Polymorphic Computing Paradigms Realized for a FPD based Multicomputer

by

Glenn K. Rosendahl

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

ISBN 0-612-16259-1

Name _____

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Parallel Processing And Field Programmable devices. (SUBJECT TERM) | 0544 (SUBJECT CODE) | U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology ... 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology .. 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics ... 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Nom _____

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

_____ ☐☐☐ U·M·I

SUJET — CODE DE SUJET

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS
Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION
Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Évaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE
Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE
Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES
Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-être social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES
Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE
Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT
Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hôpitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES
Sciences Pures
Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÈres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Électronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie
Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et thermodynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systèmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE
Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

POLYMORPHIC COMPUTING PARADIGMS REALIZED FOR A FPD BASED MULTICOMPUTER

BY

GLENN K. ROSENDAHL

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

Abstract

In general, parallel processing has not gained wide acceptance to date, due to many problems associated with cost, design effort, and a limited scope of application. Many of these problems are related to the rigid nature of hardware architecture, which prevents systems from conforming to diverse application needs. The present thesis defines a flexible parallel architecture based on a large number field programmable devices, which enables a wider application scope through architectural flexibility. Several application architectures are presented, demonstrating control flow, data driven, demand driven, and hybrid computing paradigms.

A design method for Xilinx 3000 series field programmable logic arrays is presented. This method is hierarchical in construction and enables the rapid prototyping and design of register transfer sequencers. Further, a method for expanding designs beyond chip boundaries is also presented.

Two neural network applications are discussed and the Kohonen self-organizing feature map is implemented on the present architecture. A network of 588 nodes is executed with network solution times of 2.616ms and 1.368ms per input sample for one and two processors respectively.

Field programmable devices can be an invaluable resource to multicomputers and large systems in general. As a result of this study of large system designs, a number of practical and interesting issues related to field programmable devices have been described, including design practices, testing, and architectural flexibility.

Contents

1 Introduction **7**

1.1 Background 8

1.2 Summary and Scope 11

1.3 Contributions 12

2 Hardware **14**

2.1 Hardware Specification 16

2.1.1 Communication Network 17

2.1.2 Processing Elements 20

2.1.3 User Interfaces 22

2.2 Design Tools and Methods 24

2.3 Hardware Implemented 25

2.4 Hardware Commissioning 31

2.5 Summary 33

3 FPGA Application Architectures **34**

3.1 FPGA Design Tools and Techniques 35

3.2 Message Decoder 35

3.3 Application Architectures 39

3.3.1 Control Flow Model 39

3.3.2 Data Flow Model 40

3.3.3 Data Driven Model 41

3.3.4 Demand Driven Model 43

3.3.5 Hybrid Computing Paradigms 44

3.4 Operating System Concerns 44

3.5 Summary 45

4 Applications **46**

4.1 Backpropagation Networks 46

4.1.1 Sigmoid Transfer Function 48

4.1.2 Network Task Allocation 49

4.2 Self-Organizing Feature Maps 51

4.2.1 Kohonen SOFM 52

4.2.2 Parallel Implementation 55

4.2.3 A Dynamic Learning Rule for SOFMs 56

4.3 Summary 58

5 Conclusions **59**

5.1 Retrospective 61

A Electromagnetic Transient Simulation Hardware **63**

A.1 Introduction 63

A.2 Model Characteristics 64

A.3 System Design 65

A.4 Hardware 66

A.4.1 Processing Element 66

A.4.2 Bus Master 68

A.4.3 Communication Port 69

A.4.4 Constructed Prototype System 71

A.5 Software 71

A.5.1 Switch Handling 71

A.5.2 Solution Flow 72

A.6 Architectural Observations 73

B Parallel Sparse Matrix Solutions **74**

B.1 Introduction 74

B.2 Symmetric Sparse Matrix Analysis Tool 74

B.3 Summary 78

C Overview of FPD Signal Pin Assignment and FPGA architecture 82

C.1 PAL Pin Assignment 82

C.2 FPGA Pin Assignment 84

C.3 Xilinx 3000 Series FPGA Overview 85

D Control Sequence Design Methods for Xilinx 3000 series FPGAs 90

D.1 Control Sequence Design Elements 91

D.2 Summary 94

Bibliography 95

List of Figures

2.1 Message arbitration and exchange mechanism. 18
2.2 Message decoder. 19
2.3 Processing element task queue. 21
2.4 User interface module. 23
2.5 Photograph of message decoder daughter PCB 26
2.6 Photograph of user interface module implemented 27
2.7 Photograph of dual processing element PCB (component side) . . 29
2.8 Photograph of tester for the message decoder 30
2.9 Photograph of three chip module tester with DPE mounted 32

3.1 Structure of hash table entry. 38

4.1 Backpropagation network topology 47
4.2 Sigmoid transfer function and lookup table 48
4.3 Processor Two Network Task Allocation 50
4.4 An example SOFM construction. 52
4.5 Comparison of step-wise neighbourhood (upper row) and continuous neighbourhood (lower row) methods for SOFMs. 53
4.6 Example SOFM network approximates input probability distribution. 54
4.7 Effect of learning rate to decay rate ratio on output node density. 57

A.1 Generic passive component model. 64
A.2 Parallel Processing Unit block diagram. 66
A.3 Processing Element block diagram. 67
A.4 Bus Master block diagram. 69
A.5 Communication Port block diagram. 70

A.6 Constructed prototype block diagram. 72
A.7 Sample interconnection of PPUs. 73

B.1 Solution graph of 14x14 band matrix without permutation. 75
B.2 Solution graph of 14x14 band matrix with permutation. 76
B.3 Resultant matrices for enhanced parallel ordering. 77
B.4 Parallel operations of permuted matrix. 78
B.5 35x35 matrix with half band of 1. 79
B.6 35x35 matrix with half band of 2. 80
B.7 35x35 matrix with half band of 3. 81

C.1 Some common PAL device pin assignments 83
C.2 Input/output block 86
C.3 Configurable logic block 87
C.4 Combinatorial function unit modes 88
C.5 Xilinx routing resources 89

D.1 Act1 state diagram and example logic waveforms. 91
D.2 Act2 construction, equivalent state diagram, and example logic waveforms. 92
D.3 Examples of common sequence structures. 93
D.4 FPGA chip boundary cell construction. 94

Chapter 1

Introduction

After many years of effort, parallel processing has not yielded the general purpose computation engine once thought easily possible. With the recent demise of several leading companies, a diminished faith is evident. This is in contrast to the highly successful application specific designs, where architectures are crafted to mimic application requirements. Given the large effort and expense required for each desired application, few designs stand as testimony. Further, application specific designs, are routinely found to be inflexible to applications even slightly out of their designed scope. These problems, among others, have hindered the wide acceptance of parallel processing.

With the advent of field programmable devices, and current research into rapid prototyping methods, many of the aforementioned problems may be averted. Field programmable devices enable the underlying logic and physical connections of a design to be altered after construction via software. This can increase the flexibility of previously inflexible designs and ultimately increase the scope of suitable applications.

The present thesis reflects many facets of interest in large system design. One primary interest is in determining what roles field programmable devices can play both during and after the design phase. Secondary interests include what impact these devices will have on application development and the tools for application development. In order to examine these aspects, a large system design and implementation has been undertaken. The system is composed almost exclusively (with the exception of memory and processing units) of field

programmable devices to form a flexible parallel architecture capable of handling a wide range of applications effectively. Other aspects include the use of these resources for initial debugging, as an online diagnostic tool, to provide application mission specialists, and to provide polymorphic computing paradigm capabilities for parallel applications. The following background section provides boundaries for a parallel system design and points to a level of architectural flexibility required.

1.1 Background

Parallel processors, to date, have been primarily algorithmically specific architectures. That is, the architecture is tailored to address the desired problem or class of problems [33]. This is a very reasonable approach considering how well these machines perform the tasks for which they are designed. However, if a task outside of the designed specialization is requested, performance is diminished (in some instances dramatically). The central reason is that the architectures, in general, are fixed or restricted in configuration (e.g., a mesh connected processor network), and an algorithm's use of a topology may or may not be optimal. The latter is usually the case. These algorithms and architectures have a "Control Flow" philosophy which is currently dominant.

Another philosophy which offers greater freedom in the flow of solutions and computing is "Data Flow". This approach has many forms, the primary one is "Data Driven". In data driven structures, the availability of data invokes processing, suggesting that processing proceed in an asynchronous manner (as data becomes available). The greatest advantage of data driven structures can be seen in applications to parallel processing systems, where no synchronization between procedures or processors is explicitly expressed by the user or programmer but is implicitly ensured by the machine architecture. Data driven structures enable the algorithm's natural data dempendencies to control the flow of a solution, and relieves the programmer from explicitly specifing the flow of a solution. The languages that have been developed for these machines are known as "functional languages" [24,58,65]. A large effort has been invested, and many machines have been constructed using this philosophy but due to technological limitations, none

have attained commercial success. Some have suggested the reason for this is that the architecture makes use of the finest grain of a solution such that an unrealistic overhead is associated with the transportation and identification of functions and data [24,58].

Other forms of computing can be found in the literature on capability machines or small objects (ie. object oriented machines)[24,65]. These machines are similar to data flow architectures in a number of ways, although there are a few exceptions. Capability machines were first thought of for the protection of personal data in computers to prevent unauthorized use in the 1950s. The first capability machine was built in 1958 at the University of Chicago but attained very limited success. Several other capability machines were constructed around the world. To date only a select few of these machines ever reached commercial success primarily because of the computational overhead associated with maintaining an object's "capability" and access rights. These architectures are enjoying a resurgence due to recent technological advances and an increasing concern for fault detection or data security.

Simulation of highly non-linear physical phenomena plays an indispensable role in many disciplines with respect to the design and analysis of systems. As simulation accuracy and size limitations become more acute, parallel processors have increasingly become the target for these applications. Two such applications are electromagnetic transient and transient stability analysis for power systems. Considerable work has been done concerning these two classes of power system simulations as is evident in the EPRI reports of 1977 to 1987 [7,11,16-20,45] and in the IEEE Transactions on Power [1,3-6,8,9,12-15,22,25,26,30,34-36,40,41,46,49,57,60-63,68,69,70,71,75-77]. The primary goal of this research has been to reduce the computation time and to minimize the errors incurred, by improving solutions or modeling methods. An interest in accelerating the electromagnetic transient and transient stability solutions through specialized hardware is also evident [1,4-9,12,15-19,22,25,26,30,34,40,41,45,46,49,57,62,63,68,69,70,75,76].

Electromagnetic transient simulations provide short term analysis of the power system during overvoltages and component stresses imposed by faults. Specialized hardware has been constructed for simulating this phenomenon in real time and

is used in the commissioning and evaluation of system controls and protection equipment [42,50]. These simulations require the solution of thousands of differential equations every 50 microsecond time step, where the solution of differential equations is the primary component of the simulation. This is reflected in the specialized hardware, in which the architecture incorporates a high computation bandwidth, a relatively low communication bandwidth and a control flow paradigm (an overview of this design is presented in Appendix A).

Transient stability analysis is used in operation and planning departments of power utilities for cost effective bulk power transmission. Given that the power system is growing and is being interconnected with a greater number of other systems, these studies play an important role in the management of the power system. These simulations, to date, are almost exclusively run on digital computers with very large computation requirements, usually exceeding the power utilities' computing facilities both in hardware and software. Tradeoffs in performance and accuracy are common decisions made in using these tools. Typical simulations require very large sparse matrices to be solved, which represents an estimated 32% of computation requirements, whereas the differential equation solutions consume 64% of overall computation [8]. No effective parallel solution to address this problem has been forwarded to date. One of the reasons for this is the disparity of resource requirements between the differential equation and sparse matrix solutions. That is, differential equations require high computation and low communication bandwidth, whereas sparse matrix solutions have low computation and high communication bandwidth requirements.

The sparse matrix solution represents a difficult problem to vector based machines. Partitioning of the solution is in the forefront of these problems, and more often than not, effective use of the vector processor cannot be made. This becomes especially evident with sparse matrices having a low half band size, as is the case with transient stability solutions (see Appendix B for a discussion of this problem). Many non-commercial machines have found better solutions to this problem through functional languages and data flow computing models. This enables a matrix topology to be expressed at compile time instead of at run time.

As previously mentioned, a data flow program is written such that the ordering

or partitioning of operations is not explicitly specified, but is implicitly derived from the data interdependencies. This suggests an architecture that has self organizing or partitioning features not present in other forms, and would seem to be an ideal match to the problems of sparse matrix computation.

Artificial neural networks (ANNs) are another class of applications which suffer from inadequate computing resources. Two networks of current interest are backpropagation [56], and self-organizing feature maps (SOFM) [38]. Both of these networks are very regular in construction, and are readily adaptable to vector processors. However, the sheer volume of calculations necessary (especially during learning), and a desire to incorporate these networks within consumer products, has led to very large scale integration (VLSI) implementations [10,28,67]. Both analog and digital implementations are progressing, with digital being more prevalent, so far. A new computer design would have difficulty competing with VLSI implementations on computation rate alone. However, an architecture which is both flexible and cost effective, may provide a viable test-bed for new algorithms and applications, which are subsequently targeted for VLSI implementation.

1.2 Summary and Scope

The above short overview indicates that an application can require a diverse set of resources in order to attain an efficient solution. It is also apparent that flexibility can be beneficial, in terms of communication, computation bandwidth, and the computing model. This points to a level of architectural polymorphism previously not possible, given technology constraints. With the advent of field programmable devices (FPD), a greater level of flexibility should be possible.

This thesis aims to demonstrate some of the many benefits of using FPD technologies in large system design, and in particular, the advantage FPD technologies can provide to parallel systems. This is motivated by the present lack of large system design practices for FPDs, and by the application restrictions that rigid architectures can impose on parallel systems.

In Chapter Two, the underlying hardware architecture of a multicomputer is defined, with a focus on flexibility and simplicity. Several design practices which

enhance system flexibility and testability are identified. A discussion on hardware construction, as well as the problems encountered, is presented, followed by a summary.

Chapter Three presents several of the many possible application architectures that may be implemented in the physical hardware structures of Chapter Two. To begin, a short discussion of design tools and methods is provided. Next, an associative memory processor design implemented within FPDs of the message decoder is discussed. Several computing model application architectures are explored, operating system concerns are overviewed, and a summary is provided.

Chapter Four discusses two preliminary applications implemented on the present multicomputer, with an emphasis on the influence that application architectures exert on problem partition decisions. The backpropagation neural network is first overviewed, followed by two possible problem partitions on the present architecture. An indepth discussion of the self-organizing feature map neural network algorithm is provided. Several learning rules are presented, the algorithms are partitioned and implemented, and actual system performances are recorded. Finally, a summary is provided.

Chapter Five presents conclusions, and offers a retrospective look at the overall design and implementation with suggestions for possible changes.

1.3 Contributions

Contributions of this thesis include:

i) the design and implementation of a parallel DSP machine with extensive reprogrammable logic devices. The initial design was first presented at *The Canadian Conference on Electrical and Computer Engineering* in 1991 [52], and *The First International Workshop on FPGAs*, at Oxford in 1991 [51].

ii) the exploration of rapid prototyping through the use of FPGAs and their role in facilitating system design. Several test applications such as memory system test and boundary scan were presented at *The 6th Workshop on New Directions for Testing*, held in Montreal, 1992 [53].

iii) commissioning of a working prototype and associated test methodologies. Here again, the FPGA devices were utilized to build application specific testers.

iv) development of an efficient protocol in support of algorithmic design.

v) demonstrated application implementations of two sufficiently diverse algorithms to illustrate the flexibility of the prototype.

vi) delivery of a working system to the University of Manitoba that can be utilized in the development of parallel processing applications.

Chapter 2

Hardware

Application-specific designs implemented through a fixed or rigid architecture limit application scope. Diverse applications can require diverse resources and system configurations, which can strain rigid designs beyond their capabilities. An application and architecture is matched through an iterative procedure of solution partition and evaluation, until a suitable match can be found. This is a time consuming and expensive task, with no guarantees for success. Further, current development tools offer little aid.

A key issue to the successful development of an application is choosing the right data structure for both a solution and destination architecture. However, there has been little or no effort to standardize, catalog and incorporate well-suited data structures bound to commonly-used solutions into development tools. Lacking this knowledge foundation, manufacturers of parallel hardware make poor architectural decisions. Would it not be more efficient to create development tools which are familiar with the destine architecture, data structures, and common solutions? Presently, it is the programmer's responsibility to be familiar with solutions, architectures, and data structures, which leads to ad hoc and inefficient applications. However, given the rigid nature of current architectures, it is uncertain how much help such tools could provide.

It has been said that the software should not be *tuned* to the hardware, but instead the hardware should conform to the software [24]. There is little doubt that including hardware as a dynamic participant would be beneficial in application development. With dynamic hardware, a closer match between

solution requirements and architecture may be achieved. The penalty is a more complicated application development environment, which includes both hardware and software components.

Recently, there have been some interesting proposals for object-oriented distributed operating systems (OODOS) which may solve many of the aforementioned problems [23,29,72-74,64,66]. In these proposals, each object of an application is assigned a metaspace constructed of metaobjects, which define an execution environment for the object. Through metaobjects, a virtual processor is created upon which to execute an object's methods. The virtual processor allows objects to migrate across heterogeneous system boundaries dynamically. All of these attributes are well suited to addressing the problems of parallel application development. If metaobjects of these proposals could be modified to include hardware definitions, a library of hardware metaobjects could be constructed to support a diverse set of application requirements. Then, through the metaspace and object-oriented environment, a library of application objects could be constructed that encapsulate data structures and solution methods, which, in turn, are implicitly bound to the hardware supporting metaobjects. An application development environment may be created which thereby, hides both the hardware and software complexities of architectures from the programmer and applications alike.

This chapter defines a flexible hardware base upon which application architectures are built. It is hoped such a flexible structure will yield a wider application scope than previous rigid designs. To meet this goal, a large number of field programmable devices (FPD) are employed as an integral part of the design. Other objectives of this chapter include the exploration of FPD design, quick prototyping, and testing issues. In the following sections, the hardware specification and design rationale is discussed. A short overview of design tools and methods is presented. This is followed by hardware implementation details, commissioning problems and solutions, and finally, a summary.

2.1 Hardware Specification

As a design criterion, the use of *discrete* logic was discouraged in favor of FPDs. Two types of FPDs are used; one is volatile, the other is not. The non-volatile devices are programmable array logic (PAL) and do not require configuration upon power being applied. The chosen field programmable gate array (FPGA) devices, on the other hand, do require a reconfiguration mechanism.

Due to the size and potential complexity of the design, a method of system testing should be included. This method should be flexible enough to test individual components or modules, and the connections between them. Also, with hardware now being an integral part of applications, the ability to debug the combination of hardware and software would be beneficial.

Several parallel architectures were considered before a choice was made. One of the main selection criterion was that, the architecture should be easy to develop application software for, and yet have reasonable performance. This has led to the need for an architecture which has a simple communication structure capable of communication topologies ranging from point-to-point to broadcast. Another important criterion was to give FPDs a visible role in both architecture and applications.

A central component of any parallel architecture is the communication network. If the network is topology-limited, or if rigid restrictions are imposed, application scope can be diminished. C.A.R. Hoare in 1978 introduced message-passing structures through development of the communicating sequential processes (CSP) programming method. This method not only simplified parallel programming, but enabled parallel algorithms to be proven to work. For this reason, and because of inherent advantages over shared-memory models, a message-passing model was selected.

Point-to-point communication networks are limited by the sheer number of channels required to form general topologies. These communication structures also pose message routing problems and require a significant effort to develop applications. However, broadcast networks, multiplexed in time, enable any communication topology for a minimum cost. These networks ease application

development through limitless communication topology, and are simple to construct. For these reasons, the current design defines a message-passing broadcast network as the primary communication network. The present specification is composed of three parts: the communication network, the processing elements, and the user interfaces.

2.1.1 Communication Network

Broadcast networks are the most general networks, and have many appealing attributes. Any virtual topology is possible (multiplexed in time), only one channel interface is needed, and no decision logic is required to select a channel for export information. These attributes reduce the complexity of both hardware and software. One disadvantage, is that the number of participating modules must be limited, to preserve efficiency.

A common bus provides the medium for the current message-passing broadcast network. Since the common bus represents a potential communication bottleneck, emphasis is placed on maximizing the exchange efficiency of message traffic. To accomplish this, a number of simplifications are made. Messages are defined as a number of 36 bit words, with each word containing four control bits and 32 bits for data. Destination information is encoded within the first word(s) of a message, and message length is defined by one bit in the control field, called the end-of-message (EOM) bit. Input and output message queues of every module are constructed from banks of first-in first-out (FIFO) buffers, and addresses are assigned to bus modules only, not to individual words of a message. Module addresses select the source of a communication, messages are an ordered set of words and no signals or logic are defined to exclude modules as destinations. Thus, messages are exchanged between a source module and all other modules on the bus, including the source itself. These simplifications reduce complexity of bus control, increase message-exchange efficiency, and add a generality to the communication network protocols. However, these simplifications shift complexity from bus-control logic to message-receiver logic. It was decided that this shift was justified, given the potential bottleneck posed by broadcast networks.

Control of message traffic has three phases: message arbitration, exchange,

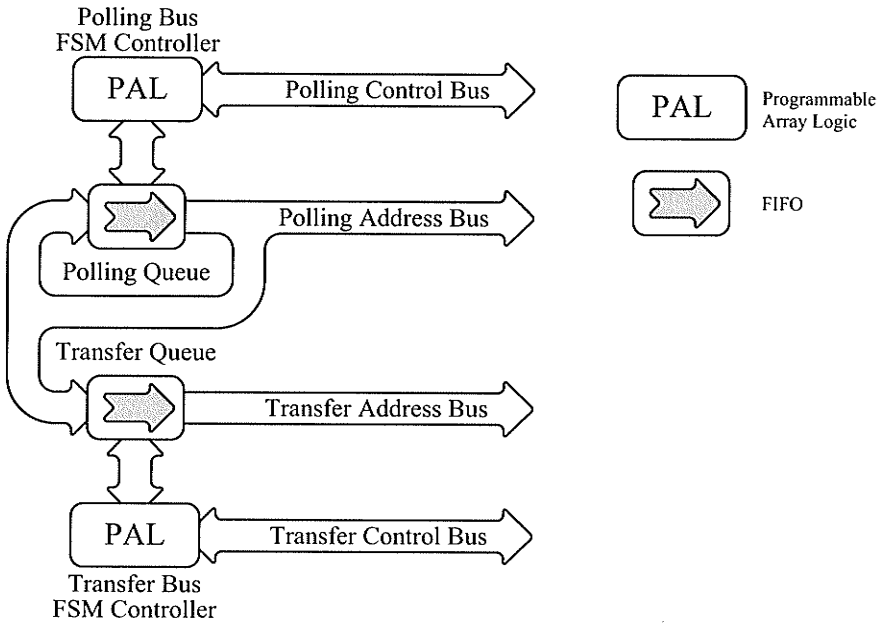


Figure 2.1: Message arbitration and exchange mechanism.

and decoding. Message arbitration and exchange is performed by a centralized bus control, while message decoding is performed by receiver logic on each module. The message arbitration and exchange phases are split onto two sub-buses of the common bus: the polling bus for message arbitration, and the transfer bus for message exchange. Each sub-bus controller is constructed from one FIFO, for queuing module source addresses, and one PAL, implementing a finite state machine (FSM) to sequence bus signals (Fig. 2.1).

To begin, a polling sequence is written to the polling queue, the transfer queue is cleared, and both FSMs are enabled. Each address of the polling queue is presented to the polling bus and the selected module may respond with a message-transfer request. Whether a request is made or not, this address is written to the back of the polling queue to preserve the initial sequence. If a transfer request is made, this address is also written to the back of the transfer queue. Messages are exchanged by presenting addresses, written to the transfer queue, to the transfer bus, selecting a source module for communication. Then, one signal is cycled once for each word of a message until the EOM bit is detected, completing the exchange. During each cycle of this signal, message data is read from the source module's

output queue, and written to every module's input queue. If the transfer queue is kept non-empty, the communication network operates at its maximum efficiency. In this way, a highly-efficient message arbitration and exchange mechanism is constructed. However, note that this efficiency is only maintainable if the message decoding logic on each module can keep pace with message traffic. Otherwise, message-transfers will be slowed to the slowest message decoder on the bus.

