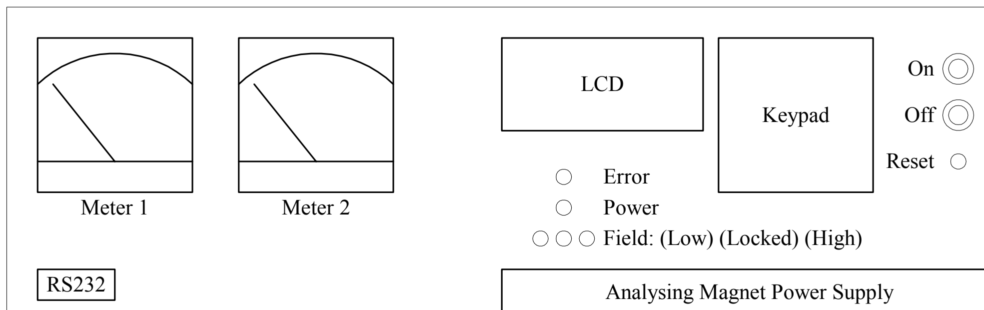


Figure 3.5: Front panel layout

Any system has indicator LEDs. There should be an indication that the power is on. A green LED is allocated for this property. Next, there should be an indication of whether the magnet power supply is occupied in tracking or settled on the desired value. Three LEDs are used for this task: two red (field low and field high) and one green (field locked). The two red LEDs are 'on' simultaneously when an NMR field-strength measurement is not yet available. Finally, a red LED is included for situations of abnormality.

A computer interface is required. A standard RS-232 serial connection has been chosen for simplicity, which can later easily be upgraded to a USB or Bluetooth interface. A DB-9 connector is thus included on the front-panel. USB is not included at this stage of the design because a USB development is not part of the research objectives.

The unit should have the option of being free-standing, thus requiring a keypad (standard 4x4) and LCD. A 4x16 LCD display was chosen. A simple menu-driven interface would be perfect for setting the desired field-strength, re-calibrating the probe, acquiring measurements and other functions.

Finally, many scientists enjoy an analogue meter, since it can be read at a glance rather than requiring interpretation of numbers. This idea was suggested by scientists at iThemba Labs. Two of these meters are included on the design of the front panel. They are driven by a dual DAC and may be configured to show any combination of measurements.

3.3.2 Layout

A 19″ rack module is convenient for installation in the NECSA accelerator facility. The inside layout of the casing designed for this project is shown in figure 3.6. The aluminium heat sink

also acts as an EMI shield between the noisy converter and the sensitive controller-board. The converter is considered noisy because of the high currents being switched at relatively high frequencies. The controller board is considered sensitive because of the high resolution (20 μV) ADCs.

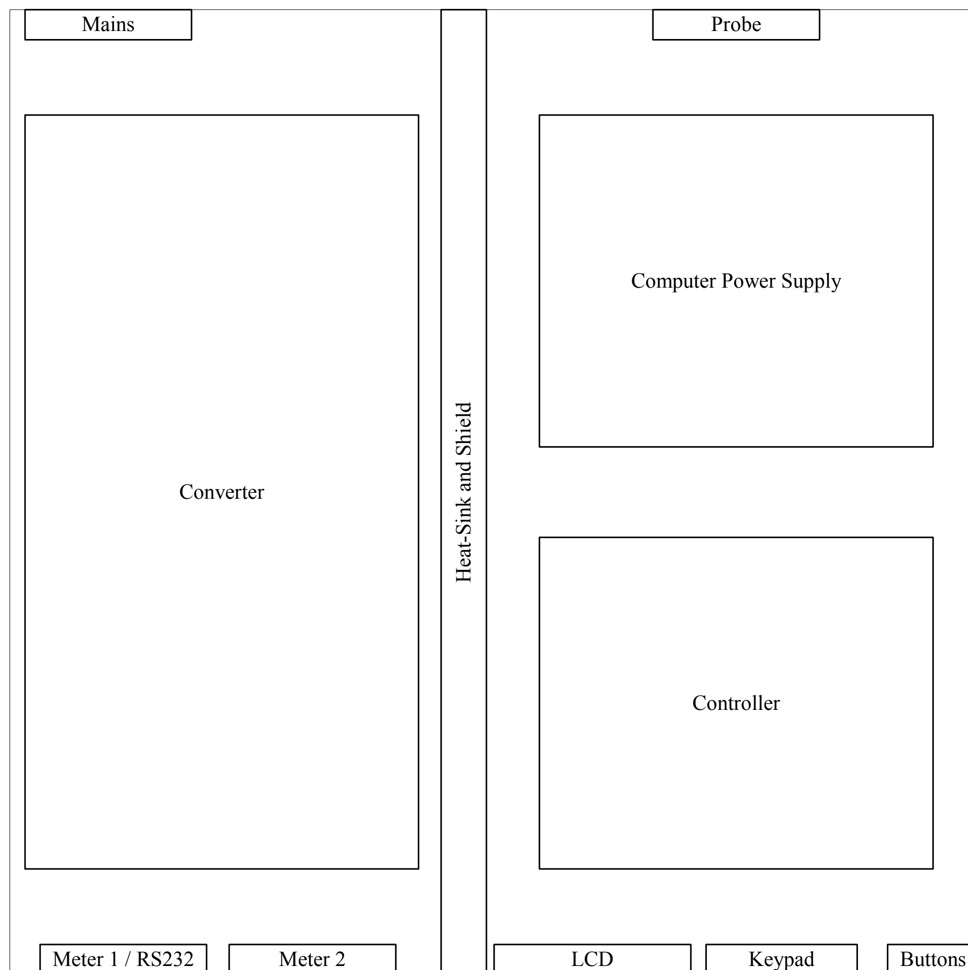


Figure 3.6: General layout

3.4 Summary

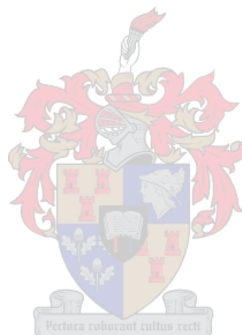
This chapter successfully split the design into four major components. These are the converter, control system, probe and controller. The chapters that follow provide detail on each of these parts.

The converter was subdivided into a rectifier, first stage, high frequency transformer and a second stage. The dual stage topology offers many advantages regarding range and

controllability.

It was decided to use an NMR-based probe. The block diagram of the probe was presented. A brief overview of the controller board was also presented.

A discussion of the physical interface and intended layout concluded this chapter.



Chapter 4

Converter

4.1 Introduction

The full design for the converter hardware is presented in this chapter. The requirements are listed and different converter topologies are considered. The passive components are carefully designed for optimum performance. The non-ideal properties of the converter are discussed. Waveforms are presented and analysed. The physical layout of the converter is also discussed.

4.2 Definitions

For purposes of this thesis, some definitions are required. For clarity, some of these definitions are different to the traditional ones.

Accuracy — The deviation of the output value from the absolute reference

Duty-cycle — The ratio between pulse width and switching period

Half time — The time taken for a decaying property (such as the voltage across an inductor) to decrease by half

Pulse width — The time from rising edge to falling edge of the voltage waveform at the input of the L-C filter

Switching frequency — The reciprocal of the switching period

Switching period — The time between rising edges of the voltage waveform at the input of the L-C filter

4.3 Specification

4.3.1 Source of Power

The converter is supplied with a 4-wire 3-phase, 400 V (line-line), 13 kVA connection to the South African power grid (ESKOM). The voltage waveform is seldom perfectly sinusoidal, balanced or constant. A maximum variation of 10% should be incorporated into the design.

4.3.2 Power Rating and Load

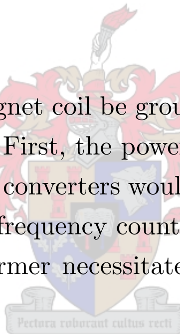
It is required to drive the magnet with a DC current I of no more than 15 A. The resistance R of the coil is 14.4 Ω. This results in a power rating P of 3.24 kW (equation 4.1), reading closer to 3.5 kW when losses are considered.

$$P = \frac{I^2}{R} \tag{4.1}$$

Although inductance of the load is 15 H, the specification requires that the ripple be less than 10 ppm for any inductive load larger than 1 H. The inductance value that provides the worst-case calculation is used.

4.3.3 Isolation

It is required that one side of the magnet coil be grounded, which implies an isolated power supply. This has many implications. First, the power rating is small enough to use a high-frequency transformer. Higher power converters would require a line-frequency transformer, physically much larger than its high-frequency counterpart.



Second, a high-frequency transformer necessitates a more meticulous design, which is discussed in section 4.7.

4.3.4 Range

The current output range of the converter is specified as 1 A to 15 A. This translates to a voltage output range of 0 V to 216 V for a 14.4 Ω load.

A larger output voltage is required to charge the load inductor from minimum to maximum output current within the acceptable 5 s. The same constraint lies with the opposite situation of discharging the load inductor. According to equation 4.2 [23], the given load will discharge from 15 A to 1 A in 2.82 s when the output voltage v is 0 V. An output voltage of zero is used in this calculation because the converter is designed to be unidirectional, which means that the output cannot be negative. This leaves 2.18 s in which the controller must settle to 10 ppm. For a single-order system with a bandwidth of 1 Hz, the 10 ppm settling time is less than 1.83 s (see section 5.2.2 on page 55).

$$t = \frac{L}{R} \ln\left(\frac{v - i_{start}R}{v - i_{end}R}\right) \tag{4.2}$$

Similarly, to charge the inductor from 1 A to 15 A within 2.82s requires a voltage of 230.4 V. Providing for losses and other non-ideal properties, a range of 0 V to 250 V was chosen.

4.3.5 Stability

According to the specification in appendix A.3 on page 165, the system must have a stability of 10 ppm.

4.3.6 Audible Noise

The finished product will be installed close to the accelerator operator. It is thus required that the switching noise be outside the human audible range of 20 Hz to 20 kHz [24].

4.4 Topology

4.4.1 Overall

In section 3.2.3 on page 18, the overall topology was derived to have a rectifier, two stages and a transformer. The block diagram is repeated in figure 4.1.

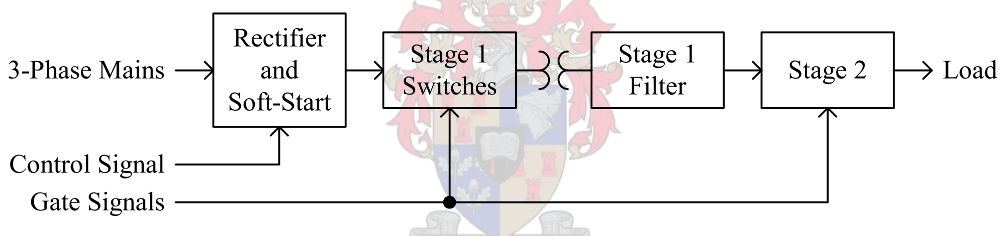


Figure 4.1: Overall topology block diagram

The reasons for a dual stage topology is more clear at this point. For a single stage system, the pulse height is fixed. This means that a very small duty cycle will be required for a small output current. This small duty cycle is difficult to generate accurately. By adding a second stage, the pulse hight is adjustable. This means that the duty cycle for the output stage may be high (80%). It is therefore more practical to generate the duty cycle accurately.

Another problem is the non-ideal properties of the transformer. The duty-cycle of the gate signals and that of the voltage waveform at the input to the L-C filter is slightly different due to the leakage inductance of the transformer (see section 4.7.3 on page 44). This difference is not constant as it depends on the inductor current at that time. It is thus difficult to compensate for this error fully.

The second stage has a much more predictable duty-cycle and is therefore more accurately controllable. The errors caused by the effect described in the previous paragraph can be removed by means of a feed-forward control scheme.

4.4.2 Rectifier

The DC bus is generated by rectifying the 3-phase mains. This may be achieved, in order of complexity, by means of a matrix converter [25], an active rectifier [6], phase-controlled rectifier [8] or passive rectifier (diode-bridge) [6]. A comparison of these options is presented in table 4-I.

Table 4-I: Rectifier comparison

Property	**Simplified matrix converter**	**Active rectifier**	**Phase-controlled rectifier**	**Passive rectifier**
Power flow	Bidirectional	Bidirectional	Unidirectional	Unidirectional
Output voltage range from 400 V 3-phase	0 to 540 V	540 V to 800 V (using 800 V MOSFETs)	0 to 540 V	566 V
External converter?	No	Yes	No	Yes
Synchronised to mains?	Yes	Yes	Yes	No
Switches	12x MOSFET / IGBT	6x MOSFET / IGBT	6x Thyristor	6x Diode
Filter inductor	3-phase HF	3-phase HF	Single LF	None
External soft-start?	No	Yes	No	Yes
Supply current waveform	Sinusoidal	Sinusoidal	Square	Pulsed

If budget and development times were of no consideration, a matrix converter would have been the option of choice. A matrix converter requires complex support circuitry, especially in an isolated converter. Apart from the 12 gate-signals, the gate drivers themselves are complex. It also requires the measurement of supply voltages and currents. These measurements are on the primary side and therefore need an isolated communication standard to bring them to the secondary side for processing.

The same problems exist for an active rectifier, with an added disadvantage being that it requires a separate converter to adjust the bus-voltage. The reason for this is the fact that the resulting bus voltage is higher than the peak line-to-line voltage.

This leaves only the phase-controlled rectifier and passive rectifier, which can be controlled with a relatively simple analogue circuit. The large low-frequency filter inductor

required for the phase-controlled rectifier poses a problem. Since simplicity and physical size is important, the passive rectifier was chosen.

Discreet diodes (1N5408) were used in the construction of the bridge rectifier. The voltage drop according to the datasheet is in the range of 0.6 V to 1.0 V for currents in the range of 1 mA to 10 A. This voltage drop is ignored during the design process presented in this thesis.

4.4.3 Stage 1

As discussed in the previous section, a standard diode rectifier bridge with a bus-capacitor is used to generate a DC bus voltage of 566 V (no-load). The purpose of the first stage is to provide the second stage with a variable bus. Stage 2 then removes voltage fluctuations from this bus by means of a feed-forward strategy. Stage 1 is the less-accurate stage and it thus seems logical to include the isolation transformer there. Various isolated topologies exist, the first of which is the fly-back converter depicted in figure 4.2. For clarity, the embedded diode within the IGBT package is omitted.

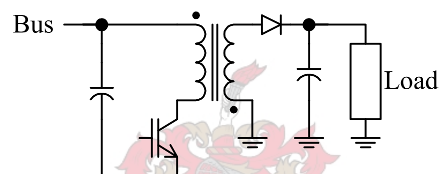


Figure 4.2: Fly-back converter

This topology consists of a single switch, making it attractive as far as simplicity is concerned. It does have many disadvantages, the greatest of which is the transformer. Since it also acts as an inductor, it requires an air-gap. This decreases the flux-linkage between primary and secondary, in turn increasing the leakage-inductance. This is a problem in medium-power converters, such as the one designed in this thesis.

The next option is the push-pull converter, presented in figure 4.3.

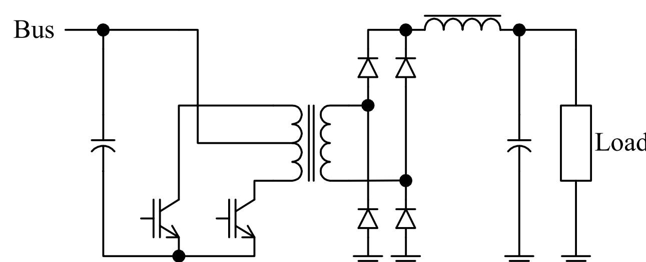


Figure 4.3: Push-pull converter

Both of the switches are driven with respect to ground, thereby making it an attractive option. It also uses the entire B-H curve of the transformer core, thus allowing a smaller transformer to be used.

The high bus-voltage poses a problem. When the one switch is on, the other has to withstand double the bus-voltage (1 132 V in this case). This is not possible for practical MOSFETs. Also, the topology has problems with core saturation and requires extra circuitry to prevent a DC current from flowing.

The two topologies to be considered in more depth are thus the half-bridge and full-bridge. Simplified schematics are shown in figures 4.4 and 4.5 and a comparison is given in table 4-II.

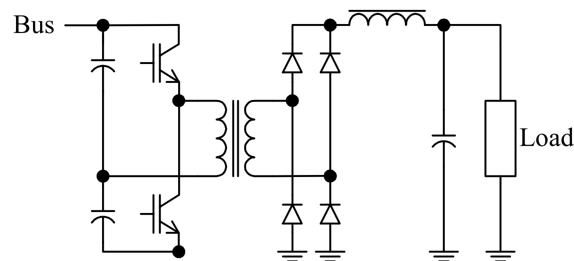


Figure 4.4: Isolated half-bridge converter

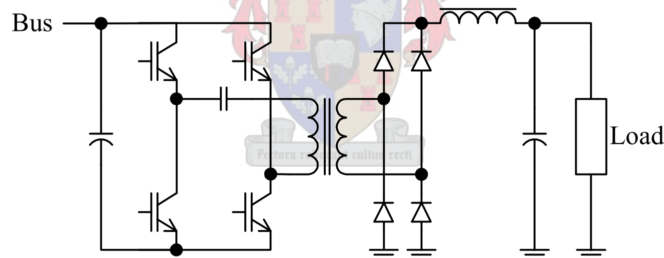


Figure 4.5: Isolated full-bridge converter

Table 4-II: Half-bridge vs full-bridge

Property	**Half-bridge**	**Full-bridge**
Number of switches	2	4
Capacitor	Bus capacitor with centre-tap	Low-voltage bipolar
Leakage-inductance problem	Oscillations	Inaccurate duty-cycle
Switched current	Twice that of full-bridge	Half that of half-bridge
Transformer primary voltages	283 V	566 V

The full-bridge has many advantages over the half-bridge and was chosen. The greatest advantages of the full-bridge are the ability to enforce a zero state as well as the fact that it switches half the current (compared to the half-bridge). This means fewer problems with leakage inductance and EMI.

The transformer must have double the turns, implying up to four times the leakage inductance for the same configuration. This does not pose a problem, as the voltage is doubled and the current halved, which cancels out the effect of the higher inductance – the discharge time is the same.

Since the switching times of the two switches are never exactly the same, it is possible for a DC voltage component to exist across the transformer. The effect is that a DC current grows in the primary of the transformer, which eventually saturates the core. A capacitor in series with the transformer solves this problem. It is easy to show that the peak-to-peak voltage ripple across this capacitor is given by equation 4.3, where D is the duty-cycle, f the switching frequency, C the capacitance and i the current through the capacitor while the switches are cross-conducting.

$$\Delta v = \frac{Di}{Cf} \tag{4.3}$$

For a capacitance of 100 μF, maximum current of 12 A and a switching frequency of 47.684 kHz, the voltage ripple is 2.5 V. This is for a duty-cycle of unity, which is the worst case scenario. It is assumed that the maximum DC voltage is given by the uncertainty of switching times, i.e. $\pm$250 ps due to the gate-resistances and 200 ns due to the MOSFET driver. The 200 ns translates to a duty-cycle uncertainty of 1%, which implies a DC offset of 5.66 V for a 566 V bus. A 16 V capacitor would thus be sufficient. Ten 16 V, 10 μF ceramic capacitors (1206 package) were connected in parallel to form the required 100 μF series capacitor. The transient voltage fluctuations are kept at a minimum by means of a soft start mechanism and low closed-loop bandwidth.

4.4.4 Stage 2

There are various topologies available that offer a directly proportional relationship between cycle-average voltage and duty-cycle. This proportional relationship is very convenient for the implementation of feed-forward. Two topologies which satisfy this requirement are the buck-converter and the full-bridge. Since a unidirectional output current (and voltage) is required, the buck-converter topology was chosen.

One detail to be considered is the presence or absence of a bottom switch. In other words: is a normal buck converter or a synchronous buck (half-bridge) converter used? The bottom switch offers numerous advantages. These include a guarantee of continuous inductor current, bi-directional power flow and lower conduction-losses.

The bottom switch may be replaced, however, by a diode. Silicon-carbide diodes have the advantage of zero reverse-recovery and therefore less switching losses. It is possible that the filter inductor carries negative or zero current during transients. If that be the case, the

flying capacitor of the MOSFET driver discharges and cannot drive the top switch. This is an undesirable scenario and for this reason the bottom switch is included. The circuit is presented in figure 4.6.

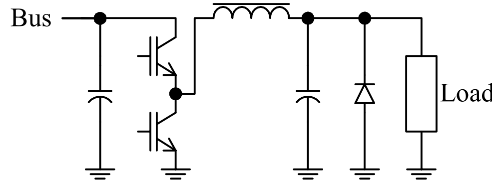
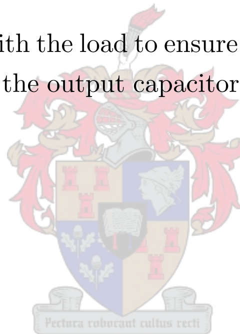


Figure 4.6: Synchronous buck converter

A problem with the circuit in figure 4.6 (at a switching frequency of 381.470 kHz) is that the required dead time becomes a significant portion of the switching period. This results in a restriction of the maximum duty-cycle. An alternative solution is to include a separate isolated power supply for the top switch gate-drive circuit and leaving the bottom switch as a diode. This would ensure that the top switch can be switched at all times, regardless of the inductor current direction.

A diode is included in parallel with the load to ensure a discharge path when the converter switches off. This is to ensure that the output capacitor, being electrolytic, is never reverse-charged.



4.5 Rectifier Detail

4.5.1 Bus Capacitor

Ideally no capacitor is needed, but the supply-lines are inductive. There is thus a parasitic L-C filter present in any practical implementation. The inductance can be very large (in the order of 100 μH). The capacitance is usually very small (in the order of 1 nF). The current waveform at the input of the first stage, as described in section 4.6.3, consists of pulses up to 12 A in magnitude at a frequency of 47.684 kHz. The maximum possible pulse width is 21.0 μs. The best way to prevent bus-voltage disturbances is to put a relatively large bus-capacitor as close to the switching elements as possible. This increases the above-mentioned 1 nF capacitor to a more acceptable value.

At 47.684 kHz the inductance can be considered an open circuit and will therefore not influence the calculation of the bus-capacitor. It was decided that the voltage-drop during any one switching cycle must be less than 1 V. By using equation 4.4 [6], calculations show that the bus capacitance needs to be larger than 140 μF. In the equation, C is the bus capacitor value, i the average supply current, f_s the switching frequency and ΔV the magnitude of the bus voltage ripple.

$$C \approx \frac{i}{f_s \Delta V} \tag{4.4}$$

The capacitor must be able to withstand a maximum of 622 V (see section 4.5.3 on page 32). Two 400 V, 470 μF capacitors in series would thus be sufficient. Large parallel resistances (390 kΩ) are included to ensure even voltage-distribution. Since the diode rectifier does not allow current to flow back into mains, no damping is required.

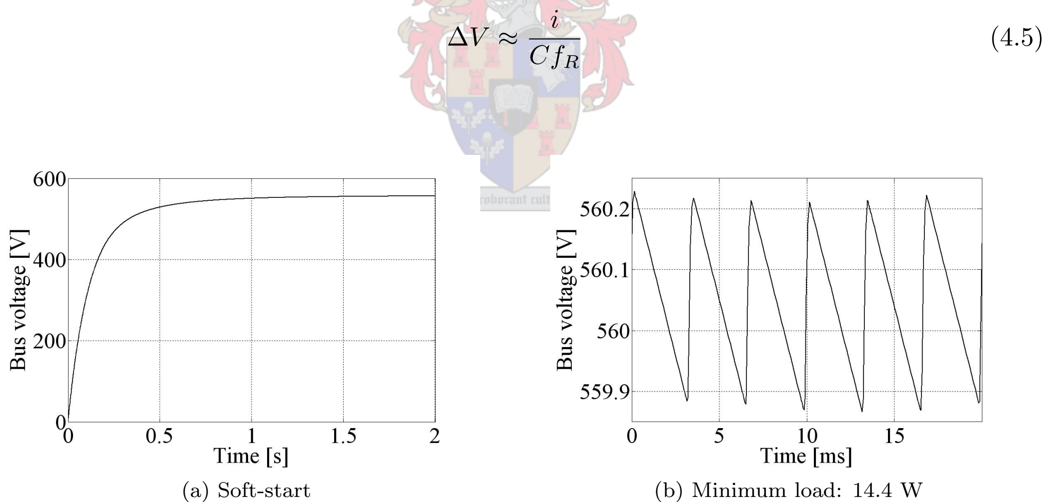
4.5.2 Waveforms

Figures 4.7a and 4.8 show the P-SPICE simulated bus voltage under different conditions. The 400 V 3-phase mains was rectified using 1N5408 diodes and filtered by a 235 μF capacitor. A current source was used to emulate the load. The current was calculated by dividing the power by 566 V.

In figure 4.7a the capacitor is being charged through a 500 Ω resistor under no load. From the resulting curve it is clear that the soft-start process requires 1.5 s before the resistor can be shorted out. More about this in section 5.8.1 on page 82.

Figure 4.7b shows the waveform under the minimum load of 14.4 W. The average current that is drawn from the DC bus is 25.4 mA. The voltage drop may be approximated using equation 4.5 [23], where f_R is the rectified waveform frequency of 300 Hz. Using equation 4.5, the expected voltage drop is 361 mv, which is confirmed by figure 4.7b

$$\Delta V \approx \frac{i}{C f_R} \tag{4.5}$$



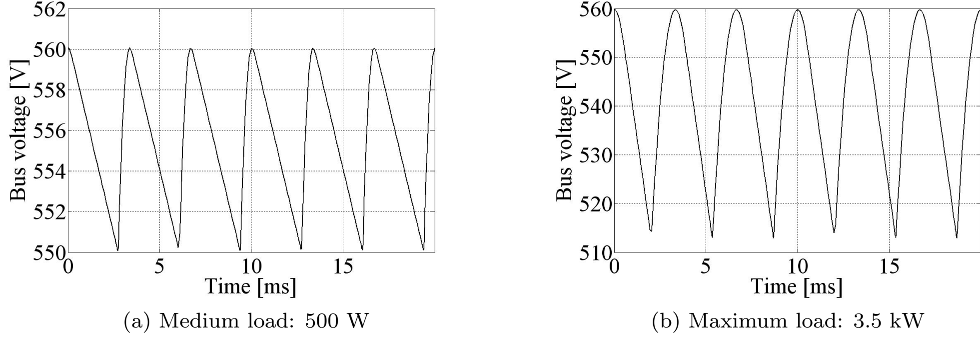
(a) Soft-start

(b) Minimum load: 14.4 W

Figure 4.7: DC bus waveforms

Figure 4.8 shows the results when repeating the process for a medium load (500 W) and the maximum load (3.5 kW). The average currents are 883 mA and 6.18 A respectively. The expected voltage drops are 12.53 V and 87.71 V respectively. These calculated values differ from the simulated waveforms because equation 4.5 does not take the crest of the rectified sinusoidal wave into account, but rather assumes an immediate drop upon reaching

the peak. The equation predicts a larger drop than in reality and is therefore safe to use in the design.



(a) Medium load: 500 W (b) Maximum load: 3.5 kW

Figure 4.8: DC bus waveforms

4.5.3 Line-Frequency Ripple

At full-load, the bus capacitor is not large enough to smooth the rectified waveform. This is shown by equation 4.6. The left hand side of equation 4.6 was adapted from [6] and the right hand side from equation 4.5. The worst-case scenario is therefore where the converter is supplied with rectified 3-phase. The maximum point is calculated using equation 4.7 [6] as 566V. The minimum is given by equation 4.8 [6] as 490 V. Providing for a 10% variation in the supply voltage, the minimum may drop to 441 V and the maximum rise to 622 V.

$$(1 - \cos(30^\circ)) \cdot 566\ V < \frac{6.5\ \text{A}}{235\ \mu\text{F} \cdot 300\ \text{Hz}} \tag{4.6}$$

$$V_{R,peak} = V_{L-L}\sqrt{2} \tag{4.7}$$

$$V_{R,min} = V_{L-L}\sqrt{2} \cdot \cos(30^\circ) \tag{4.8}$$

The waveform was shown in figure 4.8b. The Fourier-series expansion of the waveform is provided by equations 4.9 and 4.10 where f_L is the line frequency (50 Hz) and V_{L-L} the line-to-line voltage (400 V). The derivation of this is provided in appendix B.

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(12 f_L n \pi t) \tag{4.9}$$

$$a_n = \frac{6V_{L-L}\sqrt{2}}{\pi} \left(\frac{(-1)^n(-4)}{(12n-2)(12n+2)} \right) \tag{4.10}$$

The first 10 harmonics, referenced to the average of 540 V, are given in table 4-III.

Table 4-III: Bus harmonic content

Harmonic	**Frequency [Hz]**	**Amplitude [dB]**
1	300	-25
2	600	-37
3	900	-44
4	1200	-49
5	1500	-53
6	1800	-56
7	2100	-59
8	2400	-61
9	2700	-63
10	3000	-65

4.5.4 Ripple Current Through Bus Capacitor

According to the datasheet of the 470 μF, 400 V capacitors, the maximum rms current ripple is 8 A at 120 Hz and 80 °C. The equivalent series resistance is 100 Ω at 120 Hz. According to PSpice simulation, the rms current ripple at full load (3 500 W) is 7.2 A. The same conditions as in section 4.5.2 were used. It can therefore be concluded that the above arrangement is practically viable.

4.5.5 Losses

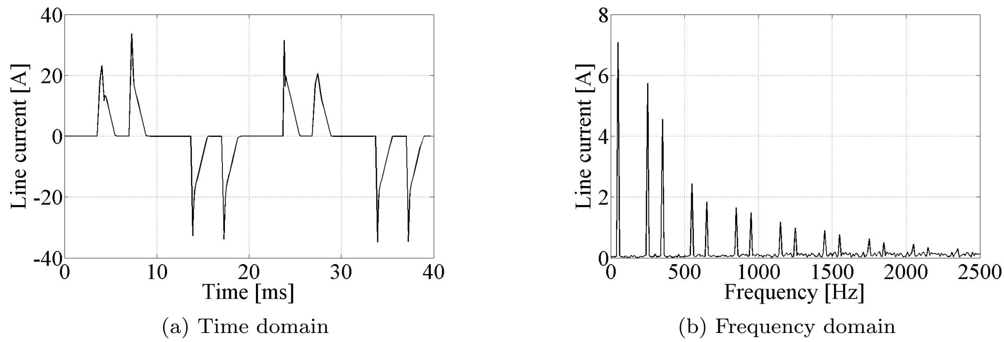
There is very little loss in the rectifier circuit. The larger power loss is through the soft-start resistor bypass MOSFET. Its RMS current is 6 A and passes through a 99 mΩ resistor. This implies a loss of 3.5 W.

It can be assumed that the average current through each of the diodes in the rectifier is 2 A. The voltage drop at that current across a 1N5408 (the chosen diode for constructing the rectifier) is less than 1 V. This means a loss of 2 W through each diode. According to the datasheet, the ambient temperature must be below 75 °C for an average current of 2.5 A per diode.

4.5.6 Supply Current

The supply current resulting from a passive rectifier is very far from sinusoidal. Figure 4.9 shows the PSpice simulation of the line current at full load, together with a frequency spectrum calculated using an FFT algorithm. The simulation uses a 1N5408 diode bridge rectifier, 235 μF bus capacitor and a 90 Ω load resistor.

A line-frequency filter, such as a standard EMI filter, must be inserted before the rectifier in order to lower the total harmonic distortion of the line current. Alternatively, future implementations of the power supply may use an active rectifier, which has sinusoidal current. The prototype has no filter.



(a) Time domain (b) Frequency domain

Figure 4.9: Supply current waveform

4.6 Stage 1 Detail

4.6.1 Switches

The high frequency switches can be implemented using either MOSFETs or IGBTs. CoolMOS MOSFETS was chosen for their superior switching times (see tables 4-V and 4-VI on page 41). Figure 4.10 shows the circuit for the first stage, along with all the voltages and currents referenced further in this thesis.

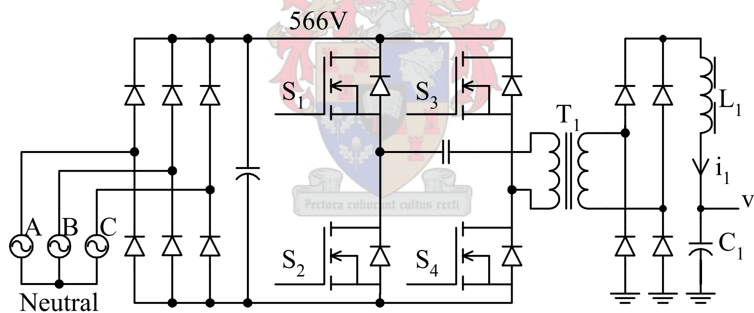


Figure 4.10: Converter Topology: Stage 1

4.6.2 Switching

The idea is to transfer a variable average voltage across the HF transformer. The transformer core, however, must not saturate, and therefore the average voltage across the transformer must be zero. This contradiction may be solved by using a sequence of alternate positive and negative pulses [6]. These pulses are rectified and filtered at the secondary side. The widths of the pulses determine the average voltage. In order to enforce this positive-to-zero-to-negative-to-zero pulse sequence, the switches are switched according to table 4-IV.

Table 4-IV: Switching sequence

State	S1	S2	S3	S4	Condition
A	1	0	0	1	Cross-conducting
B	1	0	1	0	Zero
C	0	1	1	0	Cross-conducting
D	0	1	0	1	Zero

4.6.3 Waveforms

With reference to figure 4.11, the waveforms of an isolated full-bridge converter are provided in figures 4.12 to 4.13. These graphs were generated for a bus voltage of 540 V, a 1:1 transformer, 3 mH inductor, 470 μF capacitor, 12 A load and a duty-cycle of 50%. The inductor has a 100 mΩ resistor in series with it and the capacitor a 10 mΩ. All the other components are ideal. A switching frequency of 47.864 kHz was used.

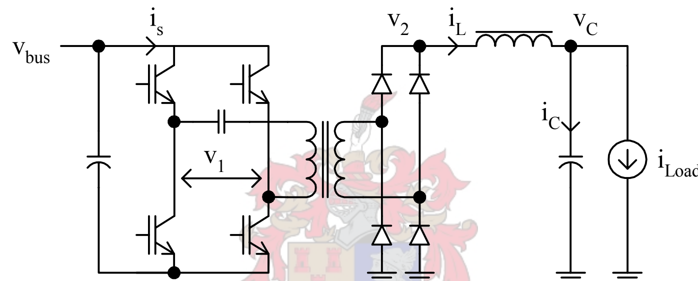
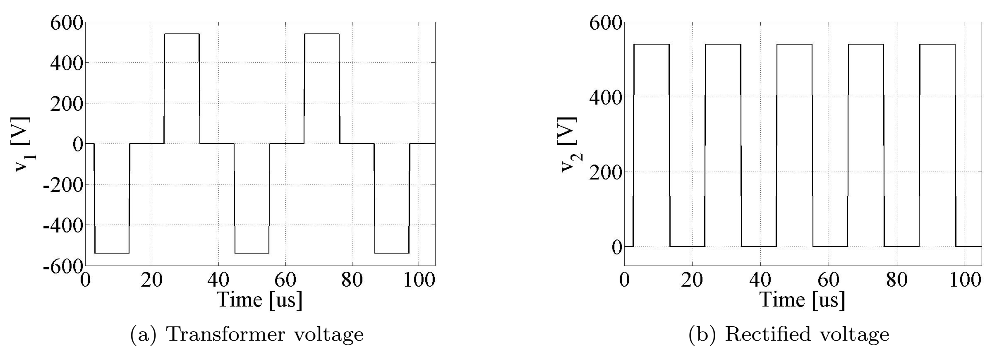


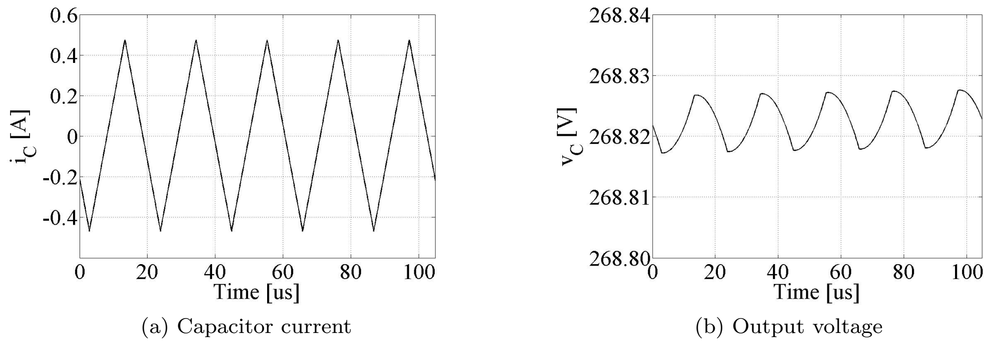
Figure 4.11: Full-bridge waveforms: circuit



(a) Transformer voltage

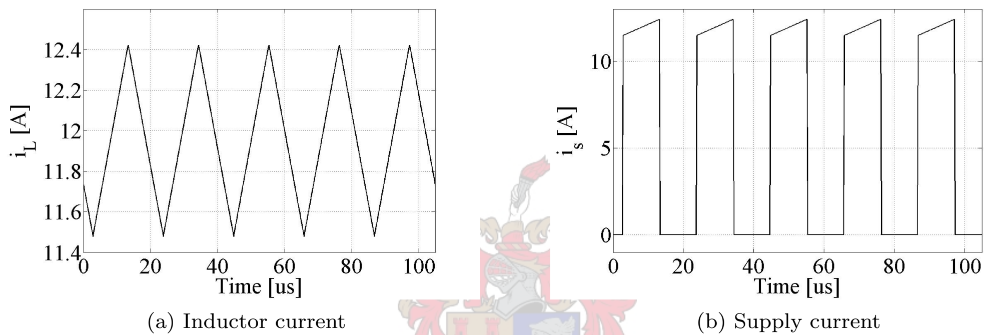
(b) Rectified voltage

Figure 4.12: Full-bridge waveforms



(a) Capacitor current (b) Output voltage

Figure 4.13: Full-bridge waveforms



(a) Inductor current (b) Supply current

Figure 4.14: Full-bridge waveforms

Figure 4.12a shows the voltage across the transformer (v_1). The first pulse is negative, which corresponds to S_2 and S_3 being closed. Then S_3 opens. The current stored in the leakage inductance of the transformer finds a path through the diode of S_4, making v_1 zero. The current in now 'stored' in the leakage inductance because of this zero voltage drop. In a practical system the on voltage of the diode in combination with the on resistance of the MOSFET will generate a voltage over the transformer, but this is small enough to be ignored at this point. Switch S_4 therefore switches on when the voltage across it is already zero, resulting in no switching loss.

The next part of the cycle is where S_2 opens, letting the current find a path through the diode in S_1. This corresponds to the second pulse. The leakage inductance now rapidly discharges, but before it can fully discharge, S_1 closes. This is also a zero voltage switching condition and therefore results in no switching loss. The secondary of the transformer is still zero at this point. This is because the current is in the 'wrong' direction. The diode bridge on the secondary side prevents the voltage from rising until the current has turned around. This is a very short and has little effect.

The process continues with S_4 closing, letting the current go through the diode in S_3,

zeroing the voltage again. The current is once again trapped and S_3 closes while there is zero voltage across it. Finally S_1 opens, S_2 closes and the process repeats itself.

Besides from the little delay in rising-edge, the secondary waveform of the transformer is identical to the primary. This is then rectified, producing figure 4.12b.

It may be assumed that the output voltage v_C is constant for all practical purposes, as can be seen in figure 4.13b. During a 'high' pulse or a 'low' pulse the voltage across the inductor is also constant. This means that the current in the inductor must rise and fall linearly, as can be seen in figure 4.14a. The average current must be the same as the output current (11.95 A in this case) in order for the system to be in steady state.

Figure 4.14b shows the supply current. This is the same as the rising sections of the inductor current. By tracing the path taken by the current through the switches this can easily be explained. While in a zero state, the current simply circulates through either the top two or bottom two switches. While in a non-zero state, the current flows from the bus, through the transformer and back through the bus capacitor. These are the pulses observed in figure 4.14b

Since the current though the load is nearly constant, the ripple component of the inductor must flow though the capacitor. This can be seen in figure 4.13a. The capacitor integrates this waveform to generate the voltage waveform observed in figure 4.13b.

4.6.4 Transformer Winding Ratio

In section 4.5.3, the bus voltage was calculated to be in the range of 441 V to 622 V. The range was specified in section 4.3.4 as 250 V.

The second stage is designed to have a duty-cycle of 80% (see chapter 5), making the maximum voltage after the filter of stage 1 equal to 312.5 V. The practical maximum duty-cycle of stage 1 is only about 80% due to the leakage inductance of the transformer. This implies that the minimum voltage at the secondary of the transformer must be larger than 391 V. Using the minimum bus voltage of 441 V as a design criterion, the winding ratio should be 441:391. Symplifying yields a winding ratio of 45:40. This means that the primary-side range of 441 V to 622 V transforms to a secondary-side range of 392 V to 553 V. This is high enough to meet the requirement of 391 V and low enough to be acceptable for the diodes' breakdown voltage of 600 V.

4.6.5 High Frequency Rectifier

Ultra-fast diodes are required to rectify the 23.842 kHz signal from the transformer. New technology Silicon-Carbide Schottky diodes were chosen for this purpose. The voltage drop during conduction is larger than conventional Schottky diodes, but they offer zero reverse-recovery. This is an advantage with fast switching frequencies, as the switching loss becomes zero. Also, they have a negative temperature coefficient. This means that the hotter the diode, the smaller the current for any given voltage-drop. It is thus easy to put these devices in parallel – no external resistances are required.

There are two configurations to choose from: half-bridge and full-bridge. These are presented in figures 4.15 and 4.16.

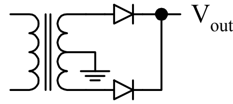


Figure 4.15: Half-bridge rectifier

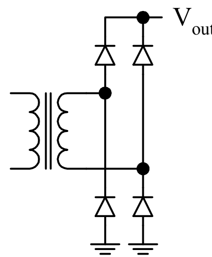


Figure 4.16: Full-bridge rectifier

The half-bridge has half the number of diodes, but requires double the number of windings on the transformer. It also requires that the diodes can withstand double the reverse-voltage. The chosen diodes can only withstand 600 V, therefore a full-bridge was used.

Pectora roborant cultus recti

4.6.6 Switching Frequency

For the switching noise to be outside human audible range, the frequency through the transformer must be higher than 20 kHz [24]. The first stage must therefore have a switching frequency higher than 40 kHz. A switching frequency of 47.684 kHz was chosen. The reason for this specific value is the 10 ps resolution of the delay-lines (see section 6.2.3 on page 92).

4.6.7 Filter

A standard L-C filter is used to remove the switching-frequency component and leave only the intended output voltage. First it is assumed that the capacitor is infinitely large and that the output voltage is perfectly constant. This is a valid assumption, as the system is designed to have a negligible voltage ripple. The current-ripple can thus be calculated using equation 4.11.

$$\Delta i_L = \frac{(V_{bus} - v_{out})\, D}{f_s L} \tag{4.11}$$

It is also known that, at steady state, D and v_{out} are given by equations 4.12 and 4.13 respectively.

$$D = \frac{v_{out}}{V_{bus}} \tag{4.12}$$

$$v_{out} = 22.5 i_{out} \tag{4.13}$$

Substituting and solving for L yields equation 4.14.

$$L = \frac{22.5 i_{out} \left(V_{bus} - 22.5 i_{out} \right)}{V_{bus} f_s \Delta i_L} \tag{4.14}$$

The current ripple is required to be smaller than twice the output current in order to ensure continuous inductor current. It is also required, as a design criterion, to be smaller than 1 A. According to equation 4.14, an inductor of 113 μH would satisfy the first constraint. For the second criterion, the maximum inductance value would be at an output current given by equation 4.15. This implies an inductance of 3 mH.

$$i_{out} = \frac{V_{bus}}{2 \cdot 22.5} \tag{4.15}$$

Due to the space restriction as well as a maximum DC current of 12 A, a laminated-steel core was chosen. This type of core is still strongly magnetic at a flux density of 1 T. Ferrite cores can only be designed up to a flux density of 200 mT before they saturate. The only disadvantage is larger core-losses, but since the current-ripple is small, this is not a problem.

Using the equations derived in appendix B.2 on page 167, the inductor was designed with an E-core of cross-sectional area 1848 mm^2, an air-gap of 1.5 mm (3 mm total) and 61 turns.

The next item is the capacitor. Two aspects must be considered. First the current ripple through the inductor and second the current pulses drawn by the second stage converter. For ripple calculations, the load will be considered to be an inductor of 1 H (worst-case) in series with a 14.4 Ω resistor. This implies a -110 dB transfer from voltage to current at 47.684 kHz. The 80% duty-cycle of stage 2 subtracts another 2 dB. A ripple rejection ratio of 106 dB is required (5 ppm) at the switching frequency, therefore the ripple must not be larger than the output voltage, which is always true.

In keeping the validity of the assumption that the voltage remains constant, however, a 1% voltage ripple would be considered acceptable. Using equation 4.16, the capacitance must be larger than 1 μF to sufficiently absorb the current-ripple. Using equation 4.17, the capacitor must be larger than 12 μF in order to smooth the current pulses of the second stage (f_2 is 381.470 kHz).

$$C = \frac{\Delta i}{8 f_s \Delta v} \tag{4.16}$$

$$C = \frac{0.8 i_{out}}{f_2 \Delta v} \tag{4.17}$$

The control system and over-voltage protection will ensure that the maximum voltage across the capacitor is 375 V. The same capacitors used for the bus (400 V, 470 μF) will be large enough, although a 400 V, 22 μF will also be sufficient. As later discussed in chapter 5, a larger capacitor better rejects the bus disturbance. For this reason the choice of a 470 μF capacitor is retained.

4.6.8 Losses

There are two types of losses to be considered in the switching elements. These are switching losses and conduction losses.

During the switching off transient of any of the four full-bridge MOSFETs, the diode can only take over the current once it becomes forward-biased. The MOSFET voltage must thus increase to full bus-voltage while carrying the full current at the same time. During the switching on transient, the diode of the MOSFET to be switched on carries the current. The voltage over the switch is therefore already zero. Negligible switching losses occur during switching on transients.

CoolMOS MOSFETs have very fast switching times. Those used for stage 1 are SPW17N80C3 from Infineon. Their rise- and fall times are 15 ns and 6 ns respectively. The waveform can be approximated as triangular [6], and a half-base-times-height equation may be used. The switching losses are given by equation 4.18 where I_s is the switched current.

$$P_s = \frac{1}{4} \left(t_{rise} + t_{fall} \right) V_{bus} I_s f_s \tag{4.18}$$

The maximum bus voltage may be taken as 566 V and the maximum switched current as 10.67 A. This current was calculated using equation 4.19. The duty-cycle D_2 is 80%, the maximum load current I_{Load} is 15 A and the transformer winding ratio N is 40/45. The switching frequency is 47.684 kHz. The switching losses are therefore no more than 1.5 W per transistor.

$$I_{sw} = I_{Load} D_2 N \tag{4.19}$$

As for conduction losses, the mean-square value of the current through each switch is considered for each of the four switching states. The MOSFETs in question have an 'on' resistance of 0.29 Ω.

Due to the leakage inductance of the transformer, the current waveform through each switch may be very closely approximated as a square wave with 50% duty-cycle. The magnitude of this square wave is equal to the switched current (10.67 A). The maximum rms current through each of the four transistors is therefore 5.33 A, resulting in conduction losses of 8.249 W per MOSFET.

The total losses due to the MOSFETs of the first stage is 9.749 W per transistor.

The average current through the high frequency rectifier has a maximum of 6 A per diode (half the maximum stage 1 output current of 12 A. The voltage across the diode is given in the datasheet as being 1.6 V. By using equation 4.20, the losses in the diodes are 9.6 W per diode. There are no switching losses in silicon-carbide diodes.

$$P_D = \overline{I_D V_D} \tag{4.20}$$

4.6.9 Dead Time

When switching between top and bottom transistors, and the two gate-signal edges occur at the same time, it might take longer for the one to switch off than for the other to switch on. This results in a state of simultaneous conduction.

There are various times involved when calculating switching times. First the gate must be charged (or discharged) through a resistor past the threshold voltage. The resistor is to prevent the gate-driver from overheating. Second, the switching process takes a specific time interval. Third, the gate-driver has delays and rise- and fall times. These delays are listed in tables 4-V and 4-VI. An IR2184 gate-driver, 8.2 Ω gate resistance and SPW17N80C3 MOSFET are used. The gate is charged to 12 V.

Table 4-V: Stage 1 MOSFET on times

Cause	**Delay [ns]**
Driver delay	680
Driver rise time	40
MOSFET delay due to resistor	35
MOSFET rise time	15
Total	770

Table 4-VI: Stage 1 MOSFET off times

Cause	**Delay [ns]**
Driver delay	270
Driver fall time	20
MOSFET delay due to resistor	145
MOSFET fall time	6
Total	441

From these tables it may be noted that if the gate driver receives the 'on' and 'off' signals for the two MOSFETs at exactly the same time, the MOSFETs will be off simultaneously

for 329 ns. This driver-MOSFET combination therefore provides an intrinsic dead time of 329 ns.

This means that the output is floating for a short period of time. This has various effects, depending on the topology and application. For the full-bridge topology designed, the effect is negligible. The reason for this is that the current stored in the leakage inductance ensures that the next 'switching state' is reached the moment the relevant switch opens (441 ns after the gate signal edge). The voltage over the closing switch is already zero by the time it closes.

This above argument is valid when the discharge time of the leakage inductance is larger than the dead time. When the discharge time is smaller than the dead time (this is true when switching a small current from a zero state to a cross-conducting state) the reverse-recovery time of the internal diode in the MOSFET is sufficient to recharge the leakage inductance in the opposite direction. Since the switch has not yet closed, the newly stored current finds a path through the original MOSFET diode, making the voltage over the transformer zero once more. The situation therefore changes from zero-voltage switching to low-current switching. The pulse width increases by the dead time as opposed to the leakage inductance discharge time.

4.7 Transformer Detail

4.7.1 Configuration

Configuration of the transformer can play a major role in its performance. There are various options, compared in table 4-VII.

A major concern is the leakage-inductance, due to the high frequency. Another is the physical size of the transformer, which must fit into a 19″ rack together with all the inductors, heat sinks, printed circuit boards and other components.

A twisted-pair transformer can be used only for a winding ratio of 1:1. The desired winding ratio is 45:40, which is very close to 1:1. If the diodes were to be replaced with ones that has a breakdown voltage of 800V, a twisted-pair transformer would be the ideal option. This would be done in the final product. The prototype used a co-axial transformer. A twisted-pair transformer was built for comparison. The design presented further in this chapter is for a twisted-pair transformer.

Ferrite is used so that core losses are minimised at the high frequency concerned. Maximum flux density is designed as 200 mT. Using equation 4.21, the minimum number of turns that would guarantee that the core will not saturate was calculated as 21. This is for a switching frequency (f_s) of 47.684 kHz, a core area (A_{core}) of 1320 mm^2 and a bus voltage (V_{bus}) of 566 V.

$$N = \frac{V_{bus}}{2B_{max}A_{core}f_s} \tag{4.21}$$

It is also required that the magnetising current be limited. A magnetising current of

Table 4-VII: Transformer configuration properties

Property	**Co-axial**	**E-core with twisted-pair windings**	**E-core with layered windings**	**E-Core with disc windings**
Leakage inductance	Very low and predictable	Low, but unpredictable	High and unpredictable	Very high and unpredictable
Magnetising inductance	Low	High	High	High
Physical size	Large	Small	Small	Small
Thermal properties	Good for core, Bad for windings	Good	Good	Good
Capacitance across terminals	Low	High	High	High
Capacitance between primary and secondary	Low	High	Low	Very low
Primary to secondary breakthrough voltage	Medium	low	High	Very high

500 mA was chosen to start with. Using equation 4.22, the number of turns to meet this requirement is 34.

$$N = \sqrt{\frac{V_{bus} l_{core}}{2\mu_r \mu_0 A_{core} f_s i_{m,max}}} \tag{4.22}$$

Due to a restriction of available materials, the final transformer was wound with 30 turns. This implies a maximum magnetising current of 636 mA, which is still acceptable. Both equations are derived in full in Appendix B.3 on page 169.

4.7.2 Non-Ideal Properties

The simplified equivalent circuit of a practical transformer is presented in figure 4.17. Resistances are small enough to be ignored.

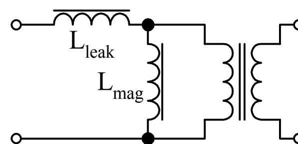


Figure 4.17: Equivalent circuit of transformer

To measure L_{mag} and L_{leak}, an open and short circuit test was done respectively. By

leaving the secondary as an open circuit, the ideal transformer becomes an open circuit. The measurement thus provides the combination of the two inductances. Since L_{mag} is much larger than L_{leak}, the measurement represents L_{mag}. Next, by short circuiting the secondary, the ideal transformer forms a short circuit across L_{mag}, leaving only L_{leak} in the measurement.

In order to perform the measurement, the transformer is driven with a square wave using a 50 Ω signal generator. By substituting the measured half time ($t_{1/2}$) into equation 4.23, L_{mag} and L_{leak} were calculated to be 2.8 mH and 4 μH respectively. In the equation, R is 50 Ω. The co-axial transformer has a leakage inductance of 38 μH.

$$L = \frac{t_{1/2}R}{\ln(2)} \tag{4.23}$$

Another high-frequency consideration is the effect of skin-depth, given by equation 4.24. At the switching frequency of 47.684 kHz, the skin-depth of copper is 0.3 mm. If the 10th harmonic of the square wave is still considered, the skin-depth is 0.1 mm.

$$\delta = \sqrt{\frac{2\rho}{\omega\mu}} \tag{4.24}$$

The ideal would thus be to wind the transformer with strands of 100 μm diameter in order to minimise the effective resistance and thus increase efficiency. The prototype transformer was wound with two strands of 1.2 mm diameter copper wire in order to save time.

4.7.3 Leakage Inductance

When using a full-bridge topology, as in stage 1, the duty-cycle error is only dependant on the output current (a slow-changing property) and not on the specific timing. Equation 4.25 provides the discharge time as a function of switched current (equal to the first stage inductor current). This is the time it takes the transformer secondary to rise after the opening switch has switched off.

$$t = \frac{2iL_{leak}}{V_{bus}} \tag{4.25}$$

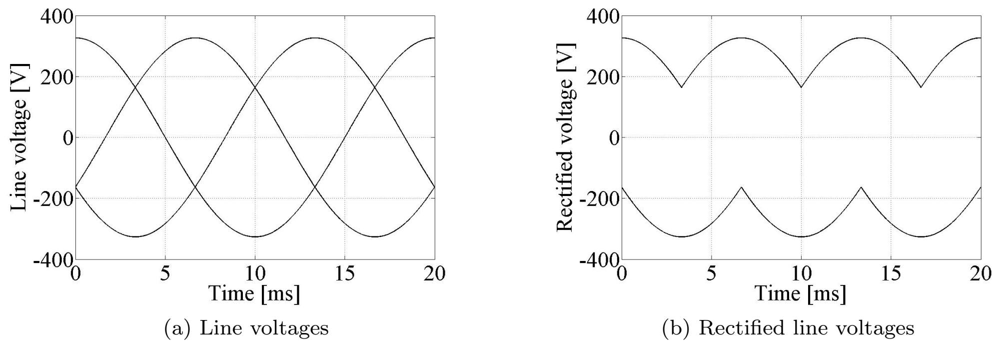
The maximum switched current is 12 A and the bus voltage is 566 V. The twisted pair transformer and the co-axial transformer has maximum discharge times of 170 ns and 1.611 μs respectively. The associated duty-cycle errors are 0.81% and 7.7% respectively. These are quoted as absolute errors, not relative errors.

4.7.4 Isolation

Isolation across the transformer is an important consideration. Standard resin-coated transformer wire has an electrical breakdown voltage of approximately 1 kV and must not be exceeded. Figure 4.18a shows the line voltages relative to neutral (or ground, assuming that neutral and ground are at the same potential). Figure 4.18b shown the output of the

Pectora roborant cultus recti

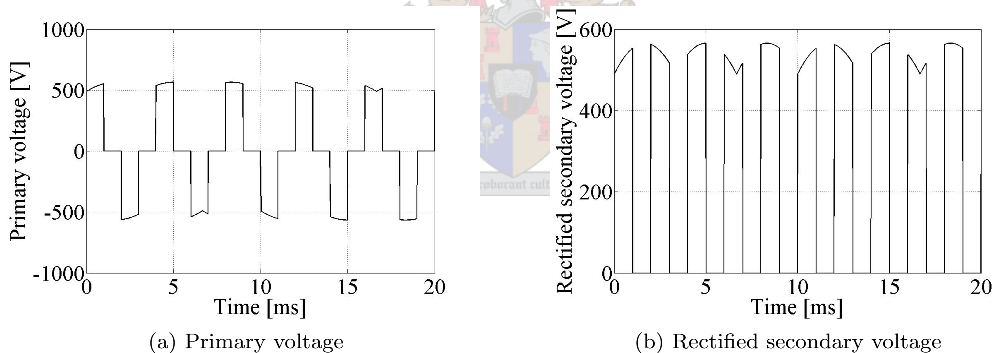
rectifier also referenced to neutral. The bottom curve is the common voltage and the top line the bus voltage.



(a) Line voltages (b) Rectified line voltages

Figure 4.18: Line voltage waveforms referenced to neutral

The full-bridge converter then connects the primary winding to either one of these two voltages. Figure 4.19a shows the resulting voltage across the primary of the transformer. For clarity, a switching frequency of 2 kHz is used. Figure 4.19b shows the rectified secondary voltage. This is shown relative to ground potential.

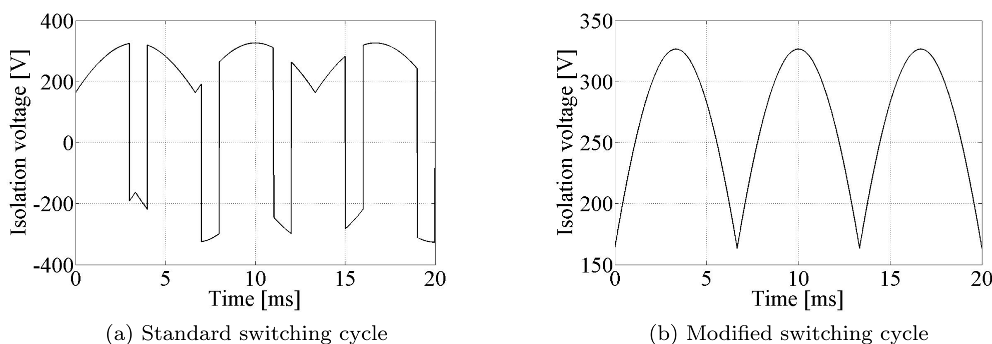


(a) Primary voltage (b) Rectified secondary voltage

Figure 4.19: Transformer voltage waveforms

The isolation voltage (primary to secondary voltage) is shown by figure 4.20. This figure represents the isolation between the two ports of the transformer with the same polarity. For opposite ports, the transformer primary voltage waveform must be added. After addition of this latter signal, the isolation voltage is still under 1 kV, which is acceptable.

Figure 4.20a represents the resulting waveform when using the switching scheme presented in table 4-IV on page 35. Figure 4.20b represents the waveform after modifying the switching scheme to be as presented in table 4-VIII.



(a) Standard switching cycle

(b) Modified switching cycle

Figure 4.20: Isolation voltage waveforms

Table 4-VIII: Switching sequence

State	S1	S2	S3	S4	Condition
A	1	0	0	1	Cross-conducting
B	0	1	0	1	Zero
C	0	1	1	0	Cross-conducting
D	0	1	0	1	Zero

There are many advantages to this scheme. One advantage is the elimination of a high-frequency component over the transformer isolation boundary. Another is that standard half-bridge switching signals may be used, which allows for an easy upgrade from half-bridge to full-bridge by simply changing the MOSFET Drivers. The operation of the circuit is not affected. The only consideration is the fact that the two bottom switches carry current for longer periods than the top switches, as opposed to the balanced situation observed previously.

4.7.5 Thermal Analysis

Both the primary and the secondary used 4.8 m of 2.26 mm^2 copper wire. The resistance of each winding is given by equation 4.26, where R is the resistance, ρ the resistivity, l the length and A the cross-sectional area. The resulting resistance of each winding is 35.64 mΩ.

$$R = \frac{\rho l}{A} \tag{4.26}$$

At an rms current of 6 A, the total loss in both windings is 2.566 W. This is very small considering the relatively large surface area of the transformer. This calculation does not take skin-effect into account.

4.8 Stage 2 Detail

4.8.1 Switches

As for stage 1, CoolMOS MOSFETs were chosen for their superior switching times. Figure 4.21 shows the circuit for the second stage, along with all the voltages and currents referenced further in this thesis.

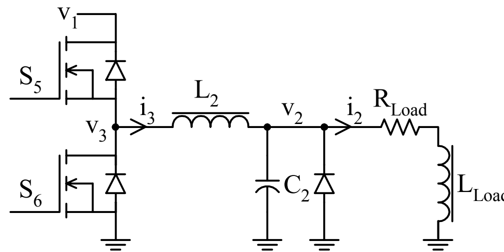
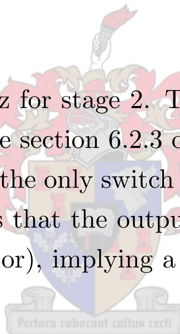


Figure 4.21: Converter Topology: Stage 1

4.8.2 Switching

A switching frequency of 381.470 kHz for stage 2. The reason for this specific value is the 10 ps resolution of the delay-lines (see section 6.2.3 on page 92).



The second stage is very simple – the only switch present is driven directly by the PWM signal. The MOSFET driver requires that the output voltage of the converter is grounded regularly (to charge its flying capacitor), implying a maximum duty-cycle. This maximum duty-cycle has been chosen as 95%.

4.8.3 Waveforms

A buck converter ‘chops’ the bus voltage into a pulse-train, which is then filtered to provide a voltage lower than the bus. The width of each pulse in the pulse-train determines the average output voltage (provided the pulse-frequency is constant). With reference to figure 4.22, figures 4.23 to 4.25 provide the waveforms for a buck converter. These graphs were generated for a bus voltage of 90 V, 200 μH inductor, 470 μF capacitor, 5 A load and a duty-cycle of 80%. The inductor has a 100 mΩ resistor in series with it and the capacitor a 10 mΩ. All the other components are ideal. A switching period of 2.621 550 μs (381.470 kHz) was used.

As expected, the switched voltage v_1 is equal to the bus voltage (90 V) for 2.096 712 μs and zero for the remaining 524.288 ns. This shows the duty-cycle of 80%.

Figure 4.24a shows how the inductor is charged (and discharged) by a constant voltage each switching state. When the top switch is on the inductor charges linearly and when the bottom switch conducts the inductor discharges linearly. The supply current is shown in

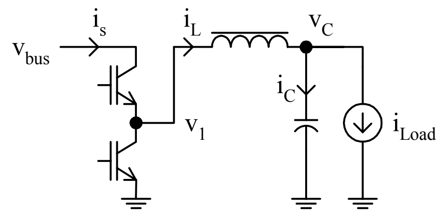


Figure 4.22: Buck converter waveforms: circuit

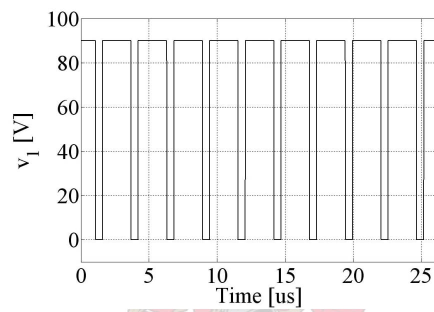
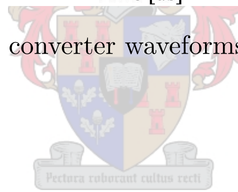
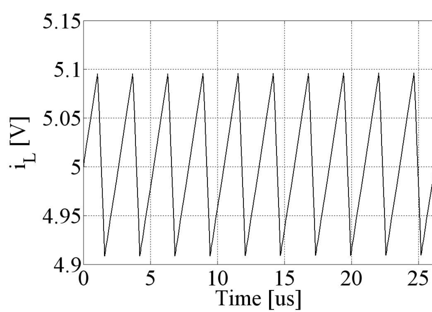
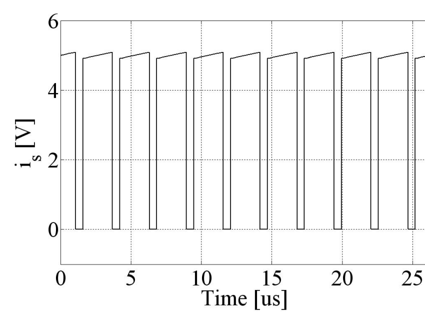


Figure 4.23: Buck converter waveforms: Switched voltage


Pectora roborant cultus recti

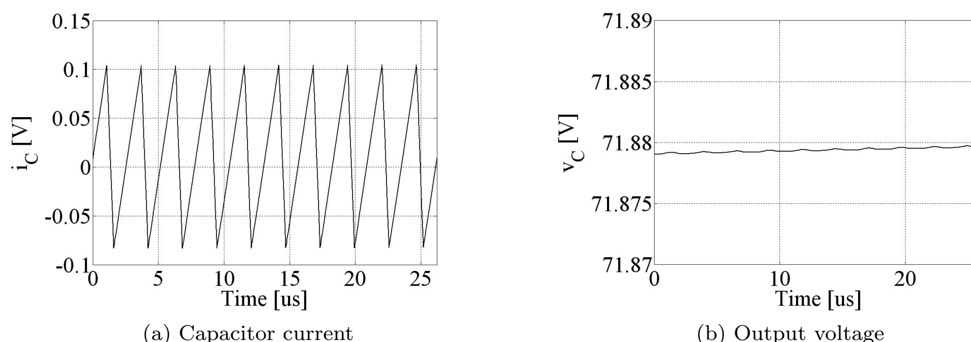


(a) Inductor current



(b) Supply current

Figure 4.24: Buck converter waveforms



(a) Capacitor current

(b) Output voltage

Figure 4.25: Buck converter waveforms

figure 4.24b. When the top switch conducts, the inductor current is drawn from the bus. No current is drawn from the bus when the bottom switch is closed.

The current that does not flow through the constant current load flows through the capacitor. The capacitor current is therefore the same as the inductor current, but with zero average. This is shown in figure 4.25a. The voltage over the capacitor is the integral of the capacitor current. The switching frequency is very high and the filter has been designed for minimal output voltage ripple. The voltage waveform shown in figure 4.25b is therefore approximately a straight line.

4.8.4 Switching Frequency

In order for the voltage ripple across the load to be as small as possible with a relatively small L-C filter, the switching frequency must be as high as possible. By using a 10 ps delay-line to generate 18-bit PWM (see section 6.2.3 on page 92), the maximum switching frequency is 381.470 kHz.

The rise and fall times of the MOSFETs are 5 ns each, which is small enough (<1% of the switching period). It is therefore possible to implement such a high switching frequency.

4.8.5 Filter

As with the first stage, an L-C filter is used to remove switching ripple. It requires 110 dB attenuation at the switching frequency of 381.470 kHz (to assist the control system). An L-C filter has a rejection of 40 dB per decade. Using equation 4.27 yields an L-C constant of $55 \cdot 10^{-9}$ s^2.

$$LC = \frac{1}{\left(2\pi\left(\frac{f_s}{10^{110/40}}\right)\right)^2} \tag{4.27}$$

First considering the inductor, the relevant equations are given by equations 4.28 to 4.31 where R is the load resistance of 14.4 Ω.

$$L = \frac{(v_{bus} - v_{out})\, D}{\Delta i f_s} \tag{4.28}$$

$$v_{bus} = \frac{v_{out}}{D} \tag{4.29}$$

$$D = 0.8 \tag{4.30}$$

$$L = \frac{0.2 R i_{out}}{\Delta i f_s} \tag{4.31}$$

The inductor must thus be 113 μH for a ripple current of smaller than 1 A. Furthermore, with a 470 μF capacitor, the inductor needs to be 117 μH to adequately remove the switching voltage ripple. A 200 μH inductor was chosen.

Using the same design-procedure as in stage 1 (section 4.6.7), 21 turns were wound round a 1056 mm^2 laminated-steel E-core. The air-gap was chosen to be 1.5 mm (3 mm total).

4.8.6 Losses

During steady state conditions, the inductor current is always positive. This implies that the top MOSFET operates under hard-switching conditions for both the 'on' and 'off' transients. This also means that the bottom MOSFET operates under zero-voltage switching for both transients. The switching loss for the bottom MOSFET is thus zero, but for the top one is given by equation 4.32.

$$P_s = (t_{rise} + t_{fall})\, v_1 i_3 f_s \tag{4.32}$$

In equation 4.32, P_s is the switching losses and t_{rise} and t_{fall} the rise and fall times of the MOSFET respectively. The switching frequency f_s is 381.470 kHz. The inductor current i_3 has a maximum value of 15 A. The input voltage v_1 has a maximum steady-state value of 270 V ($14.4\ \Omega \cdot 15\ A \cdot 1.25$).

Different MOSFETs (IPW60R099CP, Infineon) are utilised for stage 2. These MOSFETs have an 'on' resistance of 99 mΩ and rise and fall times of 5 ns each. The resulting switching loss for the top MOSFET is therefore 15.45 W.

Conduction losses are taken at a duty-cycle of 80%. The maximum rms current through the top MOSFET can thus be approximated as 12 A. Similarly the maximum rms current through the bottom MOSFET can be approximated as 3 A. This is valid for a very small inductor current ripple. The conduction losses are thus 14.256 W for the top transistor and 891 mW for the bottom one. Total losses are 29.706 W for the top transistor and 891 mW for the bottom one.

4.8.7 Dead Time

As discussed in section 4.6.9, there has to be a time between switching the one transistor off and the other one on. This time is called the dead time.

The MOSFETs used in the second stage converter are faster than those used in the first stage. A similar calculation can be performed as in tables 4-V and 4-VI on page 41. The delays applicable for stage 2 are listed in tables 4-IX and 4-X. An IR2181 gate-driver, 8.2 Ω gate resistance and IPW60R099CP MOSFET are used. The gate is charged to 12 V.

Table 4-IX: Stage 2 MOSFET on times

Cause	**Delay [ns]**
Driver delay	180
Driver rise time	40
MOSFET delay due to resistor	20
MOSFET rise time	5
Total	245

Table 4-X: Stage 2 MOSFET off times

Cause	**Delay [ns]**
Driver delay	220
Driver fall time	20
MOSFET delay due to resistor	172
MOSFET fall time	5
Total	417

Pectora roborant cultus recti

The externally applied dead time must therefore be more than 172 ns, which corresponds to a duty-cycle of 6.56% at a switching frequency of 381.470 kHz. The signal for the top switch remains unadjusted. The signal for the bottom switch rises late and falls early. This is so that the top switch duty cycle is exactly as it would have been for a standard buck converter (with a diode in the place of the bottom switch).

4.9 Heatsink

4.9.1 Contribution of Stage 1 MOSFETS

In section 4.6.8 the total losses have been calculated as 9.749 W per transistor. The total is therefore 39 W for the four MOSFETS.

According to the datasheet, the junction to case thermal resistance θ_{JC} of the MOSFETs is given as 0.6 K·W-1. By using eqation 4.33 [6], the maximum case temperature T_C can be calculated.

$$T_C = T_J - P\theta_{JC} \tag{4.33}$$

The maximum junction temperature T_J is given by the datasheet as 150 °C. This implies a maximum case temperature of 140 °C.

The thermal resistance of the pads used to isolate the heatsink from the MOSFETs have a thermal resistance of 0.4 K·W-1. The temperature of the heatsink at the MOSFET T_H must therefore be lower than 136 °C.

4.9.2 Contribution of Stage 1 Diodes

In section 4.6.8 the total losses have been calculated as 9.6 W per diode. The total is therefore 38.4 W for the four diodes.

According to the datasheet, the junction to case thermal resistance of the diodes is given as 2.6 K·W-1. The maximum junction temperature is given by the datasheet as 175 °C. This implies a maximum case temperature of 150 °C.

The thermal resistance of the pads used to isolate the heatsink from the MOSFETs have a thermal resistance of 0.4 K·W-1. The temperature of the heatsink at the diode must therefore be lower than 146 °C.

4.9.3 Contribution of Stage 2 MOSFETS

In section 4.8.6 the losses for the top and bottom MOSFET has been calculated as 29.706 W and 891 mW respectively.

According to the datasheet, the junction to case thermal resistance θ_{JC} of the MOSFETs is given as 0.5 K·W-1. The maximum junction temperature is given by the datasheet as 150 °C. This implies a maximum case temperature for the top and bottom MOSFET of 135 °C and 149.55 °C respectively.

The thermal resistance of the pads used to isolate the heatsink from the MOSFETs have a thermal resistance of 0.4 K·W-1. The temperature of the heatsink at the MOSFET must therefore be lower than 123 °C for the top MOSFET and 149.2 °C for the bottom one.

4.9.4 The Required Heatsink

All the above components are to be mounted onto the same heatsink. The temperature of that heatsink must be kept lower than 123 °C. The total power dissipation is 117.742 W. Assuming a maximum ambient of 45 °C, the thermal resistance of the heatsink must be smaller than 0.6625 K·W-1

According to the datasheet, a natural-convection SK109 heatsink from Fischer Elektronik that is 200 mm long has a thermal resistance of 0.6 K·W-1. When mounted horizontally and

cooled by forced convection at 1 m·s^{-1}, this drops to 0.36 K·W^{-1}, which is small enough for the application.

4.10 Physical Layout

Due to the high switching frequency, care has to be taken to ensure minimal EMI effects. The small-signal components need electromagnetic shielding from the large transients present during switching. Two separate PCBs can thus be used with a solid copper plate between them. This layout is depicted in figure 4.26.

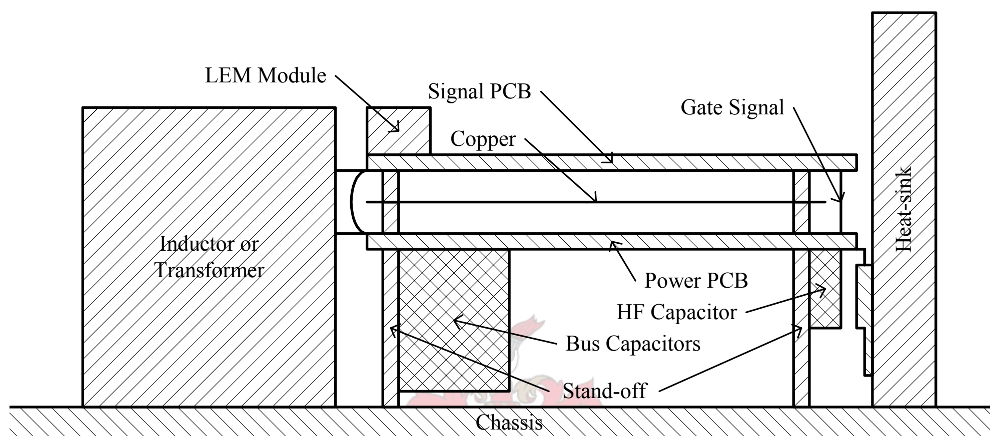


Figure 4.26: Converter layout

Furthermore, great care should be taken to ensure the shortest possible path for any high-current signal. Loops are to be avoided at all cost. Resistors (1 kΩ) are soldered onto the MOSFET between gate and source as close to the package as possible. This is to provide damping for any EMI induced pulse on the gate. High-frequency 470 nF capacitors with a very low equivalent resistance are included as close to the MOSFETs as possible in order to stabilise the bus.

The LEM modules used to measure the current through the inductor (and load) are situated as close to the inductor (or load) as possible. They are installed on the capacitor side so that they do not experience large voltage transients.

The breakdown voltage of air is 2 MV·m^{-1} [2]. This means that for the possible 1 000 V spikes, tracks should be placed no closer than 0.5 mm apart. In a practical situation, however, the PCB may become dirty with conducting substances, such as salt or damp dust. It is thus advisable to place no two tracks nearer than 3 mm apart.

The breakthrough voltage of standard PCB material (according to the Polyclad240 datasheet) is 30 MV·m^{-1}. The separation between layers is 1.5 mm. The maximum voltage between the two layers is thus 45 kV which is much larger than the expected 1 kV spikes.

The copper thickness on the power PCB is 70 μm. This means that a conductor carrying 15 A must be 8 mm wide. Similarly, a 6 A track must be 2 mm and a 12 A track 6 mm. These widths have been calculated using the "Track Amp Calculator" from North Tech Services (http://www.ntspcb.co.za). A temperature difference of 10 °C was assumed.

4.11 Summary

After considering the requirements, a two-stage topology was chosen. The two stages are implemented as an isolated full-bridge (figure 4.10) and a synchronous buck converter (figure 4.21) respectively.

The bus capacitor was designed as two 470 μF, 400 V capacitors in series. Both filter capacitors are the same 400 V, 470 μF capacitors. The filter inductors L_1 and L_2 were calculated to be 3 mH and 200 μH respectively. Their physical construction was provided.

The transformer was designed with a 1:1 winding ratio and twisted-pair windings. The use of this transformer is valid if the high frequency rectifier diodes are replaced with diodes that can withstand 800 V.

The leakage inductance was measured as 4 μH and the effect was studied.

The total losses of the components to be bolted to the heat sink were calculated as 118 W. A fan would thus need to be included as part of the casing.

Chapter 5

Control System

5.1 Introduction

In chapter 4 the converter was designed. The next component on the list is the control system. This chapter includes all aspects of the control system implemented for the converter. It starts with a discussion and mathematical model of the system to be controlled.

Next, different control strategies are presented, along with block diagrams. The final control system is designed using continuous time design methods. The measurements taken, along with their respective accuracy and precision, are then considered. Simulation results are presented. The implementation of the control system in firmware is presented and discussed.

Throughout this chapter, reference is made to signals and components of the converter. Figures 4.10 on page 34 and 4.21 on page 47 show the circuit diagrams of both stages, along with all the signals referenced in this chapter. The duty-cycle of the first stage is D_1 and that of the second stage D_2.

5.2 Specification

5.2.1 Range

The converter output voltage range was specified in section 4.3.4 on page 24 as 0 V to 250 V. The converter output current range is 0 A to 15 A.

Similarly, the stage 1 output voltage range is 0 V to 312.5 V. The stage 1 output current range is 0 A to 12 A.

5.2.2 Bandwidth

The load is characterised by an inductance of 15 H and a series resistance of 14.4 Ω. The time-constant τ of this load is thus 1.11 s (equation 5.1). This corresponds to a bandwidth β of 153 mHz (equation 5.2).

The settling time of the output current i_2 is specified as 5 s upon any step input. Equation 5.3 shows a simple 1st order response to a unity step where τ is the system time-constant and y the output.

$$\tau = \frac{L}{R} \tag{5.1}$$

$$\beta = \frac{1}{2\pi\tau} \tag{5.2}$$

$$y = 1 - e^{-t/\tau} \tag{5.3}$$

By finding the error between y and unity, the desired 10 ppm level is reached after $11.5{\cdot}\tau$. As this calculation assumes that the load is uncharged at time zero, it represents the worst case scenario. The closed-loop time constant must thus be smaller than 435 ms. This corresponds to a bandwidth of 366 mHz. Compensating for effects due to the high order of the system and practical shortcomings, a closed-loop bandwidth of 1 Hz was chosen.

5.2.3 Accuracy

The accuracy of the current source is dependant on, among others, the accuracy of measurement. Say, for example, the control system is given a 1 A reference. It measures the output as being 1 A. There would therefore seem to be no error. The reality is that the measurement of the output current is 5% in error with respect to the absolute value. Such a system would then have an accuracy of 5%.

The specified accuracy of the system is 10 ppm.

Pectora roborant cultus recti

5.2.4 Stability

The stability of the system is specified as 10 ppm. This means that the output current of the converter may not drift more than 10 ppm. It also means that the output current ripple may not be larger than 10 ppm.

5.2.5 Resolution

The stability of the system has been specified as 10 ppm. At minimum load (1 A), this is 10 μA, corresponding to a magnetic flux density resolution of 800 nT The specification in appendix A requires a resolution of 1 μT, which would imply that it is possible to implement - given that the stability of 10 ppm was reached.

A resolution on 1 μT with a range of 1.2 T would imply a reference word 24 bits wide.