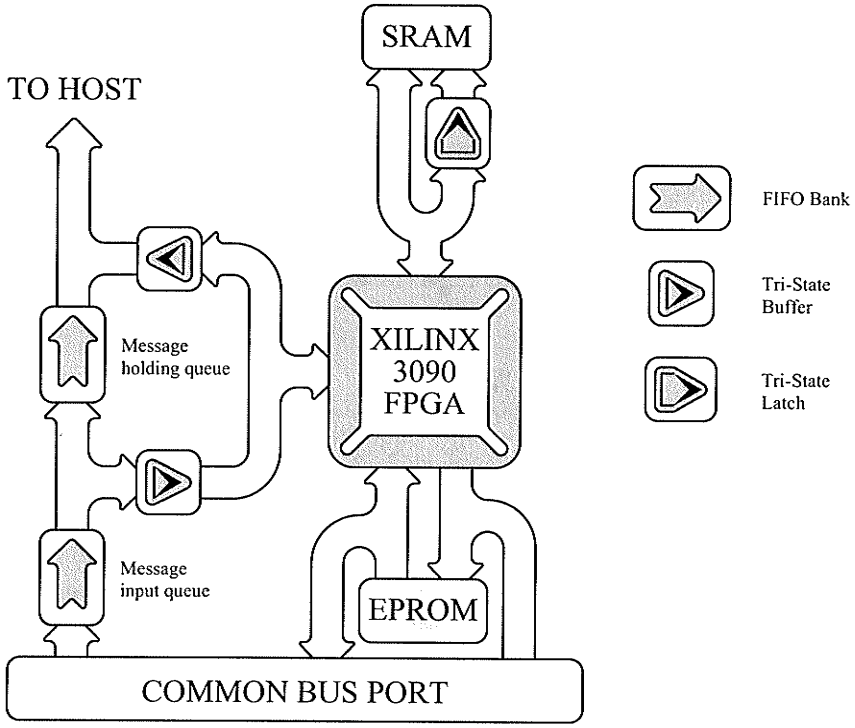


Figure 2.2: Message decoder.

In specifying the message-decoding mechanism, it is reasonable to assume that only a small fraction of the incident messages will be required by a host. Therefore, a successful implementation will depend heavily on an efficient message discard-mechanism. The present message decoder employs a three-level pipeline structure for this purpose. The first level is the input queue, accepting raw message traffic from the bus. The output from this queue has two possible destinations: the message-holding queue, or message-decoding logic (Fig. 2.2). When a message arrives, the head of the message is routed to the decoding logic. While the destination is being decoded, the body of the message is passed to the message-holding queue. At the same time, a previously-accepted message is transferred

from the message-holding queue to the host. If the decoder fails to accept a message, the message is discarded, simply by toggling the message-holding queue's reset signal.

In order to enable a wide selection of decoding schemes, the decoder logic is implemented in an FPGA. Other resources such as RAM, EPROM, and PALs are also included to further enhance decoder-selection options and message-processing speed. With this simple message arbitration, exchange, and decoding mechanism, sustainable data rates of over 40MB/s are readily achieved.

The common bus includes three additional sub-buses, an FPGA reconfiguration bus, a test-access port (TAP) and a daisy-chained module address allocation bus. The FPGA reconfiguration bus allows individual modules to be reconfigured while all other modules continue undisturbed. This may be beneficial to applications which require mission specialists to be reconfigured at different stages of a solution. Debug and system testing access is provided through the TAP interface, and adheres to the IEEE 1149.1 test access port standard. The daisy-chained module address bus removes the necessity for dip switches or similar hardware to set up individual module addresses.

2.1.2 Processing Elements

The processing element employs Motorola's 96002 digital signal processor chip (DSP). This DSP has exceptional calculation speed, flexibility and a dual external bus construction. Each processing element is modelled after a multilevel task queue (Fig. 2.3) with four distinct zones; the message decoder, pre-processor, processor and post-processor. The message stream is conditioned as it travels the queue for execution and/or export to other elements. The message decoder (described above) discards messages not relevant to the host and forms the first stage of the task queue. The pre-processor stage directs, organizes and assembles information for the host's job input queue. The processor performs computation on data presented and passes results to the job output queue. These results are delivered to the post-processor to be recycled, conditioned, or written to the message output queue for global exchange. This four zone structure is flexible and amenable to many computing paradigms.

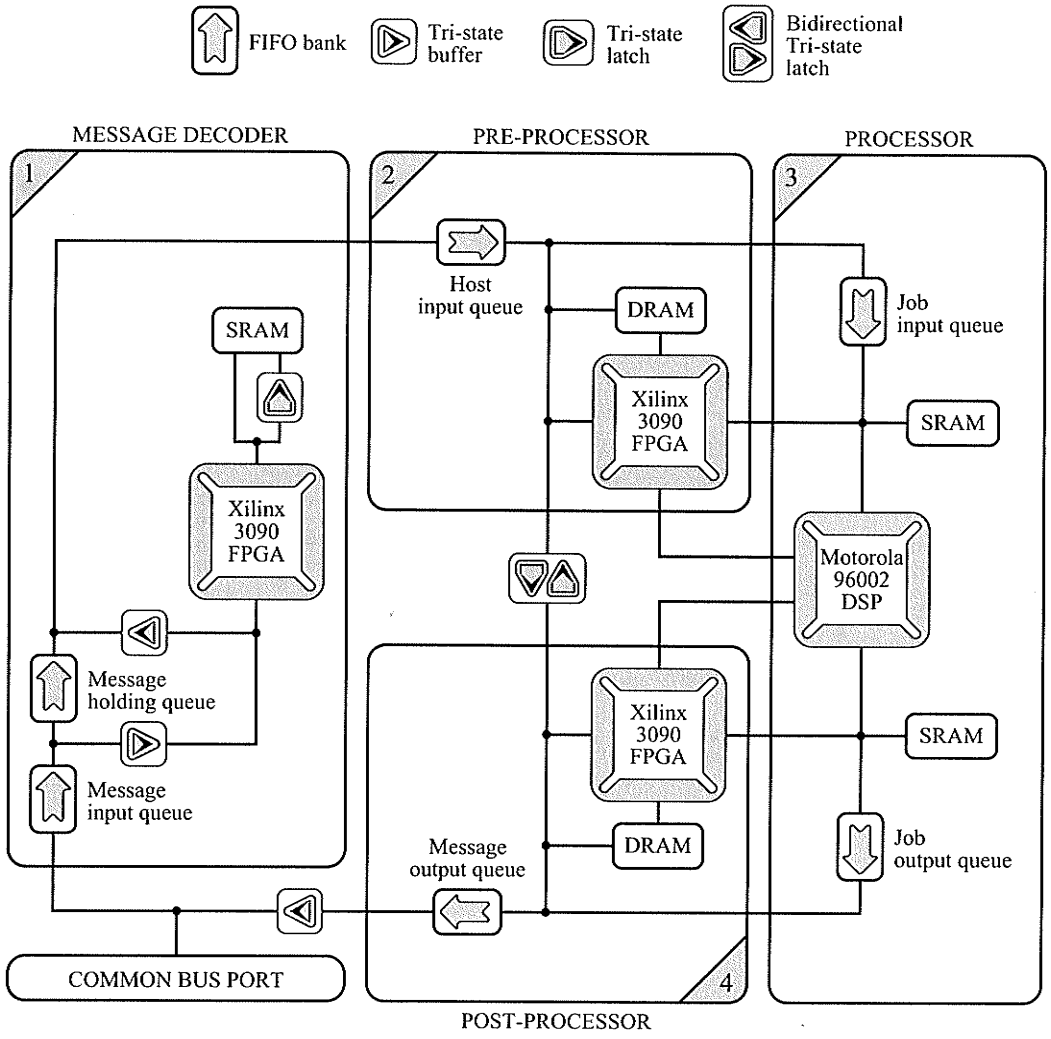


Figure 2.3: Processing element task queue.

The processor zone is constructed from one DSP, two banks of static RAM and two FIFO banks. One external bus of the DSP is defined as the job input bus, while the other is defined as the job output bus. Each external bus has 64K 32 bit words of static RAM for code and data storage. One interface between the input bus and pre-processor and one between the output bus and post-processor is also provided to increase system flexibility.

Both the pre-processor and post-processor are of similar construction; each has one FPGA, one 36 bit wide bank of Dynamic RAM and one FIFO bank for input or output of messages respectively. A bidirectional latch is placed between these two zones to enable information to be exchanged, results to be recycled,

and co-operative processing. In addition, each zone provides an interface with one external bus of the DSP to supplement operating system and computing functions. It is through these zones and processor software that the *character* of the processing element is established.

As mentioned earlier, various applications can require diverse ratios between communication and computation rate to optimize throughput. Consider an application requiring high communication and low computation rates. By configuring the pre-processor as an information decompresser and the post-processor as an information compressor the communication rate can be boosted, while keeping computation rate constant. Next consider the opposite requirement: the pre-processor and post-processor can in this case be configured to supplement computation through the bus interfaces provided, while communication rate remains constant. These are just two simple examples of how FPGA resources of the current specification may satisfy varying application requirements.

In the interest of reducing cost and increasing flexibility, two processing elements are combined onto one printed circuit board (PCB), called a dual processing element (DPE). Both processing elements share the message decoder, and all other zones are independent. A bidirectional latch is provided between both pre-processor zones to enable co-operative processing and communication between the two independent processing elements. This latch defines a secondary communication network and increases computing options.

2.1.3 User Interfaces

The user interface module is the only interface provided between users and applications. It is responsible for initializing the communication network, loading application hardware and software (for each DPE), relaying user commands to applications, and collecting results. This module is divided into two main sections: one for user communication, and one for common bus control and communication. A general purpose microprocessor, the Motorola 68020, was selected as the mediating host between these two sections. Its compatibility with a wide variety of software compilers and operating systems eases user and application interface software construction. An ample supply of EPROM, SRAM and DRAM

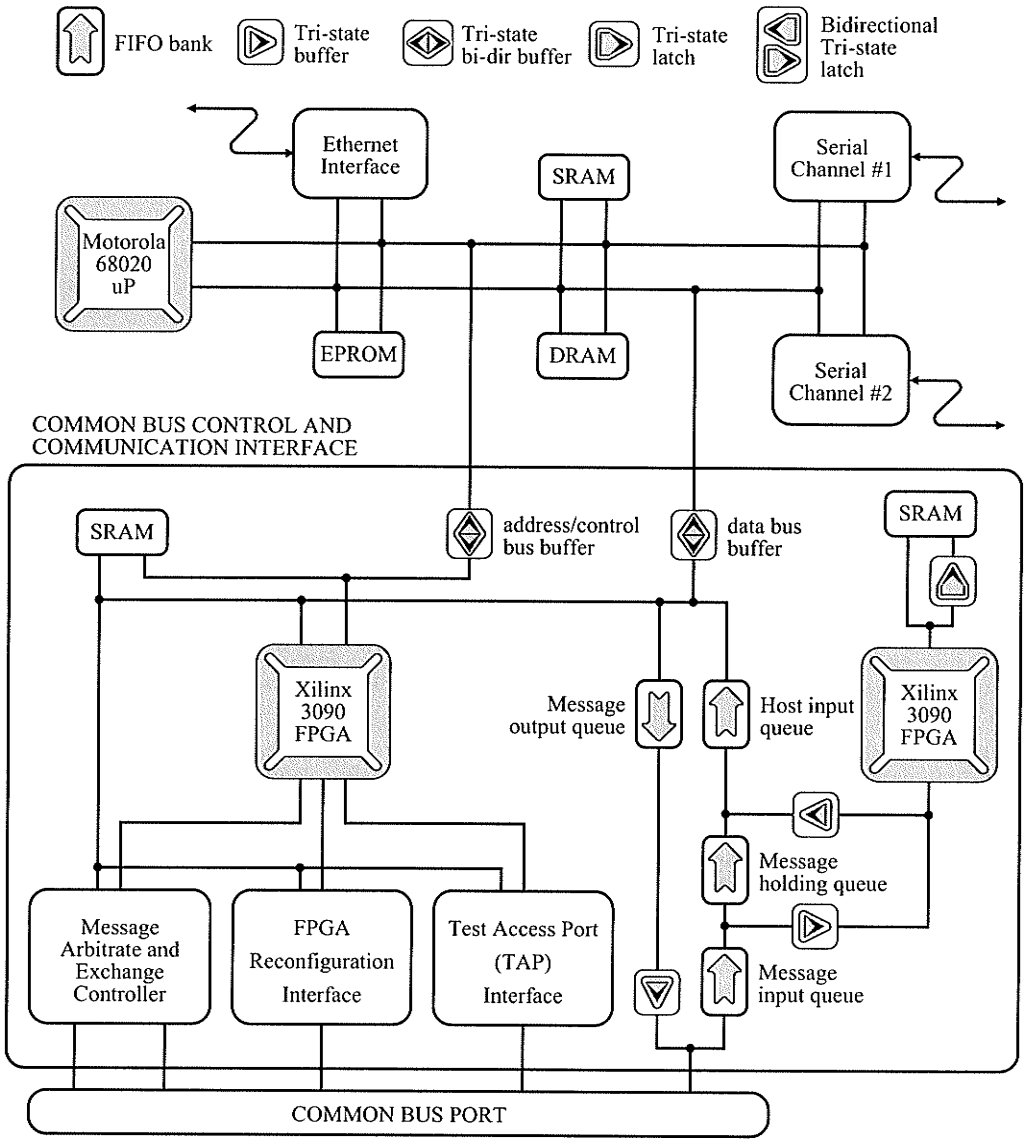


Figure 2.4: User interface module.

is provided to store these host programs and data.

An overall block diagram of the current module is shown in Fig. 2.4. User communication is provided by one ethernet and two serial communication channels. The common bus interface is divided into control and communication. The communication interface is identical to the first task queue stage of the DPEs described earlier. However, the control interface also includes circuitry for the message arbitrate and exchange mechanism, FPGA reconfiguration, and TAP functions. The common bus interface is separated from the host bus with an FPGA and SRAM for control and data storage. This enables control functions

to be implemented within the local FPGA to accelerate and relieve the host from routine management of the communication network.

2.2 Design Tools and Methods

The current hardware specification was implemented using two design tools, a schematic capture program from OrCAD and a printed circuit board (PCB) layout tool from PADS. OrCAD SDT is a very versatile hierarchical schematic entry package with user definable libraries and support for an extensive list of import and export netlist formats. This package graphically defines design schematics, and a part and connection netlist is extracted and imported into PADS PCB for layout. PCB layout packages are similar to commonly available desktop publishing packages, in which raw text (netlists) is formatted into documents (circuit layouts) suitable for publication (manufacture).

To begin with, libraries are created for every part in the design for both the schematic and PCB layout packages. Schematic libraries define the pins and functions of individual parts, whereas PCB layout libraries define the mechanical dimensions and placement of each part and pin respectively. Pins of each individual part have common names in both libraries, which enables mechanical dimensions and connections to be assigned to the design through a netlist. A board outline reflecting the physical size of a PCB is defined within the PCB layout package, a netlist is imported, parts are positioned, and connections or traces are routed. The completed design is then converted into plot and drill files suitable for the manufacturing process.

One incompatibility issue of these two software packages poses a consistency problem between schematic and PCB layout netlists. Often during the course of a PCB layout, pin connections need to be swapped or moved to provide effective placement. This has the effect of changing the pin assignments of the original design. However, since no back annotation path is provided between packages from different manufacturers, changes cannot be reflected within the schematic files automatically. Hence, it becomes the designer's responsibility to maintain consistency between netlists. Other features and functions that were not available

for the current design include: transmission line analysis, mixed-mode schematic entry, and functional simulation. However, by careful signal pin assignment, it was found that the impact of these shortcomings could be reduced.

Several design practices for FPDs have been identified, which maximize flexibility. For programmable array logic (PAL) devices, a flexible schematic pin assignment scheme has been devised, which enables a number of PAL device technologies and types to be employed, even after PCB construction. For field programmable gate arrays (FPGAs), good pin assignment aids internal routing and flexibility. An overview of these design practices and internal construction of the Xilinx 3000 series parts is given in Appendix C.

2.3 Hardware Implemented

The present design is distributed across four PCB designs: the backplane, message decoder, user interface module, and DPE. These designs were constructed in that order, and increase in complexity with each successive design. This order was adopted to encourage familiarity with the design and design tools which would develop as the project progressed, and in turn, to reduce errors and potential costs of errors.

After careful consideration of the available commercial backplanes it was decided that a custom backplane be constructed to reflect the necessary special requirements. Without the availability of transmission line and crosstalk analysis tools, this has proven to be very difficult. Currently the backplane can support clock rates on the order of 10 to 20 MHz, by employing many special impedance matching and ring dampening circuits. The backplane is a two-sided PCB, with module connectors interleaved on opposite sides of the plane. This has reduced the electrical distances, and increased the mechanical distances between modules.

All the common-bus interface circuitry required by modules has been incorporated into the message decoder PCB (Fig. 2.5). This card is a four-layer design, two layers for power and ground connections, and two for trace routing. It is designed to interface with a host module as a daughter card, through two 96-

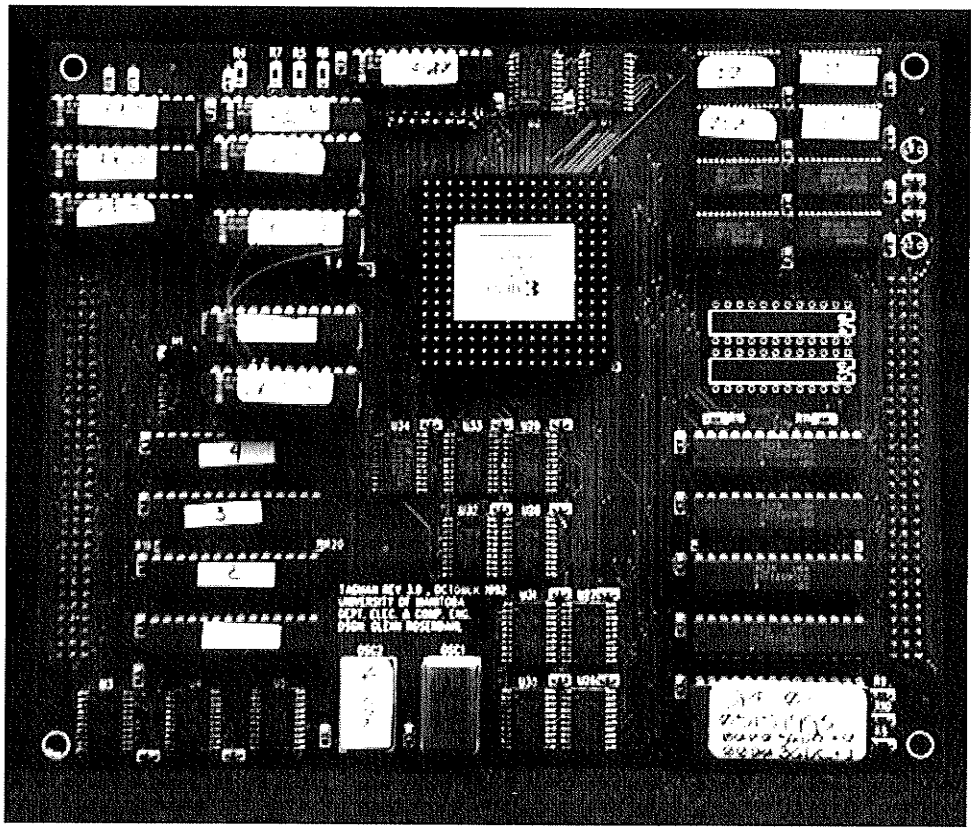


Figure 2.5: Photograph of message decoder daughter PCB

pin connectors situated at either end. Bus message traffic is routed up to the left connector, through the message-input queue (bottom left), to the message-holding queue (bottom right), and down the right connector to the host input queue. The SRAM bank is pictured top right and the EPROM at the extreme bottom right. The decoder FPGA is positioned top centre, with the buffers for striping message destination information just below. At the top left, numerous PALs provide the control logic for the common bus, FPGA, and TAP interfaces. Since the number of elements on a module varies, only the input message queue is provided, leaving the message output queue(s) to be implemented by the host module.

The user interface module is a four-layer 9U,280mm PCB, which adheres to the mechanical specifications of the VME bus (Fig. 2.6). The PCB layer assignments are the same as the message decoder. The two 96-pin female connectors, situated in the centre, accept the message-decoding daughter card, with the host input and message output queues between them (shown to the right and left respectively).

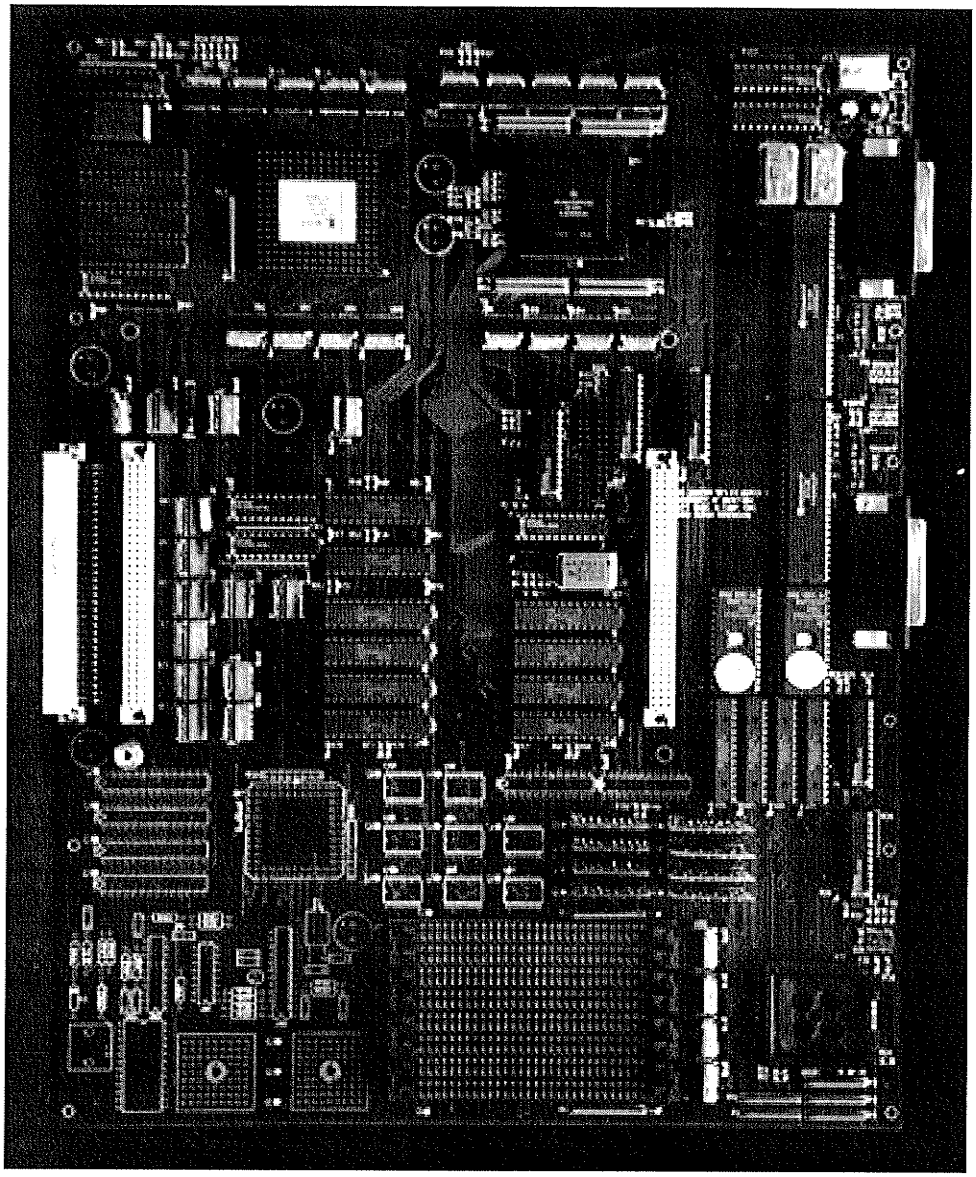


Figure 2.6: Photograph of user interface module implemented

The bus arbitrate and exchange control is implemented with two PALs and two FIFOs, situated just above and to the left of the message output queue. The bottom left of the card implements an ethernet interface (un-populated), while on the bottom right the DRAM controller and memory is shown. The common bus interface FPGA and SRAM is on the top left, and the host processor, memory and serial communications are provided along the top right and right side of the card. Many transmission-line problems are evident by the *tent* structures throughout the board. These structures introduce a series resistance with address, data, and control signal buffers to reduce transmission line ringing effects (a more detailed explanation is provided in the next section).

At present, software is written in assembly language, implementing a menu-based monitor. The monitor is designed such that patch code or new menu functions can be added with ease, but a high-level language compiler would be a definite asset. The ethernet interface is not presently implemented, but all the circuit traces and hardware are available for implementation at a later date. Without the use of a language compiler, implementing the required ethernet protocols would be extremely difficult. The user interface card can either be configured as a bus master, to control all the configuration and operation aspects of the common bus, or as a general-purpose processing element to participate in application solutions. It can also be configured to provide specialized I/O functions. Two user interface cards have been implemented in the current system configuration: one for bus master duties, and another as an interface to a real-time video display to record application results.