5.3 Block Diagrams

5.3.1 System Diagram

The initial aspect to consider is whether to control the system as a whole, or to link the two separate control systems by using stage 2 to drive stage 1's reference. By making stage 1 react at least ten times faster than stage 2 it is possible to control them separately.

This is convenient for the implementation of feed-forward. Feed-forward is the method of choice in many existing implementations [3; 9; 12]. The control system for stage 2 can specify the voltage it requires at the input of its filter (v_3). The control system for stage 1 can then ensure that the required stage 2 duty-cycle is approximately 80%. The exact duty-cycle is then calculated by the feed-forward implementation. Figure 5.1 shows the full control system block diagram, containing all the aspects discussed earlier. More detail about each block is presented in the following sub-sections.

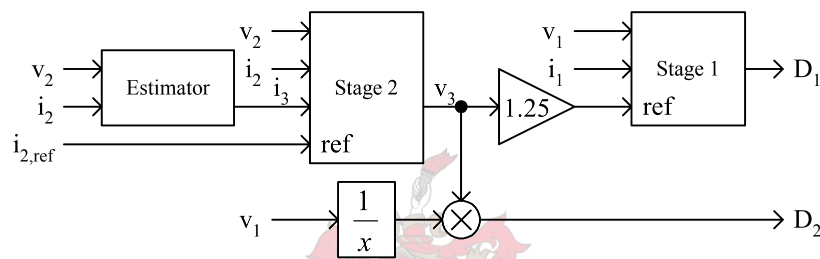


Figure 5.1: Control system block diagram

5.3.2 Stage 1

Many different control strategies exist for an isolated full-bridge topology, the simplest of which is when voltage is fed back to the PWM generator with a lead-lag network for stability [6]. This is depicted in figure 5.2. The lead-lag network may also be replaced with a proportional-integral (PI) controller [11; 12].

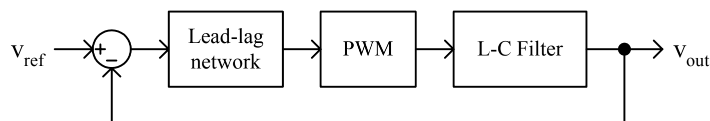


Figure 5.2: Voltage control

More commonly used is a fast current-feedback inner-loop (usually hysteresis control) with a slower voltage-feedback outer-loop [3]. This is presented in figure 5.3. The greatest

advantage is its high level of immunity to bus-voltage disturbances. Another significant advantage is the presence of an inherent over-current protection.

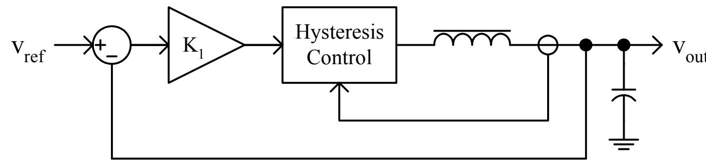


Figure 5.3: Current (hysteresis) control

The current control loop usually operates by switching the voltage across the transformer on at the start of the switching period. It is switched off when the inductor current reaches a pre-determined value. There are two main types of hysteresis control. Peak-current control switches the voltage off when the actual inductor current reaches the required value. Average current control puts the measured inductor current through a filter first, resulting in a slight lag.

The latter is more stable. Reasons for this include the fact that the filter reduces the switching noise from the measured current waveform. The input to the comparator is therefore more stable.

Another current-control strategy is to simply control the current by using a proportional controller [3]. This is illustrated in figure 5.4.

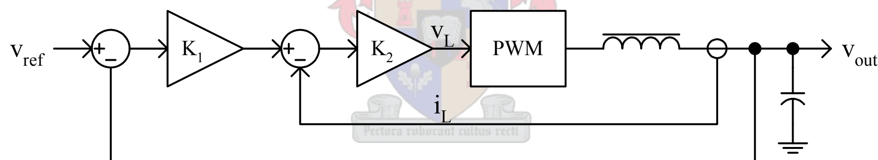


Figure 5.4: Current (feedback) control

The control law of this controller is given in equation 5.4. It has the same form as a full state feedback controller [14]. The advantage of state-space models and full state feedback is the ability to use pole-placement design techniques. This provides precise control over the behaviour of the closed-loop system. Full state feedback by means of pole-placement design is thus used.

$$v_L = (-K_1K_2)\, v_{out} + (-K_2)\, i_L + (K_1K_2)\, v_{ref} \tag{5.4}$$

The stage 1 block diagram is presented in figure 5.5.

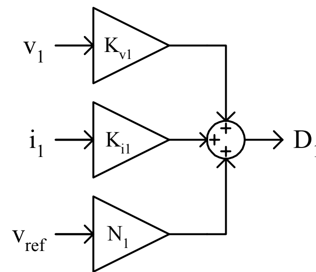


Figure 5.5: Stage 1 control system

5.3.3 Stage 2

All the states are available either through measurement (v_2, i_2) or estimation (i_3). Therefore it is possible to use full state feedback for stage 2 as well. One great advantage is that by simply modifying the value of the four constants involved (each of the three feedback gains and the gain-compensation) the control system can be calibrated for any given load. Auto-calibration is also possible with a look-up table.

In order to follow the reference with zero error, an integrator is included. The easiest way of implementing this is to control the system with full state feedback. The resulting closed-loop system may then be controlled by means of an integrator feedback loop. The final system block diagram is presented in figure 5.6.

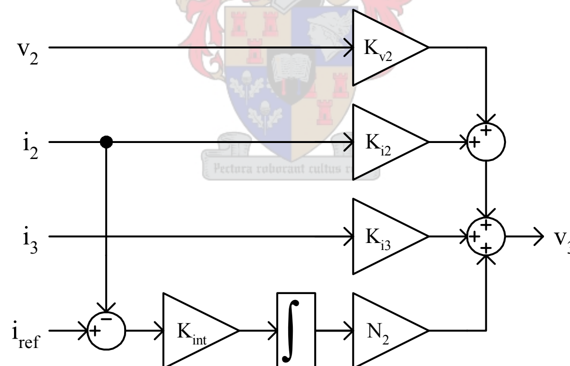


Figure 5.6: Stage 2 control system

The integrator gain (K_{int}) and gain-compensation (N_2) are kept separate in order to ease initialisation. Initialisation is performed by loading the integrator with an initial value equal to the measured value of i_2.

There exists a problem with the integrator during large transients. One example is the case where the reference is stepped from full load to minimum load. The integrator gain was designed assuming that the output voltage has infinite range. This is not the case, as the output voltage saturates to 0 V. The current error therefore exists for a longer time because

the load inductor discharges over a longer period. The resulting integrator answer would then be much more negative than it would have been in an ideal system. The consequence of this is that the output current remains smaller than the reference until the integrator has returned to normal, which is longer than designed. The same problem exists when the reference is stepped from minimum load to full load.

There are many solutions to this problem. The first is to ramp-limit the reference so that the conditions for the problem never exist. This solution would work, but it limits the bandwidth for references that would not have caused a problem.

The second is to switch the integrator off whenever the output current differs by more than 10% from the reference. The integrator is then reinitialised and switched back on when the output current is within 5% of the reference. This solution assumes that the output current resulting from the standard feedback controller does not differ by more than 5% from the refernce, which cannot be guaranteed.

The third is to internally limit the integrator answer to have a range equal (or slightly larger) to the expected range under ideal conditions. This is the most elegant solution.

The integrator block diagram, adapted from [26] is shown in figure 5.7. The 'z^{-1}' block is implemented as a D-Q latch. The time-step (Δt) is 167.772 16 μs.

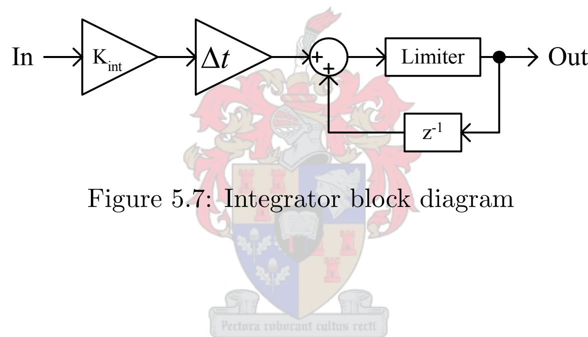


Figure 5.7: Integrator block diagram

5.3.4 Estimator

The estimator calculates i_3 according to equation 5.4. The block diagram of this system is given in figure 5.8. One again, the 'z^{-1}' block is implemented as a D-Q latch. The time-step (Δt) is 167.772 16 μs.

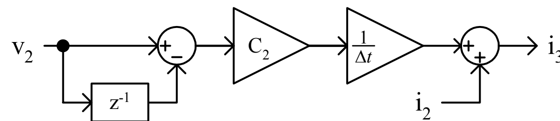


Figure 5.8: Estimator block diagram

5.3.5 Feed-Forward

There are many options for controlling the second stage. A full state feedback control system that controls the duty-cycle directly is very plausible, but does not provide the same ripple rejection properties as feed-forward does. Current-mode control is also possible and does provide superior ripple rejection, but is difficult to implement at the switching frequency involved, especially digitally.

In order to implement feed-forward, the reference to stage 1 is set such that the duty-cycle of stage 2 is approximately 80%, as explained in section 5.3.1. This is 1.25 times the required input voltage (v_3). Next, the actual duty-cycle is calculated with equation 5.5.

$$D_2 = \frac{v_3}{v_1} \tag{5.5}$$

It is advantageous to calculate the reciprocal of v_1 first and multiply it by v_3 instead of calculating the quotient directly. There are two main reasons. The first is to increase the resolution of the quotient. The output of the implemented divider is 18 bits wide while the output of a multiplier is 36 bits wide. The full-scale value of the quotient is 32, which is then limited to 1 for the duty-cycle. Five bits of resolution are lost during limiting. Limiting the output of the multiplier results in a 31-bit answer, which is then truncated to 18-bit. Limiting the output of the divider yields a 10-bit answer. There is thus no loss of bits when a multiplier output is used, but 5 bits are lost when using the divider output.

The second is to ease the implementation regarding timing issues. The divider needs to latch both numerator and denominator before it can start the conversion. The only delay on the denominator (v_1) is the limiter, whereas the numerator (v_3) requires the entire controller to settle before it can be latched, causing a significant delay.

The reciprocal is determined by performing a binary search for the correct quotient. Another technique is polynomial approximation (such as least-squares or Taylor series [27]), a faster method, but less accurate when used for practical order polynomials.

5.4 System Models

5.4.1 Stage 1

The L-C filter reacts only on the average of the pulse-train at the output of the rectifier. This average is equal to the bus voltage multiplied by the duty-cycle. Also, the series resistances of both the L and the C components must be considered.

The second stage converter follows on the other end of the filter. That converter ensures, by means of feed-forward, that v3 remains constant (in steady state). The current i3 is also constant, which implies that stage 1 has a constant power load.

This constant power load is non-linear. It can be linearised, however, about the steady state condition. The relationship between voltage and current at steady state is given in equation 5.6. The power is given by equation 5.7. Linearising at the bias point implies

that the Norton-equivalent circuit is a -22.5 Ω resistor (R1) in parallel with a current source twice that of the steady state current. This is shown in equations 5.8 to 5.11.

$$v_{ss} = \frac{R_{Load}}{D_2^2} i_{ss} \tag{5.6}$$

$$P = v_{ss} i_{ss} \tag{5.7}$$

$$v = \frac{v_{ss} i_{ss}}{i} \tag{5.8}$$

$$\frac{dv}{di} = -\frac{v_{ss} i_{ss}}{i^2}\bigg|_{i=i_{ss}} \tag{5.9}$$

$$\frac{dv}{di} = -\frac{R_{Load}}{D_2^2} \tag{5.10}$$

$$i = \left(-\frac{D_2^2}{R_{Load}}\right) v + 2i_{ss} \tag{5.11}$$

This biasing current ($2i_{ss}$) is not constant, however. Once again assuming a steady state condition, equation 5.12 shows the biasing current in terms of the output voltage v_1. This corresponds to an 11.25 Ω resistor (R_2) in the place of the current source.

$$i_{ss} = \frac{2D_2^2}{R_{Load}} v_1 \tag{5.12}$$

In order to have the effect of a constant current source at the controller frequency as well as a self-adjusting model, an inductor (L_3) is included. This circuit is provided in figure 5.9. The value of this inductor must be large enough to emulate a constant-current source at the line frequency (300 Hz). It must also be small enough for the model's bias-point to follow the real bias-point.

The load power has a small bandwidth (1 Hz). The frequency associated with the inductor has been chosen at 30 Hz (ten times slower than the line-frequency ripple yet thirty times faster than the load). This implies the need to use a 60 mH inductor (equation 5.13).

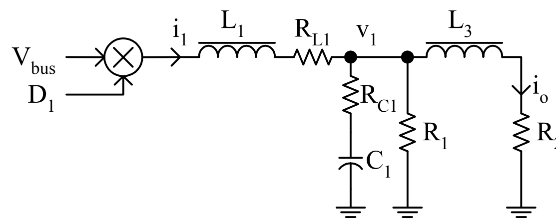


Figure 5.9: Stage 1 circuit model

$$L = \frac{R}{2\pi f} \tag{5.13}$$

This model is valid only at the steady state condition. It has been shown by simulation, however, that the control system arising from this model performs well. Results of this simulation are provided in section 5.7. As full state feedback and pole-placement techniques are used, a state-space model of this system is required [14]. By using equations 5.14 to 5.16 [23], the model may be derived through algebraic manipulation of the node-voltage equations [23]. The result is given by equations 5.17 through 5.19. By allocating a vector $\mathbf{x}_1$, as in equation 5.20, the system model can be written in the form given by equations 5.21 and 5.22, where $\mathbf{A}_1$, $\mathbf{B}_1$ and $\mathbf{C}_1$ are matrices given by equations 5.23 to 5.25.

$$v = iR \tag{5.14}$$

$$v = L\frac{di}{dt} \tag{5.15}$$

$$i = C\frac{dv}{dt} \tag{5.16}$$

$$\frac{dv_1}{dt} = \begin{cases} \left(\frac{-R_1R_{C1}L_3C_1 - R_1R_{C1}L_1C_1 - L_1L_3}{L_1L_3C_1(R_1+R_{C1})}\right)v_1 + \\ \left(\frac{R_1L_1 - R_1R_{C1}R_{L1}C_1}{L_1C_1(R_1+R_{C1})}\right)i_1 + \\ \left(\frac{R_1R_2R_{C1}C_1 - R_1L_3}{L_3C_1(R_1+R_{C1})}\right)i_0 + \\ \left(\frac{R_1R_{C1}V_{bus}}{L_1(R_1+R_{C1})}\right)D_1 \end{cases} \tag{5.17}$$

$$\frac{di_1}{dt} = \left(-\frac{1}{L_1}\right)v_1 + \left(-\frac{R_{L1}}{L_1}\right)i_1 + (0)\,i_0 + \left(\frac{V_{bus}}{L_1}\right)D_1 \tag{5.18}$$

$$\frac{di_0}{dt} = \left(\frac{1}{L_3}\right)v_1 + (0)\,i_1 + \left(-\frac{R_2}{L_3}\right)i_0 + (0)\,D_1 \tag{5.19}$$

$$\mathbf{x}_1 = \begin{bmatrix} v_1 \\ i_1 \\ i_0 \end{bmatrix} \tag{5.20}$$

$$\dot{\mathbf{x}}_1 = \mathbf{A}_1\mathbf{x}_1 + \mathbf{B}_1 D_1 V_{bus} \tag{5.21}$$

$$v_1 = \mathbf{C}_1\mathbf{x}_1 \tag{5.22}$$

$$\mathbf{A}_1 = \begin{bmatrix} \frac{-R_1R_{C1}L_3C_1 - R_1R_{C1}L_1C_1 - L_1L_3}{L_1L_3C_1(R_1+R_{C1})} & \frac{R_1L_1 - R_1R_{C1}R_{L1}C_1}{L_1C_1(R_1+R_{C1})} & \frac{R_1R_2R_{C1}C_1 - R_1L_3}{L_3C_1(R_1+R_{C1})} \\ -\frac{1}{L_1} & -\frac{R_{L1}}{L_1} & 0 \\ \frac{1}{L_3} & 0 & -\frac{R_2}{L_3} \end{bmatrix} \tag{5.23}$$

$$\mathbf{B}_1 = \begin{bmatrix} \frac{R_1 R_{C1}}{L_1(R_1+R_{C1})} \\ \frac{1}{L_1} \\ 0 \end{bmatrix} \tag{5.24}$$

$$\mathbf{C}_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \tag{5.25}$$

5.4.2 Stage 2

The second stage is much simpler. The input to the system is given as v_3, which is guaranteed by the feed-forward process. The relevant circuit diagram is thus provided by figure 5.10.

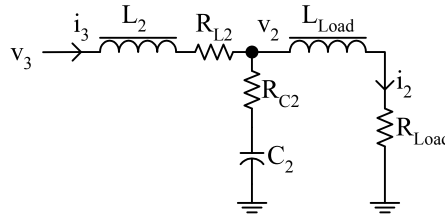


Figure 5.10: Stage 2 circuit model

The state-space model may be derived by limiting R_1 to infinity in equations 5.17 to 5.19 and substituting the relevant constants. The result is given by equations 5.26 to 5.31.

$$\dot{\mathbf{x}}_2 = \mathbf{A}_2\mathbf{x}_2 + \mathbf{B}_2 v_3 \tag{5.26}$$

$$i_2 = \mathbf{C}_2\mathbf{x}_2 \tag{5.27}$$

$$\mathbf{x}_2 = \begin{bmatrix} v_2 \\ i_2 \\ i_3 \end{bmatrix} \tag{5.28}$$

$$\mathbf{A}_2 = \begin{bmatrix} -\frac{R_{C2}}{L_2} - \frac{R_{C2}}{L_{Load}} & \frac{R_{Load}R_{C2}}{L_{Load}} - \frac{1}{C_2} & \frac{1}{C_2} - \frac{R_{L_2}R_{C2}}{L_2} \\ \frac{1}{L_{Load}} & -\frac{R_{Load}}{L_{Load}} & 0 \\ -\frac{1}{L_2} & 0 & -\frac{R_{L2}}{L_2} \end{bmatrix} \tag{5.29}$$

$$\mathbf{B}_2 = \begin{bmatrix} \frac{R_{C2}}{L_2} \\ 0 \\ \frac{1}{L_2} \end{bmatrix} \tag{5.30}$$

$$\mathbf{C}_2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \tag{5.31}$$

5.4.3 Integrator

Figure 5.11 shows a closed loop system with an integrator for removing the steady state error.

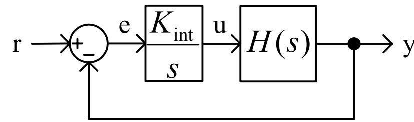


Figure 5.11: Integrator control

System H(s) is modelled as a state-space system implementing full state feedback, as in equations 5.32 and 5.33. The state-space representation of the integrator is given by equation 5.34. The closed-loop system can thus be modelled using equations 5.35 and 5.36.

$$\dot{\mathbf{x}} = [\mathbf{A} + \mathbf{BK}]\,\mathbf{x} + \mathbf{B}Nu \tag{5.32}$$

$$y = \mathbf{Cx} \tag{5.33}$$

$$\dot{u} = K_{int}e \tag{5.34}$$

$$\begin{bmatrix} \dot{\mathbf{x}} \\ u \end{bmatrix} = \begin{bmatrix} \mathbf{A} + \mathbf{BK} & \mathbf{B}N \\ -K_{int}\mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ u \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ K_{int} \end{bmatrix} r \tag{5.35}$$

Pectora roborant cultus recti

$$y = \begin{bmatrix} \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ u \end{bmatrix} \tag{5.36}$$

5.5 Pole Placement

5.5.1 Stage 1

The combination of L_3 and R_2 results in a zero at $s = -R_2/L_3$. In order for L_3 and R_2 to have negligible effect, the first pole is placed as near as possible to this zero. Since the current through L_3 is a fictional current used only in simulation, the feedback gain associated with it is made zero. This moves the pole slightly, but not enough to cause significant change in the step-response.

The maximum bandwidth of the closed-loop system is partly determined by the bandwidth of the measurements (5 kHz). Making the system bandwidth ten times lower than this ((500 Hz) causes the system to become insensitive to measurement phase-error.

The control law is given by equation 5.37. By writing the resulting state-space representation (equation 5.38), it is clear that a disturbance in bus voltage results in a slight movement of the poles and a shift in reference voltage. This in turn results in a shift in steady state voltage, an effect that must be minimised. The steady state voltage is given by equation 5.39. Conveniently, a smaller reference implies that bus voltage disturbances have a smaller effect.

$$D_1 = \mathbf{K}_1 \mathbf{x}_1 + N_1 v_{ref} \tag{5.37}$$

$$\dot{\mathbf{x}}_1 = [\mathbf{A}_1 + \mathbf{B}_1 \mathbf{K}_1 V_{bus}] \, \mathbf{x}_1 + \mathbf{B}_1 N_1 v_{ref} V_{bus} \tag{5.38}$$

$$v_{1,ss} = \mathbf{C}_1 \cdot [\mathbf{A}_1 + \mathbf{B}_1 \mathbf{K}_1 V_{bus}]^{-1} \cdot [-\mathbf{B}_1 N_1 v_{ref} V_{bus}] \tag{5.39}$$

Various options for placing the poles are available. The first option is to use a 3rd order Butterworth filter prototype [23]. This means that poles are placed as in equation 5.40. Table 5-I shows the rejection obtained for a 75 V (peak to peak), 300 Hz sinusoidal disturbance. The values provided are in relation to the output voltage. From this table it is clear that the higher the bandwidth, the better the rejection. As a result, the bandwidth of 500 Hz calculated above was retained.

$$s = \begin{cases} -\omega \\ \frac{-\omega \pm j\omega\sqrt{3}}{2} \end{cases} \tag{5.40}$$

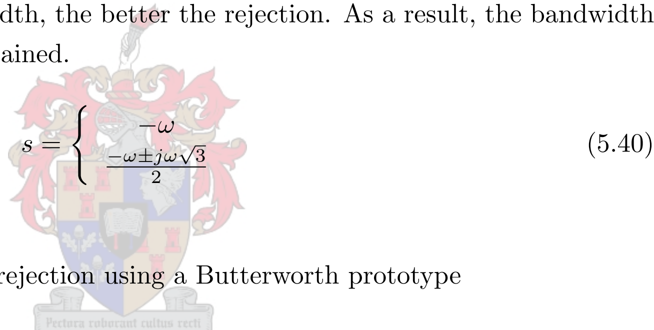


Table 5-I: Ripple rejection using a Butterworth prototype

Bandwidth [Hz]	**Change in steady-state voltage [dB]**	**Rejection due to dynamics [dB]**	**Total rejection [dB]**
100	2	16	14
200	-8	6	14
300	-15	2	17
400	-19	0	19
500	-23	0	23
600	-26	0	26
700	-29	0	29
800	-31	0	31
900	-33	0	33
1 000	-35	0	35

There is another option for pole-placement. This is to consider the often-used fast current inner loop and slow voltage outer loop control, as described in section 5.3.2. This relates to placing the one pole at 500 Hz and the other much further away from the origin. The faster

pole is then related to the current through the inductor. A higher bandwidth measurement is required. The LEM modules used to measure the current have a 0.5 dB bandwidth of 100 kHz. A good cut-off frequency for the ADC filer is thus 100 kHz (1 nF capacitor in the place of the 22 nF). Unfortunately the voltage measurement must be as noiseless as possible for feed-forward purposes and thus the 5 kHz bandwidth is retained for v_1. Table 5-II shows the effect of placing the slow pole at 500 Hz and the faster one at other frequencies. The same 75 $V_{p\text{-}p}$, 300 Hz disturbance is used.

Table 5-II: Ripple rejection using faster current feedback

Fast pole placement [Hz]	**Change in steady-state voltage [dB]**	**Rejection due to dynamics [dB]**	**Total rejection [dB]**
500	-23	2	25
700	-26	1	27
1 000	-29	1	30
1 400	-32	1	33
2 000	-35	1	36
2 600	-37	1	38
3 700	-40	1	41
5 000	-43	1	44
7 000	-46	1	47
10 000	-49	1	50

By using this technique and placing both poles ten times slower than their respective measurement filters a, 50 dB rejection is possible. This is an improvement on the 23 dB achieved when using a Butterworth prototype.

As a final consideration, the switching frequency is 47.684 kHz. 10 kHz is 5 times slower than this, an acceptable condition, although 10 times slower would be advised. The poles are thus placed at 500 Hz and 5 kHz.

The desired characteristic equation for arbitrary pole placement is given by equation 5.41. The order of the system is given by n. By using Ackermann's formula [14] (described by equations 5.42 to 5.44) the poles can be placed directly. Matrix $\mathcal{C}$ is the controllability matrix – for the system to be controllable, this matrix has to be invertible. The α terms in these equations represent the coefficients of the characteristic equation of the desired closed-loop system.

$$\alpha_c(s) = s^n + \alpha_1 s^{n-1} + \alpha_2 s^{n-2} + \cdots + \alpha_n = 0 \tag{5.41}$$

$$\alpha_c(\mathbf{A}) = \mathbf{A}^n + \alpha_1 \mathbf{A}^{n-1} + \alpha_2 \mathbf{A}^{n-2} + \cdots + \alpha_n \mathbf{I} \tag{5.42}$$

$$\mathcal{C} = \begin{bmatrix} \mathbf{B} & \mathbf{AB} & \mathbf{A}^2\mathbf{B} & \cdots & \mathbf{A}^{n-1}\mathbf{B} \end{bmatrix} \tag{5.43}$$

$$\mathbf{K} = \begin{bmatrix} 0 & \cdots & 0 & 1 \end{bmatrix} \mathcal{C}^{-1} \alpha_c(\mathbf{A}) \tag{5.44}$$

After implementation of such a full state feedback controller, the system will be stable. It is guaranteed stable because the position of the closed-loop poles have been placed on the left-hand side of the complex plane.

The steady state of the output is not guaranteed to equal the reference. This steady state error requires removal. Equations 5.45 and 5.46 describe how to calculate the reference gain (N) that will result in a total removal of this error. These equations are repeated from [14]. The calculated gains are listed in table 5-III.

$$\begin{bmatrix} \mathbf{N}_x \\ N_u \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & 0 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} \tag{5.45}$$

$$N = N_u - \mathbf{K}\mathbf{N}_x \tag{5.46}$$

Table 5-III: Stage 1 gains

Gain	Value
V_1	-0.2666
I_1	-0.1915
N_1	0.2769

5.5.2 Stage 2

The second stage essentially consists of three parts. First the control system designed by pole-placement (H(s)). Second the steady state error is removed by means of scaling the reference. Third the integrator is designed, where the gain must be determined by alternative methods.

By placing the poles of $H(s)$ using a Butterworth prototype ten times faster than the closed-loop bandwidth (10 Hz), the calculation of the integrator gain is straightforward. This also enables the closed-loop system to have a single-order response, with the advantage of no overshoot.

By inserting a pole at the origin (representing the integrator) and drawing the root-locus, the effect of the integrator gain may be noted. The two real poles move towards each other at an equal rate. The two complex poles do the same, but to a lesser extent. This is shown in figure 5.12. The crosses represent the original poles and the squares the resulting closed-loop poles.

The poles for $H(s)$ are thus placed using an 11 Hz Butterworth prototype. These poles are listed in equation 5.47. In the closed-loop system, the real pole is moved to 10 Hz and

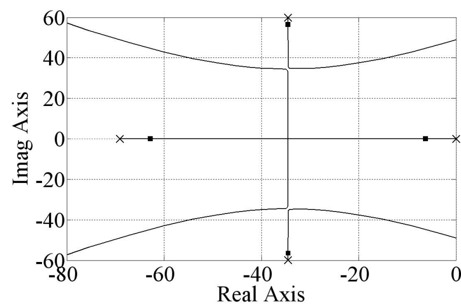


Figure 5.12: Root locus of stage 2

the imaginary poles to 10.5 Hz. Once again Ackermann's formula was used, along with the same reference-gain calculation mentioned above.

$$s = \begin{cases} -69.1150 \\ -34.5575 \pm 59.8554j \end{cases} \tag{5.47}$$

An iterative technique is then followed to find the integrator gain that results in a pole at 1 Hz. The eigenvalues of the **A** matrix in equation 5.32 (on page 65) is calculated to determine the closed-loop poles [14]. A starting gain of 0 is used, which increases in unit steps until the pole is too far (passed 1 Hz) from the origin. The gain then decreases in steps of 0.1 until the pole is too close to the origin, etc. The stage 2 gains are listed in table 5-IV.

Table 5-IV: Stage 2 gains

Gain	Value
V_2	0.9992
I_2	-0.0857
I_3	0.0231
N_2	0.1041
K_{int}	5.2401

These represent the gains needed for a 4 H, 14.4 Ω load, which is the geometric average of the two extreme values. The closed-loop poles are given in equation 5.48. According to simulation (presented in section 5.7), both a 1 H and 15 H load still result in a stable system.

$$s = \begin{cases} -6.2832 \\ -62.8339 \\ -34.5565 \pm 56.4618j \end{cases} \tag{5.48}$$

5.6 Measurements

5.6.1 Analogue vs Digital

There are obvious advantages to use a digital control system for stage 2. These include the ease of feed-forward implementation, adaptability of the control system, etc.

There are many analogue control ICs available on the market for an isolated full-bridge topology. They implement many high-performance control techniques and are inexpensive. It is possible to provide one of these ICs with a reference signal from a 12-bit DAC, resulting in the correct output voltage needed for the input to stage 2.

This 12-bit DAC costs about the same as the 12-bit ADC needed to measure i_1, solving the cost issue. Also, a high-precision measurement of v_1 is taken for feed-forward purposes, which can be used for the stage 1 control system. The FPGA is already present, and so can be used to control the first stage as well as the second stage. A digital control system also offers ease of adjustment to different control specifications. The entire control system is thus implemented digitally.

This choice of digital control also has other implications. There is no possibility of peak current control, for instance. This is because the ADCs sample too slowly. Faster ADCs have lower resolution, which is not acceptable in this high-precision application.

5.6.2 Resolution

The resolution of the measurements is mainly dictated by the available technology. The highest resolution, >400 ksps, single-sample latency ADC that was found is the AD7674 from Analog Devices. It samples at 800 ksps with a resolution of 18-bit. This is coincidently a convenient resolution, since the FPGA has on-board 18-bit multipliers.

Upon closer inspection, a resolution of 18-bit does not seem precise enough. Use v_1 as an example: it has a full-scale value of 400 V, with a resolution of 1.526 mV. This is satisfactory for the control of stage 1, but not for the feed-forward of stage 2. At minimum-load, the first stage output voltage is 18 V and 10 ppm of that is 180 μV – 8.5 times smaller than the available resolution. An extra 4 bits resolution is thus required for proper control.

There is a random component to the signal due to Gaussian noise, switching transients and switching ripple. If the past 16 samples are averaged, a 22-bit signal is obtained. These 16 samples correspond exactly to one switching cycle of stage 1 – i.e. all four switching states. The result will thus always be approximately that of the cycle-average. The standard deviation of the noise is reduced by a factor four ($\sqrt{16}$) [28].

In [11] output current measurement noise is reduced by taking the measurement at three different points on the circuit and using two different techniques. These measurements are then averaged. This is shown in figure 5.13.

Another technique that may be used to improve analogue to digital conversion is to utilise two separate ADCs of different specifications – one with high-resolution, but a low sample rate and the other with a high sample rate, but low-resolution. The inputs to the fast- and slow ADCs pass through complementary high- and low-pass filters respectively. When the

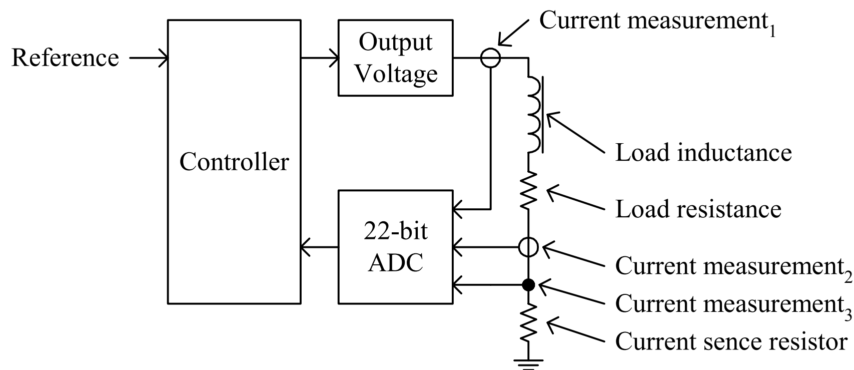


Figure 5.13: Multiple measurements

two measurements are then added, a high sample rate is obtained at high frequencies and a high resolution at low frequencies. This technique works well when the closed-loop system is required to have a fast settling time, but is not needed to follow a high-bandwidth signal with great accuracy.

This project requires that the line-frequency ripple on v_1 be removed accurately, and therefore requires the high-frequency measurement to have as small as possible latency. Very high resolution ADCs have large latencies and therefore would not be the correct choice for use in this project.

A third option exists. A multiplying DAC (such as the DAC7800 from Texas Instruments) can be used to scale the input signal. A small input signal can then be scaled 1:1, whereas larger signals can be scaled 2:1, 4:1, etc. This means that the sampled signal has better resolution for smaller signals, which would then satisfy the 10 ppm requirement.

Pectora roborant cultus recti

5.6.3 Actual Measurements

High-resolution ADCs are expensive and careful consideration must be taken as to which measurements are necessary. The mathematical models of the circuit are well known and it is therefore possible to take as few measurements as possible. In order for protection circuitry to function properly, however, more measurements are required.

There are two high-precision measurements which are absolutely necessary. These are the output current i_2 (for using current feedback when the NMR measurement is unavailable) and the feed-forward voltage v_1.

There are also two less important measurements necessary. These are the output voltage v_2 (for determining the load characteristics) and the stage 1 inductor current i_1 (for control and protection purposes). These may be calculated using the mathematical model of the circuit (described in the following section) and therefore do not need to have high precision.

Since the ADCs were obtained on sample, however, the same AD7674 ADCs were used. It is also convenient for prototyping purposes, since a lower resolution ADC can very easily be simulated using measurements from one with a high resolution, but not vice versa.

These measurements are taken at a high sampling rate (381.470 kHz) so that the switching-ripple of the first stage PWM signal (47.684 kHz) may be averaged out of the measurement completely. It is not necessary, to have a high bandwidth. In order to minimise noise and instability, as well as better the calculated average, a low-pass filter can be placed as close to the ADC as possible. This is a simple single order R-C filter. To determine its cut-off frequency, a number of factors must be considered.

First, the line frequency ripple needs to be attenuated by the control system. This is only possible if it can be measured. In section 4.5.3 on page 4.5.3 the line-frequency harmonics were calculated. Table 5-V shows these harmonics after passing through both LC filters and the load. This is not considering the attenuation provided by the control system. From this table it is clear that only the first two harmonics need to be removed by the control system, as the ripple must be smaller than -120 dB. In order for the filter to not interfere with the measurement within the frequencies of interest, a filter ten times faster than the 600 Hz harmonic is required.

Table 5-V: Line frequency harmonics at the load

Harmonic	**Frequency [Hz]**	**Magnitude [dB]**	**Load current [dB]**
1	300	-25	-75
2	600	-37	-102
3	900	-44	-135
4	1200	-49	-154
5	1500	-53	-168
6	1800	-56	-180
7	2100	-59	-190
8	2400	-61	-197
9	2700	-63	-204
10	3000	-65	-211

The primary purpose of the filter is to reduce the switching-frequency ripple so that a more accurate measurement can be made. This ripple is at 47.684 kHz. Ten times lower (to satisfy necessary design criterion) than this is approximately 4.8 kHz. The potential voltage divider at the ADC (to take the 5 V input signal down to 4.096 V) consists of a 1.82 kΩ and an 8.25 kΩ resistor. This means a thevenin-equivalent resistance of 1.49 kΩ. A 22 nF capacitor from the centre-point down to ground will provide a cut-off frequency of 4.85 kHz, which is close enough for practical purposes. This is shown in figure 5.14.

Next the full-scale values are to be considered. They must be large enough to accommodate the full practical range, as well as a bit extra for protection purposes. First the voltages are considered. The maximum output voltage of stage 1 is 375 V (by simulation). The filter capacitor has a maximum voltage rating of 400 V. A 400 V full-scale is thus chosen for v_1. The maximum voltage over the load is simulated as 300 V. For convenience, the full-scale is chosen at 80% of the full-scale of v_1 (400 V). A 320 V full-scale value was

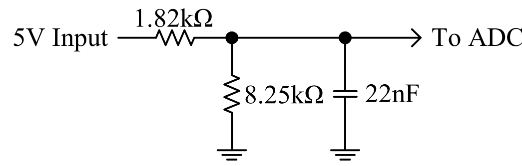


Figure 5.14: ADC pre-filter circuit diagram

thus chosen. The maximum currents through the stage 1 filter inductor (i_1) and load (i_2) are 12 A and 15 A respectively. Full-scale values of 15 A and 20 A were thus chosen.

The exact time at which to take the measurements is also an important issue. It is required that switching transients do not affect the measurement. The only time that the transistors are guaranteed to not switch is just before the rising-edge of the PWM signal. The AD7674 has an aperture delay of 2 ns (0.08% of the 2.621 44 μs sampling period). The maximum duty-cycle of the second stage PWM signal is 95% (2.49 μs). The time delay for the MOSFET to switch on or off is in the order of 200 ns. Taking the measurement at the same time as the PWM signal's rising-edge is thus optimal.

5.6.4 Estimated

The only measurement that is not taken directly is the current through the filter inductor of stage 2 (i_3). It can be calculated, however, by using equation 5.49.

$$i_3 = C_2 \frac{dv_2}{dt} + i_2 \tag{5.49}$$

As mentioned in the previous section, i_1 and v_2 can be calculated using the properties of the system. These are presented by equations 5.50 and 5.51.

$$i_1 = C_1 \frac{dv_1}{dt} + i_3 D_2 \tag{5.50}$$

$$v_2 = L_{Load} \frac{di_2}{dt} + R_{Load} i_2 \tag{5.51}$$

The load characteristics needed to calculate v_2 can be calibrated by the system. It is thus possible to have v_1 and i_2 as the only measurements. For the prototype, however, i_3 is the only estimated value – the rest are measured.

5.6.5 Noise

Any high-resolution measurement contains noise. In the case of this project, switching noise is minimised by taking the measurement just before the rising switching edge. The noise to be considered is thus thermal Gaussian noise and switching ripple. Using a 5 kHz filter before each ADC minimises the noise power and so also the measurement error.

The measurements at 381.470 ksps (f_s) include switching ripple from the first stage converter. A full switching cycle (all 4 states) occurs at a frequency of 23.842 kHz. There are thus exactly 16 samples per switching cycle, which can be averaged to remove the effect. Figure 5.15 shows the magnitude bode plot of a discreet, 16th order 'smoother' [29]. The 23.842 kHz fundamental, along with its harmonics, are thus completely removed.

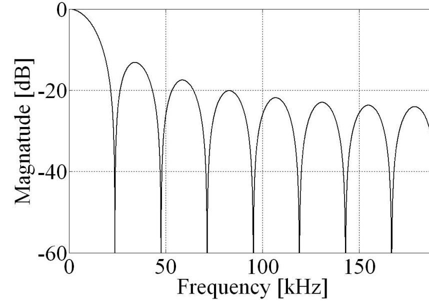


Figure 5.15: 16th order smoother bode plot

The -3 dB cut-off frequency ($f_{-3\text{dB}}$) of a smoother is given by equation 5.52, where N is the number of samples used in smoothing. By means of Newton-Raphson iteration [27], the -3 dB cut-off frequency is given by equation 5.53. The phase is given by equation 5.54, which translates to a pure delay given by equation 5.55.

$$|H(z)| = \frac{\sin\left(\frac{N\pi f}{f_s}\right)}{\left(\frac{N\pi f}{f_s}\right)} \tag{5.52}$$

$$f_{-3\text{dB}} = \frac{0.442\ 243\ 389\ 6}{N} f_s \tag{5.53}$$

$$\phi = -\frac{N\pi f}{f_s} \tag{5.54}$$

$$\Delta t = \frac{N}{2f_s} \tag{5.55}$$

The phase at 600 Hz is thus approximately -5°. The maximum error resulting from this is -21.2 dB relative to the 600 Hz component. The 600 Hz component at the load (see table 5-V) is -102 dB relative to the average current. The total error is therefore smaller than the -120 dB required and thus acceptable. The 3 dB cut-off frequency is at 10.5 kHz, which is also acceptable.

An alternative is to use every 16th sample, but this does not include the added advantage offered by a smoother. This advantage is the fact that a higher resolution measurement results from the smoothing process. It can be assumed that there exists Gaussian noise greater than one bit in magnitude superimposed on the measurement. The source of this

noise is partly thermal and partly quantisation effects. The standard deviation of the average of 16 random samples taken from a Gaussian distribution is one quarter of the standard deviation of that distribution. That implies that an extra 2 bits of resolution are obtained.

Similarly, by using a 256-sample smoother (using one M4K memory block in the cyclone II FPGA) another 4 bits are gained. An M4K block is a block of embedded memory 4 096 bits large. With this type of smoother, the output current i_2 can be measured with a resolution of 22-bit. The bandwidth of the measurement drops to 659 Hz, which is still much higher than the 1 Hz bandwidth of the controller.

5.6.6 Other

With an ultra-stable clock (<1 ppm·°C^{-1}) frequency can be measured to better than 10 ppm (provided the temperature remains within 10 °C of the average). The remaining error is removed by calibration. This accuracy is not practically possible when measuring current or voltage, however. It is thus imperative to provide the control system with the field-strength, which is dependant on a frequency measurement when using NMR techniques.

5.7 Simulation

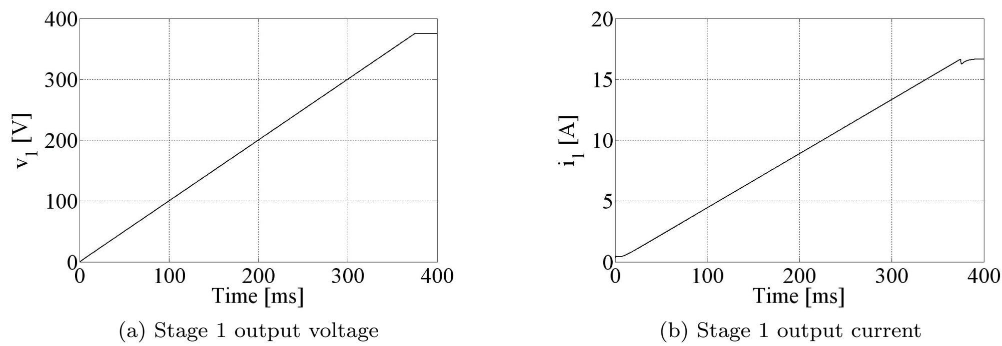
5.7.1 Stage 1

MATLAB simulations were run to validate the designed control system using a simple forward Euler integration technique [27]. The circuit in figure 5.9 was implemented. Table 5-VI lists the circuit parameters. Figure 5.16 shows the response of the system to a ramped reference of 1 kV·s^{-1} from 0 to 375 V.

Table 5-VI: Stage 1 simulation circuit parameters

Parameter	**Value**
V_{bus}	535 V
L_1	3 mH
R_{L1}	50 mΩ
C_1	470 μF
R_{C1}	10 mΩ
R_1	-22.5 Ω
L_3	60 mH
R_2	11.25 Ω

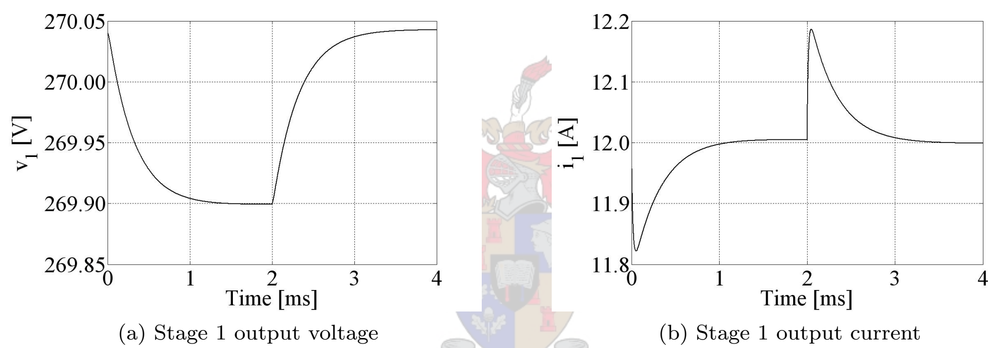
The voltage reference is followed with negligible error, even at the transient point at 0 ms and 375 ms. The current waveform shows small transients at these points, but of negligible magnitude.



(a) Stage 1 output voltage (b) Stage 1 output current

Figure 5.16: Stage 1 ramp response

Figure 5.17 shows the response of the system to a constant 270 V reference, but with the bus voltage stepped from 560 V to 485 V and back.



(a) Stage 1 output voltage (b) Stage 1 output current

Figure 5.17: Stage 1 step disturbance

As predicted by equation 5.39 on page 66, the step disturbance causes a output voltage disturbance similar in shape as the system step-response. The output current disturbance is derivative of the voltage waveform around the 12 A bias point. A rejection ratio of 47.64 dB is obtained

5.7.2 Stage 2

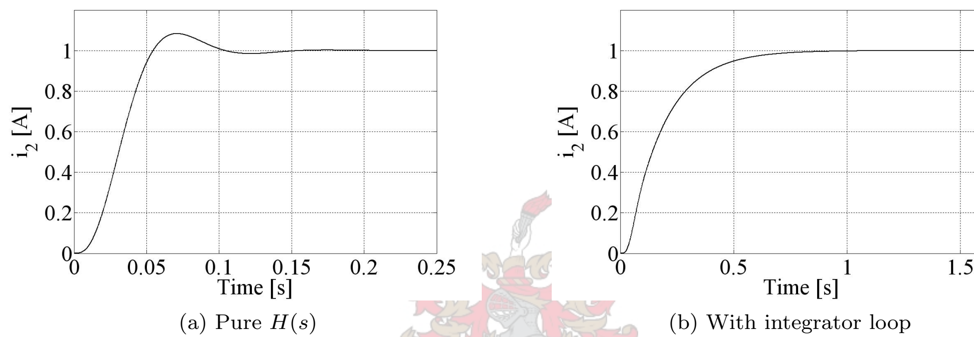
MATLAB was also used to simulate the second stage. The system is simulated using the same control gains, but with different load inductances. The control gains are calculated for a 4 H load inductor, as mentioned in section 5.5.2. The rest of the circuit parameters are listed in table 5-VII.

Figure 5.18a shows the step response of $H(s)$ (see figure 5.11), which is the closed-loop system without the integrator loop. Figure 5.18b shows the closed-loop system including

Table 5-VII: Stage 1 simulation circuit parameters

Parameter	**Value**
L_2	200 μH
R_{L2}	50 mΩ
C_2	470 μF
R_{C2}	10 mΩ
R_{Load}	14.4 Ω

the integrator loop. The results are exactly as expected – a fast, critically damped system without the integrator and a single-order response with the integrator loop present.



(a) Pure $H(s)$ (b) With integrator loop

Figure 5.18: Stage 2 simulation: 4 H

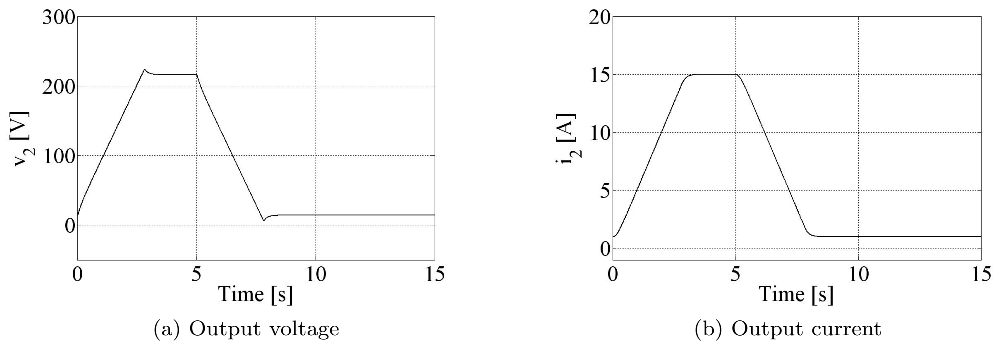
As designed in section 5.5.2, the pure $H(s)$ step response shown in figure 5.18a is a 3$^{\text{rd}}$ order Butterworth response with a bandwidth of 11 Hz. When the integrator is included, the response changes to the single order step response with a bandwidth of 1 Hz seen in figure 5.18b.

Figure 5.19 shows the result of a 1 to 15 A ramp-limited step response of the complete system. The response of a 15 A to 1 A ramp-limited step is also included on the same graphs.

Figures 5.20 and 5.21 show the results from repeating the above simulation with a 15 H load inductor. Due to the larger inductance, the pure $H(s)$ response is slower, as can be seen in figure 5.20a. It is interesting to note that the closed loop system with the integrator included still settles within 1 s.

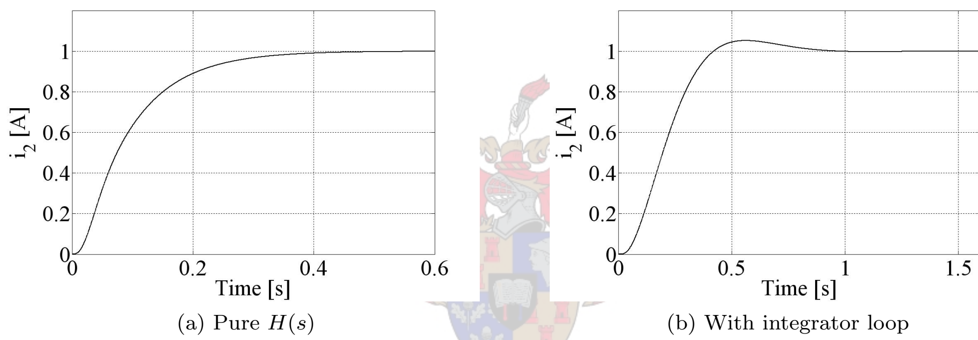
The voltage overshoot required to charge the load is much larger, as can be seen in figure 5.21a, but the resulting output current response remains the same.

Figures 5.22 and 5.23 show the results from using a 1 H load inductor. The resulting response is under damped, as can be seen in figure 5.22a. The reason for this is that the load responds quicker to an output voltage change than the control system anticipates. The



(a) Output voltage (b) Output current

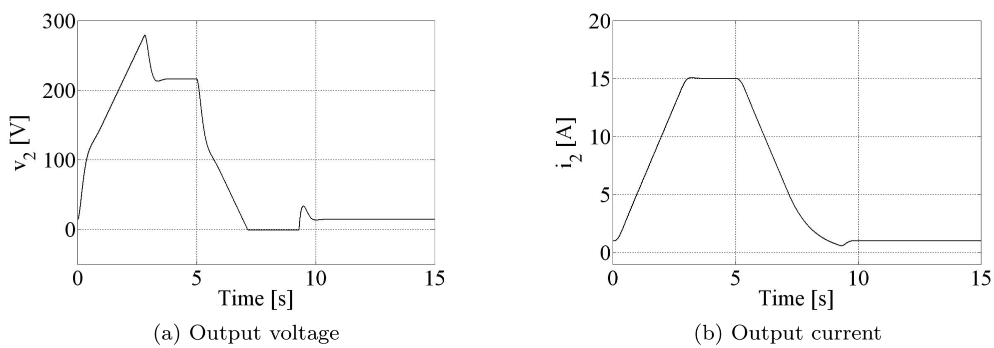
Figure 5.19: Stage 2 simulation: 4 H



(a) Pure $H(s)$ (b) With integrator loop

Figure 5.20: Stage 2 simulation: 15 H

Pectora roborant cultus recti

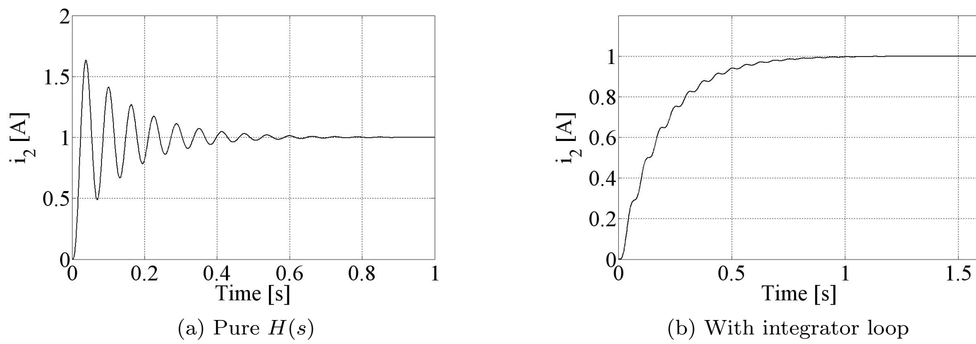


(a) Output voltage (b) Output current

Figure 5.21: Stage 2 simulation: 15 H

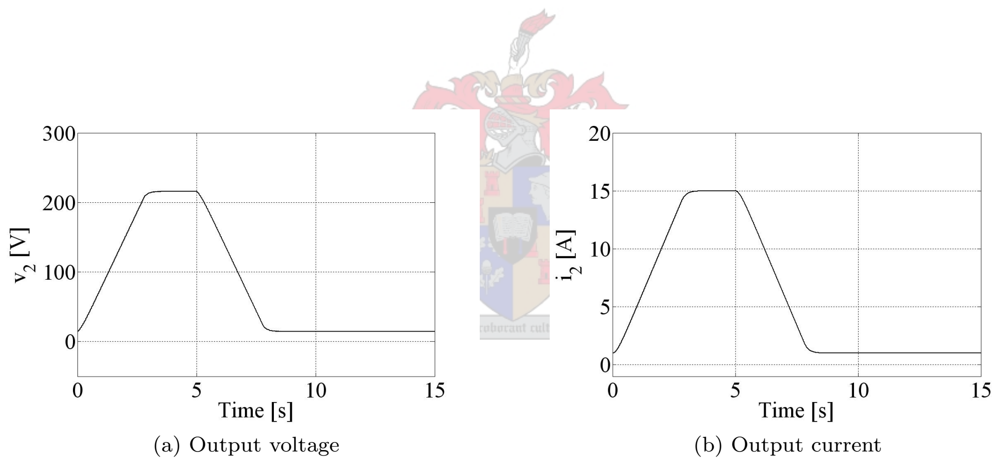
control system then overcompensates and the process continues. When the integrator is included the situation is better, although the oscillations are still present. The closed loop system with the integrator still settles within 1 s.

The response in figure 5.23 is also as expected. There is barely any voltage overshoot and the output current follows the reference very well.



(a) Pure $H(s)$ (b) With integrator loop

Figure 5.22: Stage 2 simulation: 1 H



(a) Output voltage (b) Output current

Figure 5.23: Stage 2 simulation: 1 H

From these results it is clear that the step-response of the system is heavily dependant on the load. Although the system remains stable throughout the range of possible load inductances, calibration is advised. Automatic calibration can be performed by controlling the output voltage and measuring the resulting current. This determines the resistance. The converter can then be switched off, resulting in an output voltage of zero due to the load discharging through the diode. The half-time can be measured and the inductance calculated. The resulting inductance-resistance combination can be used to calculate the control constants. To save space on the FPGA, this process can be performed via a host computer.

5.7.3 Combined

Using P-SPICE, from Orcad, a simulation was run on an ideal system. This is to verify that the control system still functions when the two stages are put together. Figure 5.24 shows the implementation of a converter (only switches). Voltage is multiplied forward and current backwards.

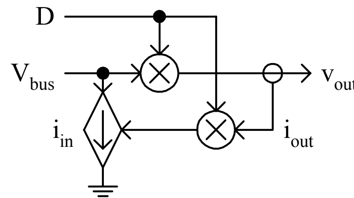
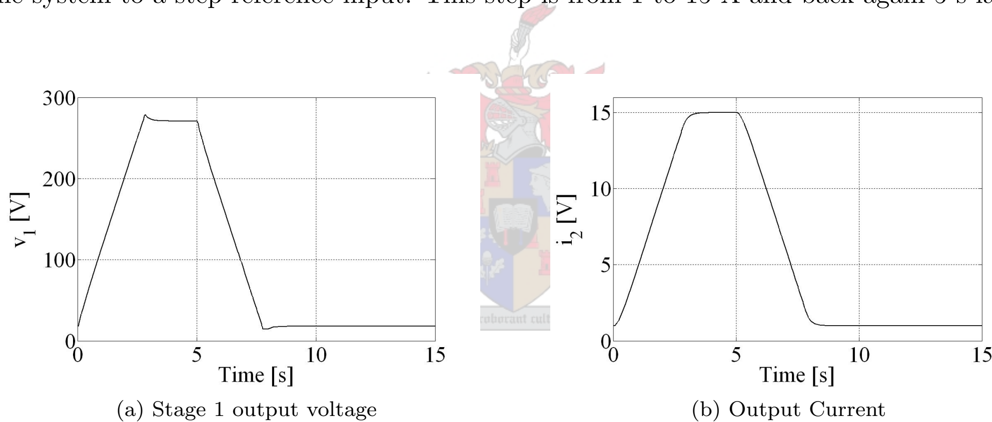


Figure 5.24: Converter in simulation

The series resistances of the passive components are taken into account, but digitisation and the non-ideal properties of the converters are not. Figures 5.25 show the response of the system to a step reference input. This step is from 1 to 15 A and back again 5 s later.



(a) Stage 1 output voltage

(b) Output Current

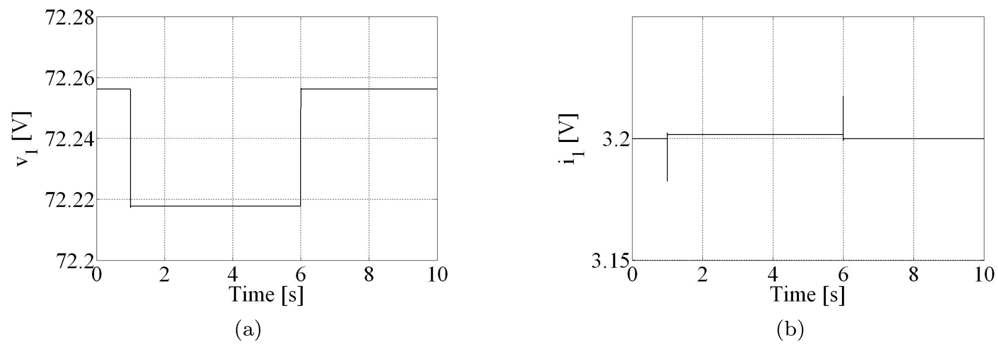
Figure 5.25: Combined simulation: Step

Figures 5.26 and 5.27 shows the response of the system to a step disturbance on the bus. The step is from 560 V to 485 V and back again 5 s later. The reference is kept constant on 4 A throughout the simulation. Figure 5.28 shows the spike at 6 s more clearly.

As with the simulation in figure 5.17, the voltage waveform is the same as for a step response and the current waveform is the differential of the voltage waveform. A rejection ratio of 47.67 dB is obtained, which is the same as when stage 1 was simulated on its own.

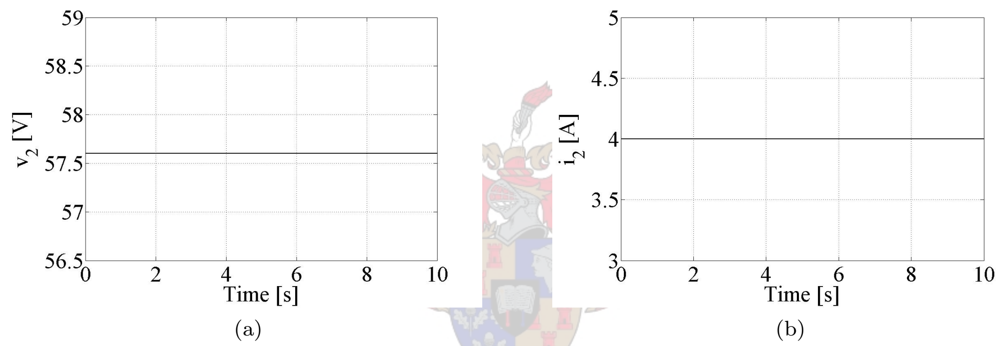
The second stage graphs are perfectly constant due to the ideal feed-forward control.

These simulations provide evidence that the control system will function properly in the practical system.



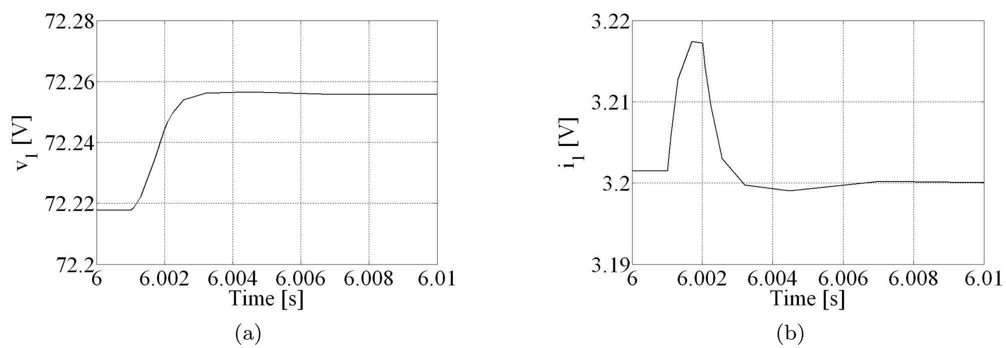
(a) (b)

Figure 5.26: Combined simulation: Disturbance



(a) (b)

Figure 5.27: Combined simulation: Disturbance



(a) (b)

Figure 5.28: Combined simulation: Disturbance

5.8 Soft-Start

5.8.1 Rectifier

Rectified 400 V 3-phase is never smaller than 490 V. For this reason, and considering the relatively large bus-capacitor, the system cannot simply be switched on without causing damage. The capacitor must be charged slowly before it can be connected directly to the mains. This can be done in many ways.

Three triacs are used to switch the mains on and off. It is possible to detect the zero-crossing between two of the three phases and switch them on at that point. The third is switched on later to complete the circuit. It is easy to show that the inrush current (ignoring line-inductance) is given by equation 5.56 where C is the capacitance of the bus-capacitor, V_{L-L} the peak line to line voltage and f the line-frequency (50 Hz). Problems with this circuit include the necessity to detect the zero-crossing as well as a very large peak inrush current.

$$i = \begin{cases} CV_{L-L}(2\pi f_L) \cdot \cos(2\pi f_L t) & , t < \frac{1}{4f_L} \\ 0 & , \text{otherwise} \end{cases} \tag{5.56}$$

A more attractive technique is to charge the capacitor through a resistor (R). This resistor has a switch in parallel with it, which closes after the bus is fully charged. A relay may be used for this purpose, as can a MOSFET. The MOSFET has no moving parts, making it the more attractive option as far as robustness and lifetime is concerned. A relay, however, does have near-zero losses and would be the preferred option for high-power converters. It is also possible to use a thyristor, but a MOSFET or relay is more controllable. The charging current and voltage are given by equations 5.57 and 5.58 respectively.

$$i = \frac{V_{L-L}}{R} e^{-t/RC} \tag{5.57}$$

$$v = V_{L-L} \left(1 - e^{-t/RC}\right) \tag{5.58}$$

This resistor-switch combination can be inserted either above or below the bus capacitor. It is advisable, however, to use the most negative node as a stable reference voltage. It is thus better to put this charging circuit above the bus-capacitor. This circuit is presented in figure 5.29.

First, a ‘power-good’ signal is received from the control board (which receives it directly from the ATX power supply). This signal triggers the triacs to switch on, after which the bus capacitors charge through the resistor. A timer is used to close the switch, connected in parallel to this resistor, after the capacitors are fully charged (after 3 s). When the ‘power-good’ signal is low, the triacs are switched off.

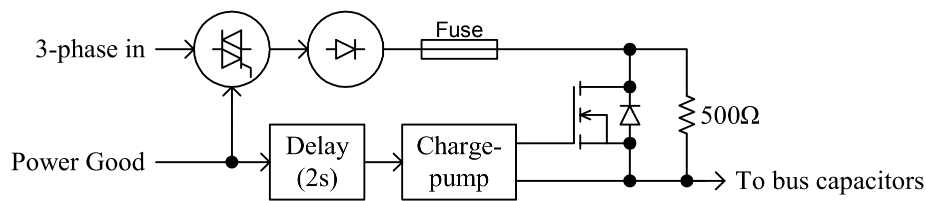


Figure 5.29: Bus charging circuit

5.8.2 Converter

Both the output filters and the transformer need to be started slowly. At start-up, the output filter of stage 2 can be assumed discharged (due to the large load). It is possible, however, to have a charged stage 1 filter. The energy stored in a 470 μF capacitor is very small. Stepping the second stage duty-cycle directly to 80% would only discharge this capacitor into the second-stage filter, where the energy would be rapidly absorbed by both the load and the inductor series resistance. A brief oscillation may be observed, but voltages remain positive (due to the load clamping-diode) and the oscillation dies away very rapidly. It is preferred, however, to slowly ramp this duty-cycle to 80%.

The first stage duty-cycle can also be slowly ramped to 13%, where the control system can then take over. This is equivalent to an output current of 4 A. The specification requires a settling time of 10 s after 'switch-on'. The DC bus soft-start takes 2 s. Once the controller takes over, the output will settle on 4 A within 1 s. The NMR probe will lock on the field-strength in less than 5 s. This leaves 2 s for the soft-start ramp, which is long enough to prevent oscillations. It is equivalent to a voltage ramp of 35 V·s^{-1}.

A second option does exist. On completion of the soft-start stage, the control system is initialised with the state of the system. It is possible, however, that the control system makes the first stage duty-cycle high too rapidly for the transformer to properly demagnetise. It should also be noted that for a short period of time the second stage filter inductor might operate in discontinuous inductor current, a state that is not included in the design of the control system. The first option is thus safer to use, even though it takes longer.

5.9 Firmware Implementation

5.9.1 Digital Representation

An obvious decision to be made is whether to use a floating point or fixed-point number system. Floating point has the advantage of high accuracy whereas fixed-point has the advantages of easy implementation and fast computation times. A fixed-point representation is implemented because of these latter two advantages.

The full-scale values of all the different nodes of the control system are carefully considered to ensure the highest possible accuracy as well as ease of implementation. The

EP2C8 FPGA contains only eighteen 18x18 bit multipliers. Implementing multipliers within logic-cells uses a significantly large number of logic-cells, something to be avoided.

There are only two processes that require larger than 18-bit resolution. These are the feed-forward control and the integrator. After smoothing, v_1 and i_2 both have a resolution of 24-bit, which is high enough to achieve the required accuracy.

Figures 5.30 to 5.34 show the properties of all the nodes. As far as possible the full-scale values have been chosen so that a simple shift operation can be used instead of a full multiplication operation. Care has been taken to ensure that any possible value will not fall outside the range of any of the operation blocks. Limiters are included so that the values remain practical. For example, the duty-cycle D_2 is limited from a full-scale of 32 to a full-scale of 1.

The reference to stage 1 is limited to a minimum of 15 V. The input to the reciprocal block is limited to a minimum of 10 V (allowing 5 V error on v_1). This is done to ensure adequate accuracy for the feed-forward control by restricting the full-scale value of the quotient.

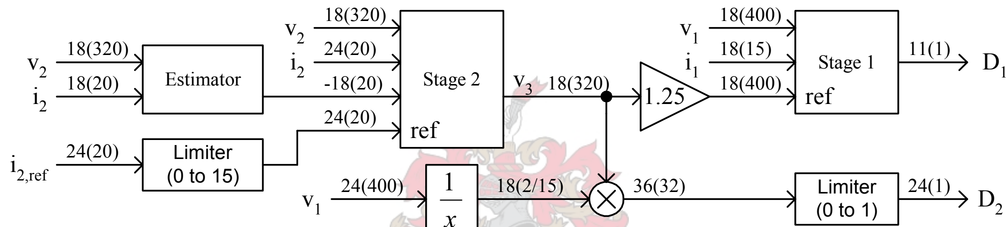


Figure 5.30: Signal representation: overall

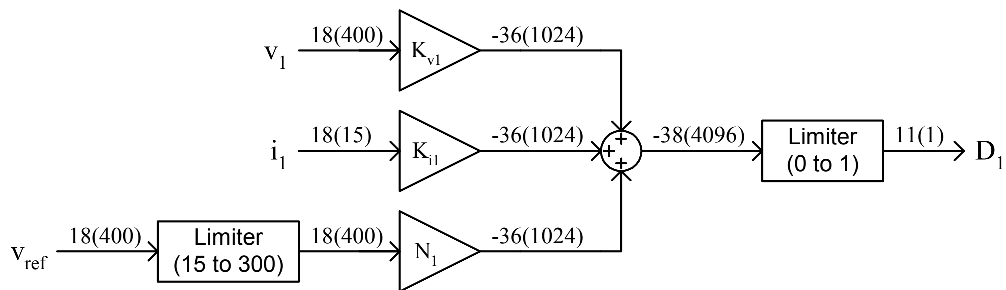


Figure 5.31: Signal representation: stage 1

The notation for digital representation is of the form $-n(f)$ were n is the number of bits and f the full-scale value. The '-' sign indicates that the number has an $(n+1)^{\text{th}}$ sign bit. As an example, '37(673)' is an unsigned, 37-bit number with a full-scale value of 673. Similarly, '-18(320)' implies a 19-bit signed number with a full-scale of 320. A

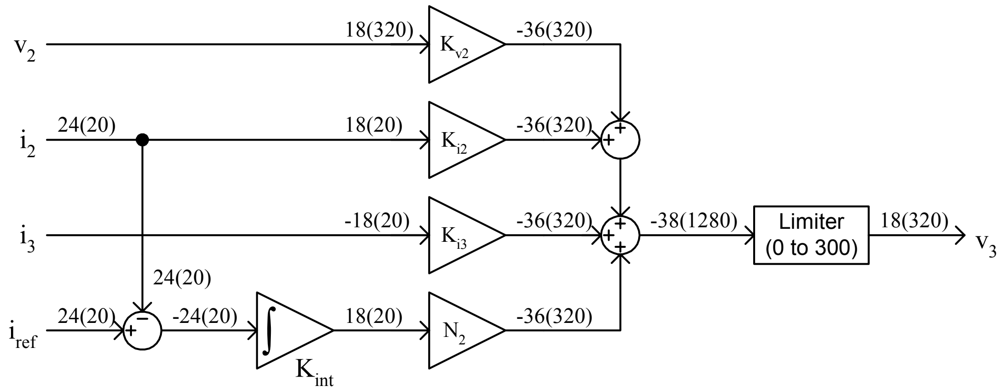


Figure 5.32: Signal representation: stage 2

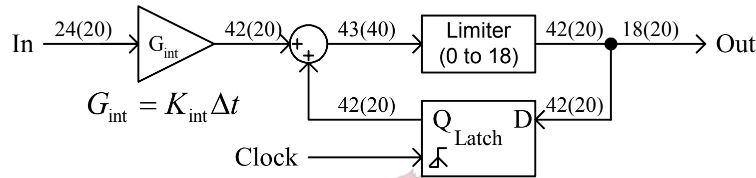


Figure 5.33: Signal representation: integrator

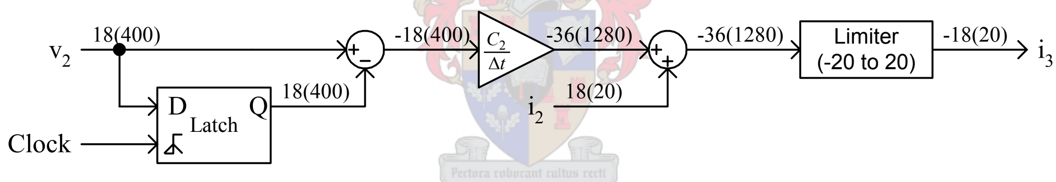


Figure 5.34: Signal representation: estimator

signed number is in two's complement form when it is an input to an adder. Inputs to a multiplier are represented differently – the most significant bit is the sign and the remaining bits represent the magnitude. Conversion between the two forms are performed between adder and multiplier blocks.

The duty-cycle of stage 2 (D_2) has been implemented as having a resolution of 24-bit. When using the delay-line, the effective duty-cycle is still only 18-bit. Using the 24-bit resolution is only possible with the noise-shaper (chapter 6).

The digital representation (d) of a real number (r) is provided by equation 5.59.

$$d = \frac{r \cdot 2^n}{f} \tag{5.59}$$

From this it is easy to deduce the digital representation of the constants. Figure 5.35 shows a gain block. The gain of 2 thus has a digital representation that is 4 bits wide and

has a full-scale value of 3.2. The process is confirmed by equation 5.60.

$$\frac{r_1 2^4}{10} \cdot \frac{r_2 2^4}{3.2} = \frac{r_3 2^8}{32} \tag{5.60}$$

From this the digital representation of the gain can be calculated, yielding '1010'. This process is repeated for every type of operation. All the non-specified gains are loaded by the CPU at start-up.

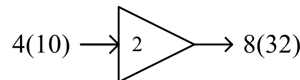


Figure 5.35: Gain block example

Due to the system being 18-bit and the integrator 24-bit, there will exist some oscillation on the output current. It remains to be measured practically. If the oscillation proves to be problematic, the controller must be upgraded to a higher resolution. That is possible if the controller is implemented by means of a finite state machine, rather than an asynchronous circuit. The advantage of a finite state machine is that the a few multipliers may be used to calculate the entire system. The system was implemented asynchronously for speed.

5.9.2 Time-Step

There are some time-dependant components, such as the integrator and estimator. The clock that is available has a period of 167.77216 μs (10 ps x 2^{24}). This is small enough to yield an accurate estimation and large enough not to sacrifice resolution. Also, it is much faster than any of the stage 2 time-constants. This time-step is thus used without further investigation.

5.9.3 Reciprocal

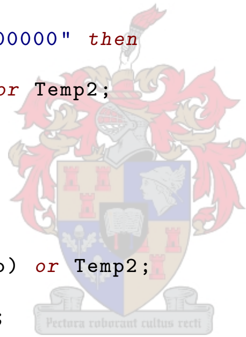
There are different ways to calculate the reciprocal of a number. First a digital circuit could be used, but this is very large and impractical to implement. Second a polynomial approximation, such as a least-square-error fit or a Taylor-series [27], could be implemented. For practical order polynomials this offers an unacceptably inaccurate answer.

Finally a binary search technique can be used to find the quotient iteratively. This takes $n + 1$ clock-cycles for an n-bit quotient, fast enough for the purposes of this project. The reciprocal block is clocked at 49 MHz. The extra clock-cycle is for latching the numerator and denominator at the start of the process.

The accuracy of this technique is the same as calculating the quotient directly. The algorithm is presented below in VHDL code [30]. The `Clk` signal is the 49 MHz clock and `ADC_Busy` is the signal used for synchronisation. The signal `x1` is the input and `x2` the output. Signal `greater` is high when

`Mul_A x Mul_B > "000001100110011001100110011001100110"`. The latter is the digital representation of the numerator (`1`). The VHDL-code for this technique is provided below.

```
process(Clk, nReset) is
begin
 if nReset = '0' then
  state <= "00";
 elsif falling_edge(Clk) then
  case state is

   when "00" =>
    if ADC_Busy = '1' then
     state <= "01";
    end if;

   when "01" =>
    if ADC_Busy = '0' then
     state <= "10";
    end if;

   when "10" =>
    Temp  <= "011111111111111111";
    Temp2 <= "010000000000000000";
    Mul_A <= "100000000000000000";
    Mul_B <= x1;
    state <= "11";

   when "11" =>
    if Temp2 = "000000000000000000" then
     if greater = '1' then
      x2 <= (Mul_A and Temp) or Temp2;
     else
      x2 <= Mul_A or Temp2;
     end if;
     state <= "00";
    else
     if greater = '1' then
      Mul_A <= (Mul_A and Temp) or Temp2;
     else
      Mul_A <= Mul_A or Temp2;
     end if;
    end if;
    Temp (16 downto 0) <= Temp (17 downto 1); Temp(17) <= '1';
    Temp2(16 downto 0) <= Temp2(17 downto 1);

   when others =>
  end case;
 end if;
end process;
```



It may be noted that there is a delay of one cycle between the ADC busy signal going low and the loading of data in state `10`. The smoother is clocked as soon as the ADC busy signal goes low (equivalent to state `01`, just before going to state `10`). The reciprocal process thus uses the most up-to-date sample possible, simply by waiting one clock cycle.

5.9.4 Smoother

The transfer function for a smoother is given by equation 5.61 [29], where N is the number of samples. This N is usually a power of 2 in order to simplify the process of dividing by this number.

$$H(z) = \frac{1 + z^{-1} + z^{-2} + \cdots + z^{-N+1}}{N} \tag{5.61}$$

By multiplying top and bottom by $(1 - z^{-1})$, equation 5.62 is obtained. The equivalent difference-equation is given in equation 5.63.

$$H(z) = \frac{1 - z^{-N}}{N\,(1 - z^{-1})} \tag{5.62}$$

$$y_n = y_{n-1} + \frac{x_n - x_{n-N}}{N} \tag{5.63}$$

The Cyclone II FPGA organises its on-board RAM in so-called 'M4K' blocks. These can be organised in a 256x18 configuration. In one such block, 256 18-bit samples can be stored. All four smoothers can be implemented in one such block. Table 5-VIII shows the number of samples for each variable, together with a brief explanation.

Table 5-VIII: Smoother: number of samples

Variable	**Number of samples**	**Explanation**
i_2	128	Must reject the noise as well as possible
i_1	8	Must reject the switching ripple, but still have high bandwidth
v_1	64	Must gain at least 6 bits for 24-bit resolution
v_2	32	32 is the lowest power of two left and v_2 does require smoothing

Pectora roborant cultus recti

5.9.5 Soft-Start

While the duty-cycles are ramped to their respective starting points, all the time-dependant components of the control system must be initialised according to the initial state of the circuit. The integrator is initialised to the same value as the reference. The integrator employs a multiplexer just after the limiter (see figure 5.33). The latch then loads the initialising (or reference) value at every clock edge rather than the output of the limiter (or integrator).

5.9.6 Timing

Timing is crucial to the success of the system. The most up-to-date sample must be used for the calculation of the next pulse width. The control system timing diagram is shown in figure 5.36. The times are rounded for clarity.

The smoothers are clocked using the same clock as the control system, namely 49 MHz. The PWM generator signals the ADC to sample just before the rising edge of the PWM

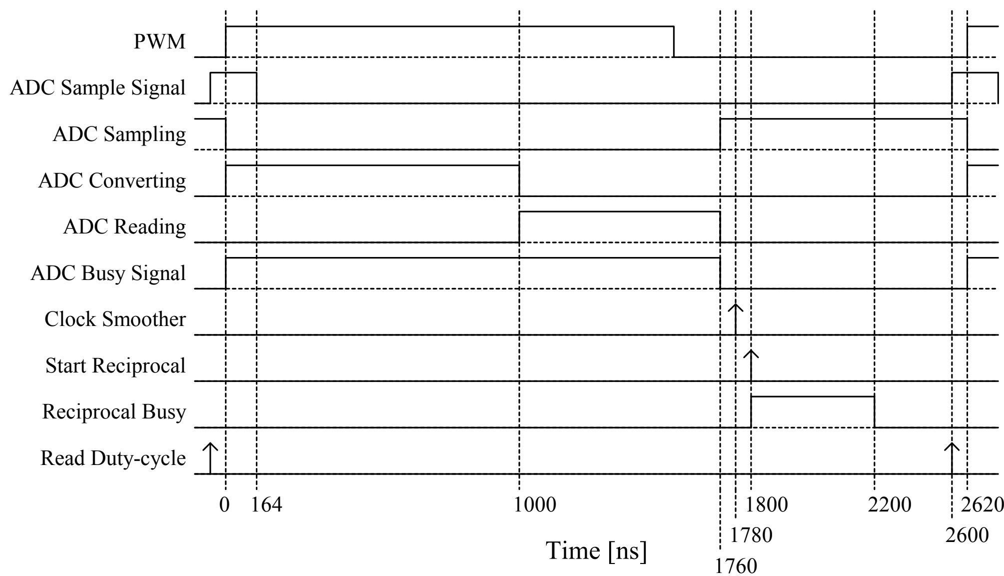


Figure 5.36: Controller timing diagram

signal. The MOSFETs take another 200 ns to switch, thereby making this the time which would result in the cleanest sample. The ADC then provides the control system with a 'busy' signal. Once the conversion is complete, the control system clocks the smoother. The reciprocal block follows on the next clock edge and the result ripples through the system. Since the PWM block contains a latch at its input, there is no need to latch the output of the control system.