The dual processing element (DPE) is a six-layer 9U,280mm card (Fig. 2.7) with two layers for power and ground, and four for routing. Processing element zero occupies the bottom half of the card, while processing element one occupies the top half. Bus message traffic proceeds up the left daughter board connector, through the message decoder and down the right connector. It is split into two streams at this point, one for both processing elements through two FIFO banks (situated centre right). These streams proceed to the pre-processors (top and bottom right), through the processors (centre top and bottom), to the post-processors (top and bottom left), and finally to the message output queues (centre

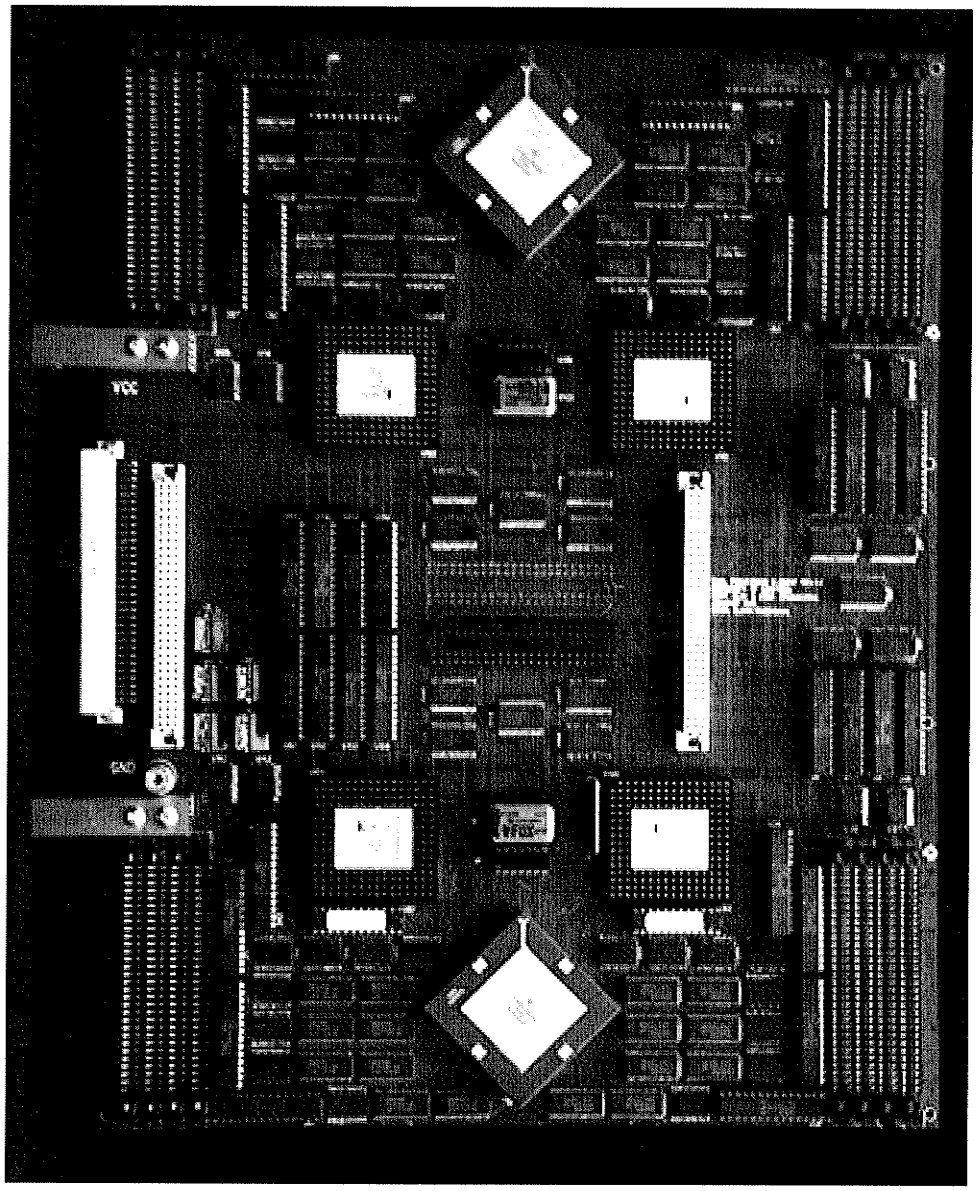


Figure 2.7: Photograph of dual processing element PCB (component side)

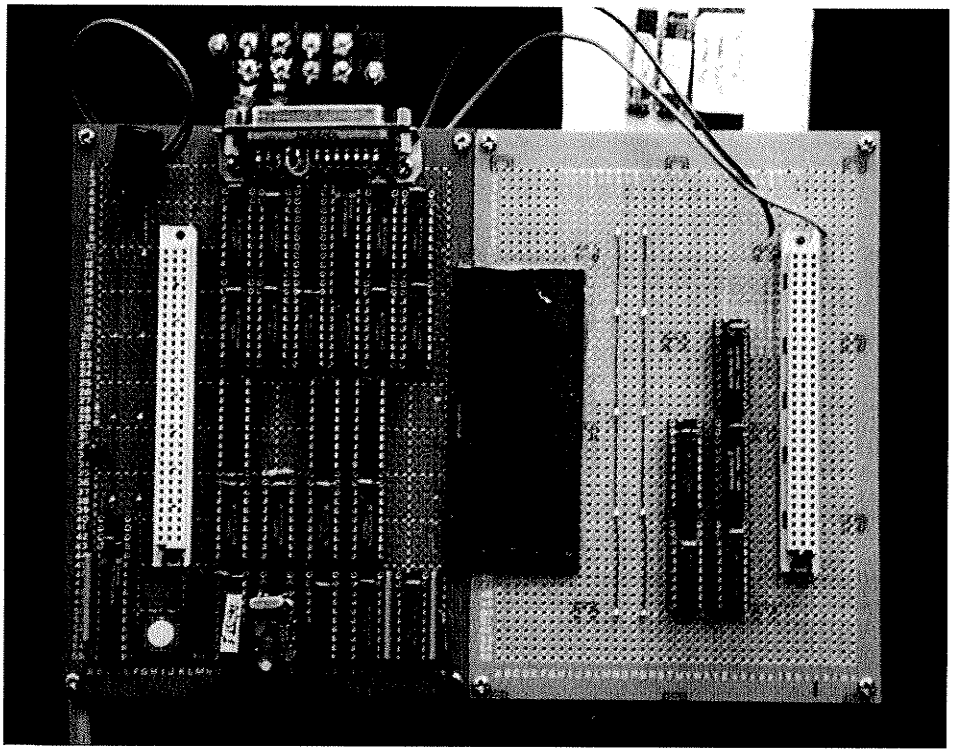


Figure 2.8: Photograph of tester for the message decoder

left). The bidirectional latch, between the two host input queues, allows the two processing elements to share information (shown centre right). The bidirectional latch bank between pre and post processors are shown at the centre, with the spare FPGA signal headers separating each bank. Each DRAM bank, located at each of the four corners, can accept DRAM SIMM modules ranging from 256KB to 16MB, for a maximum memory capacity of 64MBs per bank.

The bus lengths of all signals on each external DSP bus must be under 6.5 inches. In order to meet this requirement, the DSPs have been oriented on a 45 degree angle, as shown. A fully configured DPE can consume on the order of 20 to 30 Amperes at 5 volts, and is supplied through the two brass lugs shown left top and bottom in the figure. From the transmission-line problems experienced with the first three designs, this design employs many series resistor dampers, mounted on the solder side of the DPE (not shown). These precautions have eliminated the undesirable transmission-line effects, resulting in a successful implementation.

2.4 Hardware Commissioning

As mentioned earlier, modules were designed and constructed in order of increasing complexity. This, however, was not the ideal order for commissioning modules, and has compounded the number and severity of some errors encountered, while reducing others. As an example, the backplane could not be tested until the message decoder and user interface or DPE module were built. Further, some errors did not surface until several modules had been already built, most notably the transmission-line effects encountered. These unexpected and, in some cases, severe errors have impeded progress of the project, but in the end, design patches were devised to remedy all of these problems.

Throughout the course of commissioning the hardware, several specialized test jigs and sub-assemblies were constructed, to diagnose and correct the problems encountered. Two test jigs were constructed, one for commissioning the message decoder and one for complete modules (module with message decoder). After commissioning, these test jigs were again used as an application development platform for the many FPGA resources in the design. Discussions follow for the commissioning of the message decoder, the user interface module, and the dual processing element.

To commission the message decoder, a test jig was constructed from 26 IC packages as shown in Fig. 2.8. This test jig simulates the ultimate operating environment for the message decoder. It is interfaced to an IBM PC through one serial communications channel, a terminal program, and a small microcontroller on the test jig. Features include the ability to reconfigure the FPGA, to write messages to the input message queue, to receive messages from the message holding queue, to exercise the numerous sub-buses of the common bus and to clock the message decoder in varying ways with event feedback. This test jig has significantly accelerated the diagnosis and correction of errors.

Several minor design errors were found, requiring traces to be cut and moved. One concerned the misuse of the FPGA's LDC pin while others concerned inappropriate initial levels on device inputs. Several other modifications have been made since, to accommodate the FPGA design discussed in the next chapter.

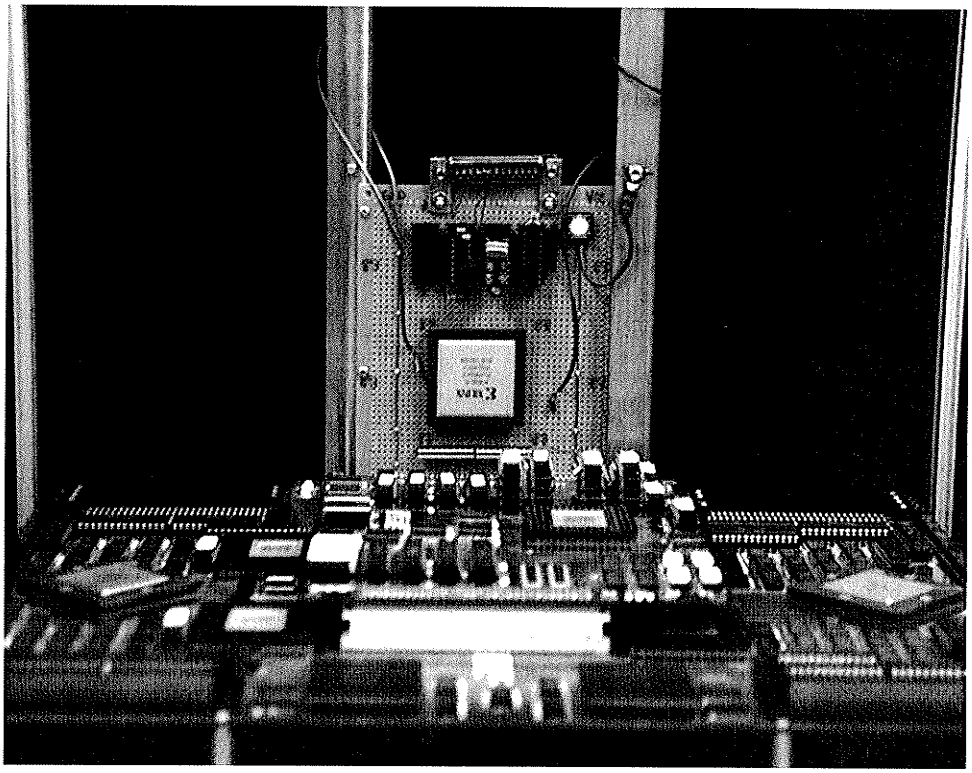


Figure 2.9: Photograph of three chip module tester with DPE mounted

To commission the user interface and DPE modules, another test jig was constructed, using only three IC packages this time (Fig. 2.9). This was accomplished by using an FPGA, instead of discrete ICs, to provide test functions; to date, only one quarter of the device is used to implement the tester. One advantage realized was that many specialized testing algorithms can be incorporated within the FPGA, to accelerate and customize testing. Again, an IBM PC was interfaced through a serial channel and a small microcontroller on the tester.

The user interface module had several minor logical errors, requiring two traces to be cut and moved, and one small PCB had to be designed to fix the CPU reset circuitry. The major problems encountered were transmission-line effects. Every signal of the CPU's address, data, and control buses displayed ringing effects, causing multiple problems. These problems were the worst encountered in the project, with some signal under-shoots measuring minus 5 volts! To alleviate these problems, two approaches were taken: a small PCB was constructed to terminate signals, and a small series resistor was added to the outputs of offending signal

buffers. These solutions have successfully quenched signal undershoot, to minus 0.3 volts throughout.

During the design of the DPE, many precautions were taken, in light of the transmission line effects experienced with the user interface module. As a result, no logical errors were found, and only one transmission-line problem was discovered, and quickly corrected. During the commissioning, a rudimentary TAP interface was developed and implemented in the FPGA resources to aid in testing. Several programs were written for the host DSPs, to exercise memory and information pathways.

2.5 Summary

The constructed hardware represents the single largest effort and expense of the present project, and has required over three man years to complete. In spite of the overwhelming transmission-line problems experienced, the hardware is now both stable and functional. However, much remains to be done concerning software interfaces, FPGA development, and ethernet communications. In the next chapter, a general application architecture is designed and implemented, using the flexibility of the FPGA resources to enable the execution of a wide variety of applications.

Chapter 3

FPGA Application Architectures

Through field programmable gate arrays (FPGAs), architectures well suited to varying application requirements may be built upon the physical hardware of the previous chapter. These application architectures allow specific algorithm concerns to be addressed more directly, leading to more effective solutions.

Several problems and dangers are readily apparent with hardware having the same freedoms associated with software. As the development of software and hardware co-design tools is still in its infancy, the dynamics of such an association are also in a relatively immature state. It is conceivable that without rigorous inter-dependency checks between hardware and software implementations, physical damage to components and systems may result. This task is further complicated by the fact that hardware and software have no common medium for specification and implementation. As a result, allowing applications their own unique architectures as opposed to generic architectures with wider application appeal may be premature at this time.

The present chapter identifies several structures which have wide application appeal, and it suggests an initial organization of tools to ease application architecture development. In the following sections, FPGA design tools are discussed, a general message decoding interface is defined and several application architectures are discussed and constructed. This is followed by operating system concerns and a summary.

3.1 FPGA Design Tools and Techniques

FPGA designs begin with a description from a schematic entry package or other source which generate an "xnf" output file format. From this file an "lca" file is created, ready for input to the XACT compiler from Xilinx. XACT compiles designs in two phases: place and route. The placement phase begins with random initial placement and through a simulated annealing process, the design is iterated to a local minimum placement. After placement, the design is routed by several auto-routing techniques until the design is either successfully routed or available routing algorithms are exhausted. It is worthwhile to note that this compiler does not guarantee success nor optimal placement of a design. However, through multiple compile attempts, better local minima for placement may be found, resulting in designs to be rerouted or timing to be improved.

The current design uses Xilinx 3000 series devices exclusively. These devices hinder (if not exclude) the use of traditional control sequence design methods. Microcoding is not an option since no internal RAM or ROM is available and wide finite state machine designs are discouraged by the limited fan-in capabilities of configurable logic blocks. To circumvent these restrictions, a custom design method was devised based on shift registers. By using shift registers for finite state machine design, a closer match with the FPGA architecture is realized. This reduces global routing constraints to primarily local ones; implementation is given a high-level language appearance and debugging tasks are greatly simplified. However, overall speed of these designs are reduced due to a precondition that related control states must be localized. This is offset by the ease of design and implementation. The interested reader is encouraged to examine a detailed description of this method presented in Appendix D.

3.2 Message Decoder

The global communication network structure is a broadcast message-passing architecture. Each bus module has a message decoder of similar physical construction to decode messages of interest to member processors. From previous discussions,

the efficiency of the communication network is heavily dependent upon the efficiency of this decoder. The following discussion defines a message decode which is not only efficient but also amenable to many computing model forms. The present message decoder provides a generalized communication structure and foundation upon which to build diverse application architectures.

A central problem to data flow, capability and object oriented machines has been the large overhead associated with the unique identification of components of a solution. Two methods have been presented to handle this problem: *time* and *space* unique identifiers. *Time* identifiers identify objects and data by assigning a unique time tag, while *space* identifiers identify components by assigning unique global virtual addresses. *Time* identifiers allow for efficient management of components through extensive mapping memories, while *space* identifiers require complex garbage collection routines. Neither of these approaches have been satisfactorily proved or disproved to be superior to the other, but there is interest in finding a third method [24].

It is clear that if a message decoder is to be widely amenable to many computation forms it must not only identify messages but must classify them as well. Common graphical user interface software like "Windows" from Microsoft demonstrates many of these desired traits. Input, depending on its source, is classified and directed implicitly to handle routines for processing, and is called an event. These event classifications are known prior to execution and are assigned standard handles. This is not the case with general processing where new objects and data may be created during the course of application execution. This identification mechanism could be easily implemented in software, but this approach would limit any advantage through inherent inefficiencies. This has led to the desire to implement identification and classification of messages within hardware.

To address application communication requirements efficiently, diverse communication topologies must be provided by the network. This implies that the present network must implement some form of selective broadcast mechanism. From previous discussions, messages are identified by an identifier at the head of a message. This identifier could be used to identify different communication topologies, contained data and local handler routines for applications. It is also

apparent that only a small number of these *global* messages (messages transmitted on the backplane) will be required by any individual module on the bus. Hence, the size of global message identifiers should be larger than local identifiers to reduce the size of local handle indices or mapping memories. To achieve this, some hash algorithm or associative memory is required to map a large sparse space to a smaller dense space uniquely.

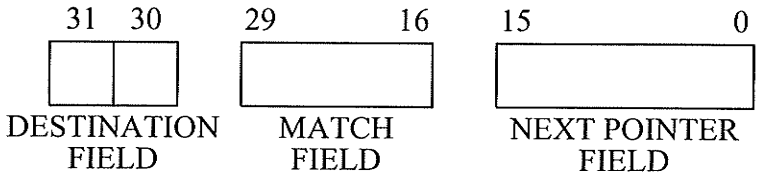
Several solutions were studied including content addressable memories but, given their immature state and high cost, other methods were sought. A simple hash algorithm was devised with high speed, a small memory size, implicit garbage collection and excellent scalability. The table size is governed by four variables: the size of local space 2^n, number of member hosts d, number of bits in the global identification tag g, and table word width m (in bits). Hash table width is established by Eqn. 3.1.

$$m = d + g + 2 \tag{3.1}$$

The hash algorithm makes use of a number of circular linked lists or search rings equal to one quarter of the local space size. A corresponding number of bits from the global identifier to be matched is used to index rings to be searched. The remaining bits from the global identifier are compared with the match field of each element in a search ring until a match is found or the ring is exhausted. If a match is found, the message is authenticated and the address of the memory location where the match was found becomes the local message or handle identifier. Otherwise, authentication fails and the message is deleted.

Each element of a search ring has a match, destination, and next pointer field as shown in Fig. 3.1 (for values $n = 16, m = 32, d = 2$ and $g = 28$). The next pointer field size is equal to n, the destination field size is equal to d and the match field size is equal to $g - n + 2$. By choosing different numbers for Eqn. 3.1, larger or smaller hash tables with different attributes may be created. It is important to note that the same hash algorithm is used for any size of local space with guaranteed success and without any loss in authentication speed.

Search rings are initialized by setting the destination field to zero and setting the next pointer to point to itself to form search rings of unit size. The remaining



DESTINATION FIELD		DESCRIPTION
31	30	
0	0	INVALID TAG ENTRY
0	1	VALID MODULE A
1	0	VALID MODULE B
1	1	VALID MODULES A&B

Figure 3.1: Structure of hash table entry.

3/4 of the space is initialized into one large circular linked list, which is called the null ring. The topmost location is reserved for the null ring pointer, and cannot be used to match an identifier. A search ring grows by taking an element from the null ring and inserting it into its own ring. Similarly, to delete an element from a search ring it is removed from the ring and added to the null ring. In this way garbage collection becomes implicit. If identifiers to be matched are equally distributed across all the search rings, no search ring will be greater than four in size and the hash algorithm will require no more than four searches to authenticate any message identifier.

The hash algorithm or associative memory processor requires at least four instructions; reset, add, delete, and search. *Reset* initializes the null and search rings. *Add* installs a new global identifier into the local space. *Delete* similarly removes a global identifier from local space. Finally, *search* looks for the occurrence of a global identifier in the local space. This algorithm has been incorporated within the message decoder's FPGA and forms a generic component for the application architectures described in the next section.

3.3 Application Architectures

From the two application areas examined (*i*) power system simulations [1,3-9,11-19,20,22,25,26,30,34-36,40-42,44-46,49,50,57,60-63,68,69,70,71,75-77] and (*ii*) neural networks [27,38,56], two computing models were found to predominate: a control flow model and a data driven model. In addition, both application areas are readily adapted to the current broadcast communication structure with only minor transformations. These two computing paradigms form the basis for the application architectures discussed and implemented here.

These application architectures use the message decoder defined above as a common component, and are restricted to processing elements of the DPE module. Each architecture is constructed from a combination of FPGA resources, the pre and post processors, and DSP software (refer to Fig. 2.3). Four generic application architectures are provided here; control flow, data driven, demand driven, and possible hybrids of computing paradigms.

Several common components are included in each implementation, to allow the DSP boot code to be loaded, memories to be initialized and messages to be shuffled among the various processing zones. To load DSP boot code, a DMA controller is implemented with the address counter doubling as the address source for DRAM accesses. After FPGA configuration, a reset routine within the FPGAs initializes memories and FIFOs to known states in preparation for applications. Each component design is constructed in a way that, if possible, it serves a double function to conserve valuable internal FPGA resources.

The following implementations are *fragile*, where no error detection or handling circuitry has been included, due the size limitations of the current hardware. If errors are presented to these implementations, operation may be unpredictable.

3.3.1 Control Flow Model

Many possible implementations of the control flow computing model exist using the present hardware. Each possibility is slightly different and can influence the implementation of application algorithms. The main objective of the control flow model implementation is to provide a general computing model, which is

amenable to several algorithms and is simple to implement. One limitation of the present implementation is that it restricts individual processing elements to one application.

A majority of the required FPGA hardware is implemented within the pre-processor FPGA. This hardware is responsible for loading the operating system and application code into DSP memory, and releasing the DSP from reset. After this, it simply passes any incoming messages to the job input queue, and generates a processor interrupt to signal that a new message has arrived. The DSP receives incoming messages from the job input queue, decodes which sub-function of the application to direct data to (through software), and passes any results to the job output queue. The post-processor simply forwards any messages from the job output queue to the message output queue, completing an execution request. This implementation is very simple and efficient.

Another possible implementation uses the DRAM resource of the pre-processor to provide a vector table for incoming messages. By using the local identifier to index this vector table, the entry point of service routines are passed directly to the DSP for execution. This removes the overhead associated with determining target service routines from software, at a cost of increased FPGA hardware complexity. One added advantage of this scheme is that individual processing elements can easily accommodate multiple applications, using only a small amount of software, and without incurring additional overhead.

3.3.2 Data Flow Model

To begin, a short review of data flow concepts, forms and notation is in order. Data flow programs are written such that the order of operations is not explicitly specified, but is implicitly derived from the data interdependencies. Two forms of data flow models are presently discussed: demand driven and data driven.

In data driven forms of data flow computation, a function node is fired if tokens (or messages) are present on all of its inputs and no token (or result) is present on the output. Control nodes are specified by switch and merge functions. The switch function has one input and two outputs. The input token is placed on one of the two outputs, depending on the value of a control input. Similarly, the merge

function places one of two input tokens on the output, depending on a control input. With these firing rules, more complex operations can be constructed, such as conditionals and loops [58].

More suitable or easily verified languages are the Functional Languages, where each node output is a function of its inputs. In these languages, solution graphs must be acyclic, and are a particularly good match for sparse matrix solutions. Two forms of computing have been derived from these languages: data driven and demand driven. Demand driven forms use the request of a variable to invoke the necessary computations to provide it.

3.3.3 Data Driven Model

The first implementation attempt adheres to the rules of functional languages with a data driven form. This was considered to be the easiest implementation, and the most useful considering its wide application to many problems. To implement the data driven model, all of the resources available to the pre-processor, processor and post-processor are required. In addition, the message decoder requires a *bind* instruction to translate multiple global tags to local ones.

Each function node has a code segment, a varying number of input variables and an output. The present approach is to allocate each of these components separately, and then bind them together to form function nodes. This requires a flexible structure. Considering each node has a variable number of inputs, individual inputs may be the source for multiple function nodes, and multiple function nodes may share a common code segment.

The pre-processor's DRAM is divided into two parts: a component attribute table, and a data storage heap. Three types of components are defined: code segment, input variable and function node. Each individual component is created separately, by adding a local identifier or tag to a destination processing element (through message decoder instructions). This creates an entry in the attribute table for this local tag with the appropriate attributes. If the component is a code segment, the attribute table contains an address of the code segment's entry point within DSP memory. If the component is an input variable, a pointer to the input data within the data storage heap is provided, and if the component

is a function node, a pointer is provided to a list of the function node's member components (stored within the data storage heap). When a new input variable message arrives the pre-processor, signals the post-processor and stores the new variable data into the data storage heap.

The post-processor is responsible for enforcing function node firing rules. To accomplish this, its DRAM is organized into link lists. Only function node and input variable identifiers are included in these lists, to associate inputs with functions. For each input of each function node, an input instance is created. Instances of the same input variable are joined into a *horizontal* linked list. Member input instances of function nodes are joined into *vertical* circular linked lists that intersect with the *horizontal* ones. When an input arrival is signaled by the pre-processor, a bit is set in each instance along the *horizontal* axis, signaling that a new value is present. Subsequently, each instance of the present *horizontal* chain is searched in the *vertical* direction (member inputs of a function node) and, if every input instance along the *vertical* axis has a new value, the function node is poised for firing.

To fire a function node, the local tag of the desired node is passed to the pre-processor with a request to fire. The function node's member list is accessed and an execution frame is assembled on the job input queue. The processor is signaled, the function is executed, and results are passed to the message output queue through the job output queue. At the same time, the post-processor resets each *new value* bit in the present *vertical* chain and proceeds to the next *vertical* chain (along the current *horizontal*), until all are exhausted, completing the fire rule check.