5.9.7 Protection

Protection is implemented by switching the converter off upon any of the measurements reaching the full-scale value. The converter is also switched off when the current through the load or the output voltage of stage 1 is unexpectedly low. This protects the circuit when there is a fault in charging the bus-capacitors as well as when there is no load connected.

5.10 Summary

The control system was designed so that the two stages may be controlled separately. The first stage reference is 25% higher than the second stage desired input voltage (v_3). Voltage v_3 is asserted by means of feed-forward control.

Different control strategies have been considered. Since all the states are available, either by direct measurement or estimation, full state feedback was implemented. The control gains were calculated and the entire system verified by means of simulation.

The digital representation was provided and the implementation discussed. Fixed-point arithmetic was used. It might be possible to use smoothers and an 18-bit controller rather

than the proposed 24-bit controller and higher resolution ADCs. This remains to be verified experimentally.

Chapter 6

Digital PWM

6.1 Introduction

In chapter 5 the control system was designed. The PWM resolution necessary to produce satisfactory results is impractical to implement directly. Two techniques for solving this problem are discussed in this chapter; they are the use of a delay-line and the implementation of a noise-shaper. The firmware implementation of both alternatives is presented.

6.2 Resolution

6.2.1 Required Resolution

The output of the converter needs to be controllable to at least 5 ppm in order to realise 10 ppm stability. The equation for calculating the resolution (R) as a fraction of the output voltage, knowing the average duty-cycle (D) and bit-resolution (R_b), is given by equation 6.1. The second stage average duty-cycle is 80%, which implies that a bit-resolution of 18-bit would be required.

$$R = \frac{2^{-R_b}}{D} \tag{6.1}$$

The first stage does not need to be as accurate. The only requirement is that it attenuates the line-frequency ripple to some extent. Table 6-I gives the controllability with various different resolutions, assuming a bus voltage of 560 V.

The full-scale value of the ADC used to convert v_1 is 400 V, as explained in chapter 5. Its resolution is 18-bit, which translates to 1.526 mV. The leakage-inductance of the transformer, however, results in an uncertainty in duty-cycle. From chapter 4 it was calculated that the maximum uncertainty in duty-cycle at minimum load is 0.027% (not considering the effect of dead time). This is an uncertainty in output voltage of 153 mV, which corresponds to a PWM resolution of 12-bit. To make the system more deterministic, 11-bit PWM was implemented.

Table 6-I: Stage 1 PWM resolution

Resolution [bits]	Controllability [mV]
8	2 187.50
9	1 093.75
10	546.88
11	273.44
12	136.72
13	68.36
14	34.18
15	17.09
16	8.55
17	4.27
18	2.14
19	1.07
20	0.53
21	0.27

6.2.2 Practical Digital PWM

When a PWM waveform is generated purely by comparing a reference to a counter, the smallest step in pulse width is equal to the period of the clock that drives that counter. The fastest clock that can drive a 16-bit counter in an Altera Cyclone II FPGA is (according to the datasheet) 401.6 MHz. This drops to 157.15 MHz for a 64-bit counter.

As a rough estimate using linear interpolation (equation 6.2), an 18-bit counter would require a clock of 391.4 MHz. This would mean that 18-bit PWM can only be generated for a switching frequency of 1.493 kHz, which is unacceptably slow. Another method must be found.

$$f = \left(\frac{(401.6 - 157.15)}{(16 - 64)} (R_b - 16) + 401.6 \right) \text{MHz} \tag{6.2}$$

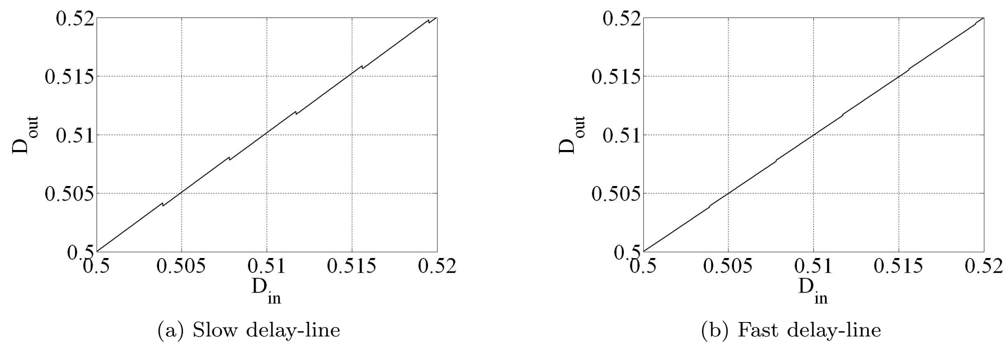
6.2.3 Delay-Line

By using a delay-line IC such as the MC10EP195 from ON Semiconductor, one can generate much higher resolution pulses. This IC can set the delay on a pulse from 2.20 ns to 12.23 ns in 10 ps steps by means of a 10-bit 'delay word'. This means that 18-bit PWM can be generated at a switching frequency of 381.470 kHz by using a counter clock of only 97.656 MHz. From this same counter, an 11-bit 47.684 kHz PWM signal can be generated for the first stage.

Another advantage of using a delay-line is that the effect of changing the duty-cycle is immediate. A noise-shaper (described in the following section) has a bandwidth limitation.

The 10 ps delay is not exact though, and does give some small error on the pulse-width. According to the datasheet, resolution is typically between 9.78 ps and 10.7 ps, depending on the temperature. The FPGA clock is only accurate to 50 ppm. The actual duty-cycle can

be plotted against the required duty-cycle. The resulting worst-case graphs are presented in figure 6.1.



(a) Slow delay-line (b) Fast delay-line

Figure 6.1: Duty-cycle error

There are three obvious solutions. The first is to ignore the error and let the control system correct the it. The duty-cycle is adjustable in small steps, even though these steps do not follow exactly as planned. They are still close enough to 10 ps for all practical purposes. It is also only 0.1% of the duty-cycle that falls on a 'glitch'.

Another more elegant solution is to utilise the fact that the FPGA uses a PLL to generate its 195.3125 MHz clock. The possible PLL configurations (using a 125 MHz external oscillator) that result in a clock frequency within 10% are given in table 6-II.

Table 6-II: FPGA PLL configurations

Divide	Multiply	Frequency [MHz]
12	17	177.083
16	23	179.688
9	13	180.556
20	29	181.250
15	22	183.333
2	3	187.500
15	23	191.667
20	31	193.750
9	14	194.444
16	25	195.312
12	19	197.917
5	8	200.000
8	13	203.125
6	10	208.333
16	27	210.938

Pectora roborant cultus recti

The final, most accurate solution is to use an external PLL with the divide and feedback counters implemented inside the FPGA. This will give as much clock-adjustment as is required. By using a far more stable 4 MHz crystal oscillator, a 10-bit divide counter and a 16-bit feedback counter, the 195 MHz clock can be adjusted to the nearest 3.906 kHz (102 fs).

The latter two options require calibration to match the delay-line at the normal operating temperature. This is a relatively complex process and thus the first option was chosen.

6.3 Noise-Shaper

6.3.1 Quantisation as a Noise Source

Rounding an n-bit signal to the nearest m-bit value adds a random error signal. The range of this error signal is given by equation 6.3, assuming unity full-scale [13].

$$e \in \left(-2^{-(m+1)}; 2^{-(m+1)}\right] \tag{6.3}$$

It is possible to reconstruct the original signal by changing the quantisation method. Say, for example, a constant signal of 5.2 is quantised. Rounding the signal to 5 is equivalent to adding a constant error of -0.2. If, however, the signal is rounded to 6 for 20% of the time and to 5 the other 80%, the average of the resulting signal is 5.2, equalling the original.

The duty-cycle is a signal that is constantly changing. The error signal is thus uncorrelated to both the PWM signal and the duty-cycle. According to the central limit theorem [28], the error signal is Gaussian in nature and has a flat frequency spectrum. A noise-shaper is a signal processing technique by which the error signal, or noise, is frequency-shaped. Lower frequency noise power is moved to higher frequencies.

This approach is of course provided that the duty-cycle signal is significantly larger than the quantisation increment. In the presence of line-frequency ripple, this would be the case, but at medium or minimum load the effect is not clear. Practical experimentation is required.

6.3.2 Block Diagram

Figure 6.2 shows the block diagram of a noise-shaper [13].

As described in the previous section, quantisation can be modelled as an error signal (e) being added to the input signal. Figure 6.3 shows the model, along with relevant signal names [13].

The respective z-transforms of x, e and e_o are X, E and E_o. Equations 6.4 and 6.5 show the relationship between H(z) and the so-called “noise transfer function” or NTF(z). It can be shown that the optimal implementation of the noise transfer function is given by equation 6.6 where N is the order of the noise-shaper [13].

$$X + E_0 = X - EH(z) + E \tag{6.4}$$

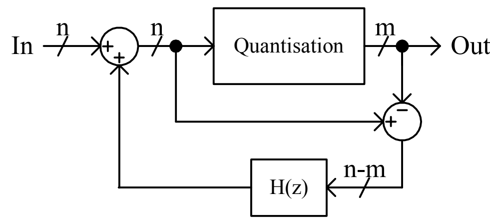


Figure 6.2: Noise-shaper block diagram

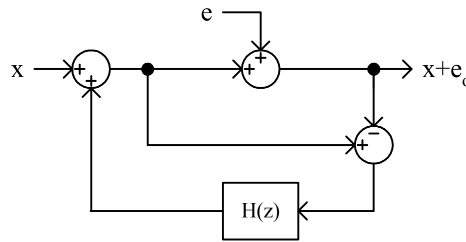


Figure 6.3: Noise-shaper block diagram

$$NTF(z) = \frac{E_0}{E} = 1 - H(z) \tag{6.5}$$

$$NTF(z) = \left(\frac{z-1}{z}\right)^N \tag{6.6}$$

The effect is thus that the noise added by quantisation is shaped by the noise transfer function before being added to the ideal output.

6.3.3 Theory of Operation

From the previous section it can seen that the signal passes through directly, but with added noise. By substituting equation 6.7 into equation 6.6, the frequency response of the noise transfer function is obtained [13]. This is shown in equation 6.8. The noise is thus shaped according to equation 6.9 where f is the frequency in question and f_s the switching frequency. The phase is given by equation 6.10.

$$z = e^{j2\pi \frac{f}{f_s}} \tag{6.7}$$

$$NTF(f) = \left[1 - \cos\left(\frac{2\pi f}{f_s}\right) + j\sin\left(\frac{2\pi f}{f_s}\right)\right]^N \tag{6.8}$$

$$|NTF(f)| = \left[2\sin\left(\frac{\pi f}{f_s}\right)\right]^N \tag{6.9}$$

$$\angle NTF(f) = N\left(\frac{\pi}{2} - \frac{\pi f}{f_s}\right) \tag{6.10}$$

A bode diagram is presented in figure 6.4 for the first 6 orders. At 1 kHz the 1st order curve is on top and the 6th order curve at the bottom. The phase is linear, as can be seen in equation 6.10.

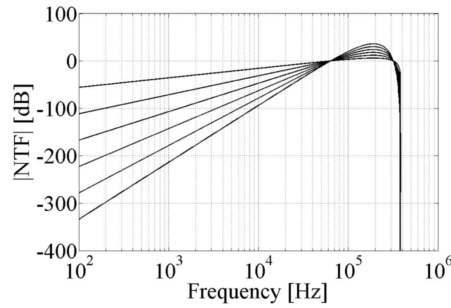


Figure 6.4: Noise shaper bode plot

From equation 6.9 it can be deduced that the transfer function has unity gain at $f = f_s/6$ and a maximum at $f = f_s/2$. It is required that the noise be rejected by 60 dB (10-bit) at a frequency of 10 kHz. The resulting signal thus has an equivalent resolution of greater than 18-bit for frequencies lower than 10 kHz. With a switching frequency of 381.470 kHz, this implies a 4th order noise-shaper. In order to implement 22-bit resolution at 10 kHz, however, an attenuation of 85 dB (14-bit) is required. This means a 6th order noise-shaper.

This analysis awakens the idea that the gate-drive signal can be generated directly by means of a noise-shaper. This means that the quantisation method is a level comparator and the output is either high or low. The noise, then, is not Gaussian anymore, but strongly correlated with the duty-cycle. The noise-shaper does not function as expected under this condition. The switching period is not deterministic, which is a disadvantage.

6.3.4 Implementation

The noise transfer function can easily be implemented as a string of differentiators. In order to implement H(z), the input value must be added to the output. A 4th order implementation is shown in figure 6.5. The z^{-1} blocks are implemented as D-Q latches. For higher-order noise-shapers, the chain can simply be lengthened. Another advantage of this implementation technique is that no multiplication is required.

Another disadvantage of digital logic is that the adders contain glitches while they are calculating the result. Implementing the block diagram directly as in figure 6.5 would result in a loop of adders, potentially ending in disaster. The loop has to be broken by means of a delay. Figure 6.6 shows the resulting block diagram. Each z^{-1} block is once again

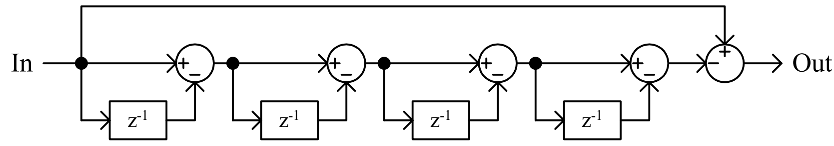


Figure 6.5: Noise-shaper $H(z)$ implementation

implemented as a D-Q latch. Rounding is performed by adding bit 9 to the 8-bit word obtained from bits 10 to 17 (bit 0 is the least significant and bit 17 the most).

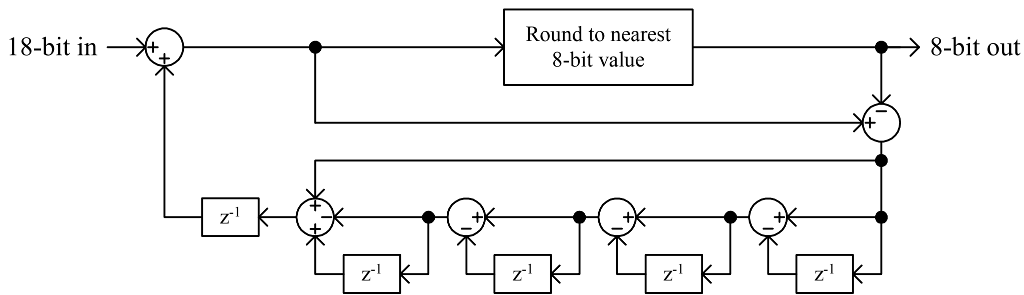


Figure 6.6: Noise-shaper implementation

From the noise transfer function it may be noted that the DC component of the noise is removed completely. It is easier to floor digitally than to round digitally. Also, the error is available immediately, i.e. the least significant bits. The 'round to nearest 8-bit value' block may thus be replaced with a 'truncate' block without a decrease in performance.

6.3.5 Performance

The system in figure 6.6 was implemented in MATLAB [31] (with flooring as apposed to rounding) and simulated with a 5 kHz, 1 $V_{p\text{-}p}$ sinusoidal input signal. Figure 6.7 shows the results.

It seems as though the noise-shaper worsens the signal. It must be remembered, though, that the noise-shaper attenuates the low-frequency noise by increasing the high-frequency noise. Figure 6.8 shows the same peak after passing all three signals through the same low-pass filter. This filter is equivalent to the second-order LC filter designed for stage 2 in the previous chapter.

From these graphs it is clear that the noise-shaper performs well. The error evident in the 8-bit signal is completely removed. It is also interesting to note that it adds negligible delay to the input signal.

Pectora roborant cultus recti

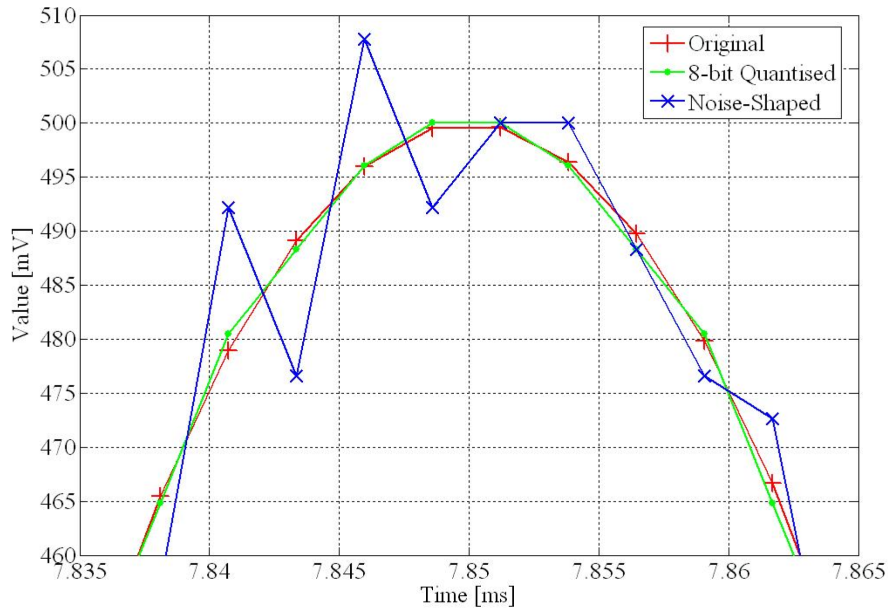
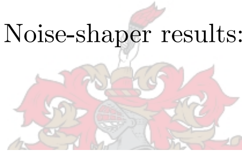


Figure 6.7: Noise-shaper results: no filtering



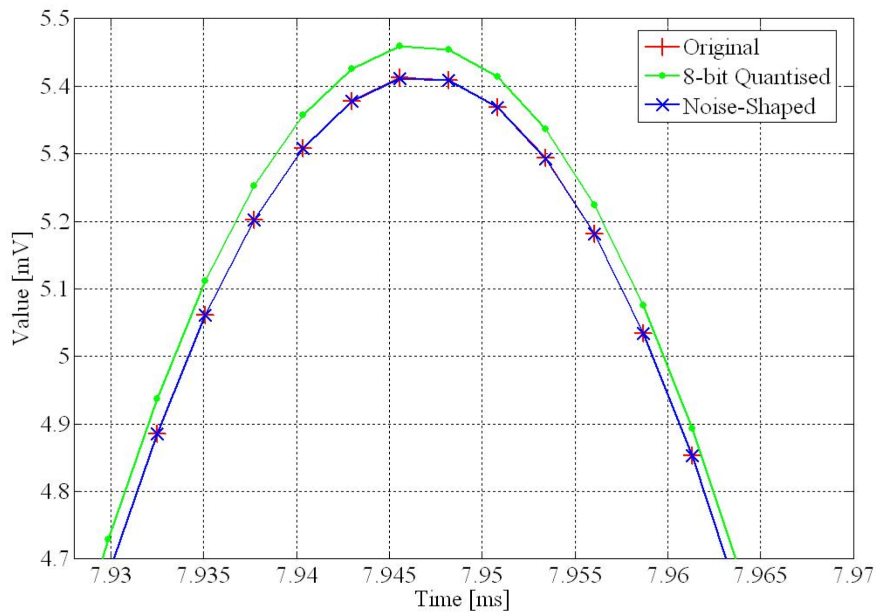


Figure 6.8: Noise-shaper results: after filtering

6.4 PWM Generator

6.4.1 Stage 1

Figure 6.9 shows the PWM generator block diagram for the first stage. The state-machine diagram is presented in figure 6.10.

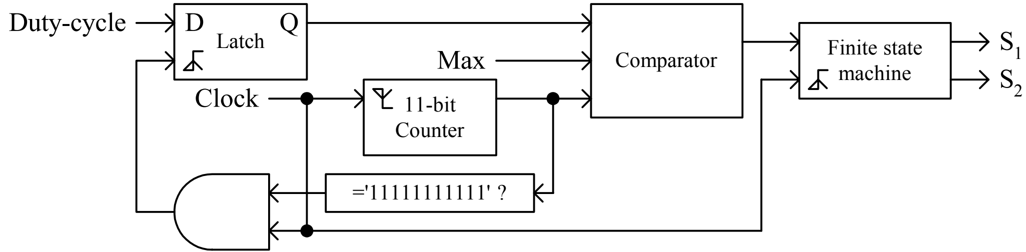


Figure 6.9: PWM generator: stage 1

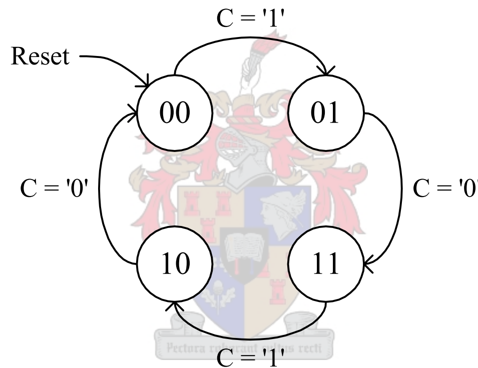


Figure 6.10: PWM state machine: stage 1

The duty-cycle, or pulse-width, is latched just before the rising edge of the PWM signal. This is done so that the pulse-width may be any value from 0 to 'Max' without creating problems. The comparator output goes high when both the duty-cycle and 'Max' are larger than the output of the counter. This is to ensure that the maximum duty-cycle is not exceeded.

It is also important to note that the counter changes on the falling edge of the clock and all the latches on the rising edge. This is to give the comparators time to settle. The clock frequency is 97.656 25 MHz.

The output of the state-machine is the same as its state (S_1 is the least significant bit and S_2 the most significant). Signal C is the output of the comparator.

6.4.2 Stage 2

Figure 6.11 shows the block diagram of the second stage PWM generator, including a delay-line. For the case where a noise-shaper is used instead, the control system incorporates the noise-shaper and the implementation below excludes the splitter and delay circuit.

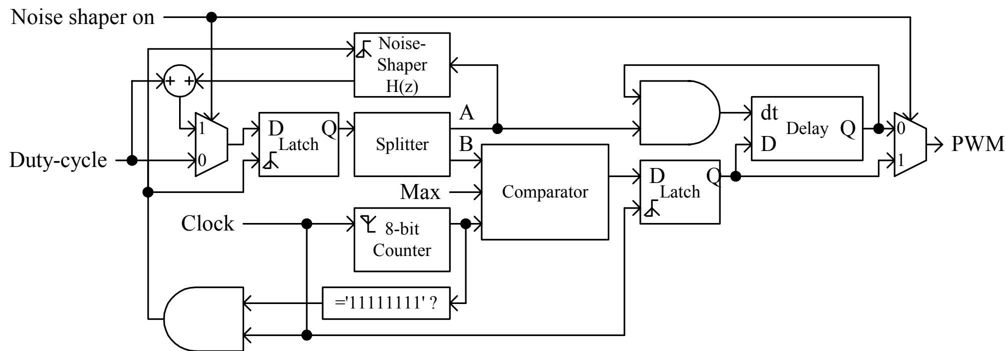
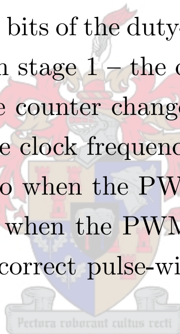


Figure 6.11: PWM generator: stage 2

Signal A is the least significant ten bits of the duty-cycle and signal B the most significant eight. The comparator is similar as in stage 1 – the only difference is that this one is 8-bit.

Also similar to the first stage, the counter changes on the falling edge of the clock and all the latches on the rising edge. The clock frequency is also 97.656 25 MHz.



The delay-line's delay word is zero when the PWM signal is low and equal to the least significant ten bits of the duty-cycle when the PWM signal is high. This is to delay only the falling edge, thus producing the correct pulse-width. The delay-line circuit is disabled when the noise shaper is active.

6.5 Summary

Although high-resolution PWM is difficult to implement digitally, at least two techniques exist which solve the problem. These are the delay-line and the noise-shaper. Uncertainties regarding the performance of both techniques may be clarified through experimentation. The delay-line does not provide the predicted duty-cycle exactly and the noise-shaper may not function properly with small signals. Both of the above techniques have been implemented with a signal allocated to switch between them.

Chapter 7

Probe

7.1 Introduction

In appendix C the physics behind NMR are described. It is strongly suggested that this be perused prior to reading this chapter in order to gain a better understanding of the processes involved.

This chapter proposes a new NMR magnetometer concept and confirms it through simulation. Much research remains to be done in this area.

Practical aspects are discussed and recommendations made, although a working prototype is not available. Measurements are included regarding the performance of the hardware. No NMR effects were tested practically.

7.2 Phase-Locked Loop NMR

7.2.1 Principle of Operation

As mentioned in appendix C, the steady state M_y is given by equation 7.1. The maximum value for M_y occurs when the denominator is equal to 2. At that point the excitation signal B_x is approximately 910 pT and M_y is $0.57 \cdot M_0$.

$$M_y = \frac{M_0 \gamma B_x T_2}{1 + \gamma^2 B_x^2 T_1 T_2} \tag{7.1}$$

When the detection- and excitation coil is wound at right-angles with each other, the coupling between them is small. It is then possible to have a continuous excitation signal and still detect the magnetisation vector.

A small error in excitation frequency will result in a rapid shift in phase between the excitation and detected signals. This phase can then be used to correct the frequency of excitation. The short-term immediate phase at time t_1 can be calculated using equation 7.2. Here ϕ_0 is the phase at time zero. A phase-locked loop may then be used to force the phase to zero. The resulting frequency of the oscillator resembles the field-strength.

$$\phi = \phi_0 + \int_0^{t_1} (\omega_{rf} - \omega_0)\, dt \tag{7.2}$$

If a direct digital synthesis IC is used to generate the excitation signal, the frequency is available immediately. This means that no frequency counter or FFT is required. The measurement repetition rate can thus be very large (in the order of 100 kHz).

The phase measurement is not absolutely accurate and a small phase error may arise. The steady state phase is given by equation 7.3. An error in phase of 5° would imply an error in frequency of 3 mHz (<1 part in 10^9). A phase error is thus no problem.

$$\phi = \arctan\left(\frac{1}{T_2\,(\omega_{rf} - \omega_0)}\right) - \frac{\pi}{2} \tag{7.3}$$

7.2.2 Phase-Locked Loop

Figure 7.1 shows a potential control system block diagram derived from equation 7.2. This resembles a phase-locked loop.

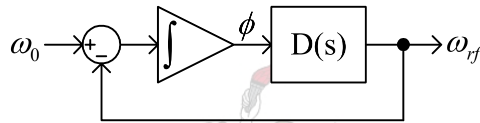


Figure 7.1: Probe phase-locked loop

The frequency ω_{rf} is generated by means of direct digital synthesis (DDS). The advantage of this is that the frequency is immediately available. No time-consuming strategy needs to be undertaken in order to determine the frequency.

In the practical system the phase will be calculated digitally. Subtraction and integration is inherent in the system. The only concern is thus the control block $D(s)$. By inserting another pole at the origin ensures that the steady state phase is always zero. It also results in a controller that follows a ramped ω_0 with zero error.

It is possible for the phase to be measured at a rate of 100 ksps (see section 7.3.5). This implies that no dynamics in the system should be faster than 10 kHz. After inserting a zero at 10 kHz, a root-locus design technique was used to derive equation 7.4. The bandwidth of the closed-loop controller was also chosen as 10 kHz. The resulting damping factor of the closed-loop system is 0.5, which is acceptable.

$$D(s) = \frac{63\ 025(s + 62\ 832)}{s} \tag{7.4}$$

By using a time-step of 10 μs (Δt), the z-transform equivalent is obtained. This is given by equation 7.5 and simplified to obtain equation 7.6. The extra z^{-1} in the numerator ensures causality.

$$D(z) = \frac{63\ 025z^{-1}\left[\left(1 - z^{-1}\right) + 62\ 832\Delta t\right]}{1 - z^{-1}} \tag{7.5}$$

$$D(z) = \frac{63\ 025z^{-1}\left(1.628 - z^{-1}\right)}{1 - z^{-1}} \tag{7.6}$$

The closed-loop step responses of both discrete and continuous time implementations are shown in figure 7.2.

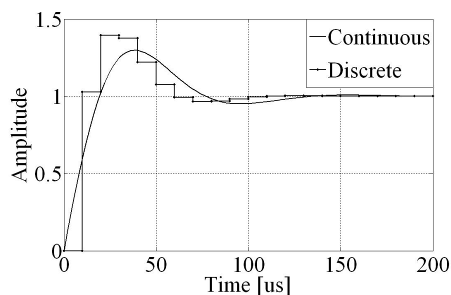


Figure 7.2: Probe PLL step-response

7.2.3 Sustaining Resonance

Using continuous-wave NMR to sustain resonance (as described in section 7.2.1) only works well in theory. Practically the exact flux density is difficult to predict and maintain. A better option is to use repetitive pulses. Using a pulsed excitation signal has many advantages. First, a cleaner detection signal is possible while the pulse is off. Also, resonance can be obtained quicker.

Providing a constant pulse-width every time the y-component of the magnetisation vector (M_y) drops below a specified limit ensures that a large enough NMR signal is always available. Figure 7.3 shows the locus of the magnetisation vector in the y-z plane. In this case resonance is attained and sustained by providing a $\pi/40$ pulse (29 ms) whenever M_y is less than half M_0. The pulses are 10 nT in magnitude and the simulation is run for 100 s. The external field is 1 T. The expected steady state condition is maintained successfully.

As the system starts with the $\mathbf{M}$ vector in the z-direction, the pulses are applied in rapid succession until M_y is above $0.5M_0$. The pulse then switches off until relaxation cases M_y to fall below $0.5M_0$. The pulse is then applied once, which is enough to push M_y over the $0.5M_0$ mark. This process then continues.

In this arrangement, it is advantageous to have larger T_1 and T_2. Larger time-constants implies that the $\mathbf{M}$ vector takes longer to relax across the threshold. This in turn implies that fewer pulses are necessary in order to sustain resonance. This is a great advantage

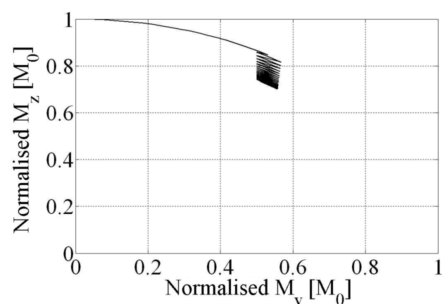


Figure 7.3: Locus for sustaining resonance

because pure water can then be used as a sample. The addition of impurities is no longer required and crystallisation of the sample is no longer a problem.

Another advantage of this technique over traditional pulsed NMR is that neither the pulse magnitude nor the pulse length is very important, as long as the resulting shift in angle is in the order of 10° or smaller. Care should be taken, however, to avoid setting the limit too large. Figure 7.4 shows the result after using a limit that is $0.6{\cdot}M_0$. As to be expected, the use of a limit larger than the theoretical maximum results in an unstable system.

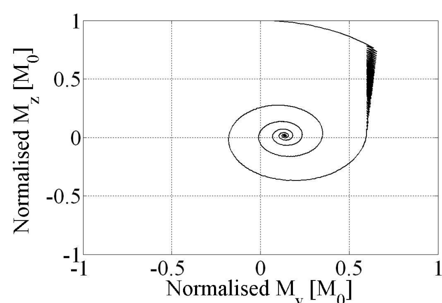


Figure 7.4: Locus for a too large limit

In this case the arc followed by **M** when the pulse is applied causes M_z to become smaller. The straight line followed by **M** when relaxation is in effect does not increase M_z enough to compensate. The result is that M_x decreases until the pulse cannot increase M_y anymore. The locus then spirals to a steady-state near the origin while the pulse is continuously applied.

Figure 7.5 shows the locus resulting from setting the limit to half M_0, but using an excitation frequency that is too high.

The excitation frequency that is too high causes the **M** vector to rotate in the x-y plane as well. The result is an M_y that becomes smaller each time the pulse is applied, rather than larger. A steady-state near the z-axes is achieved.

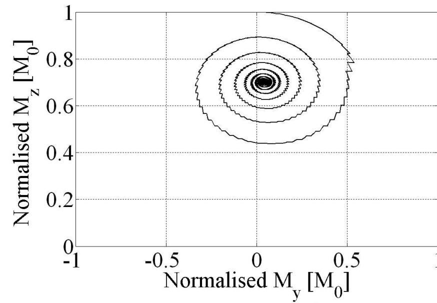
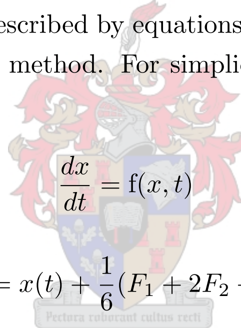


Figure 7.5: Locus using the wrong RF frequency

The practical system should correct the frequency almost immediately upon attaining resonance. The latter case should thus not pose a problem.

7.2.4 Simulation

Simulation was run using the above-mentioned digital phase-locked loop. The Bloch equations were calculated using a fourth-order numerical integration technique called "the Runge-Kutta method" [27]. It is described by equations 7.7 to 7.12. The simulations shown in figures 7.3 to 7.5 also used this method. For simplicity the continuous-wave excitation technique was used.



$$\frac{dx}{dt} = \mathrm{f}(x, t) \tag{7.7}$$

$$x(t + \Delta t) = x(t) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4) \tag{7.8}$$

$$F_1 = \Delta t \mathrm{f}\,(x, t) \tag{7.9}$$

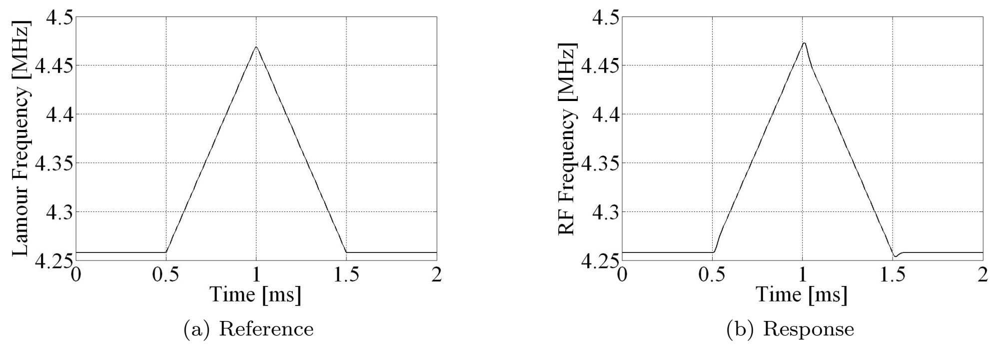
$$F_2 = \Delta t \mathrm{f}\left(x + \frac{F_1}{2}, t + \frac{\Delta t}{2}\right) \tag{7.10}$$

$$F_3 = \Delta t \mathrm{f}\left(x + \frac{F_2}{2}, t + \frac{\Delta t}{2}\right) \tag{7.11}$$

$$F_4 = \Delta t \mathrm{f}\,(x + F_3, t + \Delta t) \tag{7.12}$$

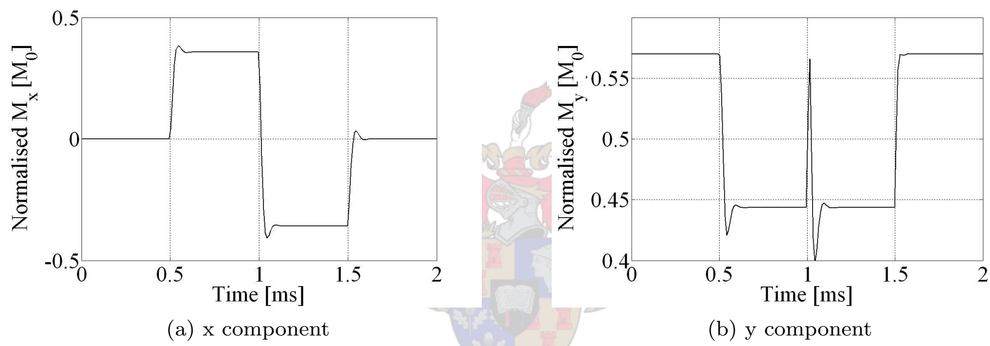
The first simulation was run using a ramped reference with a gradient of 10 T·s^{-1}. The reference and the corresponding response are shown in figure 7.6. Figures 7.7 and 7.8 show the resulting magnetisation vector components and phase.

The second simulation was for a ramped reference of 500 mT per second. This is the worst-case scenario when using the magnet power supply designed in the previous chapters.



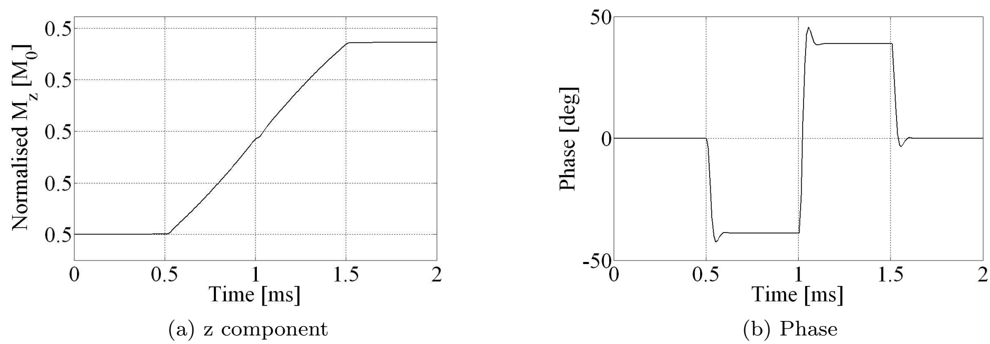
(a) Reference (b) Response

Figure 7.6: Probe ramp response



(a) x component (b) y component

Figure 7.7: Probe ramp response



(a) z component (b) Phase

Figure 7.8: Probe ramp response

The reference is ramped from 0.1 T to 1.2 T and back. This same reference is used in all further simulations. The results are shown in figures 7.9 to 7.12.

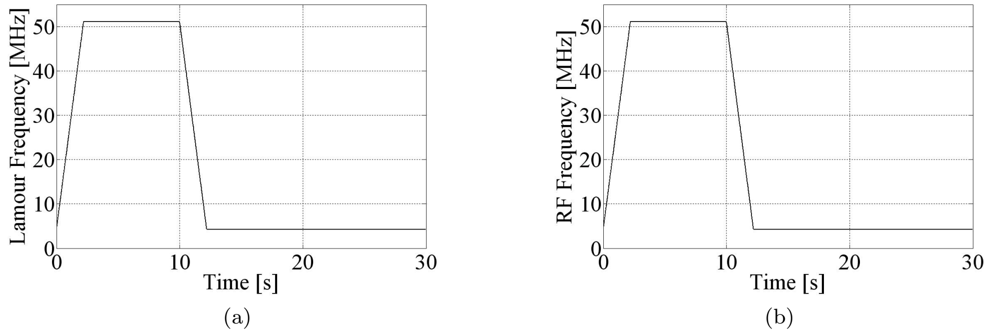


Figure 7.9: Probe long-term response: Reference and Response

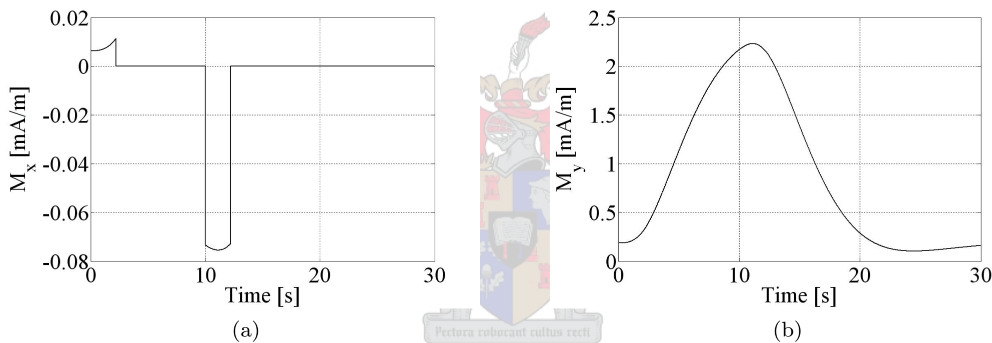
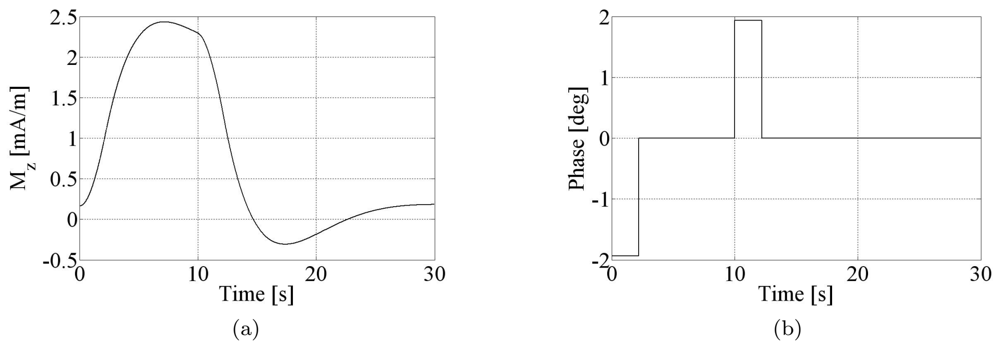


Figure 7.10: Probe long-term response: M_x and M_y

These simulation results show that the concept of phase-locked loop NMR does provide a very accurate measurement of the flux density. The ramp reference is followed with zero error (due to the extra integrator) and the system is stable.

Also, the field may be sampled at 100 ksps and can follow a 10 T per second ramp. There is room for further research in the area of the phase-locked loop itself. It might be possible to follow much faster ramp references by optimising $D(z)$.

These simulation results also suggest that care should be taken when sustaining resonance by means of pulses, especially in the case of a downward ramp. At 20 s (figures 7.10 and 7.11) M_y and M_z are 0.25 and -0.25 respectively. Any pulse would thus reduce M_y, as opposed to increasing it.



(a) (b)

Figure 7.11: Probe long-term response: M_z and Phase

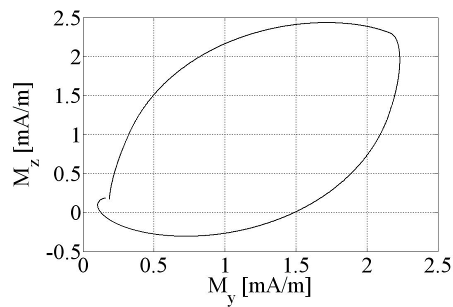


Figure 7.12: Probe long-term response: Continuous-wave locus

The same simulation was run using 10 μs, 10 μT pulses whenever M_y is below 164 μA·m^{-1}. Each pulse rotates the magnetisation vector through 1.5°. The resulting $\mathbf{M}_{y-z}$ locus is shown in figure 7.13. The problem is thus solved by using this pulsed technique.

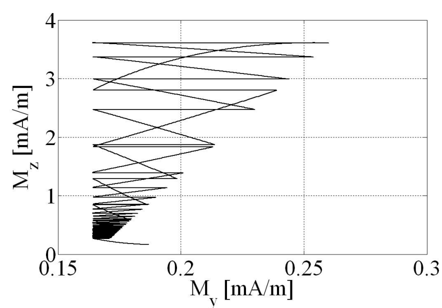


Figure 7.13: Probe long-term response: Pulsed locus

As the frequency increases, so does M_0. The result is that the locus rises upwards. The pulse length is kept constant, resulting in the same angle of rotation of $\mathbf{M}$ for each pulse,

resulting in the larger 'head' observed in figure 7.13. As the frequency decreases, the reverse path is followed.

It is also possible to sustain resonance by providing a pulse every time the induced voltage (equation 7.18) drops below some threshold (as apposed to measuring the magnitude of M_y). The advantage of this is that the receiver circuit can be designed for a very narrow range of voltages. Also, analogue comparators can be used instead of ADCs, since the only important information is the zero-crossing (explained in section 7.3.5). Resonance can be maintained by means of an analogue circuit, i.e. an envelope detector, comparator, one-shot and analogue switch. The signal for the analogue switch must also be sent to the FPGA, since no phase measurement should be taken while the excitation pulse is present. The fast and expensive ADC is then not necessary. The pin-count of the FPGA also drops. Figure 7.14 shows the resulting locus.

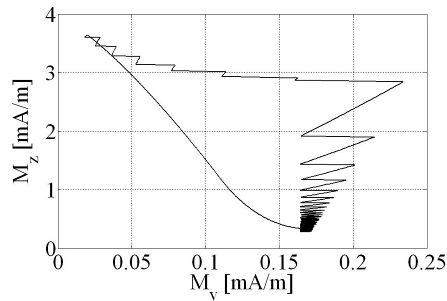


Figure 7.14: Probe long-term response: Modified pulsed locus

The long loop rising to the upper left is due to M_0 increasing and therefore also the **M** vector. There are no pulses because the frequency rises faster (and therefore the reference for M_y drops faster) than M_y. The same is not true for the return path. It this case, the reference for M_y rises, causing more frequent pulses in order to keep up. The seemingly flat reference observed in locus to the right is due to a constant frequency. The M_z component is on its way to a steady state at the bottom right.

During this simulation the fact that B_x is smaller at larger frequencies has been taken into account, resulting in the smaller changes in the top left-hand region.

7.3 Hardware Detail

7.3.1 Modified Bloch Equations

The flux density within the coil is caused partly by the magnetisation vector and partly by the magnetic field induced by the coil. By substituting equation 7.13 [5] into the Bloch equations (equation C.16 on page 176), equation 7.14 is obtained. It is now possible

to distinguish between the flux density caused by the excitation signal ($\mu_0\mathbf{H}$) and the magnetisation vector ($\mu_0\mathbf{M}$).

$$B = \mu_0(H + M) \tag{7.13}$$

$$\dot{\mathbf{M}} = \gamma\mu_0(\mathbf{M} \times \mathbf{H}) - \begin{bmatrix} \frac{M_x}{T_2} \\ \frac{M_y}{T_2} \\ \frac{M_z - M_0}{T_1} \end{bmatrix} \tag{7.14}$$

7.3.2 Excitation Coil

It is required that the excitation signal be as large as possible. The buffer that drives the coil has a maximum voltage of $2V_{p-p}$. The maximum peak voltage over the coil is thus 1 V. The required flux density is in the range of 10 μT (B_x at 5 MHz). Equation 7.17 provides the number of turns. The factor of two arises from the fact that only half of the actual induced flux density has an effect on the sample.

$$N = \frac{V}{2j\omega AB_x} \tag{7.15}$$

The number of turns multiplied by the cross-sectional area must thus be smaller than $1.6{\cdot}10^{-3}$ m^2. Another consideration is the current in the coil. This is given by equation 7.16. Substituting equation 7.15 yields equation 7.17.

$$I = \frac{VI}{j\omega N^2 A\mu_0} \tag{7.16}$$

$$lA = \frac{VI\mu_0}{4\omega B_x^2} \tag{7.17}$$

The AD8062 operational amplifier from Analog Devices provides a bandwidth of 300 MHz and a linear output-current of ± 25 mA. The volume of the coil must thus be less than 2500 mm^3.

7.3.3 Detection Coil

The detected signal voltage is given by equation 7.18. This requires that the NA constant should be maximised. It is thus chosen to be 1 600 mm^2. As seen in figure 7.13, M_y is always larger than 164 μA·m^{-1}. The resulting voltage is in the range of 10 μV to 100 μV, depending on the frequency.

$$V = j\omega NA\mu_0 M_{x-y} \tag{7.18}$$

For good mixing, a signal-strength of between 1 mV and 10 mV is required. This implies that a constant gain of 40 dB would be sufficient. RF buffers readily provide a gain of 20 dB. Two of those would thus be enough, one on either side of the cable.

Pectora roborant cultus recti

7.3.4 Using a Single Coil

The two previous sections suggest that the use of a single coil is possible. This has many advantages. The first of which is that both the excitation signal and detection signal is taken from the same point. A far more accurate calibration is then possible in order to compensate for cable-length and circuit delays.

To make the winding of the coil easier, the length must be maximized and the number of turns minimised. The constants calculated in the previous two sections imply that there is 1.5 mm per turn, whichever dimensions are chosen. The equations used to calculate these constants are only valid for a long coil, however. Inside the probe casing there is space for a 30 mm long coil, which is the length chosen. The resulting area is 80 mm^2 (10 mm diameter) and the coil must have 20 turns. It is turned using 0.7 mm diameter copper wire, resulting in a series resistance of 30 mΩ, which is acceptable. The capacitance between windings are minimised by evenly spacing each winding 0.7 mm apart. The resulting total capacitance is about 20 fF. The inductance of the coil is 1.3 μH.

During the detection stage, however, the coil must be seen as an open circuit. The excitation signal is large (1 V peak). The AD8062 operational amplifier has an output swing of 0.3 V to 4.75 V. This implies a peak signal voltage of 2.2 V.

The voltage drop over a diode is typically 0.8 V at 25 mA. Also, the current through a diode is typically 10 pA for the expected 100 μV signal. The equivalent resistances are 30 Ω and 10 MΩ respectively. When inserting a diode in series between the driver and the coil, there is effectively a short circuit from the driver to the coil, but an open circuit from the coil to the driver.

7.3.5 Measurement Technique

Different methods are available to measure the magnitude and phase of the NMR signal. The first is to sample the signals directly at the RF frequency. This does not allow for a very accurate phase-measurement. Also, high-speed ADCs are expensive.

The next option would be to mix the detection signal with the excitation signal. The result is given by equations 7.19 [1] and 7.20 [27], where y is the result.

$$y = A_{rf} \cos(\omega_{rf} t) \cdot A_0 \cos(\omega_0 t + \phi) \tag{7.19}$$

$$y = \frac{A_{rf} A_0}{2} \cos(\phi) + \frac{A_{rf} A_0}{2} \cos\Big((\omega_{rf} + \omega_0)t + \phi\Big) \tag{7.20}$$

The high-frequency component can be filtered out and the only thing remaining is the cosine of the phase. The amplitude of the NMR signal can be determined from its envelope. One problem is that the phase cannot be measured over the full 360°. This is important because of the phase shift in the cables and buffers. Although phase at the probe is approximately zero, the phase at the mixers can be any value.

In order to solve this problem, a quadrature mixer may be implemented. The resulting

equations are 7.21 [1] and 7.22 [27].

$$y = \begin{cases} A_{rf}\cos(\omega_{rf}t)\cdot A_0\cos(\omega_0 t+\phi) \\ A_{rf}\sin(\omega_{rf}t)\cdot A_0\sin(\omega_0 t+\phi) \end{cases} \tag{7.21}$$

$$y = \begin{cases} \frac{A_{rf}A_0}{2}\cos(\phi) + \frac{A_{rf}A_0}{2}\cos\Big((\omega_{rf}+\omega_0)t+\phi\Big) \\ \frac{A_{rf}A_0}{2}\sin(-\phi) + \frac{A_{rf}A_0}{2}\sin\Big((\omega_{rf}+\omega_0)t+\phi\Big) \end{cases} \tag{7.22}$$

The signal that is 90° out of phase can easily be generated using a phase-locked loop. Also, many ICs are available that offer quadrature demodulation. The two mixed signals are sampled separately and the phase can be determined over the full 360°. The amplitude measurement is very noisy, however. The calculation of the phase is complicated to implement within an FPGA.

The final option is to mix both the excitation signal and the detection signal with a third, slightly slower, 'mixing' signal [32]. The resulting equations are 7.23 [1] and 7.24 [27].

$$y = \begin{cases} A_{mix}\cos(\omega_{mix}t)\cdot A_0\cos(\omega_0 t+\phi_0) \\ A_{mix}\cos(\omega_{mix}t)\cdot A_{rf}\cos(\omega_{rf}t+\phi_{rf}) \end{cases} \tag{7.23}$$

$$y = \begin{cases} \frac{A_{mix}A_0}{2}\cos\Big((\omega_0-\omega_{mix})t+\phi_0\Big) + \frac{A_{mix}A_0}{2}\cos\Big((\omega_0+\omega_{mix})t+\phi_0\Big) \\ \frac{A_{mix}A_{rf}}{2}\cos\Big((\omega_{rf}-\omega_{mix})t+\phi_{rf}\Big) + \frac{A_{mix}A_{rf}}{2}\cos\Big((\omega_{rf}+\omega_{mix})t+\phi_{rf}\Big) \end{cases} \tag{7.24}$$

When ω_{mix} is 100 kHz below ω_{rf}, the result is two 100 kHz signals with the same phase relationship as the original excitation and detection signals. The only disadvantage is that a second DDS is required, but that is outweighed by the accuracy of measurement now possible. The probe circuit diagram is presented in figure 7.15.

Pectora roborant cultus recti

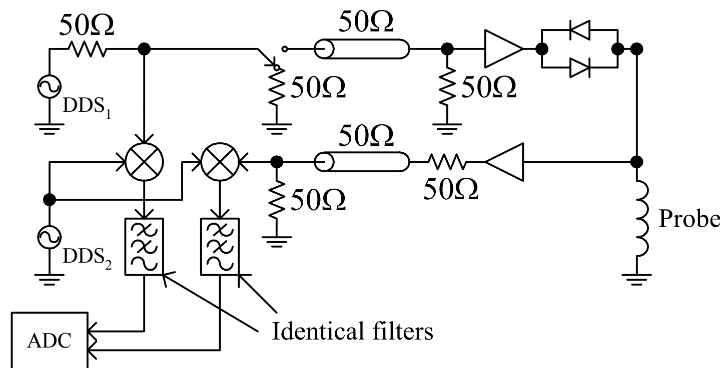


Figure 7.15: Probe circuit diagram

The two mixers are implemented with identical Gilbert cell mixers [1] (MC1496). This is a balanced mixer designed for relatively low frequency (<300 MHz) applications. The two

filters are identical 2nd order Butterworth active filters with a cut-off frequency at 150 kHz.

The phase is determined from counting the distance between zero-crossings. Care must be taken, however, in order to ensure that the noise does not cause glitches. A Schmidt-trigger would solve the problem, but for accurate phase measurement the amplitude of the two signals must then be identical. This may be achieved by means of a digital normalising algorithm, which would imply that the ADC is required.

The amplitude of the signal is determined digitally by determining the peak and holding that value for the duration of the 10 μs period. Since the peak is exactly 2.5 μs from the positive zero-crossing, this is easily implemented. When using the comparator option, however, the magnitude must be determined with an analogue circuit. The envelope of the RF-frequency is detected by means of a circuit similar to that in figure 7.16. The operational amplifier is optional, but does provide a rectifier with zero voltage-drop as well as high input impedance.

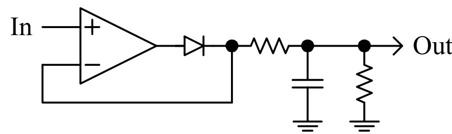


Figure 7.16: Envelope detector

7.3.6 Parasitic Components

The only potentially problematic parasitic component is the capacitance of the diodes and detection buffer. The combined capacitance is in the order of 10 pF. The effect is that a current flows in the coil, which re-induces a magnetic field in the sample. The magnitude of this field is derived in equations 7.25 to 7.30.

$$B = \mu_0 \left(H_y + M_y\right) \tag{7.25}$$

$$v = NA\mu_0 \left(\dot{H}_y + \dot{M}_y\right) \tag{7.26}$$

$$i = C\dot{v} \tag{7.27}$$

$$i = NCA\mu_0 \left(\ddot{H}_y + \ddot{M}_y\right) \tag{7.28}$$

$$i = \frac{H_y l}{N} \tag{7.29}$$

$$H_y = -M_y \frac{N^2 CA\omega^2 \mu_0}{l + N^2 CA\omega^2 \mu_0} \tag{7.30}$$

The effect is that the magnetisation vector will try to precess about this induced H_y, causing a phase-shift. In the stationary frame of reference, the induced **H** follows $\mathbf{M}_{x-y}$ so that they are always directly opposite each other. The net effect is thus an increase in precessional frequency. The phase-locked loop will compensate by adjusting the excitation frequency until the equivalent magnetic field vector ($H_{z,eq}$) is exactly opposite the magnetisation vector. This $H_{z,eq}$ is given by equations 7.31 and 7.32. The resulting error in frequency is thus given by equation 7.33.

$$H_{z,eq} = \frac{\omega_{rf} - \omega_0}{\gamma \mu_0} \tag{7.31}$$

$$H_{z,eq} = \frac{M_z H_y}{M_y} \tag{7.32}$$

$$\frac{\omega_{rf} - \omega_0}{\omega_0} = -M_y \gamma \frac{N^2 CA\omega_0 \mu_0^2}{l + N^2 CA\omega_0^2 \mu_0} \tag{7.33}$$

Provided that M_z is larger than M_y (as can be seen in figure 7.13), M_z may be approximated as M_0. This represents the worst-case scenario, as M_z will always be smaller than M_0. The error in frequency may then be approximated by equation 7.34. Figure 7.17 provides this error as a function of frequency for the coil designed in the previous section.

$$\frac{\omega_{rf} - \omega_0}{\omega_0} = \frac{a_0 N^2 CA\omega_0^2 \mu_0^2}{l + N^2 CA\omega_0^2 \mu_0} \tag{7.34}$$

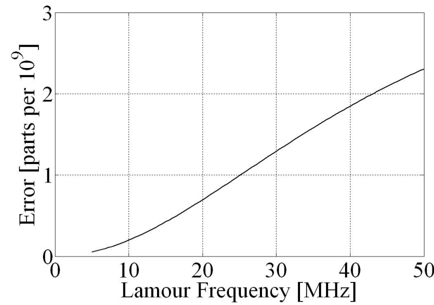
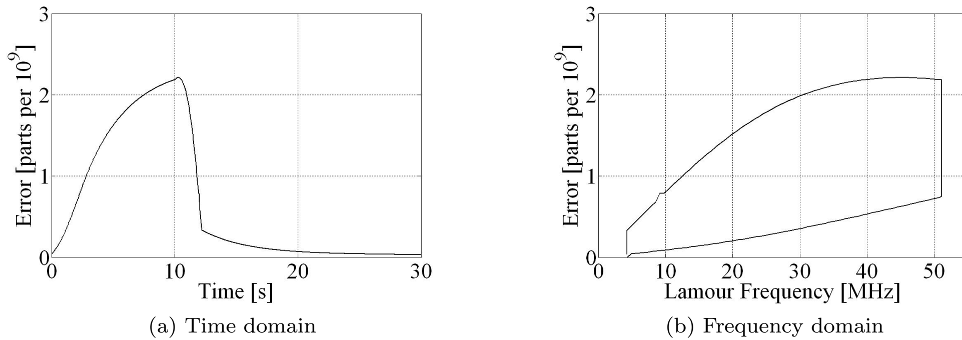


Figure 7.17: Theoretical measurement error

From this result it is obvious that the error is negligible. Two to three parts in 10^9 is much better than the desired 1 ppm. When using pulses to sustain resonance, however, the situation is a bit more complex. A simulation was run to view the effect. The same reference was used as in figure 7.9. Also, the same technique in sustaining resonance was used as in

figure 7.14. Figure 7.18 shows the results. The transient glitches have been removed for clarity.



(a) Time domain (b) Frequency domain

Figure 7.18: Simulated measurement error

From these graphs it is clear that the effect of sustaining resonance by means of pulses has negligible effect on the frequency error. The simulation confirms the theoretical analysis.

7.4 Alternative Control Circuit

7.4.1 DDS Operation

Figure 7.19 provides a block diagram of a typical DDS IC, where 'FTW' is the acronym for 'frequency tuning word'.

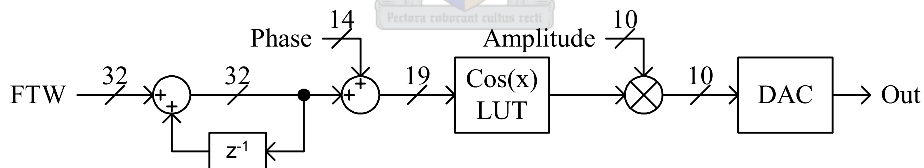


Figure 7.19: DDS core

By integrating the frequency, the phase is obtained. The digital representation of the phase wraps around automatically when the full-scale value is reached. In a standard DDS IC an optional phase may be added before the waveform is calculated by means of a cosine-function look-up table. As a final step, the amplitude may be adjusted before the output signal is generated by the DAC.

Using a DDS clock of 200 MHz and a 32-bit frequency word, any output frequency from 0 to 100 MHz can be generated with a resolution of 47 mHz. This frequency is given by equation 7.35, where f_o is the output frequency and f_c the clock frequency.

$$f_0 = \begin{cases} f_c\left(\frac{FTW}{2^{32}}\right) & , FTW \in \left[0, 2^{31}\right] \\ f_c\left(1 - \frac{FTW}{2^{32}}\right) & , FTW \in \left(2^{31}, 2^{32} - 1\right) \end{cases} \tag{7.35}$$

In the application of the probe, however, the phase and amplitude is not important. Also, the signal does not need to be sinusoidal. The output may simply be the most significant bit of the phase accumulator (i.e. digital integrator). Implementation within an FPGA is thus straightforward.

7.4.2 Frequency Content

The effect of generating a signal using DDS is to sample the output waveform at the clock frequency. This is no problem when generating sinusoidal signals, but the high-frequency harmonics of the square wave folds over into the base-band. The frequencies present within the output signal are given by equation 7.36.

$$f = nf_c + mf_0 \quad |n, m \in \mathbb{Z} \tag{7.36}$$

The intended output frequency results when n is zero and m unity. The largest output frequency required is 50 MHz. Problem areas are where n is unity and m smaller than 10 and odd. The ideal situation would thus be to have a clock frequency larger than 500 MHz, which is not possible. The largest practical clock frequency is 320 MHz (360 MHz for a 24-bit FTW). This would imply that there are problem frequencies near 32 MHz, 40 MHz and 53 MHz.

NMR has a very narrow bandwidth and the fundamental signal should have the greatest effect. A more serious problem arises when mixing all these different frequencies. This problem requires further practical investigation. Possible solutions include sharper filters and analogue mixers.

When using a square wave to excite the sample, only the first harmonic has any effect. This is due to the very narrow bandwidth of NMR. The magnitude of this first harmonic is $4/\pi$ of the magnitude of the square wave. The effective B_x is thus slightly larger using a square wave than when using a sinusoid of the same amplitude.

7.4.3 Block Diagram

In order to produce a signal that has a zero-average using logic-gates, a differential signal is used. This is then buffered by a differential amplifier before being sent to the probe.

The returning NMR signal is buffered, amplified and conditioned before being sent to the FPGA. Within the FPGA, XOR gates are used to multiply the mixing signal with the NMR signal. This result is filtered and conditioned outside the FPGA before being fed to the phase-counter and control circuitry. The resulting block diagram is shown in figure 7.20.

The envelope detector was given in figure 7.16. The differential amplifier circuit diagram is presented in figure 7.21. All resistances are 100 Ω and an AD8062 operational amplifier is used. The output has a peak-to-peak amplitude of 3.3 V and a 2.5 V bias.

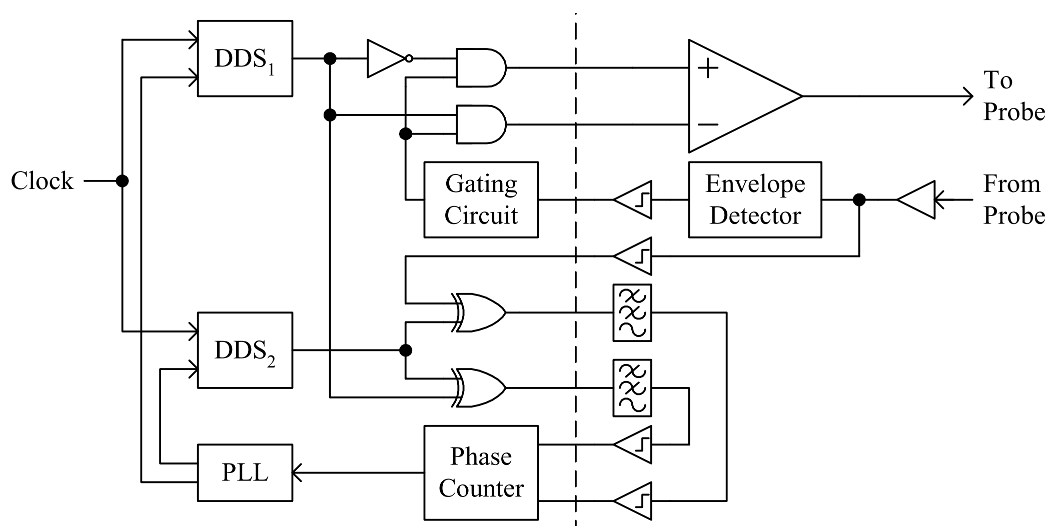


Figure 7.20: Alternative probe control circuit

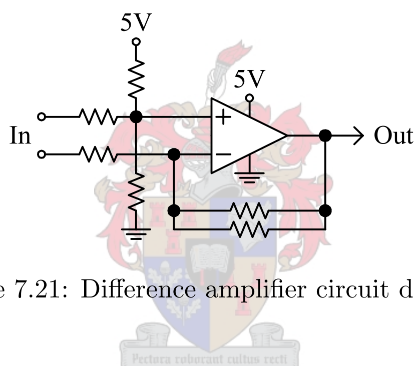


Figure 7.21: Difference amplifier circuit diagram

7.5 Summary

This chapter describes the theoretical analysis of a new NMR magnetometer based on a combination of pulsed and continuous-wave NMR. Resonance is obtained and sustained by providing short pulses at the Lamour frequency. Each pulse is large enough to keep the system in resonance, but much smaller than a $\pi/2$ pulse. Advantages include smaller pulse-amplifiers and drive circuits.

It is also possible to attain and sustain resonance by means of a very simple analogue circuit. This circuit comprises of an envelope detector, comparator and one-shot.

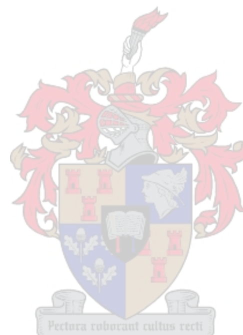
The Lamour frequency is tracked by means of a digital phase-locked loop. Simulation indicated the validity of this solution. Measurement rate can be increased to 100 ksps, which is much faster than conventional pulsed and continuous-wave NMR magnetometers. Also, it is best to use pure water as a sample so that the time-constants T_1 and T_2 are as large as possible. This solves the problem of crystallisation.

The excitation and detection coil may be the same coil. It has been designed as having

a cylindrical shape. The diameter is 10 mm, length 30 mm and the coil has 20 turns.

The NMR signal magnitude and phase is measured by mixing both detection and excitation signals with the same mixing signal. The frequency of the mixing signal is 100 kHz below that of the excitation signal. The amplitude and phase is measured digitally.

The hardware may be simplified by implementing both DDS and mixer circuits within the FPGA. Numerous advantages have been mentioned, but it does have potential disadvantages as well. These ideas were not implemented practically but left to future research.



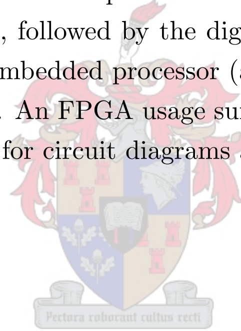
Chapter 8

Controller

8.1 Introduction

The previous chapters provided detail on the design of the converter, control system and digital PWM generator. This chapter describes the design of the controller. The controller is the central unit from which all other components are controlled.

The hardware is discussed first, followed by the digital circuits within the FPGA. The design and implementation of an embedded processor (a processor implemented within the FPGA) is described in more detail. An FPGA usage summary is also included. For the full design, please refer to appendix D for circuit diagrams and appendix E for VHDL code.



8.2 Hardware

8.2.1 Block Diagram

A detailed block diagram of the controller hardware is presented in figure 8.1.

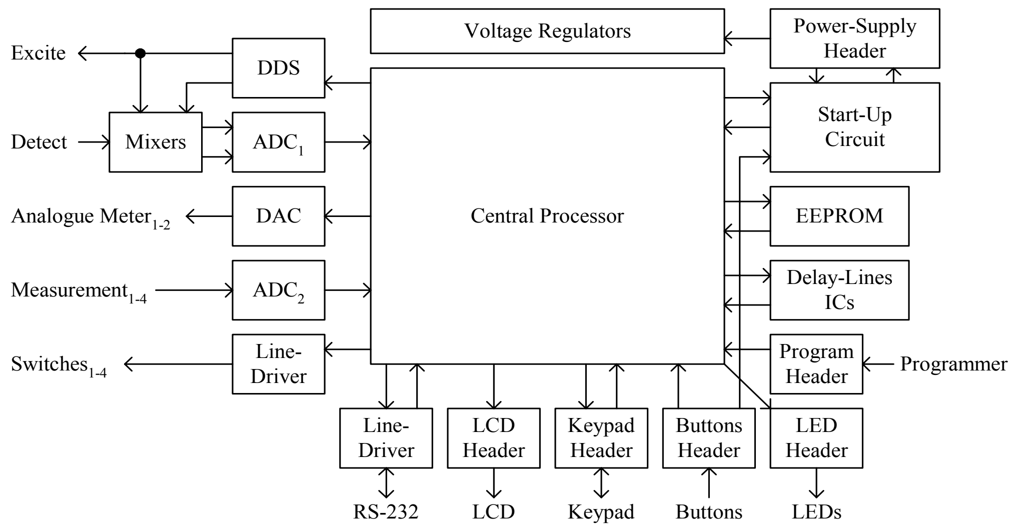


Figure 8.1: Control board block diagram

8.2.2 Central Processor

Any digital circuit requires some form of central processing unit. The main options are to use either a processor (such as a DSP) or programmable logic (such as an FPGA).

A processor has the advantage of containing various useful embedded modules, such as a serial port, timers, etc., implemented and ready to use. One disadvantage, however, is the difficulty of implementing a fast real time system. Since the program code is evaluated sequentially, events cannot occur in parallel.

This is where programmable logic takes the lead. Massive parallelisation is possible where, for example, the power supply controller can execute freely from the CPU. Implementing fast real time systems thus becomes possible.

It would also follow logically to use a single processing chip for all the digital tasks. It is for this reason an FPGA was chosen as the processor of choice. This makes it possible to implement the interface, peripheral drivers, controller, probe, PWM generator, etc., all within the same FPGA. An ALTERA Cyclone II FPGA is used (EP2C8Q208C7).

8.2.3 Power Supply

An ATX power supply switches on when its 'power-on' signal is grounded. This is achieved by pressing the 'on' button. The power supply responds by bringing all voltages to a regulated state and the asserting the 'power-good' signal to 5 V. This latter signal is then used to drive the gate of a small-signal MOSFET (through a resistor), which then keeps the 'power-on' signal low. By shorting the gate to ground (i.e. pressing the 'off' button), the power supply is switched off. This circuit is presented in figure 8.2.

When the voltages drift outside the specified range of 5%, the 'power-good' signal becomes low, switching off the MOSFET and thus also switching off the power supply.

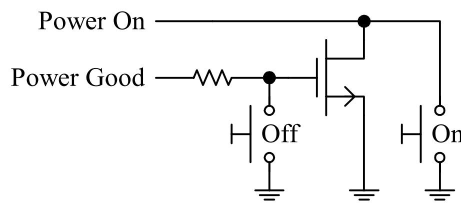


Figure 8.2: Soft-switch circuit diagram

This protects the control circuit against both over- and under-voltages.

Numerous different voltages are required for the various circuits. Figure 8.3 shows how they are obtained. Analogue power is separated from digital power so that the analogue circuits do not experience the digital switching noise. Also, each oscillator has its own regulator in order to ensure good stability and low jitter. L-C filters consist of a 6 μH ferrite-bead and a 10 μF ceramic capacitor (1206 package).

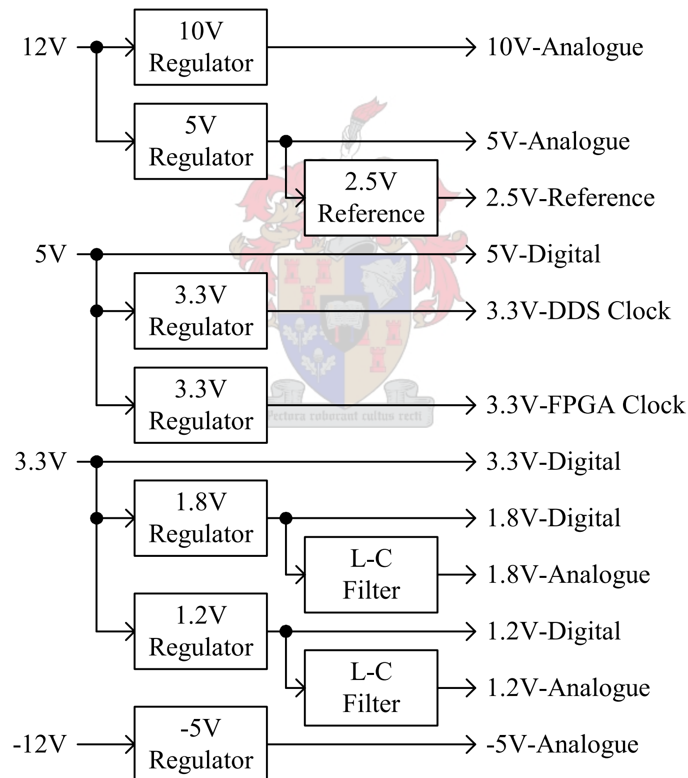


Figure 8.3: Power distribution

8.2.4 Start-Up Sequence

FPGA configuration is started by pulsing its 'n_config' pin low. This signal is held low for 100 ms after the 'power good' signal is received from the power supply. Once configuration has completed successfully, the FPGA asserts the 'init-done' pin high.

While this 'init-done' is low, the 'reset' signal is kept low (as in the case of holding the reset button down). The reset is released 100 ms after the rising edge of the 'init-done' signal. This ensures that all the components within the FPGA are properly reset. A push-button is also provided for system reset. Figure 8.4 provides a simplified block diagram of the process. Delays are implemented using an R-C circuit and open-collector comparator.

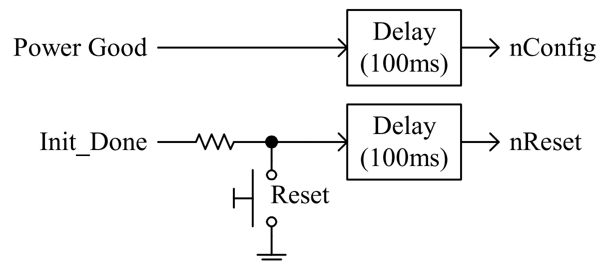


Figure 8.4: FPGA start-up sequence

8.2.5 Programmer

Since no special programming requirements exist for this project, a simple active-serial programming scheme was chosen. The ALTERA EPCS4 programming chip was used, together with a parallel-port ByteBlaster II. A standard IDC-10 header was used for connecting the ByteBlaster to the control board.

8.2.6 Interface Headers

An IDC-16 header is used for both the three switches (on, off and reset) and the 5 LEDs (power, error, field locked, field high and field low). Every second pin is connected to ground. On the control board itself all the abovementioned LEDs are included, along with an additional two: one indicating that the FPGA configured correctly and one indicating that the power-on reset has been completed. These are both green.

Another IDC-16 header is included for the LCD. The 16 pins are used for power, ground, brightness, 3 control signals (of which only 2 are used), 8 data signals and 2 for the back-light. An IDC-10 header is used for the keypad, of which 2 pins remain unconnected. The 8 remaining pins are divided into two groups: 4 for the x-connections and 4 for the y-connections. A DB-9 header is used for the RS-232 port.

8.2.7 ADC

Provision is made for four 18-bit measurements from the current source. These are all single-ended signals in the range of 0 to 5 V. The ADCs, however, require a differential input (each end 0 to 4.096 V). First, the incoming signal is scaled down to 0 to 4.096 V using a resistor voltage-divider. A capacitor is included to form a 100 kHz, first-order low-pass filter. The scaled signal is then buffered to produce the positive end of the differential signal. Next, an inverting operational amplifier is used to generate the negative end.

The required sampling frequency is at least once every PWM period, which is 2.620 μs. There must also be enough time left for the control system to work out the next duty-cycle, which is 655 ns (discussed in section 5.9.6 on page 88). This leaves 1.965 μs for the sampling process to complete. In order for the digital signals to not corrupt the analogue sample, the digital data is read after the conversion is complete. In order to minimise FPGA pin usage, serial-interface ADCs have been chosen. Using a 24 MHz clock to read the 18-bit sample through the serial interface takes 750 ns. This implies a conversion time of at most 1.215 μs. The AD7674 from Analog Devices has a conversion time of 1 μs (in 'warp-mode') and is thus sufficient.

8.2.8 DAC

An 18-bit, 750 ksps dual DAC (AD1868) is included for the purpose of testing. It is also used to drive the analogue meters on the front panel. Its output is in the range of 1.5 V to 3.5 V. The AD1868 conveniently provides a 2.5 V bias voltage, around which the signal is amplified to have a range 0 V to 5 V. The amplifier is then followed by a second-order Butterworth filter with a cut-off frequency of 325 kHz. This frequency was chosen so that it is less than half the sample-frequency (Nyquist criterion [29]), and practical component values also played a role. Both amplifiers and filters are implemented using a single quad operational amplifier (TLE2084).

8.2.9 Gate Signals

It is important to keep the MOSFET gate signals as jitter-free and noise-free as possible. A high-speed logic standard (LVPECL) [33] is thus used for transferring the PWM pulses to the current source. Single-ended LVPECL has the advantage of using fewer cables, as well as easier routing. SMA connectors are used with 50 Ω co-axial cable.

To make the LVPECL signals single-ended, the negative signal is terminated on the control board. The driver thus perceives no difference in load. The LVPECL receiver has a differential-amplifier input. The negative input is connected to the midpoint of the single-ended voltage-swing. The incoming signal is connected to the positive input. This enables the receiver to function normally. Figure 8.5 shows the circuit diagram for single-ended LVPECL.

The FPGA has pull-up resistors on its output pins during its configuration stage. In the case of a half-bridge or synchronous buck converter, this would mean that both MOSFETs

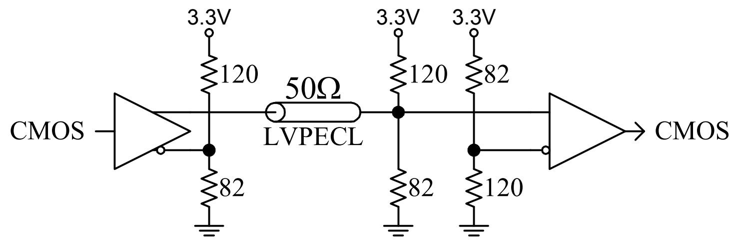


Figure 8.5: Single-ended LVPECL

are on during initialisation, shorting out the bus. A simple external logic circuit using the four NOR gates in a 74LVC02 IC solves this problem. The schematic is presented in figure 8.6. The idea is to let the switching signals through as long as they are not both high at the same time. When they are high simultaneously, both outputs are low. In the case of a standard buck converter and a full-bridge, this circuit is not required.

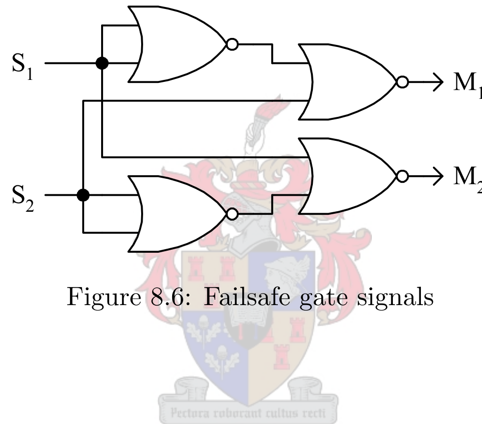


Figure 8.6: Failsafe gate signals

8.2.10 Delay Lines

In chapter 6 the need for delay lines was discussed. Two delay-lines are included for experimentation purposes, one for each converter stage.

Since the delay-line functions with LVPECL, the circuit includes a dual CMOS to LVPECL converter, the two delay-line ICs and a dual LVPECL to CMOS converter.

8.2.11 EEPROM

Often the need arises to store data in non-volatile memory. This would include the control system parameters, settings prior to power-down, CPU execution code, etc. 64 kB of EEPROM is provided in eight serial-interface 64 kb ICs.

The AT25640 from Atmel has a read cycle as fast as the serial-interface can handle (6.4 μs). Unfortunately, it has a write-cycle of 5 ms. Data are usually written sequentially. Since there are eight ICs, the chip address can either be the most-significant three bits of the full 16-bit address, or the least-significant three. Choosing the least significant three has

the advantage that the next byte can be written while the previous one's IC is still busy. This means that instead of writing data at a maximum of 200 Bps, data may be written at 1.6 kBps. This is advantageous when programming the CPU.