Many inefficiencies with the above implementation are readily apparent. Firing rules are processed in a slow and complicated manner, function node management is cumbersome, and operating system software is overly complex. However, if the problem is re-organized to define a "trigger" word for each function node, such that when all the bits of this word are set, the function node is fired, a more efficient firing rule may result.

In this scheme, each input instance defines a bit offset to access the individual bits of a function node's trigger word. These bits are then set as new inputs arrive

until the trigger word contains all ones, upon which the function is fired. To reset the trigger word, a reset trigger word is simply written to the trigger word location. This reduces the *vertical* circular link lists of the previous implementation to linear link lists, and the firing rule check algorithm to $O(n)$ instead of $O(n^2)$ time complexity. This function node installation routine is more complex than the previous definition, but given the potential increase in execution time, this is an agreeable tradeoff.

3.3.4 Demand Driven Model

To implement this model, concepts from a new class of computation structure, "transport-triggered architectures" [47] are employed. In these architectures only one instruction is provided, the "move" instruction. A number of function units (adders, multipliers, etc) are defined, each with three registers: an operand, a trigger and a result register. An operand is first moved into the operand register and a second operand into the trigger register. The act of moving a variable into the trigger register "triggers" the function unit to provide a result.

This trigger or firing rule can be used to implement a demand driven model of execution, where one or more inputs of a function node are given this *trigger* attribute. Using the FPD configurations of the above data driven model as a basis for this model, the memory structure of the pre-processor is unaffected, and only minor changes are made to that of the post-processor. In brief, components are allocated and bound as before with one subtle difference: during the binding process, individual input instances (items of the *horizontal* lists) can be defined to be *trigger* inputs. By assigning trigger attributes to input instances, instead of input components, greater application freedom is achieved. Now, a simple search of an input variable's *horizontal* list for instances marked as *trigger* determine which nodes are fired. In this way, requests for a result can trigger a chain of function nodes to provide it.

3.3.5 Hybrid Computing Paradigms

Many different hybrid computing models are possible with the present hardware. Encoding context information (or model selection) within messages and/or implemented structures are viable options in forming hybrids. The *trigger* structure implemented in the previous section is essentially a generalized version of the control flow paradigm. This structure is suitable for executing object oriented applications and other applications that match a control flow model. The data driven model implementation can be readily merged with the previous *trigger* mechanism to form a control flow, data driven, and demand driven computing model hybrid.

The present hardware yields a flexibility not previously available to the application engineer or programmer. The many application tradeoffs presented will require further study before an optimal hardware architecture can be determined.

3.4 Operating System Concerns

In past designs, the operating system was composed strictly from software; this is not the case with the current system. Hardware plays an active role in operating system construction. As seen in the previous sections, hardware assumes a number of tasks which are associated with software operating systems. Indeed, a case could be made that this hardware constitutes the operating system, with only minor functions being implemented in software. Higher level operating functions for installing and removing functions is possible using these structures. As an example, a chain of function nodes can be deleted with only one command, where this delete command is propagated to each successive node in the chain using the links that bind them. This also raises the possibility of applications creating their own chains of function nodes. Again, this is a design tradeoff between software and hardware, in which application requirements, hardware availability, and design complexity all play a part in tradeoff decisions.

To date, the software written to supplement the hardware operating system structures has been less than 512 words in size. This combination of software and hardware has achieved a high performance operating system, which supports

basic functions with very low overhead. The more complicated operating system functions such as user and file I/O are implemented on bus master modules, where software support is more readily available.

3.5 Summary

Throughout the course of these discussions it has become evident that the present hardware structures are flexible, with many possible options for satisfying application requirements. Several application implementation issues, which relate to system design tradeoff decisions have been introduced. The notion that the operating system can be encoded within hardware offers some very interesting perspectives. The current FPGA devices are relatively small in comparison to the devices now available (in excess of 50K gates). With these next generation devices, full operating system implementations may be possible within hardware.

In the next chapter, the application architectures discussed here are applied to a number of neural network applications. Particular attention is given to the partitioning of these solutions on the present multicomputer, and to the extent to which application architectures influence these decisions.

Chapter 4

Applications

The backpropagation learning algorithm is an important advancement to the field of artificial neural networks. Both analog and digital implementations are progressing, with digital implementations prevalent so far. However, digital implementations suffer from size limitations and from long learning times imposed by inadequate computing resources, which often restricts them primarily to problems of academic interest. Another artificial neural network of interest is the self-organizing feature map, with applications in the vector quantization of speech and video. Both of these algorithms require substantial computation for learning. However, through parallel processing these computation requirements may be met. In this chapter, these two neural network applications are discussed and implementation issues are addressed.

4.1 Backpropagation Networks

The general form of the backpropagation network consists of an input layer, one or more hidden layers, and an output layer of nodes (see Fig. 4.1). Complete bipartite directed graphs are constructed between each adjacent layer of nodes using weighted connections. Data is presented to the input layer, and each component is multiplied by a weight and summed at nodes of the connecting layer. These weighted sums are then passed through a reversible nonlinear transfer function forming input to the following layer. This procedure is repeated until reaching the output layer, completing a forward pass. The interconnecting weights between layers are trained using a collection of paired input and desired output

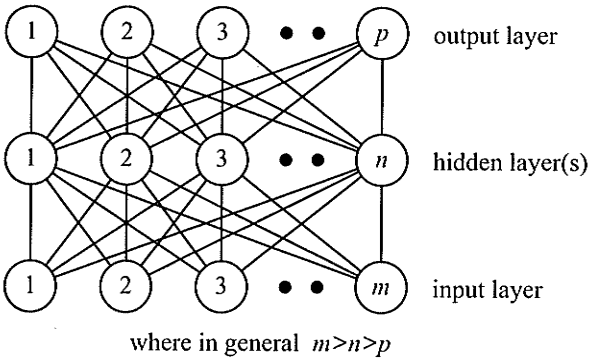


Figure 4.1: Backpropagation network topology

values forming a training set. Each case from the training set is presented to the network and a *forward pass* is performed. This output is compared to the desired output and an error vector is created. These errors are then *backpropagated* through the network adjusting connection weights according to the well-known backpropagation algorithm [56]. After many iterations of the training set the connection weights settle to a local minimum of the output error over the training set. By repeated training with randomly selected initial connection weights, better minima may often be found.

This solution can be highly parallelized by taking advantage of the bipartite nature of these networks. Assigning a processor to each node of a n node layer reduces the required computation time by a factor of n. The traditional drawback of parallel processing is the inter-processor communication bottleneck. This is easily addressed for backpropagation networks by means of a broadcast bus for communication, since each output of a given layer is broadcast to each node of the next layer. On the reverse (learning) pass, the error derivatives may be similarly broadcast.

By choosing a digital signal processor as the node processor, high computational efficiency is achieved. This is due to the algorithm's requirement for weighted sums, which is an identical operation to that of the finite impulse response filter, an algorithm the instruction set of DSPs (in particular the single cycle multiply-accumulate instruction) were originally designed to perform efficiently.

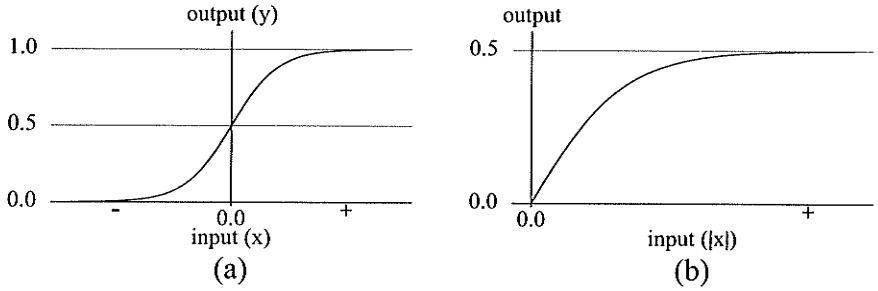


Figure 4.2: Sigmoid transfer function and lookup table

4.1.1 Sigmoid Transfer Function

The sigmoid transfer function expressed by Eqn. 4.1 represents a significant portion of the computational load. To reduce this load, a lookup table was examined as an alternate method. A number of discrete values of the function are tabled and a lookup routine accesses this table, either returning the closest value or doing linear interpolation between entries.

$$y = \frac{1}{1 + e^{-x}} \tag{4.1}$$

First consider a plot of the sigmoid function (Fig. 4.2a). Notice the symmetry about the output axis; this reduces the required table size by half. Observing this symmetry, the function to be tabled is shown in Fig. 4.2b, and the original transfer function is easily reconstructed using the following pseudo-code segment;

$$y = 0.5 + Sign(x)Lookup(|x|) \tag{4.2}$$

This method was initially implemented on a Motorola 56000 DSP using a table size of 1024. Two methods of table lookup were examined: one a direct lookup method and the other with linear interpolation. While the direct lookup method is faster by 6 clock cycles the accuracy is somewhat less than the 2 ppm error recorded by the linear interpolation method. Another method considered was patching quadratic functions together to yield the desired curve. Although this method executed properly, it required more clock cycles and yielded a maximum error of 7 ppm using 50 quadratic segments (150 table entries).

Both execution speed and accuracy are important parameters in determining which method should be used. It requires 30 clock cycles to compute the sigmoid

using linear interpolation and 2 clock cycles to perform one connection (one multiplication and one addition). Therefore, 15 connections can be done for each sigmoid. This gives a measure of computational efficiency and indicates the size of networks that map efficiently onto one processor.

Using the present Motorola 96000 DSP to calculate Eqn. 4.1 directly, requires 80 clock cycles (or 2 *us* @ 40MHz). This method is simpler to implement on the present architecture, avoiding algorithm complexity and space constraints associated with tables. However, both methods are suitable for the current application, which offers a choice of implementation to better match appication requirements. Other possibilities exist for implementing the sigmoid transfer function including specialized hardware, which achieves potentially higher speeds and reduces the connection sigmoid ratio.

4.1.2 Network Task Allocation

Task scheduling for a parallel processor is difficult for some applications, but for the backpropagation algorithm it is straightforward. This is due in part to the bipartite nature of the network graph. Consider a parallel processor with n processing elements and, referring to Figure 4.1, assign a processor to each node of the hidden layer. Notice no processor has a connection that overlaps with any other processor since the network is bipartite. This provides a completely balanced task schedule, provided a 1 to j mapping between processors and hidden layer nodes is made, where j is some integer ≥ 1.

Continue this task assignment for each hidden layer and the output layer. If the number of processors do not match the number of nodes on a layer, assign the processors in round robin fashion until each node has a processor. Now consider the connection and node assignments of processor number two as shown in Fig. 4.3. Solid connections are used for the *forward pass*, and outlined connections for backpropagating errors during learning.

The manner in which the network solution progresses can easily be seen. An input vector is presented to the input layer and hidden layer node processors preform a weighted sum of these inputs. This value is passed through the sigmoid transfer function and the result is communicated to every other processor. After

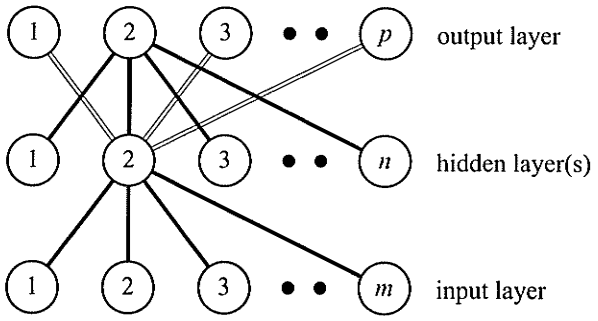


Figure 4.3: Processor Two Network Task Allocation

this exchange the process is repeated with the next hidden layer until the output layer is reached, completing a *forward pass.*

If the network is in recall mode, these outputs are communicated to the user or are passed as inputs to other networks. In learning mode, these outputs are compared to the desired outputs, forming an error vector which is exchanged with all processors. Using the outlined weight connectionsi, these errors are backpropagated through the network until reaching the first hidden layer. Each connection is updated, including the outlined connections, completing the *backward pass.*

Note that, in learning mode, this task allocation scheme has assigned the weight connections between hidden layers and the output layer twice. This means that, during learning, all these weights must be updated and the new outlined weights must be communicated to all other nodes, increasing considerably the computational and communication load for learning. But since this is not required for the weights between the input and the first hidden layer, and in most cases $m > n > p$, (Fig. 4.1) this effect is reduced.

One particularly limiting characteristic of message-passing architectures is the fact the order in which messages arrive is unpredictable. This is unlike the communication network presented in Appendix A, where messages are explicitly ordered at compile time. Since the backpropagation algorithm has the general form of a vector problem, it may suffer under message-passing architectures. This poses problems with the present allocation scheme, where results must be put into vector form before they are passed to following layers. However, with the present FPGA hardware, this may be accomplished through a data driven architecture.

The present allocation scheme, if implemented on the present hardware for the backpropagation algorithm, would be complex and inefficient. For learning, a more efficient allocation scheme may be to implement complete backpropagation networks on several processors, and have each learn a subset of the overall training set [32]. After several passes through each training subset, the complete weight space is averaged amongst all the processors, and a global error measure is recorded using the entire training set. This process is repeated until the global error is below some preset bound.

This allocation scheme is generally referred to as training set parallelism, and has several advantages over the previous one. Communication requirements are drastically reduced, to a single large communication at the end of each training subset epoch. Computation by each processor is reduced by eliminating redundant computations (duplicated weights between hidden layers and output layer), and operating system overhead is reduced by a decrease in message traffic.

4.2 Self-Organizing Feature Maps

The self-organizing feature map (SOFM) algorithm is an unsupervised learning algorithm belonging to a class of algorithms known as competitive learning (CL). The main objective of CL algorithms are to classify an input data set into a representative set of categories. Neurons are *placed* in positions that minimize, on average, the distance between the weight vector of the closest neuron and input vectors. A neuron with the minimum distance to an input is called the "winner" and is representative of this input.

Many forms of competitive learning exist: *hard*, *soft*, *frequency sensitive*, etc. The main difference between these forms is the learning rule. Each form classifies input data in a slightly different way, and influences the choice of algorithm employed in applications. SOFMs are closely related to *soft* competitive learning algorithms. Unlike *hard* competitive learning algorithms, where only the "winner" is updated, *soft* competitive learning updates the "winner" as well as a select neighbourhood about the "winner". In self-organizing feature maps, a predetermined order or topology of neurons is established, such that during learning this order is preserved or encouraged. The most well known SOFM algorithms are by

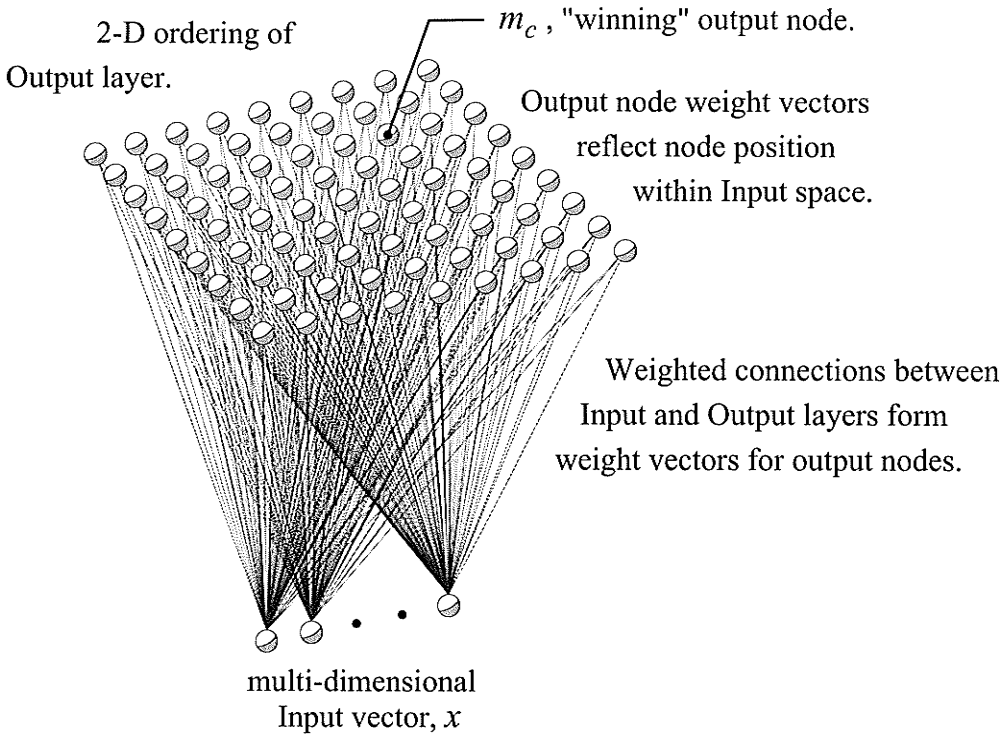


Figure 4.4: An example SOFM construction.

Kohonen [38].

4.2.1 Kohonen SOFM

The general construction of a SOFM with a two dimensional output layer is shown in Fig. 4.4. A multi-dimensional input vector is connected to every node on the output layer by weighted connections. These connections form a weight vector for each output node, and have the same dimension as the input. Each output node also has an index denoting its position within the output layer. If the input vector dimension is greater than the output layer dimension, the algorithm attempts to perform dimensionality reduction. The closest weight vector m_c to input vector x is found by Eqn. 4.3,

$$\|x - m_c\| = \min_i \{\|x - m_i\|\} \tag{4.3}$$

where, c denotes the "winning" node index and centre of the update neighbourhood. Typically, the Euclidean metric is used to determine winning nodes, but other metrics may be defined to provide better solutions. To determine

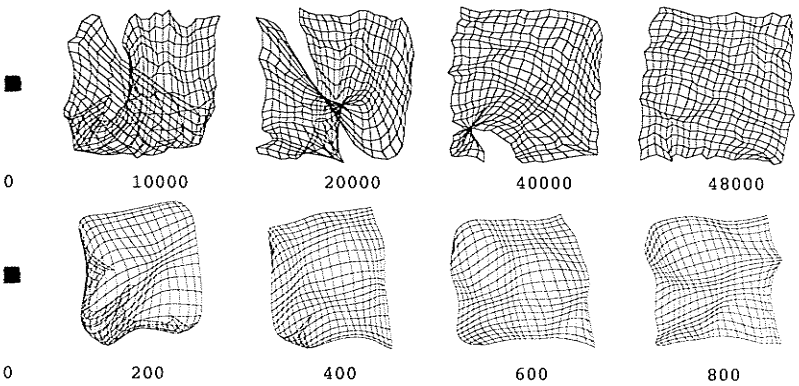


Figure 4.5: Comparison of step-wise neighbourhood (upper row) and continuous neighbourhood (lower row) methods for SOFMs.

the "winner", an exhaustive search of output weight space is required, which represents a large computational load.

Weights are updated according to the discrete time version of Eqn. 4.4 below,

$$m_i(t+1) = m_i(t) + h_{ci}(t)[x(t) - m_i(t)] \quad \text{, for all } i. \tag{4.4}$$

where, $h_{ci}(t)$ defines a time dependent scalar function with two components: the learning rate, and the update neighbourhood. The learning rate component of $h_{ci}(t)$ is solely dependent on time, and is a monotonically decreasing function, with values between 1 and 0. The neighbourhood function is dependent on both the index of output nodes in relation to the winner's index, and the time. Two methods have been cited by Kohonen: step-wise and continuous neighbourhoods. The step-wise method returns a one if an output node is within the update neighbourhood, and a zero otherwise. This update neighbourhood is reduced with time in discrete integral steps (or in a step-wise manner). The continuous method defines a function that returns values between 0 and 1, where larger values are returned for node indices closer to the winner's, and smaller values for those further away. Typically, continuous neighbourhood functions are defined as a gaussian, where σ becomes progressively smaller with time. Both methods begin with a large update neighbourhood, to provide an initial global order for output nodes, and as time progresses, this neighbourhood is reduced to improve the spatial resolution of output nodes.

Continuous neighbourhood functions accelerate the global ordering of net-

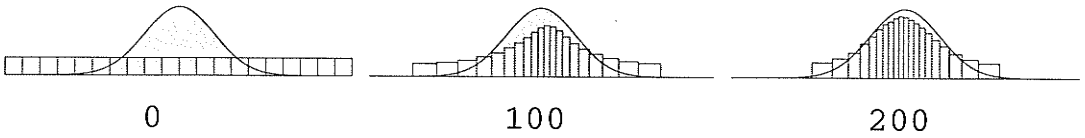


Figure 4.6: Example SOFM network approximates input probability distribution.

works, when compared with step-wise neighbourhood functions. By scaling the learning rate with respect to node index, network topology is directly reinforced. This is unlike step-wise neighbourhoods, which rely on the random nature of input data to unravel twists and kinks in the network topology. This is easily demonstrated by modelling two networks identical in every way except for the neighbourhood function. In Fig. 4.5 a network with a step-wise neighbourhood is shown on the top row, and a continuous neighbourhood function on the bottom row. Each network begins with the same set of random weights, and each is presented with the same input data. The step-wise case requires 48000 updates to unravel the network topology, while the continuous case only requires 800. One also notices a discontinuous appearance for the step-wise network, in constrast with the *smooth* appearance of the continuous network.

Another easily demonstrated attribute of SOFMs is their tendency to approximate the probability distribution of input data. Consider a network with the input nodes, output weights, and output layer all defined in one dimension. Weights are initialized such that they are equally spaced over some region, and an equal probability density is assigned to each space between adjacent nodes. Then, by dividing this density by the distance between adjacent nodes, rectangles are constructed with equal area. The shaded areas of Fig. 4.6 represent the input data probability distribution, and the rectangles represent the probability distribution of the output weights. As learning progresses, output weights assume positions that approximate the input data distribution. To help guard against a possible feature correlation with the Gaussian input distribution, the neighbourhood was chosen as a linear decay function.

SOFMs also have a number of discouraging attributes. The solution is highly dependent on network construction, from the choice of initial weights to the order

of presentation of the input vectors. While varying networks manage to extract similar features from common input, the number of updates required and learning parameters may vary radically between solutions. This is a common attribute of artificial neural network (ANN) solutions, where the choice of an ANN and its construction strongly depends on the application. The current SOFM algorithms are restricted to inputs with stationary probability distributions, as a consequence of the algorithm's time dependence. Further, output nodes tend to gather at positions with densities higher than that represented by the input. This poses problems to vector quantization applications, where several weight vectors may have similar values. This leads to codeword entries which are non-representative of the local probability distribution, and as a result, codeword utilization of the codebook is non-uniform [39].

4.2.2 Parallel Implementation

The SOFM learning computations may be partitioned among several processors, by assigning each processor to an equal number of output nodes. To allow arbitrary assignment of output nodes to processors, each output weight vector includes the output index. Each processor is presented with the same input data vector and a search for the best *local* match is performed. To determine the global winner either a logarithmic or linear search of local winners may be performed. However, given a broadcast communication structure, the logarithmic search would be of little advantage. Instead, assigning one processor to perform a linear search of local winners provides good performance and is simple to implement (for a small number of processors). After the global winner is decided, its index is broadcast to all processors with an update command. Each processor updates its assigned output nodes and the network is ready for a new input. This solution uses a control flow computing model implemented within the available FPGA resources.

A pseudo benchmark of SOFMs is learning to represent a uniform probability distribution on a two dimensional map or network. This problem was performed on the present architecture using a network of 588 nodes. A Gaussian distribution function was used for the neighbourhood function and all output nodes were

updated for each input sample. The time required to perform 500 network updates with one processor was 1.308 seconds, and with two processors was 0.684 seconds. This translates to 2.616 ms per input sample for the unit processor case, and 1.368 ms per input sample for the dual processor case. At present only one DPE is available, which has prevented further scaling of the problem. Results from these network solutions were recorded onto video tape via the real-time video interface described above, and these videos were found to offer interesting insights into the progression of SOFM solutions.

4.2.3 A Dynamic Learning Rule for SOFMs

As mentioned in the previous section, SOFM algorithms to date have learning rules which are time dependent. This limits their application to inputs with stationary probability distributions. In this section, a time independent learning rule is presented, which not only allows inputs to have non-stationary statistics, but simplifies selection of learning parameters. Further, control over the maximum density of output nodes is provided, which may aid in the solution of vector quantization problems.

Winners for the dynamic SOFM algorithm are determined as before Eqn. 4.3, and the new learning rule (in discrete time notation) is shown in Eqn. 4.5.

$$m_i(t+1) = m_i(t)+[x(t)-m_i(t)]\alpha e^{-\frac{\|r_i-r_c\|^2}{\sigma^2(x(t),m_c(t))}}+\rho(h_i-m_i(t)) \quad \text{, for all } i. \quad (4.5)$$

The learning rate is taken to be as a constant, α. The neighbourhood function is a Gaussian distribution with standard deviation, σ, defined as a function of distance between the input vector x and the associated "winning" weight vector m_c. The further away an input vector is from the winning weight vector, the larger σ becomes, thereby widening the update neighbourhood. Output nodes are constructed as before, but with the addition of a *home* weight vector. The *home* weight vector values are constant throughout learning and are set to the initial values given to corresponding output weights. A weight decay term is added to the learning rule, which allow outputs to *decay* toward home positions (h) during periods of inactivity. The decay rate, ρ is defined as a constant. After learning, the *home* weight vectors are not required, and may be removed.