8.2.12 DDS

The DDS IC used for this project (AD9859 from Analog Devices) has a differential current output. This output needs to be biased externally to 1.8 V. Ferrite beads (6 μH) are used for this purpose. Also, a 50 Ω output impedance is required in order to match the transmission line used to send the signal to the probe. The output current range of each side of the differential output is 0 to 10 mA. The maximum voltage range is 1.8 ± 0.5 V. Loading the circuit with a 50 Ω resistor would thus meet the requirements. The circuit is shown in figure 8.7. Capacitances are 100 nF.

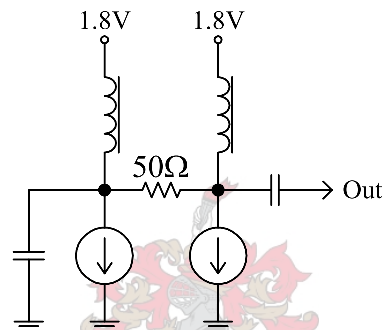


Figure 8.7: DDS biasing circuit

This circuit may be simplified to the circuit in figure 8.8, which removes the need for ferrite-beads. The output of the DDS IC is then at its limit, which is not a good situation. The circuit above was used instead. Another advantage of the above circuit is that the ferrite beads isolate the analogue output from any digital transient voltage disturbances that might be present on the 1.8 V supply.

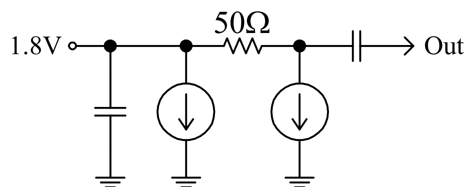


Figure 8.8: Simplified DDS biasing circuit

8.2.13 Mixer

A Gilbert cell mixer [1] is shown in figure 8.9, along with the biasing used in the probe circuit. The MC1496 from ON Semiconductor was used.

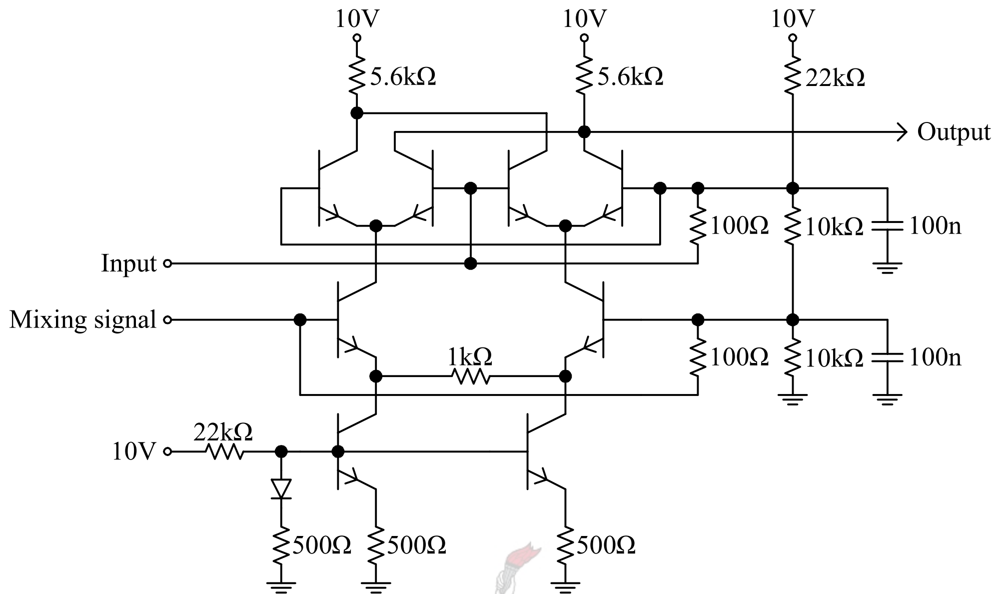


Figure 8.9: Gilbert cell mixer [1]

The basic operation is based on the fact that the gain of a common-emitter transistor amplifier is directly proportional to its biasing current [34]. The bottom pair alters the biasing currents of the top two pairs, thus altering their gain. The result is the product of the two input signals.

8.2.14 Transmission Lines

Both switching signals and probe signals are high-frequency signals that need to travel over extended distances. Transmission lines are thus needed on the PCB so that the connections to the co-axial lines are impedance-matched. Figure 8.10 shows the cross-section of a strip-line transmission line.

The equation for the characteristic impedance of this line is giving by equation 8.1 [35], where ϵ_r is the relative permittivity of the PCB material.

$$Z_0 = \frac{87}{\sqrt{\epsilon_r + 1.41}} \ln\left(\frac{5.98d}{0.8w + t}\right) \Omega \tag{8.1}$$

For a standard 1.55 mm 4-layer or 6-layer PCB, t is 15 μm, d 238 μm and ϵ_r 4.6. This implies a width of 416 μm. Similarly, for a 1.55 mm double-layer PCB, d is 1.5 mm and the required width is 2.72 mm.

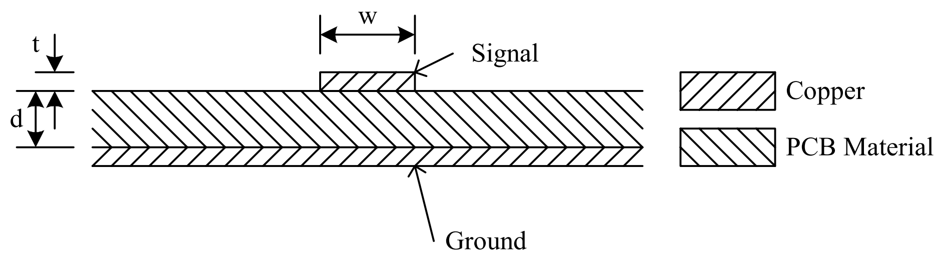


Figure 8.10: Strip-line transmission line

8.2.15 FPGA I/O Pin-Count

Conveniently, many components share the same signal, thus decreasing the FPGA I/O pin-count. These signals include data lines, clocks, etc. The full I/O pin-count is presented in table 8-I.

Table 8-I: FPGA I/O pin-count

Peripheral	**Component**	**Pins**
Power and start-up	Power-good signal	1
Interface	Reset switch	1
	LEDs	6
	LCD	11
	Keypad	8
	RS-232	2
Probe	DDS (x2)	5
	ADC	9
ADC	ADC (x4)	11
DAC	DAC	4
Switches	Switches	4
Delay lines	Delay lines (x2)	24
EEPROM	EEPROM (x8)	11
Clock	125 MHz oscillator	1
	Total	**98**

Many components listed in this table are for experimentation purposes only. The pin-count of the final product will be much lower and a smaller FPGA may thus be used.

8.2.16 Physical Layout

The control board PCB consists of four layers. These are, from top to bottom:

- Top signal and transmission lines
- Ground (single solid plane)

- VCC planes and power lines, including the following:
  - 3.3 V (digital)
  - 1.8 V (digital)
  - 1.2 V (digital)
  - 10 V (analogue)
  - 5 V (analogue)
  - 1.8 V (analogue)
  - -5 V (analogue)
- Bottom signal

Some signals are routed in the VCC layer, but this is minimised so that a solid power distribution can be achieved. Also, there are no signals on the ground-plane, therefore reducing EMI effects. Care was taken to keep the digital and analogue signals separate. When viewed from above, analogue circuitry is at the top left while digital circuitry occupies the remaining area.

8.3 FPGA Detail

8.3.1 Block Diagram

A modular approach was taken in the design of the digital system. Figure 8.11 presents the system block diagram.

The peripheral drivers are implemented with state-machines and provide a simple interface with the other modules in the system. It is necessary to define a few aspects at this point.

All the drivers (except for the DDS) are connected to the bus, which gives the CPU the impression that the peripheral drivers are simply part of the system RAM. The CPU is also responsible for providing the control-constants (stored in EEPROM) for the controller as well as controlling the soft-start sequence.

The CPU itself is implemented with a Harvard architecture [36]. This means that the program memory (included within the CPU block) is separate from the data memory. In a classical Harvard architecture, however, the CPU cannot write to the program memory. In this project the CPU can write to the program memory by means of a special instruction. This makes it possible for the CPU to program itself with data in EEPROM, which is the first thing that happens during the boot sequence.

The RS-232 interface is implemented as a set of instructions to the CPU. These instructions are listed in table 8-V. The code is sent first, followed by the argument stack (bottom-up). The CPU will then send the return data upon completion.

To read and write directly to and from EEPROM is a mere convenience to allow use of fewer communication words during programming. The '00 hex' return data is used to

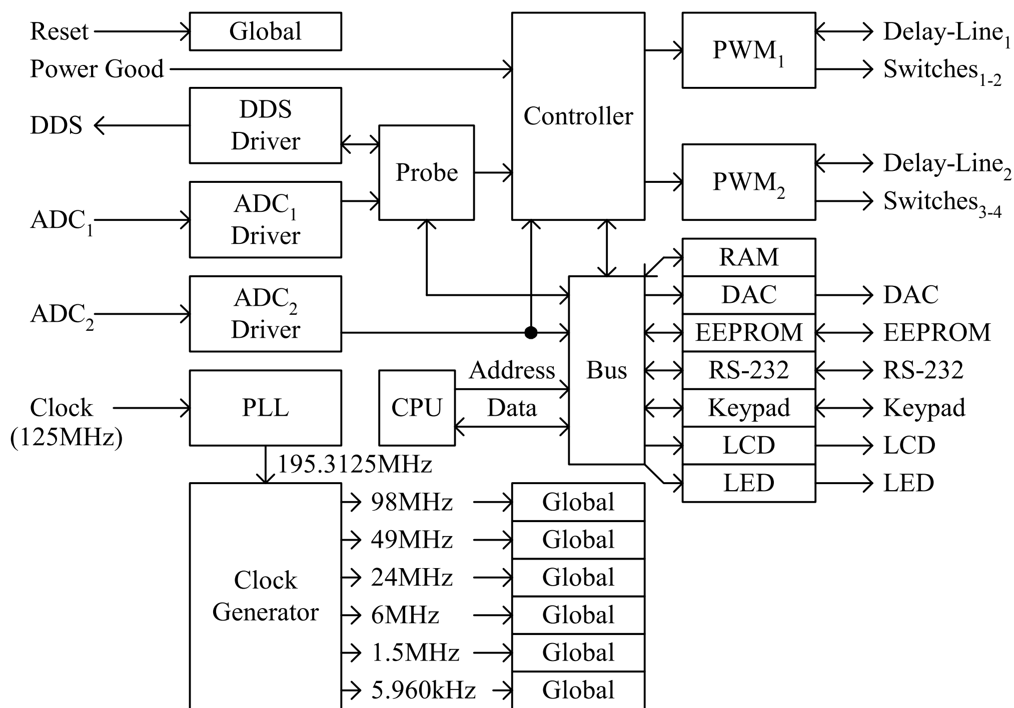


Figure 8.11: FPGA block diagram

signal that the next EEPROM 'write' instruction may be given. This is included because the EEPROM write-cycle is much slower than the time it takes for the instruction to be sent via 115200 baud RS-232.

As the CPU perceives all the peripherals as RAM, reading and writing to and from RAM enables the computer to fully control the system if desired.

8.3.2 Clock Requirements and Generation

Table 8-II shows the components implemented within the FPGA, along with their clock requirements. These are calculated using timing requirements of the components, such as maximum SPI clock frequency. Since the component clock is generated using state-machines, the required frequency is doubled.

The only component that needs an exact clock is the PWM, since its period needs to be an exact integer multiple of the delay-line resolution (10 ps). A 125 MHz oscillator IC is used as a master clock for the FPGA. The internal phase-locked loop is then used to divide this frequency by 16 and multiply the result by 25. This provides a clock of 195.3125 MHz, which is exactly double the desired PWM clock frequency.

This 195 MHz signal is used to clock a 15-bit synchronous up-counter in order to generate the various in-phase harmonic clocks needed by the system. Also included in table 8-II is the counter tap (equivalent to the power of 2 by which the 195 MHz signal is divided) and the actual clock used in the design.

Table 8-II: FPGA clock requirements

Component	**Clock requirement [Hz]**	**Counter tap**	**Actual Clk [Hz]**
ADC (Probe)	< 60 000 000	2	48 828 125.000
ADC (Controller)	< 80 000 000	2	48 828 125.000
COMM	>> 115 200	5	6 103 515.625
Controller	≈ 50 000 000	2	48 828 125.000
Controller : Δt	≈ 4 000	15	5 960.464
CPU	≈ 1 000 000	7	1 525 878.906
DAC	< 27 000 000	3	24 414 062.500
DDS	< 50 000 000	2	48 828 125.000
EEPROM	< 10 000 000	5	6 103 515.625
Keypad	≈ 4 000	15	5 960.464
LCD	< 4 000 000	7	1 525 878.906
PWM	= 97 656 250	1	97 656 250.000

8.4 Peripheral Software

8.4.1 EEPROM

Each EEPROM IC has four control lines. These are ‘data in’, ‘data out’, ‘data clock’ and ‘chip-select’. The data lines are shared between the eight ICs, implying a total of 11 lines.

After reset, each IC is initialised. This is performed by reading the status register of each and determining whether or not it is set up correctly. If not, the problem is corrected. The status register is non-volatile, so a write operation will only occur upon the first power-up of the controller.

Before each interaction with a particular IC, the ‘Busy’ bit in the status register is read. The controller waits for that particular IC to finish before sending the relevant instruction.

In order to speed up the writing process, the least significant 3 bits of the 16-bit address are used to decode the chip-select signal. The result is that the next byte in the stream may be written while the previous seven ICs are still busy.

On the CPU side, the data and address are set up before an instruction is provided. Once the instruction is given, the I/O registers become read-only until the operation is complete. Figure 8.12 provides a flow-diagram.

The address allocation table is presented in table 8-III.

Much space remains available between addresses ‘4000’ and ‘8000’ to store data needed for future developments. Such developments might include data redundancy so that no data loss occurs during a power-failure.

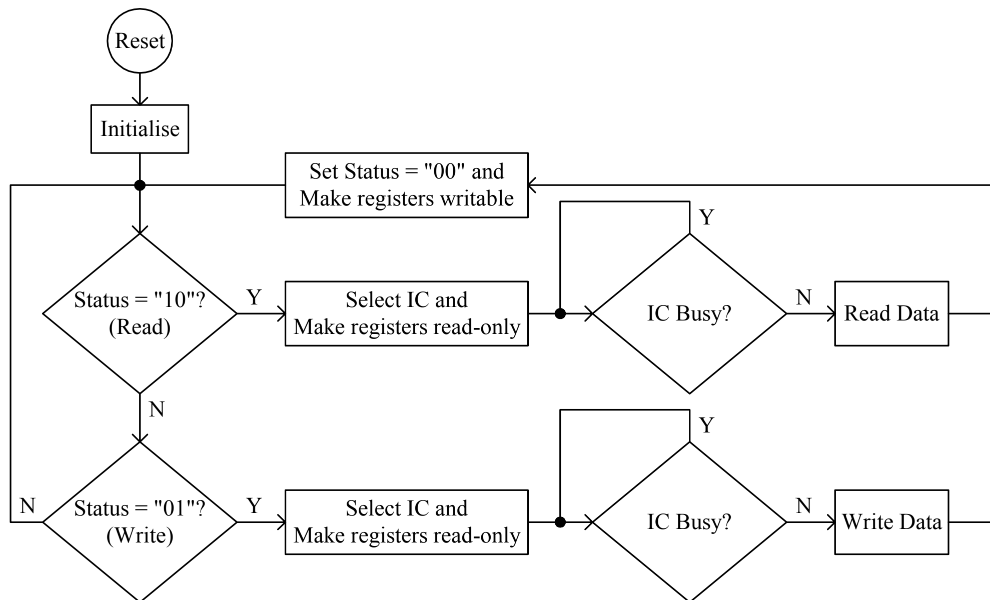


Figure 8.12: EEPROM driver flow diagram

Table 8-III: EEPROM address allocations

Group	Data	Start address	End address
Program	Execution code	0000	1FFF
Menus	Menu text	8000	FFFF
Stage 1 controller	V_1 gain	2000	2002
	I_1 gain	2004	2006
	Reference gain	2008	200A
	Reference ramp limiter	200C	200E
Stage 2 controller	V_2 gain	2010	2012
	I_2 gain	2014	2016
	I_3 gain	2018	201A
	Integrator gain	201C	201E
	Reference gain	2020	2022
	Reference ramp limiter	2024	2026
Controller	Status	2027	2027
	Reference	2028	202B
Probe	Field constant	2FFC	2FFF
	Phase compensation LUT	3000	3FFF

8.4.2 LCD

The LCD driver is very simple. Once the LCD has been initialised after reset, the LCD buffer (RAM) is continuously copied to the LCD. It takes 4.2 ms to copy the 80 characters and as such the LCD is updated at a frequency of 238 Hz. This is much faster than can be perceived by the human eye.

All the CPU sees is a RAM block of 128 bytes. It can read and write to this block. The most significant 2 bits of the 7-bit address represents the line and the least significant 5 bits the character within that line.

8.4.3 Keypad

The keypad driver consists of two parts: a keypad reader and a FIFO queue. Figure 8.13 shows the connections within a keypad. By pressing a button the two lines crossing underneath are connected.

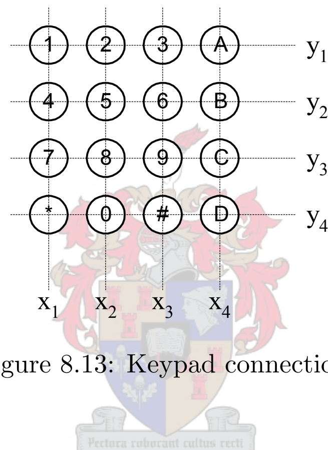


Figure 8.13: Keypad connections

The keypad reader alternately drives each of the four y-connections high while keeping the other three at high-impedance. The x-connections are then read at each point to map the status of the 16 buttons. When the button is not pushed, its corresponding x-connection is pulled low by means of a 10 kΩ resistor. In order to lower sensitivity to glitches, the past 5 samples must be the same before the status changes.

When a button has been pushed and the status updated, the event is stored in a circular FIFO queue of length eight. When two buttons are pressed at once, the event is ignored. The button must be released before another event can be registered.

The CPU reads the data from the keypad driver to determine whether a key has been pressed. The least significant 4 bits represent the key code (table 8-IV). The most significant (8th) bit is high when the queue is empty. By writing to the keypad driver the data output is updated with the next item in the queue.

Table 8-IV: Key codes

Code	Key
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	*
1111	#

8.4.4 ADC

The ADC conversion cycle is started by pulsing the 'conversion-start' signal low. The ADC then responds by making its 'busy' signal high.

In order to synchronise the four ADCs, their 'conversion start' signals are all connected together. Their 'data' and 'busy' signals are kept separate in order to allow all four to be read at the same time. The read cycle is started once all four 'busy' signals become low.

Pectora roborant cultus recti

8.4.5 DAC

The DAC driver works similarly to that of the LCD. The CPU can read and write to the DAC buffer. The DAC driver continuously updates the DAC IC at a rate of 650 ksps.

The driver also automatically converts the absolute value given by the CPU to a two's complement representation accepted by the DAC. The most significant bit of the data is inverted while all the others are not.

Each channel of the dual DAC is followed by a 2nd order Butterworth low-pass active filter with a cut-off frequency at 325 kHz.

8.4.6 RS-232

The fastest reliable RS-232 connection was found to be 115200 baud. A 6 MHz clock is used to calculate the necessary delays.

The RS-232 driver has three modules. These are a sender, receiver and controller. The sender retains a high output until it is required to send, or for a minimum of 8.68 μs. This

represents the 'stop bit'. The data is preceded by a 'start bit' (output low) before being sent. It is sent, least significant bit first, at one bit every 8.68 μs.

The receiver waits for a falling edge ('start bit'), after which it pauses for 13.0 μs (one and a half bits). It then loads the data (least significant bit first) once every 8.68 μs. This ensures that the data is sampled in the middle of the bit, rather than near an edge.

The controller provides an interface to the CPU. The transmit data register can be read and written. The received data register is read-only. The status register has two bits, a 'send' bit and 'received' bit. The CPU can set the 'send' bit, but not clear it. Similarly, the 'received bit' can be cleared, but not set. Thus writing '11' to the status register starts transmitting data without influencing the status of the receiver. Similarly, writing '00' to the status register acknowledges that the received data has been read without influencing the status of the transmitter. The controller resets the 'send' bit upon completion and sets the 'received' bit upon receiving data.

Although not yet implemented, it is intended for the RS-232 to have a FIFO queue [37] (for received data) 512 bytes long (1 M4K block). There is no 'send' buffer and the CPU must wait for the one byte to finish sending before it can send the next. No RS-232 command requires more than one byte return data, so this is not a problem. Table 8-V presents the commands.

Table 8-V: RS-232 Instructions

Operation	**Code**	**Argument stack**	**Return data**
No operation	0x00	Empty	None
Write EEPROM	0x01	Data Address1 Address0	0x00
Read EEPROM	0x02	Address1 Address0	Data
Reboot	0x03	Empty	None
Write RAM	0x04	Data Address1 Address0	None
Read RAM	0x05	Address1 Address0	Data

Any of the CPU functions can be controlled by manipulating the RAM. It is for this reason there are so few instructions. The ability to manipulate the EEPROM directly is simply to decrease the time it takes to read or write large amounts of data via the relatively slow serial connection.

8.4.7 Bus

The purpose of the bus module is to map all the different components to a RAM address space. It achieves this by decoding the address and then connecting the correct data port and latch to the CPU data port and latch. The assigned addresses are presented in table 8-VI.

Table 8-VI: Bus address allocations

Group	**Data**	**Start address**	**End address**	**Notes**
RAM	RAM	0000	01FF	1 M4K block
LCD	LCD buffer	8000	807F	
RS-232	Transmit data	8080	8080	
	Received data	8081	8081	
	Status	8082	8082	"000000" : Received : Send
EEPROM	Address	8090	8091	16-bit, MSB at higher address
	Data	8092	8092	
	Status	8093	8093	"000000" : Read : Write
LED	LED	80A0	80A0	"000" : LCD Backlight : Field low : Field high : Field locked : Error
Keypad	Keypad	80B0	80B0	Empty : "000" : Data
DAC	DAC1	80C0	80C2	24-bit value
	DAC2	80C4	80C6	24-bit value
ADC	V_1	80D0	80D2	24-bit value
	I_1	80D4	80D6	24-bit value
	V_2	80D8	80DA	24-bit value
	I_2	80DC	80DE	24-bit value
Timer	Timer	8089	8089	Increments every second
Stage 1 controller	V_1 gain	8100	8102	
	I_1 gain	8104	8106	
	Reference gain	8108	810A	
	Reference ramp limiter	810C	810E	
Stage 2 controller	V_2 gain	8110	8112	
	I_2 gain	8114	8116	
	I_3 gain	8118	811A	
	Integrator gain	811C	811E	
	Reference gain	8120	8122	
	Reference ramp limiter	8124	8126	
Controller	Status and control	8128	8128	
Probe	Measured field	8200	8203	
	Field constant	8204	8207	
	Status and control	8208	8208	
	Phase compensation LUT	9000	9FFF	

8.5 CPU

8.5.1 Justification

There are three options available when implementing a large interface. The first is to use a very large state-machine, which occupies much valuable space within the FPGA. The second is to use an embedded processor, which is basically a much smaller, but general purpose state-machine. The third option is to use an external processor, such as a 'PIC' from Microchip.

Altera offers an embedded processor called 'Nios', which can be customised to the specific application and programmed using the *C* programming language. This saves development time on a complex processor, but is very expensive intellectual property. The external processor is a very attractive option, since it has a built-in RS-232 and USB interface.

The 16 lines needed for the bus can easily be implemented with a standard 4-port PIC. It also provides more program memory than is available on the FPGA. A minor disadvantage is the fact that it uses space on the controller PCB and valuable I/O pins. The next iteration of the design must consider this option.

The controller board that was built, however, does not include an external processor. Since this project requires a very simple processor with little functionality, a new embedded processor was developed. The total development time was two weeks, including writing an assembler, programmer and boot loader.

8.5.2 Block Diagram

A Harvard architecture is used, where the program memory is separate from data memory. This is to simplify the instruction fetch stage. Both program memory and data memory may be manipulated. A 16-bit address space is used for both. Also, the essence of operation is a simple stack machine. The block diagram of the processor is presented in figure 8.14. A description of each signal path follows in table 8-VII.

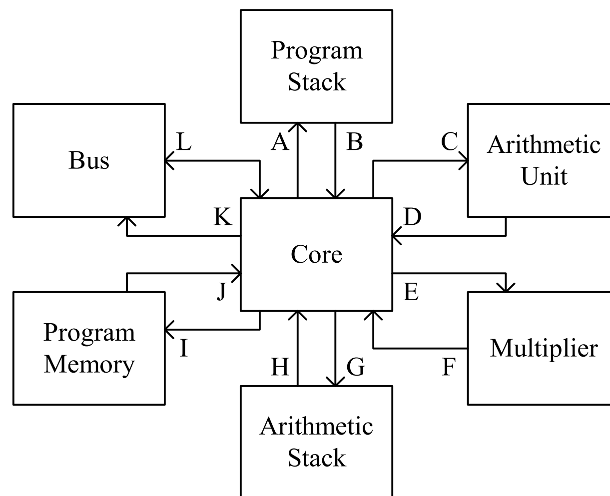


Figure 8.14: CPU block diagram

It must be noted that both the arithmetic stack and the program stack do not provide any indication of when they are full. Both are implemented as circular lists and will never overflow. The program and arithmetic stacks have lengths of 256 and 8 respectively. It is left to the programmer to ensure that the data stored within is not overwritten without intention.

Pectora roborant cultus recti

Table 8-VII: CPU signal descriptions

Component	Path	Signal	Width [bits]	Description
PC stack	A	Data	16	Data to be written to the program stack
		Address	8	Address of the topmost item on the stack
		Latch	1	Pushes the data onto the stack
	B	Top	16	The top-most item on the stack
Arithmetic unit	C	Data A	8	Data input A
		Data B	8	Data input B
		Task	4	Task to be performed by the unit (see table 8-VIII)
		Carry in	1	Carry flag status prior to the operation
		Zero in	1	Zero flag status prior to the operation
	D	Data Y	8	Answer
		Carry out	1	Carry flag status after the operation
		Zero out	1	Zero flag status after the operation
Multiplier	E	Data A	8	Data input A
		Data B	8	Data input B
	F	Data Y	16	Answer (A x B)
Arithmetic stack	G	Input	8	Data to be stored on the stack
		Address	3	Address relative to top of stack ('0' is the topmost item)
		Task	2	Task to be performed (see table 10-VII)
		Latch	1	Performs the task
	H	Out 0	8	Item on top of the stack
		Out 1	8	Second item on the stack
		Out A	8	Item at address 'Address'
Program memory	I	Address (Read)	16	Address to read from (the top 13 bits are active)
		Address (Write)	16	Address to write to (the top 13 bits are active)
		Data	8	Data to be written
		Latch	1	Write the data
	J	Instruction	8	Instruction stored at the 'Read' address
Bus	K	Read (not Write)	1	Controls the direction of data on the 'Data' line
		Address	16	Address of the data
		Latch	1	Writes data to the bus
	L	Data	8	Data read or data to be written

Table 8-VIII: Arithmetic unit tasks

Task	Output Y	Output Carry	Output Zero
0	A	Carry in	Zero in
1	A + B + Carry	Overflow	Y = '00000000' ?
2	A + B	Overflow	Y = '00000000' ?
3	A and B (bitwise)	Carry in	Y = '00000000' ?
4	Not A + 1 (signed negate)	Carry in	Y = '00000000' ?
5	Not A	Carry in	Y = '00000000' ?
6	A or B (bitwise)	Carry in	Y = '00000000' ?
7	A - B - Carry	The borrow bit	Y = '00000000' ?
8	A - B	The borrow bit	Y = '00000000' ?
9	A xor B (bitwise)	Carry in	Y = '00000000' ?
10	Rotate A left through carry	Most significant bit of A	Y = '00000000' ?
11	Rotate A right through carry	Least significant bit of A	Y = '00000000' ?
12	Rotate A left	Carry in	Y = '00000000' ?
13	Rotate A right	Carry in	Y = '00000000' ?
14	Shift A left (with carry)	Carry in	Y = '00000000' ?
15	Shift A right (with carry)	Carry in	Y = '00000000' ?

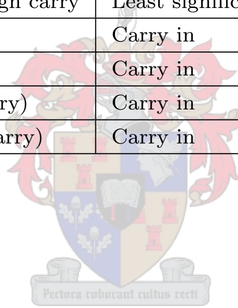

Pectora roborant cultus recti

Table 8-IX: Arithmetic stack tasks

Task	Description
'00'	Store 'Input' at top of stack
'01'	Push 'Input' onto stack
'10'	Store 'Input' second from the top and pop the stack
'11'	Swap the topmost item with the one at address 'Address'

8.5.3 Instruction Set

In order to save storage space, an 8-bit instruction set was developed. The first two bits are conditional execution bits. An instruction with the 'carry' condition set, for example, will only execute if the carry flag is set. An instruction with both condition bits cleared will always execute. The next bits have different meanings, depending on the instruction. The full instruction set is presented in table 8-X.

Table 8-X: CPU instruction set

Instruction	**Description**	**Opcode**								**Operand**
		7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**[bits]**
adc	Add with carry	C	Z	0	0	0	0	0	Pop	0
add	Add	C	Z	0	0	0	0	1	Pop	0
and	And	C	Z	0	0	0	1	0	Pop	0
div	Unsigned divide	C	Z	0	0	0	1	1	Pop	0
mul	Unsigned multiply	C	Z	0	0	1	0	0	Pop	0
neg	Negate	C	Z	0	0	1	0	1	0	0
not	Not	C	Z	0	0	1	0	1	1	0
or	Or	C	Z	0	0	1	1	0	Pop	0
sbb	Subtract with borrow	C	Z	0	0	1	1	1	Pop	0
sub	Subtract	C	Z	0	1	0	0	0	Pop	0
xor	Exclusive or	C	Z	0	1	0	0	1	Pop	0
rcl	Rotate through carry left	C	Z	0	1	0	1	0	0	0
rcr	Rotate through carry right	C	Z	0	1	0	1	0	1	0
rol	Rotate left	C	Z	0	1	0	1	1	0	0
ror	Rotate right	C	Z	0	1	0	1	1	1	0
shl	Shift left	C	Z	0	1	1	0	0	0	0
shr	Shift right	C	Z	0	1	1	0	0	1	0
cc	Clear carry	C	Z	0	1	1	0	1	0	0
cz	Clear zero	C	Z	0	1	1	0	1	1	0
nc	Not carry	C	Z	0	1	1	1	0	0	0
nz	Not zero	C	Z	0	1	1	1	0	1	0
sc	Set carry	C	Z	0	1	1	1	1	0	0
sz	Set zero	C	Z	0	1	1	1	1	1	0
swp	Swop stack items	C	Z	1	0	0	Adr2	Adr1	Adr0	0
pop	Pop stack	C	Z	1	0	1	0	0	0	0
ret	Return from function	C	Z	1	0	1	0	0	1	0
call	Call function	C	Z	1	0	1	0	1	Absolute /Relative	16 8
jmp	Jump	C	Z	1	0	1	1	0	Absolute /Relative	16 8
ld	Load from bus	C	Z	1	0	1	1	1	Reference /Value	16
lds	Load from stack	C	Z	1	1	0	Adr2	Adr1	Adr0	0
ldi	Load immediate	C	Z	1	1	1	0	0	0	8
ldpr	Load from program memory	C	Z	1	1	1	0	0	1	16
stpr	Store to program memory	C	Z	1	1	1	0	1	Pop	16
st	Store to bus	C	Z	1	1	1	1	Pop	Reference /Value	16

When the 'pop' bit is set, the stack is popped during the instruction. The instruction 'add pop', for example, will take the top two elements in the stack, add them, store the answer second from the top and then pop the stack. The advantage is that separate 'swp 1' and 'pop' instructions are not needed, saving program space. With the 'pop' bit cleared, the top item of the stack is replaced with the answer.

The answer to multiplication is pushed most-significant byte first. When using the 'pop' option with multiplication, the stack height stays the same, otherwise it increases by one.

Using the 'div' instruction is similar. The denominator must be pushed first, followed by the numerator (most significant byte first). The answer is stored in place of the two-byte numerator, decreasing the stack height by one. Using the 'pop' option decreases the stack height by two.

The program flow instructions have two addressing options. These are an 8-bit relative address and a 16-bit absolute address. The relative address is relative to the address at which the jump or call opcode is stored. The address is given after the opcode (least significant byte first).

Bus memory access also has two addressing modes, both 16-bit. The 'value' mode means that the value at the address is manipulated. The 'reference' mode loads the value at the given address and then uses that value as the address for manipulation.

8.5.4 Boot Loader

At power-up, the program memory is initialised with the boot-loader. When the CPU starts, the boot-loader code thus runs first. The task of the boot-loader is to copy the program from EEPROM to the program memory.

The first few instructions are to check whether the same first few instructions stored in EEPROM are the same as those in the boot-loader. These must remain constant, for the program is busy running while the memory it is running from is being manipulated. If it is not the same, it means that either the EEPROM is un-initialised (or corrupt) or the program loaded into EEPROM does not have the boot-loader as its first instructions. If that be the case, no copying takes place and the CPU carries on running only the initialisation code.

After the boot-loader, the initialisation code also includes an RS-232 command handler. This enables RAM and EEPROM manipulation without having the full interface loaded.

8.5.5 Tests

All the instructions were thoroughly tested and everything is fully operational. Arithmetic and shift instructions were tested using the RS-232 port. The computer sent the two operands and the CPU returned an answer. All possibilities were compared with the answer calculated by the computer for the same inputs. Program flow and conditional instruction execution was tested by writing loops and if statements. The boot loader, assembler and programmer are all fully operational.

8.6 Program Layout

8.6.1 Core

The program is not designed in full, but a few ideas are in place. The first code to be executed is the boot loader, which loads the program from EEPROM into the program ROM. The rest of it will be similar to an event-driven program, such as a Windows application.

At the heart of the program core is a series of case statements within a continuous loop. In the case of an event, the respective handler function is called. Once the handler is done, the next series of events is checked for. The RS-232 is checked first in order to see if any commands have been sent. The keypad is checked next for key-presses, etc.

At the end of the loop are functions that must be called during every iteration. These include functions for writing the new DAC value, updating the LEDs, updating the LCD output (if required by the current menu position), etc.

As mentioned above, the RS-232 receive buffer is intended to be 512 bytes long. It is the responsibility of the computer to insert an instruction that requires return data once every 500 bytes of sent data. This makes sure that the buffer is empty before sending the next 500 bytes.

Also, to prevent a stall in the program, the RS-232 event handler does not wait for the required data of a command before returning control to the core loop. It simply stores its state (similar to the menus) and continues when new data arrives.

8.6.2 Interface

The interface has not been designed, although a menu-driven interface is held in mind. The text to be displayed at each point is stored in EEPROM, rather than hard-coding it. This saves program memory space. It also provides the extra functionality of changing the language. There is enough EEPROM space available to store over 1000 20-character strings. Many languages can thus be stored. The CPU simply adds a constant to the base address in order to retrieve the correct string and copy it to the LCD buffer.

The position in the menu tree is stored in RAM. A large case statement then determines which piece of code must be executed upon entering the event-handler.

Pectora roborant cultus recti

8.7 Summary

The controller hardware has been discussed and is fully operational. The components implemented within the FPGA were also discussed briefly. Some recommendations have been made. These include a FIFO queue for the RS-232 receiver, an external processor, etc.

An imbedded processor has been designed and is fully operational. It can be programmed using the RS-232 port. It is intended for implementation of a menu-driven interface.

Chapter 9

Measurements and Results

9.1 Introduction

In order to verify the theoretical design of the previous chapters, practical experimentation was performed. This chapter describes the conditions under which each experiment was performed and provides the results. A brief interpretation of each result is included.

The design in the previous chapters are for the final product, which differs in some ways to the prototype tested in this chapter. The objectives of the prototype were to verify the concept, rather than provide a very accurate product. It contains less complex hardware and software than the final implementation would.

9.2 Practical Test Setup

9.2.1 Test Magnet

In order to test the design without disassembling the existing setup, an equivalent test magnet was designed and built. Using the relevant technique as described in chapter 4, an inductor was designed. A laminated steel C-core with cross-sectional area 5 472 mm^2 was available. It has two air-gaps, the first is 20.6 mm (the same as the real magnet) and the other 10 mm. The inductor has 1 948 turns with 2.5 mm diameter wire. An external variable resistor was used to achieve the desired 14.4 Ω.

The resulting magnet has a theoretical inductance and field-to-current relationship of 1.27 H and 118 mT·A^{-1} respectively. The inductance was measured as 3.42 H. The magnetic field-to-current relationship was not measured practically.

Figure 9.1: Test magnet

9.2.2 Data Logger

In order to log the performance of the converter, the RS-232 port was used to send a continuous stream of data to the computer. The computer stores these data-points to hard-drive. One frame of data consists of 13 bytes. The first byte is the hexadecimal number 'AB' and is used for synchronisation. The three that follow represent the 24-bit representation of V_1. The most significant byte is sent first. The 9 that follow are the 24-bit representations of I_1, V_2 and I_2 respectively. The 24-bit representations are obtained by implementing a 64-bit smoother for all four data-streams.

A connection speed of 115 200 baud was used. Each byte is 10 bits long (start, stop and data). This implies that the maximum data rate is 886 frames per second. A data rate of 800 frames per second was chosen. Since this is constant, no time-stamp is needed. The computer calculates the time from the known sample rate.

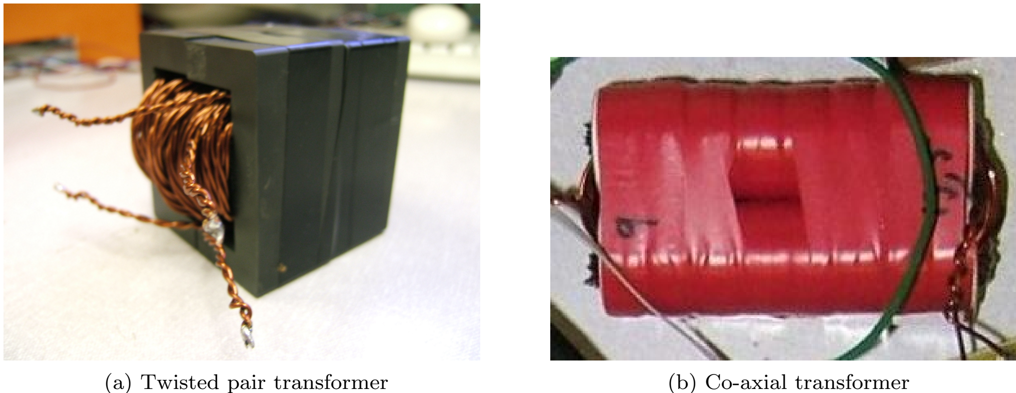
By means of the keypad, another option was made available. Samples are taken at the same sampling rate as the controller and stored in a buffer. When the buffer is full (1024 samples), it is sent to the computer and emptied. Each set of data is preceded by a full-scale frame ('AB' followed by 12 'FF' bytes). This is used to signal the start of a new set to the computer.

9.2.3 Test Conditions

At the time of prototyping, the exact layout of the power PCBs were still unclear. The prototype thus differs from the design in chapter 4. It was found that EMI poses a very large problem. For this reason, a 100 V (line to line) input voltage was used instead of the full 400 V. This implies a much lower power rating and further experimentation should be performed in future work. The full-scale voltages were decreased to 300 V and 240 V for V_1

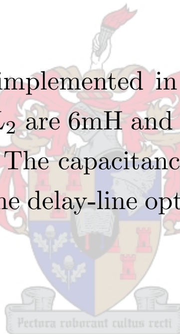
and V_2 respectively.

A co-axial transformer was used in these measurements instead of the twisted-pair transformer as described in section 4.7 on page 42. This was so that the effect of leakage inductance could be observed to a greater extent. The two transformers are shown in figure 9.2.



(a) Twisted pair transformer (b) Co-axial transformer

Figure 9.2: Transformers



The smoothing algorithms were implemented in the data logger, but not used in the control system. Inductances L_1 and L_2 are 6mH and 2 mH respectively. The control system constants were adjusted accordingly. The capacitance values are the same as designed.

All graphs are taken when using the delay-line option. No visible difference was observed when using the noise-shaper instead.

9.2.4 Photos

Figure 9.3 shows the practical setup.

No heat sink is required at the power rating of the tests. The EEPROM and probe circuit has not been implemented on the second iteration PCB yet. The first iteration is shown in figure 9.4.

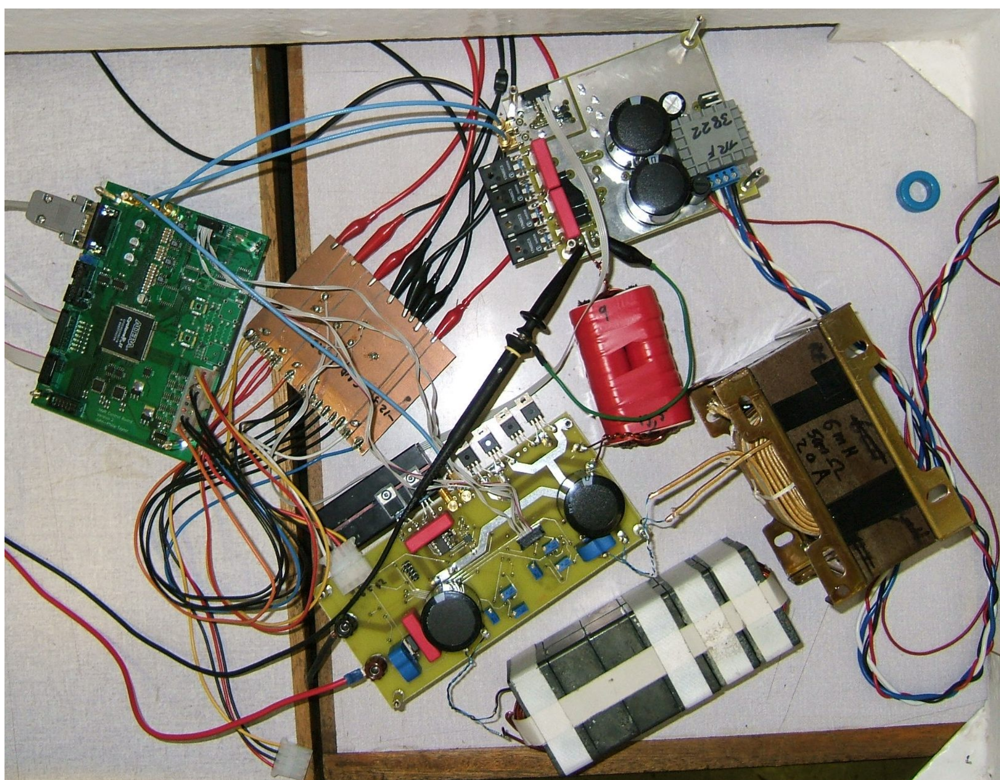


Figure 9.3: Practical setup

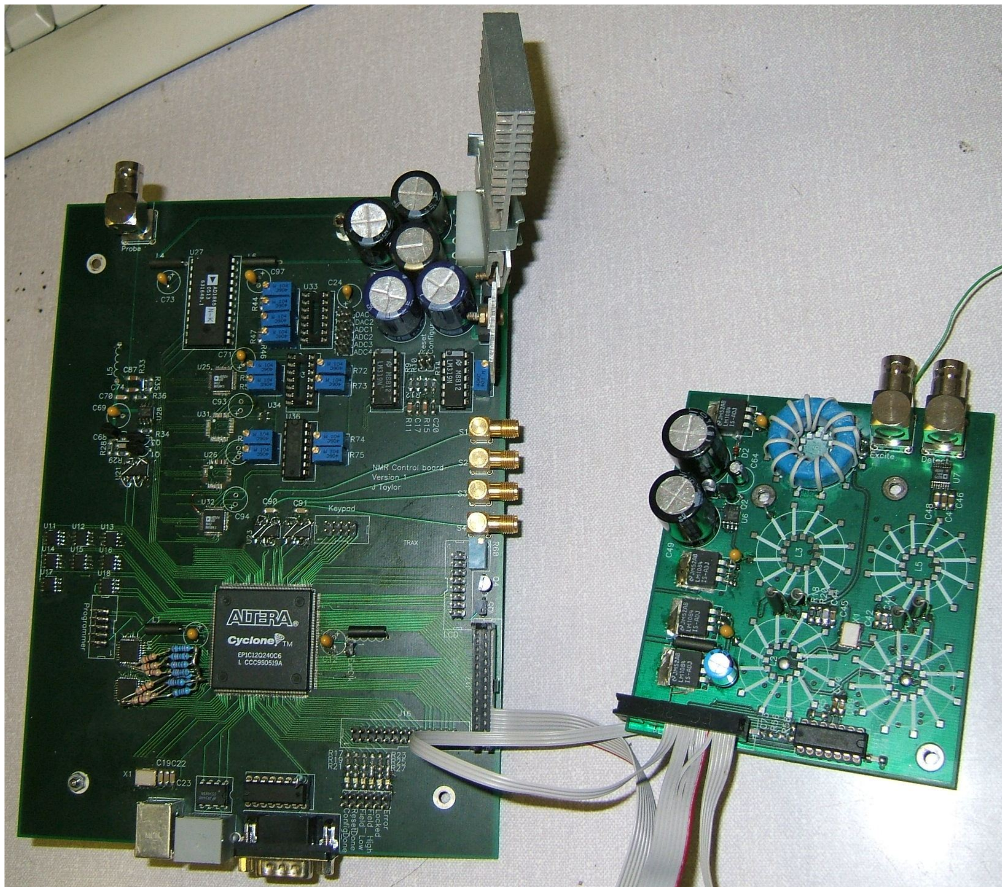


Figure 9.4: First iteration PCB

9.3 Converter Results

9.3.1 Switching Transients

Figure 9.5 shows the measured transformer primary voltage and current. The output current was set on 1 A and and bus voltage on 70 V. A lower bus voltage was used so that the duty-cycle can be larger while keeping the output current at 1 A. When a heatsink is included in future iterations of the project, the full bus voltage can be used with a larger output current instead.

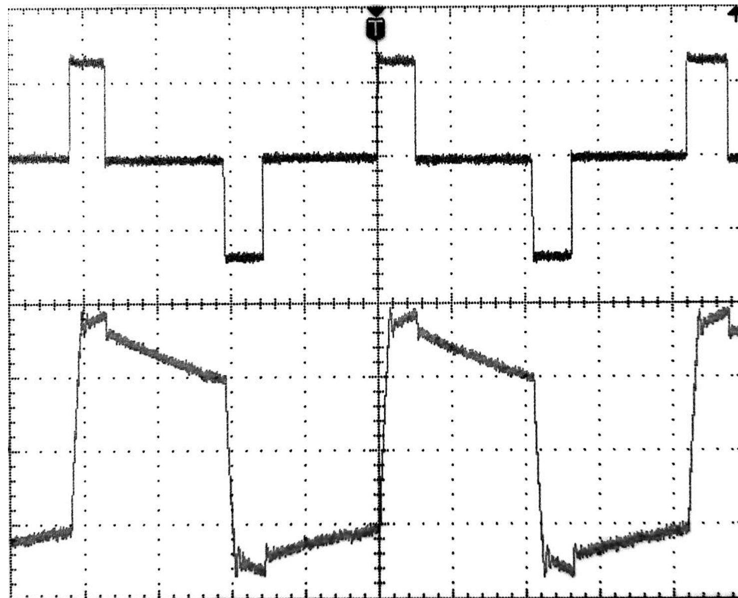


Figure 9.5: Transformer primary voltage and current

Top – Voltage , 50 V/div
Bottom – Current , 1 A/div
Time scale, 10 μs/div

The voltage waveform is as expected and predicted in figure 4.12a on page 35. The current waveform is more interesting. At the start of a voltage pulse, the leakage inductance start to charge. When fully charged, the current is equal to the filter inductor current for the duration of the pulse. At the end of the voltage pulse the leakage inductance charges the parasitic capacitance in the circuit, resulting in the small dip in current magnitude at that point. The remainder of the time spent in a zero voltage state the current circulates through the on-resistances of the MOSFETs. The resulting voltage drop discharges the leakage inductance, as can be seen by the slope during this state. The process repeats itself when the next pulse starts.

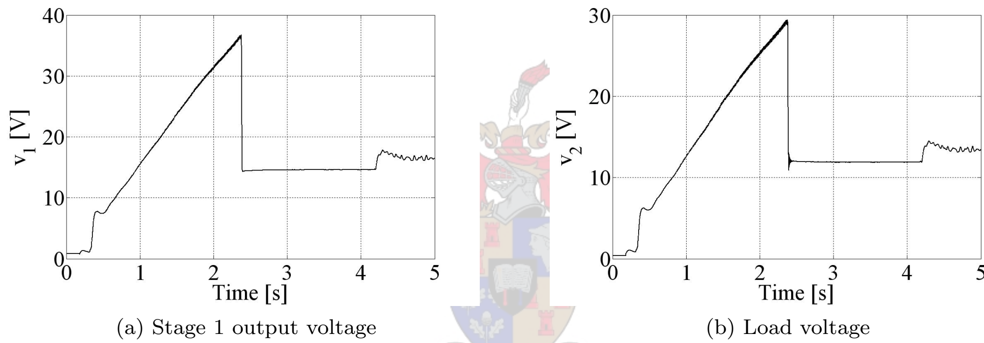
The bandwidth of the current probe used is 1 MHz. The charging slope observed at the start of a voltage pulse should be in the order of 1 μs long due the leakage inductance,

switched current and voltage pulse magnitude combination. This is also the same length expected due to the bandwidth limitation of the probe. The measured slope can therefore not be trusted.

9.3.2 Without Feed-Forward

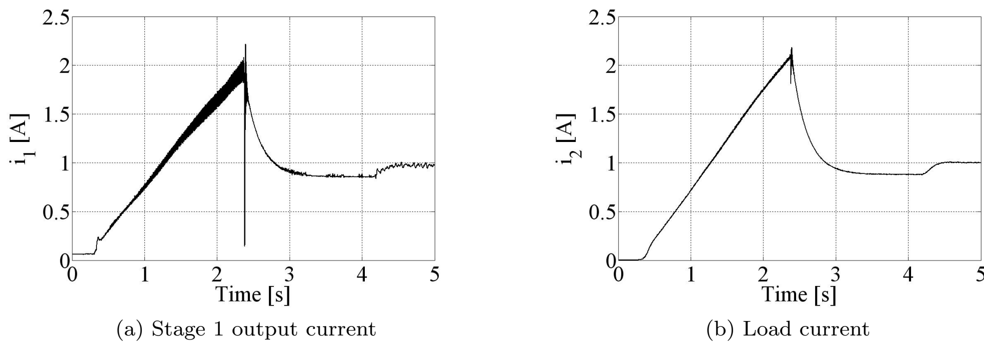
In this section, the second stage duty-cycle is fixed at 80%. The rest of the control system, however, is fully implemented. These results therefore show the effect of not using feed-forward.

The first set of graphs (figure 9.6 and 9.7) shows the system during start-up. The duty-cycles are ramped as described in chapter 5. When the controller takes over at 2.3 s, the duty-cycles are too high for the desired load current of 1 A. The controller successfully rectifies the situation by dropping the output voltage of stage 1 to the limited value of 15 V. The load then discharges sufficiently for the integrator to start recovering. At 4.2 s the integrator has recovered and forces the output current to be equal to the reference of 1 A.



(a) Stage 1 output voltage (b) Load voltage

Figure 9.6: Start-up without feed-forward: Voltages



(a) Stage 1 output current (b) Load current

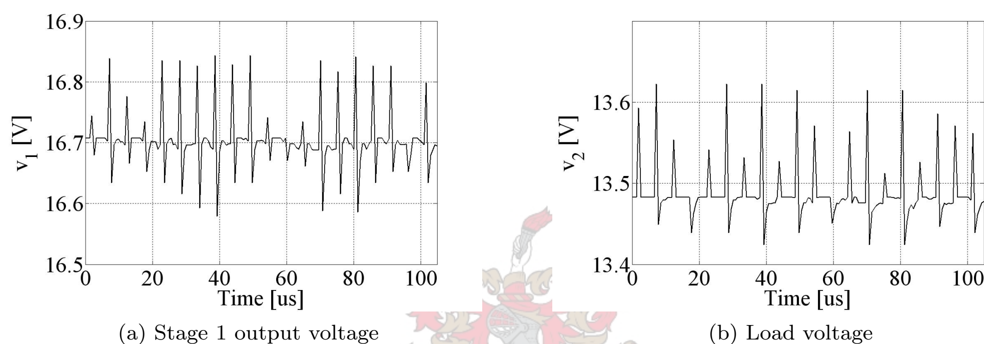
Figure 9.7: Start-up without feed-forward: Currents

The second set (figures 9.8 and 9.9) show the ripple on the measurements. No smoothing was performed on these measurements.

The voltage spikes occur at a frequency of 190.735 kHz, which is half the second stage switching frequency. It is also four times the switching frequency of stage 1. The cause is EMI and can be eliminated by using a more carefully laid out PCB design.

The current ripple on both the stage 1 output current and the load current is 47.684 kHz. This indicates that the very fast second stage switching frequency is filtered out by the L-C filter, whereas the lower stage 1 frequency is not. The final feed-forward implementation should solve this problem.

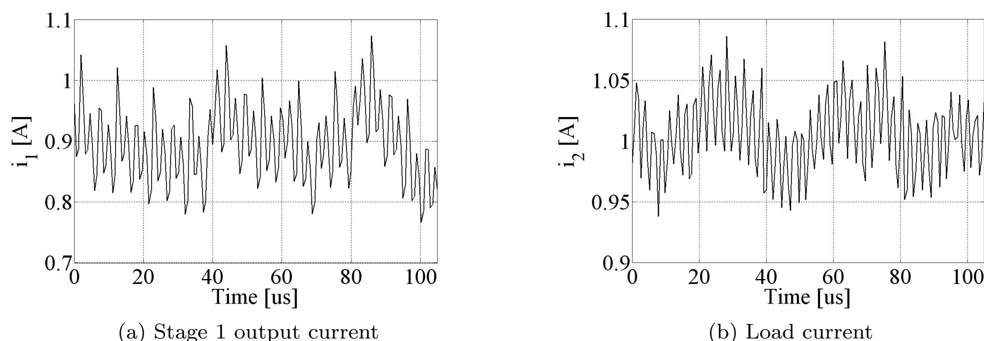
The noise is due to the amplification of the small LEM module output signal. A better quality amplifier positioned physically closer to the module will reduce this noise.



(a) Stage 1 output voltage (b) Load voltage

Figure 9.8: Ripple without feed-forward: Voltages

Pectora roborant cultus recti

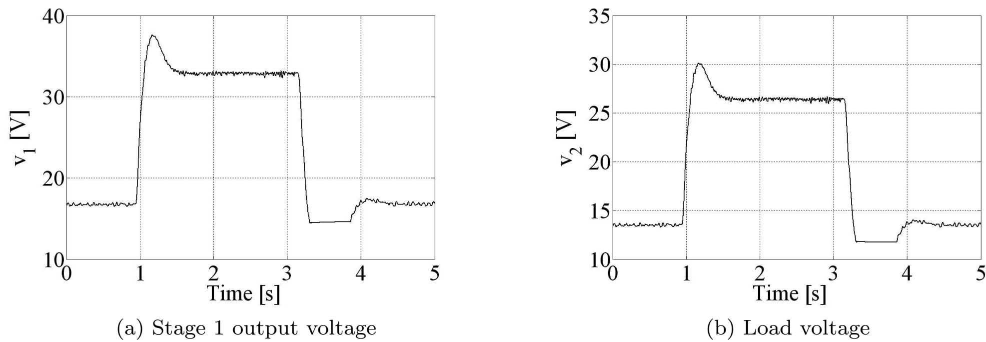


(a) Stage 1 output current (b) Load current

Figure 9.9: Ripple without feed-forward: Currents

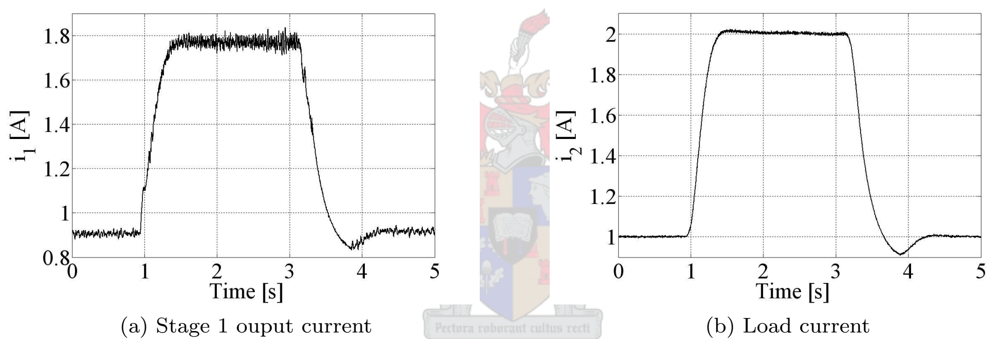
The response to a 1 A to 2 A to 1 A step reference is provided by figures 9.10 and 9.11. The result is as predicted by simulation in figure 5.19 on page 78. The current shows a first-

order response, as was designed. The overshoot on the downward step is due to the voltage saturating. Step responses up to 4 A were also tested and produced similar waveforms.



(a) Stage 1 output voltage (b) Load voltage

Figure 9.10: Step response without feed-forward: Voltages



(a) Stage 1 ouput current (b) Load current

Pectora roborant cultus recti

Figure 9.11: Step response without feed-forward: Currents

The primary bus voltage was stepped from 200 V to 140 V to 200 V. The resulting output current is shown in figure 9.12. The waveform is similar to the simulation results in figure 5.28 on page 81. The difference in amplitude may be accounted for by the different control constants.

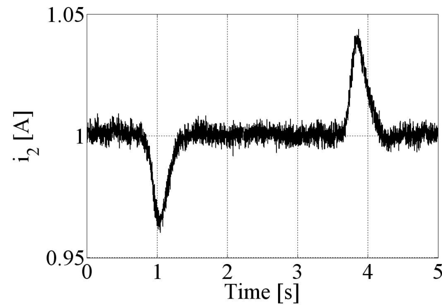
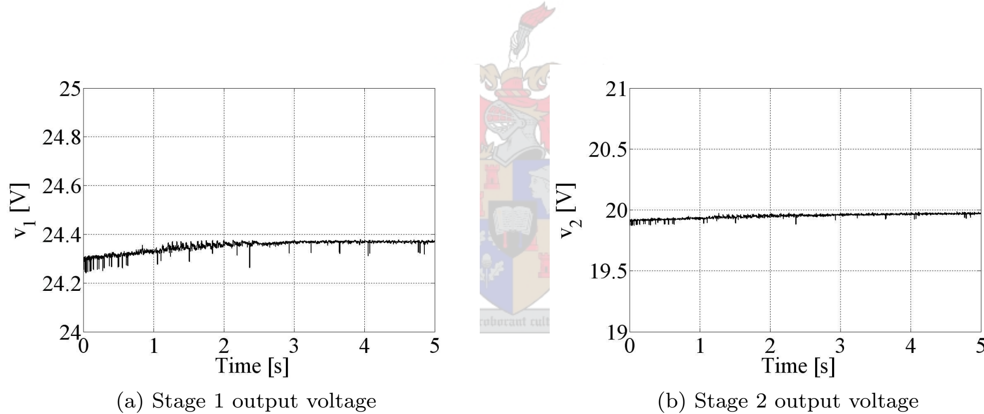


Figure 9.12: Step disturbance without feed-forward

9.3.3 With Feed-Forward

The feed-forward strategy was tested first by providing a fixed v_3 reference of 20 V. The resulting voltages are shown in figure 9.13. As expected, the first stage output voltage settled on some value close to 25 V. It is lower than expected due to the line voltage being lower. The second stage provided the 20 V by means of feed-forward, cancelling out the effect of the lower line voltage. It is not exactly 20 V due to non-linear properties in the converter.



(a) Stage 1 output voltage (b) Stage 2 output voltage

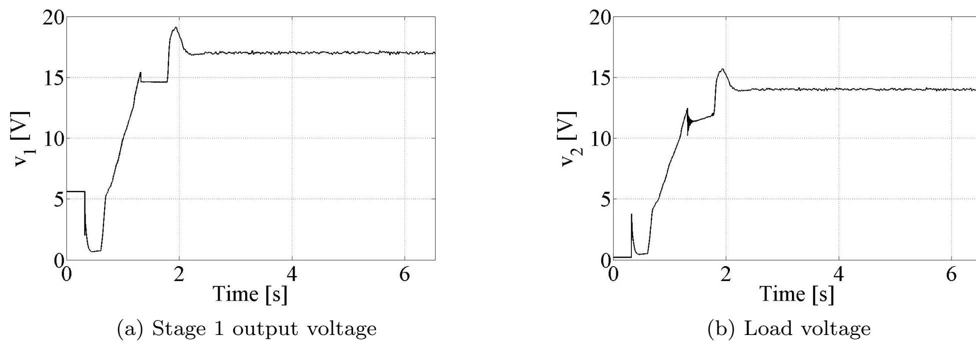
Figure 9.13: Feed-forward voltage reference

The system was then started, as before, with ramped duty-cycles. The large overshoot is not present because the duty-cycles do not ramp as high is in the previous section. The resulting graphs are shown in figures 9.14 and 9.15.

It this case, the stage 1 filter capacitor was charged to 6 V at the start of the sequence. The second stage duty-cycle switched on at 0.3 s, discharging the capacitor into the second stage inductor. The current spike is clearly visible at 0.3 s. The effect of the ramping stage 1 duty-cycle is observed late due to the large stage 1 L-C filter.

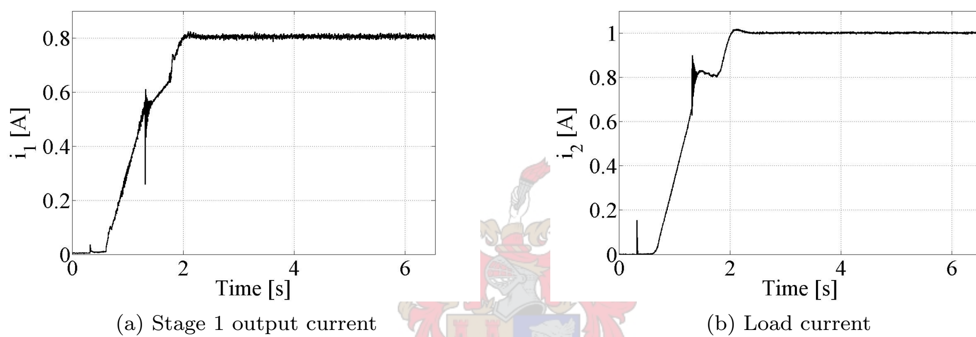
The control system takes over at 1.35 s, where a small glitch is noticeable. The integrator takes effect at 1.8 s. The load current is settled on 1 A after 2.4 s.

The response to a 1 A to 2 A to 1 A to 3 A reference is shown by figures 9.16 and 9.17.



(a) Stage 1 output voltage (b) Load voltage

Figure 9.14: Start-up with feed-forward: Voltages

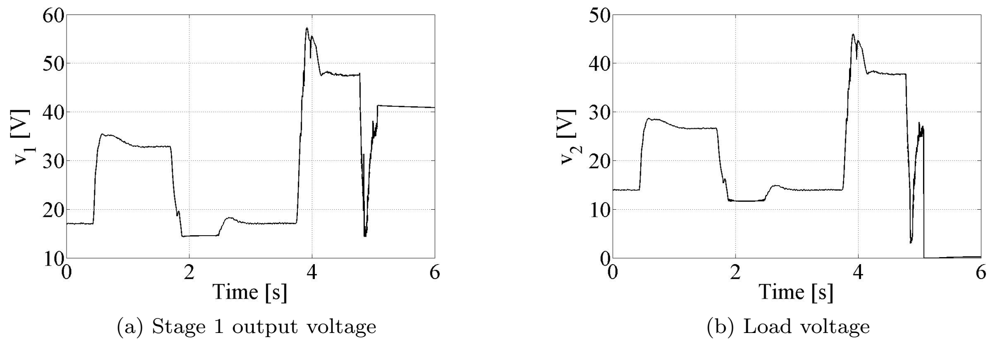


(a) Stage 1 output current (b) Load current

Figure 9.15: Start-up with feed-forward: Currents

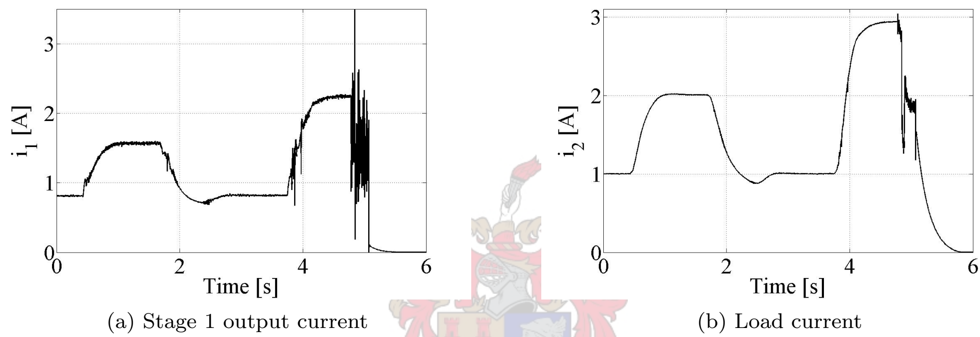
The 2 A step response is as expected, but the feed-forward seems to fail on the downward step. Both the first and second stage voltages show a flat area from 1.8 s to 2.4 s. Only the first stage voltage should be flat. The second stage voltage should be able to go down all the way to zero. This indicates a mistake in the VHDL code.

While trying to respond to the 3 A step reference, the output current levels off too early. It seems to not be able to reach the required 3 A. This once again indicates an error in the VHDL code, since the converter could reach 4 A without a problem when feed-forward was disabled. The error will be investigated in future work.



(a) Stage 1 output voltage (b) Load voltage

Figure 9.16: Step response with feed-forward: Voltages



(a) Stage 1 output current (b) Load current

Figure 9.17: Step response with feed-forward: Currents

9.4 Probe Measurements

9.4.1 Functionality

At the time of building the probe circuit, the analogue switch was excluded. Apart from this, the circuit is fully functional. The gain from DDS_1 (the excitation signal) to the output of the excitation mixer is 9 dB. The gain from the detection signal to the detection mixer output is 32 dB. Both of these are taken when the output of DDS_2 (the mixing signal) is at full-scale.

9.4.2 Circuit Delays

A 4.125 m cable was used from the controller PCB to the probe and a 1.5 m cable returning to the controller. The total delay in the cable thus 28.125 ns, assuming that the speed of the signal is $2{\cdot}10^8$ m·s^{-1}. Table 9-I shows the results of connecting the two cables together directly. Table 9-II shows the results of including the probe in the path.

These phase delays can easily be stored within a look-up table and simply subtracted

Table 9-I: Circuit delay (controller side)

Frequency [MHz]	**Phase (lagging) [deg]**	**Delay [ns]**	**Circuit delay [ns]**	**Circuit phase shift [deg]**
5	55.7	30.94	2.815	5.067
10	111.0	30.83	2.705	9.738
20	218.0	30.28	2.155	15.516

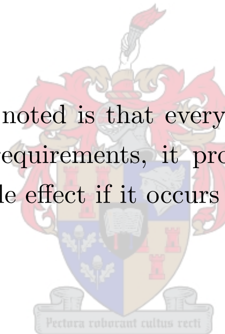
Table 9-II: Circuit delay (Probe side)

Frequency [MHz]	**Phase (lagging) [deg]**	**Delay [ns]**	**Probe delay [ns]**	**Probe phase shift [deg]**
5	57.1	31.72	0.78	1.404
10	124.0	34.44	3.61	12.996
20	274.7	38.15	7.87	56.664

from the measured phase in order to obtain the actual phase.

9.4.3 Glitches

One interesting observation to be noted is that every time the DDS is updated with new frequency, phase and amplitude requirements, it produces a small glitch on the output waveform. This glitch has negligible effect if it occurs only once every 10 μs.



9.5 Summary

The converter was tested, but at a lower line-voltage. The control system is stable and performs well when feed-forward control is disengaged. When feed-forward is used, the system becomes unstable for large output currents. An error within the VHDL code is suspected, but the exact cause is unclear and left to future investigation.

The probe circuit hardware was tested successfully, but no NMR effects were measured.

Chapter 10

Conclusions and Future Work

10.1 Introduction

This final chapter provides a summary of research findings, possible improvements for the next iteration and future research topics.

10.2 Research Findings

10.2.1 Integration

All the digital control components of a typical modern magnet power supply were successfully integrated within a single FPGA. These include the control system, probe and interface.

Massive parallelisation within an FPGA enables the different digital components to execute at speeds. The implementation of very fast control schemes is possible. Implementing a system with 100 kHz bandwidth and a 100 ns time-step is no problem, provided the ADC is fast enough. This eases the migration process from a continuous- time design to a digital implementation.

10.2.2 Converter

In order to realise the precision required over the full output range, a dual stage converter topology has been designed and implemented. The first stage successfully adjusts the input to the second stage to 125% of the desired output voltage. From there the second stage easily achieves the desired 10 ppm precision and stibility.

10.2.3 Digital Control

The only problem with digital control for high-precision systems is the resolution of the ADCs. Resolution may be increased, but this is left to future research. The desired 10 ppm stability was not obtained using 18-bit ADCs, although the principle is verified.

Feed-forward control provides an elegant way of increasing the rejection bandwidth of a low-bandwidth control system. The input disturbance is neutralised before it causes an effect on the output. Feedback control, in contrast, waits for the effect on the output to be present before it can remove it. A feed-forward control strategy was successfully implemented.

10.2.4 Digital PWM

Two methods of increasing the resolution of digitally generated PWM was investigated. It was found that the noise shaper reigns superior over the delay-line. Some of the advantages of the noise shaper are included in the following list:

- The ability to use any clock-frequency (as apposed to one dependant on the delay-line interval)
- Superior resolution (>18-bit) at a high switching frequency (>350 kHz)
- No external components are required
- Ease of implementation (a simple digital differentiator chain is required with unity gain)
- The high resolution required from the digital PWM (>18-bit) is easily reached with a 4th order noise shaper and 8-bit PWM generator. This is true provided that the switching frequency is at least 36 times higher than the frequency at which the resolution is required.

10.2.5 Probe

A new NMR probe scheme was proposed, designed and simulated. The Lamour frequency is followed by means of a phase-locked loop while the sample is kept in a continuously resonant state. The excitation signal is generated digitally and therefore the frequency of resonance is immediately available. Also, it is best to use pure water as a sample so that the time-constants T_1 and T_2 are as large as possible. This solves the problem of crystallisation.

Measurement rate may be increased to 100 ksps. A magnetic field-strength ramp of 10 $\text{T}\cdot\text{s}^{-1}$ can be followed with negligible error. Direct NMR-measured field-strength feedback is thus practical. Much research and development remains, however.

10.3 Improvements and Future Work

10.3.1 Converter

The soft-start may be implemented differently. The three triacs and diode-bridge may be replaced with the circuit in figure 10.1. This prevents unnecessary use of three components. The soft-charge resistor bypass MOSFET may also be replaced with a triac. The driving circuit is then much simpler.

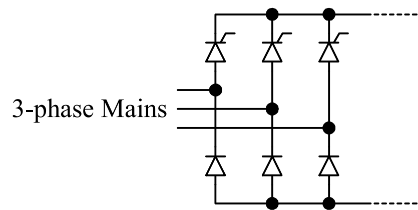


Figure 10.1: Proposed soft-start circuit

Another improvement would be to implement the second stage with a full-bridge converter. The project would then be more versatile by providing bi-directional output current.

10.3.2 Control System

With greater insight into the requirements of analogue to digital conversion it is possible to optimise the choice of IC. Current i_2 must have a higher resolution, but does not require a high sample rate. Current i_1 and voltage v_2 require a high sample rate, but not necessarily a good resolution. Voltage v_1 requires superior resolution and sample rate, and might be improved by the addition of a digital potentiometer. The input to the ADC can then be enlarged for smaller signals, improving their resolution while still using a practical ADC.

The control system implementation can be improved by using a finite state machine instead of an asynchronous implementation. The same few multipliers may then be used to implement many gain blocks. Higher resolution is then possible, through the availability of more multipliers.

An estimator (Kalman filter [26]) can be implemented to improve the quality of measurement. This means that the only two high-resolution measurements are v_1 and the flux density. A low-resolution measurement of v_2 may be used to characterise the load.

10.3.3 PWM Generator

Future research in the area of PWM generation could include the optimisation of the noise-shaper. A lower-order noise-shaper may provide satisfactory results.

10.3.4 Probe

Much research is outstanding regarding the probe. The phase-locked loop must be optimised (by modifying $D(z)$) in order to follow a faster-changing flux density.

Resonance may be maintained by means of an analogue circuit (as explained in chapter 7). This simplifies the task of the FPGA, but remains to be verified practically.

Also, the probe circuit may be considerably simplified (as explained in section 7.4), by implementing the DDS within the FPGA. Problems associated by using a square waveform, such as harmonic content and sampling effects, remains to be investigated experimentally.

10.3.5 Controller

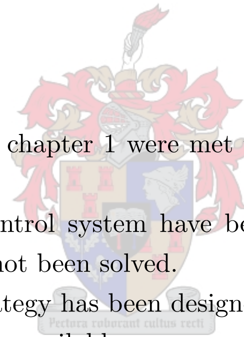
The most obvious improvement is the use of a power supply that requires less quiescent current. This would decrease overall losses.

The analogue signals can be sent to the controller board using differential signals. Advantages include the absence of ground-loops, greater noise immunity and the ability to use a standard RJ-45 connector with CAT-5 network cable. ADCs generally have differential inputs, so no further conversion would be required. The use of an external processor has numerous advantages and must be considered strongly for the next iteration, as mentioned in section 8.5.1.

10.4 Learning Experience

Much has been learned outside the scope of this project. Many mistakes were made during the planning phase and skills in this area have improved greatly. The overall scope of the project was found to be too great for a single inexperienced person to complete fully in two years. The experience gained, however, would without doubt prove invaluable in future projects.

10.5 Summary



The research objectives set out in chapter 1 were met theoretically. Practical verification, however, is not complete.

The implementation of the control system have been verified successfully. Problems associated with EMI effects have not been solved.

A new NMR measurement strategy has been designed and verified successfully through simulation. No practical results are available.

As a final conclusion, the project was greatly enjoyed and much valuable experience has been gained.

Bibliography

[1] Pozar, D.M.: *Microwave and RF design of Wireless Systems*. John Wiley & Sons, 2001.

[2] Serway, R.A. and Beichner, R.J.: *Physics for Scientists and Engineers with Modern Physics*. 5th edn. Prentice Hall, 2000.

[3] Hassanzadegan, H.: Digital Regulation of Accelerator Power Supplies. Available at: `http://www.epn-online.com/popup.php?content=page&page=11401&print=1`, [25 February 2005].

[4] Liu, K. and Fann, C.: Reducing Output Current Ripple of Power Supply with Component Replacement. In: *Proceedings of EPAC*. Lucerne, Switzerland, 2004.

[5] Haus, H.A. and Melcher, J.R.: *Electromagnetic Fields and Energy*. Prentice Hall, 1989.

[6] Mohan, N., Undeland, T.M. and Robbins, W.P.: *Power Electronics*. John Wiley & Sons, 2003.

[7] Starkey, J.R. and Irving, C.F.: *NMR Analysing Magnet Controller MK II*. NECSA, 1978. Van der Graaff Accelerator Document Archive: EL 455 I/TN/35/78.

[8] Liang, R. and Dewan, S.B.: A Low Ripple Power Supply for High-Current Magnet Load. *IEEE Transactions on Industry Applications*, vol. 30, no. 4, pp. 1006–1015, July/August 1994.

[9] Liang, R. and Dewan, S.B.: Modeling and Control of Magnet Power Supply System with Switch-Mode Ripple Regulator. *IEEE Transactions on Industry Applications*, vol. 31, no. 2, pp. 264–272, March/April 1995.

[10] Qi, X. and Xu, Z.: Resonant Magnet Power Supply System for the Rapid Cycle Synchrotron of Chinese Spallation Neutron Source. In: *Proceedings of APAC*. Gyeongiu, Korea, 2004.

[11] Bellomo, P., Berndt, M., de Lira, A., Leyh, G., Lipari, J.J., Rafael, F. and Widmeyer, M.: Spear 3 DC Magnet Power Supplies – an Overview. In: *Proceedings of APAC*. Gyeongiu, Korea, 2004. SLAC-PUB-10383.

[12] Wang, J.: A New Current Regulator for the APS Storage Ring Correction Magnet Bipolar Switching Mode Converters. In: *Proceedings of EPAC*. Lucerne, Switzerland, 2004.

[13] Jacobs, D.: *Digital Pulse Width Modulation for Class-D Audio Amplifiers*. Master's thesis, Electronic Engineering, University of Stellenbosch, Stellenbosch, South Africa, 2006.

[14] Franklin, G.F., Powell, J.D. and Emami-Naeini, A.: *Feedback Control of Dynamic Systems*. 3rd edn. Addison-Wesley Publishing, 1994.

[15] The Hall Effect. Available at: `http://www.tf.unikiel.de/matwis/amat/mw2_ge/kap_2/backbone/r2_1_3.html`, [17 August 2005].

[16] Beltrán, D., Bordas, J., Campmany, J., Molins, A., Perlas, J.A. and Traveria, M.: An Instrument for Precision Magnetic Measurements of Large Magnetic Structures. *Nuclear Instruments and Methods in Physics Research, Section A 459*, pp. 285–294, 2001.

[17] Cowan, B.: *Nuclear Magnetic Resonance and Relaxation.* Cambridge University Press, 1997.

[18] Fei, X., Hughes, V.W. and Prigl, R.: Precision Measurement of the Magnetic Field in Terms of the Free-Proton NMR Frequency. *Nuclear Instruments and Methods in Physics Research, Section A 394*, pp. 349–356, 1997.

[19] Meller, R.E. and Hartill, D.L.: Pulsed NMR magnetometers for CESR. *Proceedings of the Particle Accelerator Conference*, vol. 4, pp. 2339 – 2341, 2003.

[20] Prigl, R., Haeberlenb, U., Jungmann, K., Putlitz, G.Z. and von Walter, P.: A High Precision Magnetometer Based on Pulsed NMR. *Nuclear Instruments and Methods in Physics Research, Section A 374*, pp. 118–126, 1996.

[21] Taylor, J., Mouton, H. du T. and Jones, T.: NMR-Controlled Analysing Magnet Power Supply. In: *Proceedings of SAUPEC.* Durban, South Africa, 2006.

[22] Flowers, J.L., Franks, P.W. and Petley, B.W.: Correcting High Precision Pulsed NMR Flux Density Measurements for Asymmetries in the Lineshape. *IEEE Transactions on Instrumentation and Measurement*, vol. 44, no. 2, pp. 488–490.

[23] Nilsson, J.W. and Riedel, S.A.: *Electric Circuits.* 6th edn. Prentice-Hall, 2000.

[24] Sensitivity of Human Ear. Available at: `http://hyperphysics.phy-astr.gsu.edu/hbase/sound/earsens.html`, [13 February 2008].

[25] Wei, L., Lipo, T.A. and Chan, H.: Matrix Converter Topologies With Reduced Number of Switches. In: *Conference Record of the 20th WEMPEC Anniversary Meeting.* 2001.

[26] Phillips, C.L. and Nagle, H.T.: *Digital Contol System Analysis and Design.* 3rd edn. Prentice Hall, 1995.

[27] Spiegel, M.R. and Liu, J.: *Mathematical Handbook of Formulas and Tables.* 2nd edn. McGraw-Hill, 1999.

[28] Peebles Jr, P.Z.: *Probability, Random Variables and Random Signal Principles.* 4th edn. McGraw-Hill, 2001.

[29] Proakis, J.G. and Manolakis, D.G.: *Digital Signal Processing.* 3rd edn. Prentice-Hall, 1996.