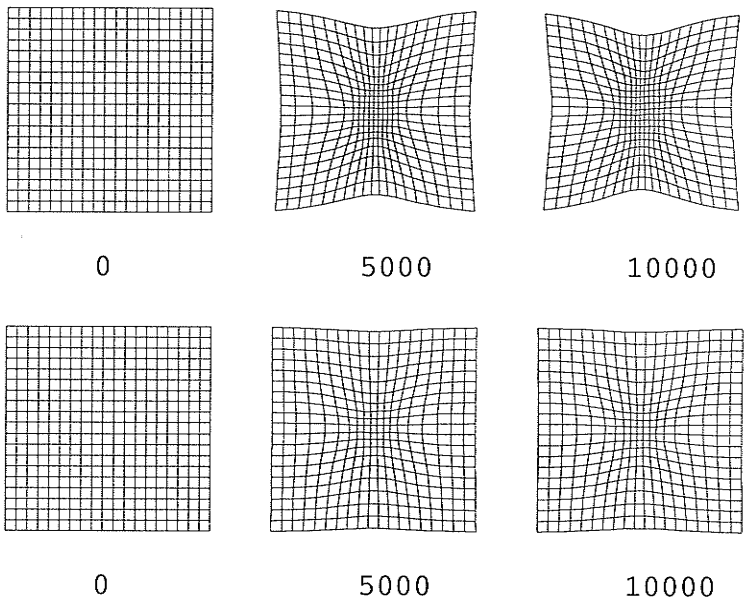


Figure 4.7: Effect of learning rate to decay rate ratio on output node density.

Two parameters control the learning characteristics of this SOFM: the learning rate and the decay rate. The learning rate controls how quickly nodes assume input vector positions, and the decay rate controls the persistence of output node weights. The ratio between learning and decay rates controls the number of output nodes allowed in a region. The higher the learning rate : decay rate ratio, the greater output node density allowed. To demonstrate this, two solutions are shown in Fig. 4.7, where the learning rate for both cases is set to 0.01. The top row solution uses a ratio of 200:1 or a ρ of 0.00005, while the bottom solution has a ratio of 20:1 ($\rho = 0.0005$). Both solutions converge by step 5000, and no appreciable change is evident by step 10000. Notice, the top solution has a higher density of output nodes than the bottom solution. This density does not increase with further network updates as in previous implementations. This learning rule may be applied to vector quantization problems, and may provide a more uniform codeword usage for codebooks.

Enhanced spatial resolution for stationary probability inputs may be achieved by smaller learning rates. This increases resolution, but at the cost of increased learning time. To compensate, learning and decay rates may be reduced (in a fixed ratio) with time. For inputs with dynamic probability distributions, a trade-

off decision between "tracking speed" of the network and spatial resolution is required, and selection is application dependent.

As a practical application, this learning rule may be useful in determining dynamic load schedules for parallel networks. For non-deterministic problems, the flow of solution is often dependent upon input data or interim results, which can pose problems to load scheduling. Further, these effects may not be predictable prior to execution. With the current SOFM algorithm, processor loads can be tracked as the solution progresses and may provide better information to dynamic load schedulers.

The present dynamic learning rule is implemented on the current architecture using a control flow computing model. Task allocation and execution times are both similar to the time dependent implementation. Several initial test cases have been run on the present architecture, and video recordings were made.

4.3 Summary

Although the results presented here are preliminary for applications, the present architecture has demonstrated a flexibility not present on other architectures. Both sample problems presented have a number of possible implementations, each with its own mix of hardware and software. As applications become more well defined, the mix between hardware and software may be tuned to yield higher performance, or alternatively to reduce costs. The present architecture has many advantages in prototyping solutions for product development.

Chapter 5

Conclusions

Field programmable devices (FPDs) can be an invaluable resource to multicomputers and large systems in general. The flexibility of these devices aid in the design, debugging, and application of large systems.

Design tradeoff decisions are eased, where only capability requirements are initially specified through pin placement and generic FPD pin-out selection. This is particularly advantageous with PAL designs, where the number of pin compatible devices available are the largest. The actual internal implementation of FPD resources need not be specified in the initial stages of design, and allow internal designs to change with changing system specifications. An opportunity to parallelize portions of the product development design cycle is also provided, where PCB designs may proceed in parallel with FPD internal design and selection. Further, design life cycle may be enhanced through FPD configuration upgrades or device replacement.

When systems are designed with both PALs and FPGAs, a number of performance enhancing options are possible. Higher clock speeds of PAL devices coupled with a small PCB footprint, make PALs an ideal choice for small high-speed tasks. With smaller footprints than FPGAs, PALs can be placed closer to where they are needed, to reduce high-speed trace lengths. FPGAs, on the other hand, can incorporate large amounts of decision logic. By using FPGAs to make high-level decisions and having PALs carry out these decisions, an agreeable design mix may be achieved.

Through FPDs, system test and debug tasks are accelerated. The available

FPD resources can be configured as self-test tools to provide stimulus and/or record results of troubled areas. Test applications may be designed to focus on specific problems, which are difficult to diagnose by more conventional test equipment. Further, these testing strategies can be accomplished without incurring additional hardware costs, by selection of re-programmable devices.

Systems designed with FPDs enable rapid prototyping. If initial prototypes are constructed with a large number of FPDs, prototyping design tradeoffs are possible. The design tradeoffs associated with a mix of software and hardware for applications can be analyzed more fully, resulting in final designs which have an optimum balance between cost and function. Initial applications may begin as software and gradually be moved to hardware to meet changing performance requirements.

In the present system, FPDs coupled with a four zoned task queue structure for processing elements have yielded a system with surprising flexibility. Through reconfiguration of the FPD task queue resources, many computing paradigms may be implemented, including hybrid models. Further, enhancements to communication transfer rates and/or computation rates are also possible. The present thesis has only presented preliminary results for a select few applications; however, a potentially wide application scope is evident.

Of the several problems encountered during implementation, the transmission-line effect problems were the worst. These effects were under-estimated during the design phase, and are the single most detracting aspect of the current project. Without the use of transmission-line analysis tools many of these problems were unpredictable prior to manufacture. However, with several specially designed ring-dampeners and terminators these problems were satisfactorily solved.

From this practical large system design study, a number of unique and interesting design practices, testing, and architectural flexibility issues related to FPDs have been brought to light.

Contributions of this thesis include:

i) the design and implementation of a parallel DSP machine with extensive reprogrammable logic devices. The initial design was first presented at *The Canadian Conference on Electrical and Computer Engineering* in 1991 [52], and *The First International Workshop on FPGAs*, at Oxford in 1991 [51].

ii) the exploration of rapid prototyping through the use of FPGAs and their role in facilitating system design. Several test applications such as memory system test and boundary scan were presented at *The 6th Workshop on New Directions for Testing*, held in Montreal, 1992 [53].

iii) commissioning of a working prototype and associated test methodologies. Here again, the FPGA devices were utilized to build application specific testers.

iv) development of an efficient protocol in support of algorithmic design.

v) demonstrated application implementations of two sufficiently diverse algorithms to illustrate the flexibility of the prototype.

vi) delivery of a working system to the University of Manitoba that can be utilized in the development of parallel processing applications.

5.1 Retrospective

The current design uses the Xilinx 3000 series FPGA parts exclusively, where the Xilinx 4000 series FPGA parts represent a significant improvement over these parts in both routing and logic resources. Several components of the present design would benefit from these FPGAs, most notably the associative memory processor implementation.

One architectural modification that would benefit the current design, is to have the hash table of the associative memory processor be accessible to host processors. This would enable host processors to index the global identifiers associated with local identifiers, and accelerate operating system functions associated with the installation and removal of function components.

In retrospect, the present project represented as large a management task as a design task. Difficulties were experienced in management of the over 450 schematics, and in particular, managing revisions for the FPDs. All the current software was written in a combination of four assembly languages, which posed its own difficulties. This project represented a significant challenge in the management of several diverse technologies, CAD tools, and software languages.

Appendix A

Electromagnetic Transient Simulation Hardware

A.1 Introduction

In 1987 a multiprocessor was designed by G. Rosendahl and constructed at the Manitoba HVDC Research Centre for the solution of electromagnetic transients in real-time [50]. The model equations are the ones proposed by Dommel [13] and are currently used in commercial non-real-time simulation software (EMTDC and EMTP). The necessity for real-time operation is a direct desire to use the actual system controls on the simulator for commissioning and proof of concept experimentation (non-destructively).

Previously, the simulation of power system transient phenomenon in real-time could only be done on analog simulators. In this method, scaled down components of the network are used to model the system, presenting the results as continuous analog quantities. The main advantage of analog simulators is the capability to interface system controls and maintain real-time operation, allowing the accurate analysis of various network and control configurations (under transient conditions). This provides a valuable tool in the evaluation of new control schemes and system problems. In spite of this, a number of disadvantages exist when comparing this method with a digital program's inherent flexibility and ease of use. Typically, analog simulators are expensive ($5-10M), relatively inflexible (due to "patch cording" of the networks), and they present difficulties in term of recording results.

Digital programs such as EMTDC (Electromagnetic Transients Program for Direct Current)[77] and EMTP (Electro-Magnetic Transients Program)[13] can perform these studies. However, because of their non real-time operation, the actual controls cannot be interfaced. The desired controls must be modeled digitally within the program. The advantages of these tools include their cost effectiveness, flexibility, and ease of use.

In this appendix, the specifics of the electromagnetic transient solution will be discussed insofar as it influences machine architectures. The hardware constructed will be discussed in a fair amount of detail, including comments on some aspects of the architecture in an attempt to characterize the architecture's weak and limiting points. This is followed by some architectural observations to aid in design and specification of the current work.

A.2 Model Characteristics

The two dominant simulation programs, EMTDC and EMTP, use component models described by Dommel [13]. In these programs capacitors and inductors are modeled as a current source I_h in parallel with a resistor (as shown in fig. A.1). Similar representations can be derived for transformers and transmission lines [13,40].

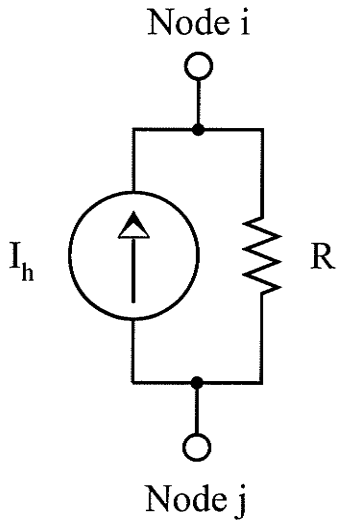


Figure A.1: Generic passive component model.

By allowing networks to be modeled as resistors in parallel with current sources, nodal equations are derived. The simulation progresses in a sequential fashion, where the voltage across an element provides a current I_h for each element attached to a node. These I_h terms are summed, creating the vector entry I_i in equ. (A.1) for node i in a network.

$$\overline{V} = [Gmatrix]^{-1}\overline{I} \tag{A.1}$$

Multiplying the summed current vector $\overline{I}$ with the network's inverted conductance matrix $[Gmatrix]^{-1}$ provides a corresponding voltage vector $\overline{V}$, enabling a new time step to begin. The time step chosen is a factor in the accuracy of the solution and determines the maximum representable frequency response. In general, the smaller the time step the better the solution, but a lower bound exists as round-off errors become significant.

EMTDC solves a number of small network matrices, called subsystems. These subsystems are linked to each other by transmission lines or by other suitable models forming an equivalent to the overall system. These subsystems can be separated only if a time delay of one time step exists between them in the real system (i.e., the time it takes for a wave to travel the interconnecting transmission line is greater than or equal to the time step employed in the simulation). This scheme has a number of advantages over one large matrix. Each of the subsystems can be solved in parallel and then combined to obtain the overall system solution. This technique is in the form of a coarse parallel solution.

A.3 System Design

The system design encompasses both hardware and software to achieve real-time simulation of power system transient phenomenon. The hardware and software are designed with modularity and simplicity in mind and are modeled after EMTDC, supporting the concept of subsystems. The real-time operation of the system suggests a coordinated [33] form of execution as will be seen later on.

The coarse parallelism is taken advantage of by designing subsystem solvers or units to solve each subsystem of a network, thus, a one to one relation between

subsystems and solvers is formed. The following sections describe these concepts and how they are applied to build a real-time simulator.

A.4 Hardware

The machine is comprised of autonomous units called parallel processing units (PPUs). These PPUs are the subsystem solvers, and provide the design with modularity. For simplicity, only three functionally distinct system components or cards are used to form a PPU. As shown in fig. A.2 each PPU requires one bus master (BM), and a plurality of Processing Elements (PEs) and Communication Ports (CPs) all installed on a common bus.

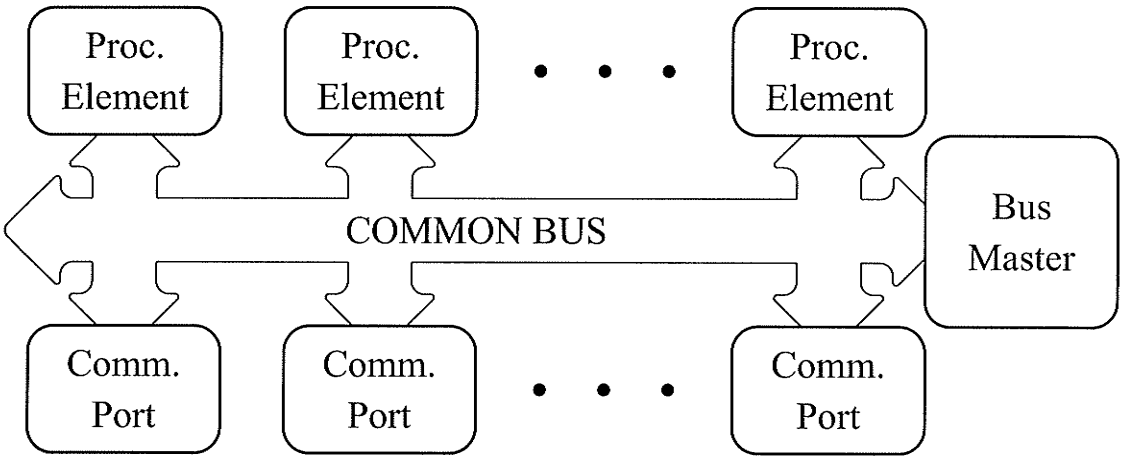


Figure A.2: Parallel Processing Unit block diagram.

The target application determines the number of PEs and CPs installed on each PPU and the communication topology between these units. By careful selection of the communication topology patch-cording can be minimized.

The number of PPUs that can be connected via communication ports is limited and is related to the communication memory size and the way it is shared between connected PPUs. This will become evident in the following sections.

A.4.1 Processing Element

The Processing Elements (PEs) provide the machine with its computational speed (over 13 MFlops for each PE). As shown in fig. A.3 each PE contains a Digital

Signal Processor (DSP), a local static RAM, a communications memory, one or two digital to analog converters DAC, a bus interface and control logic. Two operating modes are supported, one for system initialization and the other for simulation execution. During the reset mode, all memory and input/output resources are mapped onto the bus master's memory map. In run mode each processing element controls its own resources and functions autonomously.

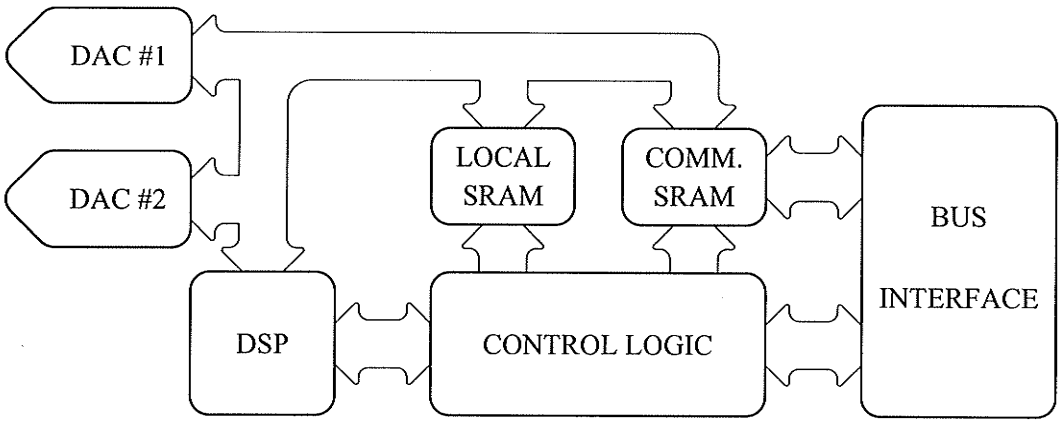


Figure A.3: Processing Element block diagram.

The local communication memory can be thought of as a selectively shared memory among all the PEs installed on the bus. That is, specific communication memory locations of a PE are shared with a selected number of other PEs. As an example, if each of n PEs produced one voltage entry in a vector (e.g., the voltage vector of eqn. A.1), each entry can be shared with the $(n-1)$ remaining PEs, such that after sharing, each PE has a complete image of the voltage vector within its own communications memory. Further, this voltage vector will occupy exactly the same space in each of these communication memories. This is a particularly expensive method of sharing information. It will ultimately restrict the number of PPUs that can be connected directly (adjacent via a communications port).

The communication memories information is shared via the common bus which is arbitrated by two communication request channels. This enables each PE of a PPU to belong to one of two communication groups (or information pools), or to belong to both. In the initial version of the machine, hardware handshaking of these request channels is not implemented. Instead the PE software is aligned

such that a communication termination is timed within the software. This is a very inefficient and crude method of synchronization which adds to software complexity. The logical extension of this is to use hardware semaphores with interrupt generation of communication terminations and procedural firings. Despite this, adequate flexibility exists for the solution of the model equations, but more channels and handshaking would certainly improve performance and clarity of program and data.

A.4.2 Bus Master

The bus master controls which mode (reset or run) the PPU is operating in and all transactions that take place on the common bus. As shown in fig. A.4, the bus master (BM) contains a minimal computer, a communication state machine controller, and a host communication link. The minimal computer manages the bus in reset mode, while the communications controller governs in the run mode. The communications link with a host (e.g., a workstation) is maintained continuously by the minimal computer, allowing the user access and control of the PPU.

Communication between processing elements is provided by the common bus and is coordinated by the bus master's communication state machine controller. This controller arbitrates the common bus via the two request channels. When a channel becomes active (e.g., all pool members on the bus consent to an information exchange) the state machine executes a communications program to share select information between modules on the bus. After completion, pool members are released to continue execution. The communication state machine operation is defined by a communication program which has three operand/control fields: program flow control, a PE source address, and a common source/destination address of the data to be exchanged.

Program flow control determines if communication should continue given the request status of the bus, or if it should terminate after the current transfer, and/or if a branch should be performed after the current transfer. The PE source address identifies which card or module on the bus will be the information source. The source/destination address is the common address among all the communication

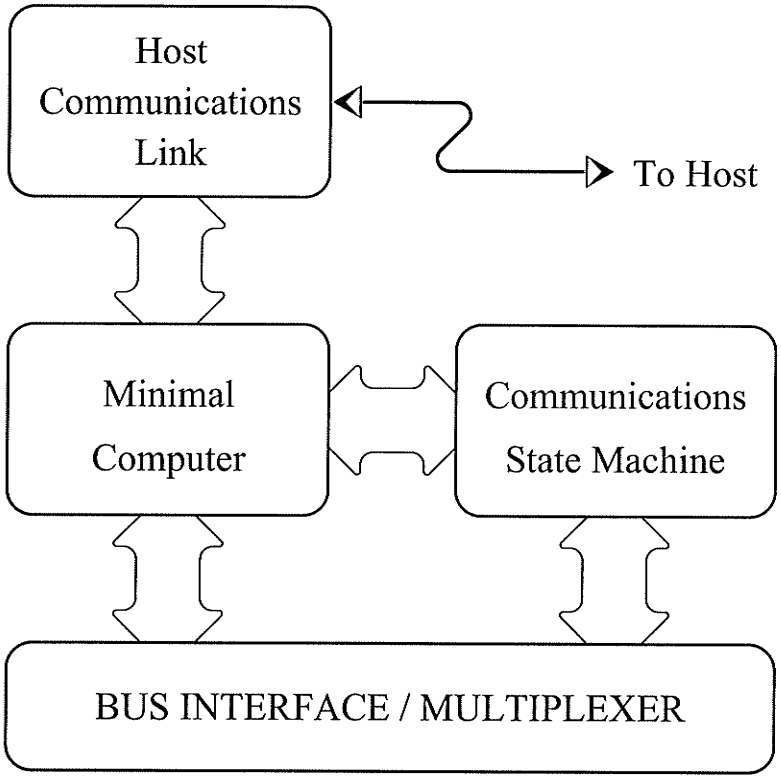


Figure A.4: Bus Master block diagram.

memories slated for communication. Information is read from the source PE and is written to all other destination PEs. In this way, a selective shared memory is implemented.

This communication architecture imposes many limitations on the size and diversity of solutions. Since all PEs must be synchronized, communication memories are of equal size, and the general structure hinders use of heterogeneous processing elements. These problems are attributed to the rigid nature of the communication interface.

A.4.3 Communication Port

Units are connected to each other through two Communication Ports (one on each PPU), forming a communication link pair. By adding CPs, the system designer has greater flexibility in realizing an optimal communications topology for a given application. The CP consists of an eight kilobyte block of dual ported RAM (same size as the PEs), a common bus interface, and a cable driver interface (to and from its counter part), as shown in fig. A.5. This module observes the

two operating modes as the PEs do, has no processing capabilities, and is under the strict control of the Bus Master. The CP records all the transactions taking place on its own PPU and relays this information to its partner on another PPU (unless the BM prohibits it). This saves computation time and scheduling/routing problems associated with determining export information. At the same time, its partner does likewise, so if information from an adjacent PPU is needed, a read of the associated CP is all that is necessary.

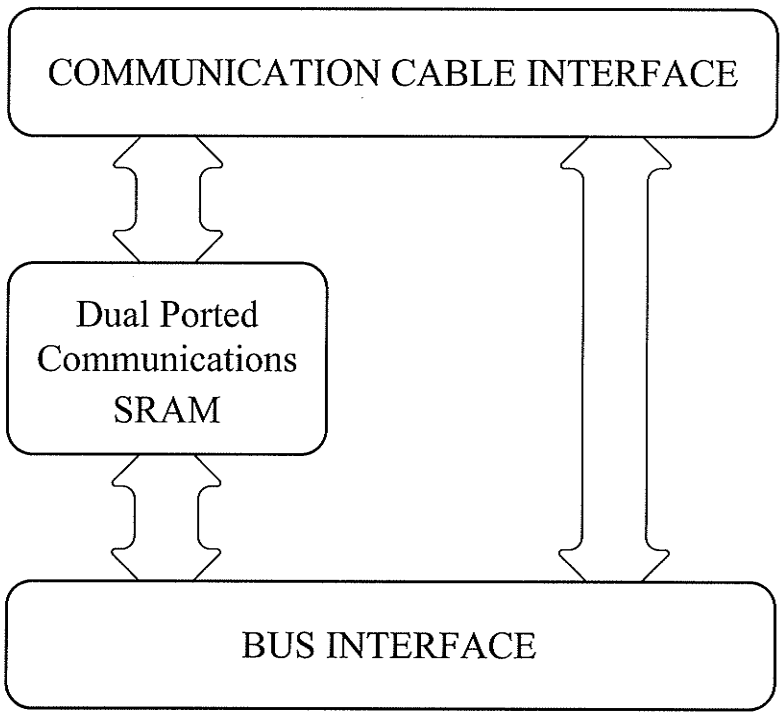


Figure A.5: Communication Port block diagram.

In the interest of reducing software complexity, information stored in each PPU's shared communications memory is mapped to the base page. This can pose a problem when two PPUs must communicate through a communication port's dual ported memory, as each will attempt to use the same memory space. This problem is easily solved by re-mapping the communication port's memory into logical pages such that no conflict will occur. This re-mapping also has an added advantage in that other PPU memory images can be viewed by the reference PPU in an uniform relative manner, further simplifying the software task.

A.4.4 Constructed Prototype System

The prototype constructed consists of two PPUs with sixteen PEs and one CP each, as shown in fig. A.6. The system is connected to a workstation host via two serial channels (one for each PPU). This system configuration allows experimentation with the system, solution concepts, various modeling techniques, and real-time control interfaces. A wide diversity of system configurations is possible by patch-cording a number of PPUs to form the required topology. An example of such an interconnection is shown in fig. A.7, where edges represent communication port interconnections and nodes represent PPUs.

A.5 Software

The software is modeled after EMTDCs concept of subsystems [9,77], where each subsystem matrix is linked together through transmission lines forming the larger equivalent system matrix. Instead of solving the large system matrix as one, each of the smaller subsystem matrices are solved and then assembled forming the system solution. This enables each subsystem solution to progress in parallel.

A.5.1 Switch Handling

Switches are an essential component to transient studies, but in order to perform a switch (modification of the network impedances), the new conductance matrix must be inverted to allow the solution to continue. This becomes a problem when considering real-time simulation given that inversions are very time consuming.

This problem can be circumvented to some degree, since all switches can be identified prior to simulation. Each of the possible networks can be created and inverted off line (prior to simulation). By forming a look-up table structure within the network processors, the various switch configurations can be indexed in real-time. This technique suffers from excessive memory usage, which limits the number of switches that can be implemented at any one time. However, by identifying switching sequences and limiting the size of subsystems (isolating the switches into a small subsystem), this escalation of memory usage can be curbed.

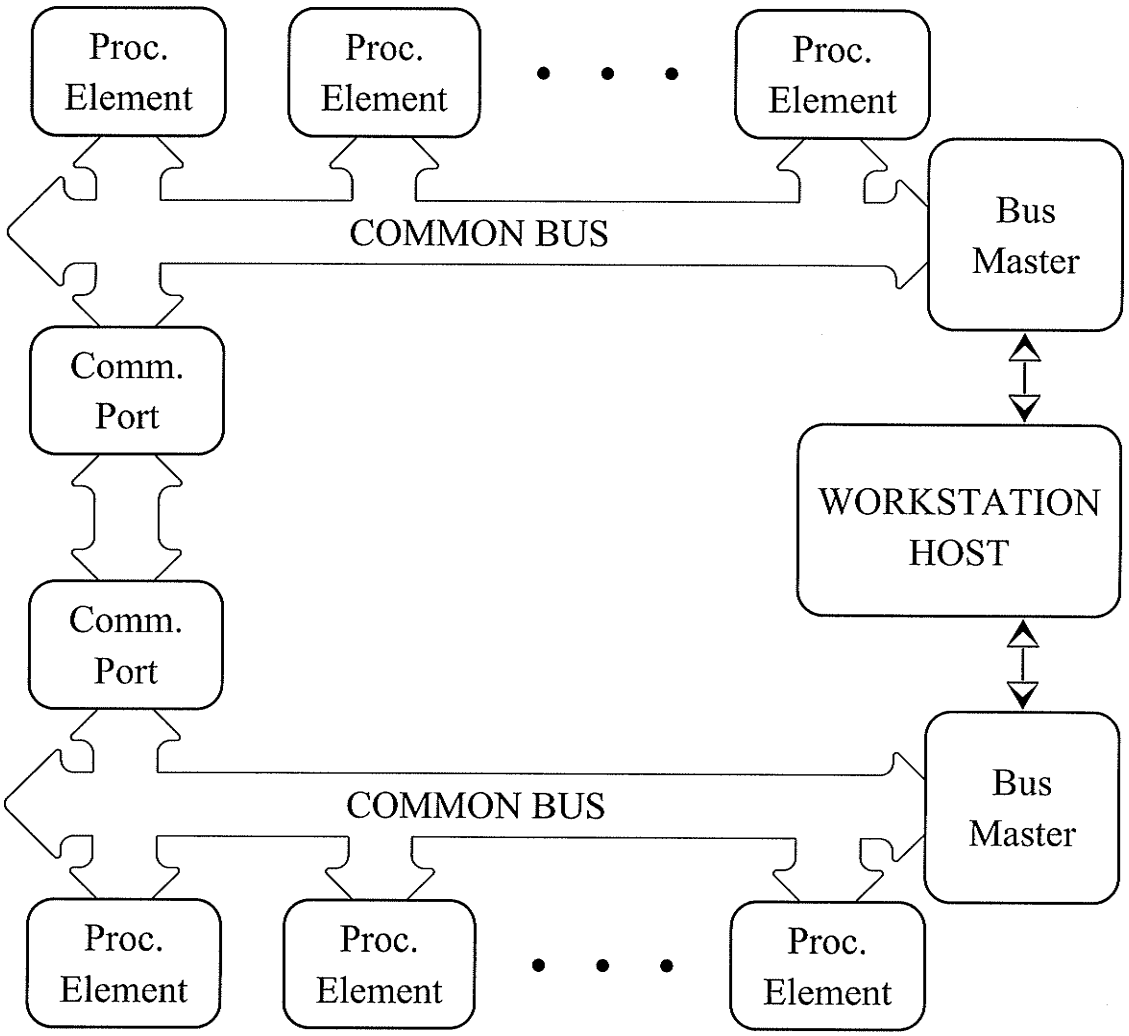


Figure A.6: Constructed prototype block diagram.

A.5.2 Solution Flow

During each time step of the simulation, three definable communication periods must take place: sharing current injections and control data, the vector current (see equ.(A.1)), and the resultant node voltages. Communication requests for these periods must be aligned among all requesting PEs. This alignment constraint is loosely enforced between system PPUs to facilitate the sharing of injection currents, control information, and subsystem voltages.

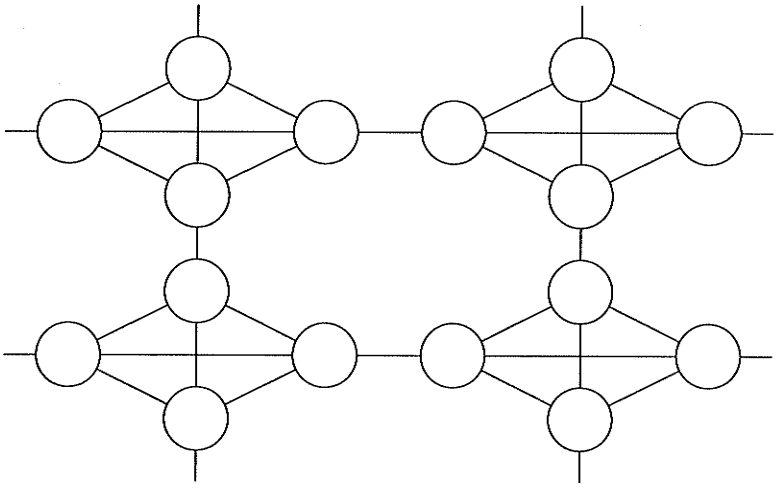


Figure A.7: Sample interconnection of PPUs.

A.6 Architectural Observations

The multiprocessor described performs real-time electromagnetic transient simulations exceptionally well, and at present, is a great commercial success. This architecture is a close fit to the traditional application-specific design mold; it is specialized, has a high specific application performance, and is architecturally inflexible. And as with traditional designs, this design too, does poorly on applications slightly out of its scope.

Of the many attributes of this architecture the most limiting and rigid is the communication interface. Composed of many inflexible component parts the centralized communication controller is by far the most rigid. It uses a communication program to govern information exchanges in an explicit manner, and each information exchange must be reflected in the individual code of each processing element. Only two hardwired communication request channels are provided, which limits each application to two communication topologies. This communication structure is very detrimental to many aspects of application development, with one exception, the electromagnetic transient solution.

Appendix B

Parallel Sparse Matrix Solutions

B.1 Introduction

Sparse matrix technology has been a well defined science for a number of years. The advent of the computer has enabled the use of this technology in a great number of applications [44]. But since computers have been primarily serial machines most of the work on sparse matrices has been concerned with reducing storage space and computation requirements. Thus, the operations on sparse matrices have become primarily a matter of "book keeping" and are designed specifically for serial machines.

It is the objective of this appendix to examine some aspects of current sparse matrix ordering techniques and perhaps to offer some insights into parallel sparse matrix solutions. The following section discusses a tool written for this purpose and demonstrates some insights gained.

B.2 Symmetric Sparse Matrix Analysis Tool

In order to study the effect different permutation matrices have on matrix factorization, a program was written to examine possibilities graphically and interactively (since the possible matrix orderings are $n!$ of an n x n matrix). A graphical environment is used to both view and manipulate various aspects of a matrices: topology, ordering, and resultant factorization. An input matrix is specified symbolically (i.e., for a given element either a number is present or it is not); this ensures that the maximum possible fill is represented following a

symbolic factorization. The various aspects are manipulated via a mouse interface, allowing the user to manipulate and evaluate different permutations quickly and with little effort. The program allows the original and permutation matrices to be viewed and manipulated, and it allows the resultant permuted and factorized matrices, along with the solution tree, to be viewed.

The original matrix A is a symmetrical matrix of size n x n. This matrix is read from disk and its topology is changed by adding or deleting symmetrical elements. The orthogonal permutation matrix P (ie. $P^{-1} = P^T$) is initialized as the identity matrix and can be viewed and manipulated. The resultant matrices B ($B = PAP^T$) and the gaussian eliminatation of matrix B (GEB) can be viewed. A solution tree is formed by extracting the data dependencies from the LU decomposed matrix displaying the hierarchical data dependencies and possible parallel operations, level by level.

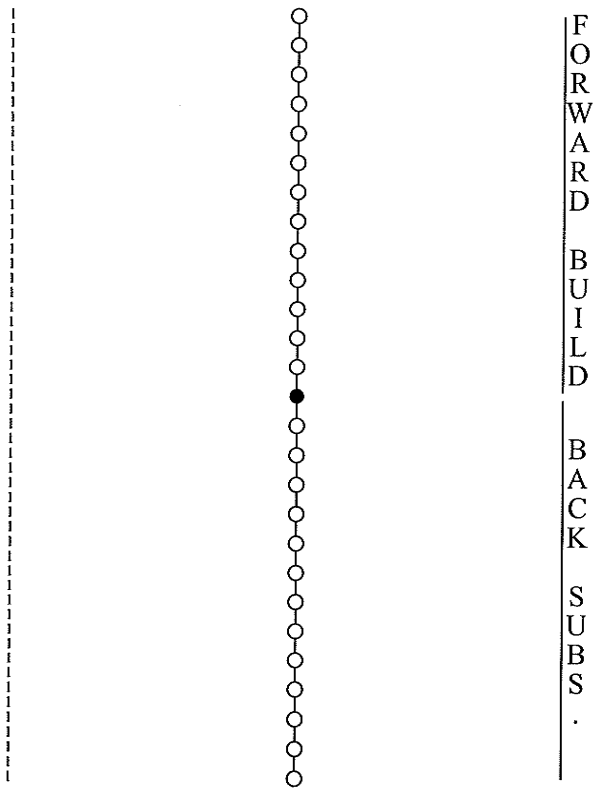


Figure B.1: Solution graph of 14x14 band matrix without permutation.

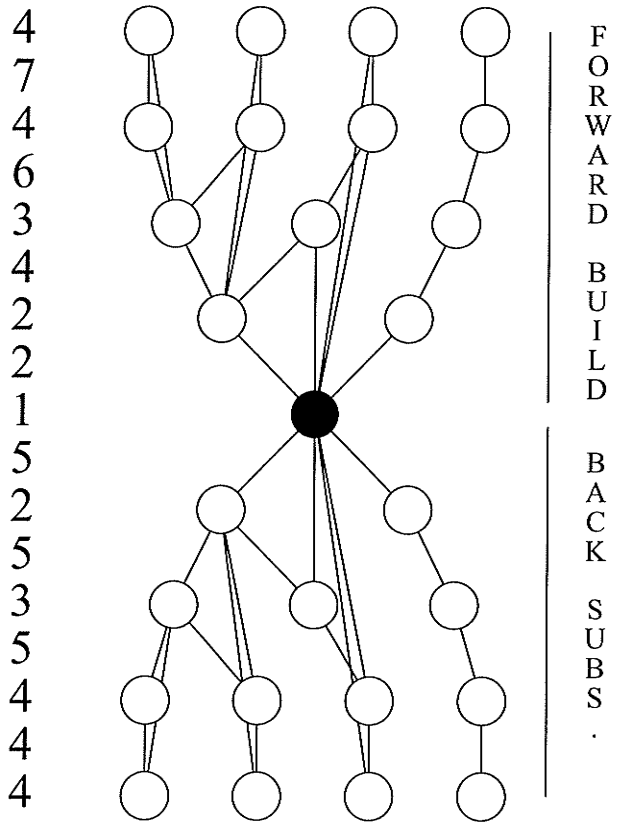


Figure B.2: Solution graph of 14x14 band matrix with permutation.

Only band limited matrices have been studied to date, since they are the closest regular structure to the ones found in transient stability studies. A range of half band sizes have been studied, from one to four. The object of this study of band matrices was to discover if a permutation matrix exists that could be generalized to increase the inherent parallelism in the sparse matrix solution, and if so, to define it.

Some interesting properties have been found. Consider a 14 x 14 band matrix with a half band of one as an example. If this matrix was factorized without altering its order, the solution tree would be completely serial as shown in fig. B.1. However, solution parallelism is greatly enhanced (see fig. B.2) if the permutation matrix of fig. B.3 is applied to the original matrix before factorization.

Both figs. B.1 and B.2 represent directed solution graphs from top to bottom. The solution graph is displayed in a level dependence manner such that each

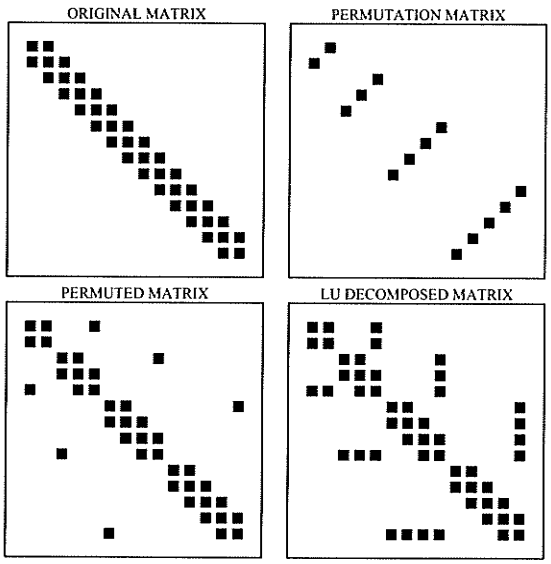


Figure B.3: Resultant matrices for enhanced parallel ordering.

numbered row (see left column of numbers in figures) represents a solution level. This number also indicates the number of parallel operations at each of these levels. During the forward building phase of the solution each numbered row indicates a calculation to be performed (both node and edge levels). In the back substitution phase only the edge levels require computation, while the node levels indicate the number of additional values solved (starting with the blackened node). This class of permutation matrice has good parallelizing traits and is easily generated as is evident in the following discussion.

Notice that the permutation matrix of fig. B.3 has four sub-matrix blocks and each sub-matrix is of a "reverse diagonal" form. Starting with the upper leftmost sub-matrix, it is a 2 x 2 matrix the next lower right is a 3 x 3, and so on; this trend continues to the bottom right sub-matrix, where each successive sub-matrix is increased in size by one or the half band size of the band matrix. As another example, if a band matrix has a half band of two then each successive sub-matrix would be increased in size by two. This manner of forming a permutation matrix for band matrices works well for the half band sizes tested, with good parallelism and acceptable fill initially, but as the half band size increases fill becomes more of a problem (see figs. B.5, B.6, and B.7 at the end of this appendix). The solution graph of fig. B.2 exhibits well balanced parallelism especially during the back

substitution solution phase. This is better seen in the plot of fig. B.4, the back substitution level processing requirements. This flat resource requirement helps to balance a parallel algorithm and realize an optimal usage and performance of a parallel architecture.

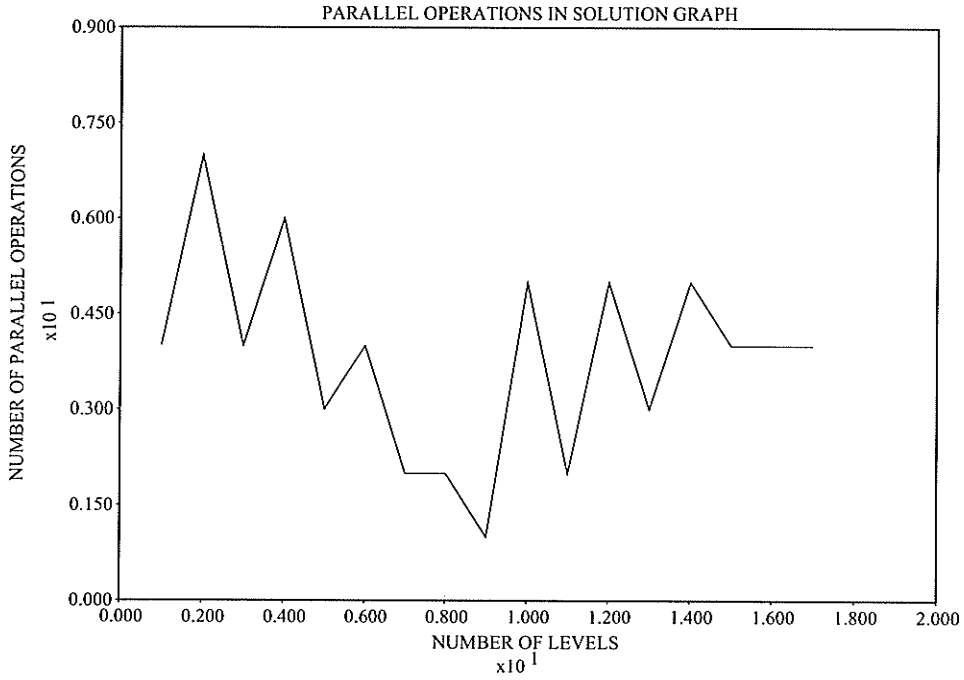
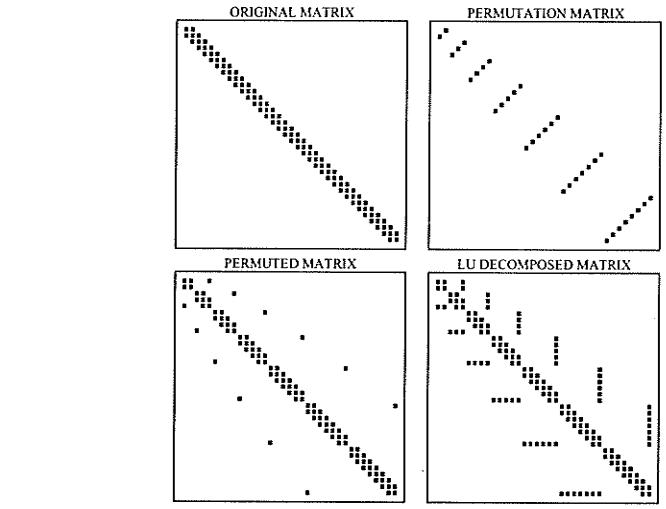


Figure B.4: Parallel operations of permuted matrix.

B.3 Summary

The above discussion has suggested a method to increase the number of parallel operations of banded sparse matrices by reordering the matrix prior to execution. The transient stability studies require matrices very similar to band limited matrices. Through many algorithms that reduce the band size of matrices [44] some parallel advantage may be gained.

The directed solution graphs of these matrices show that there is a direct correspondence with functional languages and data flow models. To ease the burden of high communication traffic, many clustering algorithms could be applied to reduce the number of information packets involved in a solution.



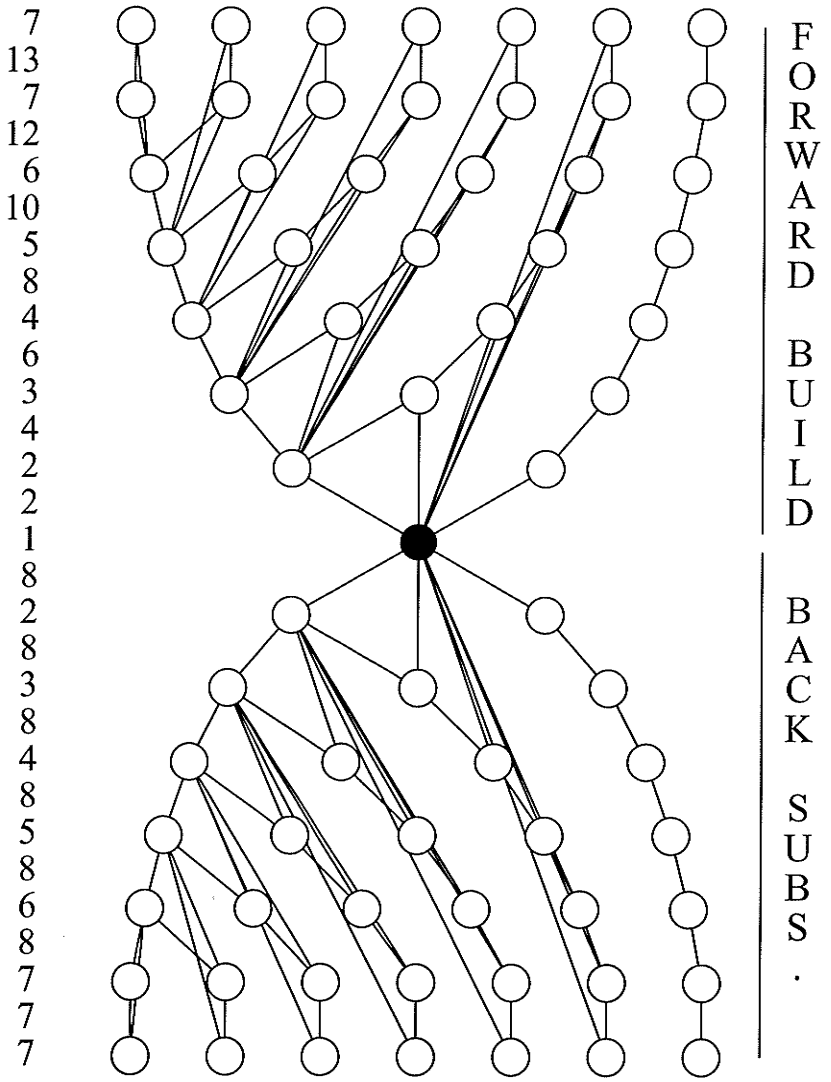


Figure B.5: 35x35 matrix with half band of 1.

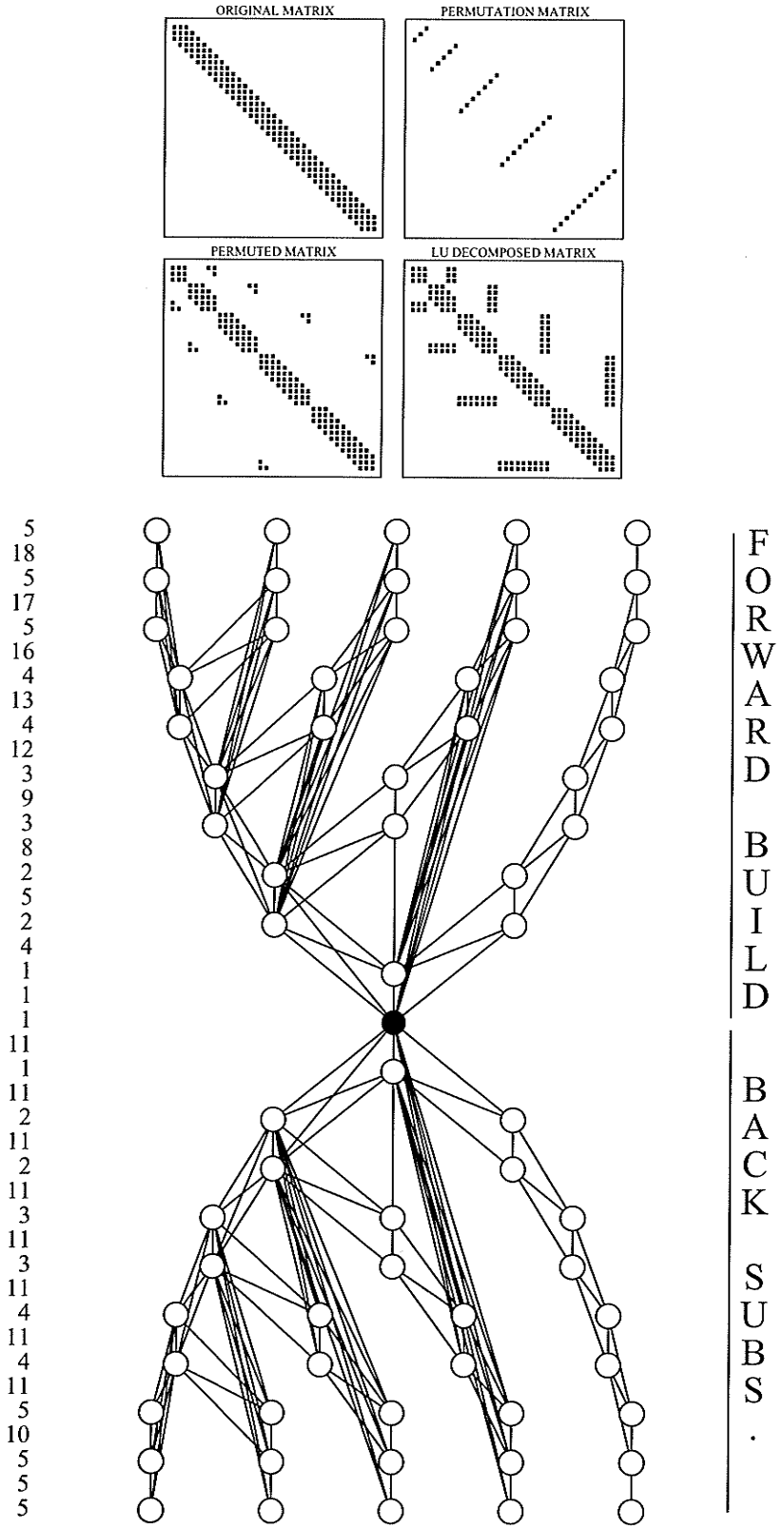


Figure B.6: 35x35 matrix with half band of 2.