[30] Zwolinski, M.: *Digital System Design with VHDL.* 2nd edn. Prentice Hall, 2004.

[31] Hanselman, D. and Littlefield, B.: *The Student Edition of MATLAB Version 5 User's Guide.* PWS Publishing Company, 1997.

[32] Reader, H.C.: Vector network analyser circuit. Personal conversation, October 2006.

[33] On Semiconductor: 3.3 V ECL Programmable Delay Chip. Available at: `http://www.onsemi.com/pub/Collateral/MC10EP195-D.PDF`, [21 February 2005].

[34] Neamen, D.A.: *Electronic Circuit Analysis and Design.* 2nd edn. McGraw-Hill, 2001.

[35] Microstrip Trace Impedance Calculator. Available at: `http://www.emclab.umr.edu/pcbtlc/microstrip.html`, [15 October 2005].

[36] Wolf, W.: *Computers as Components.* Morgan Kaufman Publishers, 2001.

[37] Goodrich, M.T. and Tamassia, R.: *Data Structures and Algorithms in JAVAtm.* 2nd edn. Wiley Publishers, 2001.

[38] Levitt, M.H.: *Spin Dynamics.* John Wiley & Sons, 2001.

Appendix A

Specification

A.1 Introduction

The specification listed in this appendix was presented to NECSA and approved by Dr Franklyn.

A.2 Input From User

A.2.1 Analogue

- No analogue input is required from the user

A.2.2 Digital

- Source of digital input
  - 4 x 4 Keypad
  - RS232 (Standard DB-9 connector, 115200 baud)
  - USB (Standard square USB connector) - Optional
  - RS485 (Standard RJ-45 connector) - Optional
- Required flux density
  - 0.1 T to 1.2 T, 1 μT steps
- Manual current override
  - 1 A to 15 A, 10 μA steps

A.3 Actions

- The system must ensure that the current through the magnet is such that the resulting field has the desired flux density.
- The flux density must then be kept stable with fluctuations less than 10 ppm.
- In the case of manual current override, the current must be kept stable with fluctuations less than 500 ppm.
- The system must settle in less than 10 s upon switching on.
- The system must settle in less than 5 s upon any step change in required flux density.

A.4 Output to User

A.4.1 Analogue

- Clock used for calibration (Standard 50 Ω BNC connector)
- Magnet current (Analogue meter)
- Magnet voltage (Analogue meter)

A.4.2 Digital

- Source of digital output
  - 16 x 4 LCD display
  - RS232 (Standard DB-9 connector)
  - USB (Standard square USB connector) - Optional
  - RS485 (Standard DB-45 connector) - Optional
- Measured flux density
  - <10 ppm error
- Measured current and voltage
  - <1% error
- Field measurement unit status
  - Is the probe reading within 10 ppm of the real value?
- Field control unit status
  - Is the field locked within 10 ppm?

A.5 Power

- A 400 V, 13 kVA, 4-wire 3-phase connection is available for use by the current source. It is required to comply with ESKOM regulations.

A.6 Dimensions

A.6.1 Probe

- The probe must be 20 x 62 x 270 mm
- The field is perpendicular to the 62 x 270 mm face

A.6.2 Power Supply and Controller

- The power supply must be a standard 19″ rack module, 4 units (7″) high

A.7 Other

- All settings must me non-volatile.
- The NMR controller interfaces with the probe via a single 50 Ω co-axial line, connected via BNC on both ends. This cable is <30 m.
- The 20 A cable from the current source to the magnet is <30 m.
- High frequency (>100 kHz) ripple:
- The voltage ripple over the magnet must be less than 100 ppm
  - The current ripple through the magnet must be less than 10 ppm
  - The magnetic flux ripple of the magnet must not exceed 1 ppm.
  - This is provided that the magnet has an inductance larger that 1 H throughout the frequency range of the ripple.
- The maximum current through the magnet must not exceed 15 A.
- The minimum current required through the magnet when active will not be less than 1 A.
- The operating temperature range must be from 0 °C to 40 °C.

Pectora roborant cultus recti

Appendix B

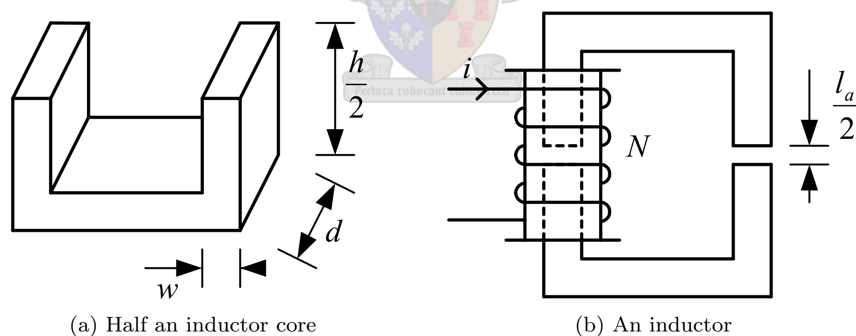
Formula Derivations

B.1 Introduction

This appendix provides the derivation of some of the equations mentioned in the chapters.

B.2 Inductor Design

Consider the inductor core shown in figure B.1a. It is assumed that the core has uniform width w and depth d throughout. Figure B.1b shows how the inductor is physically constructed.



(a) Half an inductor core (b) An inductor

Figure B.1: Inductor dimensions

The magnetic flux lines at the air-gap fringe, increasing the effective cross-sectional area A_a. According to [6], the effecting air-gap cross-sectional area dimensions may be approximated as in figure B.2. The area A_a is therefore given by equation B.1.

$$A_a = (w + \frac{l_a}{2})(d + \frac{l_a}{2}) \tag{B.1}$$

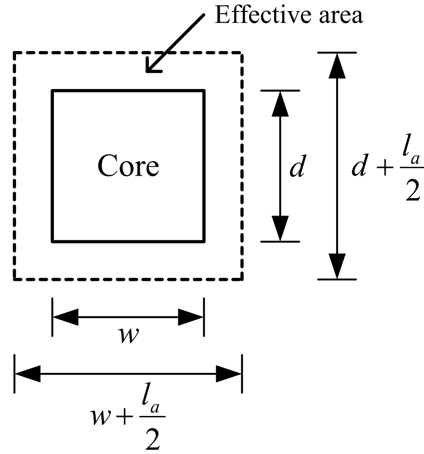


Figure B.2: Effective area of an inductor air-gap

According to magnetic field theory [5], equation B.2 must hold true, where l_c is the effective length of the core.

$$Ni = \phi \sum \frac{l}{\mu A} = \phi \left(\frac{l_c}{\mu_c A_c} + \frac{l_a}{\mu_0 A_a} \right) \tag{B.2}$$

One of the design criteria is that the flux-density within the core must not exceed a given maximum when the rated current flows through the windings. This is provided by equation B.3.

$$B_c = \frac{\phi}{A_c} = \frac{Ni}{A_c \left(\frac{l_c}{\mu_c A_c} + \frac{l_a}{\mu_0 A_a} \right)} \tag{B.3}$$

Pectora roborant cultus recti

Another design criterion is the required inductance. This is derived by equation B.4 and given by equation B.5.

$$v = L\frac{di}{dt} = N\frac{d\phi}{dt} = \frac{N^2}{\left(\frac{l_c}{\mu_c A_c} + \frac{l_a}{\mu_0 A_a} \right)} \cdot \frac{di}{dt} \tag{B.4}$$

$$L = \frac{N^2}{\left(\frac{l_c}{\mu_c A_c} + \frac{l_a}{\mu_0 A_a} \right)} \tag{B.5}$$

By simple manipulation of equations B.3 and B.5, equation B.6 is found.

$$N = \frac{Li_{max}}{A_c B_{c,max}} \tag{B.6}$$

By substituting N into equation B.3 and assuming that $A_a \approx A_c$ (true when $l_a << w$ and $l_a << d$), the air-gap may be found using equation B.7

$$l_a = \left(\frac{Li^2}{A_c B_c^2} - \frac{l_c}{\mu_c} \right) \mu_0 \tag{B.7}$$

The next step is to substitute the closest practical values for l_a and N into equations B.3 and B.5 and finding the best solution iteratively. The same process can be followed in the case of an E-core.

B.3 Transformer Design

The first consideration is to ensure that the core never saturates. A certain maximum flux density B_{max} must not be exceeded. The voltage over the transformer primary is the same as the bus voltage V_{bus}. This voltage may also be given by equation B.8 [6]. After manipulation, the design equation B.9 results. The bus voltage is assumed constant.

$$V_{bus} = NA_{core}\frac{dB}{dt} \tag{B.8}$$

$$N = \frac{V_{bus}}{2B_{max}A_{core}f_s} \tag{B.9}$$

The next consideration is the maximum magnetising current i_m. This is given by equation B.10 [5]. By substituting equation B.9, equation B.11 results.

$$i_m = \frac{Bl}{N\mu} \tag{B.10}$$

$$N = \sqrt{\frac{V_{bus}l_{core}}{2\mu_r\mu_0 A_{core}f_s i_{m,max}}} \tag{B.11}$$

B.4 First Order Ramp Response

Figure B.3 shows an R-L system with feed-back control. It may be described by equations B.12 and B.13.

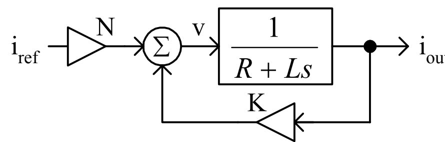


Figure B.3: Single order R-L system with feedback

$$\frac{di_{out}}{dt} = \left(-\frac{R}{L}\right) i_{out} + \frac{v}{L} \tag{B.12}$$

$$v = Ki_{out} + Ni_{ref} \tag{B.13}$$

By designing the system to have a zero steady state error as well as a time-constant τ, equations B.14 and B.15 are obtained.

$$K = -\frac{L}{\tau} + R \tag{B.14}$$

$$N = R - K \tag{B.15}$$

A general ramp reference is given by equation B.16 where m is the gradient of the ramp and c the initial value. For a ramp reference I_0 equals c. For a step reference, however, c is the final value of the step, I_0 the initial value and m is zero. Equation B.17 result from this reference.

$$i_{ref} = mt + c \tag{B.16}$$

$$\frac{di_{out}}{dt} = \left(-\frac{R-K}{L}\right) i_{out} + \frac{N}{L}(mt + c) \tag{B.17}$$

Substituting equation B.15 and taking the Laplace transform of equation B.17 yields equation B.18.

$$Is - I_0 = \left(-\frac{N}{L}\right) I + \frac{mN}{Ls^2} + \frac{cN}{Ls} \quad , N = R - K \tag{B.18}$$

Solving for I gives equation B.19. By using a partial fraction expansion (equation B.20) and taking the inverse Laplace transform yields equation B.21.

$$I = \frac{I_0 s^2 + \frac{cN}{L} s + \frac{mN}{L}}{s^2\left(s + \frac{N}{L}\right)} \tag{B.19}$$

$$I = \frac{m}{s^2} + \left(c - \frac{mL}{N}\right) s + \left(I_0 - c + \frac{mL}{N}\right) \frac{1}{\left(s + \frac{N}{L}\right)} \tag{B.20}$$

$$i_{out} = mt + \left(c - \frac{mL}{N}\right) + \left(I_0 - c + \frac{mL}{N}\right) e^{-\frac{N}{L}t} \tag{B.21}$$

By substituting equations B.14 and B.15 into equation B.21, the general expression for the ramp-response for any first-order system is obtained. This is given in equation B.22.

$$i = mt + c - m\tau + (I_0 - c + m\tau)e^{-t/\tau} \tag{B.22}$$

Also, the error signal can be obtained by using equation B.13.

B.5 Fourier Expansions

Equation B.23 shows the equation for rectified 400 V, 50 Hz 3-phase over a single period of rectified waveform.

$$f(t) = 565\cos(100\pi t) \quad , t \in \left(-\frac{1}{600}, \frac{1}{600}\right] \tag{B.23}$$

The Fourier expansion of the expression is given by equations B.24 to B.26 [27].

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \Big(a_n \cos(600n\pi t) + b_n \sin(600n\pi t) \Big) \tag{B.24}$$

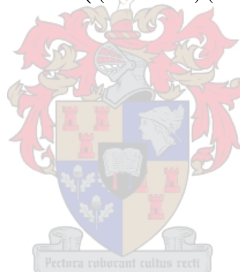
$$a_n = 600 \int_{-\frac{1}{600}}^{\frac{1}{600}} f(t)\cos(600n\pi t)dt \tag{B.25}$$

$$b_n = 600 \int_{-\frac{1}{600}}^{\frac{1}{600}} f(t)\sin(600n\pi t)dt \tag{B.26}$$

By noting that $f(t)$ is symmetrical about the y-axis, b_n is zero for all n. Equation B.25 can easily be solved to obtain equations B.27 and B.28 [27].

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(12 f_L n\pi t) \tag{B.27}$$

$$a_n = \frac{2400\sqrt{2}}{\pi} \left(\frac{(-1)^n(-4)}{(12n-2)(12n+2)} \right) \tag{B.28}$$



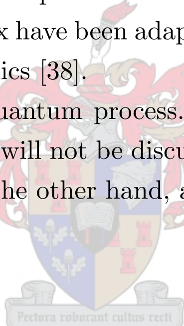
Appendix C

Theory of NMR

C.1 Introduction

In order to design an NMR probe, a deeper understanding of the physical processes must be achieved. This appendix delves into the physics of NMR. It also provides a brief explanation regarding popular measurement techniques. Unless otherwise noted, all equations and explanations included in this appendix have been adapted from Nuclear Magnetic Resonance and Relaxation [17] and Spin Dynamics [38].

On an atomic level, NMR is a quantum process. This detail is not important for the purposes of this thesis, and therefore will not be discussed in much depth. The macroscopic effect and mathematical models, on the other hand, are thoroughly perused.



C.2 Basic Principle

C.2.1 Proton Properties

Protons, like all atomic particles, spin. They also have a significant mass ($1.673{\cdot}10^{-27}$ kg) and charge ($160.218{\cdot}10^{-21}$ C). There are numerous implications that result. First, any massive object that spins has angular momentum. It thus resists changing its axis of rotation. Also, because it is positively charged, there is a circular current flowing around the axis of rotation. This current induces a magnetic moment (μ), given by equation C.1. The constant γ is termed the "gyromagnetic constant" of the particle in question and $\hbar$ the normalised Planck's constant. A graphical representation is presented in figure C.1.

$$\mu = \frac{\gamma\hbar}{2} \tag{C.1}$$

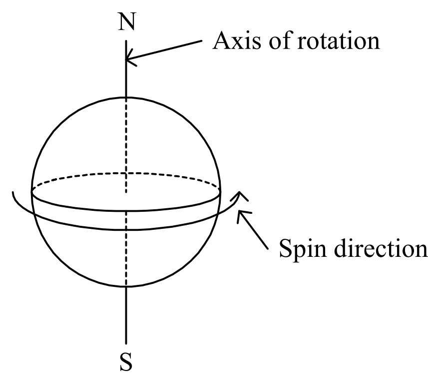
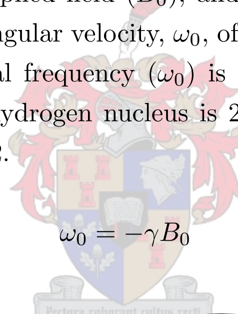


Figure C.1: Proton properties

C.2.2 Precession

When an object with angular momentum (such as a spinning top) experiences a force (such as gravity) it will wobble about the axis of that force. This phenomenon is called precession.

Protons are not influenced by gravity, but their magnetic moments 'want' to align themselves with any externally applied field (B_0), and will induce precessional movement about the axis of that field. The angular velocity, ω_0, of this precessional movement is given by equation C.2. The precessional frequency (ω_0) is often called the Lamour frequency. The gyromagnetic constant of a hydrogen nucleus is 267.522 128 0·10^6 T^{-1}·s^{-1}. A graphic representation follows in figure C.2.



$$\omega_0 = -\gamma B_0 \tag{C.2}$$

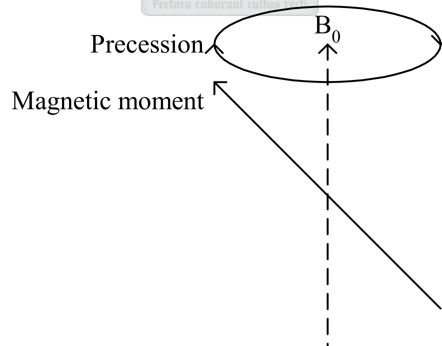


Figure C.2: Precessional movement

Since there are no resistive forces, this movement will continue for as long as the field is applied. On a larger scale these small magnetic moments cancel each other out. It is thus not possible to measure this movement directly.

C.3 Relaxation

C.3.1 Principle

Due to the temperature of the sample along with the presence of so many other particles, the magnetic field experienced by each particle has a random component. This has the effect of biasing the movement of each particle slightly in the direction of the external field. After a certain time, dictated by T_1 (the longitudinal time constant), a magnetisation will develop. Magnetisation is defined as the net magnetic moment per unit volume. This magnetisation, M_0, is given by equation C.3 where N_v is the number of protons per unit volume, k Boltzmann's constant and T the temperature of the sample. For small values of B_0 (<300 kT), equation C.3 is closely approximated by equation C.4. Substituting equation C.1 yields equation C.5. This may be re-written by equation C.6 where a_0 is the proportionality constant.

$$M_0 = N_v \mu \tanh\left(\frac{\hbar\gamma B_0}{2kT}\right) \tag{C.3}$$

$$M_0 = \frac{N_v \mu \hbar \gamma}{2kT} B_0 \tag{C.4}$$

$$M_0 = \frac{N_v \hbar^2 \gamma^2}{4kT} B_0 \tag{C.5}$$

$$M_0 = a_0 B_0 \tag{C.6}$$

C.3.2 Time Constants

The longitudinal time constant T_1 is given by equation C.7. The various constants are given by equation C.8 to C.10. The constant a is the molecular radius of water and η the viscosity.

$$T_1^{-1} = \gamma^2 \langle B_x^2 \rangle J(\omega_0) \tag{C.7}$$

$$J(\omega) = \frac{\tau_c}{1 + (\omega\tau_c)^2} \tag{C.8}$$

$$\tau_c = \frac{a_2}{6D} \tag{C.9}$$

$$D = \frac{kT}{8\pi a \eta} \tag{C.10}$$

In equation C.7, $\langle B_x^2 \rangle$ is the mean square amplitude of the random field B_x (typically $1.3 \cdot 10^{-6}$ T^2 for water at 20 °C). T_1 is approximately 3.6 s for pure water at 20 °C. By the addition of paramagnetic ions the time constant T_1 can be lowered to the order of milliseconds.

T_2 is a similar time constant, approximately 4.7 s for pure water. It is called the transverse relaxation time constant and is given by equation C.11. This time constant dictates the time it takes for a magnetisation vector in precession to decay to have a component only in the direction of the applied field. One can think of each of the particles as a clock, started at the same time, with the second hand representing the direction of the magnetic moment in the x-y plane (the applied field is in the z-direction). After a long time, due to random differences in period, the clocks will differ so much that the average direction of the hands will be zero.

$$T_2^{-1} = \frac{3}{20} b^2 \left\{ 3J(0) + 5J(\omega_0) + 2J(2\omega_0) \right\} \tag{C.11}$$

The constant b is the dipole-dipole coupling constant and is given by equation C.12. The distance between spins r may be greatly increased by the addition of impurities. The time constant T_2 can thus also be reduced to be in the order of milliseconds.

$$b = \frac{-\mu_0 \hbar \gamma^2}{4\pi r^3} \tag{C.12}$$

The other effect of increasing the distance between water molecules is that the hydrogen density decreases. This means a decrease in M_0 and thus a much smaller signal. The detection coil must then be more carefully designed and is often impedance-matched at the point of resonance. Large frequency range is implemented using varactor tuning diodes.

C.4 Precession in an RF Field

C.4.1 Principle

Say the applied magnetic field is still in the z-direction. This would mean that precession can be modelled by a magnetisation component rotating in the x-y plane. The magnetic field about which the magnetisation vector precesses can be altered by the addition of a magnetic field in the x-y plane. When this extra external field rotates about the z-axis, interesting things start to happen - especially when that field rotates at the same speed as the precessional movement.

C.4.2 Bloch Equations

The Bloch equations describe in full the behaviour of the magnetisation vector under an applied magnetic field. They are given by equations C.13 to C.15 and in matrix form in equation C.16. M_x, M_y and M_z are the magnetisation vector ($\mathbf{M}$) in the x-, y- and z-directions respectively. Similarly, B_x, B_y and B_z are the components of the applied magnetic field ($\mathbf{B}$).

$$\frac{dM_x}{dt} = \gamma(M_y B_z - M_z B_y) - \frac{M_x}{T_2} \tag{C.13}$$

Pectora roborant cultus recti

$$\frac{dM_y}{dt} = \gamma(M_z B_x - M_x B_z) - \frac{M_y}{T_2} \tag{C.14}$$

$$\frac{dM_z}{dt} = \gamma(M_x B_y - M_y B_x) - \frac{M_z - M_0}{T_1} \tag{C.15}$$

$$\dot{\mathbf{M}} = \gamma(\mathbf{M} \times \mathbf{B}) - \begin{bmatrix} \frac{M_x}{T_2} \\ \frac{M_y}{T_2} \\ \frac{M_z - M_0}{T_1} \end{bmatrix} \tag{C.16}$$

The cross-product of **M** and **B** indicates that the movement progresses in a circular motion around the axis of **B**. This corresponds to precession. The proportions of **M** that are subtracted indicate movement towards a position in line with **B** and with magnitude M_0. This corresponds to relaxation.

C.4.3 Rotating Frame

In order to analyse the effect of an applied RF field, an axes-transformation may be made. The viewpoint in transformed in such a way as to rotate at the same speed about the z-axis as the applied RF field. This RF field is then seen as a constant field in the x-direction of the new 'rotating frame'.

If the rotating frame rotates at the same rate as the precessional movement, the precessional movement seems non-existent. The same effect can be achieved by lowering the applied magnetic field in the z-direction. The equivalent magnetic field ($B_{z,eq}$) is thus given by equation C.17 where ω_0 is the precession frequency and ω_{rf} the frequency of the rotating frame.

$$B_{z,eq} = \frac{\omega_{rf} - \omega_0}{\gamma} \tag{C.17}$$

This new $B_{z,eq}$ may then be substituted into the above Bloch equations in order to make them valid within the rotating frame. The magnetisation vector now behaves under the influence of the equivalent applied magnetic field ($\mathbf{B}_{eq}$). It now rotates (or precesses) about $\mathbf{B}_{eq}$ with angular velocity given by equation C.18.

$$\omega = -\gamma\,|\mathbf{B}_{eq}| \tag{C.18}$$

C.4.4 Applying the RF Field

From the explanations in the previous sections it may be assumed that a purely rotating RF field is required. Such a field can be generated by using two coils to generate the separate x and y components. The two coils are then driven 90° out of phase in order to create a rotating field in the x-y plane.

There is an easier way though. The effect of an RF field that rotates with the precessional movement lowers $B_{z,eq}$, whereas one rotating in the opposite direction increases $B_{z,eq}$. The

RF field is typically smaller than 100 μT, whereas the applied constant field is typically larger than 1 T. This means that an RF field rotating opposite to the precessional movement has negligible effect.

A pulsing RF field along the x-axis has two rotating components in the x-y plane. These two components rotate in opposite directions. The one rotating in the positive direction may be ignored due to reasons explained in the previous paragraph. The rotating RF field to be considered thus has half the magnitude of the applied pulsed magnetic field.

From this point forward, B_x is the applied RF magnetic field rotating in the negative direction within the x-y plane. Within the rotating frame it is stationary and pointing in the positive x-direction. It has half the magnitude of the applied pulsed magnetic field.

C.4.5 Excitation Bandwidth

When the RF frequency is very close to the Lamour frequency, the magnetisation vector precesses about the x-axis. The idea is thus to deflect the magnetisation vector as much as possible from the z-axis in order to be able to measure it.

The bandwidth of the excitation signal may be defined as the frequency range for which the maximum deflection is within 3 dB of the peak. The peak deflection is when the RF frequency is exactly the same as the Lamour frequency.

One way to approach this problem mathematically is to note that the maximum deflection starts to decrease at the point where $\mathbf{B}_{eq}$ is 45° elevated from the x-y plane. This is when $B_{z,eq}$ is the same as the applied B_x. The bandwidth in that case is given by equation C.19.

$$\beta = 2\gamma B_x \tag{C.19}$$

Outside this narrow band the maximum deviation is given by equation C.20. It can easily be derived by considering the path taken by the magnetisation vector graphically.

$$\Delta M_{xy} = M_0 \sin\left(2\arctan\left(\frac{\gamma B_x}{\omega_{rf} + \gamma B_0}\right)\right) \tag{C.20}$$

The -3 db bandwidth is thus given by equation C.21, which may be approximated as $4.82\gamma B_x$.

$$\beta = \frac{2\gamma B_x}{\tan\left(\frac{\arcsin(10^{-3/20})}{2}\right)} \tag{C.21}$$

C.5 Continuous Wave NMR

C.5.1 Basic Principle

By exciting the sample using a very small signal (in the order of 1 nT) it is possible to induce a steady state condition where the magnetisation vector remains at an angle with the z-axis. This steady state condition is given by equations C.22 and C.23.

$$M_y = \frac{M_0 \gamma B_x T_2}{1 + \gamma^2 B_x^2 T_1 T_2} \tag{C.22}$$

$$M_z = \frac{M_0}{1 + \gamma^2 B_x^2 T_1 T_2} \tag{C.23}$$

The maximum value for M_y occurs when the denominator is equal to 2. At that point the excitation signal B_x is approximately 910 pT and M_y is $0.57 \cdot M_0$. With smaller time constants B_x mast be larger, but the steady state will be reached sooner.

Since M_y is at 90° with the excitation signal B_x, the induced voltage will be in phase with the excitation current. The coil will therefore seem more resistive. This relates to a change in the coil's quality-factor, which can be detected in various ways.

The bandwidth of resonance is very small due to the small excitation field strength. Figure C.3 shows the deviation of the magnetisation vector from the z-axis (ignoring relaxation) with respect to M_0. The applied field is 1 T and B_x 1 nT. Resonance thus occurs only within 45 mHz of the Lamour frequency (at these conditions).

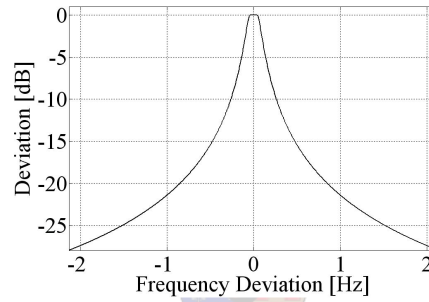


Figure C.3: Continuous wave NMR bandwidth

Pectora roborant cultus recti

C.5.2 Detecting Resonance

The equivalent resistance added to the coil at resonance is given by equation C.24 where i is the excitation current, N the number of turns on the detection coil and A the coil's cross-sectional area. A more convenient form is given by equation C.25.

$$R = \frac{N A \omega_0 \mu_0}{i} M_y \tag{C.24}$$

$$R = \frac{0.57 N A \omega_0^2 \mu_0 a_0}{i \gamma} \tag{C.25}$$

Typical values include: N = 30 turns; A = 100 mm^2; ω_0 = $200 \cdot 10^6$ s^{-1}; i = 2 μA. This yields a resistance of 527 Ω, which is easily detectable through impedance measurement.

The problem with this setup is that the time constants are still in the order of seconds. This would not be a problem if the measurement could be made continuously. The measurement cannot be made continuously as the NMR signal is much smaller than induced voltage due to the coil inductance.

The solution is to sweep either the frequency (ω_{rf}) or B_0 back and forth about the resonance point. This brings the coil in and out of resonance repetitively. The advantage of this is that AC-coupled amplifiers may be used to amplify the NMR signal and discard the induced voltage due to inductance. The disadvantage is that the time constants must be decreased. This, as explained earlier, makes the NMR signal even smaller and thus more difficult to detect. The detection coil must be carefully tuned using adjustable capacitors or tuning diodes.

C.5.3 Field Measurement

The field strength is measured by determining the frequency of resonance. When the field strength is swept and the frequency adjusted so that resonance occurs at the mid-point of the field sweeping waveform, the frequency can be determined by means of a frequency counter. Another alternative is to use a phase-locked loop in generating the RF signal. The programmable counter value is then used to calculate the frequency.

C.6 Pulsed NMR

C.6.1 Basic Principle

The idea of pulsed NMR is to apply the RF field only long enough for the magnetisation vector to rotate through 90°. In the stationary frame, the magnetisation vector then rotates in the x-y plane and induces a current in the detection coil at exactly the Lamour frequency. By determining this frequency the field may be measured.

The time for which the RF field must be applied is given by equation C.26. An RF pulse of this length is called a $\pi/2$ pulse.

$$\Delta t = \frac{\pi}{2\gamma B_x} \tag{C.26}$$

C.6.2 Determining the Frequency

Figure C.4 shows the maximum deviation of the magnetisation vector from the z-axis with respect to M_0. The applied magnetic field is 1 T and B_x is 100 μT. In order for the $\pi/2$ pulse to result in a 90° rotation, the RF frequency must be within 4.26 kHz of the centre frequency (in this case).

Determining the frequency at which to excite the sample is easy once the field-strength is known, but the whole point is to measure the field strength using that frequency. Using a calibrated current measurement to determine the field strength is only accurate to within 0.1%. The resonance point must thus be found by trail-and-error.

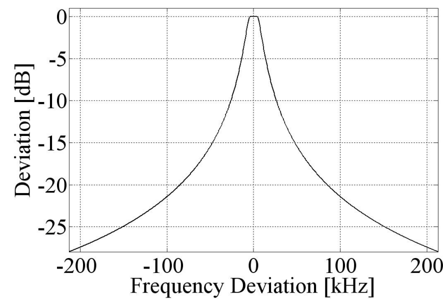


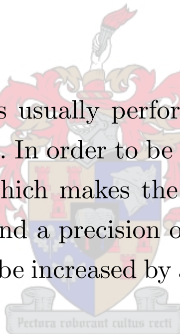
Figure C.4: Pulsed NMR bandwidth

A $\pi/2$ pulse at the above conditions is 60 μs long. Assuming 40 μs are needed to determine whether resonance occurred or not, it would take 10 ms to sweep through 100 frequencies. Assuming they are spaced at 50 ppm of the resonance frequency, the field strength needs to be known only to 0.5%. The calibrated current measurement would thus be sufficient. Once the frequency is known, a phase-locked loop may be used to follow it permanently.

C.6.3 Field Measurement

The measurement of field strength is usually performed by either frequency-counting the detected signal of calculating its FFT. In order to be more accurate the NMR signal is often mixed down to a lower frequency, which makes the measurement more immune to noise. Measurements accurate to 0.1 ppm and a precision of 0.01 ppm are possible [20].

Measurement repetition rate may be increased by adding paramagnetic ions to the water-sample, as discussed earlier.



C.7 Summary

An in-depth study of the physics of NMR has been performed. It is possible to manipulate the magnetisation vector by applying an external oscillating magnetic field. Two popular methods of determining the field strength has also been discussed. These are continuous-wave NMR and pulsed NMR. Pulsed NMR is the more accurate, but both have a limited measurement repetition rate.

Appendix D

Circuit Diagrams

D.1 Introduction

This appendix presents the circuit diagrams designed for the prototype. The controller was built and tested as it is presented herein. A simplified version of the converter was built.

D.2 Controller

Figure D.1 shows the DDS circuitry, figure D.2 the analogue components, figure D.3 the FPGA and figure D.4 the other components.

D.3 Converter

The converter primary is presented in figure D.5 and the secondary in figure D.6.

D.4 Probe

The probe circuit is presented by figure D.7.

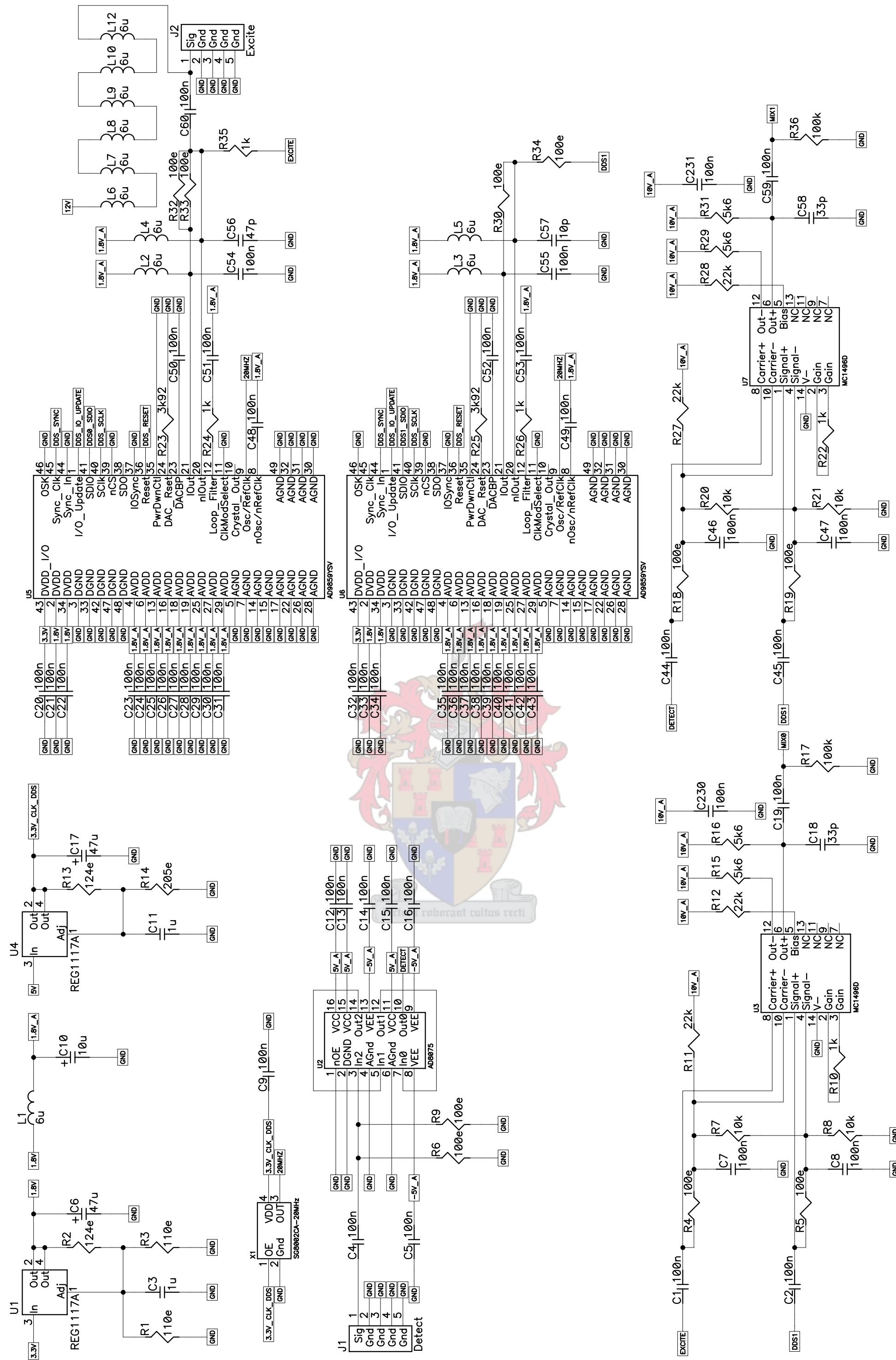


Figure D.1: DDS circuit diagram

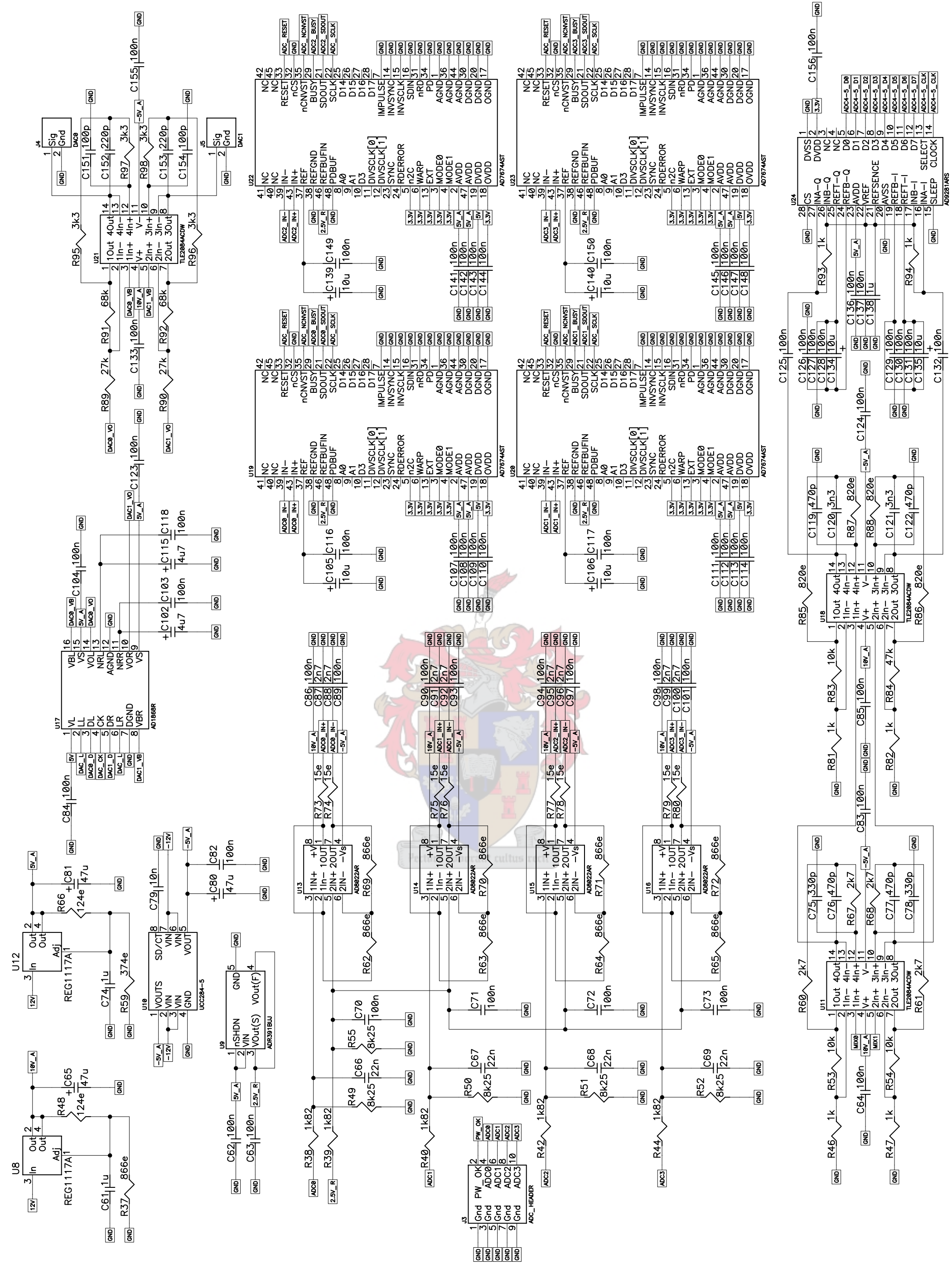


Figure D.2: Analogue circuit diagram

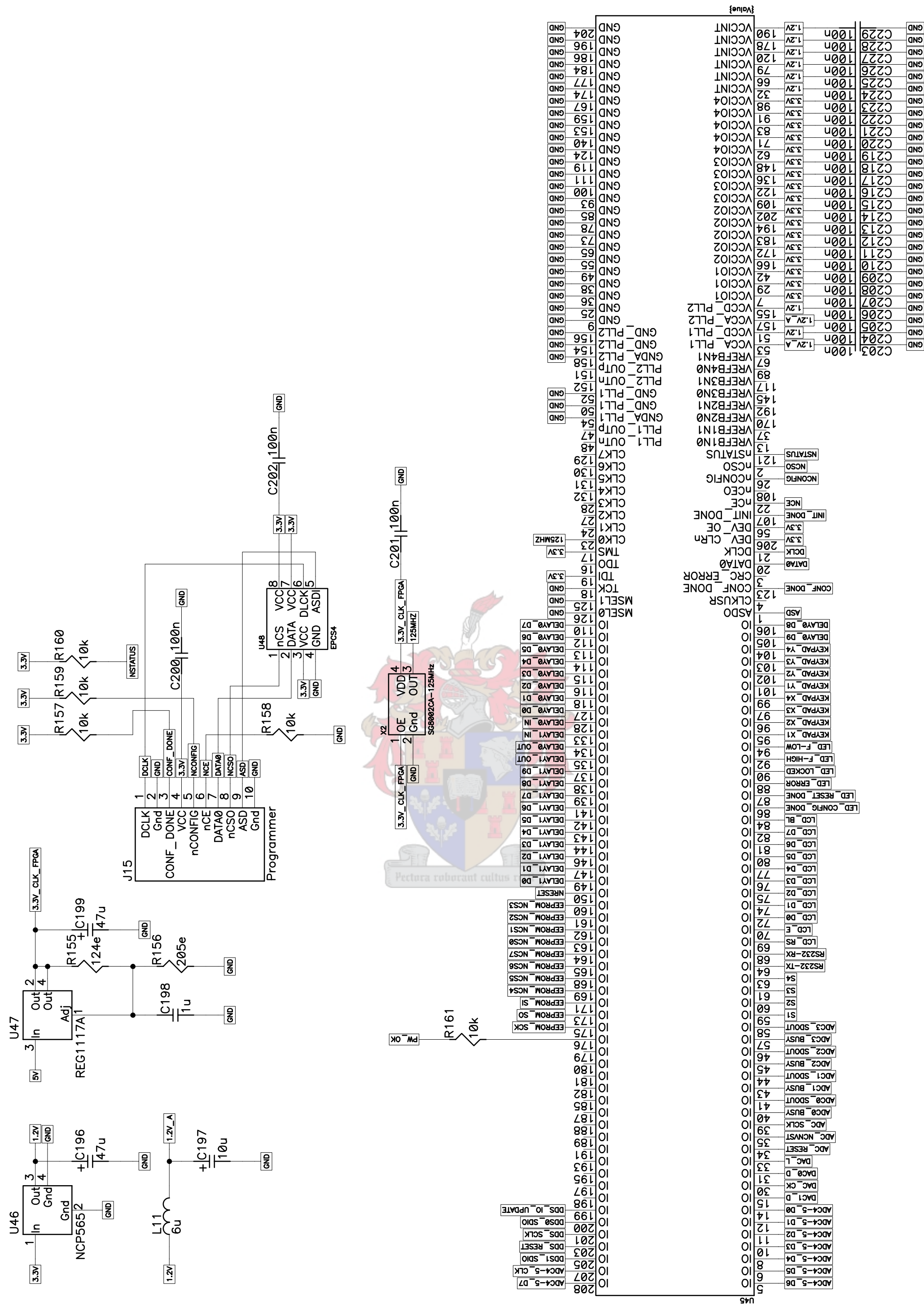


Figure D.3: FPGA circuit diagram

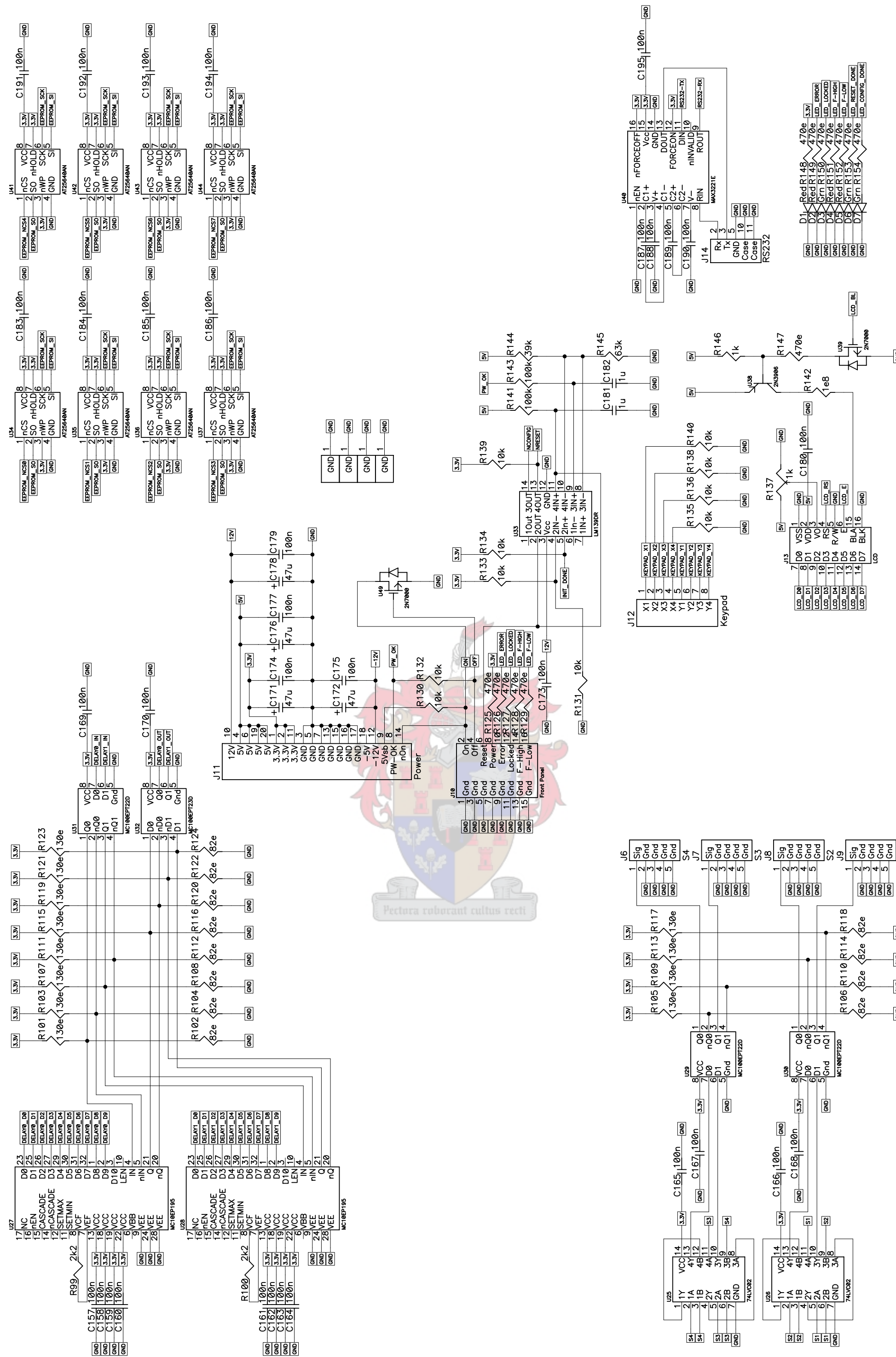


Figure D.4: Other controller components circuit diagram

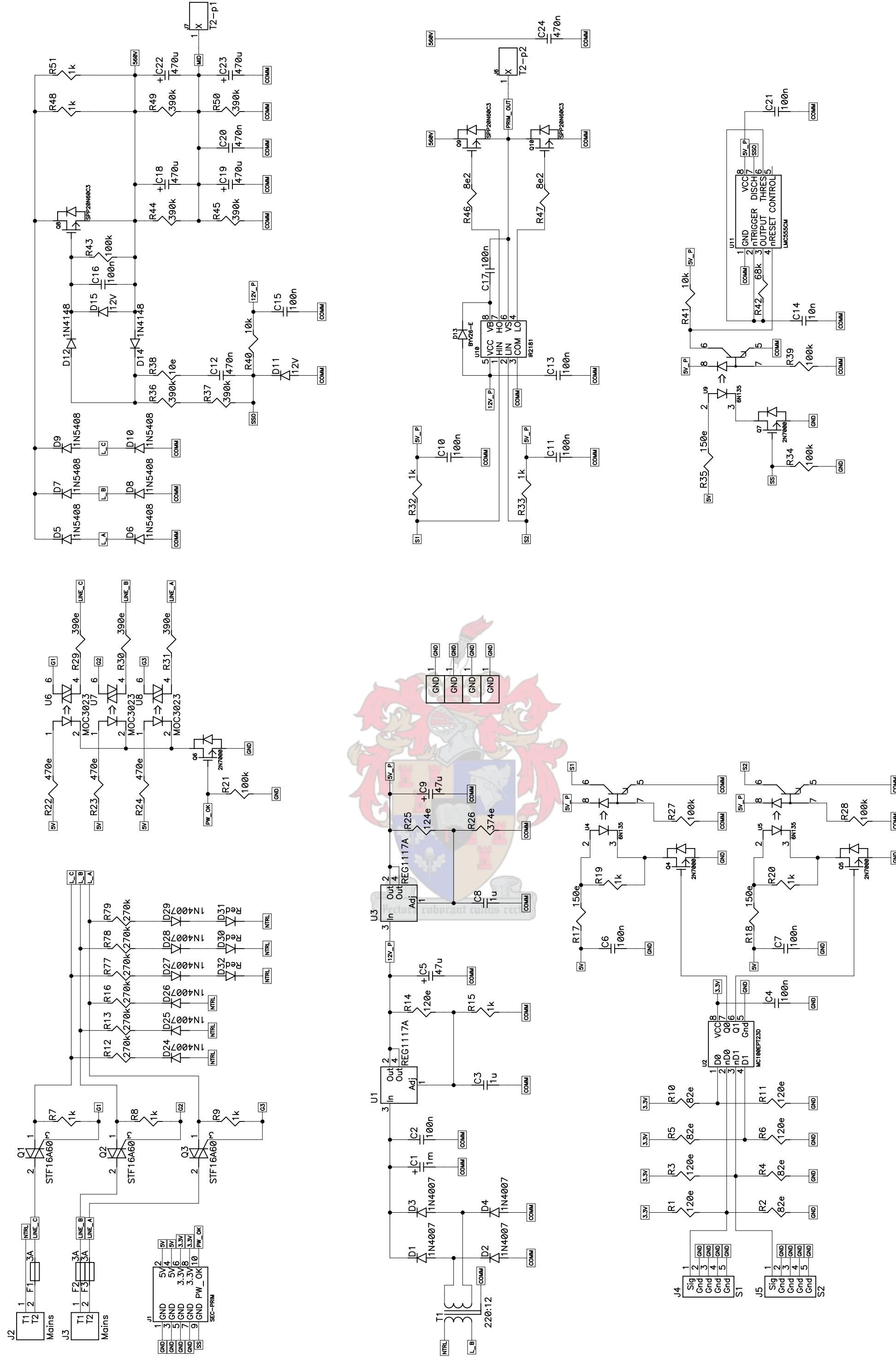


Figure D.5: Converter primary circuit diagram

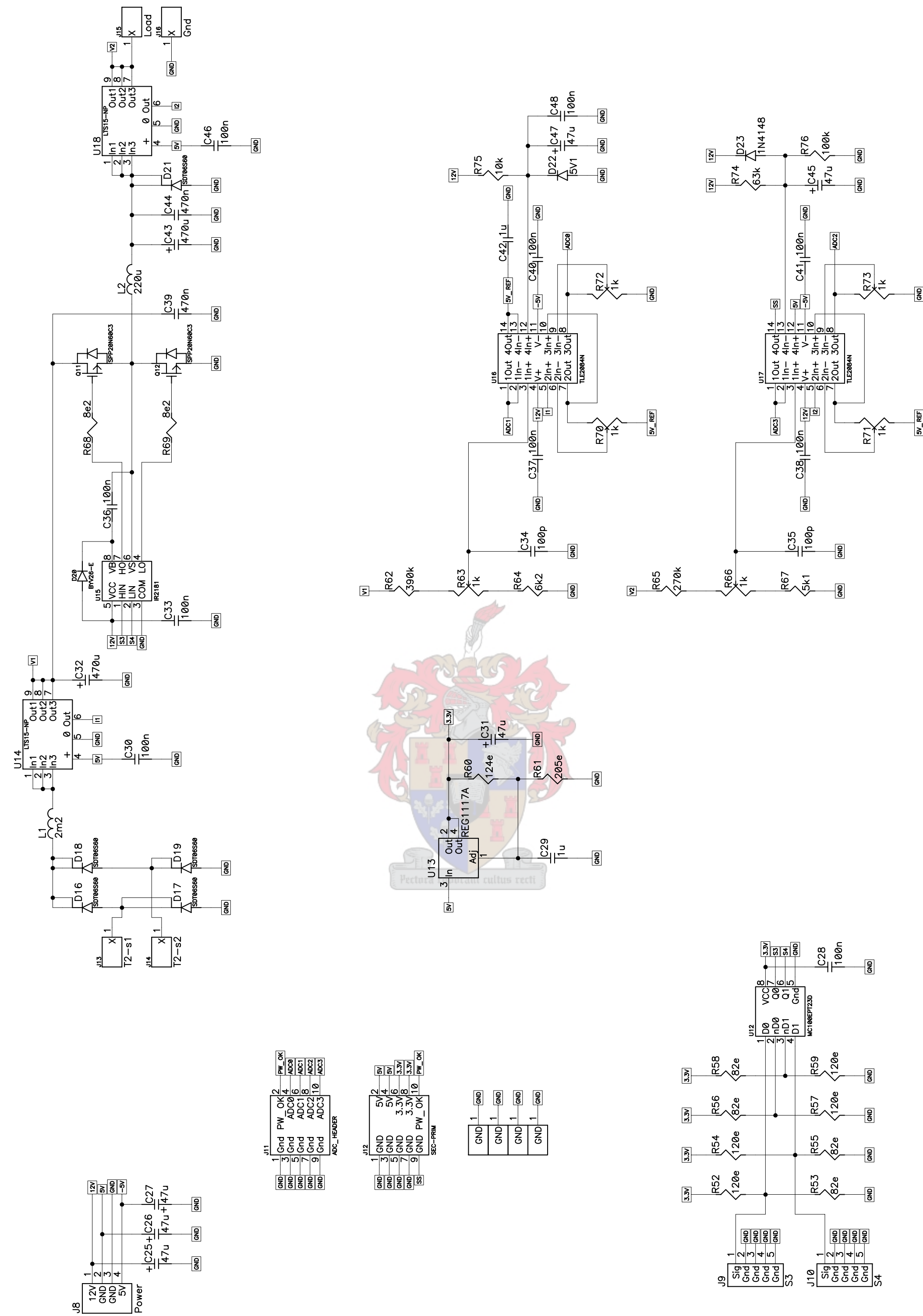


Figure D.6: Converter secondary circuit diagram

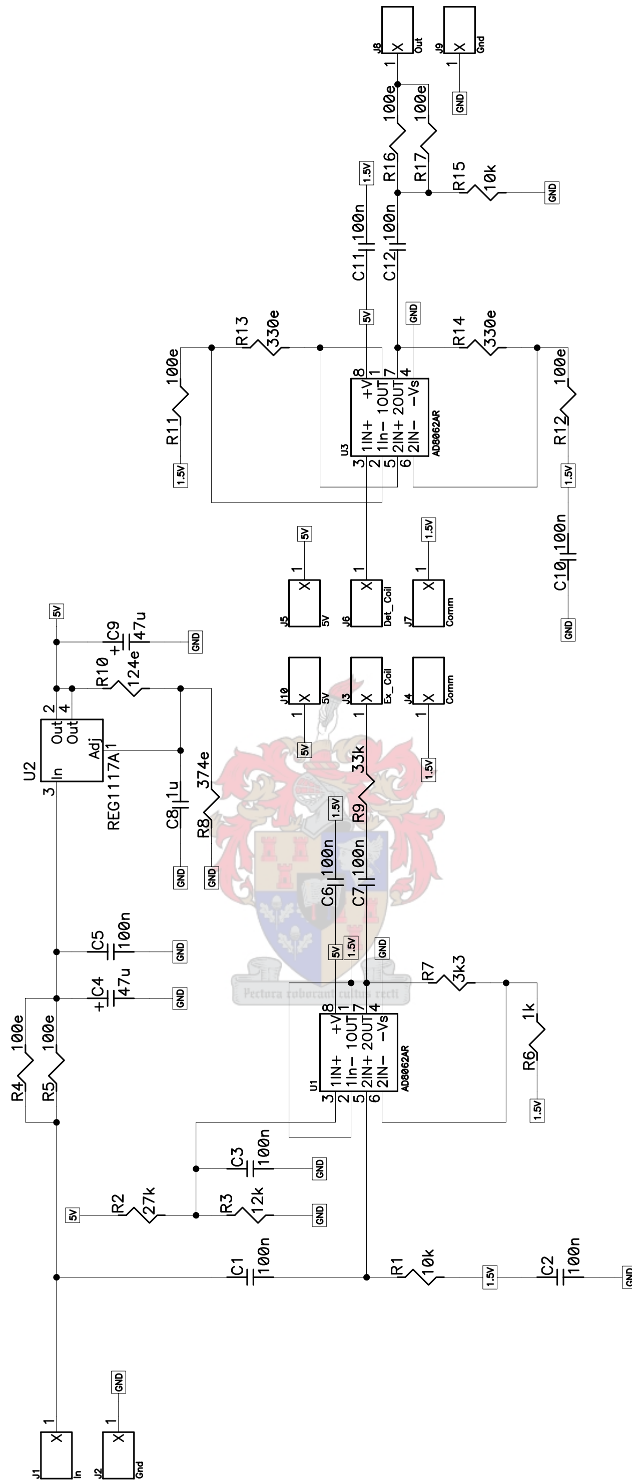


Figure D.7: Probe circuit diagram

Appendix E

FPGA Source Code

E.1 Introduction

This appendix presents the FPGA code used for practical measurements. The first set ('Interface') is for the CPU and probe circuit. A PWM generator is included to test the principal. This set was implemented on the first iteration.

The second set ('Controller') is for the converter and was implemented on the second iteration. The integration between these two is not complete. In both cases the top entity in the hierarchy is given first.

E.2 Interface

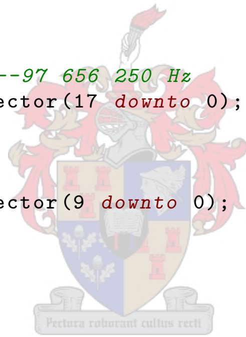
E.2.1 All_Test

```
library ieee;
use ieee.std_logic_1164.all;

entity All_Test is
 port(
  Clk_125M   : in std_logic; -- 125 MHz
  nReset     : in std_logic;

  DDS_Reset     : out std_logic;
  DDS_IO_Update : out std_logic;
  DDS_SCLK      : out std_logic;
  DDS_nCS1      : out std_logic;
  DDS_nCS2      : out std_logic;
  DDS_SDIO1     : out std_logic;
  DDS_SDIO2     : out std_logic;

  Delay0_D   : out std_logic_vector(9 downto 0);
  Delay0_In  : out std_logic;
  Delay0_Out : in  std_logic;

  Delay1_D   : out std_logic_vector(9 downto 0);
  Delay1_In  : out std_logic;
  Delay1_Out : in  std_logic;
```

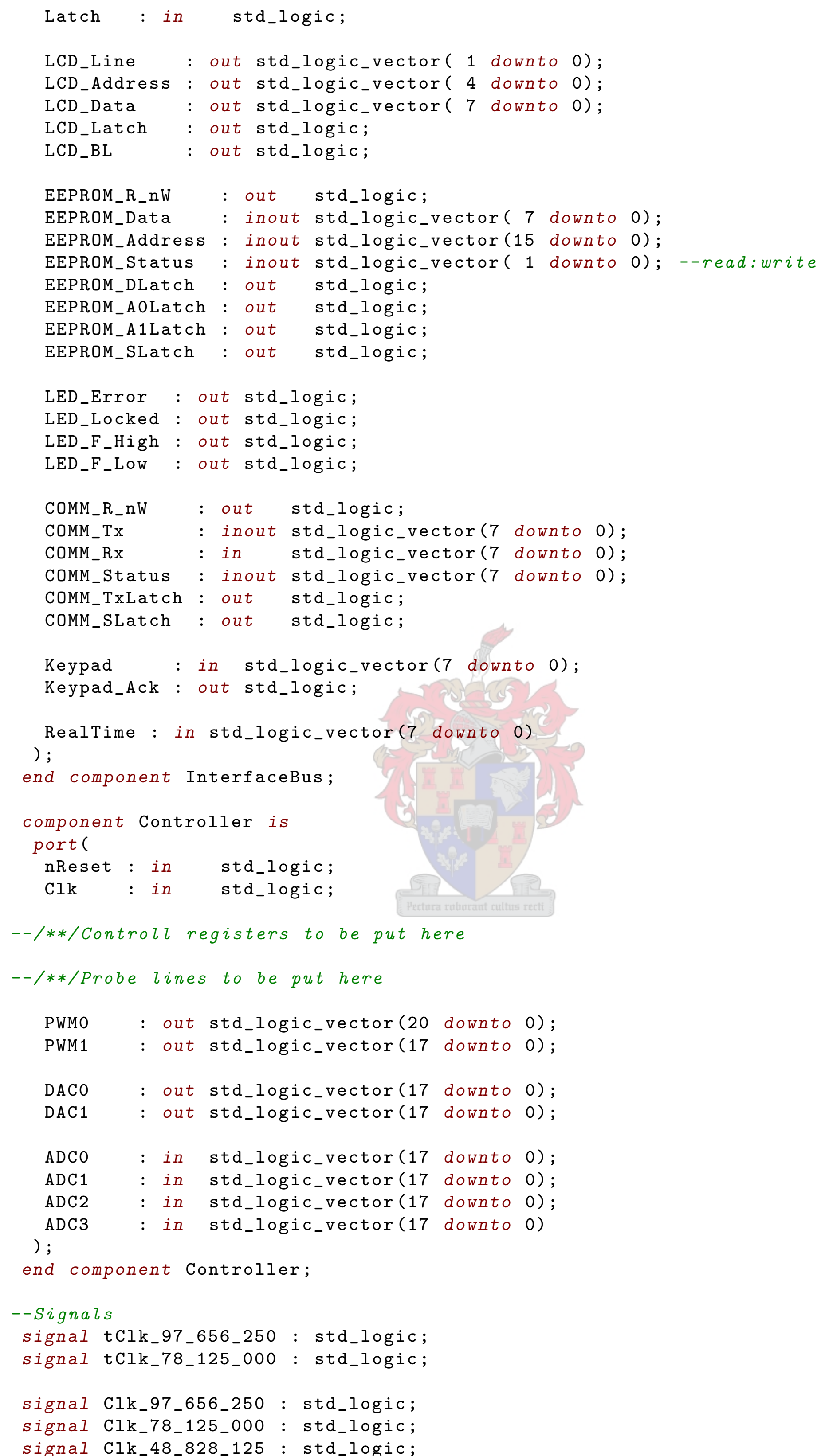
```
  DAC_DR      : out std_logic;
  DAC_DL      : out std_logic;
  DAC_Latch   : out std_logic;
  DAC_Clk     : out std_logic;

  ADC_Reset   : out std_logic;
  ADC_SClk    : out std_logic;
  ADC_nCS     : out std_logic;
  ADC_nCNVST  : out std_logic;
  ADC1_Busy   : in  std_logic;
  ADC1_SDOUT  : in  std_logic;
  ADC2_Busy   : in  std_logic;
  ADC2_SDOUT  : in  std_logic;
  ADC3_Busy   : in  std_logic;
  ADC3_SDOUT  : in  std_logic;
  ADC4_Busy   : in  std_logic;
  ADC4_SDOUT  : in  std_logic;

  S1          : out std_logic;
  S2          : out std_logic;
  S3          : out std_logic;
  S4          : out std_logic;

  LCD_RS      : out std_logic;
  LCD_E       : out std_logic;
  LCD_D       : out std_logic_vector(7 downto 0);
  LCD_BL      : out std_logic;

  Keypad_X    : in  std_logic_vector( 4 downto 1);
  Keypad_Y    : out std_logic_vector( 4 downto 1);

  EEPROM_NCS  : out std_logic_vector( 7 downto 0);
  EEPROM_SCK  : out std_logic;
  EEPROM_SI   : out std_logic;
  EEPROM_SO   : in  std_logic;

  RS232_Tx      : out std_logic;
  RS232_Rx      : in  std_logic;

  LED_Error   : out std_logic;
  LED_Locked  : out std_logic;
  LED_F_High  : out std_logic;
  LED_F_Low   : out std_logic;
  LED_Reset   : out std_logic; --Reset done
  LED_Config  : out std_logic   --Config Done
 );
end entity All_Test;

architecture a1 of All_Test is
 component PLL is
  port(
   inclk0 : in  std_logic;
   c0     : out std_logic; --97 656 250.000 Hz
   c1     : out std_logic; --78 125 000.000 Hz
   locked : out std_logic
  );
 end component PLL;

 component Global is
  port(
   a_in : in  std_logic;
   a_out: out std_logic
  );
 end component Global;
```

```
component Counter5 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(4 downto 0)
 );
end component Counter5;

component Counter14 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(13 downto 0)
 );
end component Counter14;

component RealTime is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic; --9 765 625 Hz
  Q     : out std_logic_vector(7 downto 0)
 );
end component RealTime;

component DAC is
 port(
  nReset : in  std_logic;
  Clock  : in  std_logic;
  Data0  : in  std_logic_vector(17 downto 0);
  Data1  : in  std_logic_vector(17 downto 0);

  DR     : out std_logic;
  DL     : out std_logic;
  Latch  : out std_logic;
  Clk    : out std_logic
 );
end component DAC;

component ADC18 is
 port(
  nReset : in  std_logic;
  Clock  : in  std_logic;
  Data1  : out std_logic_vector(17 downto 0);
  Data2  : out std_logic_vector(17 downto 0);
  Data3  : out std_logic_vector(17 downto 0);
  Data4  : out std_logic_vector(17 downto 0);

  Reset  : out std_logic;
  SClk   : out std_logic;
  nCS    : out std_logic;
  nCNVST : out std_logic;
  Busy1  : in  std_logic;
  SDOUT1 : in  std_logic;
  Busy2  : in  std_logic;
  SDOUT2 : in  std_logic;
  Busy3  : in  std_logic;
  SDOUT3 : in  std_logic;
  Busy4  : in  std_logic;
  SDOUT4 : in  std_logic
 );
end component ADC18;

component Probe is
 port(
  nReset: in  std_logic;
```

```
   Clk    : in  std_logic;

   DDS_Reset      : out std_logic;
   DDS_IO_Update : out std_logic;
   DDS_SCLK       : out std_logic;
   DDS_nCS1       : out std_logic;
   DDS_nCS2       : out std_logic;
   DDS_SDIO1      : out std_logic;
   DDS_SDIO2      : out std_logic
  );
 end component Probe;

 component PWM1 is
  port(
   nReset     : in  std_logic;
   Clk        : in  std_logic; --97 656 250 Hz
   Duty       : in  std_logic_vector(20 downto 0);

   Delay_Out : out std_logic;
   Delay_In  : in  std_logic;
   Delay     : out std_logic_vector(9 downto 0);

   S1         : out std_logic;
   S2         : out std_logic
  );
 end component PWM1;

 component PWM2 is
  port(
   nReset     : in  std_logic;
   Clk        : in  std_logic; --97 656 250 Hz
   Duty       : in  std_logic_vector(17 downto 0);

   Delay_Out : out std_logic;
   Delay_In  : in  std_logic;
   Delay     : out std_logic_vector(9 downto 0);

   S1         : out std_logic;
   S2         : out std_logic
  );
 end component PWM2;

 component LCD is
  port(
   nReset   : in  std_logic;
   Clk      : in  std_logic;

   Line     : in  std_logic_vector(1 downto 0);
   Address  : in  std_logic_vector(4 downto 0);
   Data     : in  std_logic_vector(7 downto 0);
   Latch    : in  std_logic;

   RS       : out std_logic;
   E        : out std_logic;
   D        : out std_logic_vector(7 downto 0)
  );
 end component LCD;

 component Keypad_Driver is
  port(
   nReset : in  std_logic;
   Clk    : in  std_logic;

   Data   : out std_logic_vector(7 downto 0); --Data(7) = Empty;
                                               --Data(3 downto 0) = Key
```


Pectora roborant cultus recti

```
   Ack     : in  std_logic;

   X       : in  std_logic_vector( 4 downto 1);
   Y       : out std_logic_vector( 4 downto 1)
  );
 end component Keypad_Driver;

 component Interface is
  port(
   nReset      : in  std_logic;
   Clk         : in  std_logic;

   Bus_R_nW    : out   std_logic;
   Bus_Data    : inout std_logic_vector(7 downto 0);
   Bus_Address : out   std_logic_vector(15 downto 0);
   Bus_Latch   : out   std_logic
  );
 end component Interface;

 component EEPROM is
  port(
   nReset     : in    std_logic;
   Clk        : in    std_logic;

   R_nW       : in    std_logic;
   Data       : inout std_logic_vector( 7 downto 0);
   Address    : inout std_logic_vector(15 downto 0);
   Status     : inout std_logic_vector( 1 downto 0); --read : write
   DLatch     : in    std_logic;
   A0Latch    : in    std_logic;
   A1Latch    : in    std_logic;
   SLatch     : in    std_logic;

   NCS        : out   std_logic_vector( 7 downto 0);
   SCK        : out   std_logic;
   SI         : out   std_logic;
   SO         : in    std_logic
  );
 end component EEPROM;

 component COMM is
  port(
   nReset : in std_logic;
   Clk    : in std_logic; --9 765 625 Hz

   R_nW    : in    std_logic;
   Tx      : inout std_logic_vector(7 downto 0);
   Rx      : out   std_logic_vector(7 downto 0);
   Status  : inout std_logic_vector(7 downto 0);
   TxLatch : in    std_logic;
   SLatch  : in    std_logic;

   RS232_Tx     : out std_logic;
   RS232_Rx     : in  std_logic
  );
 end component COMM;

 component InterfaceBus is
  port(
   nReset : in    std_logic;
   Clk    : in    std_logic; --The CPU clock

   R_nW    : in    std_logic;
   Data    : inout std_logic_vector(7 downto 0);
   Address : in    std_logic_vector(15 downto 0);
```



```
   Latch   : in    std_logic;

   LCD_Line    : out std_logic_vector( 1 downto 0);
   LCD_Address : out std_logic_vector( 4 downto 0);
   LCD_Data    : out std_logic_vector( 7 downto 0);
   LCD_Latch   : out std_logic;
   LCD_BL      : out std_logic;

   EEPROM_R_nW    : out   std_logic;
   EEPROM_Data    : inout std_logic_vector( 7 downto 0);
   EEPROM_Address : inout std_logic_vector(15 downto 0);
   EEPROM_Status  : inout std_logic_vector( 1 downto 0); --read:write
   EEPROM_DLatch  : out   std_logic;
   EEPROM_A0Latch : out   std_logic;
   EEPROM_A1Latch : out   std_logic;
   EEPROM_SLatch  : out   std_logic;

   LED_Error  : out std_logic;
   LED_Locked : out std_logic;
   LED_F_High : out std_logic;
   LED_F_Low  : out std_logic;

   COMM_R_nW    : out   std_logic;
   COMM_Tx      : inout std_logic_vector(7 downto 0);
   COMM_Rx      : in    std_logic_vector(7 downto 0);
   COMM_Status  : inout std_logic_vector(7 downto 0);
   COMM_TxLatch : out   std_logic;
   COMM_SLatch  : out   std_logic;

   Keypad     : in  std_logic_vector(7 downto 0);
   Keypad_Ack : out std_logic;

   RealTime : in std_logic_vector(7 downto 0)
  );
 end component InterfaceBus;

 component Controller is
  port(
   nReset : in    std_logic;
   Clk    : in    std_logic;

--/**/Controll registers to be put here

--/**/Probe lines to be put here

   PWM0    : out std_logic_vector(20 downto 0);
   PWM1    : out std_logic_vector(17 downto 0);

   DAC0    : out std_logic_vector(17 downto 0);
   DAC1    : out std_logic_vector(17 downto 0);

   ADC0    : in  std_logic_vector(17 downto 0);
   ADC1    : in  std_logic_vector(17 downto 0);
   ADC2    : in  std_logic_vector(17 downto 0);
   ADC3    : in  std_logic_vector(17 downto 0)
  );
 end component Controller;

--Signals
 signal tClk_97_656_250 : std_logic;
 signal tClk_78_125_000 : std_logic;

 signal Clk_97_656_250 : std_logic;
 signal Clk_78_125_000 : std_logic;
 signal Clk_48_828_125 : std_logic;
```

```
 signal Clk_24_414_063 : std_logic;
 signal Clk_9_765_625  : std_logic;
 signal Clk_3_051_758  : std_logic;
 signal Clk_1_220_703  : std_logic;
 signal Clk_4_768      : std_logic;
 signal tnReset        : std_logic;

 signal Clk0_Temp   : std_logic_vector( 5 downto 1);
 signal Clk1_Temp   : std_logic_vector(14 downto 1);
 signal PLL_Locked : std_logic;

 signal tPWM0 : std_logic_vector(20 downto 0);
 signal tPWM1 : std_logic_vector(17 downto 0);

 signal tDAC0 : std_logic_vector(17 downto 0);
 signal tDAC1 : std_logic_vector(17 downto 0);

 signal tADC0 : std_logic_vector(17 downto 0);
 signal tADC1 : std_logic_vector(17 downto 0);
 signal tADC2 : std_logic_vector(17 downto 0);
 signal tADC3 : std_logic_vector(17 downto 0);

 signal LCD_Line    : std_logic_vector( 1 downto 0);
 signal LCD_Address : std_logic_vector( 4 downto 0);
 signal LCD_Data    : std_logic_vector( 7 downto 0);
 signal LCD_Latch   : std_logic;

 signal Keypad_Data : std_logic_vector(7 downto 0);
 signal Keypad_Ack  : std_logic;

 signal EEPROM_R_nW     : std_logic;
 signal EEPROM_Data     : std_logic_vector( 7 downto 0);
 signal EEPROM_Address : std_logic_vector(15 downto 0);
 signal EEPROM_Status   : std_logic_vector( 1 downto 0);
 signal EEPROM_DLatch   : std_logic;
 signal EEPROM_A0Latch : std_logic;
 signal EEPROM_A1Latch : std_logic;
 signal EEPROM_SLatch   : std_logic;

 signal COMM_R_nW    : std_logic;
 signal COMM_Tx      : std_logic_vector(7 downto 0);
 signal COMM_Rx      : std_logic_vector(7 downto 0);
 signal COMM_Status  : std_logic_vector(7 downto 0);
 signal COMM_TxLatch : std_logic;
 signal COMM_SLatch  : std_logic;

 signal Bus_R_nW    : std_logic;
 signal Bus_Data    : std_logic_vector(7 downto 0);
 signal Bus_Address : std_logic_vector(15 downto 0);
 signal Bus_Latch   : std_logic;

 signal RealTimeQ   : std_logic_vector(7 downto 0);
begin
 PLL1   : PLL        port map(Clk_125M, tClk_97_656_250, tClk_78_125_000,
                               PLL_Locked);
 Count1 : Counter5   port map(tnReset, tClk_97_656_250, Clk0_Temp);
 Count2 : Counter14  port map(tnReset, tClk_78_125_000, Clk1_Temp);

 Global1 : Global port map(tClk_97_656_250, Clk_97_656_250);
 Global2 : Global port map(tClk_78_125_000, Clk_78_125_000);
 Global3 : Global port map(Clk0_Temp( 1)   , Clk_48_828_125);
 Global4 : Global port map(Clk0_Temp( 2)   , Clk_24_414_063);
 Global5 : Global port map(Clk1_Temp( 3)   , Clk_9_765_625 );
 Global6 : Global port map(Clk0_Temp( 5)   , Clk_3_051_758 );
 Global7 : Global port map(Clk1_Temp( 6)   , Clk_1_220_703 );
```

```
Global8 : Global port map(Clk1_Temp(14)  , Clk_4_768     );
tnReset <= nReset;

RealTime1 : RealTime port map(tnReset, Clk_9_765_625, RealTimeQ);

DAC1 : DAC port map(tnReset, Clk_24_414_063, tDAC0, tDAC1,
                    DAC_DR, DAC_DL, DAC_Latch, DAC_Clk);

ADC1 : ADC18 port map(tnReset, Clk_48_828_125, tADC0, tADC1, tADC2, tADC3,
                      ADC_Reset, ADC_SClk, ADC_nCS, ADC_nCNVST,
                      ADC1_Busy, ADC1_SDOUT,
                      ADC2_Busy, ADC2_SDOUT,
                      ADC3_Busy, ADC3_SDOUT,
                      ADC4_Busy, ADC4_SDOUT);

Probe1 : Probe port map(tnReset  , Clk_3_051_758,
                        DDS_Reset, DDS_IO_Update, DDS_SCLK,
                        DDS_nCS1 , DDS_nCS2,
                        DDS_SDIO1, DDS_SDIO2);

PWM1_1 : PWM1 port map(tnReset, Clk_97_656_250, tPWM0,
                       Delay0_In, Delay0_Out, Delay0_D, S1, S2);
PWM2_1 : PWM2 port map(tnReset, Clk_97_656_250, tPWM1,
                       Delay1_In, Delay1_Out, Delay1_D, S3, S4);

LCD1 : LCD port map(tnReset, Clk_3_051_758,
                    LCD_Line, LCD_Address, LCD_Data, LCD_Latch,
                    LCD_RS, LCD_E, LCD_D);

Keypad1 : Keypad_Driver port map(tnReset, Clk_4_768,
                                 Keypad_Data, Keypad_Ack,
                                 Keypad_X, Keypad_Y);

Interface1 : Interface port map(tnReset, Clk_1_220_703,
                                Bus_R_nW, Bus_Data, Bus_Address, Bus_Latch);

EEPROM1 : EEPROM port map(tnReset, Clk_9_765_625,
                          EEPROM_R_nW, EEPROM_Data, EEPROM_Address,
                          EEPROM_Status, EEPROM_DLatch,
                          EEPROM_A0Latch, EEPROM_A1Latch, EEPROM_SLatch,
                          EEPROM_NCS, EEPROM_SCK, EEPROM_SI, EEPROM_SO);

COMM1 : COMM port map(tnReset, Clk_9_765_625,
                      COMM_R_nW, COMM_Tx, COMM_Rx, COMM_Status,
                      COMM_TxLatch, COMM_SLatch,
                      RS232_Tx, RS232_Rx);

Bus1 : InterfaceBus port map(tnReset, Clk_1_220_703,
                             Bus_R_nW, Bus_Data, Bus_Address, Bus_Latch,
                             LCD_Line, LCD_Address,
                             LCD_Data, LCD_Latch, LCD_BL,
                             EEPROM_R_nW, EEPROM_Data, EEPROM_Address,
                             EEPROM_Status, EEPROM_DLatch,
                             EEPROM_A0Latch, EEPROM_A1Latch, EEPROM_SLatch,
                             LED_Error, LED_Locked, LED_F_High, LED_F_Low,
                             COMM_R_nW, COMM_Tx, COMM_Rx, COMM_Status,
                             COMM_TxLatch, COMM_SLatch,
                             Keypad_Data, Keypad_Ack,
                             RealTimeQ);

Controller1 : Controller port map(tnReset, Clk_1_220_703,
                                  tPWM0, tPWM1,
                                  tDAC0, tDAC1,
                                  tADC0, tADC1, tADC2, tADC3);
```

```
 LED_Reset  <= tnReset;
 LED_Config <= PLL_Locked;
end architecture a1;
```

E.2.2 ADC18

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ADC18 is
 port(
  nReset : in  std_logic;
  Clock  : in  std_logic;
  Data1  : out std_logic_vector(17 downto 0);
  Data2  : out std_logic_vector(17 downto 0);
  Data3  : out std_logic_vector(17 downto 0);
  Data4  : out std_logic_vector(17 downto 0);

  Reset  : out std_logic;
  SClk   : out std_logic;
  nCS    : out std_logic;
  nCNVST : out std_logic;
  Busy1  : in  std_logic;
  SDOUT1 : in  std_logic;
  Busy2  : in  std_logic;
  SDOUT2 : in  std_logic;
  Busy3  : in  std_logic;
  SDOUT3 : in  std_logic;
  Busy4  : in  std_logic;
  SDOUT4 : in  std_logic
 );
end entity ADC18;

architecture a1 of ADC18 is
 signal state : std_logic_vector( 2 downto 0);
 signal count : std_logic_vector( 4 downto 0);
 signal Busy  : std_logic;
 signal Temp1 : std_logic_vector(17 downto 0);
 signal Temp2 : std_logic_vector(17 downto 0);
 signal Temp3 : std_logic_vector(17 downto 0);
 signal Temp4 : std_logic_vector(17 downto 0);
 signal Cp1   : std_logic_vector( 4 downto 0);
begin
 Reset <= not nReset;
-- Busy  <= Busy1 or Busy2 or Busy3 or Busy4;
 Busy <= Busy1 or Busy4; --only these two are installed
 nCS  <= '0';
 Cp1 <= Count + '1';

 process(Clock, nReset) is
 begin
  if nReset = '0' then
   state  <= "000";
   count  <= "00000";
   SClk   <= '0';
   nCNVST <= '1';
   Temp1  <= "000000000000000000";
   Temp2  <= "000000000000000000";
   Temp3  <= "000000000000000000";
   Temp4  <= "000000000000000000";
  elsif rising_edge(Clock) then
   case state is
    when "000" =>
```