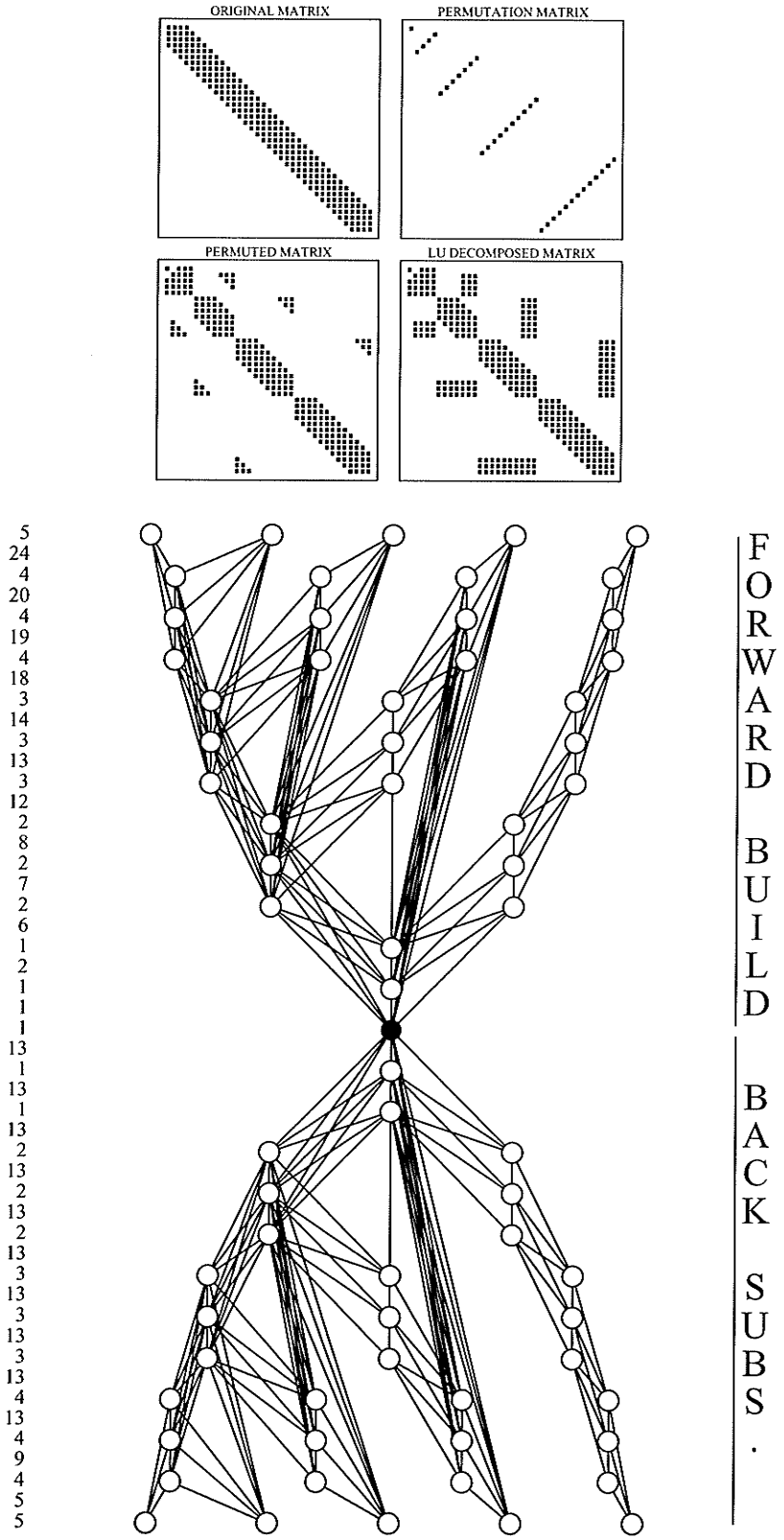


Figure B.7: 35x35 matrix with half band of 3.

Appendix C

Overview of FPD Signal Pin Assignment and FPGA architecture

C.1 PAL Pin Assignment

A wide variety of programmable array logic (PAL) devices are available. These components vary in complexity and functional attributes, but a common theme of pin assignment is evident, as shown in Fig. C.1. Power, ground, clock, and output enable pins all have common positions. Inputs are predominately on the left, while outputs are on the right of the package. These simple constraints have been exploited in the current design to maximize the choice of devices through schematic pin assignment.

When making schematic pin assignments for PAL devices, attempts were made to reserve the four corner pins for power, ground, clock and output enable, even if a registered device does not appear to be required. The clock pin should be wired to a local clock source, and the output enable to a set of pull-up and pull-down resistors. This will allow any of the devices of Fig. C.1 to be used in the same PCB location.

Output pin assignments were made such that outputs with the highest estimated min-term counts are situated to the centre of the package, that is, pins 18 and 19 of a 24 pin package. Proceeding outwards to the outputs with the lowest min-term counts, are pins 15 and 22. The reason for this strategy is that the

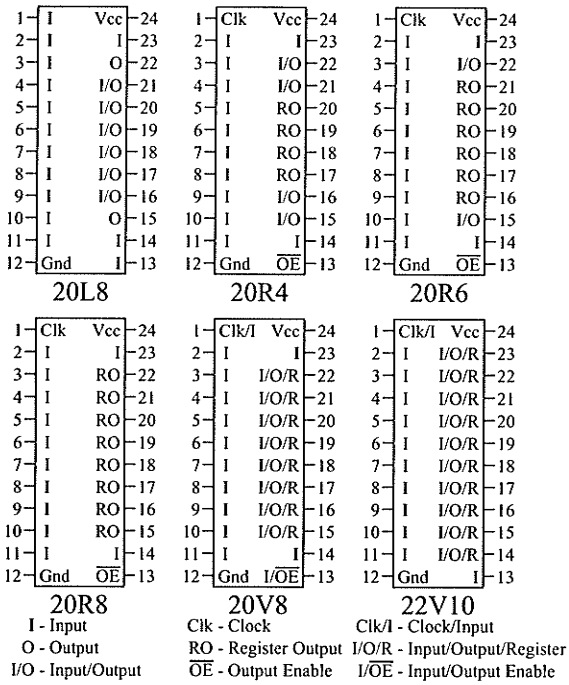


Figure C.1: Some common PAL device pin assignments

22V10 device scales the number of min-terms in this fashion: 15 min-terms are available to pins 18 and 19, while only 10 min-terms are available for pins 15 and 22. This device also has 10 possible outputs instead of the standard 8, and pins 14 and 23 have 8 min-terms each. When assigning inputs, pins 14 and 23 should be left to the last, and any spare inputs should be wired to a set of pull-up and pull-down resistors for maximum flexibility.

These strategies also prove useful at various stages of prototyping. As a simple example, consider a PAL used as a bus decoder for accessing memory or I/O interfaces. This PAL may be replaced with a finite state machine to simulate bus activity without the CPU installed, exercising local circuitry and simplifying debug tasks. Many different formats of PAL devices are available, including electrically-erasable, ultraviolet-erasable, or one-time programmable technologies. The electrically-erasable ones are the most useful for debugging tasks. They allow the device to be reprogrammed at least 100 times, and many devices can be configured to mimic common register or combinatorial PAL configurations.

C.2 FPGA Pin Assignment

Schematic pin assignments for field-programmable gate arrays (FPGAs) are very dependent upon the application. For the current design, Xilinx 3090 175 pin-grid array parts were used exclusively. Briefly, the 3090 has a 16 by 20 array of configurable logic blocks (CLBs), 144 usable I/O pins, and an equivalent gate count of 9000 (section C.3 gives a brief background of the Xilinx 3000 series parts). An integrated TAP interface is not provided, but is available on the 4000 series. Unfortunately Xilinx, did not provide an upgrade path to the 4000 series for this part, limiting any future growth through pin compatible replacements to the 3100 series. The lack of an integrated TAP interface does not pose a problem to the current design, since a requirement to include non-standard interfaces (such as Motorola's on-chip emulation (ONCE) port) into the TAP chain is an objective. This does, however, put a strain on the number and diversity of functions that can be implemented in the space provided.

A number of pins are to be given special attention during pin assignment: HDC, LDC and INIT. These pins should not be wired to outputs, or to chip enables, without careful consideration. An effective role for the LDC (Low During Configuration) pin is as a reset input of a slave CPU. When power is applied, this pin will remain low, holding the CPU in reset until the FPGA configuration is complete. A small amount of internal logic can then be included to release and assert the reset pin on command after configuration.

When assigning address and data bus lines to the FPGA, they should be assigned to opposite sides of the package, and aligned with the internal horizontal long lines. This will reduce internal routing constraints, and facilitate designs modelled after a register file structure. Fast or high fanout signals should be wired towards the centre of any package side, in order to increase internal routing options. Slower or non-critical signals can be assigned to package corner positions with less routing access. Any spare I/O locations should be brought out to a header, to ease debugging and circuit patches and/or additions.

The partitioning of a design between FPGA and PAL devices also depends on the application. PAL devices typically have greater speed and smaller footprints

than FPGAs, allowing them to be placed closer to where they are needed. PALs serve as effective subroutine functions for FPGAs, with a minimum number of control signals. This allows FPGAs to be placed centrally amongst many PAL devices under their control, which reduces high-speed trace lengths. A suitable application for this scheme is in the aforementioned message decoder, where an FPGA performs information flow decisions, and PALs implement these decisions.

C.3 Xilinx 3000 Series FPGA Overview

Xilinx logic cell arrays (LCAs) are field-programmable gate array devices which employ static RAM cells for configuration. These devices are constructed using three basic programmable resource types: input/output blocks (IOBs), configurable logic blocks (CLBs), and routing resources. Each is described in the following paragraphs.

The IOB construction is shown in Fig. C.2. The available configuration options are indicated in five boxes along the top of the figure. A programmable high impedance pull-up resistor is provided. Inputs can be either CMOS or TTL compatible. This is a global function and is not selectable for individual IOBs. Input signals from an I/O pad can be either a direct or latched input (or both). The latched inputs can be clocked on either the rising or falling edge of one of two clocks (see bottom centre of the figure). Outputs can be either direct or latched, inverted or not, and with or without tristate control. Tristate control can be active on a high or low signal, and data can be latched on either the rising or falling edge of one of two clocks, as in the input case. One unique feature is control of output slew rate. Two choices are provided: slow or fast. The slow slew rate reduces power supply noise and gives some control over transmission-line effects (which depend on rise and fall times of signal edges). The fast slew rate option is faster, but incurs more power supply noise, so much in fact that their number must be limited. This slew rate option is also helpful to align or misalign output signals.

The CLBs form the heart of the array (Fig. C.3). Each has two flip-flops and a combinatorial function unit. Two outputs x and y can have either a flip-flop or

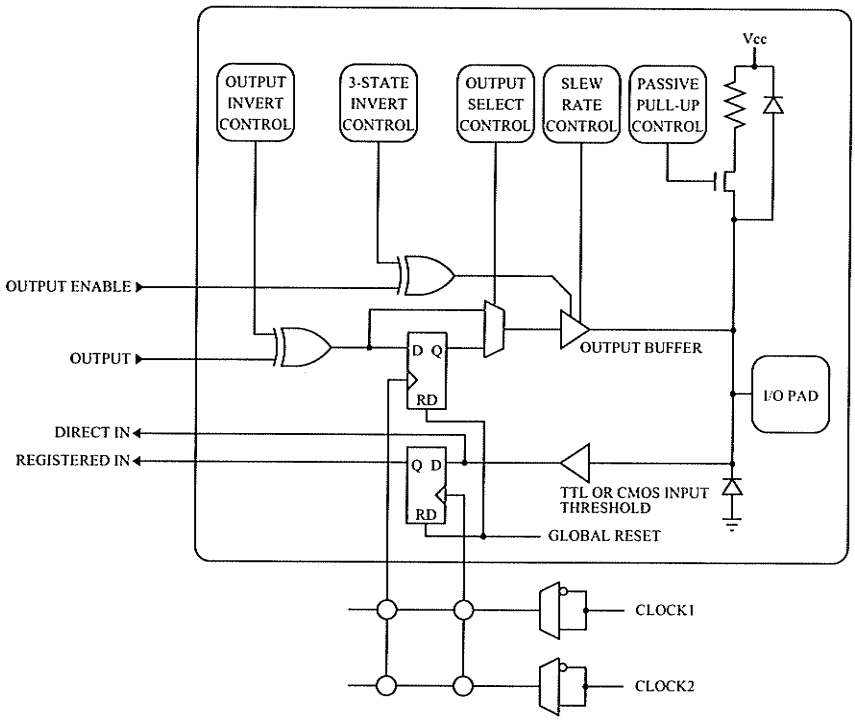


Figure C.2: Input/output block

combinatorial function as input. The flip-flops have a common clock (either edge), reset, and enable inputs. Inputs to these flip-flops are selected from one of two sources: DATA IN direct, or from one of combinatorial function units (either the F or G output). The combinatorial function unit is constructed from two 16-bit look-up tables (implemented in RAM) with two outputs (F and G), and a choice of seven input sources, via a multiplexer.

One of three configuration modes can be assumed by the combinatorial function unit (Fig. C.4). The first mode (Fig. C.4a) provides for two functions of up to four variables each, with the A input common to both. The other three inputs are chosen from the remaining six as the multiplexers dictate. The second mode (Fig. C.4b) combines the two look-up tables to form one function of five variables. Again the inputs are chosen as the input multiplexers dictate. The last mode (Fig. C.4c) allows two functions of up to four variables to be selected, based on the value of the E input. Also note that since the combinatorial block uses a lookup table structure, multileveled logic functions reduce to a single level.

Currently-available LCAs provide a maximum CLB toggle rate of 250 MHz,

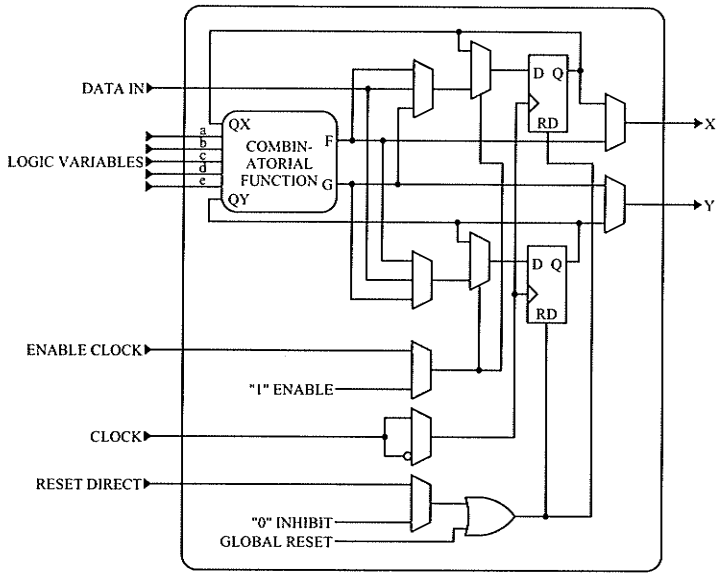


Figure C.3: Configurable logic block

and up to 440 CLBs in a X3195 part. This offers good flexibility and resources to the designer. The greater challenge is in routing the maximum number of these CLBs and IOBs into useful circuits. Typically, only about 70% can be exploited, due to insufficient routing resources.

The LCA is laid out as shown in Fig. C.5, with the IOBs along the die edges, and the CLBs arranged into a two-dimensional array, each framed by routing resources. Routing resources are split into four categories: nearest neighbour, horizontal and vertical long lines, general interconnect, and global. Each routing resource offers its own unique attributes in routing a given design.

In Fig. C.5, the possible nearest neighbor connections between CLBs are shown in the large circled area. The x output can be connected directly to the c input of the previous CLB, and/or to the b input of the following CLB on the same row. Similarly, the y output can be connected to the d input of the previous row, and/or to the a input of the next row. CLBs bordering the array have similar capabilities with neighboring IOBs. These connections are the cheapest routing resource available, and provide the smallest routing delay. More importantly, they do not overlap with any other available routing resources. As such, these routes should be used whenever possible, to reduce constraints on the other more valuable routing resources.

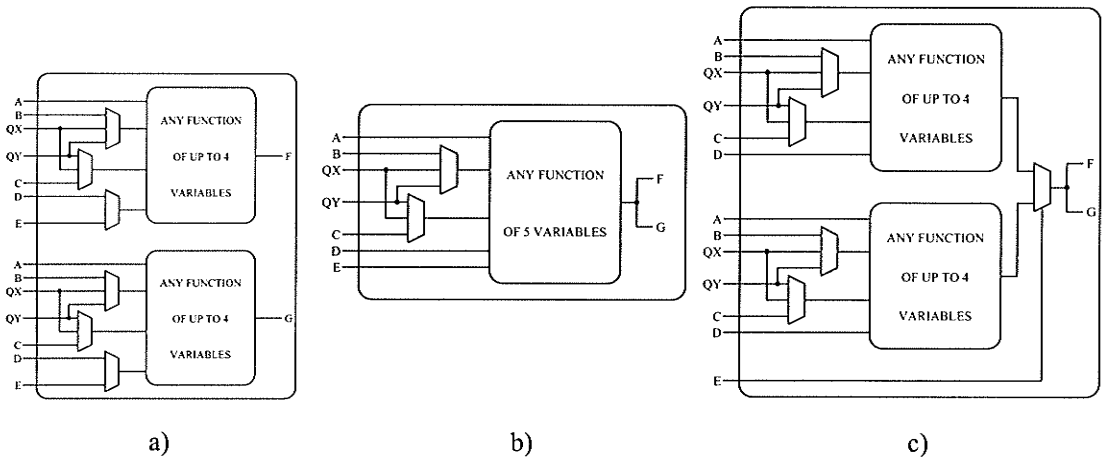


Figure C.4: Combinatorial function unit modes

Horizontal and vertical long lines span the array in both the horizontal and vertical directions respectively. Three vertical long lines are available per column of CLBs. Two of these vertical long lines can be divided into two segments at the mid-point of the array, but only for arrays 12 by 12 or greater. No tristate capability is provided for these long lines. Horizontal long lines, on the other hand, do have tri-state buffers, one for each column of CLBs, plus one. In addition, each has two pull-up resistors, which can be independently enabled or disabled for each long line. This allows wide *AND* functions to be formed by configuring tri-state buffers as open collector gates, and enabling the pull up resistors. These long lines cannot be split at the array mid-point, unlike the vertical ones. This routing resource enables register and bus structures to be implemented in a straightforward manner, and is a very important routing resource.

The general interconnect resource frames each CLB with a combination of five straight metal segments, connected to *magic boxes* or analog cross-point switches at the corners. This provides the most general routing resource for the array, but is also the slowest. Each *magic box* implements an incomplete grid of analog cross point switches, such that only a select number of points can be connected from any given point. This, however, does offer fairly good flexibility in routing. One concern is the routing delays incurred, since every analog switch in the signal path increases the signal delay by $2ns$. In addition, signals need to be boosted

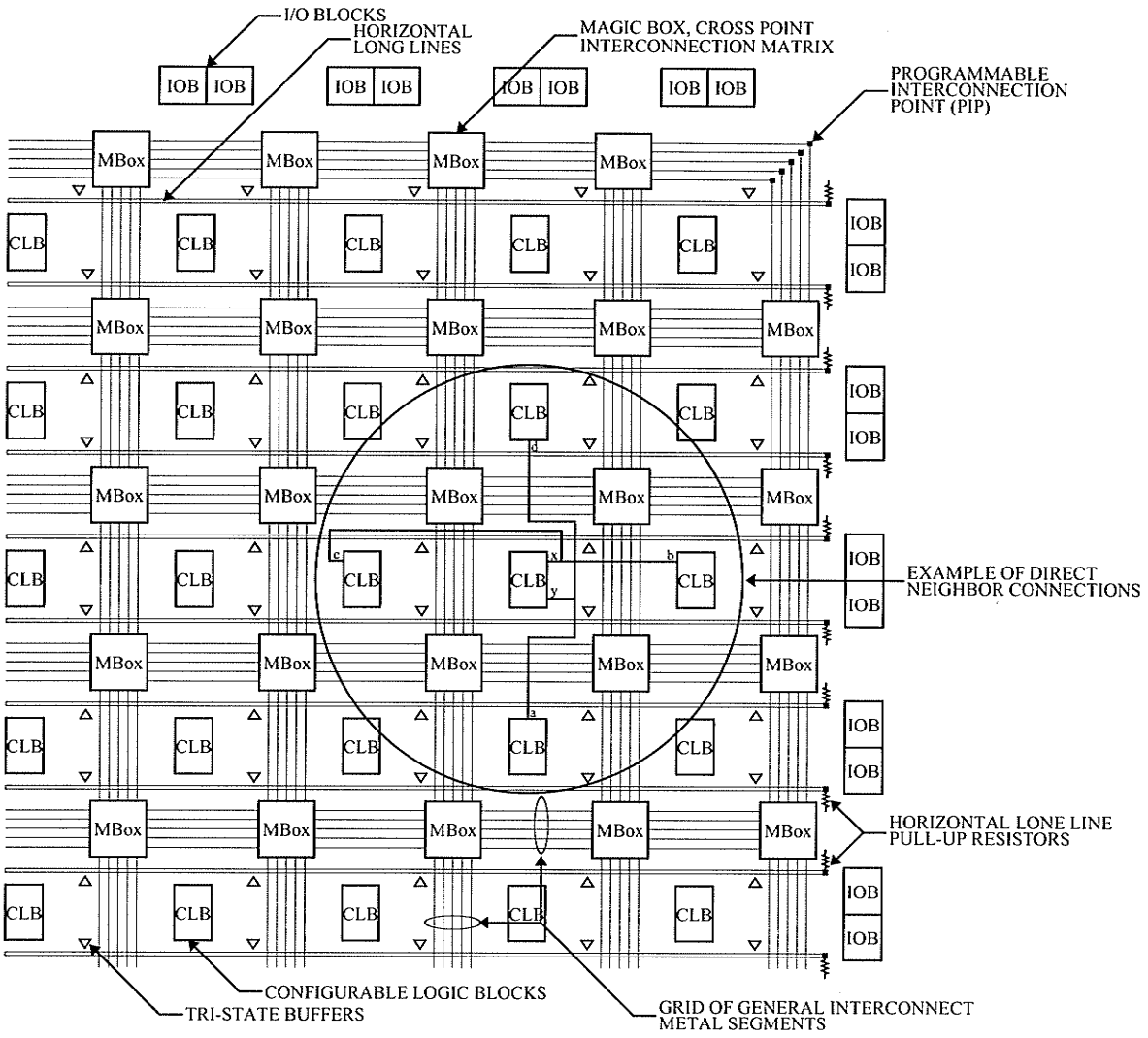


Figure C.5: Xilinx routing resources

by buffers after traversing several of these analog switches. All non-critical signal paths should make maximum use of this resource.

The global routing resources are limited to two signals, and are usually reserved for clocking CLBs. Two low skew buffers ($3ns$) are provided to drive each global line, located at opposite corners of the array. In addition, one crystal oscillator circuit is provided to drive an externally attached crystal. Only a brief overview of this architecture's many features was presented here. For further information, one may consult the Xilinx data book.

Appendix D

Control Sequence Design Methods for Xilinx 3000 series FPGAs

Application architectures implemented within the FPGA resources of the current multicomputer represent a large design task. Many diverse and specialized control sequences are required for each application architecture. A number of control sequence design options exist; microcode, wide finite state machine (FSM), or shift register based finite state machine (SRFSM) structures. The chosen design method should have a good structural match with Xilinx's 3000 series FPGAs, and should ease sequence design.

Microcoded structures are easy to program, once support structures are in place. However, this method is inappropriate for the current 3000 series FPGA architecture, where no RAM or ROM is provide to store the microprogram. Wide FSMs are amenable to the present architecture, but require significant routing resources and are difficult to design. While SRFSMs require less routing resources and are particularly easy to design, they are slower than FSM designs. It was decided that speed was a secondary concern to design ease, given the number and diversity of sequencers required.

The SRFSM design method presented here is a synchronous register transfer method. All control sequence designs are hierarchical in construction, and have a single element at their root; "Act1". This element is constructed from one very small FSM to implement the sequence design protocol, which closely mimics shift

register operation, and is used throughout. Through hierarchical construction generic libraries of commonly used functions are easily created and used. This organization has enabled the quick and efficient design of complex sequences [43].

D.1 Control Sequence Design Elements

The present method defines a protocol which enables components of a design to interact in an uniform manner. This protocol defines two actions, *start* and *end*, where *start* activates a function, and *end* terminates a function. All functions and components defined here are synchronous, therefore, all state changes are understood to happen on the rising edge of a global clock. Further, all sequences begin in "state 0" after a global reset.