```
     if count = "11111" then
      state <= "001";
     end if;
     count <= Cp1;
    when "001" =>
     count <= "00000";
     if Busy = '0' then
      nCNVST <= '0';
      state <= "010";
     end if;
    when "010" =>
     nCNVST <= '1';
     SClk <= '1';
     Temp1(17 downto 1) <= Temp1(16 downto 0);
     Temp2(17 downto 1) <= Temp2(16 downto 0);
     Temp3(17 downto 1) <= Temp3(16 downto 0);
     Temp4(17 downto 1) <= Temp4(16 downto 0);
     Count <= Cp1;
     state <= "011";
    when "011" =>
     SClk <= '0';
     Temp1(0) <= SDOUT1;
     Temp2(0) <= SDOUT2;
     Temp3(0) <= SDOUT3;
     Temp4(0) <= SDOUT4;
     if count = "10010" then --18
      state <= "100";
     else
      state <= "010";
     end if;
    when "100" =>
     Data1 <= Temp1;
     Data2 <= Temp2;
     Data3 <= Temp3;
     Data4 <= Temp4;
     state <= "001";
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.2.3 Adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Adder is
 port(
  A        : in  std_logic_vector(7 downto 0);
  B        : in  std_logic_vector(8 downto 0);
  Carry_In : in  std_logic;
  Y        : out std_logic_vector(7 downto 0);
  Carry    : out std_logic
 );
end entity Adder;

architecture a1 of Adder is
 signal tY : std_logic_vector(8 downto 0);
begin
 tY     <= A + B + Carry_In;
 Y      <= tY(7 downto 0);
 Carry  <= tY(8);
```

```
end architecture a1;
```

E.2.4 Alternator

```
library ieee;
use ieee.std_logic_1164.all;

entity Alternator is
 port(
  nReset: in  std_logic;
  PWM   : in  std_logic;
  S1    : out std_logic;
  S2    : out std_logic
 );
end entity Alternator;

architecture a1 of Alternator is
 signal Q: std_logic_vector(1 downto 0);
begin
 process(PWM, nReset) is
 begin
  if nReset = '0' then
   Q <= "01";
  elsif falling_edge(PWM) then
   case Q is
    when "01" =>
     Q <= "10";
    when "10" =>
     Q <= "01";
    when others =>
   end case;
  end if;
 end process;
 S1 <= Q(0) and PWM and nReset;
 S2 <= Q(1) and PWM and nReset;
end architecture a1;
```

E.2.5 Arith

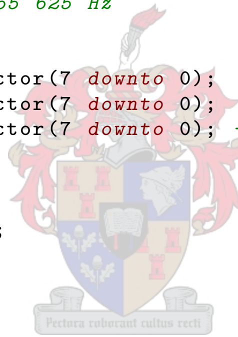
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Arith is
 port(
  A        : in  std_logic_vector(7 downto 0);
  B        : in  std_logic_vector(7 downto 0);
  Task     : in  std_logic_vector(3 downto 0);
  Carry_In : in  std_logic;
  Zero_In  : in  std_logic;
  Y        : out std_logic_vector(7 downto 0);
  Carry    : out std_logic;
  Zero     : out std_logic
 );
end entity Arith;

architecture a1 of Arith is
 component Adder is
  port(
   A        : in  std_logic_vector(7 downto 0);
   B        : in  std_logic_vector(8 downto 0);
   Carry_In : in  std_logic;
   Y        : out std_logic_vector(7 downto 0);
```

```
   Carry     : out std_logic
  );
 end component Adder;

 signal A_B     : std_logic_vector(8 downto 0);
 signal A_C_In  : std_logic;
 signal A_Y     : std_logic_vector(7 downto 0);
 signal A_C     : std_logic;

 signal tY  : std_logic_vector(7 downto 0);
 signal Y3  : std_logic_vector(7 downto 0);
 signal Y4  : std_logic_vector(7 downto 0);
 signal Y5  : std_logic_vector(7 downto 0);
 signal Y6  : std_logic_vector(7 downto 0);
 signal Y9  : std_logic_vector(7 downto 0);
 signal Y10 : std_logic_vector(7 downto 0);
 signal Y11 : std_logic_vector(7 downto 0);
 signal Y12 : std_logic_vector(7 downto 0);
 signal Y13 : std_logic_vector(7 downto 0);
 signal Y14 : std_logic_vector(7 downto 0);
 signal Y15 : std_logic_vector(7 downto 0);

 signal tC  : std_logic;
 signal tZ  : std_logic;

 signal C10 : std_logic;
 signal C11 : std_logic;
begin
 Addd1 : Adder port map(A, A_B, A_C_In, A_Y, A_C);
 with Task select
  A_B(8) <= '1' when "0111" | "1000",
            '0' when others;

 A_B(7 downto 0) <= (not B) when A_B(8) = '1' else B;

 with Task select
  tC <= Carry_In when "0001" | "0111",
        '0'      when others;

 A_C_In <= (not tC) when A_B(8) = '1' else tC;

 Y3  <= A and B;
 Y4  <= Y5 + '1';
 Y5  <= not A;
 Y6  <= A or B;
 Y9  <= A xor B;
 C10  <= A(7); Y10(7 downto 1) <= A(6 downto 0); Y10(0) <= Carry_In;
 C11  <= A(0); Y11(6 downto 0) <= A(7 downto 1); Y11(7) <= Carry_In;
 Y12(7 downto 1) <= A(6 downto 0); Y12(0) <= A(7);
 Y13(6 downto 0) <= A(7 downto 1); Y13(7) <= A(0);
 Y14(7 downto 1) <= A(6 downto 0); Y14(0) <= Carry_In;
 Y15(6 downto 0) <= A(7 downto 1); Y15(7) <= Carry_In;

 with Task select
  tY <= A_Y when "0001" | "0010" | "0111" | "1000",
        Y3  when "0011",
        Y4  when "0100",
        Y5  when "0101",
        Y6  when "0110",
        Y9  when "1001",
        Y10 when "1010",
        Y11 when "1011",
        Y12 when "1100",
        Y13 when "1101",
        Y14 when "1110",
```

```
        Y15 when "1111",
        A   when others;
 Y <= tY;

 tZ   <= '1'     when tY   = "00000000" else '0';
 Zero <= Zero_In when Task = "0000"     else tZ ;

 with Task select
  Carry <= A_C      when "0001" | "0010" | "0111" | "1000",
           C10      when "1010",
           C11      when "1011",
           Carry_In when others;
end architecture a1;
```

E.2.6 COMM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

--/**/Give a FIFO queue - received remains low when queue is empty -
--     512 bytes long in M4K block

entity COMM is
 port(
  nReset : in std_logic;
  Clk    : in std_logic; --9 765 625 Hz

  R_nW    : in    std_logic;
  Tx      : inout std_logic_vector(7 downto 0);
  Rx      : out   std_logic_vector(7 downto 0);
  Status  : inout std_logic_vector(7 downto 0); --Received : Send
  TxLatch : in    std_logic;
  SLatch  : in    std_logic;

  RS232_Tx     : out std_logic;
  RS232_Rx     : in  std_logic
 );
end entity COMM;

architecture a1 of COMM is
 component Latch8 is
  port(
   nReset     : in  std_logic;
   Latch      : in  std_logic;

   Data       : in  std_logic_vector(7 downto 0);
   lData      : out std_logic_vector(7 downto 0)
  );
 end component Latch8;

 component Latch1 is
  port(
   nReset     : in  std_logic;
   Latch      : in  std_logic;

   Data       : in  std_logic;
   lData      : out std_logic
  );
 end component Latch1;

 component RS232Rx is
  port(
   nReset    : in  std_logic;
```


Pectora roborant cultus recti

```
   Clk      : in  std_logic; --9 765 625 Hz
   Rx       : in  std_logic;
   Data     : out std_logic_vector(7 downto 0);
   Received : in  std_logic; --Must be low for received data to be latched
   SR       : out std_logic  --Pulsed high at end of reception
  );
 end component RS232Rx;

 component RS232Tx is
  port(
   nReset   : in  std_logic;
   Clk      : in  std_logic; --9 765 625 Hz
   Tx       : out std_logic;
   Data     : in  std_logic_vector(7 downto 0);
   Send     : in  std_logic; --Pulsed high to start transmission
   CS       : out std_logic  --Pulsed high at end of transmission
  );
 end component RS232Tx;

 signal lTx     : std_logic_vector(7 downto 0);
 signal lStatus : std_logic_vector(7 downto 0);

 signal CS : std_logic;
 signal SR : std_logic;

 signal tRS232_Rx : std_logic_vector(7 downto 0);
 signal tTx : std_logic;

 signal L1_Latch : std_logic;
 signal L4_Latch : std_logic;
begin
 L1_Latch <= SLatch when lStatus(1) = '1' else (not SR);
 L1 : Latch1 port map(nReset, L1_Latch, lStatus(1) nand (not Status(1)),
                      lStatus(1));
 L2 : Latch8 port map(nReset, not SR, tRS232_Rx, Rx);
 RS232Rx1 : RS232Rx port map(nReset, Clk, RS232_Rx, tRS232_Rx,
                             lStatus(1), SR);

 L4_Latch <= sLatch when lStatus(0) = '0' else (not CS);
 L3 : Latch8 port map(nReset, TxLatch and (not lStatus(0)), Tx, lTx);
 L4 : Latch1 port map(nReset, L4_Latch, Status(0) and (not lStatus(0)),
                      lStatus(0));
 RS232Tx1 : RS232Tx port map(nReset, Clk, tTx, lTx, lStatus(0), CS);
 RS232_Tx <= tTx;

 lStatus(7 downto 2) <= "000000";

 Tx     <= lTx     when R_nW = '1' else "ZZZZZZZZ";
 Status <= lStatus when R_nW = '1' else "ZZZZZZZZ";
end architecture a1;
```

E.2.7 Controller

```
library ieee;
use ieee.std_logic_1164.all;

entity Controller is
 port(
  nReset : in    std_logic;
  Clk    : in    std_logic;

--/**/Controll registers to be put here

--/**/Probe lines to be put here
```

```
  PWM0     : out std_logic_vector(20 downto 0);
  PWM1     : out std_logic_vector(17 downto 0);

  DAC0     : out std_logic_vector(17 downto 0);
  DAC1     : out std_logic_vector(17 downto 0);

  ADC0     : in  std_logic_vector(17 downto 0);
  ADC1     : in  std_logic_vector(17 downto 0);
  ADC2     : in  std_logic_vector(17 downto 0);
  ADC3     : in  std_logic_vector(17 downto 0)
 );
end entity Controller;

architecture a1 of Controller is
 component Counter21 is
  port(
   nReset: in  std_logic;
   Clk   : in  std_logic;
   Q     : out std_logic_vector(20 downto 0)
  );
 end component Counter21;

 signal Count : std_logic_vector(20 downto 0);
begin
 Counter : Counter21 port map(nReset, Clk, Count);
-- PWM0 <= "000011001100110011010";--5%
 PWM0 <= "000110011001100110011";--10%
-- PWM0 <= "010000000000000000000";--25%
-- PWM0 <= "100000000000000000000";--50%
-- PWM0 <= "101100110011001100110";--70%
-- PWM0 <= "110011001100110011010";--80%
 PWM1 <= "110011001100110011"; --80%
 DAC0 <= ADC3;
 DAC1 <= ADC0;
end architecture a1;
```

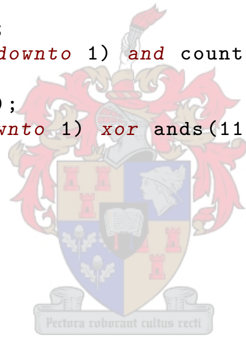
E.2.8 Counter11

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter11 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(10 downto 0)
 );
end entity Counter11;

architecture a1 of Counter11 is
 signal count: std_logic_vector(10 downto 0);
 signal D    : std_logic_vector(10 downto 0);
 signal ands : std_logic_vector(10 downto 1);
begin
 ands(          1) <= count(0);
 ands(10 downto 2) <= ands(9 downto 1) and count(9 downto 1);

 D(          0) <= not count(0);
 D(10 downto 1) <= count(10 downto 1) xor ands(10 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
```

```
    count <= "00000000000";
  elsif falling_edge(Clk) then
    count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

E.2.9 Counter12

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter12 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(11 downto 0)
 );
end entity Counter12;

architecture a1 of Counter12 is
 signal count: std_logic_vector(11 downto 0);
 signal D    : std_logic_vector(11 downto 0);
 signal ands : std_logic_vector(11 downto 1);
begin
 ands(          1) <= count(0);
 ands(11 downto 2) <= ands(10 downto 1) and count(10 downto 1);

 D(          0) <= not count(0);
 D(11 downto 1) <= count(11 downto 1) xor ands(11 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
    count <= "000000000000";
  elsif falling_edge(Clk) then
    count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```



E.2.10 Counter14

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter14 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(13 downto 0)
 );
end entity Counter14;

architecture a1 of Counter14 is
 signal count: std_logic_vector(13 downto 0);
 signal D    : std_logic_vector(13 downto 0);
 signal ands : std_logic_vector(13 downto 1);
begin
```

```
 ands(           1) <= count(0);
 ands(13 downto 2) <= ands(12 downto 1) and count(12 downto 1);

 D(           0) <= not count(0);
 D(13 downto 1) <= count(13 downto 1) xor ands(13 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "00000000000000";
  elsif falling_edge(Clk) then
   count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

E.2.11 Counter21

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter21 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(20 downto 0)
 );
end entity Counter21;

architecture a1 of Counter21 is
 signal count: std_logic_vector(20 downto 0);
 signal D    : std_logic_vector(20 downto 0);
 signal ands : std_logic_vector(20 downto 1);
begin
 ands(           1) <= count(0);
 ands(20 downto 2) <= ands(19 downto 1) and count(19 downto 1);

 D(           0) <= not count(0);
 D(20 downto 1) <= count(20 downto 1) xor ands(20 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "000000000000000000000";
  elsif falling_edge(Clk) then
   count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

E.2.12 Counter22

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter22 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
```

```
  Q      : out std_logic_vector(21 downto 0)
 );
end entity Counter22;

architecture a1 of Counter22 is
 signal count: std_logic_vector(21 downto 0);
 signal D     : std_logic_vector(21 downto 0);
 signal ands : std_logic_vector(21 downto 1);
begin
 ands(           1) <= count(0);
 ands(21 downto 2) <= ands(20 downto 1) and count(20 downto 1);

 D(           0) <= not count(0);
 D(21 downto 1) <= count(21 downto 1) xor ands(21 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "0000000000000000000000";
  elsif falling_edge(Clk) then
   count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

E.2.13 Counter5

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter5 is
 port(
  nReset: in  std_logic;
  Clk    : in  std_logic;
  Q      : out std_logic_vector(4 downto 0)
 );
end entity Counter5;

architecture a1 of Counter5 is
 signal count: std_logic_vector(4 downto 0);
 signal D     : std_logic_vector(4 downto 0);
 signal ands : std_logic_vector(4 downto 1);
begin
 ands(           1) <= count(0);
 ands(4 downto 2) <= ands(3 downto 1) and count(3 downto 1);

 D(           0) <= not count(0);
 D(4 downto 1) <= count(4 downto 1) xor ands(4 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "00000";
  elsif falling_edge(Clk) then
   count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

E.2.14 Counter8

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter8 is
 port(
  nReset: in  std_logic;
  Clk    : in  std_logic;
  Q      : out std_logic_vector(7 downto 0)
 );
end entity Counter8;

architecture a1 of Counter8 is
 signal count: std_logic_vector(7 downto 0);
 signal D     : std_logic_vector(7 downto 0);
 signal ands : std_logic_vector(7 downto 1);
begin
 ands(          1) <= count(0);
 ands(7 downto 2) <= ands(6 downto 1) and count(6 downto 1);

 D(          0) <= not count(0);
 D(7 downto 1) <= count(7 downto 1) xor ands(7 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "00000000";
  elsif falling_edge(Clk) then
   count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

E.2.15 DAC

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity DAC is
 port(
  nReset : in  std_logic;
  Clock  : in  std_logic;
  Data0  : in  std_logic_vector(17 downto 0);
  Data1  : in  std_logic_vector(17 downto 0);

  DR     : out std_logic;
  DL     : out std_logic;
  Latch  : out std_logic;
  Clk    : out std_logic
 );
end entity DAC;

architecture a1 of DAC is
 signal D0    : std_logic_vector(17 downto 0);
 signal D1    : std_logic_vector(17 downto 0);
 signal state: std_logic_vector( 1 downto 0);
 signal count: std_logic_vector( 4 downto 0);
begin
 process(Clock, nReset) is
 begin
```

```
  if nReset = '0' then
   state <= "00";
   count <= "00001";
   Clk   <= '0';
   Latch <= '1';
  elsif rising_edge(Clock) then
   case state is
    when "00" =>
     count <= "00001";
     Clk   <= '0';
     Latch <= '1';
     D0(17) <= Data0(17);
     D1(17) <= Data1(17);
     D0(16 downto 0) <= not Data0(16 downto 0);
     D1(16 downto 0) <= not Data1(16 downto 0);
     state <= "01";
    when "01" =>
     Clk <= '1';
     state <= "10";
    when "10" =>
     Clk <= '0';
     D0(17 downto 1) <= D0(16 downto 0);
     D1(17 downto 1) <= D1(16 downto 0);
     if count = "10010" then --18
      state <= "11";
     else
      state <= "01";
     end if;
     count <= count + 1;
    when "11" =>
     latch <= '0';
     state <= "00";
    when others =>
   end case;
  end if;
 end process;
 DR <= D0(17);
 DL <= D1(17);
end architecture a1;
```

E.2.16 DeadTime

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity DeadTime is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Count : in  std_logic_vector(7 downto 0);
  PWM   : in  std_logic;
  Dead  : in  std_logic_vector(7 downto 0);
  S1    : out std_logic;
  S2    : out std_logic
 );
end entity DeadTime;

architecture a1 of DeadTime is
 signal dClk    : std_logic_vector(7 downto 0);
 signal greater : std_logic;
 signal zero    : std_logic;
begin
 greater <= '1' when (not Dead) > Count else '0';
```

```
 zero    <= '1' when dClk = "00000000"  else '0';

 process(Clk, nReset, Dead) is
 begin
  if nReset = '0' then
   dClk <= Dead;
  elsif rising_edge(Clk) then
   if greater = '0' then
    dClk <= Dead;
   elsif (zero = '0') and (PWM = '0') then
    dClk <= dClk - 1;
   end if;
  end if;
 end process;

 S1   <= PWM and nReset;
 S2   <= (not PWM) and zero and greater and nReset;
end architecture a1;
```

E.2.17 EEPROM

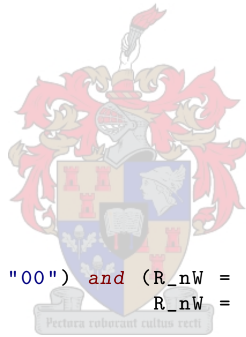
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

--To use this: set up data, address and the required status:
--   read("10") or write("01").
--Status will return to "00" when done.

entity EEPROM is
 port(
  nReset      : in    std_logic;
  Clk         : in    std_logic; --10MHz max

  R_nW        : in    std_logic;
  Data        : inout std_logic_vector( 7 downto 0); --Read-only when busy
  Address     : inout std_logic_vector(15 downto 0); --Read-only when busy
  Status      : inout std_logic_vector( 1 downto 0); --Read-only when busy
  DLatch      : in    std_logic;
  A0Latch     : in    std_logic;
  A1Latch     : in    std_logic;
  SLatch      : in    std_logic;

  NCS         : out   std_logic_vector( 7 downto 0);
  SCK         : out   std_logic;
  SI          : out   std_logic;
  SO          : in    std_logic
 );
end entity EEPROM;

architecture a1 of EEPROM is
 component Latch8 is
  port(
   nReset      : in  std_logic;
   Latch       : in  std_logic;

   Data        : in  std_logic_vector(7 downto 0);
   lData       : out std_logic_vector(7 downto 0)
  );
 end component Latch8;

 component Latch1 is
  port(
   nReset      : in  std_logic;
```

```
   Latch      : in  std_logic;

   Data       : in  std_logic;
   lData      : out std_logic
  );
 end component Latch1;

 component EEPROM_LatchS is
  port(
   nReset     : in  std_logic;
   ClearStatus: in  std_logic;
   Latch      : in  std_logic;

   Status     : in  std_logic_vector(1 downto 0);

   lStatus    : out std_logic_vector(1 downto 0)
  );
 end component EEPROM_LatchS;

 signal state        : std_logic_vector( 4 downto 0);
 signal t1state      : std_logic_vector( 4 downto 0);
 signal t2state      : std_logic_vector( 4 downto 0);
 signal t3state      : std_logic_vector( 4 downto 0);
 signal tNCS         : std_logic_vector( 7 downto 0);
 signal rData        : std_logic_vector( 7 downto 0);
 signal rAddress     : std_logic_vector( 2 downto 0);
 signal lData        : std_logic_vector( 7 downto 0);
 signal lAddress     : std_logic_vector(15 downto 0);
 signal lStatus      : std_logic_vector( 1 downto 0);
 signal tData        : std_logic_vector( 7 downto 0);
 signal Count        : std_logic_vector( 2 downto 0);
 signal ClearStatus  : std_logic;
 signal tsLatch      : std_logic;
 signal StatusNull   : std_logic;

 constant Read_SR         : std_logic_vector(4 downto 0) := "00111";
 constant Wait_for_ready  : std_logic_vector(4 downto 0) := "00100";
 constant Write_enable    : std_logic_vector(4 downto 0) := "01100";
 constant Read            : std_logic_vector(4 downto 0) := "01111";
 constant Write           : std_logic_vector(4 downto 0) := "01000";
 constant Read_tData      : std_logic_vector(4 downto 0) := "11111";
 constant Write_tData     : std_logic_vector(4 downto 0) := "11100";
begin
 StatusNull <= lStatus(0) nor lStatus(1);
 L1 : Latch8  port map(nReset, DLatch and StatusNull, Data, lData);
 L2 : Latch8  port map(nReset, A0Latch and StatusNull,
                          Address(7 downto 0), lAddress(7 downto 0));
 L3 : Latch8  port map(nReset, A1Latch and StatusNull,
                          Address(15 downto 8), lAddress(15 downto 8));
 tsLatch <= sLatch when StatusNull = '1' else ClearStatus;
 L4 : Latch1 port map(nReset, tsLatch, Status(0) and StatusNull, lStatus(0));
 L5 : Latch1 port map(nReset, tsLatch, Status(1) and StatusNull, lStatus(1));

 with rAddress select
  tNCS  <= "11111110" when "000",
           "11111101" when "001",
           "11111011" when "010",
           "11110111" when "011",
           "11101111" when "100",
           "11011111" when "101",
           "10111111" when "110",
           "01111111" when "111",
           "XXXXXXXX" when others;

 process(Clk, nReset) is
```

```
 begin
  if nReset = '0' then
   state     <= "00000";
   t1state   <= "00000";
   Count     <= "000";
   NCS       <= "11111111";
   rData     <= "00000000";
   rAddress  <= "000";
   SCK       <= '0';
   tData     <= "00000001";
  elsif rising_edge(Clk) then
   case state is
--INIT
--Check status register and fix if necessary
    when "00000" =>
     NCS  <= "11111111";
     if tData(0) = '1' then
      t2state <= "00000";
      state   <= Read_SR;
     elsif tData(7) = '1' or tData(3) = '1' or tData(2) = '1' then
      t2state <= "00001";
      state   <= Write_enable;
     else
      state <= "00010";
     end if;
    when "00001" =>
     NCS     <= tNCS;
     tData   <= "00000001";
     t3state <= "00011";
     state   <= Write_tData;
    when "00011" =>
     tData   <= "00000010";
     t3state <= "00010";
     state   <= Write_tData;
    when "00010" =>
     if rAddress = "111" then
      ClearStatus <= '1';
      state       <= "00110";
     else
      tData <= "00000001";
      state <= "00000";
     end if;
     rAddress <= rAddress + 1;
--MAIN
--Wait for operation
    when "00110" =>
     ClearStatus <= '0';
     if lStatus(1) = '1' then --Read
      rAddress <= lAddress(2 downto 0);
      t1state <= Read;
      tData   <= "00000001";
      state   <= Wait_for_ready;
     elsif lStatus(0) = '1' then --Write
      rAddress <= lAddress(2 downto 0);
      rData     <= lData;
      t1state   <= Write;
      tData     <= "00000001";
      state     <= Wait_for_ready;
     end if;
--Read SR
    when "00111" =>
     NCS     <= tNCS;
     tData   <= "00000101";
     t3state <= "00101";
     state   <= Write_tData;
```

```
    when "00101" =>
     t3state <= t2state;
     state   <= Read_tData;
--Wait for ready
    when "00100" =>
     NCS <= "11111111";
     if tData(0) = '1' then
      t2state <= "00100";
      state   <= Read_SR;
     else
      state <= t1state;
     end if;
--Write enable
    when "01100" =>
     NCS     <= tNCS;
     tData   <= "00000110";
     t3state <= "01101";
     state   <= Write_tData;
    when "01101" =>
     NCS   <= "11111111";
     state <= t2state;
--Read
    when "01111" =>
     NCS     <= tNCS;
     tData   <= "00000011";
     t3state <= "01110";
     state   <= Write_tData;
    when "01110" =>
     tData(7 downto 5) <= "000";
     tData(4 downto 0) <= lAddress(15 downto 11);
     t3state <= "01010";
     state   <= Write_tData;
    when "01010" =>
     tData   <= lAddress(10 downto 3);
     t3state <= "01011";
     state   <= Write_tData;
    when "01011" =>
     t3state <= "01001";
     state   <= Read_tData;
    when "01001" =>
     NCS   <= "11111111";
     rData <= tData;
     ClearStatus <= '1';
     state       <= "00110";
--Write
    when "01000" =>
     t2state <= "11000";
     state   <= Write_enable;
    when "11000" =>
     NCS     <= tNCS;
     tData   <= "00000010";
     t3state <= "11001";
     state   <= Write_tData;
    when "11001" =>
     tData(7 downto 5) <= "000";
     tData(4 downto 0) <= lAddress(15 downto 11);
     t3state <= "11011";
     state   <= Write_tData;
    when "11011" =>
     tData   <= lAddress(10 downto 3);
     t3state <= "11010";
     state   <= Write_tData;
    when "11010" =>
     tData   <= rData;
     t3state <= "11110";
```

```
     state    <= Write_tData;
    when "11110" =>
     NCS          <= "11111111";
     ClearStatus  <= '1';
     state        <= "00110";
--Read tData
    when "11111" =>
     SCK <= '1';
     tData(7 downto 1) <= tData(6 downto 0);
     tData(0) <= SO ;
     state <= "11101";
    when "11101" =>
     SCK <= '0';
     if Count = "111" then
      state <= t3state;
     else
      state <= Read_tData;
     end if;
     Count <= Count + 1;
--Write tData
    when "11100" =>
     SCK   <= '1';
     state <= "10100";
    when "10100" =>
     SCK <= '0';
     tData(7 downto 1) <= tData(6 downto 0);
     if Count = "111" then
      state <= t3state;
     else
      state <= Write_tData;
     end if;
     Count <= Count + 1;
    when others =>
   end case;
  end if;
 end process;

 SI   <= tData(7);
 Data <= rData when (lStatus = "00") and (R_nW = '1') else
         lData when                        R_nW = '1'   else
         "ZZZZZZZZ";
 Address <= lAddress when R_nW = '1' else
            "ZZZZZZZZZZZZZZZZ";
 Status  <= lStatus when R_nW = '1' else "ZZ";
end architecture a1;
```



E.2.18 Interface

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Interface is
 port(
  nReset      : in  std_logic;
  Clk         : in  std_logic;

  Bus_R_nW    : out   std_logic;
  Bus_Data    : inout std_logic_vector( 7 downto 0);
  Bus_Address : out   std_logic_vector(15 downto 0);
  Bus_Latch   : out   std_logic
 );
end entity Interface;
architecture a1 of Interface is
```

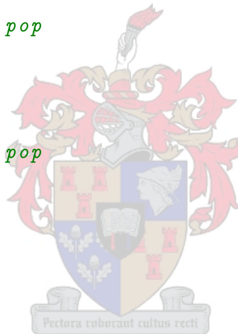
```
component ROM_Interface is
 port(
  data      : in  std_logic_vector( 7 downto 0);
  wraddress : in  std_logic_vector(13 downto 0);
  rdaddress : in  std_logic_vector(13 downto 0);
  wrclock   : in  std_logic;
  rdclock   : in  std_logic;
  q         : out std_logic_vector( 7 downto 0)
 );
end component ROM_Interface;

component ProgStack is
 port(
  data      : in  std_logic_vector(15 downto 0);
  wraddress : in  std_logic_vector( 7 downto 0);
  rdaddress : in  std_logic_vector( 7 downto 0);
  wrclock   : in  std_logic;
  rdclock   : in  std_logic;
  q         : out std_logic_vector(15 downto 0)
 );
end component ProgStack;

component InterfaceStack is
 port(
  nReset      : in  std_logic;

  Address     : in  std_logic_vector(2 downto 0);
  Input       : in  std_logic_vector(7 downto 0);
  Out0        : out std_logic_vector(7 downto 0);
  Out1        : out std_logic_vector(7 downto 0);
  OutA        : out std_logic_vector(7 downto 0);

  Latch       : in  std_logic;
  Task        : in  std_logic_vector(1 downto 0) --"00" = Store in s0
                                                 --"01" = Push onto stack
                                                 --"10" = Store in s1,
                                                 --        then Pop Stack
                                                 --"11" = Swop s0 and sA
 );
end component InterfaceStack;
```

```
component Arith is
 port(
  A        : in  std_logic_vector(7 downto 0);
  B        : in  std_logic_vector(7 downto 0);
  Task     : in  std_logic_vector(3 downto 0);
  Carry_In : in  std_logic;
  Zero_In  : in  std_logic;
  Y        : out std_logic_vector(7 downto 0);
  Carry    : out std_logic;
  Zero     : out std_logic
 );
end component Arith;

component Mul8 is
 port(
  A : in  std_logic_vector( 7 downto 0);
  B : in  std_logic_vector( 7 downto 0);
  Y : out std_logic_vector(15 downto 0)
 );
end component Mul8;

signal state     : std_logic_vector( 4 downto 0);
signal ret_state : std_logic_vector( 4 downto 0);
```

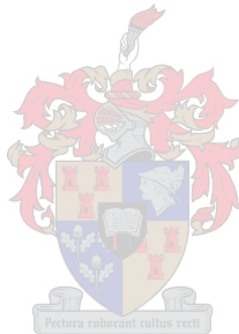
```
 signal PC          : std_logic_vector(15 downto 0); --Program counter
 signal PC_W        : std_logic_vector(15 downto 0); --Address to write
                                                     --to program ROM;
 signal Prog        : std_logic_vector( 7 downto 0); --Program ROM
 signal Prog_Data   : std_logic_vector( 7 downto 0); --Data to be written
                                                     --to program ROM;
 signal Prog_Latch : std_logic;

 signal Bus_tR_nW     : std_logic;
 signal Bus_tData     : std_logic_vector( 7 downto 0);
 signal Bus_tAddress : std_logic_vector(15 downto 0);

 signal S_Address : std_logic_vector(2 downto 0);
 signal S_Input   : std_logic_vector(7 downto 0);
 signal S_Out0    : std_logic_vector(7 downto 0);
 signal S_Out1    : std_logic_vector(7 downto 0);
 signal S_OutA    : std_logic_vector(7 downto 0);
 signal S_Latch   : std_logic;
 signal S_Task    : std_logic_vector(1 downto 0);

 signal PS_Top      : std_logic_vector(15 downto 0);
 signal PS_Latch    : std_logic;
 signal PS_Address : std_logic_vector( 7 downto 0);
 signal PS_Data     : std_logic_vector(15 downto 0);

 signal Mul_A : std_logic_vector( 7 downto 0);
 signal Mul_B : std_logic_vector( 7 downto 0);
 signal Mul_Y : std_logic_vector(15 downto 0);

 signal Arith_A         : std_logic_vector(7 downto 0);
 signal Arith_B         : std_logic_vector(7 downto 0);
 signal Arith_Task      : std_logic_vector(3 downto 0);
 signal Arith_Carry     : std_logic;
 signal Arith_Zero      : std_logic;

 signal Opcode   : std_logic_vector( 7 downto 0);
 signal Temp     : std_logic_vector( 7 downto 0);
 signal Temp2    : std_logic_vector( 7 downto 0);
 signal Div_Num : std_logic_vector(15 downto 0);

 signal Carry  : std_logic;
 signal Zero   : std_logic;

 signal Skip       : std_logic;

 signal PCp1 : std_logic_vector(15 downto 0); --PC + 1
 signal PCpS : std_logic_vector(15 downto 0); --PC + Skip count
begin
 ROM1      : ROM_Interface  port map(Prog_Data, PC_W(13 downto 0),
                                     PC(13 downto 0), Prog_Latch,
                                     not Clk, Prog);
 PC_Stack : ProgStack      port map(PS_Data, PS_Address, PS_Address,
                                     PS_Latch, not Clk, PS_Top);
 Stack1    : InterfaceStack port map(nReset, S_Address, S_Input, S_Out0,
                                     S_Out1, S_OutA,
                                     S_Latch, S_Task);
 Arith1    : Arith          port map(Arith_A, Arith_B, Arith_Task,
                                     Carry, Zero, S_Input,
                                     Arith_Carry, Arith_Zero);
 MUL       : Mul8           port map(Mul_A, Mul_B, Mul_Y);

 Skip <= ((not Carry) and Prog(7)) or ((not Zero) and Prog(6));
 PCp1 <= PC + '1';
 with Prog(5 downto 0) select
  PCpS <= PC + "10" when "101010" |
```

```
                         "101100" |
                         "111000" ,
          PC + "11" when "101011" |
                         "101101" |
                         "101110" |
                         "101111" |
                         "111001" |
                         "111010" |
                         "111011" |
                         "111100" |
                         "111101" |
                         "111110" |
                         "111111" ,
          PCp1      when others;

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state     <= "11101";
   ret_state <= "00000";

   PC         <= "0000000000000000";
   Prog_Data  <= "00000000";
   Prog_Latch <= '0';
   Opcode     <= "00000000";

   S_Address <= "000";
   S_Latch   <= '0';
   S_Task    <= "00";

   PS_Address <= "11111111";
   PS_Data    <= "0000000000000000";
   PS_Latch   <= '0';

   Bus_tR_nW     <= '1';
   Bus_tData     <= "00000000";
   Bus_tAddress  <= "0000000000000000";
   Bus_Latch     <= '0';

   Arith_A    <= "00000000";
   Arith_B    <= "00000000";
   Arith_Task <= "0000";

   Mul_A    <= "00000000";
   Mul_B    <= "00000000";

   Temp      <= "00001010"; --Startup wait for 10 cycles
   Temp2     <= "00000000";
   Div_Num   <= "0000000000000000";

   Carry     <= '0';
   Zero      <= '0';
  elsif rising_edge(Clk) then
   case state is
--Decode instruction and do first part
    when "00000" =>
     S_Latch     <= '0';
     PS_Latch    <= '0';
     Prog_Latch  <= '0';
     Bus_Latch   <= '0';
     if Skip = '1' then -- Instruction prefix = {c; z; c&z}
      PC <= PCpS;
      state <= "00000";
     else
      case Prog(5 downto 0) is
```

```
when "000000" |   -- adc
     "000001" => -- adc pop
 Arith_A <= S_Out1;
 Arith_B <= S_Out0;
 Arith_Task <= "0001";
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 PC <= PCp1;
 state <= "00001";
when "000010" |   -- add
     "000011" => -- add pop
 Arith_A <= S_Out1;
 Arith_B <= S_Out0;
 Arith_Task <= "0010";
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 PC <= PCp1;
 state <= "00001";
when "000100" |   -- and
     "000101" => -- and pop
 Arith_A <= S_Out1;
 Arith_B <= S_Out0;
 Arith_Task <= "0011";
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 PC <= PCp1;
 state <= "00001";
when "000110" |   -- div
     "000111" => -- div pop
 Arith_Task <= "0000";
 S_Address  <= "010";
 state      <= "00101";
when "001000" |   -- mul
     "001001" => -- mul pop
 Arith_Task <= "0000";
 Mul_A <= S_Out1;
 Mul_B <= S_Out0;
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 state <= "00010";
when "001010" => -- neg
 Arith_A <= S_Out0;
 Arith_Task <= "0100";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "001011" => -- not
 Arith_A <= S_Out0;
 Arith_Task <= "0101";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "001100" |   -- or
     "001101" => -- or pop
 Arith_A <= S_Out1;
 Arith_B <= S_Out0;
 Arith_Task <= "0110";
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 PC <= PCp1;
 state <= "00001";
when "001110" |   -- sbb
     "001111" => -- sbb pop
 Arith_A <= S_Out1;
 Arith_B <= S_Out0;
```



```
 Arith_Task <= "0111";
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 PC <= PCp1;
 state <= "00001";
when "010000" |  -- sub
     "010001" => -- sub pop
 Arith_A <= S_Out1;
 Arith_B <= S_Out0;
 Arith_Task <= "1000";
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 PC <= PCp1;
 state <= "00001";
when "010010" |  -- xor
     "010011" => -- xor pop
 Arith_A <= S_Out1;
 Arith_B <= S_Out0;
 Arith_Task <= "1001";
 S_Task(0) <= '0';
 S_Task(1) <= Prog(0);
 PC <= PCp1;
 state <= "00001";
when "010100" => -- rcl
 Arith_A <= S_Out0;
 Arith_Task <= "1010";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "010101" => -- rcr
 Arith_A <= S_Out0;
 Arith_Task <= "1011";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "010110" => -- rol
 Arith_A <= S_Out0;
 Arith_Task <= "1100";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "010111" => -- ror
 Arith_A <= S_Out0;
 Arith_Task <= "1101";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "011000" => -- shl
 Arith_A <= S_Out0;
 Arith_Task <= "1110";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "011001" => -- shr
 Arith_A <= S_Out0;
 Arith_Task <= "1111";
 S_Task <= "00";
 PC <= PCp1;
 state <= "00001";
when "011010" => -- cc
 Carry <= '0';
 PC <= PCp1;
 state <= "00000";
when "011011" => -- cz
 Zero <= '0';
```



```
 PC <= PCp1;
 state <= "00000";
when "011100" => -- nc
 Carry <= not Carry;
 PC <= PCp1;
 state <= "00000";
when "011101" => -- nz
 Zero <= not Zero;
 PC <= PCp1;
 state <= "00000";
when "011110" => -- sc
 Carry <= '1';
 PC <= PCp1;
 state <= "00000";
when "011111" => -- sz
 Zero <= '1';
 PC <= PCp1;
 state <= "00000";
when "100000" |  -- swp 0
     "100001" |  -- swp 1
     "100010" |  -- swp 2
     "100011" |  -- swp 3
     "100100" |  -- swp 4
     "100101" |  -- swp 5
     "100110" |  -- swp 6
     "100111" => -- swp 7
 Arith_Task <= "0000";
 S_Address <= Prog(2 downto 0);
 S_Task    <= "11";
 PC <= PCp1;
 state <= "00001";
when "101000" => -- pop
 Arith_Task <= "0000";
 Arith_A    <= S_Out1;
 S_Task     <= "10";
 PC         <= PCp1;
 state      <= "00001";
when "101001" => -- ret
 PC <= PS_Top;
 PS_Address <= PS_Address - '1';
 state <= "01001";
when "101010" |  -- call rel ??? | call ???
     "101011" => -- call abs ???
 PS_Data    <= PCpS;
 PS_Address <= PS_Address + '1';
 state <= "01011";
when "101100" |  -- jmp rel ??? | jmp ???
     "101101" => -- jmp abs ???
 Opcode <= Prog;
 PC     <= PCp1;
 state  <= "01100";
when "101110" |  -- ld ???
     "101111" => -- ld ref ???
 Arith_Task <= "0000";
 Opcode     <= Prog;
 PC         <= PCp1;
 Bus_tR_nW  <= '1';
 ret_state  <= "10010";
 state      <= "01110";
when "110000" |  -- lds 0
     "110001" |  -- lds 1
     "110010" |  -- lds 2
     "110011" |  -- lds 3
     "110100" |  -- lds 4
     "110101" |  -- lds 5
```

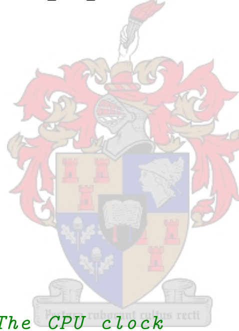
```
            "110110" |  -- lds 6
            "110111" => -- lds 7
        Arith_Task <= "0000";
        S_Address  <= Prog(2 downto 0);
        S_Task     <= "01";
        PC         <= PCp1;
        state      <= "10011";
       when "111000" => -- ldi ???
        Arith_Task <= "0000";
        S_Task     <= "01";
        PC         <= PCp1;
        state      <= "10100";
       when "111001" => -- ldpr ???
        Arith_Task <= "0000";
        PC         <= PCp1;
        Opcode     <= "00000001"; --Reference
        Bus_tR_nW  <= '1';
        ret_state  <= "10101";
        state      <= "01110";
       when "111010" |  -- stpr
            "111011" => -- stpr pop
        Arith_Task <= "0000";
        PC         <= PCp1;
        Opcode(1)  <= Prog(0); --Pop
        Opcode(0)  <= '1'; --Reference
        Bus_tR_nW  <= '1';
        ret_state  <= "11000";
        state      <= "01110";
       when "111100" |  -- st ???
            "111101" |  -- st [???]
            "111110" |  -- st pop ???
            "111111" => -- st pop [???]
        Arith_Task <= "0000";
        Opcode     <= Prog;
        PC         <= PCp1;
        Bus_tR_nW  <= '1';
        ret_state  <= "11011";
        state      <= "01110";
       when others =>
      end case;
     end if;
--Latch arithmetic stack and set flags
    when "00001" =>
     S_Latch  <= '1';
     Carry    <= Arith_Carry;
     Zero     <= Arith_Zero;
     state    <= "00000";
--Mul
    when "00010" =>
     Arith_A <= Mul_Y(15 downto 8);
     state <= "00011";
    when "00011" =>
     S_Latch <= '1';
     state <= "00100";
    when "00100" =>
     S_Latch <= '0';
     Arith_A <= Mul_Y(7 downto 0);
     if Mul_Y = "0000000000000000" then
      Zero <= '1';
     else
      Zero <= '0';
     end if;
     S_Task   <= "01";
     PC <= PCp1;
     state    <= "00001";
```

```
--Div
    when "00101" =>
     Div_Num(15 downto 8) <= S_OutA;
     Div_Num( 7 downto 0) <= S_Out1;
     Temp       <= "01111111";
     Temp2      <= "01000000";
     Mul_A      <= "10000000";
     Mul_B      <= S_Out0;
     state <= "00110";
    when "00110" =>
     if Mul_Y > Div_Num then
      Mul_A <= (Mul_A and Temp) or Temp2;
     else
      Mul_A <= Mul_A or Temp2;
     end if;
     if Temp2 = "00000000" then
      if Mul_A = "00000000" then
       Zero <= '1';
      else
       Zero <= '0';
      end if;
      S_Task(0) <= '0';
      S_Task(1) <= Prog(0);
      PC <= PCp1;
      state <= "00111";
     else
      state <= "00110";
     end if;
     Temp (6 downto 0) <= Temp (7 downto 1); Temp(7) <= '1';
     Temp2(6 downto 0) <= Temp2(7 downto 1);
--Pop or store twice - to get the LSb and stack correct
    when "00111" =>
     S_Latch <= '1';
     state    <= "01000";
    when "01000" =>
     S_Latch <= '0';
     Arith_A <= Mul_A;
     state    <= "00001";
--Latch new program stack address
    when "01001" =>
     PS_Data <= PS_Top;
     state    <= "01010";
    when "01010" =>
     PS_Latch <= '1';
     state     <= "00000";
--Latch PC onto stack;
    when "01011" =>
     PS_Latch <= '1';
     Opcode    <= Prog;
     PC        <= PCp1;
     state     <= "01100";
--Jump
    when "01100" =>
     Temp <= Prog;
     if Opcode(0) = '1' then
      PC     <= PCp1;
      state <= "01101";
     else
      if Prog(7) = '1' then --negative jump
       PC <= (PC + "1111111011111111") + Prog; --PC - 1 + sign extend + offset
      else
       PC <= (PC + "1111111111111111") + Prog; --PC - 1 + offset
      end if;
      state <= "00000";
     end if;
```

```
    when "01101" =>
     PC( 7 downto 0) <= Temp;
     PC(15 downto 8) <= Prog;
     state <= "00000";
--ld
    when "01110" =>
     Temp <= Prog;
     PC <= PCp1;
     state <= "01111";
    when "01111" =>
     Bus_tAddress(15 downto 8) <= Prog;
     Bus_tAddress( 7 downto 0) <= Temp;
     state <= "10000";
    when "10000" =>
     if Opcode(0) = '1' then
      Temp <= Bus_Data;
      Bus_tAddress <= Bus_tAddress + '1';
      state <= "10001";
     else
      state <= ret_state;
     end if;
    when "10001" =>
     Bus_tAddress(15 downto 8) <= Bus_Data;
     Bus_tAddress( 7 downto 0) <= Temp;
     state <= ret_state;
    when "10010" =>
     Arith_A <= Bus_Data;
     S_Task  <= "01";
     PC      <= PCp1;
     state   <= "00001";
--Push S_OutA onto stack;
    when "10011" =>
     Arith_A <= S_OutA;
     state <= "00001";
--Push Prog onto stack;
    when "10100" =>
     Arith_A <= Prog;
     PC      <= PCp1;
     state   <= "00001";
--ldpr
    when "10101" =>
     PC_W <= PCp1;
     PC <= Bus_tAddress;
     state <= "10111";
    when "10111" =>
     Arith_A <= Prog;
     S_Task  <= "01";
     PC      <= PC_W;
     state   <= "00001";
--stpr
    when "11000" =>
     PC_W <= Bus_tAddress;
     Prog_Data <= S_Out0;
     PC <= PCp1;
     state <= "11010";
    when "11010" =>
     Prog_Latch <= '1';
     if Opcode(1) = '1' then
      Arith_A <= S_Out1;
      S_Task  <= "10";
      state   <= "00001";
     else
      state <= "00000";
     end if;
--st
```

```
    when "11011" =>
     Bus_tR_nW <= '0';
     Bus_tData <= S_Out0;
     PC        <= PCp1;
     state     <= "11100";
    when "11100" =>
     Bus_Latch <= '1';
     if Opcode(1) = '1' then
      Arith_A <= S_Out1;
      S_Task  <= "10";
      state   <= "00001";
     else
      state <= "00000";
     end if;
--Startup wait
    when "11101" =>
     if Temp = "00000000" then
      state <= "00000";
     else
      Temp <= Temp - '1';
     end if;
    when others =>
   end case;
  end if;
 end process;

 Bus_R_nW    <= Bus_tR_nW;
 Bus_Data    <= Bus_tData when Bus_tR_nW = '0' else "ZZZZZZZZ";
 Bus_Address <= Bus_tAddress;
end architecture a1;
```

E.2.19 InterfaceBus



```
library ieee;
use ieee.std_logic_1164.all;

entity InterfaceBus is
 port(
  nReset : in    std_logic;
  Clk    : in    std_logic; --The CPU clock

  R_nW    : in    std_logic;
  Data    : inout std_logic_vector(7 downto 0);
  Address : in    std_logic_vector(15 downto 0);
  Latch   : in    std_logic;

  LCD_Line    : out std_logic_vector( 1 downto 0);
  LCD_Address : out std_logic_vector( 4 downto 0);
  LCD_Data    : out std_logic_vector( 7 downto 0);
  LCD_Latch   : out std_logic;
  LCD_BL      : out std_logic;

  EEPROM_R_nW     : out   std_logic;
  EEPROM_Data     : inout std_logic_vector( 7 downto 0);
  EEPROM_Address  : inout std_logic_vector(15 downto 0);
  EEPROM_Status   : inout std_logic_vector( 1 downto 0); --read:write
  EEPROM_DLatch   : out   std_logic;
  EEPROM_A0Latch  : out   std_logic;
  EEPROM_A1Latch  : out   std_logic;
  EEPROM_SLatch   : out   std_logic;

  LED_Error  : out std_logic;
  LED_Locked : out std_logic;
  LED_F_High : out std_logic;
```

```
  LED_F_Low   : out std_logic;

  COMM_R_nW    : out   std_logic;
  COMM_Tx      : inout std_logic_vector(7 downto 0);
  COMM_Rx      : in    std_logic_vector(7 downto 0);
  COMM_Status  : inout std_logic_vector(7 downto 0);
  COMM_TxLatch : out   std_logic;
  COMM_SLatch  : out   std_logic;

  Keypad     : in  std_logic_vector(7 downto 0);
  Keypad_Ack : out std_logic;

  RealTime : in std_logic_vector(7 downto 0)
 );
end entity InterfaceBus;

architecture a1 of InterfaceBus is
 component RAM_M4K is
  port(
   data      : in  std_logic_vector(7 downto 0);
   wraddress : in  std_logic_vector(8 downto 0);
   rdaddress : in  std_logic_vector(8 downto 0);
   wrclock   : in  std_logic;
   rdclock   : in  std_logic;
   q         : out std_logic_vector(7 downto 0)
  );
 end component RAM_M4K;

 component RAM_LCD is
  port(
   data      : in  std_logic_vector(7 downto 0);
   wraddress : in  std_logic_vector(6 downto 0);
   rdaddress : in  std_logic_vector(6 downto 0);
   wrclock   : in  std_logic;
   rdclock   : in  std_logic;
   q         : out std_logic_vector(7 downto 0)
  );
 end component RAM_LCD;

 signal RAM_Clk         : std_logic;
 signal tData           : std_logic_vector( 7 downto 0);
 signal tLED            : std_logic_vector( 7 downto 0);
 signal tEEPROM_Status  : std_logic_vector( 7 downto 0);
 signal tCOMM_Tx        : std_logic_vector( 7 downto 0);
 signal tCOMM_Status    : std_logic_vector( 7 downto 0);
 signal RAM0_Latch      : std_logic;
 signal RAM0_Data       : std_logic_Vector( 7 downto 0);
-- signal RAM1_Latch      : std_logic;
-- signal RAM1_Data       : std_logic_Vector( 7 downto 0);
-- signal RAM2_Latch      : std_logic;
-- signal RAM2_Data       : std_logic_Vector( 7 downto 0);
-- signal RAM3_Latch      : std_logic;
-- signal RAM3_Data       : std_logic_Vector( 7 downto 0);
-- signal RAM4_Latch      : std_logic;
-- signal RAM4_Data       : std_logic_Vector( 7 downto 0);
-- signal RAM5_Latch      : std_logic;
-- signal RAM5_Data       : std_logic_Vector( 7 downto 0);
-- signal RAM6_Latch      : std_logic;
-- signal RAM6_Data       : std_logic_Vector( 7 downto 0);
-- signal RAM7_Latch      : std_logic;
-- signal RAM7_Data       : std_logic_Vector( 7 downto 0);
 signal tLCD_Latch      : std_logic;
 signal tLCD_Data       : std_logic_vector( 7 downto 0);
begin
 RAM_Clk <= not Clk;
```

```
 M4K0 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
                         RAM0_Latch, RAM_Clk, RAM0_Data);
-- M4K1 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
--                         RAM1_Latch, RAM_Clk, RAM1_Data);
-- M4K2 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
--                         RAM2_Latch, RAM_Clk, RAM2_Data);
-- M4K3 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
--                         RAM3_Latch, RAM_Clk, RAM3_Data);
-- M4K4 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
--                         RAM4_Latch, RAM_Clk, RAM4_Data);
-- M4K5 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
--                         RAM5_Latch, RAM_Clk, RAM5_Data);
-- M4K6 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
--                         RAM6_Latch, RAM_Clk, RAM6_Data);
-- M4K7 : RAM_M4K port map(Data, Address(8 downto 0), Address(8 downto 0),
--                         RAM7_Latch, RAM_Clk, RAM7_Data);
 LCD_RAM : RAM_LCD port map(Data, Address(6 downto 0), Address(6 downto 0),
                            tLCD_Latch, RAM_Clk, tLCD_Data);

 RAM0_Latch <= Latch when Address(15 downto 9) = "0000000" else '0';
-- RAM1_Latch <= Latch when Address(15 downto 9) = "0000001" else '0';
-- RAM2_Latch <= Latch when Address(15 downto 9) = "0000010" else '0';
-- RAM3_Latch <= Latch when Address(15 downto 9) = "0000011" else '0';
-- RAM4_Latch <= Latch when Address(15 downto 9) = "0000100" else '0';
-- RAM5_Latch <= Latch when Address(15 downto 9) = "0000101" else '0';
-- RAM6_Latch <= Latch when Address(15 downto 9) = "0000110" else '0';
-- RAM7_Latch <= Latch when Address(15 downto 9) = "0000111" else '0';

 LCD_Line    <= Address(6 downto 5);
 LCD_Address <= Address(4 downto 0);
 LCD_Data    <= Data;
 tLCD_Latch  <= Latch when Address(15 downto 7) = "100000000" else '0';
                                                  --8000 -> 807F
 LCD_Latch   <= tLCD_Latch;

 EEPROM_R_nW     <= R_nW;
 EEPROM_Data     <= Data when R_nW = '0' else "ZZZZZZZZ";
 EEPROM_Address( 7 downto 0) <= Data when R_nW = '0' else "ZZZZZZZZ";
 EEPROM_Address(15 downto 8) <= Data when R_nW = '0' else "ZZZZZZZZ";
 EEPROM_Status   <= Data(1 downto 0) when R_nW = '0' else "ZZ";
 EEPROM_DLatch   <= Latch when Address = "1000000010001011" else '0'; --808B
 EEPROM_A0Latch  <= Latch when Address = "1000000010001001" else '0'; --8089
 EEPROM_A1Latch  <= Latch when Address = "1000000010001010" else '0'; --808A
 EEPROM_SLatch   <= Latch when Address = "1000000010001100" else '0'; --808C
 tEEPROM_Status(7 downto 2) <= "000000";
 tEEPROM_Status(1 downto 0) <= EEPROM_Status;

 COMM_R_nW    <= R_nW;
 COMM_Tx      <= Data  when R_nW = '0' else "ZZZZZZZZ";
 COMM_Status  <= Data  when R_nW = '0' else "ZZZZZZZZ";
 COMM_TxLatch <= Latch when Address = "1000000010000000" else '0'; --8080
 COMM_SLatch  <= Latch when Address = "1000000010000010" else '0'; --8082

 Keypad_Ack <= Latch when Address = "1000000010001110" else '0'; --808E

 process(Latch, nReset) is
 begin
  if nReset = '0' then
   tLED          <= "00000000";
  elsif rising_edge(Latch) then
   case Address is
    when "1000000010001101" => --808D
     tLED <= Data;
    when others =>
   end case;
```

```
  end if;
 end process;

  tData <= RAM0_Data when Address(15 downto 9) = "0000000" else
--           RAM1_Data when Address(15 downto 9) = "0000001" else
--           RAM2_Data when Address(15 downto 9) = "0000010" else
--           RAM3_Data when Address(15 downto 9) = "0000011" else
--           RAM4_Data when Address(15 downto 9) = "0000100" else
--           RAM5_Data when Address(15 downto 9) = "0000101" else
--           RAM6_Data when Address(15 downto 9) = "0000110" else
--           RAM7_Data when Address(15 downto 9) = "0000111" else

           tLCD_Data when Address(15 downto 7) = "100000000" else

           EEPROM_Data                 when Address = "1000000010001011" else
           EEPROM_Address( 7 downto 0) when Address = "1000000010001001" else
           EEPROM_Address(15 downto 8) when Address = "1000000010001010" else
           tEEPROM_Status              when Address = "1000000010001100" else

           tLED when Address = "1000000010001101" else

           COMM_Tx     when Address = "1000000010000000" else
           COMM_Rx     when Address = "1000000010000001" else
           COMM_Status when Address = "1000000010000010" else

           Keypad when Address = "1000000010001110" else

           RealTime when Address = "1000000010010101" else
           "00000000";

 Data <= tData when R_nW = '1' else "ZZZZZZZZ";

 LCD_BL     <= tLED(4);
 LED_F_Low  <= tLED(3);
 LED_F_High <= tLED(2);
 LED_Locked <= tLED(1);
 LED_Error  <= tLED(0);
end architecture a1;
```

E.2.20 InterfaceStack

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity InterfaceStack is
 port(
  nReset      : in  std_logic;

  Address     : in  std_logic_vector(2 downto 0);
  Input       : in  std_logic_vector(7 downto 0);
  Out0        : out std_logic_vector(7 downto 0);
  Out1        : out std_logic_vector(7 downto 0);
  OutA        : out std_logic_vector(7 downto 0);

  Latch       : in  std_logic;
  Task        : in  std_logic_vector(1 downto 0) --"00" = Store in s0
                                                 --"01" = Push onto stack
                                                 --"10" = Store is s1,
                                                 --       then Pop Stack
                                                 --"11" = Swop s0 and sA
 );
end entity InterfaceStack;
```

```
architecture a1 of InterfaceStack is
 signal s0 : std_logic_vector(7 downto 0);
 signal s1 : std_logic_vector(7 downto 0);
 signal s2 : std_logic_vector(7 downto 0);
 signal s3 : std_logic_vector(7 downto 0);
 signal s4 : std_logic_vector(7 downto 0);
 signal s5 : std_logic_vector(7 downto 0);
 signal s6 : std_logic_vector(7 downto 0);
 signal s7 : std_logic_vector(7 downto 0);
 signal sA : std_logic_vector(7 downto 0);
begin
 Out0 <= s0;
 Out1 <= s1;
 with Address select
  sA <= s0 when "000",
        s1 when "001",
        s2 when "010",
        s3 when "011",
        s4 when "100",
        s5 when "101",
        s6 when "110",
        s7 when "111",
        "XXXXXXXX" when others;
 OutA <= sA;

 process(nReset, Latch) is
 begin
  if nReset = '0' then
   s0 <= "00000000";
   s1 <= "00000000";
   s2 <= "00000000";
   s3 <= "00000000";
   s4 <= "00000000";
   s5 <= "00000000";
   s6 <= "00000000";
   s7 <= "00000000";
  elsif rising_edge(Latch) then
   case Task is
    when "00" =>
     s0 <= Input;
    when "01" =>
     s1 <= s0;
     s2 <= s1;
     s3 <= s2;
     s4 <= s3;
     s5 <= s4;
     s6 <= s5;
     s7 <= s6;
     s0 <= Input;
    when "10" =>
     s0 <= Input;
     s1 <= s2;
     s2 <= s3;
     s3 <= s4;
     s4 <= s5;
     s5 <= s6;
     s6 <= s7;
    when "11" =>
     case Address is
      when "001" =>
       s1 <= s0;
      when "010" =>
       s2 <= s0;
      when "011" =>
       s3 <= s0;
```

```
      when "100" =>
       s4 <= s0;
      when "101" =>
       s5 <= s0;
      when "110" =>
       s6 <= s0;
      when "111" =>
       s7 <= s0;
      when others =>
     end case;
     s0 <= sA;
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.2.21 Keypad

```
library ieee;
use ieee.std_logic_1164.all;

entity Keypad is
 port(
  nReset : in  std_logic;
  Clk    : in  std_logic;
  Status : out std_logic_vector(15 downto 0);

  X      : in  std_logic_vector( 4 downto 1);
  Y      : out std_logic_vector( 4 downto 1)
 );
end entity Keypad;

architecture a1 of Keypad is
 signal state : std_logic_vector(3 downto 0);
begin
 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state <= "0001";
  elsif rising_edge(Clk) then
   state(3 downto 1) <= state(2 downto 0);
   state(         0) <= state(         3);
  elsif falling_edge(Clk) then
   case state is
    when "0001" =>
     Status( 3 downto  0) <= X;
    when "0010" =>
     Status( 7 downto  4) <= X;
    when "0100" =>
     Status(11 downto  8) <= X;
    when "1000" =>
     Status(15 downto 12) <= X;
    when others =>
   end case;
  end if;
 end process;

 Y(1) <= '1' when state(0) = '1' else 'Z';
 Y(2) <= '1' when state(1) = '1' else 'Z';
 Y(3) <= '1' when state(2) = '1' else 'Z';
 Y(4) <= '1' when state(3) = '1' else 'Z';
end architecture a1;
```

E.2.22 Keypad_Driver

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Keypad_Driver is
 port(
  nReset : in  std_logic;
  Clk    : in  std_logic;

  Data   : out std_logic_vector(7 downto 0); --Data(7) = Empty;
                                             --Data(3 downto 0) = Key
  Ack    : in  std_logic;

  X      : in  std_logic_vector(4 downto 1);
  Y      : out std_logic_vector(4 downto 1)
 );
end entity Keypad_Driver;

architecture a1 of Keypad_Driver is
 component Keypad is
  port(
   nReset : in  std_logic;
   Clk    : in  std_logic;
   Status : out std_logic_vector(15 downto 0);

   X      : in  std_logic_vector( 4 downto 1);
   Y      : out std_logic_vector( 4 downto 1)
  );
 end component Keypad;

 signal Keys    : std_logic_vector(15 downto 0);
 signal state   : std_logic;
 signal KeyDown : std_logic;
 signal Key     : std_logic_vector( 3 downto 0);

 signal FIFO0 : std_logic_vector(3 downto 0);
 signal FIFO1 : std_logic_vector(3 downto 0);
 signal FIFO2 : std_logic_vector(3 downto 0);
 signal FIFO3 : std_logic_vector(3 downto 0);
 signal FIFO4 : std_logic_vector(3 downto 0);
 signal FIFO5 : std_logic_vector(3 downto 0);
 signal FIFO6 : std_logic_vector(3 downto 0);
 signal FIFO7 : std_logic_vector(3 downto 0);

 signal W     : std_logic_vector(2 downto 0);
 signal Wp1   : std_logic_vector(2 downto 0);
 signal R     : std_logic_vector(2 downto 0);
 signal Empty : std_logic;
begin
 Keypad1 : Keypad port map(nReset, Clk, Keys, X, Y);

 KeyDown <= '0' when Keys = "0000000000000000" else '1';

 Wp1 <= W + '1';
 Empty <= '1' when W = R else '0';

 with Keys select
  Key <= "0000" when "0010000000000000",
         "0001" when "0000000000000001",
         "0010" when "0000000000000010",
         "0011" when "0000000000000100",
         "0100" when "0000000000010000",
         "0101" when "0000000000100000",
```

```
         "0110" when "0000000001000000",
         "0111" when "0000000100000000",
         "1000" when "0000001000000000",
         "1001" when "0000010000000000",
         "1010" when "0000000000001000",
         "1011" when "0000000010000000",
         "1100" when "0000100000000000",
         "1101" when "1000000000000000",
         "1110" when "0001000000000000",
         "1111" when "0100000000000000",
         "0000" when others;

process(Clk, nReset) is
begin
 if nReset = '0' then
  state <= '0';
  FIFO0 <= "0000";
  FIFO1 <= "0000";
  FIFO2 <= "0000";
  FIFO3 <= "0000";
  FIFO4 <= "0000";
  FIFO5 <= "0000";
  FIFO6 <= "0000";
  FIFO7 <= "0000";
  W <= "000";
 elsif rising_edge(Clk) then
  case state is
   when '0' =>
    if (KeyDown = '1') and (Wp1 /= R) then
     case W is
      when "000" =>
       FIFO0 <= Key;
      when "001" =>
       FIFO1 <= Key;
      when "010" =>
       FIFO2 <= Key;
      when "011" =>
       FIFO3 <= Key;
      when "100" =>
       FIFO4 <= Key;
      when "101" =>
       FIFO5 <= Key;
      when "110" =>
       FIFO6 <= Key;
      when "111" =>
       FIFO7 <= Key;
      when others =>
     end case;
     W <= Wp1;
     state <= '1';
    end if;
   when '1' =>
    if KeyDown = '0' then
     state <= '0';
    end if;
   when others =>
  end case;
 end if;
end process;

process(Ack, nReset) is
begin
 if nReset = '0' then
  R <= "000";
 elsif rising_edge(Ack) then
```

```
   if Empty = '0' then
    R <= R + 1;
   end if;
  end if;
 end process;

 Data(          7) <= Empty;
 Data(6 downto 4) <= "000";
 with R select
  Data(3 downto 0) <= FIFO0  when "000",
                      FIFO1  when "001",
                      FIFO2  when "010",
                      FIFO3  when "011",
                      FIFO4  when "100",
                      FIFO5  when "101",
                      FIFO6  when "110",
                      FIFO7  when "111",
                      "0000" when others;
end architecture a1;
```

E.2.23 Latch1

```
library ieee;
use ieee.std_logic_1164.all;

entity Latch1 is
 port(
  nReset     : in  std_logic;
  Latch      : in  std_logic;

  Data       : in  std_logic;
  lData      : out std_logic
 );
end entity Latch1;

architecture a1 of Latch1 is
begin
 process(Latch, nReset) is
 begin
  if nReset = '0' then
   lData <= '0';
  elsif rising_edge(Latch) then
   lData <= Data;
  end if;
 end process;
end architecture a1;
```

E.2.24 Latch8

```
library ieee;
use ieee.std_logic_1164.all;

entity Latch8 is
 port(
  nReset     : in  std_logic;
  Latch      : in  std_logic;

  Data       : in  std_logic_vector(7 downto 0);
  lData      : out std_logic_vector(7 downto 0)
 );
end entity Latch8;

architecture a1 of Latch8 is
```

```
begin
 process(Latch, nReset) is
 begin
  if nReset = '0' then
   lData <= "00000000";
  elsif rising_edge(Latch) then
   lData <= Data;
  end if;
 end process;
end architecture a1;
```

E.2.25 LCD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity LCD is
 port(
  nReset  : in  std_logic;
  Clk     : in  std_logic;

  Line    : in  std_logic_vector(1 downto 0);
  Address : in  std_logic_vector(4 downto 0);
  Data    : in  std_logic_vector(7 downto 0);
  Latch   : in  std_logic;

  RS      : out std_logic;
  E       : out std_logic;
  D       : out std_logic_vector(7 downto 0)
 );
end entity LCD;

architecture a1 of LCD is
 component RAM_LCD is
  port(
   data      : in  std_logic_vector(7 downto 0);
   wraddress : in  std_logic_vector(6 downto 0);
   rdaddress : in  std_logic_vector(6 downto 0);
   wrclock   : in  std_logic;
   rdclock   : in  std_logic;
   q         : out std_logic_vector(7 downto 0)
  );
 end component RAM_LCD;

 signal state      : std_logic_vector( 4 downto 0);
 signal retstate   : std_logic_vector( 4 downto 0);
 signal count      : std_logic_vector(18 downto 0);

 signal WrAddress1 : std_logic_vector(6 downto 0);
 signal RdAddress1 : std_logic_vector(6 downto 0);

 signal RdLine     : std_logic_vector(1 downto 0);
 signal RdAddress  : std_logic_vector(4 downto 0);
 signal RdLatch    : std_logic;
 signal RdData     : std_logic_vector(7 downto 0);
begin
 WrAddress1(6 downto 5) <= Line;
 WrAddress1(4 downto 0) <= Address;
 RdAddress1(6 downto 5) <= RdLine;
 RdAddress1(4 downto 0) <= RdAddress;
 RAM : RAM_LCD port map(Data, WrAddress1, RdAddress1, Latch, RdLatch, RdData);

 process(Clk, nReset) is
```

```
 begin
  if nReset = '0' then
   state <= "00000";
   RS    <= '0';
   E     <= '1';
   D     <= "00000000";
   count <= "0000000000000000000";

   RdLine    <= "00";
   RdAddress <= "00000";
   RdLatch   <= '0';
  elsif rising_edge(Clk) then
   case state is
--INIT:
    when "00000" =>
     count    <= "1001010100000011000"; --305176 (100ms)
     retstate <= "00010";
     state    <= "00001";
    when "00001" =>
     if count = "0000000000000000000" then
      state <= retstate;
     end if;
     count <= count - '1';
--8-bit, 2-line, 5x10
    when "00010" =>
     E     <= '1';
     state <= "00110";
    when "00110" =>
     D     <= "00111111"; --5x10
     state <= "00111";
    when "00111" =>
     E        <= '0';
     count    <= "0000000000010011001"; --153 (50us)
     retstate <= "00101";
     state    <= "00001";
--Display on, cursor off, blinking off
    when "00101" =>
     E     <= '1';
     state <= "00100";
    when "00100" =>
     D     <= "00001100";
     state <= "01100";
    when "01100" =>
     E        <= '0';
     count    <= "0000000000010011001"; --153 (50us)
     retstate <= "01101";
     state    <= "00001";
--Display clear
    when "01101" =>
     E     <= '1';
     state <= "01111";
    when "01111" =>
     D     <= "00000001";
     state <= "01110";
    when "01110" =>
     E        <= '0';
     count    <= "0000001011111011000"; --6104 (2ms)
     retstate <= "01011";
     state    <= "00001";
--UPDATE
--Set cursor to start of line
    when "01011" =>
     E         <= '1';
     RS        <= '0';
     RdAddress <= "00000";
```

Pectora roborant cultus recti

```
     state      <= "01001";
    when "01001" =>
     case RdLine is
      when "00" =>
       D <= "10000000";
      when "01" =>
       D <= "11000000";
      when "10" =>
       D <= "10010100";
      when "11" =>
       D <= "11010100";
      when others =>
     end case;
     state <= "01000";
    when "01000" =>
     E         <= '0';
     count     <= "0000000000010011001"; --153 (50us)
     retstate  <= "11000";
     state     <= "00001";
--Write line to LCD
    when "11000" =>
     if RdAddress /= "10100" then --Equivalent to "RdAddress < 20"
      RdLatch  <= '1';
      state <= "11001";
     else
      RdLine <= RdLine + 1;
      state <= "01011";
     end if;
    when "11001" =>
     RdLatch    <= '0';
     RdAddress <= RdAddress + 1;
     state <= "11011";
    when "11011" =>
     E        <= '1';
     RS       <= '1';
     state    <= "11010";
    when "11010" =>
     D      <= RdData;
     state <= "11110";
    when "11110" =>
     E         <= '0';
     count     <= "0000000000010011001"; --153 (50us)
     retstate  <= "11000";
     state     <= "00001";
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.2.26 Probe

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Probe is
 port(
  nReset: in  std_logic;
  Clk    : in  std_logic;

  DDS_Reset      : out std_logic;
  DDS_IO_Update  : out std_logic;
  DDS_SCLK       : out std_logic;
```

```
  DDS_nCS1      : out std_logic;
  DDS_nCS2      : out std_logic;
  DDS_SDIO1     : out std_logic;
  DDS_SDIO2     : out std_logic
 );
end entity Probe;

architecture a1 of Probe is
 signal state : std_logic_vector(2 downto 0);
 signal ret   : std_logic_vector(2 downto 0);

 signal Freq1  : std_logic_vector(31 downto 0);
 signal Freq2  : std_logic_vector(31 downto 0);
 signal Ampl1  : std_logic_vector(13 downto 0);
 signal Ampl2  : std_logic_vector(13 downto 0);

 signal Temp1   : std_logic_vector(39 downto 0);
 signal Temp2   : std_logic_vector(39 downto 0);
 signal Len    : std_logic_vector( 5 downto 0);
begin
 DDS_nCS1  <= '0';
 DDS_nCS2  <= '0';
 DDS_Reset <= not nReset;

-- Freq1  <= "00000011001100110011001100110011"; -- 5 000kHz (1.47A)
-- Freq2  <= "00000011001000101101000011100101"; -- 4 900kHz
-- Freq1  <= "00000110011001100110011001100110"; --10 000kHz (2.94A)
-- Freq2  <= "00000110010101100000001000011001"; -- 9 900kHz
 Freq1  <= "00001100110011001100110011001101"; --20 000kHz (5.87A)
 Freq2  <= "00001100101111000110101001111111"; --19 900kHz
-- Freq1  <= "00100000000000000000000000000000"; --50 000kHz (5.87A)
-- Freq2  <= "00011111111011111001110110110010"; --49 900kHz

 Ampl1  <= "00000010100100"; --10uA (peak-peak)
-- Ampl1  <= "00110011001101"; --200uA (peak-peak)
-- Ampl1  <= "11111111111111"; --1mA (peak-peak)
 Ampl2  <= "11111111111111"; --200mV

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state <= "000";
   DDS_SCLK <= '0';
   DDS_IO_Update <= '0';
  elsif rising_edge(Clk) then
   case state is
    when "000" =>
     Temp1(39 downto 32) <= "00000000"; --CFR1
     Temp2(39 downto 32) <= "00000000"; --CFR1
     Len <= "101000"; --40
     Temp1(31 downto 0) <= "00000010000000000000100000000000";
     Temp2(31 downto 0) <= "00000010100000000000100000000010";
     ret   <= "001";
     state <= "110";
    when "001" =>
     Temp1(39 downto 32) <= "00000001"; --CFR2
     Temp2(39 downto 32) <= "00000001"; --CFR2
     Len <= "100000"; --32
     Temp1(31 downto 8) <= "000000000000100010000100";
                    --for 25MHz chrystal, 400MHz clock
     Temp2(31 downto 8) <= "000000000000100010000100";
                    --for 25MHz chrystal, 400MHz clock
     ret   <= "010";
     state <= "110";
    when "010" =>
```

```
     DDS_IO_Update <= '0';
     Temp1(39 downto 32) <= "00000010"; --Amplitude
     Temp2(39 downto 32) <= "00000010"; --Amplitude
     Len <= "011000"; --24
     Temp1(31 downto 30) <= "00";
     Temp2(31 downto 30) <= "00";
     Temp1(29 downto 16) <= Ampl1;
     Temp2(29 downto 16) <= Ampl2;
     ret   <= "011";
     state <= "110";
    when "011" =>
     Temp1(39 downto 32) <= "00000100"; --Frequency
     Temp2(39 downto 32) <= "00000100"; --Frequency
     Len <= "101000"; --40
     Temp1(31 downto 0) <= Freq1;
     Temp2(31 downto 0) <= Freq2;
     ret   <= "100";
     state <= "110";
    when "100" =>
     DDS_IO_Update <= '0';
     state <= "101";
    when "101" =>
     DDS_IO_Update <= '1';
     state <= "101";
    when "110" =>
     DDS_SCLK <= '0';
     Len <= Len - '1';
     DDS_SDIO1 <= Temp1(39);
     DDS_SDIO2 <= Temp2(39);
     Temp1(39 downto 1) <= Temp1(38 downto 0);
     Temp2(39 downto 1) <= Temp2(38 downto 0);
     state <= "111";
    when "111" =>
     DDS_SCLK <= '1';
     if Len = "000000" then
      state <= ret;
     else
      state <= "110";
     end if;
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.2.27 PWM1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity PWM1 is
 port(
  nReset    : in  std_logic;
  Clk       : in  std_logic; --97 656 250 Hz
  Duty      : in  std_logic_vector(20 downto 0);

  Delay_Out : out std_logic;
  Delay_In  : in  std_logic;
  Delay     : out std_logic_vector(9 downto 0);

  S1        : out std_logic;
  S2        : out std_logic
 );
```

```
end entity PWM1;

architecture a1 of PWM1 is
 constant Max : std_logic_vector(10 downto 0) := "11110011010"; --95%
-- constant Max : std_logic_vector(10 downto 0) := "11001100110"; --80%

 component Counter11 is
  port(
   nReset: in  std_logic;
   Clk   : in  std_logic;
   Q     : out std_logic_vector(10 downto 0)
  );
 end component Counter11;

 component Alternator is
  port(
   nReset: in  std_logic;
   PWM   : in  std_logic;
   S1    : out std_logic;
   S2    : out std_logic
  );
 end component Alternator;

 signal Count   : std_logic_vector(10 downto 0);
 signal greater : std_logic;
begin
 Counter : Counter11 port map(nReset, Clk, Count);

 greater <= '1' when (Duty(20 downto 10) > Count) and
                     (Max                > Count) else '0';

 process(Clk) is
 begin
  if rising_edge(Clk) then
   Delay_Out <= greater;
  end if;
 end process;

 Delay(0)  <= Duty(0) and Delay_In;
 Delay(1)  <= Duty(1) and Delay_In;
 Delay(2)  <= Duty(2) and Delay_In;
 Delay(3)  <= Duty(3) and Delay_In;
 Delay(4)  <= Duty(4) and Delay_In;
 Delay(5)  <= Duty(5) and Delay_In;
 Delay(6)  <= Duty(6) and Delay_In;
 Delay(7)  <= Duty(7) and Delay_In;
 Delay(8)  <= Duty(8) and Delay_In;
 Delay(9)  <= Duty(9) and Delay_In;
 Altenator1 : Alternator port map(nReset, Delay_In, S1, S2);
end architecture a1;
```

E.2.28 PWM2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity PWM2 is
 port(
  nReset    : in  std_logic;
  Clk       : in  std_logic; --97 656 250 Hz
  Duty      : in  std_logic_vector(17 downto 0);

  Delay_Out : out std_logic;
```

```
  Delay_In   : in  std_logic;
  Delay      : out std_logic_vector(9 downto 0);

  S1         : out std_logic;
  S2         : out std_logic
 );
end entity PWM2;

architecture a1 of PWM2 is
 constant Dead : std_logic_vector(7 downto 0) := "00001101"; --133ns (5%)
-- constant Dead : std_logic_vector(7 downto 0) := "00110011"; --522ns (20%)

 component Counter8 is
  port(
   nReset: in  std_logic;
   Clk   : in  std_logic;
   Q     : out std_logic_vector(7 downto 0)
  );
 end component Counter8;

 component DeadTime is
  port(
   nReset: in  std_logic;
   Clk   : in  std_logic;
   Count : in  std_logic_vector( 7 downto 0);
   PWM   : in  std_logic;
   Dead  : in  std_logic_vector( 7 downto 0);
   S1    : out std_logic;
   S2    : out std_logic
  );
 end component DeadTime;

 signal Count   : std_logic_vector( 7 downto 0);
 signal greater : std_logic;
begin
 Counter : Counter8 port map(nReset, Clk, Count);

 greater <= '1' when Duty(17 downto 10) > Count else '0';

 process(Clk) is
 begin
  if rising_edge(Clk) then
   Delay_Out <= greater;
  end if;
 end process;

 Delay(0)  <= Duty(0) and Delay_In;
 Delay(1)  <= Duty(1) and Delay_In;
 Delay(2)  <= Duty(2) and Delay_In;
 Delay(3)  <= Duty(3) and Delay_In;
 Delay(4)  <= Duty(4) and Delay_In;
 Delay(5)  <= Duty(5) and Delay_In;
 Delay(6)  <= Duty(6) and Delay_In;
 Delay(7)  <= Duty(7) and Delay_In;
 Delay(8)  <= Duty(8) and Delay_In;
 Delay(9)  <= Duty(9) and Delay_In;
 DeadTime1 : DeadTime port map(nReset, Clk, Count, Delay_In, Dead, S1, S2);
end architecture a1;
```

E.2.29 RealTime

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

Pectora roborant cultus recti

```
entity RealTime is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic; --9 765 625 Hz
  Q     : out std_logic_vector(7 downto 0)
 );
end entity RealTime;

architecture a1 of RealTime is
 signal count: std_logic_vector(23 downto 0);
 signal D    : std_logic_vector(23 downto 0);
 signal ands : std_logic_vector(23 downto 1);
 signal tQ   : std_logic_vector( 7 downto 0);
begin
 ands(          1) <= count(0);
 ands(23 downto 2) <= ands(22 downto 1) and count(22 downto 1);

 D(          0) <= not count(0);
 D(23 downto 1) <= count(23 downto 1) xor ands(23 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "000000000000000000000000";
   tQ    <= "00000000";
  elsif falling_edge(Clk) then
   if D = "100101010000001011111001" then
   count <= "000000000000000000000000";
   tQ <= tQ + '1';
   else
    count <= D;
   end if;
  end if;
 end process;

 Q <= tQ;
end architecture a1;
```

E.2.30 RS232Rx

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RS232Rx is
 port(
  nReset   : in  std_logic;
  Clk      : in  std_logic; --9 765 625 Hz
  Rx       : in  std_logic;
  Data     : out std_logic_vector(7 downto 0);
  Received : in  std_logic; --Must be low for received data to be latched
  SR       : out std_logic  --Pulsed high at end of reception
 );
end entity RS232Rx;

architecture a1 of RS232Rx is
 signal state    : std_logic_vector(1 downto 0);
 signal retstate : std_logic_vector(1 downto 0);
 signal count    : std_logic_vector(6 downto 0); --115200 baud
 signal tdata    : std_logic_vector(7 downto 0);
 signal count2   : std_logic_vector(2 downto 0);
begin
 process(Clk, nReset) is
```

```
 begin
  if nReset = '0' then
   state  <= "00";
   count  <= "0000000";
   count2 <= "111";
   Data   <= "00000000";
   tdata  <= "00000000";
   SR     <= '0';
  elsif rising_edge(Clk) then
   case state is
    when "00" =>
     SR <= '0';
     if Rx = '0' then
      count    <= "1111111"; --127
      retstate <= "10";
      state    <= "01";
     end if;
    when "01" =>
     if count = "0000000" then
      state <= retstate;
     end if;
     count <= count - '1';
    when "10" =>
     count2 <= count2 - '1';
     count  <= "1010100"; --84
     state  <= "01";
     if count2 = "000" then
      if Received = '0' then
       SR <= '1';
       Data(7) <= Rx;
       Data(6 downto 0) <= tdata(7 downto 1);
      end if;
      retstate <= "00";
     else
      tdata(7) <= Rx;
      tdata(6 downto 0) <= tdata(7 downto 1);
      retstate <= "10";
     end if;
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.2.31 RS232Tx

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RS232Tx is
 port(
  nReset   : in  std_logic;
  Clk      : in  std_logic; --9 765 625 Hz
  Tx       : out std_logic;
  Data     : in  std_logic_vector(7 downto 0);
  Send     : in  std_logic; --Pulsed high to start transmission
  CS       : out std_logic  --Pulsed high at end of transmission
 );
end entity RS232Tx;

architecture a1 of RS232Tx is
 signal state    : std_logic_vector(1 downto 0);
 signal retstate : std_logic_vector(1 downto 0);
```

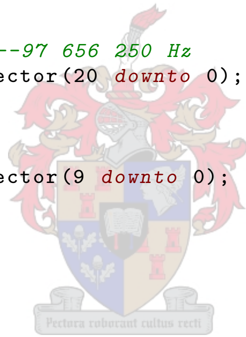
```
 signal count    : std_logic_vector(6 downto 0); --115200 baud
 signal tdata    : std_logic_vector(7 downto 0);
 signal count2   : std_logic_vector(2 downto 0);
begin
 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state  <= "00";
   count  <= "0000000";
   count2 <= "111";
   tdata  <= "00000000";
   Tx     <= '1';
   CS     <= '0';
  elsif rising_edge(Clk) then
   case state is
    when "00" =>
     if Send = '1' then
      Tx       <= '0';
      tData    <= Data;
      count    <= "1010100"; --84
      retstate <= "10";
      state    <= "01";
     end if;
    when "01" =>
     if count = "0000000" then
      state <= retstate;
     end if;
     count <= count - '1';
    when "10" =>
     Tx <= tdata(0);
     tdata(6 downto 0) <= tdata(7 downto 1);
     count2 <= count2 - '1';
     count  <= "1010100"; --84
     state  <= "01";
     if count2 = "000" then
      CS <= '1';
      retstate <= "11";
     else
      retstate <= "10";
     end if;
    when "11" =>
     Tx <= '1';
     CS <= '0';
     count    <= "1010100"; --84
     retstate <= "00";
     state    <= "01";
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.3 Controller

E.3.1 Controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Controller is
 port(
  Clk_125M    : in std_logic; -- 125 MHz
  nReset      : in std_logic;
  PW_OK       : in std_logic;

  LED_Error   : out std_logic;
  LED_Locked  : out std_logic;
  LED_F_High  : out std_logic;
  LED_F_Low   : out std_logic;
  LED_Reset   : out std_logic; --Reset done
  LED_Config  : out std_logic; --Config Done

  S1          : out std_logic;
  S2          : out std_logic;
  S3          : out std_logic;
  S4          : out std_logic;

  Delay0_D    : out std_logic_vector(9 downto 0);
  Delay0_In   : out std_logic;
  Delay0_Out  : in  std_logic;

  Delay1_D    : out std_logic_vector(9 downto 0);
  Delay1_In   : out std_logic;
  Delay1_Out  : in  std_logic;

  ADC_Reset   : out std_logic;
  ADC_SClk    : out std_logic;
  ADC_nCNVST  : out std_logic;
  ADC1_Busy   : in  std_logic;
  ADC1_SDOUT  : in  std_logic;
  ADC2_Busy   : in  std_logic;
  ADC2_SDOUT  : in  std_logic;
  ADC3_Busy   : in  std_logic;
  ADC3_SDOUT  : in  std_logic;
  ADC4_Busy   : in  std_logic;
  ADC4_SDOUT  : in  std_logic;

  DAC_Clk     : out std_logic;
  DAC_Update  : out std_logic;
  DAC_Data    : out std_logic;

  Keypad_X    : in  std_logic_vector( 4 downto 1);
  Keypad_Y    : out std_logic_vector( 4 downto 1);

  RS232_Tx  : out std_logic;
  RS232_Rx  : in  std_logic
 );
end entity Controller;

architecture a1 of Controller is
 component PLL is
  port(
   inclk0 : in  std_logic;
   c0     : out std_logic; --195 312 500.000 Hz
   locked : out std_logic
```

```
 );
end component PLL;

component Global is
 port(
  a_in : in  std_logic;
  a_out: out std_logic
 );
end component Global;

component Counter15 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(14 downto 0)
 );
end component Counter15;

component RealTime is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic; --9 765 625 Hz
  Q     : out std_logic_vector(7 downto 0)
 );
end component RealTime;

component PWM1 is
 port(
  nReset    : in  std_logic;
  Clk       : in  std_logic; --97 656 250 Hz
  Duty      : in  std_logic_vector(20 downto 0);

  Delay_Out : out std_logic;
  Delay_In  : in  std_logic;
  Delay     : out std_logic_vector(9 downto 0);

  S1        : out std_logic;
  S2        : out std_logic
 );
end component PWM1;

component PWM2 is
 port(
  nReset    : in  std_logic;
  Clk       : in  std_logic; --97 656 250 Hz
  Duty      : in  std_logic_vector(17 downto 0);
  Sample    : out std_logic; --ADC can sample when this high
  NS        : in  std_logic; --noise-shaper on when '1', delay-line when '0'

  Delay_Out : out std_logic;
  Delay_In  : in  std_logic;
  Delay     : out std_logic_vector(9 downto 0);

  S1        : out std_logic;
  S2        : out std_logic
 );
end component PWM2;

component ADC18 is
 port(
  nReset : in  std_logic;
  Clock  : in  std_logic;
  Sample : in  std_logic;
  Data1  : out std_logic_vector(17 downto 0);
  Data2  : out std_logic_vector(17 downto 0);
```



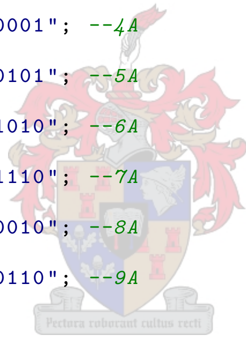
```
   Data3  : out std_logic_vector(17 downto 0);
   Data4  : out std_logic_vector(17 downto 0);
   Busy   : out std_logic;

   Reset  : out std_logic;
   SClk   : out std_logic;
   nCNVST : out std_logic;
   Busy1  : in  std_logic;
   SDOUT1 : in  std_logic;
   Busy2  : in  std_logic;
   SDOUT2 : in  std_logic;
   Busy3  : in  std_logic;
   SDOUT3 : in  std_logic;
   Busy4  : in  std_logic;
   SDOUT4 : in  std_logic
  );
 end component ADC18;

 component Control is
  port(
   nReset   : in std_logic;
   Clk      : in std_logic; --48 828 125 Hz, 20.48 ns
   dtClk    : in std_logic; --5 960 Hz, 167.772 16 us
   PW_OK    : in std_logic;
   ADC_Busy : in std_logic;

   Ref       : in  std_logic_vector(17 downto 0);
   Integrate : in std_logic;

   KV1  : in std_logic_vector(18 downto 0); --sign:abs(x)
   KI1  : in std_logic_vector(18 downto 0); --sign:abs(x)
   N1   : in std_logic_vector(17 downto 0);
   R1   : in std_logic_vector(17 downto 0);

   KV2   : in std_logic_vector(18 downto 0); --sign:abs(x)
   KI2   : in std_logic_vector(18 downto 0); --sign:abs(x)
   KI3   : in std_logic_vector(18 downto 0); --sign:abs(x)
   N2    : in std_logic_vector(17 downto 0);
   R2    : in std_logic_vector(17 downto 0);
   KInt  : in std_logic_vector(17 downto 0);

   Err    : out std_logic;
   High   : out std_logic;
   Low    : out std_logic;
   Locked : out std_logic;

   PWM0  : out std_logic_vector(20 downto 0);
   PWM1  : out std_logic_vector(17 downto 0);
   I3    : out std_logic_vector(17 downto 0);
   V3    : out std_logic_vector(17 downto 0);

   ADC0 : in  std_logic_vector(17 downto 0);
   ADC1 : in  std_logic_vector(17 downto 0);
   ADC2 : in  std_logic_vector(17 downto 0);
   ADC3 : in  std_logic_vector(17 downto 0)
  );
 end component Control;

 component Keypad_Driver is
  port(
   nReset : in  std_logic;
   Clk    : in  std_logic;

   Data   : out std_logic_vector(7 downto 0); --Data(7) = Empty;
                                              --Data(3 downto 0) = Key
```

```
   Ack     : in  std_logic;

   X       : in  std_logic_vector( 4 downto 1);
   Y       : out std_logic_vector( 4 downto 1)
  );
 end component Keypad_Driver;

 component Comm is
  port(
   nReset    : in std_logic;
   Clk       : in std_logic; --6 103 516 Hz

   Live      : in std_logic;

   ADC0      : in std_logic_vector(17 downto 0);
   ADC1      : in std_logic_vector(17 downto 0);
   ADC2      : in std_logic_vector(17 downto 0);
   ADC3      : in std_logic_vector(17 downto 0);
   ADC_Busy : in std_logic;

   Tx        : out std_logic;
   Rx        : in std_logic
  );
 end component Comm;

 component DAC is
  port(
   nReset : in  std_logic;
   Clock  : in  std_logic; -- Smaller than 25MHz

   DAC0   : in std_logic_vector(17 downto 0);
   DAC1   : in std_logic_vector(17 downto 0);
   Latch  : in std_logic;

   Clk    : out std_logic;
   Update : out std_logic;
   Data   : out std_logic
  );
 end component DAC;

--Signals
 signal tnReset           : std_logic;
 signal tClk_195_312_500 : std_logic;

 signal Clk_195_312_500 : std_logic;
 signal Clk_97_656_250  : std_logic;
 signal Clk_48_828_125  : std_logic;
 signal Clk_24_414_063  : std_logic;
 signal Clk_6_103_516   : std_logic;
 signal Clk_1_525_879   : std_logic;
 signal Clk_5_960       : std_logic;

 signal Clk_Temp   : std_logic_vector(15 downto 1);
 signal PLL_Locked : std_logic;

 signal tPWM0   : std_logic_vector(20 downto 0);
 signal tPWM1   : std_logic_vector(17 downto 0);
 signal tSample : std_logic;

 signal RealTimeQ   : std_logic_vector(7 downto 0);

 signal tADC0     : std_logic_vector(17 downto 0);
 signal tADC1     : std_logic_vector(17 downto 0);
 signal tADC2     : std_logic_vector(17 downto 0);
 signal tADC3     : std_logic_vector(17 downto 0);
```

```
 signal tADC_Busy : std_logic;

 signal Keypad_Data : std_logic_vector(7 downto 0);
 signal Keypad_Ack  : std_logic;

 signal Ref        : std_logic_vector(17 downto 0);

 signal KV1 : std_logic_vector(18 downto 0);
 signal KI1 : std_logic_vector(18 downto 0);
 signal N1  : std_logic_vector(17 downto 0);
 signal R1  : std_logic_vector(17 downto 0);

 signal KV2  : std_logic_vector(18 downto 0);
 signal KI2  : std_logic_vector(18 downto 0);
 signal KI3  : std_logic_vector(18 downto 0);
 signal N2   : std_logic_vector(17 downto 0);
 signal R2   : std_logic_vector(17 downto 0);
 signal KInt : std_logic_vector(17 downto 0); --Integral * dt

 signal tI3   : std_logic_vector(17 downto 0);
 signal tV3   : std_logic_vector(17 downto 0);
 signal DAC0  : std_logic_vector(17 downto 0);
 signal DAC1  : std_logic_vector(17 downto 0);
 signal DAC_L : std_logic;

--Temp
 signal state     : std_logic;
 signal Integrate : std_logic;
 signal LiveScope : std_logic;
 signal NS        : std_logic;
 signal Mux0      : std_logic_vector(2 downto 0);
 signal Mux1      : std_logic_vector(2 downto 0);
begin
 Global1 : Global port map(nReset      , tnReset      );
 PLL1    : PLL       port map(Clk_125M, tClk_195_312_500, PLL_Locked);
 Global2 : Global    port map(tClk_195_312_500, Clk_195_312_500);

 Count1  : Counter15 port map(tnReset, Clk_195_312_500, Clk_Temp);

 Global3 : Global port map(Clk_Temp( 1), Clk_97_656_250);
 Global4 : Global port map(Clk_Temp( 2), Clk_48_828_125);
 Global5 : Global port map(Clk_Temp( 3), Clk_24_414_063);
 Global6 : Global port map(Clk_Temp( 5), Clk_6_103_516 );
 Global7 : Global port map(Clk_Temp( 7), Clk_1_525_879 );
 Global8 : Global port map(Clk_Temp(15), Clk_5_960     );

 RealTime1 : RealTime port map(tnReset, Clk_48_828_125, RealTimeQ);

 PWM1_1 : PWM1 port map(tnReset, Clk_97_656_250, tPWM0,
                        Delay0_In, Delay0_Out, Delay0_D, S1, S2);
 PWM2_1 : PWM2 port map(tnReset, Clk_97_656_250, tPWM1, tSample, NS,
                        Delay1_In, Delay1_Out, Delay1_D, S3, S4);

 ADC1 : ADC18 port map(tnReset, Clk_48_828_125, tSample,
                      tADC0, tADC1, tADC2, tADC3, tADC_Busy,
                      ADC_Reset, ADC_SClk, ADC_nCNVST,
                      ADC1_Busy, ADC1_SDOUT,
                      ADC2_Busy, ADC2_SDOUT,
                      ADC3_Busy, ADC3_SDOUT,
                      ADC4_Busy, ADC4_SDOUT);

 Control1 : Control port map(tnReset, Clk_48_828_125, Clk_5_960,
                            PW_OK, tADC_Busy,
                            Ref, Integrate,
                            KV1, KI1, N1, R1,
```

```
                             KV2, KI2, KI3, N2, R2, KInt,
                             LED_Error, LED_F_High, LED_F_Low, LED_Locked,
                             tPWM0, tPWM1, tI3, tV3,
                             tADC0, tADC1, tADC2, tADC3);

 Keypad1 : Keypad_Driver port map(tnReset, Clk_5_960,
                                  Keypad_Data, Keypad_Ack,
                                  Keypad_X, Keypad_Y);

 Comm1 : Comm port map(tnReset, Clk_6_103_516, LiveScope,
                       tADC0, tADC1, tADC2, tADC3, tADC_Busy,
                       RS232_Tx, RS232_Rx);

 DAC2 : DAC port map(tnReset, Clk_24_414_063,
                     DAC0, DAC1, DAC_L,
                     DAC_Clk, DAC_Update, DAC_Data);

 LED_Reset  <= tnReset;
 LED_Config <= PLL_Locked;

 with Mux0 select DAC0 <= tPWM0(20 downto 3)   when "000",
                          tPWM1                when "001",
                          tADC0                when "010",
                          tADC1                when "011",
                          tADC2                when "100",
                          tADC3                when "101",
                          tI3                  when "110",
                          tV3                  when "111",
                          "000000000000000000" when others;

 with Mux1 select DAC1 <= tPWM0(20 downto 3)   when "000",
                          tPWM1                when "001",
                          tADC0                when "010",
                          tADC1                when "011",
                          tADC2                when "100",
                          tADC3                when "101",
                          tI3                  when "110",
                          tV3                  when "111",
                          "000000000000000000" when others;

 DAC_L <= not tADC_Busy;

-- KV1  <= "101101011010101000 1"; --Full = 3.413, sign:abs(x)
-- KI1  <= "100000011110110000 0"; --Full = 68.27, sign:abs(x)
-- N1   <= "011011110001111101";  --Full = 3.413
-- R1   <= "000000000010010011";  --Full = 1.788 MV/s

-- KV2   <= "011111101100100011 0"; --Full = 1 , sign:abs(x)
-- KI2   <= "100010101101000111 0"; --Full = 12, sign:abs(x)
-- KI3   <= "100000010100010000 1"; --Full = 12, sign:abs(x)
-- N2    <= "000111110101101100";  --Full = 12
-- R2    <= "000000000000001011";  --Full = 119.2 kA/s
-- KInt  <= "000000000111001101";  --Full = 2980 (0.5/dt); dt=10e-12 x 2^24

 KV1  <= "1001111101110011001"; --Full = 3.413, sign:abs(x)
 KI1  <= "1000000001011100001"; --Full = 68.27, sign:abs(x)
 N1   <= "001111100110011001";  --Full = 3.413
 R1   <= "000000000010010011";  --Full = 1.788 MV/s

 KV2   <= "0111111100000110000"; --Full = 1 , sign:abs(x)
 KI2   <= "1000011011000011111"; --Full = 12, sign:abs(x)
 KI3   <= "1000000011110001011"; --Full = 12, sign:abs(x)
 N2    <= "000101100000011011";  --Full = 12
 R2    <= "000000000001000010";  --Full = 119.2 kA/s
 KInt  <= "000000001000010000";  --Full = 2980 (0.5/dt); dt=10e-12 x 2^24
```

```
 process(Clk_1_525_879, tnReset) is
 begin
  if tnReset = '0' then
   state       <= '0';
   Integrate   <= '1';
   NS          <= '0';
   LiveScope   <= '1';
   Keypad_Ack  <= '0';
   Ref  <= "000100010001000100"; --1A
   Mux0  <= "100";
   Mux1  <= "101";
  elsif rising_edge(Clk_1_525_879) then
   case state is
    when '0' =>
     if Keypad_Data(7) = '0' then
      case Keypad_Data(3 downto 0) is
       when "0000" => --0
        Ref  <= "000000000000000000"; --0.00A
        Mux0  <= "000";
        Mux1  <= "000";
       when "0001" => --1
        Ref  <= "000100010001000100"; --1A
       when "0010" => --2
        Ref  <= "001000100010001001"; --2A
       when "0011" => --3
        Ref  <= "001100110011001101"; --3A
       when "0100" => --4
        Ref  <= "010001000100010001"; --4A
       when "0101" => --5
        Ref  <= "010101010101010101"; --5A
       when "0110" => --6
        Ref  <= "011001100110011010"; --6A
       when "0111" => --7
        Ref  <= "011101110111011110"; --7A
       when "1000" => --8
        Ref  <= "100010001000100010"; --8A
       when "1001" => --9
        Ref  <= "100110011001100110"; --9A
       when "1010" => --A
        NS  <= '1';
       when "1011" => --B
        NS  <= '0';
       when "1100" => --C
        LiveScope  <= '1';
       when "1101" => --D
        LiveScope  <= '0';
       when "1110" => --*
        Mux0  <= Mux0 + '1';
       when "1111" => --#
        Mux1  <= Mux1 + '1';
       when others =>
      end case;
      Keypad_Ack  <= '1';
      state  <= '1';
     end if;
    when '1' =>
     Keypad_Ack  <= '0';
     state  <= '0';
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```



E.3.2 ADC18

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ADC18 is
 port(
  nReset : in  std_logic;
  Clock  : in  std_logic;
  Sample : in  std_logic;
  Data1  : out std_logic_vector(17 downto 0);
  Data2  : out std_logic_vector(17 downto 0);
  Data3  : out std_logic_vector(17 downto 0);
  Data4  : out std_logic_vector(17 downto 0);
  Busy   : out std_logic;

  Reset  : out std_logic;
  SClk   : out std_logic;
  nCNVST : out std_logic;
  Busy1  : in  std_logic;
  SDOUT1 : in  std_logic;
  Busy2  : in  std_logic;
  SDOUT2 : in  std_logic;
  Busy3  : in  std_logic;
  SDOUT3 : in  std_logic;
  Busy4  : in  std_logic;
  SDOUT4 : in  std_logic
 );
end entity ADC18;

architecture a1 of ADC18 is
 signal state : std_logic_vector( 2 downto 0);
 signal count : std_logic_vector( 4 downto 0);
 signal tBusy : std_logic;
 signal Temp1 : std_logic_vector(17 downto 0);
 signal Temp2 : std_logic_vector(17 downto 0);
 signal Temp3 : std_logic_vector(17 downto 0);
 signal Temp4 : std_logic_vector(17 downto 0);
 signal Cp1   : std_logic_vector( 4 downto 0);
begin
 Reset <= not nReset;
 tBusy <= Busy1 or Busy2 or Busy3 or Busy4;
 Cp1 <= Count + '1';

 process(Clock, nReset) is
 begin
  if nReset = '0' then
   state  <= "000";
   count  <= "00000";
   SClk   <= '0';
   nCNVST <= '1';
   Temp1  <= "000000000000000000";
   Temp2  <= "000000000000000000";
   Temp3  <= "000000000000000000";
   Temp4  <= "000000000000000000";
   Busy   <= '1';
  elsif rising_edge(Clock) then
   case state is
    when "000" =>
     if count = "11111" then
      state <= "001";
     end if;
     count <= Cp1;
    when "001" =>
```

```
      count <= "00000";
      if tBusy = '0' then
       state <= "010";
      end if;
     when "010" =>
      SClk <= '1';
      Temp1(17 downto 1) <= Temp1(16 downto 0);
      Temp2(17 downto 1) <= Temp2(16 downto 0);
      Temp3(17 downto 1) <= Temp3(16 downto 0);
      Temp4(17 downto 1) <= Temp4(16 downto 0);
      Count <= Cp1;
      state <= "011";
     when "011" =>
      SClk <= '0';
      Temp1(0) <= SDOUT1;
      Temp2(0) <= SDOUT2;
      Temp3(0) <= SDOUT3;
      Temp4(0) <= SDOUT4;
      if count = "10010" then --18
       state <= "100";
      else
       state <= "010";
      end if;
     when "100" =>
      Data1 <= Temp1;
      Data2 <= Temp2;
      Data3 <= Temp3;
      Data4 <= Temp4;
      state <= "101";
     when "101" =>
      Busy <= '0';
      if Sample = '0' then
       state <= "110";
      end if;
     when "110" =>
      if Sample = '1' then
       Busy   <= '1';
       nCNVST <= '0';
       state  <= "111";
      end if;
     when "111" =>
      nCNVST <= '1';
      state <= "001";
     when others =>
    end case;
   end if;
  end process;
end architecture a1;
```

E.3.3 Averager

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Averager is
 port(
  nReset   : in std_logic;
  Clk      : in std_logic; --6 103 516 Hz
  ADC_Busy : in std_logic;

  Input  : in std_logic_vector(17 downto 0);

  Output : out std_logic_vector(23 downto 0)
```

```
 );
end entity Averager;

architecture a1 of Averager is
 component RAM_512x18 is
  port(
   clock     : in std_logic;
   data      : in std_logic_vector(17 downto 0);
   rdaddress : in std_logic_vector( 8 downto 0);
   wraddress : in std_logic_vector( 8 downto 0);
   wren      : in std_logic := '1';
   q         : out std_logic_vector(17 downto 0)
  );
 end component RAM_512x18;

 signal Data      : std_logic_vector(17 downto 0);
 signal Address   : std_logic_vector( 8 downto 0);
 signal AddressP1 : std_logic_vector( 8 downto 0);
 signal RAM       : std_logic_vector(17 downto 0);
 signal at        : std_logic_vector(26 downto 0);
 signal state     : std_logic_vector( 1 downto 0);
begin
 AddressP1 <= Address + '1';
 FIFO : RAM_512x18 port map(not Clk, Data, AddressP1, Address, '1', RAM);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   Data    <= "000000000000000000";
   Address <= "000000000";
   at      <= "000000000000000000000000000";
   state   <= "00";
  elsif rising_edge(Clk) then
   case state is
    when "00" =>
     Address <= AddressP1;
     if Address = "111111111" then
      state <= "01";
     end if;
    when "01" =>
     if ADC_Busy = '0' then
      state <= "10";
     end if;
    when "10" =>
     if ADC_Busy = '1' then
      Data    <= Input;
      Address <= AddressP1;
      at      <= at - RAM + Input;
      state <= "01";
     end if;
    when others =>
   end case;
  end if;
 end process;

 Output <= at(26 downto 3);
end architecture a1;
```

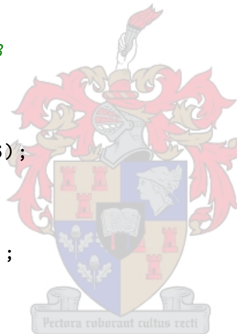
E.3.4 Comm

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
--/**/once the tests are done,
--    Implement the sender and receiver and put the interfaced RS232 in here
--Remember to add the 512 byte FIFO receive buffer

entity Comm is
 port(
  nReset   : in std_logic;
  Clk      : in std_logic; --6 103 516 Hz

  Live     : in std_logic;

  ADC0     : in std_logic_vector(17 downto 0);
  ADC1     : in std_logic_vector(17 downto 0);
  ADC2     : in std_logic_vector(17 downto 0);
  ADC3     : in std_logic_vector(17 downto 0);
  ADC_Busy : in std_logic;

  Tx       : out std_logic;
  Rx       : in std_logic
 );
end entity Comm;

architecture a1 of Comm is
 component Averager is
  port(
   nReset   : in std_logic;
   Clk      : in std_logic; --6 103 516 Hz
   ADC_Busy : in std_logic;

   Input  : in std_logic_vector(17 downto 0);

   Output : out std_logic_vector(23 downto 0)
  );
 end component Averager;

 component RS232Tx is
  port(
   nReset   : in  std_logic;
   Clk      : in  std_logic; --6 103 516 Hz
   Tx       : out std_logic;
   Data     : in  std_logic_vector(7 downto 0);
   Send     : in  std_logic; --Pulsed high to start transmission
   CS       : out std_logic  --Pulsed high at end of transmission
  );
 end component RS232Tx;

 component RAM_1024x18 is
  port(
   clock     : in  std_logic;
   data      : in  std_logic_vector(17 downto 0);
   rdaddress : in  std_logic_vector( 9 downto 0);
   wraddress : in  std_logic_vector( 9 downto 0);
   wren      : in  std_logic := '1';
   q         : out std_logic_vector(17 downto 0)
  );
 end component RAM_1024x18;

 signal Buf_Clk       : std_logic;
 signal Buf_Data0     : std_logic_vector(17 downto 0);
 signal Buf_Data1     : std_logic_vector(17 downto 0);
 signal Buf_Data2     : std_logic_vector(17 downto 0);
 signal Buf_Data3     : std_logic_vector(17 downto 0);
 signal Buf_rdAddress : std_logic_vector( 9 downto 0);
 signal Buf_wrAddress : std_logic_vector( 9 downto 0);
 signal Buf_Out0      : std_logic_vector(17 downto 0);
```

```
 signal Buf_Out1      : std_logic_vector(17 downto 0);
 signal Buf_Out2      : std_logic_vector(17 downto 0);
 signal Buf_Out3      : std_logic_vector(17 downto 0);

 signal Data : std_logic_vector(7 downto 0);
 signal Send : std_logic;
 signal CS   : std_logic;

 signal count : std_logic_vector(12 downto 0);
 signal clear : std_logic;

 signal tADC0 : std_logic_vector(23 downto 0);
 signal tADC1 : std_logic_vector(23 downto 0);
 signal tADC2 : std_logic_vector(23 downto 0);
 signal tADC3 : std_logic_vector(23 downto 0);

 signal lADC0 : std_logic_vector(23 downto 0);
 signal lADC1 : std_logic_vector(23 downto 0);
 signal lADC2 : std_logic_vector(23 downto 0);
 signal lADC3 : std_logic_vector(23 downto 0);

 signal state    : std_logic_vector(5 downto 0);
 signal retstate : std_logic_vector(5 downto 0);
begin
 Tx1 : RS232Tx port map(nReset, Clk, Tx, Data, Send, CS);

 Avg1 : Averager port map(nReset, Clk, ADC_Busy, ADC0, tADC0);
 Avg2 : Averager port map(nReset, Clk, ADC_Busy, ADC1, tADC1);
 Avg3 : Averager port map(nReset, Clk, ADC_Busy, ADC2, tADC2);
 Avg4 : Averager port map(nReset, Clk, ADC_Busy, ADC3, tADC3);

 Buf0 : RAM_1024x18 port map(Buf_Clk,
                             Buf_Data0,
                             Buf_rdAddress, Buf_wrAddress, '1',
                             Buf_Out0);
 Buf1 : RAM_1024x18 port map(Buf_Clk,
                             Buf_Data1,
                             Buf_rdAddress, Buf_wrAddress, '1',
                             Buf_Out1);
 Buf2 : RAM_1024x18 port map(Buf_Clk,
                             Buf_Data2,
                             Buf_rdAddress, Buf_wrAddress, '1',
                             Buf_Out2);
 Buf3 : RAM_1024x18 port map(Buf_Clk,
                             Buf_Data3,
                             Buf_rdAddress, Buf_wrAddress, '1',
                             Buf_Out3);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   Data     <= "00000000";
   Send     <= '1';
   count    <= "0000000000000";
   clear    <= '0';
   lADC0    <= "000000000000000000000000";
   lADC1    <= "000000000000000000000000";
   lADC2    <= "000000000000000000000000";
   lADC3    <= "000000000000000000000000";

   Buf_Clk       <= '0';
   Buf_Data0     <= "000000000000000000";
   Buf_Data1     <= "000000000000000000";
   Buf_Data2     <= "000000000000000000";
   Buf_Data3     <= "000000000000000000";
```

```
   Buf_rdAddress <= "0000000000";
   Buf_wrAddress <= "0000000000";

   state     <= "000000";
   retstate <= "000000";
  elsif falling_edge(Clk) then
   if clear = '1' then
    count <= "0000000000000";
   else
    count <= count + '1';
   end if;
  elsif rising_edge(Clk) then
   case state is
--Live
    when "000000" =>
     if count = "1110111001101" then --7629 => 800 Hz
      clear <= '1';
      lADC0 <= tADC0;
      lADC1 <= tADC1;
      lADC2 <= tADC2;
      lADC3 <= tADC3;
      if Live = '1' then
       state <= "000001";
      else
       state <= "010000";
      end if;
     end if;
    when "000001" =>
     clear <= '0';
     Data <= "10101011"; --0xAB
     retstate <= "000010";
     state <= "001110";
    when "000010" =>
     Data <= lADC0(23 downto 16);
     retstate <= "000011";
     state <= "001110";
    when "000011" =>
     Data <= lADC0(15 downto 8);
     retstate <= "000100";
     state <= "001110";
    when "000100" =>
     Data <= lADC0(7 downto 0);
     retstate <= "000101";
     state <= "001110";
    when "000101" =>
     Data <= lADC1(23 downto 16);
     retstate <= "000110";
     state <= "001110";
    when "000110" =>
     Data <= lADC1(15 downto 8);
     retstate <= "000111";
     state <= "001110";
    when "000111" =>
     Data <= lADC1(7 downto 0);
     retstate <= "001000";
     state <= "001110";
    when "001000" =>
     Data <= lADC2(23 downto 16);
     retstate <= "001001";
     state <= "001110";
    when "001001" =>
     Data <= lADC2(15 downto 8);
     retstate <= "001010";
     state <= "001110";
    when "001010" =>
```



```
     Data <= lADC2(7 downto 0);
     retstate <= "001011";
     state <= "001110";
    when "001011" =>
     Data <= lADC3(23 downto 16);
     retstate <= "001100";
     state <= "001110";
    when "001100" =>
     Data <= lADC3(15 downto 8);
     retstate <= "001101";
     state <= "001110";
    when "001101" =>
     Data <= lADC3(7 downto 0);
     retstate <= "000000";
     state <= "001110";
--Send
    when "001110" =>
     if CS = '0' then
      Send <= '1';
      state <= "001111";
     end if;
    when "001111" =>
     if CS = '1' then
      Send <= '0';
      state <= retstate;
     end if;
--Buffer
    when "010000" =>
     if ADC_Busy = '1' then
      state <= "100010";
     end if;
    when "100010" =>
     if ADC_Busy = '0' then
      Buf_Data0 <= ADC0;
      Buf_Data1 <= ADC1;
      Buf_Data2 <= ADC2;
      Buf_Data3 <= ADC3;
      if Live = '0' then
       state <= "010001";
      else
       clear <= '0';
       state <= "000000";
      end if;
     end if;
    when "010001" =>
     Buf_rdAddress(1 downto 0) <= Buf_rdAddress(1 downto 0) + '1';
     Buf_Clk <= '1';
     state <= "010010";
    when "010010" =>
     Buf_Clk <= '0';
     if Buf_wrAddress = "1111111111" then
      Buf_rdAddress <= "0000000000";
      state <= "010011";
     else
      Buf_rdAddress(1 downto 0) <= Buf_rdAddress(1 downto 0) + '1';
      Buf_wrAddress <= Buf_wrAddress + '1';
      state <= "010000";
     end if;
    when "010011" =>
     Data <= "10101011"; --0xAB
     retstate <= "010100";
     state <= "001110";
    when "010100" =>
     Data <= "11111111"; --0xFF
     state <= "001110";
```

```
     if Buf_rdAddress(3 downto 0) = "1011" then
      retstate <= "010101";
      Buf_rdAddress(3 downto 0) <= "0000";
     else
      retstate <= "010100";
      Buf_rdAddress(3 downto 0) <= Buf_rdAddress(3 downto 0) + '1';
     end if;
    when "010101" =>
     Buf_Clk <= '1';
     Data <= "10101011"; --0xAB
     retstate <= "010110";
     state <= "001110";
    when "010110" =>
     Buf_Clk <= '0';
     Data <= Buf_Out0(17 downto 10);
     retstate <= "010111";
     state <= "001110";
    when "010111" =>
     Data <= Buf_Out0(9 downto 2);
     retstate <= "011000";
     state <= "001110";
    when "011000" =>
     Data(7 downto 6) <= Buf_Out0(1 downto 0);
     Data(5 downto 0) <= "000000";
     retstate <= "011001";
     state <= "001110";
    when "011001" =>
     Data <= Buf_Out1(17 downto 10);
     retstate <= "011010";
     state <= "001110";
    when "011010" =>
     Data <= Buf_Out1(9 downto 2);
     retstate <= "011011";
     state <= "001110";
    when "011011" =>
     Data(7 downto 6) <= Buf_Out1(1 downto 0);
     Data(5 downto 0) <= "000000";
     retstate <= "011100";
     state <= "001110";
    when "011100" =>
     Data <= Buf_Out2(17 downto 10);
     retstate <= "011101";
     state <= "001110";
    when "011101" =>
     Data <= Buf_Out2(9 downto 2);
     retstate <= "011110";
     state <= "001110";
    when "011110" =>
     Data(7 downto 6) <= Buf_Out2(1 downto 0);
     Data(5 downto 0) <= "000000";
     retstate <= "011111";
     state <= "001110";
    when "011111" =>
     Data <= Buf_Out3(17 downto 10);
     retstate <= "100000";
     state <= "001110";
    when "100000" =>
     Data <= Buf_Out3(9 downto 2);
     retstate <= "100001";
     state <= "001110";
    when "100001" =>
     Data(7 downto 6) <= Buf_Out3(1 downto 0);
     Data(5 downto 0) <= "000000";
     state <= "001110";
     if Buf_rdAddress = "1111111111" then
```

```
      Buf_wrAddress <= "0000000000";
      retstate <= "010000";
     else
      retstate <= "010101";
     end if;
     Buf_rdAddress <= Buf_rdAddress + '1';
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.3.5 Comp_Avg

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Comp_Avg is
 port(
  nReset : in std_logic;
  Clk    : in std_logic;

  Hi : in std_logic;
  Lo : in std_logic;

  Locked : out std_logic;
  High   : out std_logic;
  Low    : out std_logic
 );
end entity Comp_Avg;

architecture a1 of Comp_Avg is
 signal L     : std_logic_vector(31 downto 0);
 signal Lt    : std_logic_vector( 5 downto 0);
 signal H     : std_logic_vector(31 downto 0);
 signal Ht    : std_logic_vector( 5 downto 0);

 signal state : std_logic_vector(1 downto 0);
begin
 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state <= "00";
   L     <= "00000000000000000000000000000000";
   Lt    <= "000000";
   H     <= "00000000000000000000000000000000";
   Ht    <= "000000";
  elsif falling_edge(Clk) then
   L(0) <= Lo;
   L(31 downto 1) <= L(30 downto 0);
   H(0) <= Hi;
   H(31 downto 1) <= H(30 downto 0);

   Lt <= Lt - L(31) + Lo;
   Ht <= Ht - H(31) + Hi;

   case state is
    when "00" => --Locked
     if Ht(5) = '1' then --P(High) = 100%
      state <= "01";
     elsif Lt(5) = '1' then --P(Low) = 100%
      state <= "10";
     end if;
```

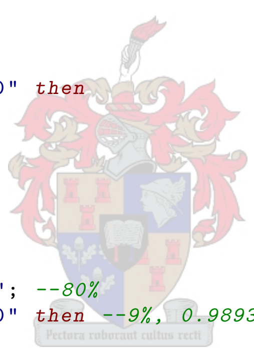
```
    when "01" => --High
     if Ht < "010000" then --P(High) < 50%
      state <= "00";
     end if;
    when "10" => --Low
     if Lt < "010000" then --P(Low) < 50%
      state <= "00";
     end if;
    when others =>
   end case;
  end if;
 end process;

 Locked <= state(0) nor state(1);
 High   <= state(0);
 Low    <= state(1);
end architecture a1;
```

E.3.6 Control

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Control is
 port(
  nReset   : in std_logic;
  Clk      : in std_logic; --48 828 125 Hz, 20.48 ns
  dtClk    : in std_logic; --5 960 Hz, 167.772 16 us
  PW_OK    : in std_logic;
  ADC_Busy : in std_logic;

  Ref        : in std_logic_vector(17 downto 0);
  Integrate : in std_logic;

  KV1 : in std_logic_vector(18 downto 0); --sign:abs(x)
  KI1 : in std_logic_vector(18 downto 0); --sign:abs(x)
  N1  : in std_logic_vector(17 downto 0);
  R1  : in std_logic_vector(17 downto 0);

  KV2  : in std_logic_vector(18 downto 0); --sign:abs(x)
  KI2  : in std_logic_vector(18 downto 0); --sign:abs(x)
  KI3  : in std_logic_vector(18 downto 0); --sign:abs(x)
  N2   : in std_logic_vector(17 downto 0);
  R2   : in std_logic_vector(17 downto 0);
  KInt : in std_logic_vector(17 downto 0);

  Err        : out std_logic;
  High       : out std_logic;
  Low        : out std_logic;
  Locked     : out std_logic;

--/**/Probe lines to be put here

  PWM0 : out std_logic_vector(20 downto 0);
  PWM1 : out std_logic_vector(17 downto 0);
  I3   : out std_logic_vector(17 downto 0);
  V3   : out std_logic_vector(17 downto 0);

--/**/DAC to be put here

  ADC0 : in  std_logic_vector(17 downto 0);
  ADC1 : in  std_logic_vector(17 downto 0);
  ADC2 : in  std_logic_vector(17 downto 0);
```

```
  ADC3 : in  std_logic_vector(17 downto 0)
 );
end entity Control;

architecture a1 of Control is
 component Stage1 is
  port(
   Enable : in std_logic;
   dtClk  : in std_logic; --5 960 Hz, 167.772 16 us

   Ref : in std_logic_vector(17 downto 0);
   V1  : in std_logic_vector(17 downto 0);
   I1  : in std_logic_vector(17 downto 0);

   KV1 : in std_logic_vector(18 downto 0); --sign:abs(x)
   KI1 : in std_logic_vector(18 downto 0); --sign:abs(x)
   N1  : in std_logic_vector(17 downto 0);
   R1  : in std_logic_vector(17 downto 0);

   D1 : out std_logic_vector(20 downto 0)
  );
 end component Stage1;

 component Stage2 is
  port(
   nReset   : in std_logic;
   Enable   : in std_logic;
   Clk      : in std_logic; --48 828 125 Hz, 20.48 ns
   dtClk    : in std_logic; --5 960 Hz, 167.772 16 us
   ADC_Busy : in std_logic;

   Ref : in std_logic_vector(17 downto 0);
   Int : in std_logic;

   KV2  : in std_logic_vector(18 downto 0); --sign:abs(x)
   KI2  : in std_logic_vector(18 downto 0); --sign:abs(x)
   KI3  : in std_logic_vector(18 downto 0); --sign:abs(x)
   N2   : in std_logic_vector(17 downto 0);
   R2   : in std_logic_vector(17 downto 0);
   KInt : in std_logic_vector(17 downto 0);

   V1  : in std_logic_vector(17 downto 0);
   V2  : in std_logic_vector(17 downto 0);
   I2  : in std_logic_vector(17 downto 0);

   Locked : out std_logic;
   High   : out std_logic;
   Low    : out std_logic;

   V3 : out std_logic_vector(17 downto 0);
   I3 : out std_logic_vector(17 downto 0);
   D2 : out std_logic_vector(17 downto 0)
  );
 end component Stage2;
--Soft-start
 signal count  : std_logic_vector(13 downto 0);
 signal state  : std_logic_vector( 1 downto 0);
 signal enable : std_logic;
 signal x1     : std_logic_vector(13 downto 0); --soft-start PWM0
 signal x2     : std_logic_vector(17 downto 0); --soft-start PWM1
--Main timing
 signal Mstate : std_logic_vector(1 downto 0);
--Stage1
 signal V1  : std_logic_vector(17 downto 0);
 signal I1  : std_logic_vector(17 downto 0);
```

```
 signal D1  : std_logic_vector(20 downto 0);
--Stage2
 signal V2   : std_logic_vector(17 downto 0);
 signal I2   : std_logic_vector(17 downto 0);
 signal tV3  : std_logic_vector(17 downto 0);
 signal tI3  : std_logic_vector(17 downto 0);
 signal D2   : std_logic_vector(17 downto 0);
 signal Lock : std_logic;
 signal Hi   : std_logic;
 signal Lo   : std_logic;
begin
 process(dtClk, nReset) is
 begin
  if nReset = '0' then
   count  <= "11111111111111"; --2.749 sek
   enable <= '0';
   x1     <= "00000000000000";
   x2     <= "000000000000000000";
   Err    <= '0';
   Locked <= '0';
   High   <= '0';
   Low    <= '0';
   state  <= "00";
  elsif falling_edge(dtClk) then
   case state is
    when "00" =>
     if PW_OK = '1' then
      state <= "01";
     end if;
    when "01" =>
     if count = "00000000000000" then
      state <= "10";
     else
      count <= count - '1';
     end if;
    when "10" =>
     x1 <= count;
     count <= count + '1';
     x2 <= "110011001100110011"; --80%
     if count = "01011100001010" then --9%, 0.9893 sek
      enable <= '1';
      state  <= "11";
     end if;
    when "11" =>
     x1 <= "00000000000000";
     x2 <= "000000000000000000";
     if PW_OK = '0' then
      enable <= '0';
      Err    <= '1';
      Locked <= '0';
      High   <= '0'; --Error 1
      Low    <= '1';
     elsif I1 > "111101011100001010" then --14.4A
      enable <= '0';
      Err    <= '1';
      Locked <= '0';
      High   <= '1'; --Error 2
      Low    <= '0';
     elsif V1 > "111110111011101111" then --295V
      enable <= '0';
      Err    <= '1';
      Locked <= '0';
      High   <= '1'; --Error 3
      Low    <= '1';
     elsif I2 > "111001100110011010" then --18A
```



```
     enable <= '0';
     Err    <= '1';
     Locked <= '1';
     High   <= '0'; --Error 4
     Low    <= '0';
    elsif V2 > "111110101010101011" then --235V
     enable <= '0';
     Err    <= '1';
     Locked <= '1';
     High   <= '0'; --Error 5
     Low    <= '1';
    elsif V1 < "000010001000100010" then --10V
     enable <= '0';
     Err    <= '1';
     Locked <= '1';
     High   <= '1'; --Error 6
     Low    <= '0';
    elsif I2 < "000001100110011010" then --500mA
     enable <= '0';
     Err    <= '1';
     Locked <= '1';
     High   <= '1'; --Error 7
     Low    <= '1';
    elsif enable = '1' then
     Locked <= Lock;
     High   <= Hi;
     Low    <= Lo;
    end if;
   when others =>
  end case;
 end if;
end process;

process(Clk, nReset) is
begin
 if nReset = '0' then
  Mstate <= "00";
 elsif falling_edge(Clk) then
  case Mstate is
   when "00" =>
    if enable = '0' then
     PWM0(20 downto 19) <= "00";
     PWM0(18 downto  5) <= x1;
     PWM0( 4 downto  0) <= "00000";
     PWM1               <= x2;
    else
     Mstate <= "01";
    end if;
   when "01" =>
    if ADC_Busy = '1' then
     Mstate <= "10";
    end if;
   when "10" =>
    if ADC_Busy = '0' then
     V1 <= ADC1;
     I1 <= ADC0;
     V2 <= ADC3;
     I2 <= ADC2;
     Mstate <= "11";
    end if;
   when "11" =>
    if enable = '0' then
     Mstate <= "00";
    else
     PWM0 <= D1;
```

```
--      PWM0 <= "000101110000101001000"; --9%
--      PWM1 <= D2;
      PWM1 <= "110011001100110011"; --80%
      if ADC_Busy = '1' then
       Mstate <= "10";
      end if;
     end if;
    when others =>
   end case;
  end if;
 end process;

 Stage1_1 : Stage1 port map(enable, dtClk,
                           tV3, V1, I1,
                           KV1, KI1, N1, R1,
                           D1);

 Stage2_1 : Stage2 port map(nReset, enable, Clk, dtClk, ADC_Busy,
                           Ref, Integrate,
                           KV2, KI2, KI3, N2, R2, KInt,
                           V1, V2, I2,
                           Lock, Hi, Lo,
                           tV3, tI3, D2);

 V3 <= tV3;
 I3 <= tI3;
end architecture a1;
```

E.3.7 Counter11

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter11 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(10 downto 0)
 );
end entity Counter11;

architecture a1 of Counter11 is
 signal count: std_logic_vector(10 downto 0);
 signal D    : std_logic_vector(10 downto 0);
 signal ands : std_logic_vector(10 downto 1);
begin
 ands(          1) <= count(0);
 ands(10 downto 2) <= ands(9 downto 1) and count(9 downto 1);

 D(          0) <= not count(0);
 D(10 downto 1) <= count(10 downto 1) xor ands(10 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "00000000000";
  elsif falling_edge(Clk) then
   count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

E.3.8 Counter15

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter15 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(14 downto 0)
 );
end entity Counter15;

architecture a1 of Counter15 is
 signal count: std_logic_vector(14 downto 0);
 signal D    : std_logic_vector(14 downto 0);
 signal ands : std_logic_vector(14 downto 1);
begin
 ands(          1) <= count(0);
 ands(14 downto 2) <= ands(13 downto 1) and count(13 downto 1);

 D(          0) <= not count(0);
 D(14 downto 1) <= count(14 downto 1) xor ands(14 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "000000000000000";
  elsif falling_edge(Clk) then
   count <= D;
  end if;
 end process;

 Q <= count;
end architecture a1;
```

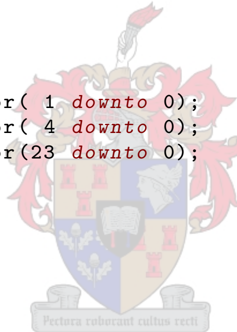
E.3.9 Counter8

```
library ieee;
use ieee.std_logic_1164.all;

entity Counter8 is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Q     : out std_logic_vector(7 downto 0)
 );
end entity Counter8;

architecture a1 of Counter8 is
 signal count: std_logic_vector(7 downto 0);
 signal D    : std_logic_vector(7 downto 0);
 signal ands : std_logic_vector(7 downto 1);
begin
 ands(          1) <= count(0);
 ands(7 downto 2) <= ands(6 downto 1) and count(6 downto 1);

 D(          0) <= not count(0);
 D(7 downto 1) <= count(7 downto 1) xor ands(7 downto 1);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   count <= "00000000";
```

```
   elsif falling_edge(Clk) then
    count <= D;
   end if;
  end process;

  Q <= count;
end architecture a1;
```

E.3.10 DAC

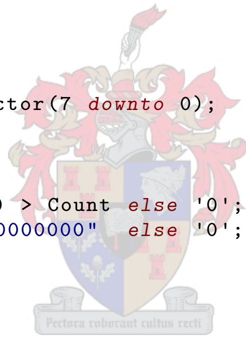
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity DAC is
 port(
  nReset : in  std_logic;
  Clock  : in  std_logic; -- Smaller than 25MHz

  DAC0   : in std_logic_vector(17 downto 0);
  DAC1   : in std_logic_vector(17 downto 0);
  Latch  : in std_logic;

  Clk    : out std_logic;
  Update : out std_logic;
  Data   : out std_logic
 );
end entity DAC;

architecture a1 of DAC is
 signal state : std_logic_vector( 1 downto 0);
 signal count : std_logic_vector( 4 downto 0);
 signal Temp  : std_logic_vector(23 downto 0);
begin
 process(Clock, nReset) is
 begin
  if nReset = '0' then
   Clk    <= '1';
   Update <= '1';
   state  <= "00";
   count  <= "00000";
   Temp   <= "000000000000000000000000";
  elsif rising_edge(Clock) then
   case state is
    when "00" =>
     Update <= '1';
     if Latch = '1' then
      Temp(23 downto 12) <= DAC0(17 downto 6);
      Temp(11 downto  0) <= DAC1(17 downto 6);
      count <= "10111"; --23
      state <= "01";
     end if;
    when "01" =>
     Clk   <= '0';
     state <= "10";
    when "10" =>
     Clk   <= '1';
     if count = "00000" then
      state <= "11";
     else
      Temp(23 downto 1) <= Temp(22 downto 0);
      count <= count - '1';
      state <= "01";
     end if;
```



```
    when "11" =>
     Update <= '0';
     state  <= "00";
    when others =>
   end case;
  end if;
 end process;

 Data <= Temp(23);
end architecture a1;
```

E.3.11 DeadTime

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity DeadTime is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic;
  Count : in  std_logic_vector(7 downto 0);
  PWM   : in  std_logic;
  Dead  : in  std_logic_vector(7 downto 0);
  S1    : out std_logic;
  S2    : out std_logic
 );
end entity DeadTime;

architecture a1 of DeadTime is
 signal dClk    : std_logic_vector(7 downto 0);
 signal greater : std_logic;
 signal zero    : std_logic;
begin
 greater <= '1' when (not Dead) > Count else '0';
 zero    <= '1' when dClk = "00000000"  else '0';

 process(Clk, nReset, Dead) is
 begin
  if nReset = '0' then
   dClk <= Dead;
  elsif rising_edge(Clk) then
   if greater = '0' then
    dClk <= Dead;
   elsif (zero = '0') and (PWM = '0') then
    dClk <= dClk - '1';
   end if;
  end if;
 end process;

 S1   <= PWM and nReset;
 S2   <= (not PWM) and zero and greater and nReset;
end architecture a1;
```



E.3.12 Estimator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Estimator is
 port(
  nReset   : in std_logic;
```

```
  Clk       : in std_logic; --48 828 125 Hz, 20.48 ns
  dtClk     : in std_logic; --5 960 Hz, 167.772 16 us
  ADC_Busy : in std_logic;

  V2 : in std_logic_vector(17 downto 0);
  I2 : in std_logic_vector(17 downto 0);

  I3 : out std_logic_vector(18 downto 0)
 );
end entity Estimator;

architecture a1 of Estimator is
 component Mul18 is
  port(
   X1   : in std_logic_vector(17 downto 0);
   X2   : in std_logic_vector(17 downto 0);
   Sign : in std_logic;

   Y    : out std_logic_vector(36 downto 0)
  );
 end component Mul18;

 component RAM_64x18 is
  port(
   clock     : in std_logic;
   data      : in std_logic_vector(17 downto 0);
   rdaddress : in std_logic_vector( 5 downto 0);
   wraddress : in std_logic_vector( 5 downto 0);
   wren      : in std_logic := '1';
   q         : out std_logic_vector(17 downto 0)
  );
 end component RAM_64x18;

 signal x1  : std_logic_vector(17 downto 0);
 signal x2  : std_logic_vector(18 downto 0);
 signal x3  : std_logic_vector(18 downto 0);
 signal x4  : std_logic_vector(18 downto 0);
 signal x4a : std_logic_vector(18 downto 0);
 signal x5  : std_logic_vector(36 downto 0);
 signal x6  : std_logic_vector(35 downto 0);
 signal x7  : std_logic_vector(36 downto 0);
 signal x7a : std_logic_vector(36 downto 0);
 signal I3a : std_logic_vector(18 downto 0);

 signal Data      : std_logic_vector(17 downto 0);
 signal Address   : std_logic_vector( 5 downto 0);
 signal AddressP1 : std_logic_vector( 5 downto 0);
 signal RAM       : std_logic_vector(17 downto 0);
 signal at        : std_logic_vector(23 downto 0);

 signal state : std_logic_vector(1 downto 0);
begin
 AddressP1 <= Address + '1';
 FIFO : RAM_64x18 port map(not Clk, Data, AddressP1, Address, '1', RAM);

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   Data    <= "000000000000000000";
   Address <= "000000";
   at      <= "000000000000000000000000";
   state   <= "00";
  elsif rising_edge(Clk) then
   case state is
    when "00" =>
```

```
      Address <= AddressP1;
      if Address = "111111" then
       state <= "01";
      end if;
     when "01" =>
      if ADC_Busy = '0' then
       state <= "10";
      end if;
     when "10" =>
      if ADC_Busy = '1' then
       Data     <= V2;
       Address  <= AddressP1;
       at       <= at - RAM + V2;
       state <= "01";
      end if;
     when others =>
    end case;
   end if;
  end process;

  process(dtClk) is
  begin
   if falling_edge(dtClk) then
    x2(17 downto 0) <= x1;
    x1               <= at(22 downto 5);
   end if;
  end process;
  x2(18) <= '0';
  x3 <= (not x2) + '1';
  x4 <= x1 + x3;
  x4a <= (not x4) + '1' when x4(18) = '1' else x4;
  Mul1 : Mul18 port map(x4a(17 downto 0), "100001100111011111", x4(18), x5);
  x6(35 downto 30) <= "000000";
  x6(29 downto 12) <= I2;
  x6(11 downto  0) <= "000000000000";
  x7 <= x5 + x6;
  x7a <= (not x7) + '1' when x7(36) = '1' else x7;
  I3a <= "011111111111111111" when
         x7a(36 downto 30) /= "0000000" else x7a(30 downto 12);
  I3 <= (not I3a) + '1' when x7(36) = '1' else I3a;
end architecture a1;
```

E.3.13 Invert

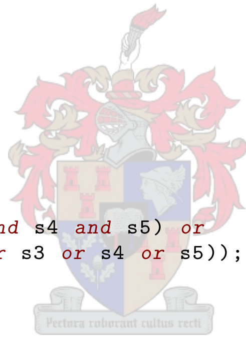
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Invert is
 port(
  Clk      : in  std_logic;
  ADC_Busy : in  std_logic;
  nReset   : in  std_logic;
  x1       : in  std_logic_vector(17 downto 0); --300  full-scale
  x2       : out std_logic_vector(17 downto 0)  --2/15 full-scale
 );
end entity Invert;

architecture a1 of Invert is
 constant Div_Num : std_logic_vector(35 downto 0) :=
                    "000001100110011001100110011001100110";
 signal    Temp    : std_logic_vector(17 downto 0);
 signal    Temp2   : std_logic_vector(17 downto 0);
 signal    Mul_A   : std_logic_vector(17 downto 0);
```

```
 signal   Mul_B   : std_logic_vector(17 downto 0);
 signal   Mul_Y   : std_logic_vector(35 downto 0);
 signal   greater : std_logic;
 signal   state   : std_logic_vector( 1 downto 0);
begin

 Mul_Y   <= Mul_A * Mul_B;
 greater <= '1' when Mul_Y > Div_Num else '0';

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state <= "00";
  elsif falling_edge(Clk) then
   case state is
    when "00" =>
     if ADC_Busy = '1' then
      state <= "01";
     end if;
    when "01" =>
     if ADC_Busy = '0' then
      state <= "10";
     end if;
    when "10" =>
     Temp  <= "011111111111111111";
     Temp2 <= "010000000000000000";
     Mul_A <= "100000000000000000";
     Mul_B <= x1;
     state <= "11";
    when "11" =>
     if Temp2 = "000000000000000000" then
      if greater = '1' then
       x2 <= (Mul_A and Temp) or Temp2;
      else
       x2 <= Mul_A or Temp2;
      end if;
      state <= "00";
     else
      if greater = '1' then
       Mul_A <= (Mul_A and Temp) or Temp2;
      else
       Mul_A <= Mul_A or Temp2;
      end if;
     end if;
     Temp (16 downto 0) <= Temp (17 downto 1); Temp(17) <= '1';
     Temp2(16 downto 0) <= Temp2(17 downto 1);
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.3.14 Keypad

```
library ieee;
use ieee.std_logic_1164.all;

entity Keypad is
 port(
  nReset : in  std_logic;
  Clk    : in  std_logic;
  Status : out std_logic_vector(15 downto 0);

  X      : in  std_logic_vector( 4 downto 1);
```

```
  Y       : out std_logic_vector( 4 downto 1)
 );
end entity Keypad;

architecture a1 of Keypad is
 signal state : std_logic_vector( 3 downto 0);
 signal s0    : std_logic_vector(15 downto 0);
 signal s1    : std_logic_vector(15 downto 0);
 signal s2    : std_logic_vector(15 downto 0);
 signal s3    : std_logic_vector(15 downto 0);
 signal s4    : std_logic_vector(15 downto 0);
 signal s5    : std_logic_vector(15 downto 0);
 signal s6    : std_logic_vector(15 downto 0);
 signal H     : std_logic_vector(15 downto 0);
 signal L     : std_logic_vector(15 downto 0);
begin
 H <= s1 and s2 and s3 and s4 and s5;
 L <= s1 or  s2 or  s3 or  s4 or  s5;

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state <= "0001";
  elsif rising_edge(Clk) then
   state(3 downto 1) <= state(2 downto 0);
   state(         0) <= state(         3);
  elsif falling_edge(Clk) then
   case state is
    when "0001" =>
     s5 <= s4;
     s4 <= s3;
     s3 <= s2;
     s2 <= s1;
     s1 <= s0;
     s0( 3 downto  0) <= X;
    when "0010" =>
     s6 <= (s1 and s2 and s3 and s4 and s5) or
           (s6 and (s1 or s2 or s3 or s4 or s5));
     s0( 7 downto  4) <= X;
    when "0100" =>
     s0(11 downto  8) <= X;
    when "1000" =>
     s0(15 downto 12) <= X;
    when others =>
   end case;
  end if;
 end process;

 Status <= s6;

 Y(1) <= '1' when state(0) = '1' else 'Z';
 Y(2) <= '1' when state(1) = '1' else 'Z';
 Y(3) <= '1' when state(2) = '1' else 'Z';
 Y(4) <= '1' when state(3) = '1' else 'Z';
end architecture a1;
```



E.3.15 Keypad_Driver

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Keypad_Driver is
 port(
```

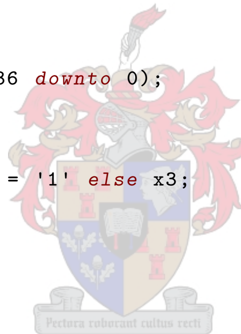
```
  nReset : in  std_logic;
  Clk    : in  std_logic;

  Data   : out std_logic_vector(7 downto 0); --Data(7) = Empty;
                                             --Data(3 downto 0) = Key
  Ack    : in  std_logic;

  X      : in  std_logic_vector(4 downto 1);
  Y      : out std_logic_vector(4 downto 1)
 );
end entity Keypad_Driver;

architecture a1 of Keypad_Driver is
 component Keypad is
  port(
   nReset : in  std_logic;
   Clk    : in  std_logic;
   Status : out std_logic_vector(15 downto 0);

   X      : in  std_logic_vector( 4 downto 1);
   Y      : out std_logic_vector( 4 downto 1)
  );
 end component Keypad;

 signal Keys    : std_logic_vector(15 downto 0);
 signal state   : std_logic;
 signal KeyDown : std_logic;
 signal Key     : std_logic_vector( 4 downto 0);

 signal FIFO0 : std_logic_vector(3 downto 0);
 signal FIFO1 : std_logic_vector(3 downto 0);
 signal FIFO2 : std_logic_vector(3 downto 0);
 signal FIFO3 : std_logic_vector(3 downto 0);
 signal FIFO4 : std_logic_vector(3 downto 0);
 signal FIFO5 : std_logic_vector(3 downto 0);
 signal FIFO6 : std_logic_vector(3 downto 0);
 signal FIFO7 : std_logic_vector(3 downto 0);

 signal W     : std_logic_vector(2 downto 0);
 signal Wp1   : std_logic_vector(2 downto 0);
 signal R     : std_logic_vector(2 downto 0);
 signal Empty : std_logic;
begin
 Keypad1 : Keypad port map(nReset, Clk, Keys, X, Y);

 KeyDown <= '0' when Keys = "0000000000000000" else '1';

 Wp1 <= W + '1';
 Empty <= '1' when W = R else '0';

 with Keys select
  Key <= "10000" when "0010000000000000", --0
         "10001" when "0000000000000001", --1
         "10010" when "0000000000000010", --2
         "10011" when "0000000000000100", --3
         "10100" when "0000000000010000", --4
         "10101" when "0000000000100000", --5
         "10110" when "0000000001000000", --6
         "10111" when "0000000100000000", --7
         "11000" when "0000001000000000", --8
         "11001" when "0000010000000000", --9
         "11010" when "0000000000001000", --A
         "11011" when "0000000010000000", --B
         "11100" when "0001000000000000", --C
         "11101" when "1000000000000000", --D
```

```
        "11110" when "0001000000000000", --*
        "11111" when "0100000000000000", --#
        "00000" when others;

process(Clk, nReset) is
begin
 if nReset = '0' then
  state <= '0';
  FIFO0 <= "0000";
  FIFO1 <= "0000";
  FIFO2 <= "0000";
  FIFO3 <= "0000";
  FIFO4 <= "0000";
  FIFO5 <= "0000";
  FIFO6 <= "0000";
  FIFO7 <= "0000";
  W <= "000";
 elsif rising_edge(Clk) then
  case state is
   when '0' =>
    if (KeyDown = '1') and (Wp1 /= R) and (Key(4) = '1') then
     case W is
      when "000" =>
       FIFO0 <= Key(3 downto 0);
      when "001" =>
       FIFO1 <= Key(3 downto 0);
      when "010" =>
       FIFO2 <= Key(3 downto 0);
      when "011" =>
       FIFO3 <= Key(3 downto 0);
      when "100" =>
       FIFO4 <= Key(3 downto 0);
      when "101" =>
       FIFO5 <= Key(3 downto 0);
      when "110" =>
       FIFO6 <= Key(3 downto 0);
      when "111" =>
       FIFO7 <= Key(3 downto 0);
      when others =>
     end case;
     W <= Wp1;
     state <= '1';
    end if;
   when '1' =>
    if KeyDown = '0' then
     state <= '0';
    end if;
   when others =>
  end case;
 end if;
end process;

process(Ack, nReset) is
begin
 if nReset = '0' then
  R <= "000";
 elsif rising_edge(Ack) then
  if Empty = '0' then
   R <= R + 1;
  end if;
 end if;
end process;

Data(         7) <= Empty;
Data(6 downto 4) <= "000";
```

```
 with R select
  Data(3 downto 0) <= FIFO0  when "000",
                      FIFO1  when "001",
                      FIFO2  when "010",
                      FIFO3  when "011",
                      FIFO4  when "100",
                      FIFO5  when "101",
                      FIFO6  when "110",
                      FIFO7  when "111",
                      "0000" when others;
end architecture a1;
```

E.3.16 Mul18

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Mul18 is
 port(
  X1   : in std_logic_vector(17 downto 0);
  X2   : in std_logic_vector(17 downto 0);
  Sign : in std_logic;

  Y    : out std_logic_vector(36 downto 0)
 );
end entity Mul18;

architecture a1 of Mul18 is
 signal x3 : std_logic_vector(36 downto 0);
begin
 x3(36) <= '0';
 x3(35 downto 0) <= X1 * X2;
 Y <= (not x3) + '1' when Sign = '1' else x3;
end architecture a1;
```



E.3.17 PWM1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity PWM1 is
 port(
  nReset    : in  std_logic;
  Clk       : in  std_logic; --97 656 250 Hz
  Duty      : in  std_logic_vector(20 downto 0);

  Delay_Out : out std_logic;
  Delay_In  : in  std_logic;
  Delay     : out std_logic_vector(9 downto 0);

  S1        : out std_logic;
  S2        : out std_logic
 );
end entity PWM1;

architecture a1 of PWM1 is
 constant Max : std_logic_vector(10 downto 0) := "11110011010"; --95%

 component Counter11 is
  port(
   nReset: in  std_logic;
```

```
   Clk    : in  std_logic;
   Q      : out std_logic_vector(10 downto 0)
  );
 end component Counter11;

 signal D       : std_logic_vector(20 downto 0);
 signal Count   : std_logic_vector(10 downto 0);
 signal greater : std_logic;
 signal state   : std_logic_vector(1 downto 0);
begin
 Counter : Counter11 port map(nReset, Clk, Count);

 greater <= '1' when (D(20 downto 10) > Count) and
                     (Max             > Count) else '0';

 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state <= "00";
  elsif rising_edge(Clk) then
   case state is
    when "00" =>
     if greater = '1' then
      state <= "01";
     end if;
    when "01" =>
     if greater = '0' then
      state <= "11";
     end if;
    when "11" =>
     if greater = '1' then
      state <= "10";
     end if;
    when "10" =>
     if greater = '0' then
      state <= "00";
     end if;
    when others =>
   end case;
   if Count = "11111111111" then
    D <= Duty;
   end if;
  end if;
 end process;

 S1 <= state(0);
 S2 <= state(1);

 Delay_Out <= '0';
 Delay     <= "0000000000";
end architecture a1;
```

E.3.18 PWM2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity PWM2 is
 port(
  nReset    : in  std_logic;
  Clk       : in  std_logic; --97 656 250 Hz
  Duty      : in  std_logic_vector(17 downto 0);
  Sample    : out std_logic; --ADC can sample when this high
```

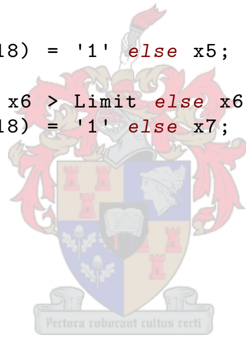
```
  NS        : in  std_logic; --noise-shaper on when '1', delay-line when '0'

  Delay_Out : out std_logic;
  Delay_In  : in  std_logic;
  Delay     : out std_logic_vector(9 downto 0);

  S1        : out std_logic;
  S2        : out std_logic
 );
end entity PWM2;

architecture a1 of PWM2 is
 constant Max  : std_logic_vector(7 downto 0) := "11110011"; --95%

 component Counter8 is
  port(
   nReset: in  std_logic;
   Clk   : in  std_logic;
   Q     : out std_logic_vector(7 downto 0)
  );
 end component Counter8;

 component DeadTime is
  port(
   nReset: in  std_logic;
   Clk   : in  std_logic;
   Count : in  std_logic_vector(7 downto 0);
   PWM   : in  std_logic;
   Dead  : in  std_logic_vector(7 downto 0);
   S1    : out std_logic;
   S2    : out std_logic
  );
 end component DeadTime;

 signal D       : std_logic_vector(17 downto 0);
 signal Count   : std_logic_vector( 7 downto 0);
 signal greater : std_logic;
 signal PWM     : std_logic;
 signal S0      : std_logic;

--Noise shaper signals:
 signal De   : std_logic_vector(10 downto 0);
 signal x1   : std_logic_vector(10 downto 0);
 signal x2   : std_logic_vector(11 downto 0);
 signal x3   : std_logic_vector(11 downto 0);
 signal x4   : std_logic_vector(12 downto 0);
 signal x5   : std_logic_vector(12 downto 0);
 signal x6   : std_logic_vector(13 downto 0);
 signal x7   : std_logic_vector(13 downto 0);
 signal x8   : std_logic_vector(14 downto 0);
 signal x8n  : std_logic_vector(14 downto 0);
 signal x9   : std_logic_vector(19 downto 0);
 signal x10  : std_logic_vector(19 downto 0);

 constant Dead : std_logic_vector(7 downto 0) := "00001101"; --133ns (5%)
-- constant Dead : std_logic_vector(7 downto 0) := "00110011"; --522ns (20%)
begin
 Counter : Counter8 port map(nReset, Clk, Count);

 greater <= '1' when (D(17 downto 10) > Count) and
                     (Max             > Count) else '0';

 process(Clk, nReset) is
 begin
  if nReset = '0' then
```

```
   Sample <= '0';
   PWM <= '0';
  elsif rising_edge(Clk) then
   PWM <= greater;
   if Count = "11111111" then
    if NS = '1' then
     if x10(19) = '1' then
      D <= "000000000000000000";
     elsif x10(18) = '1' then
      D <= "111111111111111111";
     else
      D <= x10(17 downto 0);
     end if;
    else
     D <= Duty;
    end if;
    Sample <= '1';
    x1 <= (not De) + '1';
    x3 <= (not x2) + '1';
    x5 <= (not x4) + '1';
    x7 <= (not x6) + '1';
   elsif Count(4) = '1' then
    Sample <= '0';
   end if;
  end if;
 end process;

 De( 9 downto  0) <= D(9 downto 0);
 De(           10) <= '0';
 x2(10 downto  0) <= De + x1;
 x2(           11) <= x2(10);
 x4(11 downto  0) <= x2 + x3;
 x4(           12) <= x4(11);
 x6(12 downto  0) <= x4 + x5;
 x6(           13) <= x6(12);
 x8(13 downto  0) <= x6 + x7;
 x8(           14) <= x8(13);
 x8n <= (not x8) + '1';
 x9(14 downto  0) <= x8 + De;
 x9(19 downto 15) <= "11111" when x9(14) = '1' else "00000";
 x10 <= x9 + Duty;

 Delay(0)  <= D( 6) and Delay_In;
 Delay(1)  <= D( 7) and Delay_In;
 Delay(2)  <= D( 8) and Delay_In;
 Delay(3)  <= D( 9) and Delay_In;
 Delay(4)  <= D(10) and Delay_In;
 Delay(5)  <= D(11) and Delay_In;
 Delay(6)  <= D(12) and Delay_In;
 Delay(7)  <= D(13) and Delay_In;
 Delay(8)  <= D(14) and Delay_In;
 Delay(9)  <= D(15) and Delay_In;
 Delay_Out <= PWM;
 S0 <= Delay_In when NS = '0' else PWM;

 DeadTime1 : DeadTime port map(nReset, Clk,
                               Count, S0, Dead,
                               S1, S2);
end architecture a1;
```

E.3.19 Ramp

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all;

entity Ramp is
 port(
  dtClk  : in  std_logic; --4 768 kHz, 209.715 200 us
  x1     : in  std_logic_vector(17 downto 0);
  Limit  : in  std_logic_vector(17 downto 0);
  x2     : out std_logic_vector(17 downto 0)
 );
end entity Ramp;

architecture a1 of Ramp is
 signal tx2 : std_logic_vector(18 downto 0);
 signal x3  : std_logic_vector(18 downto 0);
 signal x4  : std_logic_vector(18 downto 0);
 signal x5  : std_logic_vector(18 downto 0);
 signal x6  : std_logic_vector(18 downto 0);
 signal x7  : std_logic_vector(18 downto 0);
 signal x8  : std_logic_vector(18 downto 0);
begin
 process(dtClk) is
 begin
  if falling_edge(dtClk) then
   x3(17 downto 0) <= tx2(17 downto 0);
  end if;
 end process;
 x3(18) <= '0';
 x4 <= (not x3) + '1';
 x5 <= x1 + x4;
 x6 <= (not x5) + '1' when x5(18) = '1' else x5;
 x7(18) <= '0';
 x7(17 downto 0) <= Limit when x6 > Limit else x6(17 downto 0);
 x8 <= (not x7) + '1' when x5(18) = '1' else x7;
 tx2 <= x8 + x3;
 x2 <= tx2(17 downto 0);
end architecture a1;
```



E.3.20 RealTime

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RealTime is
 port(
  nReset: in  std_logic;
  Clk   : in  std_logic; --48 828 125 Hz
  Q     : out std_logic_vector(7 downto 0)
 );
end entity RealTime;

architecture a1 of RealTime is
 signal count: std_logic_vector(25 downto 0);
 signal D    : std_logic_vector(25 downto 0);
 signal ands : std_logic_vector(25 downto 1);
 signal tQ   : std_logic_vector( 7 downto 0);
begin
 ands(          1) <= count(0);
 ands(25 downto 2) <= ands(24 downto 1) and count(24 downto 1);

 D(          0) <= not count(0);
 D(25 downto 1) <= count(25 downto 1) xor ands(25 downto 1);

 process(Clk, nReset) is
```

```
 begin
  if nReset = '0' then
   count <= "00000000000000000000000000";
   tQ    <= "00000000";
  elsif falling_edge(Clk) then
   if D = "10111010010000111011011101" then
   count <= "00000000000000000000000000";
   tQ <= tQ + '1';
   else
    count <= D;
   end if;
  end if;
 end process;

 Q <= tQ;
end architecture a1;
```

E.3.21 RS232Tx

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RS232Tx is
 port(
  nReset   : in  std_logic;
  Clk      : in  std_logic; --6 103 516 Hz
  Tx       : out std_logic;
  Data     : in  std_logic_vector(7 downto 0);
  Send     : in  std_logic; --Pulsed high to start transmission
  CS       : out std_logic  --Pulsed high at end of transmission
 );
end entity RS232Tx;

architecture a1 of RS232Tx is
 signal state    : std_logic_vector(1 downto 0);
 signal retstate : std_logic_vector(1 downto 0);
 signal count    : std_logic_vector(5 downto 0); --115200 baud
 signal tdata    : std_logic_vector(7 downto 0);
 signal count2   : std_logic_vector(2 downto 0);
begin
 process(Clk, nReset) is
 begin
  if nReset = '0' then
   state  <= "00";
   count  <= "000000";
   count2 <= "111";
   tdata  <= "00000000";
   Tx     <= '1';
   CS     <= '0';
  elsif rising_edge(Clk) then
   case state is
    when "00" =>
     if Send = '1' then
      Tx       <= '0';
      tData    <= Data;
      count    <= "110100"; --52
      retstate <= "10";
      state    <= "01";
     end if;
    when "01" =>
     if count = "000000" then
      state <= retstate;
     end if;
```

```
     count <= count - '1';
    when "10" =>
     Tx <= tdata(0);
     tdata(6 downto 0) <= tdata(7 downto 1);
     count2 <= count2 - '1';
     count  <= "110100"; --52
     state  <= "01";
     if count2 = "000" then
      CS <= '1';
      retstate <= "11";
     else
      retstate <= "10";
     end if;
    when "11" =>
     Tx <= '1';
     CS <= '0';
     count    <= "110100"; --52
     retstate <= "00";
     state    <= "01";
    when others =>
   end case;
  end if;
 end process;
end architecture a1;
```

E.3.22 Stage1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Stage1 is
 port(
  Enable : in std_logic;
  dtClk  : in std_logic; --5 960 Hz, 167.772 16 us

  Ref  : in std_logic_vector(17 downto 0);
  V1   : in std_logic_vector(17 downto 0);
  I1   : in std_logic_vector(17 downto 0);

  KV1 : in std_logic_vector(18 downto 0); --sign:abs(x)
  KI1 : in std_logic_vector(18 downto 0); --sign:abs(x)
  N1  : in std_logic_vector(17 downto 0);
  R1  : in std_logic_vector(17 downto 0);

  D1 : out std_logic_vector(20 downto 0)
 );
end entity Stage1;

architecture a1 of Stage1 is
 component Ramp is
  port(
   dtClk   : in  std_logic; --5 960 Hz, 167.772 16 us
   x1      : in  std_logic_vector(17 downto 0);
   Limit   : in  std_logic_vector(17 downto 0);
   x2      : out std_logic_vector(17 downto 0)
  );
 end component Ramp;

 component Mul18 is
  port(
   X1   : in std_logic_vector(17 downto 0);
   X2   : in std_logic_vector(17 downto 0);
   Sign : in std_logic;
```

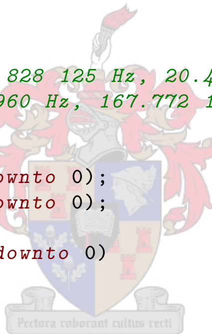
Pectora roborant cultus recti

```
   Y    : out std_logic_vector(36 downto 0)
  );
 end component Mul18;

 signal x1  : std_logic_vector(17 downto 0);
 signal x2  : std_logic_vector(17 downto 0);
 signal x3  : std_logic_vector(17 downto 0);
 signal x4  : std_logic_vector(38 downto 0);
 signal x5  : std_logic_vector(38 downto 0);
 signal x6  : std_logic_vector(38 downto 0);
 signal x7  : std_logic_vector(38 downto 0);
 signal x7a : std_logic_vector(37 downto 0);
begin
 x1   <= V1 when Enable = '0' else Ref;
 x2   <= "000011001100110011" when x1 < "000011001100110011" else x1;
 Ramp1 : Ramp port map(dtClk, x2, R1, x3);
 Mul1 : Mul18 port map(I1, KI1(17 downto 0), KI1(18), x4(36 downto 0));
 Mul2 : Mul18 port map(V1, KV1(17 downto 0), KV1(18), x5(36 downto 0));
 Mul3 : Mul18 port map(x3, N1                , '0'    , x6(36 downto 0));
 x4(38)  <= x4(36);
 x4(37)  <= x4(36);
 x5(38)  <= x5(36);
 x5(37)  <= x5(36);
 x6(38)  <= x6(36);
 x6(37)  <= x6(36);
 x7  <= x4 + x5 + x6;
 x7a  <= "00000000000000000000000000000000000000" when x7(38) = '1' else
                                                      x7(37 downto 0);
 D1  <= "11111111111111111111" when x7a(37 downto 26) /= "000000000000" else
                                    x7a(25 downto 5);
end architecture a1;
```

E.3.23 Stage2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity Stage2 is
 port(
  nReset    : in std_logic;
  Enable    : in std_logic;
  Clk       : in std_logic; --48 828 125 Hz, 20.48 ns
  dtClk     : in std_logic; --5 960 Hz, 167.772 16 us
  ADC_Busy : in std_logic;

  Ref : in std_logic_vector(17 downto 0);
  Int : in std_logic;

  KV2  : in std_logic_vector(18 downto 0); --sign:abs(x)
  KI2  : in std_logic_vector(18 downto 0); --sign:abs(x)
  KI3  : in std_logic_vector(18 downto 0); --sign:abs(x)
  N2   : in std_logic_vector(17 downto 0);
  R2   : in std_logic_vector(17 downto 0);
  KInt : in std_logic_vector(17 downto 0);

  V1  : in std_logic_vector(17 downto 0);
  V2  : in std_logic_vector(17 downto 0);
  I2  : in std_logic_vector(17 downto 0);