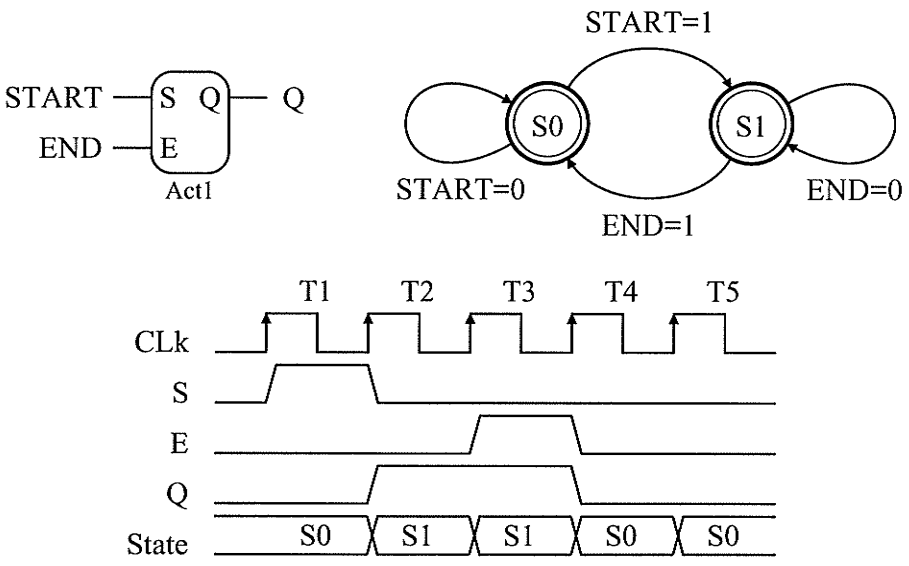


Figure D.1: Act1 state diagram and example logic waveforms.

The state diagram of Act1 is shown in Fig. D.1. Act1 has two inputs START and END, and one output Q. In the inactive state (state 0), Q is low or zero, and in the active state (state 1), Q is high or one. Beginning from state 0, Act1 is activated by a one on the START input. Once active, the only way to revert to the inactive state is a one on the END input (other than a global reset). Referring to the logic waveforms (Fig. D.1), Act1 begins at state 0 in clock period 1, a START is detected in period 2, activating the element (state 1), and Q remains active

until an END is detected in period 4. If the START input was presented again in period 5, the element would have been reactivated. This is a versatile sequence design element, and forms the basis for every sequence design.

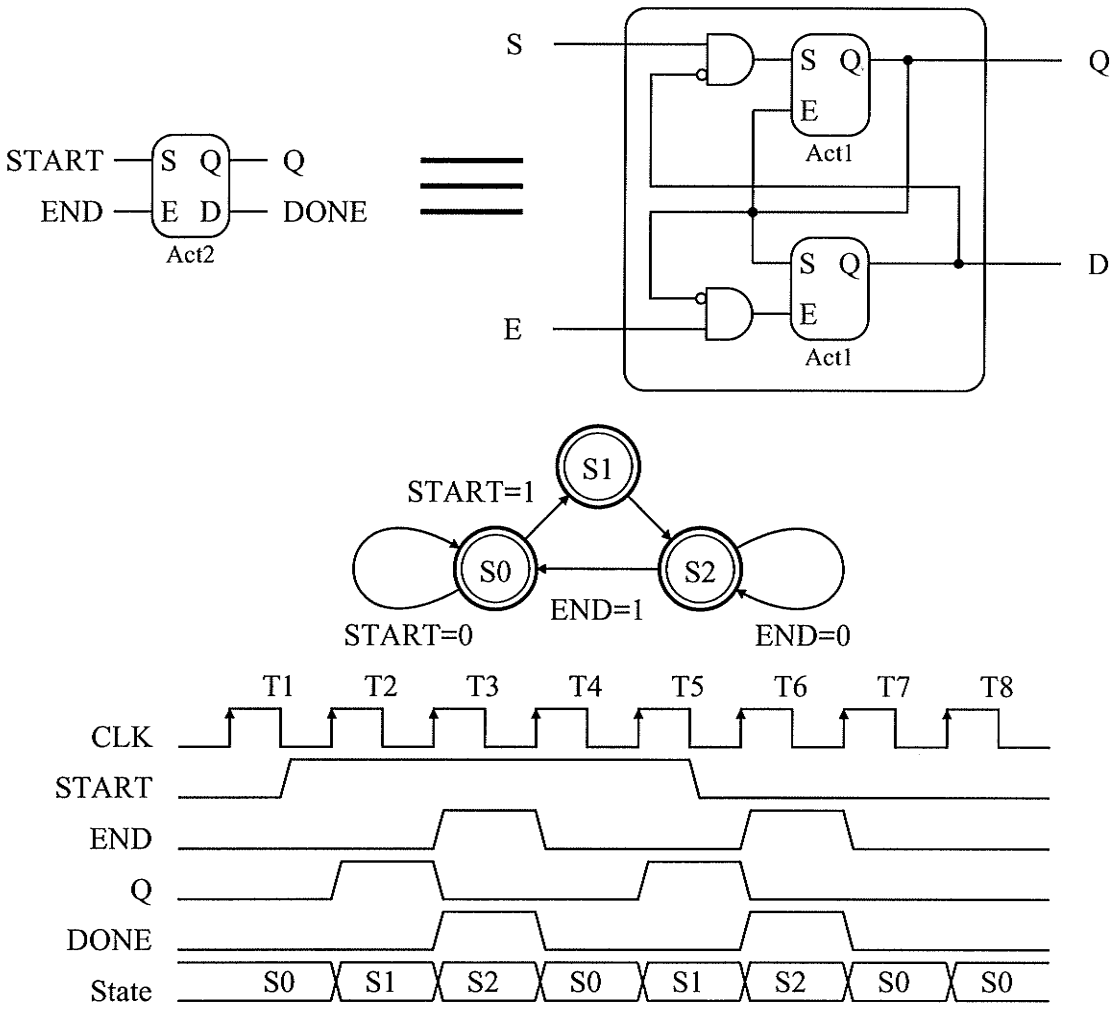


Figure D.2: Act2 construction, equivalent state diagram, and example logic waveforms.

A commonly used and more complex sequence design element, "Act2", is shown in Fig. D.2. This element is constructed from two Act1 elements and two AND gates. Act2 has a START and END input, and two outputs, Q and DONE. Referring to the equivalent state diagram, Act2 is active if in state 1 or state 2, and inactive if in state 0. In state 1, the Q output is a one, and zero otherwise. Similarly, in state 2, DONE is a one, and zero otherwise. As with Act1, Act2 is activated by a one on the START input, the output Q goes high

for one clock period and then returns to zero. At the same time, when the Q output returns to zero, the DONE output goes high, signaling that it is done. The DONE output remains high until a one on the END input is presented, terminating the function. If DONE is wired to END, START provides a trigger input for the logic waveforms of Fig. D.2. This sequencer operates much like a one-shot, where the START input is the trigger for a predefined sequence of events. Several commonly used control structures are shown in Fig. D.3, which have high-level language equivalents.

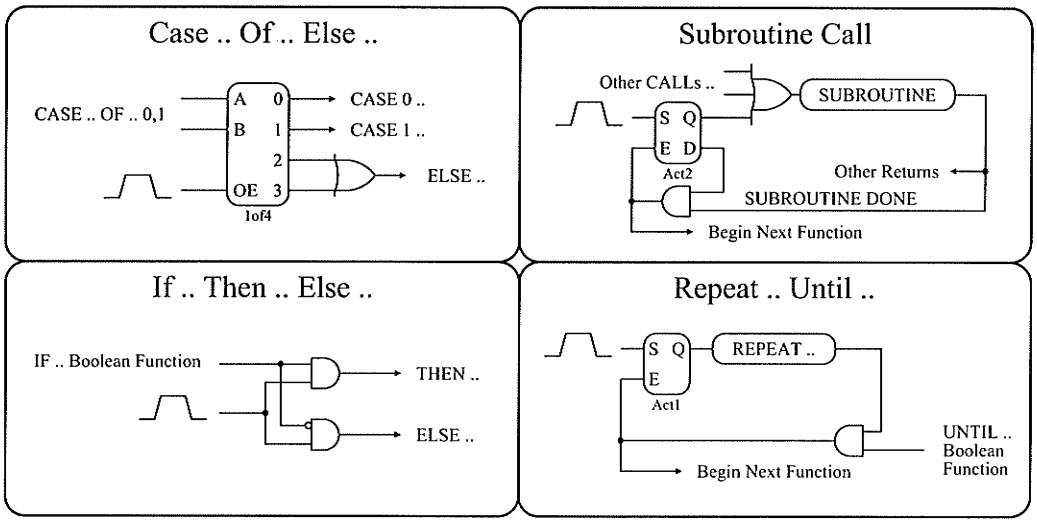


Figure D.3: Examples of common sequence structures.

Partitioning a design across multiple FPGA chip boundaries is difficult for some control schemes, but is very straightforward for the present scheme. To enable this, two new library elements are defined; BcS and BcR. These two components form a link pair between two FPGAs, where the calling or source FPGA, through BcS, evokes a control sequence on the destination FPGA, through BcR. BcS and BcR enable the standard control sequence protocol to be implemented using only one I/O pin on each FPGA (Fig. D.4). BcS calls a function on another FPGA by single pulse on the START input, and receives confirmation of the function's completion by a single pulse on the DONE output. Similarly, BcR receives the START pulse and generates an internal pulse to evoke the function and relays the DONE pulse to the calling FPGA upon completion. To ensure

recognition of START and DONE pulses between FPGAs, the global clocks of both are synchronized, but 180 degrees out of phase. This strategy is scalable to large FPGA arrays, by assigning the two clocks in a checker-board fashion.

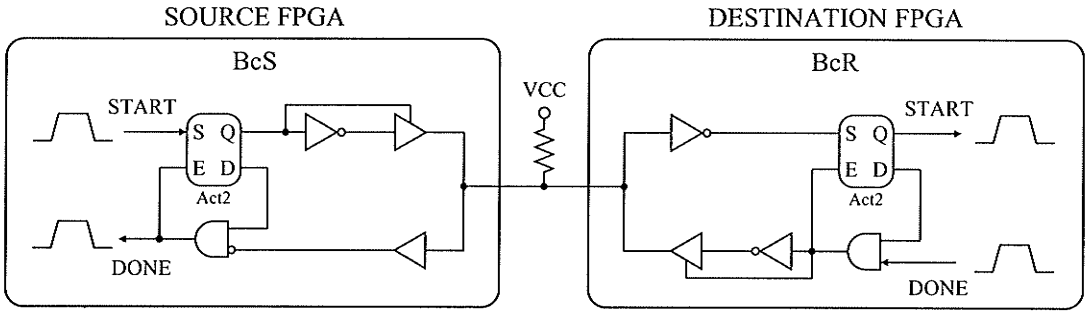


Figure D.4: FPGA chip boundary cell construction.

Application designs begin with the definition of a set of registers, which provide the necessary functionality to satisfy application requirements. Through the definition of register transfer operations, applications are defined. This may be accomplished by either register transfer pseudo-code or a flow chart. These sequences are then directly implemented using the present sequence design elements and methods.

D.2 Summary

The method presented here offers a uniform approach to sequence design using multiple Xilinx 3000 series FPGAs. Hierarchical construction of components enables very complex structures to be easily built and included into component libraries, and yields favorable flexibility. Rapid prototyping attributes are also evident, through the use of sequential structures with high-level language equivalents. Although this method is slower and requires more configurable logic blocks than wide FSM designs, routing constraints are reduced and applications can be constructed in a straightforward manner, which accelerates the design task.

Bibliography

[1] Ali Abur. "A Parallel Scheme for the Forward/Backward Substitutions in Solving Sparse Linear Equations", IEEE Trans. on PWRS, Vol. 3, No. 4, November 1988.

[2] Subhas Chandra Agrawal. "MetaModeling A Study of Approximations in Queuing Models", MIT press Cambridge Mass, London, England, 1985.

[3] F. Alvarado, R. Lasseter, J. Sanchez. "Testing of Trapezoidal Integration with Damping for the Solution of Power Transient Problems", IEEE Trans. on PAS. Vol. PAS-102, No. 12, December 1983.

[4] F. Alvarado, D. Yu and R. Betancourt. "Partitioned Sparse A-1 Methods", IEEE/PES 1989 Summer Meeting, Long Beach, Cal. July 9-14, 1989.

[5] T.Berry, A.R.Daniels, R.W.Dunn. "Development of Real Time Power System Simulators", UPEC Sept. 23,1989, Belfast, U.K. pp. 429-432.

[6] R. Betancourt. "An Effieient Heuristic Ordering Algorithm for Partial Matrix Refactorization", IEEE Trans. on PWRS. Vol. 3, No. 3, August 1988.

[7] F. M. Brasch Jr., J.E. Van Ness and S.C. Kang. "Design of Multiprocessor Structures for Simulation of Power System Dynamics", EPRI Report EL-1756, March 1981.

[8] F. M. Brasch Jr., J.E. Van Ness and S. C. Kang. "Simulation of a Multiprocessor Network for Power System Problems", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-101, No. 2, February 1982.

[9] G.E. Bridges, D.M. Burek, W. Pries, C.W. Lee, R.D. McLeod, A.M. Gole, R.M. Mathur. "A Multiple Processor Based Power System Digital Simulator", IEEE 4th Intl. Conf. on AC and DC Power Transmission, No. 225, London, England, 23-26 Sept. 1985.

[10] H.Card. "Analog Circuits for Relaxation Networks", International Journal of Neural Systems, Vol. 4, No. 4 (December, 1993) pp. 359-379.

[11] B. Dembart, A.M. Erisman, E.G. Cate, M.A. Epton and H.W. Dommel. "Power System Dynamic Analysis", Rp 670-1 Final Report, EPRI EL-484, July 1977.

[12] Diane M. Detig. "Effects of Special Purpose Hardware in Scientific Computation with Emphasis on Power System Applications", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-101, No. 2, February 1982.

[13] H.W. Dommel. "Digital Computer Solution of Electromagnetic Transients in Single and Multiphase Networks", IEEE Trans. on PAS, Vol. PAS-88, No. 4, pp. 388-399, April 1969.

[14] H.W. Dommel, N. Sato. "Fast Transient Stability Solutions", IEEE Trans, on PAS, Vol. 91 pp. 1643-50, July/August 1972.

[15] L. Elder and M.J. Metcalfe. "An Efficient Method for Real- Time Simulation of Large Power System Distrubances", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-101, No. 2, February 1982.

[16] EPRI. "Parallel Processing Workshop", October 21-28, 1980.

[17] EPRI EL-566-SR Special Report. "Exploring Applications of Parallel Processing to Power System Analysis Problems", October 1977.

[18] EPRI EL-946. "Technology Assessment Study of Near Term Computer Capabilities and Their Impact on Power Flow and Stability Simulation Programs", December 1978.

[19] EPRI EL-947. "Evaluation of Multiprocessor Algorithms for Transient Stability Problems" , November 1978.

[20] EPRI EL-4610. "Extended Transient - Midterm Stability Package", Final Report , Project 1208, January 1987.

[21] John Fishburn. "Analysis of Speedup in Distributed Algorithms", UMI Research Press, 1984.

[22] J. Fong and C. Pottle. "Parallel Processing of Power System Analysis Problems Via Simple Parallel Microcomputer Structures", IEEE Transcations on Power Apparatus and Systems, Vol. PAS-97, No. 5, September/October 1978.

[23] N.Fujinami and Y.Yokote. "Naming and Addressing of Objects without Unique Identifiers", Technical Report: SCSL-TR-92-004, Sony Computer Science Laboratory, 1994 Open House CD-ROM.

[24] Edward Gehringer. "Capability Architectures and Small Objects", UMI Research Press, 1982.

[25] A. Gomez and L. Franquelo. "An Efficient Ordering Algorithm to Improve Sparse Vector Methods", IEEE Trans. on PWRS, Vol. 3, No. 4 November 1988.

[26] A. Gomez and L. Franquelo. "Node Ordering Algorithms for Sparse Vector Method Improvement", IEEE Trans. PWRS, Vol. 3, No. 1, February 1988.

[27] R.M.Gray. "Vector Quantization",IEEE ASSP Magazine, April 1984, pp. 4-28.

[28] D.Hammerstrom. "A Highly Parallel Digital Architecture for Neural Network Emulation", Proceedings of the International Conference on VLSI for AI and Neural Networks, ed. W.R.Moore and J.Delado-frias, Oxford Sept. 1990, to be published by Plenum Press.

[29] Y.Honda and M.Tokoro. "Reflection and Time-Dependent Computing: Expreiences with the R^2 Architecture", Technical Report: SCSL-TR-93-017, Sony Computer Science Laboratory, 1994 Open House CD-ROM.

[30] J.P. Hulskamp, S.M. Chan and J.F. Faxio. "Power Flow Outage Studies Using an Array Processor", IEEE Transcations on Power Apparatus and Systems, Vol. PAS-101, No. 1, January 1982.

[31] K. Hwang and F. Briggs. "Computer Architecture and Parallel Processing", McGraw-Hill Book Company, 1984.

[32] R.Jacobs, M.Jordan, S.Nowlan, G.Hinton. "Adaptive Mixtures of Local Experts", Neural Computation, Vol. 3, 79-87, 1991.

[33] Snyder L. Jamieson, D. Gannon, H. Siegel editors. "Algorithmically Specialized Parallel Computers", Academic Press, 1985.

[34] R. Joetten, T. Web, J. Wolters, H. Ring and B. Bjoernsson. "A New Real-Time Simulator for Power System Studies", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-104, September 1985.

[35] R. Johnson, B. Cory, M. Short. "A Tunable Integration Method for the Simulation of Power System Dynamics", IEEE Trans. on PWRS, Vol. 3, No. 4, November 1988.

[36] R. Johnson, M. Short, B. Cory. "Improved Simulation Techniques for Power System Dynamics", IEEE Trans. on PWRS, Vol. 3, No. 4, November 1988.

[37] Svetlana P. Kartashev editor, Steven I. Karaskev editor. "Designing and Programming Modern Computers and Systems", Prentice-Hall, INC., 1982.

[38] T. Kohonen. "The Self-Organizing Map", PROC. of the IEEE, Vol.78, No.9, September 1990, pp. 1464-1480.

[39] A. K. Krishnamurthy, S.C.Ahalt, D.E.Melton, and P.Chen. "Neural Networks for Vector Quantization of Speech and Images", IEEE Journal on Selected Areas in Communications, Vol. 8, No. 8, October 1990, pp. 1449-1457.

[40] R. M. Mathur and X. Wang. "Real-Time Digital Simulator of the Electromagnetic Transients of Power Transmission Lines", IEEE Trans. on Power Delivery, Vol. 4, No. 2, April 1989.

[41] S. R. Maxminn and R. J. Thomas. "Microprocessor Simulation of Synchronous Machine Dynamics in Real-Time", IEEE Transactions on Power Systems, Vol. PWRS-1, No. 3, August 1986.

[42] P. G. McLaren, R. Kuffel, R. Wierckx, J. Giesbrecht, L. Arendt. "A Real Time Digital Simulator for Testing Relays", IEEE Trans. on Power Delivery, Vol. 7 No.1, January 1992

[43] I. Page and W. Luk. editors. "Compiling Occam into FPGAs", "FPGAs", 1991, Abingdon EE&CS Books pp. 271-283.

[44] Sergio Pissanetzky. "Sparse Matrix Technology", Academic Press Inc. , 1984.

[45] C. Pottle. "Array and Parallel Processors in On-Line Computations", EPRI Report EL-2363, April 1982.

[46] R. Pritchard and C. Pottle. "High-Speed Power Flows Using Attached Scientific (Array) Processors", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-101, No. 1 January 1982.

[47] D. Pountain. "Transport-Triggered Architectures", Byte, Feburary 1995, pp.151-152, McGraw-Hill.

[48] Micheal J. Quinn. "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill, 1987.

[49] P. Ristanovic, M. Bjelogrlic and B.S. Babic. "Improvements in Sparse Matrix/Vector Technique Applications for On-Line Load Flow Calculation", IEEE Transcations on Power Systems, Vol. 4, No. 1, February 1989.

[50] G. Rosendahl, R. Wierckx, T. Maguire, D. Woodford. "A Parallel Machine for Real Time Power System Simulations", Canadian Conference on Electrical and Computer Engineering, Montreal, Quebec, Sept. 17-20, 1989, pp. 1086-1089.

[51] G. Rosendahl, T. Paille and R. McLeod. "In System Reprogrammable LCAs Provide a Versatile Interface for a DSP Based Parallel Machine", FPGAs, ed. W. Moore and W. Luk, Abingdon EE&CS Books, Abingdon, England, Sept. 1991, pp 392-400.

[52] G. Rosendahl, R. McLeod and H. Card. "Acceleration of Backpropagation Learning using a Parallel DSP Machine", Canadian Conference on Electrical and Computer Engineering, Quebec City, Quebec, Sept. 1991, pp. 44.2.1-44.2.4.

[53] G. Rosendahl and R. McLeod. "Measurement of RAM Access Timing Using FPGAs", Proceedings of the 6th Workshop on New Directions for Testing, Montreal, Quebec, May 1992, pp. 46-57.

[54] G. Rosendahl and H. Card. "An Associative Memory Processor Implemented in FPGA Technology", The First Canadian Workshop on Field Prgrammable Gate Arrays, Winnipeg, Manitoba, May 31 - June 2, 1993, pp. 15-1 - 15-9.

[55] G. Rosendahl, R. McLeod, H. Card. "Practical Considerations in Large System Designs Prototyped using Field Programmable Logic", Canadian Workshop on Field Programmable Devices, Kingston, Ontario, June 13-16, 1994.

[56] D.Rumelhart, G.E.Hinton, and R.J.Williams. "Learning Internal Representations by Backpropagating Errors", Nature, 323:533-536 1986b.

[57] M.A. Shanblatt and Y.Y.J. Leung. "The Promise of VLSI for Load Flow Computatin Enhancement", IEEE Transactions on Power Apparatus and Systems, Vol. PAS-103, No. 7, July 1984.

[58] J.A. Sharp. "Data Flow Computing", Ellis Horwood Limited, 1985.

[59] Jan A. Spriet and Ghislain C. Vansteenkiste. "Computer-aided modeling and simulation", Academic Press, 1982.

[60] Brian Stott. "Power System Dynamic Response Calculations", Proceedings of the IEEE, Vol. 67 No. 2, February 1979.

[61] M. Stubbe, A. Bihain, J. Baader. "Stag - A New Unified Software Program for the Study of The Dynamic Behavior of Electrical Power Systems", IEEE Trans. on PWRS, Vol. 4, No. 1, February 1989.

[62] M. Takatoo, S. Abe, T. Bando, K. Hirasawa, M. Goto, K. Kato and T. Kanke. "Floating Vector Processor for Power System Simulation", IEEE Transcations on Power Apparatus and Systems, Vol. PAS-104, No. 12 December 1985.

[63] H. Taoka, S. Abe and S. Takeda. "Fast Transient Stability Solution Using an Array Processor", IEEE Transcations on Power Apparatus and Systems, Vol. PAS-102, No. 12, December 1983.

[64] T.Tenma, Y.Yokote, and M.Tokoro. "Implementing Persistent Objects in the Apertos Operating System", Technical Report: SCSL-TR-92-015, Sony Computer Science Laboratory, 1994 Open House CD-ROM.

[65] J. Tiberghien editor. "New Computer Architectures", Academic Press, 1984.

[66] M.Tokoro. "The Society of Objects", Technical Report: SCSL-TR-93-018, Sony Computer Science Laboratory, 1994 Open House CD-ROM.

[67] P.Treleaven, M.Pacheco, M.Vellasco. "VLSI Architectures for Neural Networks", IEEE Micro, Vol. 9, No. 1, pp. 8-27, December 1989.

[68] D. Tylavsky and B. Gopalakrishnan. "Precedence Relationship Performance of an Indirect Matrix Solver", IEEE/PES 1989 Summer Meeting, Long Beach, Cal., July 4-14, 1989.

[69] J.E. Van Ness and G. Molina. "The Use of Multiple Factoring in the Parallel Solution of Algebraic Equations", IEEE Transcations on Power Apparatus and Systems, Vol. PAS-102, No. 10, October 1983.

[70] A. Viegas de Vasconcelos and G. Hosemann. "Transient Studies on a Multiprocesor", IEEE Transcations on Power Apparatus and Systems, Vol. PAS-103, No. 11, November 1984.

[71] G.T. Vuong, A. Valette. "A Complex Y-Matrix Algorithm for Transient Stability Study", IEEE Trans on PAS, Vol PAS-104, No. 12, December 1985.

[72] Y. Yokote. "The Apertos Reflective Operating System: The Concept and Its Implemenation", Technical Report: SCSL-TR-92-014, Sony Computer Science Laboratory, 1994 Open House CD-ROM.

[73] Y.Yokote. "Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach", Technical Report: SCSL-TR-93-014, Sony Computer Science Laboratory, 1994 Open House CD-ROM.

[74] Y.Yokote. "The New Mechanism for Object-Oriented System Programming", Technical Report: SCSL-TR-92-005, Sony Computer Science Laboratory, 1994 Open House CD-ROM.

[75] D. Yu and H. Wang. "A New Approach for the Forward and Backward Substitutions of Parallel Solution of Sparse Linear Equations-Based on Dataflow Architecture", PICA Conference, Seattle, Washington May 1989.

[76] D. Yu and H. Wang. "A Parallel LU Decmoposition Method, IEEE/PES 1989 Summer Meeting", Long Beach, Cal., July 9-14, 1989.

[77] D.A. Woodford, A.M. Gole, R.W. Menzies. "Digital Simulation of DC Links and AC Machines", IEEE Trans. on PAS, Vol. PAS- 102, pp. 1616-1623, June 1983.

[78] Authur Wouk, editor. "New Computing Environments: Parallel, Vector and Systolic", US Army Research Office SIAM, Philadelphia, 1986.