  Locked : out std_logic;
  High   : out std_logic;
  Low    : out std_logic;
```

```
  V3 : out std_logic_vector(17 downto 0);
  I3 : out std_logic_vector(17 downto 0);
  D2 : out std_logic_vector(17 downto 0)
 );
end entity Stage2;

architecture a1 of Stage2 is
 component Ramp is
  port(
   dtClk   : in  std_logic; --5 960 Hz, 167.772 16 us
   x1      : in  std_logic_vector(17 downto 0);
   Limit   : in  std_logic_vector(17 downto 0);
   x2      : out std_logic_vector(17 downto 0)
  );
 end component Ramp;

 component Invert is
  port(
   Clk      : in  std_logic;
   ADC_Busy : in  std_logic;
   nReset   : in  std_logic;
   x1       : in  std_logic_vector(17 downto 0); --300  full-scale
   x2       : out std_logic_vector(17 downto 0)  --2/15 full-scale
  );
 end component Invert;

 component Estimator is
  port(
   nReset   : in std_logic;
   Clk      : in std_logic; --48 828 125 Hz, 20.48 ns
   dtClk    : in std_logic; --5 960 Hz, 167.772 16 us
   ADC_Busy : in std_logic;

   V2 : in std_logic_vector(17 downto 0);
   I2 : in std_logic_vector(17 downto 0);

   I3 : out std_logic_vector(18 downto 0)
  );
 end component Estimator;

 component Comp_Avg is
  port(
   nReset : in std_logic;
   Clk : in std_logic;

   Hi : in std_logic;
   Lo : in std_logic;

   Locked : out std_logic;
   High   : out std_logic;
   Low    : out std_logic
  );
 end component Comp_Avg;

 component Mul18 is
  port(
   X1   : in std_logic_vector(17 downto 0);
   X2   : in std_logic_vector(17 downto 0);
   Sign : in std_logic;

   Y    : out std_logic_vector(36 downto 0)
  );
 end component Mul18;
```



```
 signal x1   : std_logic_vector(35 downto 0);
 signal x2   : std_logic_vector(17 downto 0);
 signal x3   : std_logic_vector(17 downto 0);
 signal x3a  : std_logic_vector(17 downto 0);
 signal x4   : std_logic_vector(18 downto 0);
 signal x5   : std_logic_vector(18 downto 0);
 signal x5a  : std_logic_vector(18 downto 0);
 signal x6   : std_logic_vector(36 downto 0);
 signal x7   : std_logic_vector(35 downto 0);
 signal x8   : std_logic_vector(36 downto 0);
 signal x8a  : std_logic_vector(35 downto 0);
 signal x9   : std_logic_vector(35 downto 0);
 signal x10  : std_logic_vector(35 downto 0);
 signal x11  : std_logic_vector(38 downto 0);
 signal x12  : std_logic_vector(38 downto 0);
 signal x13  : std_logic_vector(38 downto 0);
 signal x14  : std_logic_vector(38 downto 0);
 signal x15  : std_logic_vector(38 downto 0);
 signal x15a : std_logic_vector(37 downto 0);
 signal x16  : std_logic_vector(17 downto 0);
 signal x17  : std_logic_vector(17 downto 0);
 signal x18  : std_logic_vector(35 downto 0);

 signal tV3 : std_logic_vector(17 downto 0);
 signal tI3 : std_logic_vector(18 downto 0);
 signal I3a : std_logic_vector(18 downto 0);

 signal Lock : std_logic;
 signal Hi   : std_logic;
 signal Lo   : std_logic;
begin
 Estimator1 : Estimator port map(nReset, Clk, dtClk, ADC_Busy, V2, I2, tI3);
 x1  <= Ref * "110000000000000000";
 x2  <= I2 when Enable = '0' else x1(35 downto 18);
 Ramp1 : Ramp port map(dtClk, x2, R2, x3);
 x3a(17)  <= '0';
 x3a(16 downto 0)  <= x3(17 downto 1);
 x4(18 downto 17)  <= "00";
 x4(16 downto  0)  <= I2(17 downto 1);
 x5  <= x3a + ((not x4) + '1');
 x5a  <= (not x5) + '1' when x5(18) = '1' else x5;
 MulInt : Mul18 port map(x5a(17 downto 0), KInt, x5(18), x6);
 process(dtClk) is
 begin
  if falling_edge(dtClk) then
   x7  <= x10;
  end if;
 end process;
 x8  <= x6 + x7;
 x8a  <= "000000000000000000000000000000000000" when x8(36) = '1' else
                                                  x8(35 downto 0);
 x9  <= "111001100110011001100110011001100110" when
        x8a > "111001100110011001100110011001100110" else x8a;
 x10(35 downto 18)  <= x3                          when (Enable and Int) = '0' else
                                                 x9(35 downto 18);
 x10(17 downto  0)  <= "000000000000000000" when (Enable and Int) = '0' else
                                                 x9(17 downto  0);
 I3a  <= (not tI3) + '1' when tI3(18) = '1' else tI3;
 I3   <= tI3(17 downto 0) when tI3(18) = '0' else "000000000000000000";
 Mul1  : Mul18 port map(I3a(17 downto 0), KI3(17 downto 0),
                      KI3(18) xor tI3(18), x11(36 downto 0));
 Mul2  : Mul18 port map(V2 , KV2(17 downto 0), KV2(18), x12(36 downto 0));
 Mul3  : Mul18 port map(I2 , KI2(17 downto 0), KI2(18), x13(36 downto 0));
 Mul4  : Mul18 port map(x10(35 downto 18), N2, '0', x14(36 downto 0));
 x11(38)  <= x11(36);
```

```
 x11(37) <= x11(36);
 x12(38) <= x12(36);
 x12(37) <= x12(36);
 x13(38) <= x13(36);
 x13(37) <= x13(36);
 x14(38) <= x14(36);
 x14(37) <= x14(36);
 x15 <= X11 + x12 + x13 + x14;
 x15a <= "000000000000000000000000000000000000000" when x15(38) = '1' else
                                                        x15(37 downto 0);
 tV3 <= "111111111111111111" when x15a(37 downto 36) /= "00" else
                                  x15a(35 downto 18);
 V3  <= tV3;
 x16 <= "000010001000100010" when V1 < "000010001000100010" else V1;
 Invert1 : Invert port map(Clk, ADC_Busy, nReset, x16, x17);
 x18 <= x17 * tV3;
 D2  <= "111111111111111111" when x18(35 downto 31) /= "00000" else
                                  x18(30 downto 13);

 Hi      <= x5(18);
 Lo      <= '1' when (x5(18) = '0') and (x5(17 downto 0) /=
                                        "000000000000000000") else '0';
 Comp_Avg1 : Comp_Avg port map(nReset, dtClk, Hi, Lo, Locked, High, Low);
end architecture a1;
```

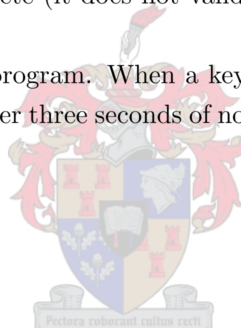
Appendix F

CPU Source Code

F.1 Introduction

The source code in this appendix in the source that has been loaded onto the CPU in order to test it. The allocated address of the bus differs from those listed in section 8.4.7 on page 135. The boot-loader is not complete (it does not validate the contents of the EEPROM), but it is functional.

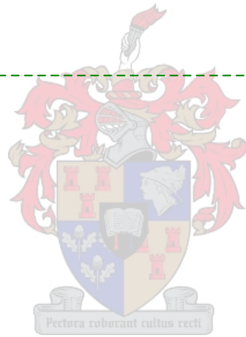
The interface is a simple test program. When a key is pressed, it prints it to the LCD and switches the back-light on. After three seconds of no key-press, the backlight is switched off.



F.2 Bootloader

```
Bootloader:
  ldi 0x00
  st 0x0000
  ldi 0x00
  st 0x0001
Start:
  call EEPROM_Read
  stpr pop 0x0000
  ld  0x0001
  ld  0x0000
  ldi 0x01
  add pop
  swp 1
  ldi 0x00
  adc pop
  swp 1
  st  0x0000
  swp 1
  st  0x0001
  ldi 0xC0
  and
  pop
z jmp Start
  pop
  pop
```

```
  jmp Boot
;-----------------------------------------------------------------------

EEPROM_Wait:
  ld  0x808C
  rol
  pop
z ret
  jmp EEPROM_Wait
;-----------------------------------------------------------------------

EEPROM_Read: ;Push as Address0, Address1
  call EEPROM_Wait
  st pop 0x808A
  st pop 0x8089
  ldi 0x02
  st pop 0x808C
  call EEPROM_Wait
  ld 0x808B
  ret
;-----------------------------------------------------------------------

EEPROM_Write: ;Push as Address0, Address1, Data
  call EEPROM_Wait
  st pop 0x808B
  st pop 0x808A
  st pop 0x8089
  ldi 0x01
  st pop 0x808C
  ret
;-----------------------------------------------------------------------

COMM_Read:
  ldi 0x02
  ld  0x8082
  and pop
  pop
  z jmp COMM_Read
  ld  0x8081
  ldi 0x00
  st  pop 0x8082
  ret
;-----------------------------------------------------------------------

COMM_Write:
  ldi 0x01
  ld  0x8082
  and pop
  pop
  nz
  z jmp COMM_Write
  st  pop 0x8080
  ldi 0x11
  st  pop 0x8082
  ret
;-----------------------------------------------------------------------

Boot:
  jmp Main
;-----------------------------------------------------------------------

ProgrammerCheck:
  ldi 0x02
  ld  0x8082
  and pop
```



```
  pop
z ret
  call COMM_Read
case1: ;Write eeprom
  ldi 0x01;
  sub
  pop
  nz
  z jmp case2
  pop
  call COMM_Read
  call COMM_Read
  call COMM_Read
  call EEPROM_Write
  ldi  0x00
  call COMM_Write
  ret
case2: ;Read eeprom
  ldi 0x02;
  sub
  pop
  nz
  z jmp case3
  pop
  call COMM_Read
  call COMM_Read
  call EEPROM_Read
  call COMM_Write
  ret
case3: ;Reboot
  ldi 0x03;
  sub
  pop
  nz
  z jmp default
  pop
  ldi 0x00
  st pop 0x808D ;Switch off LCD backlight
  jmp Bootloader
default:
  pop
  ret
;-------------------------------------------------------------------------
;--END OF BOOTLOADER, MAY EDIT FROM HERE ONWARDS-------------------------

Main:
Loop:
  call ProgrammerCheck
  jmp  Loop
;-------------------------------------------------------------------------
```

F.3 Interface

```
;Temporary   0000 -> 0001
;LCD_Address 0002 -> 0003
;Timer       0004 -> 0004

Bootloader:
  ldi 0x00
  st 0x0000
  ldi 0x00
  st 0x0001
Start:
```

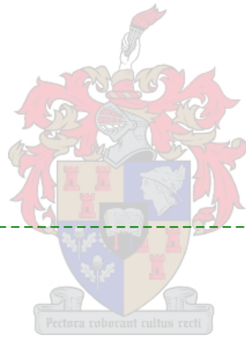
```
  call EEPROM_Read
  stpr pop 0x0000
  ld   0x0001
  ld   0x0000
  ldi 0x01
  add pop
  swp 1
  ldi 0x00
  adc pop
  swp 1
  st   0x0000
  swp 1
  st   0x0001
  ldi 0xC0
  and
  pop
z jmp Start
  pop
  pop
  jmp Boot
;------------------------------------------------------------------------

EEPROM_Wait:
  ld   0x808C
  rol
  pop
z ret
  jmp EEPROM_Wait
;------------------------------------------------------------------------

EEPROM_Read: ;Push as Address0, Address1
  call EEPROM_Wait
  st pop 0x808A
  st pop 0x8089
  ldi 0x02
  st pop 0x808C
  call EEPROM_Wait
  ld 0x808B
  ret
;------------------------------------------------------------------------
```

```
EEPROM_Write: ;Push as Address0, Address1, Data
  call EEPROM_Wait
  st pop 0x808B
  st pop 0x808A
  st pop 0x8089
  ldi 0x01
  st pop 0x808C
  ret
;------------------------------------------------------------------------

COMM_Read:
  ldi 0x02
  ld   0x8082
  and pop
  pop
  z jmp COMM_Read
  ld   0x8081
  ldi 0x00
  st   pop 0x8082
  ret
;------------------------------------------------------------------------

COMM_Write:
  ldi 0x01
```

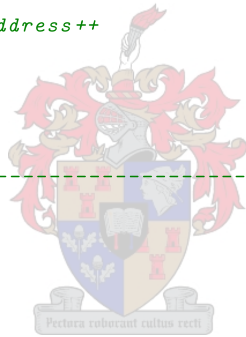
```
  ld  0x8082
  and pop
  pop
  nz
  z jmp COMM_Write
  st  pop 0x8080
  ldi 0x11
  st  pop 0x8082
  ret
;------------------------------------------------------------------------

Boot:
  jmp Main
;------------------------------------------------------------------------

ProgrammerCheck:
  ldi 0x02
  ld  0x8082
  and pop
  pop
z ret
  call COMM_Read
case1: ;Write eeprom
  ldi 0x01;
  sub
  pop
  nz
  z jmp case2
  pop
  call COMM_Read
  call COMM_Read
  call COMM_Read
  call EEPROM_Write
  ldi  0x00
  call COMM_Write
  ret
case2: ;Read eeprom
  ldi 0x02;
  sub
  pop
  nz
  z jmp case3
  pop
  call COMM_Read
  call COMM_Read
  call EEPROM_Read
  call COMM_Write
  ret
case3: ;Reboot
  ldi 0x03;
  sub
  pop
  nz
  z jmp CommCheck
  pop
  ldi 0x00
  st pop 0x808D ;Switch off LCD backlight
  jmp Bootloader
;------------------------------------------------------------------------
;--END OF BOOTLOADER, MAY EDIT FROM HERE ONWARDS-------------------------

Keypad:
  ld  0x808E
  rcl
  pop
```

Pectora roborant cultus recti

```
c jmp Keypad
  ld  0x808E
  st  0x808E
  ret
;-----------------------------------------------------------------------

LCD_Clear:
  ldi 0x00
  st  0x0002
  ldi 0x80
  st pop 0x0003
S1:
  ldi 0x20
  st pop ref 0x0002
  ldi 0x01
  add pop
  st 0x0002
  rcl
  shr
  nc
c jmp S1
  pop
  ldi  0x00
  st pop 0x0002
  ret
;-----------------------------------------------------------------------

Main:
  ldi  0x02
  st pop 0x808D
  call LCD_Clear  ;Clear LCD
Loop:
  call ProgrammerCheck
  call KeypadCheck
  call TimerCheck
  jmp  Loop
;-----------------------------------------------------------------------

CommCheck:
case4: ;Write RAM
  ldi 0x04;
  sub
  pop
  nz
  z jmp case5
  pop
  call COMM_Read
  st pop 0x0000
  call COMM_Read
  st pop 0x0001
  call COMM_Read
  st pop ref 0x0000
  ret
case5: ;Read RAM
  ldi 0x05;
  sub
  pop
  nz
  z jmp default2
  pop
  call COMM_Read
  st pop 0x0000
  call COMM_Read
  st pop 0x0001
  ld ref 0x0000
```



```
  call COMM_Write
  ret
default2:
  pop
  ret
;-------------------------------------------------------------------------

KeypadCheck:
  ld  0x808E
  rcl
  pop
c ret
  call Keypad            ;Read keypad key
  ldi 0x41
  add pop                ;Add 'A' to key
  st pop ref 0x0002      ;Write key to LCD
  ld 0x0002
  ldi 0x01
  add pop
  ld 0x8095
  st pop 0x0004          ;Timer = TickCount
  ld 0x808D
  ldi 0x10
  or pop
  st pop 0x808D
  rcl
c jmp LCD_Full
  shr
  st pop 0x0002          ;LCD_Address++
  ret
LCD_Full:
  pop
  call LCD_Clear
  ret
;-------------------------------------------------------------------------

TimerCheck:
  ld 0x8095
  ld 0x0004
  sub pop
  ldi 0x03
  sub pop
  pop
  nz
z ret
  ld 0x808D
  ldi 0xEF
  and pop
  st pop 0x808D
  ret
;-------------------------------------------------------------------------
```



Appendix G

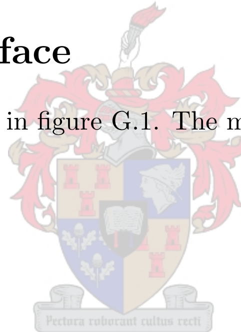
Programmer Source Code

G.1 Introduction

The CPU programmer was programmed in Borland C++ Builder 4 and is fully-functional. This appendix presents the program code.

G.2 Graphical Interface

The programmer window is shown in figure G.1. The menus are shown in figure G.2.



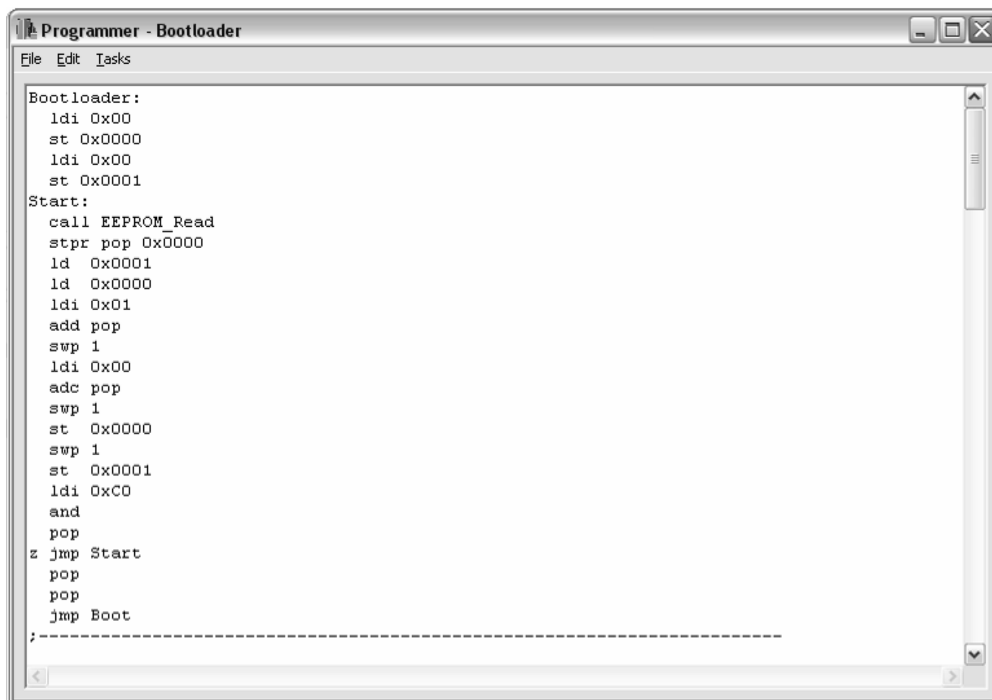


Figure G.1: Programmer screen-shot

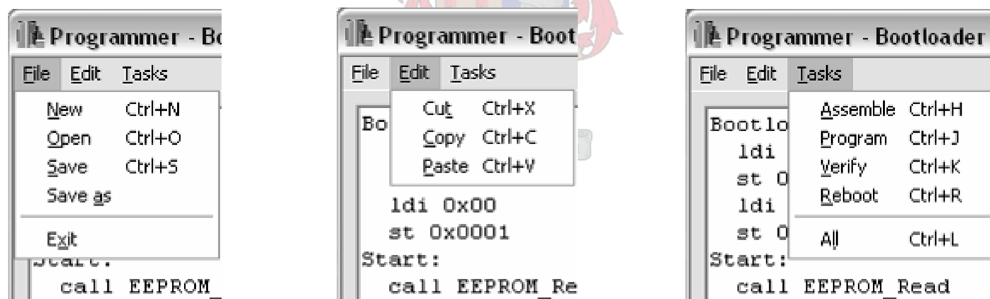


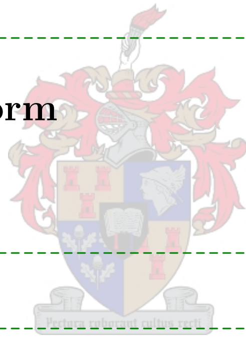
Figure G.2: Programmer menus

G.3 Programmer

G.3.1 Body

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
USERES("Programmer.res");
USEFORM("FrmProgrammer.cpp", Prog);
USEUNIT("..\..\..\..\Shared\Cpp\File\JFile.cpp");
USEUNIT("..\..\..\..\Shared\Cpp\Comm\JComm.cpp");
USEUNIT("Assembler.cpp");
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
     Application->Initialize();
     Application->CreateForm(__classid(TProg), &Prog);
     Application->Run();
  }
  catch (Exception &exception)
  {
     Application->ShowException(&exception);
  }
  return 0;
}
//---------------------------------------------------------------------------
```

G.4 Programmer Form



G.4.1 Header

```
//---------------------------------------------------------------------------
#ifndef FrmProgrammerH
#define FrmProgrammerH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <clipbrd.hpp>
#include <Menus.hpp>
#include <Dialogs.hpp>
//---------------------------------------------------------------------------
#include "JFile.h"
#include "JComm.h"
#include "Assembler.h"
//---------------------------------------------------------------------------
class TProg : public TForm
{
__published:     // IDE-managed Components
  TMemo *Asm;
  TMainMenu *MainMenu1;
  TMenuItem *File1;
  TMenuItem *Edit1;
  TMenuItem *Tasks1;
  TMenuItem *New1;
  TMenuItem *Open1;
  TMenuItem *Save1;
  TMenuItem *Saveas1;
```

```
  TMenuItem *N1;
  TMenuItem *Exit1;
  TMenuItem *Cut1;
  TMenuItem *Copy1;
  TMenuItem *Paste1;
  TMenuItem *Assemble1;
  TMenuItem *Program1;
  TMenuItem *Verify1;
  TMenuItem *N2;
  TMenuItem *All1;
  TOpenDialog *OpenDialog;
  TSaveDialog *SaveDialog;
  TMenuItem *Reboot1;
  void __fastcall FormResize(TObject *Sender);
  void __fastcall New1Click(TObject *Sender);
  void __fastcall Open1Click(TObject *Sender);
  void __fastcall Save1Click(TObject *Sender);
  void __fastcall Saveas1Click(TObject *Sender);
  void __fastcall Exit1Click(TObject *Sender);
  void __fastcall Cut1Click(TObject *Sender);
  void __fastcall Copy1Click(TObject *Sender);
  void __fastcall Paste1Click(TObject *Sender);
  void __fastcall Assemble1Click(TObject *Sender);
  void __fastcall Program1Click(TObject *Sender);
  void __fastcall Verify1Click(TObject *Sender);
  void __fastcall All1Click(TObject *Sender);
  void __fastcall AsmChange(TObject *Sender);
  void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
  void __fastcall Reboot1Click(TObject *Sender);
private:        // User declarations
  JFile File;
  JComm Comm;
  JAssembler Assembler;
  bool Saved;
  bool Success;
  bool Messages;
  AnsiString Path    ; //Including the ending "/"
  AnsiString Filename; //Only the name, not the extension
public:         // User declarations
  __fastcall TProg(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TProg *Prog;
//---------------------------------------------------------------------------
#endif
```

G.4.2 Body

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "FrmProgrammer.h"
//---------------------------------------------------------------------------

#pragma package(smart_init)
#pragma resource "*.dfm"
TProg *Prog;
//---------------------------------------------------------------------------

__fastcall TProg::TProg(TComponent* Owner)
  : TForm(Owner)
{
 Saved    = true;
```

Pectora roborant cultus recti

```
 Path     = "";
 Filename = "";
 Caption  = "Programmer - [Untitled]";
 Messages = true;
}
//---------------------------------------------------------------------------

void __fastcall TProg::FormResize(TObject *Sender)
{
 Asm->Width  = ClientWidth  - 16;
 Asm->Height = ClientHeight - 16;
}
//---------------------------------------------------------------------------

void __fastcall TProg::New1Click(TObject *Sender)
{
 if(!Saved){
  switch(Application->MessageBox("Would you like to save first",
                                 "Not saved",
                                 MB_ICONQUESTION | MB_YESNOCANCEL)){
   case IDYES:
    Save1Click(this);
   case IDNO:
    break;
   default:
    return;
  }
 }
 Saved    = true;
 Path     = "";
 Filename = "";
 Asm->Lines->Clear();
 Caption  = "Programmer - [Untitled]";
}
//---------------------------------------------------------------------------

void __fastcall TProg::Open1Click(TObject *Sender)
{
 if(!Saved){
  switch(Application->MessageBox("Would you like to save first",
                                 "Not saved",
                                 MB_ICONQUESTION | MB_YESNOCANCEL)){
   case IDYES:
    Save1Click(this);
   case IDNO:
    break;
   default:
    return;
  }
 }
 if(!OpenDialog->Execute()) return;
 int j;
 Filename = OpenDialog->FileName;
 for(j = Filename.Length(); Filename[j] != '\\'; j--);
 Path     = Filename.SubString(1, j);
 Filename = Filename.SubString(j+1, Filename.Length()-j);
 if(Filename.SubString(Filename.Length()-3, 4).UpperCase() == ".ASM"){
  Filename = Filename.SubString(1, Filename.Length()-4);
 }else{
  if(!CopyFile((Path + Filename).c_str(),
             (Path + Filename + ".asm").c_str(), true)){
   Application->MessageBox("Could not copy file to have a \".asm\" extention",
                           "Error",
                           MB_ICONERROR);
   return;
```

```
  }
 }
 File.FileName = Path + Filename + ".asm";
 if(!File.Open(JFile::Read)){
  File.ShowLastError();
  return;
 }
 if(File.Size){
  char* c = new char[File.Size+1];
  c[File.Size] = 0;
  bool b;
  if(!File.ReadBuffer(c, File.Size, b)){
   File.ShowLastError();
   File.Close();
   delete[] c;
   return;
  }
  Asm->Lines->Text = c;
  delete[] c;
 }else{
  Asm->Lines->Clear();
 }
 File.Close();
 Saved = true;
 Caption = "Programmer - " + Filename;
}
//---------------------------------------------------------------------------

void __fastcall TProg::Save1Click(TObject *Sender)
{
 if(!Path.Length() || !Filename.Length()){
  Saveas1Click(this);
  return;
 }
 if(Saved) return;
 File.FileName = Path + Filename + ".asm";
 if(!File.Open(JFile::Create)){
  File.ShowLastError();
  return;
 }
 if(Asm->Lines->Text.Length()){
  bool b;
  if(!File.WriteBuffer(Asm->Lines->Text.c_str(),
                       Asm->Lines->Text.Length(), b)){
   File.ShowLastError();
   File.Close();
   return;
  }
 }
 File.Close();
 Saved = true;
 Caption = "Programmer - " + Filename;
}
//---------------------------------------------------------------------------

void __fastcall TProg::Saveas1Click(TObject *Sender)
{
 if(!SaveDialog->Execute()) return;
 int j;
 Filename = SaveDialog->FileName;
 for(j = Filename.Length(); Filename[j] != '\\'; j--);
 Path     = Filename.SubString(1, j);
 Filename = Filename.SubString(j+1, Filename.Length()-j);
 if(Filename.SubString(Filename.Length()-3, 4).UpperCase() == ".ASM"){
  Filename = Filename.SubString(1, Filename.Length()-4);
```

Pectora roborant cultus recti

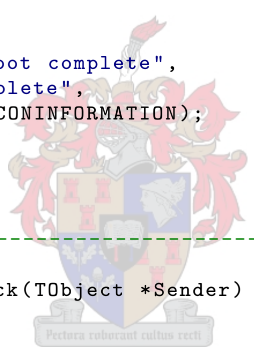
```
 }
 Saved = false;
 Save1Click(this);
}
//---------------------------------------------------------------------------

void __fastcall TProg::Exit1Click(TObject *Sender)
{
 if(!Saved){
  switch(Application->MessageBox("Would you like to save first",
                                 "Not saved",
                                 MB_ICONQUESTION | MB_YESNOCANCEL)){
   case IDYES:
    Save1Click(this);
   case IDNO:
    break;
   default:
    return;
  }
 }
 Application->Terminate();
}
//---------------------------------------------------------------------------

void __fastcall TProg::Cut1Click(TObject *Sender)
{
 Clipboard()->AsText = Asm->SelText;
 Asm->SelText = "";
}
//---------------------------------------------------------------------------

void __fastcall TProg::Copy1Click(TObject *Sender)
{
 Clipboard()->AsText = Asm->SelText;
}
//---------------------------------------------------------------------------

void __fastcall TProg::Paste1Click(TObject *Sender)
{
 Asm->SelText = Clipboard()->AsText;
}
//---------------------------------------------------------------------------

void __fastcall TProg::Assemble1Click(TObject *Sender)
{
 Success = false;
 Save1Click(this);
 union{
  struct{
   unsigned __int8 lsb;
   unsigned __int8 msb;
  };
  unsigned __int16 i;
 };
 int Size;
 bool b;
 AnsiString s;
 int j;
 char* Code = Assembler.Assemble(Asm->Lines->Text.c_str(), Size);
 if(!Code){
  Asm->SelStart  = Size;
  Asm->SelLength = 0;
  return;
 }
 File.FileName = Path + Filename + ".bin";
```

```
 if(!File.Open(JFile::Create)){
  File.ShowLastError();
  delete[] Code;
  return;
 }
 if(!File.WriteBuffer(Code, Size, b)){
  File.ShowLastError();
  delete[] Code;
  File.Close();
  return;
 }
 File.Close();
 File.FileName = Path + Filename + ".hex";
 if(!File.Open(JFile::Create)){
  File.ShowLastError();
  delete[] Code;
  return;
 }
 for(j = 0; j < Size; j++){
  s  = ":01";          //One byte in data
  s += IntToHex(j, 4); //Starting address of data
  s += "00";           //Data type
  s += IntToHex((unsigned __int8)Code[j]           , 2); //The actual data
  i = j;
  s += IntToHex((unsigned __int8)(~(lsb+msb+1+Code[j])+1), 2); //Checksum
  s += "\r\n";
  if(!File.WriteBuffer(s.c_str(), 15, b)){
   File.ShowLastError();
   delete[] Code;
   File.Close();
   return;
  }
 }
 if(!File.WriteBuffer(":00000001FF", 11, b)){
  File.ShowLastError();
  delete[] Code;
  File.Close();
  return;
 }
 File.Close();
 delete[] Code;
 if(Messages){
  Application->MessageBox("Assembly complete",
                          "Complete",
                          MB_ICONINFORMATION);
 }
 Success = true;
}
//---------------------------------------------------------------------------

void __fastcall TProg::Program1Click(TObject *Sender)
{
 Success = false;
 Save1Click(this);
 union{
  struct{
   unsigned __int8 lsb;
   unsigned __int8 msb;
  };
  unsigned __int16 i;
 };
 char* Data;
 char  Ack;
 bool  b;
 int j;
```

```
int l;
if(!Comm.Open("COM1", CBR_115200, 1000)){
 File.ShowLastError();
 return;
}
File.FileName = Path + Filename + ".bin";
if(!File.Open(JFile::Read)){
 File.ShowLastError();
 Comm.Close();
 return;
}
l = File.Size;
if(l > 0x4000){
 Application->MessageBox("File too large",
                        "Error",
                        MB_ICONERROR);
 File.Close();
 Comm.Close();
 return;
}
Data = new char[l];
if(!File.ReadBuffer(Data, l, b)){
 File.ShowLastError();
 delete[] Data;
 File.Close();
 Comm.Close();
 return;
}
File.Close();
for(j = 0; j < l; j++){
 i = j;
 if(!Comm.Write(0x01)){ //Command for Write EEPROM
  delete[] Data;
  Comm.Close();
  return;
 }
 if(!Comm.Write(lsb)){ //lsb of Address
  delete[] Data;
  Comm.Close();
  return;
 }
 if(!Comm.Write(msb)){ //msb of Address
  delete[] Data;
  Comm.Close();
  return;
 }
 if(!Comm.Write(Data[j])){ //Data to be written
  delete[] Data;
  Comm.Close();
  return;
 }
 if(!Comm.Read(Ack)){
  delete[] Data;
  Comm.Close();
  return;
 }
}
delete[] Data;
Comm.Close();
if(Messages){
 Application->MessageBox("Programming complete",
                        "Complete",
                        MB_ICONINFORMATION);
}
Success = true;
```

```
}
//---------------------------------------------------------------------------

void __fastcall TProg::Verify1Click(TObject *Sender)
{
 Success = false;
 Save1Click(this);
 union{
  struct{
   unsigned __int8 lsb;
   unsigned __int8 msb;
  };
  unsigned __int16 i;
 };
 char* Data;
 char  Ack;
 bool  b;
 int j;
 int l;
 if(!Comm.Open("COM1", CBR_115200, 1000)){
  File.ShowLastError();
  return;
 }
 File.FileName = Path + Filename + ".bin";
 if(!File.Open(JFile::Read)){
  File.ShowLastError();
  Comm.Close();
  return;
 }
 l = File.Size;
 Data = new char[l];
 if(!File.ReadBuffer(Data, l, b)){
  File.ShowLastError();
  delete[] Data;
  File.Close();
  Comm.Close();
  return;
 }
 File.Close();
 for(j = 0; j < l; j++){
  i = j;
  if(!Comm.Write(0x02)){ //Command for Read EEPROM
   delete[] Data;
   Comm.Close();
   return;
  }
  if(!Comm.Write(lsb)){ //lsb of Address
   delete[] Data;
   Comm.Close();
   return;
  }
  if(!Comm.Write(msb)){ //msb of Address
   delete[] Data;
   Comm.Close();
   return;
  }
  if(!Comm.Read(Ack)){
   delete[] Data;
   Comm.Close();
   return;
  }
  if(Ack != Data[j]){
   Application->MessageBox("Veryfication failed",
                          "Error",
                          MB_ICONERROR);
```

```
    delete[] Data;
    Comm.Close();
    return;
   }
  }
  delete[] Data;
  Comm.Close();
  if(Messages){
   Application->MessageBox("Veryfication complete",
                           "Complete",
                           MB_ICONINFORMATION);
  }
  Success = true;
}
//---------------------------------------------------------------------------

void __fastcall TProg::Reboot1Click(TObject *Sender)
{
  Success = false;
  if(!Comm.Open("COM1", CBR_115200, 1000)){
   File.ShowLastError();
   return;
  }
  if(!Comm.Write(0x03)){ //Command for reboot
   Comm.Close();
   return;
  }
  Comm.Close();
  if(Messages){
   Application->MessageBox("Reboot complete",
                           "Complete",
                           MB_ICONINFORMATION);
  }
  Success = true;

}
//---------------------------------------------------------------------------

void __fastcall TProg::All1Click(TObject *Sender)
{
  Messages = false;
  Assemble1Click(this);
  if(!Success){
   Messages = true;
   return;
  }
  Program1Click(this);
  if(!Success){
   Messages = true;
   return;
  }
  Verify1Click(this);
  if(!Success){
   Messages = true;
   return;
  }
  Reboot1Click(this);
  Messages = true;
  Application->MessageBox("All operations complete",
                          "Complete",
                          MB_ICONINFORMATION);
}
//---------------------------------------------------------------------------

void __fastcall TProg::AsmChange(TObject *Sender)
```

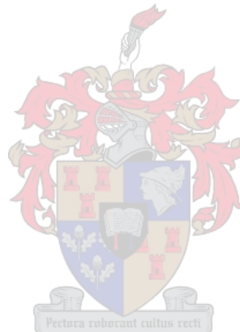

Pectora roborant cultus recti

```
{
 Saved = false;
}
//---------------------------------------------------------------------------

void __fastcall TProg::FormClose(TObject *Sender, TCloseAction &Action)
{
 Action = caNone;
 Exit1Click(this);
}
//---------------------------------------------------------------------------
```

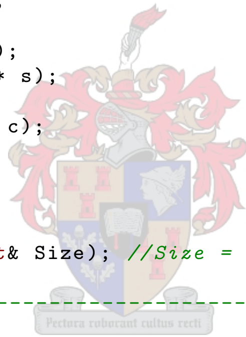
G.5 Assembler

G.5.1 Header

```
//---------------------------------------------------------------------------
#ifndef AssemblerH
#define AssemblerH
//---------------------------------------------------------------------------
#include <Forms.hpp>
#include <system.hpp>
//---------------------------------------------------------------------------

class JAssembler{
 private:
  enum{
   Label    = 0x81,
   LabelRef = 0x82,
   Number   = 0x83,
   Sref  = 0x84,
   Sval  = 0x85,
   Sline = 0x86,
   Send  = 0x87,
   Sc    = 0x80,
   Sz    = 0x40,
   Sc_z  = 0xC0, //"c&z"
   Sadc  = 0x00,
   Sadd  = 0x02,
   Sand  = 0x04,
   Sdiv  = 0x06,
   Smul  = 0x08,
   Sneg  = 0x0A,
   Snot  = 0x0B,
   Sor   = 0x0C,
   Ssbb  = 0x0E,
   Ssub  = 0x10,
   Sxor  = 0x12,
   Srcl  = 0x14,
   Srcr  = 0x15,
   Srol  = 0x16,
   Sror  = 0x17,
   Sshl  = 0x18,
   Sshr  = 0x19,
   Scc   = 0x1A,
   Scz   = 0x1B,
   Snc   = 0x1C,
   Snz   = 0x1D,
   Ssc   = 0x1E,
   Ssz   = 0x1F,
   Sswp  = 0x20,
   Spop  = 0x28,
   Sret  = 0x29,
```



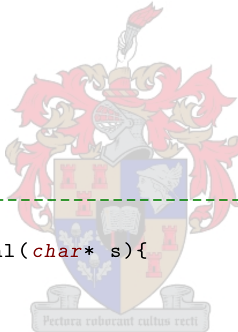
```
   Scall = 0x2A,
   Sjmp  = 0x2C,
   Sld   = 0x2E,
   Slds  = 0x30,
   Sldi  = 0x38,
   Sldpr = 0x39,
   Sstpr = 0x3A,
   Sst   = 0x3C
  } Sym;
  struct JUMP{
   int Address;
   int Character;
   AnsiString Label;
   JUMP* Next;
  };
  JUMP* Labels;
  JUMP* Jumps;
  char* Data;
  unsigned __int8* Code;
  int    DataCount;
  int    CodeCount;
  int    CodeSize;
  unsigned __int16 I;
  AnsiString S;
  bool ErrorSet;
  void Error(char* s);
  void  Larger(void);
  void  Put(unsigned __int8 c);
  void  Get(void);
  unsigned __int16 Val(char* s);
  unsigned __int16 ValHex(char* s);
  void  Line(void);
  void  Opcode(unsigned __int8 c);
  void  SolveJumps(void);
  void  Cleanup(void);
 public:
        JAssembler(void);
  char* Assemble(char* Asm, int& Size); //Size = 0 => Error
};
//----------------------------------------------------------------------------
#endif
```



G.5.2 Body

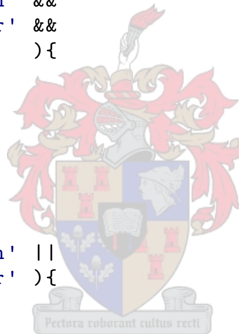
```
/*Do error handling - quit on error*/

#include "Assembler.h"
#include <stdlib.h>
//----------------------------------------------------------------------------

JAssembler::JAssembler(void){
 Jumps = 0;
 Labels = 0;
}
//----------------------------------------------------------------------------

void JAssembler::Larger(void){
 unsigned __int8* c = new unsigned __int8[CodeSize << 1];
 for(int j = 0; j < CodeSize; j++){
  c[j] = Code[j];
 }
 CodeSize <<= 1;
 delete[] Code;
 Code = c;
```

```
}
//---------------------------------------------------------------------------

void JAssembler::Put(unsigned __int8 c){
 if(ErrorSet) return;
 if(CodeCount == CodeSize) Larger();
 Code[CodeCount++] = c;
}
//---------------------------------------------------------------------------

void JAssembler::Error(char* s){
 if(ErrorSet) return;
 ErrorSet = true;
 Application->MessageBox(s, "Error", MB_ICONERROR);
}
//---------------------------------------------------------------------------

char* JAssembler::Assemble(char* Asm, int& Size){
 ErrorSet   = false;
 Data       = Asm;
 DataCount = 0;
 CodeCount = 0;
 Code       = new unsigned __int8[2];
 CodeSize   = 2;
 Get();
 Line();
 Size = CodeCount;
 if(ErrorSet){
  Size = DataCount;
  Cleanup();
  delete[] Code;
  Code = 0;
 }
 return (char*)Code;
}
//---------------------------------------------------------------------------
```




```
unsigned __int16 JAssembler::Val(char* s){
 if(ErrorSet) return 0;
 unsigned __int16 i = 0;
 int j = 0;
 while(s[j]){
  if(s[j] >= '0' && s[j] <= '9'){
   i *= (unsigned __int16)10;
   i += (unsigned __int16)s[j] - (unsigned __int16)'0';
  }else{
   Error("Illigal character in number");
   return 0;
  }
  j++;
 }
 return i;
}
//---------------------------------------------------------------------------

unsigned __int16 JAssembler::ValHex(char* s){
 if(ErrorSet) return 0;
 unsigned __int16 i = 0;
 int j = 0;
 while(s[j]){
  i *= (unsigned __int16)16;
  if(s[j] >= '0' && s[j] <= '9'){
   i += (unsigned __int16)s[j] - (unsigned __int16)'0';
  }else if(s[j] >= 'A' && s[j] <= 'F'){
   i += (unsigned __int16)s[j] - (unsigned __int16)'A';
```

```
   i += (unsigned __int16)10;
  }else if(s[j] >= 'a' && s[j] <= 'f'){
   i += (unsigned __int16)s[j] - (unsigned __int16)'a';
   i += (unsigned __int16)10;
  }else{
   Error("Illigal character in number");
   return 0;
  }
  j++;
 }
 return i;
}
//---------------------------------------------------------------------------

void JAssembler::Get(void){
 if(ErrorSet) return;
 while(Data[DataCount] == ' ') DataCount++;
 AnsiString s = "";
 while(Data[DataCount] != ' '  &&
       Data[DataCount] != ';'  &&
       Data[DataCount] != '\n' &&
       Data[DataCount] != '\r' &&
       Data[DataCount] !=  0   ){
  s += Data[DataCount++];
 }
 if(s == ""){
  if(Data[DataCount] == ';'){
   while(Data[DataCount] != '\n' &&
         Data[DataCount] != '\r' &&
         Data[DataCount] !=  0   ){
    DataCount++;
   }
  }
  if(Data[DataCount] == 0){
   Sym = Send;
  }else{
   Sym = Sline;
   while(Data[DataCount] == '\n' ||
         Data[DataCount] == '\r' ){
    DataCount++;
   }
  }
 }else if(s == "adc"){
  Sym = Sadc;
 }else if(s == "add"){
  Sym = Sadd;
 }else if(s == "and"){
  Sym = Sand;
 }else if(s == "div"){
  Sym = Sdiv;
 }else if(s == "mul"){
  Sym = Smul;
 }else if(s == "neg"){
  Sym = Sneg;
 }else if(s == "not"){
  Sym = Snot;
 }else if(s == "or"){
  Sym = Sor;
 }else if(s == "sbb"){
  Sym = Ssbb;
 }else if(s == "sub"){
  Sym = Ssub;
 }else if(s == "xor"){
  Sym = Sxor;
 }else if(s == "rcl"){
```



```
  Sym = Srcl;
 }else if(s == "rcr"){
  Sym = Srcr;
 }else if(s == "rol"){
  Sym = Srol;
 }else if(s == "ror"){
  Sym = Sror;
 }else if(s == "shl"){
  Sym = Sshl;
 }else if(s == "shr"){
  Sym = Sshr;
 }else if(s == "cc"){
  Sym = Scc;
 }else if(s == "cz"){
  Sym = Scz;
 }else if(s == "nc"){
  Sym = Snc;
 }else if(s == "nz"){
  Sym = Snz;
 }else if(s == "sc"){
  Sym = Ssc;
 }else if(s == "sz"){
  Sym = Ssz;
 }else if(s == "swp"){
  Sym = Sswp;
 }else if(s == "pop"){
  Sym = Spop;
 }else if(s == "ret"){
  Sym = Sret;
 }else if(s == "call"){
  Sym = Scall;
 }else if(s == "jmp"){
  Sym = Sjmp;
 }else if(s == "ld"){
  Sym = Sld;
 }else if(s == "lds"){
  Sym = Slds;
 }else if(s == "ldi"){
  Sym = Sldi;
 }else if(s == "ldpr"){
  Sym = Sldpr;
 }else if(s == "stpr"){
  Sym = Sstpr;
 }else if(s == "st"){
  Sym = Sst;
 }else if(s == "c"){
  Sym = Sc;
 }else if(s == "z"){
  Sym = Sz;
 }else if(s == "z&c"){
  Sym = Sc_z;
 }else if(s == "c&z"){
  Sym = Sc_z;
 }else if(s == "ref"){
  Sym = Sref;
 }else if(s == "val"){
  Sym = Sval;
 }else if(s[s.Length()] == ':'){ //Label
  Sym = Label;
  S   = s.SubString(1, s.Length()-1);
 }else if(s.SubString(1, 2).UpperCase() == "0X"){ //Hexadecimal
  Sym = Number;
  I   = ValHex(s.SubString(3, s.Length()-2).c_str());
 }else if(s[1] >= '0' && s[1] <= '9'){ //Decimal
  Sym = Number;
```

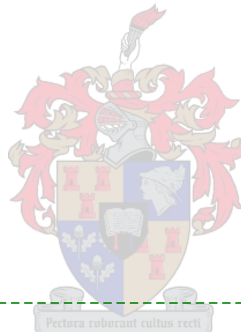
```
  I   = Val(s.c_str());
 }else{ //Label reference
  Sym = LabelRef;
  S   = s;
 }
}
//---------------------------------------------------------------------------

void JAssembler::SolveJumps(void){
 if(ErrorSet) return;
 union{
  struct{
   unsigned __int8 lsb;
   unsigned __int8 msb;
  };
  unsigned __int16 i;
 };
 JUMP* Label;
 JUMP* Jump;
 JUMP* Temp;
 int j;
 Jump = Jumps;
 while(Jump){
  Label = Labels;
  while(Label && Label->Label != Jump->Label) Label = Label->Next;
  if(!Label){
   DataCount = Jump->Character;
   Error("Label not found");
   return;
  }
  if(abs(Label->Address - Jump->Address) <= 0x7F){
   Temp = Labels;
   while(Temp->Address > Jump->Address){
    Temp->Address--;
    Temp = Temp->Next;
   }
   Temp = Jumps;
   while(Temp->Address > Jump->Address){
    Temp->Address--;
    Temp = Temp->Next;
   }
   j = Jump->Address+2;
   while(j < CodeCount-1){
    Code[j] = Code[j+1];
    j++;
   }
   CodeCount--;
  }else{
   Code[Jump->Address]++;
  }
  Jump = Jump->Next;
 }
 while(Jumps){
  Label = Labels;
  while(Label && Label->Label != Jumps->Label) Label = Label->Next;
  if(Code[Jumps->Address] & 0x01){
   i = Label->Address;
   Code[Jumps->Address+1] = lsb;
   Code[Jumps->Address+2] = msb;
  }else{
   Code[Jumps->Address+1] = Label->Address - Jumps->Address;
  }
  Temp = Jumps;
  Jumps = Jumps->Next;
  delete Temp;
```

```
 }
 while(Labels){
  Temp = Labels;
  Labels = Labels->Next;
  delete Temp;
 }
}
//---------------------------------------------------------------------------

void JAssembler::Line(void){
 JUMP* Temp;
 unsigned __int8 c;
 while(!ErrorSet && Sym != Send){
  c = 0x00;
  while(Sym == Sline) Get();
  if(Sym == Label){
   Temp = Labels;
   while(Temp){
    if(Temp->Label == S){
     Error("Duplicate lable");
     return;
    }
    Temp = Temp->Next;
   }
   Temp = new JUMP;
   Temp->Address = CodeCount;
   Temp->Label = S;
   Temp->Next = Labels;
   Labels = Temp;
   Get();
  }
  while(Sym == Sline) Get();
  if(Sym == Sc   ||
     Sym == Sz   ||
     Sym == Sc_z){
   c = Sym;
   Get();
  }
  Opcode(c);
 }
 SolveJumps();
}
//---------------------------------------------------------------------------

void JAssembler::Opcode(unsigned __int8 c){
 if(ErrorSet) return;
 union{
  struct{
   unsigned __int8 lsb;
   unsigned __int8 msb;
  };
  unsigned __int16 i;
 };
 JUMP* Temp;
 switch(Sym){
  case Sadc:
  case Sadd:
  case Sand:
  case Sdiv:
  case Smul:
  case Sor:
  case Ssbb:
  case Ssub:
  case Sxor:
   c += Sym;
```

```
 Get();
 if(Sym == Spop){
  Put(c+(unsigned __int8)1);
  Get();
 }else{
  Put(c);
 }
 break;
case Sneg:
case Snot:
case Srcl:
case Srcr:
case Srol:
case Sror:
case Sshl:
case Sshr:
case Scc:
case Scz:
case Snc:
case Snz:
case Ssc:
case Ssz:
case Spop:
case Sret:
 Put(c+(unsigned __int8)Sym);
 Get();
 break;
case Sswp:
case Slds:
 c += Sym;
 Get();
 if(Sym == Number && I <= 7){
  Put(c+(unsigned __int8)I);
 }else{
  Error("Expected number");
  return;
 }
 Get();
 break;
case Scall:
case Sjmp:
 Put(c+(unsigned __int8)Sym);
 Get();
 if(Sym == LabelRef){
  Temp = new JUMP;
  Temp->Address = CodeCount-1;
  Temp->Label = S;
  Temp->Character = DataCount;
  Temp->Next = Jumps;
  Jumps = Temp;
  Put(0x00);
  Put(0x00);
 }else{
  Error("Expected label");
  return;
 }
 Get();
 break;
case Sld:
 c += Sym;
 Get();
 if(Sym == Sref){
  c++;
  Get();
 }else if(Sym == Sval){
```

```
    Get();
   }
   Put(c);
   if(Sym == Number){
    i = I;
    Put(lsb);
    Put(msb);
   }else{
    Error("Expected number");
    return;
   }
   Get();
   break;
  case Sldi:
   Put(c+(unsigned __int8)Sym);
   Get();
   if(Sym == Number){
    Put(I);
   }else{
    Error("Expected number");
    return;
   }
   Get();
   break;
  case Sldpr:
   Put(c+(unsigned __int8)Sym);
   Get();
   if(Sym == Number){
    i = I;
    Put(lsb);
    Put(msb);
   }else{
    Error("Expected number");
    return;
   }
   Get();
   break;
  case Sstpr:
   c += Sym;
   Get();
   if(Sym == Spop){
    Put(c+(unsigned __int8)1);
    Get();
   }else{
    Put(c);
   }
   if(Sym == Number){
    i = I;
    Put(lsb);
    Put(msb);
   }else{
    Error("Expected number");
    return;
   }
   Get();
   break;
  case Sst:
   c += Sym;
   Get();
   if(Sym == Spop){
    c += (unsigned __int8)2;
    Get();
   }
   if(Sym == Sref){
    c++;
```

```
    Get();
   }else if(Sym == Sval){
    Get();
   }
   Put(c);
   if(Sym == Number){
    i = I;
    Put(lsb);
    Put(msb);
   }else{
    Error("Expected number");
    return;
   }
   Get();
   break;
  case Label:
  case Sline:
  case Send:
   break;
  default:
   Error("Expected opcode");
   break;
 }
}
//---------------------------------------------------------------------------

void JAssembler::Cleanup(void){
 JUMP* Temp;
 while(Jumps){
  Temp = Jumps;
  Jumps = Jumps->Next;
  delete Temp;
 }
 while(Labels){
  Temp = Labels;
  Labels = Labels->Next;
  delete Temp;
 }
}
//---------------------------------------------------------------------------
```



G.6 JComm

Components 'JComm' and 'JFile' were developed for prior projects and is not included in the development time.

G.6.1 Header

```
//---------------------------------------------------------------------------
#ifndef JCommH
#define JCommH
//---------------------------------------------------------------------------
#include <system.hpp>
//---------------------------------------------------------------------------

class JComm{
 private:
  HANDLE Handle;
 public:
       JComm(void);
       ~JComm(void);
```

```
  bool Open(char* Port, int Baud, int TimeOut);
  bool Read(char& c);
  bool Write(char c);
  void Close(void);
};
//---------------------------------------------------------------------------
#endif
```

G.6.2 Body

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "JComm.h"

//---------------------------------------------------------------------------
#pragma package(smart_init)

JComm::JComm(void){
 Handle = 0;
}
//---------------------------------------------------------------------------

JComm::~JComm(void){
 Close();
}
//---------------------------------------------------------------------------

bool JComm::Open(char* Port, int Baud, int TimeOut){
 DCB          dcb;
 COMMTIMEOUTS cto;
 Handle = CreateFile(Port,
                     GENERIC_READ | GENERIC_WRITE,
                     0,    /* comm devices must be opened w/exclusive-access */
                     NULL, /* no security attrs */
                     OPEN_EXISTING, /* comm devices must use OPEN_EXISTING */
                     0,    /* not overlapped I/O */
                     NULL);  /* hTemplate must be NULL for comm devices */
 if(Handle == INVALID_HANDLE_VALUE) return false;
 if(!GetCommState(Handle, &dcb)){
  CloseHandle(Handle);
  return false;
 }
 dcb.BaudRate = Baud;
 dcb.ByteSize = 8;
 dcb.Parity = NOPARITY;
 dcb.StopBits = ONESTOPBIT;
 if(!SetCommState(Handle, &dcb)){
  CloseHandle(Handle);
  return false;
 }
 cto.ReadIntervalTimeout = TimeOut;
 cto.ReadTotalTimeoutMultiplier = TimeOut;
 cto.ReadTotalTimeoutConstant = 0;
 cto.WriteTotalTimeoutMultiplier = TimeOut;
 cto.WriteTotalTimeoutConstant = 0;
 if(!SetCommTimeouts(Handle, &cto)){
  CloseHandle(Handle);
  return false;
 }
 return true;
}
//---------------------------------------------------------------------------
```

```
bool JComm::Read(char& c){
 unsigned long l;
 ReadFile(Handle, &c, 1, &l, 0);
 if(l == 0){
  return false;
 }
 return true;
}
//---------------------------------------------------------------------------

bool JComm::Write(char c){
 unsigned long l;
 WriteFile(Handle, &c, 1, &l, 0);
 if(l == 0){
  return false;
 }
 return true;
}
//---------------------------------------------------------------------------

void JComm::Close(void){
 if(Handle) CloseHandle(Handle);
 Handle = 0;
}
//---------------------------------------------------------------------------
```

G.7 JFile

Components 'JComm' and 'JFile' were developed for prior projects and is not included in the development time.

G.7.1 Header

```
//---------------------------------------------------------------------------
#ifndef JFileH
#define JFileH
//---------------------------------------------------------------------------
class JFile{
 public:
  enum ACCESS{
   Read   = 0,
   Write  = 1,
   Create = 2
  };
 private:
  #include "winbase.h"
//---------------------------------------------------------------------------
  ACCESS     CurrentAccess;
  void*      Handle;
  char*      Filename;
  _OFSTRUCT* OpenBuffer;
//---------------------------------------------------------------------------
  bool LInput(AnsiString& s);
  bool LPrint(AnsiString s);
  void Copy(char* to, char* from);
//---------------------------------------------------------------------------
  AnsiString    GetFileName(void);
  void          SetFileName(AnsiString Value);
  unsigned long GetSize(void);
  unsigned long GetPosition(void);
```

```
  void           SetPosition(unsigned long Value);
  bool           GetBusy(void);
 public:
  struct ERR{}Err;
                 JFile         (void);
                 ~JFile        (void);
  void           ShowLastError(void);
  bool           Open          (ACCESS Access);
  AnsiString     ReadLine      (bool& Value);
  bool           WriteLine     (AnsiString Value);
  unsigned long ReadBuffer     (char* Buffer, unsigned long MustRead,
                                bool& Value);
  unsigned long WriteBuffer    (char* Buffer, unsigned long MustWrite,
                                bool& Value);
  void           Close         (void);
//---------------------------------------------------------------------------
  __property AnsiString FileName     = {read=GetFileName,
                                        write=SetFileName};
  __property unsigned long Size      = {read=GetSize};
  __property unsigned long Position  = {read=GetPosition,
                                        write=SetPosition};
  __property bool          Busy      = {read=GetBusy};
  __fastcall JFile(TComponent* Owner);
};
//---------------------------------------------------------------------------
#endif
```

G.7.2 Body

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "JFile.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------

JFile::JFile(){
 try{
  Filename = new char[0xFFFF];
 }catch(...){
  Application->MessageBox ("Out of memory.", "Error", MB_ICONERROR);
  Application->Terminate();
 }
 Filename[0]    = (char)0;
 Handle         = 0;
 OpenBuffer     = new _OFSTRUCT;
 CurrentAccess = Read;
}
//---------------------------------------------------------------------------

bool JFile::LInput(AnsiString& s){
 unsigned long X;
 char          x[2];
 s = "";
 if (!ReadFile(Handle, x, 1, &X, 0)) return false;
 if (X <= 0) return false;
 x[1] = x[0];
 while(true){
  if (!ReadFile(Handle, x, 1, &X, 0)) return true;
  if (X <= 0){
   s += x[1];
   return true;
  }
```

```
  if (((int)x[1] == 13 && (int)x[0] == 10) ||  // return + line feed
       (int)x[1] == 10 ||                      // line feed
       (int)x[1] == 26) return true;           // end of file
  s += x[1];
  x[1] = x[0];
 }
}
//---------------------------------------------------------------------------

bool JFile::LPrint(AnsiString s){
 unsigned long X;
 AnsiString    x;
 x = s + "\r\n";
 return WriteFile(Handle, x.c_str(), x.Length(), &X, 0);
}
//---------------------------------------------------------------------------

JFile::~JFile(void){
 Close();
 delete[] Filename;
}
//---------------------------------------------------------------------------

void JFile::Close(void){
 if (Handle){
  CloseHandle(Handle);
  Handle = 0;
 }
}
//---------------------------------------------------------------------------

bool JFile::Open(ACCESS Access){
 Close();
 switch (Access){
  case Read:   Handle = (void*)OpenFile(Filename, OpenBuffer,
                                        OF_READ);
               if ((int)Handle < 1){
                Handle = 0;
                return false;
               }
               CurrentAccess = Read;
               break;
  case Write:  Handle = (void*)OpenFile(Filename, OpenBuffer,
                                        OF_WRITE);
               if ((int)Handle < 1){
                Handle = 0;
                return false;
               }
               unsigned long x;
               SetFilePointer(Handle, GetFileSize(Handle, &x), 0,
                              FILE_BEGIN);
               CurrentAccess = Write;
               break;
  case Create: Handle = (void*)OpenFile(Filename, OpenBuffer,
                                        OF_WRITE + OF_CREATE);
               if ((int)Handle < 1){
                Handle = 0;
                return false;
               }
               CurrentAccess = Create;
               break;
 }
 return true;
}
//---------------------------------------------------------------------------
```

```
void JFile::ShowLastError(void){
 AnsiString A;
 char       Buffer[50];
 FormatMessage((unsigned long)FORMAT_MESSAGE_FROM_SYSTEM,
               (const void*)FORMAT_MESSAGE_FROM_HMODULE,
               (unsigned long)GetLastError(),
               (unsigned long)0,
               (char*)Buffer,
               (unsigned long)50,
               (const void*)"");
 A = IntToStr(GetLastError()) + ": " + Buffer;
 if (Application->MessageBox(A.c_str(), "Error",
                              MB_ICONERROR + MB_OKCANCEL) == IDCANCEL){
  Application->Terminate();
 }
}
//---------------------------------------------------------------------------

void JFile::Copy(char* to, char* from){
 for (int j = 0; (from[j-1] != (char)0 || j == 0); j++){
  to[j] = from[j];
 }
}
//---------------------------------------------------------------------------

AnsiString JFile::GetFileName(void){
 AnsiString t;
 t = (AnsiString)Filename;
 return t;
}
//---------------------------------------------------------------------------

void JFile::SetFileName(AnsiString Value){
 Close();
 if(Value.Length() < 3){
  Filename[0] = (char)0;
  return;
 }
 if (Value[2] != ':' || Value[3] != '\\'){
  AnsiString t;
  t  = "Invalid path and filename: ";
  t += Value;
  t += ".";
  Application->MessageBox (t.c_str(), "Error", MB_ICONERROR);
  Filename[0] = (char)0;
  return;
 }
 Copy(Filename, Value.c_str());
}
//---------------------------------------------------------------------------

AnsiString JFile::ReadLine(bool& Value){
 if (!CurrentAccess){
  AnsiString Val;
  Value = LInput(Val);
  return Val;
 }else{
  Value = false;
  return "";
 }
}
//---------------------------------------------------------------------------

bool JFile::WriteLine(AnsiString Value){
```

```
 if (CurrentAccess){
  if (!LPrint(Value)) return false;
 }
 return true;
}
//---------------------------------------------------------------------------

unsigned long JFile::ReadBuffer(char* Buffer, unsigned long MustRead,
                                bool& Value){
 unsigned long X = 0;
 if (!CurrentAccess){
  Value = ReadFile(Handle, Buffer, MustRead, &X, 0);
 }else{
  Value = false;
 }
 return X;
}
//---------------------------------------------------------------------------

unsigned long JFile::WriteBuffer(char* Buffer, unsigned long MustWrite,
                                 bool& Value){
 unsigned long X = 0;
 if (CurrentAccess){
  Value = WriteFile(Handle, Buffer, MustWrite, &X, 0);
 }else{
  Value = false;
 }
 return X;
}
//---------------------------------------------------------------------------

unsigned long JFile::GetSize(void){
 unsigned long x;
 if(Handle) return GetFileSize(Handle, &x);
 return 0;
}
//---------------------------------------------------------------------------

unsigned long JFile::GetPosition(void){
 if(Handle) return SetFilePointer(Handle, 0, 0, FILE_CURRENT);
 return 0;
}
//---------------------------------------------------------------------------

void JFile::SetPosition(unsigned long Value){
 if(Handle) SetFilePointer(Handle, Value, 0, FILE_BEGIN);
}
//---------------------------------------------------------------------------

bool JFile::GetBusy(void){
 return (bool)Handle;
}
//---------------------------------------------------------------------------
```

Appendix H

Simulation Source

H.1 Introduction

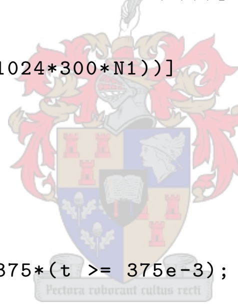
This appendix presents the simulation code used for the converter (MATLAB) and probe (C++) simulations.

H.2 Converter

H.2.1 Stage 1

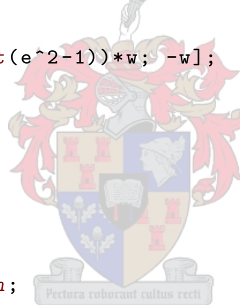
```
%LC low-pass with R // R-L load, driven by switching between Vbus and Gnd

close all;
clear all;
format compact;
format short;
clc;

L1 = 3e-3;
RL = 50e-3;
C1 = 470e-6;
RC = 10e-3;
R1 = -22.5;
L2 = 60e-3;
R2 = 11.25;
Vbus = 535;
freq1 = 500;
freq2 = 5e3;

A = [(-L1-R1*RC*C1)/(L1*C1*(R1+RC)) (R1*L1-R1*RC*RL*C1)/(L1*C1*(R1+RC));
     -1/L1                            -RL/L1                              ];
B = [Vbus*R1*RC/(L1*(R1+RC));
     Vbus/L1                 ];
C = [1 0];
D = 0;

e = 1/sqrt(2);
w = 2*pi*freq1;
P = [-e*w+sqrt((e^2-1)*w^2) -e*w-sqrt((e^2-1)*w^2)];
P = [-2*pi*freq1 -2*pi*freq2]; %more immune to bus-disturbance
```

```
K = -acker(A, B, P) %Included as a comparrison only

A = [(-R1*RC*L2*C1-R1*RC*L1*C1-L1*L2)/(L1*L2*C1*(R1+RC))
     (R1*L1-R1*RC*RL*C1)/(L1*C1*(R1+RC))
     (R1*R2*RC*C1-R1*L2)/(L2*C1*(R1+RC));
     -1/L1         -RL/L1      0           ;
     1/L2             0       -R2/L2      ];
B = [R1*RC*Vbus/(L1*(R1+RC));
     Vbus/L1                 ;
     0                       ];
C = [1 0 0];
Ci= [0 1 0];
D = 0;

P = [P -R2/L2];
K = acker(A, B, P);
K = -K
K = -K;
K(3) = 0;

N=[A B; C D]^-1*[0;0;0;1];
N=N(4)+K*N(1:3);

sys=ss(A-B*K, B*N, C, D);

K1 = -K
KV1 = ['    ' Binary(round(2^18/1024*300*K1(1)))]
KI1 = ['    ' Binary(round(2^18/1024*15 *K1(2)))]

N1 = N
N1 = ['    ' Binary(round(2^18/1024*300*N1))]

sys=ss(A-B*K, B*N, Ci, D);
sys=ss(A-B*K, B*N, C, D);

dt  = 10e-12*2^21;
t   = 0:dt:400e-3;
Q   = length(t);
Vref = 1000*t.*(t < 375e-3) + 375*(t >= 375e-3);
x   = [0; 0; 0];
y   = C*x;
for j = 1:Q-1
    dx = (A*x(:,j)+B*(-K*x(:,j) + N*Vref(j+1)))*dt;
    x(:,j+1) = x(:,j) + dx;
    y(j+1) = C*x(:,j+1);
end
figure; plot(t/1e-3, y);
figure; plot(t/1e-3, x(2,:));

Vref = 270;
x   = ((A-B*K/Vbus*560)^-1)*(-B*N*Vref/Vbus*560);
y   = C*x;
Q   = 5000;
t   = linspace(0, 4e-3, Q);
Vb  = 485*(t < 2e-3) + 560*(t >= 2e-3);
dt  = t(2) - t(1);
for j = 1:Q-1
    dx = (A*x(:,j)+B*(-K*x(:,j) + N*Vref)/Vbus*Vb(j+1))*dt;
    x(:,j+1) = x(:,j) + dx;
    y(j+1) = C*x(:,j+1);
end
figure; plot(t/1e-3, y);
figure; plot(t/1e-3, x(2,:));
```

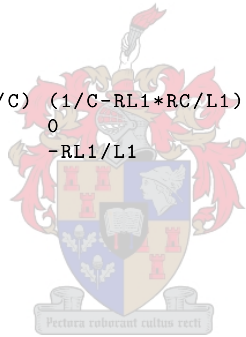


H.2.2 Stage 2

```
%LC low-pass with RL load, driven by input voltage

close all;
clear all;
format compact;
format short;
clc;

L1=200e-6;
RL1=50e-3;
C=470e-6;
RC=10e-3;
RL2=14.4;
L2=4;

A=[(-RC/L1-RC/L2) (RL2*RC/L2-1/C) (1/C-RL1*RC/L1);
   1/L2           -RL2/L2         0              ;
   -1/L1          0               -RL1/L1        ];
B=[RC/L1;
   0    ;
   1/L1 ];
C=[0 1 0 ];
D=0;

fast=11; % This is the 3 fast poles frequency
slow=1 ; % This is the closed loop frequency
e=1/2;
w=2*pi*fast;
P=[(-e+sqrt(e^2-1))*w; (-e-sqrt(e^2-1))*w; -w];
P_orig = eig(A)
P_moved = P

K=acker(A, B, P);

N=[A B; C D]^-1*[0;0;0;1];
N=N(4)+K*N(1:3);

sys=ss(A-B*K, B*N, C, D);return;

K2 = -K
KV2 = ['    ' Binary(round(2^18/240*240*K2(1)))]
KI2 = ['    ' Binary(round(2^18/240*20 *K2(2)))]
KI3 = ['    ' Binary(round(2^18/240*20 *K2(3)))]

N2 = N
N2 = ['    ' Binary(round(2^18/240*20*N))]

ki = 0;
dk = 1;
Ci = [0 1 0 0];
busy = 1;
while busy == 1
    ki = ki + dk;
    Ai = [A-B*K B*N;
          0 -ki 0 0];
    Bi = [0;
          0;
          0;
          ki];

    sys = ss(Ai, Bi, Ci, 0);
    p = eig(Ai);
    p = p(4);
```



```
    if dk == 1
        if p < -2*pi*slow
            dk = -0.1;
        end;
    elseif dk == -0.1
        if p > -2*pi*slow
            dk = 0.01;
        end;
    elseif dk == 0.01
        if p < -2*pi*slow
            dk = -0.001;
        end;
    elseif dk == -0.001
        if p > -2*pi*slow
            dk = 0.0001;
        end;
    elseif dk == 0.0001
        if p <= -2*pi*slow
            busy = 0;
        end;
    end;
end;
ki = ki
poles_int=eig(Ai)

L1=200e-6;
RL1=50e-3;
C=470e-6;
RC=10e-3;
RL2=14.4;
L2=15;
A=[(-RC/L1-RC/L2) (RL2*RC/L2-1/C) (1/C-RL1*RC/L1);
   1/L2             -RL2/L2         0              ;
   -1/L1            0               -RL1/L1        ];
B=[RC/L1;
   0    ;
   1/L1 ];
C=[0 1 0 ];
D=0;
Ai = [A-B*K B*N;
      0 -ki 0 0];
Bi = [0;
      0;
      0;
      ki];

Q = 2000;
% t = linspace(0, 0.25, Q); %For 4H
t = linspace(0, 0.6, Q); %For 15H
% t = linspace(0, 1, Q); %For 1H
ref = 1*(t >= 0);
x  = [0; 0; 0];
dt = t(2) - t(1);
for j = 1:Q-1
    dx = (A*x(:,j)+B*(-K*x(:,j) + N*ref(j+1)))*dt;
    x(:,j+1) = x(:,j) + dx;
end
Fig(t, C*x, 'Time [s]', 'i_2 [A]', 40, [0 0.6 0 1.2]);


Q  = 2000;
t  = linspace(0, 1.6, Q);
ref = 1*(t >= 0);
x  = [0; 0; 0; 0];
dt = t(2) - t(1);
```



```
for j = 1:Q-1
    dx = (Ai*x(:,j)+Bi*ref(j+1))*dt;
    x(:,j+1) = x(:,j) + dx;
end
Fig(t, Ci*x, 'Time [s]', 'i_2 [A]', 40, [0 1.6 0 1.2]);

P = 10000;
t = linspace(0, 15, P);
r = (5*t+1).*(t<=2.8) + 15.*(t>2.8 & t<=5) +
    (-5*t + 40).*(t>5 & t<=7.8) + 1.*(t>7.8);

dt  = t(2)-t(1);
I2  = 1;
V2  = I2*RL2;
I2  = I2;
I3  = I2;
u   = I2;
x   = [V2; I2; I3];
V3  = N*u - K*x;
e   = r(1) - C*x;
for j = 2:P
    x = [V2(j-1); I2(j-1); I3(j-1)];
    e(j) = r(j)-C*x;
    u(j) = u(j-1) + ki*e(j)*dt;
    if u(j) < 0; u(j) = 0; end; %Integrator internal limit
    V3(j) = N*u(j) - K*x;
    if V3(j) < -1; V3(j) = -1; end; %Buck converter property (diode)
    dx = (A*x + B*V3(j));
    dx = dx*dt;
    V2 (j) = V2 (j-1) + dx(1);
    if V2(j) < -1; V2(j) = -1; end; %Clamping diode (load)
    I2 (j) = I2 (j-1) + dx(2);
    I3 (j) = I3 (j-1) + dx(3);
    if I3(j) < 0; I3(j) = 0; end; %Buck converter (not synchronous buck)
end;
V1 = 1.25*V3;
V1 = V1.*(V1>10) + 10*(V1<=10);
D2 = V3./V1;

Fig(t, V2, 'Time [s]', 'v_2 [V]', 40, [0 15 -50 300]);
Fig(t, I2, 'Time [s]', 'i_2 [A]', 40, [0 15 -1 20]);
```

H.3 Probe

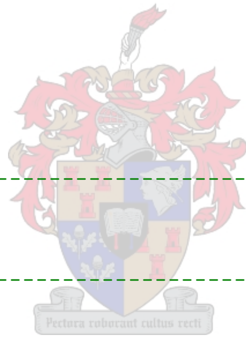
H.3.1 Header

```
//---------------------------------------------------------------------------
#ifndef FrmBlockH
#define FrmBlockH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "JFile.h"
//---------------------------------------------------------------------------
class TBlockEq : public TForm
{
__published:    // IDE-managed Components
  void __fastcall FormClick(TObject *Sender);
private:        // User declarations
 JFile File;
 class VECTOR{
```

```
  public:
   double x;
   double y;
   double z;

   VECTOR operator+ (VECTOR v);
   VECTOR operator+ (double d);
   VECTOR operator- (VECTOR v);
   VECTOR operator- (double d);
   VECTOR operator* (VECTOR v);
   VECTOR operator* (double d);
   VECTOR operator/ (double d);
   void   operator= (VECTOR v);
 };
public:          // User declarations
 __fastcall TBlockEq(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TBlockEq *BlockEq;
//---------------------------------------------------------------------------
#endif
```

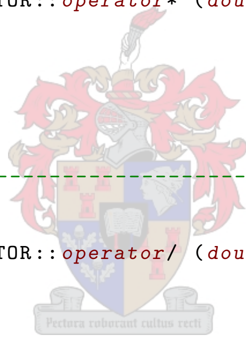
H.3.2 Body

```
#pragma inline
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "FrmBlock.h"
#include "math.h"
#include "stdlib.h"
#include "dos.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TBlockEq *BlockEq;
//---------------------------------------------------------------------------
```



```
__fastcall TBlockEq::TBlockEq(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------

TBlockEq::VECTOR TBlockEq::VECTOR::operator+ (VECTOR v){
 VECTOR ans;
 ans.x = x + v.x;
 ans.y = y + v.y;
 ans.z = z + v.z;
 return ans;
}
//---------------------------------------------------------------------------

TBlockEq::VECTOR TBlockEq::VECTOR::operator+ (double d){
 VECTOR ans;
 ans.x = x + d;
 ans.y = y + d;
 ans.z = z + d;
 return ans;
}
//---------------------------------------------------------------------------
```

```
TBlockEq::VECTOR TBlockEq::VECTOR::operator- (VECTOR v){
 VECTOR ans;
 ans.x = x - v.x;
 ans.y = y - v.y;
 ans.z = z - v.z;
 return ans;
}
//---------------------------------------------------------------------------

TBlockEq::VECTOR TBlockEq::VECTOR::operator- (double d){
 VECTOR ans;
 ans.x = x - d;
 ans.y = y - d;
 ans.z = z - d;
return ans;
}
//---------------------------------------------------------------------------

TBlockEq::VECTOR TBlockEq::VECTOR::operator* (VECTOR v){
 VECTOR ans;
 ans.x = (y * v.z) - (z * v.y);
 ans.y = (z * v.x) - (x * v.z);
 ans.z = (x * v.y) - (y * v.x);
 return ans;
}
//---------------------------------------------------------------------------

TBlockEq::VECTOR TBlockEq::VECTOR::operator* (double d){
 VECTOR ans;
 ans.x = x * d;
 ans.y = y * d;
 ans.z = z * d;
 return ans;
}
//---------------------------------------------------------------------------
```




```
TBlockEq::VECTOR TBlockEq::VECTOR::operator/ (double d){
 VECTOR ans;
 ans.x = x / d;
 ans.y = y / d;
 ans.z = z / d;
 return ans;
}
//---------------------------------------------------------------------------

void TBlockEq::VECTOR::operator= (VECTOR v){
 x = v.x;
 y = v.y;
 z = v.z;
}
//---------------------------------------------------------------------------

void __fastcall TBlockEq::FormClick(TObject *Sender)
{
 Screen->Cursor = crHourGlass;

//Constants
 double pi = 3.14159265358979;
 double a  = 2*pi*10e3;
 double G  = 63025;

 double g  = 267.522128e6;
 double Bx = 10e-6;//910e-12;
```

```
 double T1 = 3.6;
 double T2 = 4.7;
 double a0 = 3.281003e-3;

 double Limit = 0.05*a0; //The threshhold for pulses.

 double tmax  = 30;
 int    N     = tmax*100e3;
 double dt    = 1./100e3;

//Variables
 VECTOR B;
 VECTOR f1;
 VECTOR f2;
 VECTOR f3;
 VECTOR f4;
 VECTOR temp;
 VECTOR temp2;
 int j;

//Vectors
 double* t   = new double[N];
 double* B0  = new double[N];
 double* w0  = new double[N];
 double* M0  = new double[N];
 VECTOR* M   = new VECTOR[N];
 double* wrf = new double[N];
 double* phi = new double[N];


 for(j = 0; j < N; j++){
  t[j] = dt*j;
  if(t[j] < 2.2){
   B0[j] = 0.5*t[j] + 0.1;
  }else if(t[j] < 10){
   B0[j] = 1.2;
  }else if(t[j] < 12.2){
   B0[j] = -0.5*(t[j] - 10) + 1.2;
  }else{
   B0[j] = 0.1;
  }
///*Temp*/B0[j] = 2*pi*50e6/g;/*Temp*/
  w0[j] = -g*B0[j];
  M0[j] = a0*B0[j];
 }

 M[0].x = 0.;
 M[0].y = 164e-6;
 M[0].z = M0[0];
 wrf[0] = w0[0];
 phi[0] = atan2(M[0].y, M[0].x) - pi/2;
 a = G*(1+a*dt);
 for(j = 0; j < N-1; j++){
  phi[j+1] = atan2(M[j].y, M[j].x) - pi/2;
  wrf[j+1] = wrf[j] + a*phi[j+1] - G*phi[j];

  if(M[j].y < Limit*g*0.1/(-w0[j+1])){
   B.x = Bx*g*0.1/(-w0[j+1]);
  }else{
   B.x = 0.;
  }

  B.y = -M[j].y*(496.1004265e-27 * w0[j+1]* w0[j+1])/(30e-3 +
        (394.7841759e-21 * w0[j+1]* w0[j+1]));
```

```
  B.z = (wrf[j+1]-w0[j+1])/g;

  temp.x = M[j].x/T2;
  temp.y = M[j].y/T2;
  temp.z = (M[j].z - M0[j])/T1;
  f1 = ((M[j]*B)*g - temp)*dt;
  temp2 = M[j]+(f1/2.);
  temp.x = temp2.x/T2;
  temp.y = temp2.y/T2;
  temp.z = (temp2.z - M0[j])/T1;
  f2 = ((temp2*B)*g - temp)*dt;
  temp2 = M[j]+(f2/2.);
  temp.x = temp2.x/T2;
  temp.y = temp2.y/T2;
  temp.z = (temp2.z - M0[j])/T1;
  f3 = ((temp2*B)*g - temp)*dt;
  temp2 = M[j]+f3;
  temp.x = temp2.x/T2;
  temp.y = temp2.y/T2;
  temp.z = (temp2.z - M0[j])/T1;
  f4 = ((temp2*B)*g - temp)*dt;
  M[j+1] = M[j] + (f1 + (f2*2.) + (f3*2.) + f4)/6.;
 }


 AnsiString s;
 bool b;
 File.FileName = "E:\\Buffer\\Data.txt";
 if(File.Open(JFile::Create)){
  File.WriteLine("t    w0    Mx    My    Mz    phi    wrf");
  for(j = 0; j < N; j += N/1000){
   s = FloatToStr(t[j]) + "    ";
   File.WriteBuffer(s.c_str(), s.Length(), b);
   s = FloatToStr(w0[j]) + "    ";
   File.WriteBuffer(s.c_str(), s.Length(), b);
   s = FloatToStr(M[j].x) + "    ";
   File.WriteBuffer(s.c_str(), s.Length(), b);
   s = FloatToStr(M[j].y) + "    ";
   File.WriteBuffer(s.c_str(), s.Length(), b);
   s = FloatToStr(M[j].z) + "    ";
   File.WriteBuffer(s.c_str(), s.Length(), b);
   s = FloatToStr(phi[j]) + "    ";
   File.WriteBuffer(s.c_str(), s.Length(), b);
   s = FloatToStr(wrf[j]) + "    ";
   File.WriteBuffer(s.c_str(), s.Length(), b);
   File.WriteBuffer("\r\n", 2, b);
  }
 }else{
  File.ShowLastError();
 }

 delete[] t  ;
 delete[] B0 ;
 delete[] w0 ;
 delete[] M0 ;
 delete[] M  ;
 delete[] wrf;
 delete[] phi;

 Screen->Cursor = crDefault;
}
//---------------------------------------------------------------------------
```