Control Structures for Mesh-connected Networks

Peter Edward Strazdins

February 1990

A thesis submitted for the degree of Doctor of Philosophy of the Australian National University

Some of the results of this thesis were established with the collaboration of Dr Heiko Schröder and Dr Patrick Lenders. The material of Section 3.1, Section 3.4 and part of Section 3.6 is based largely on Dr Schröder's work, and that of Sections 6.2 and 6.3 is based largely on Dr Lenders' work.

Much of the material of Chapter 2 discusses and builds upon the works of other authors.

Elsewhere, except where otherwise stated, the work described in this thesis is my own.

P. Strazdins

(Peter Edward Strazdins)

Preface

This thesis presents the results of a study of the topic of control structures for (large-scale) mesh connected networks. This topic is part of the wider issue of making large-scale parallel architectures efficiently programmable.

The course of this study was from February 1987 to February 1990 at the Computer Sciences Laboratory of the Research School of Physical Sciences in the Australian National University. The work reported herein was supervised by Dr Heiko Schröder; and Professor Richard Brent acted as an advisor.

Historical context

The research presented herein began by making a case study of a boolean Instruction Systolic Array (ISA), considering the design issues for the ISA itself and its data and program interfaces. The results of this study eventually became Chapter 2. It became clear that a boolean ISA would be limited by the input bandwidth of the program information, from which came the idea of *program compression* to reduce this bandwidth. By considering run-length encoding representations of ISA programs, the ISAC method of program compression arose. Also, having fixed an instruction set for the boolean ISA, it became apparent that the flexibility and/or performance for some algorithms could be improved by using more powerful instructions. This led to application of *microprogramming* techniques to the ISA to provide this flexibility and program compression.

Dr Schröder suggested that the already existing concepts of the *Single Instruction Systolic Array* and the ISA *Subroutining* high-level language could also be regarded as methods of ISA program compression. Based on these four program compression methods, a joint paper [57] was written with Dr Schröder, and became a prototype of Chapter 3. However, the non-program compression aspects of ISA microprogramming required further development (Chapter 4). Also, an implementation of the Subroutining program compression method had to be developed, from which Chapter 5 arose. Application of (ISAC-like) program

compression methods to more powerful meshes than the microprogrammed ISA became the subject of Chapter 7.

Dr Patrick Lenders suggested that the weakest precondition semantics for the ISA model that he developed jointly with Dr Schröder could be developed similarly for the more general microprogrammed ISA model. This resulted in a joint paper [36]. From this research, and the desire to develop a practical proof method for (microprogrammed) ISA programs, written in *wavefront*-based languages such as ISA Subroutining, Chapter 6 arose.

Publications arising from this study

Preliminary results from this study were presented at the 12th Australian Computer Science Conference ACS-12, Wollongong, February 1987 and at the 22nd Conference on Microprogramming MICRO-22, Dublin, August 1989, and published in the respective conference proceedings [57,36].

Outline and main results

A brief introduction to mesh-connected network issues is given in Chapter 1.

Chapter 2 provides motivations for development of control structures for program compression and ISA microprogramming. The main results of this chapter are the design of a boolean ISA instruction set and a modular and partitionable mesh data interface design which also supports partitioning.

Chapter 3 introduces the concept of program compression for the ISA, and demonstrates that it can significantly reduce the overall system architecture's cost and the potentially performance-limiting program input bandwidth of an ISA. These methods may be used in combination to optimize compression rates, hardware overhead and flexibility, according to an ISA system's requirements and budgets. These techniques can also be applied to SIMD arrays using 'row/column vectors'.

Chapter 4 develops the microprogrammed ISA as a generalization of the ISA, and practical algorithms are given to demonstrate its greater flexibility and/or efficiency than the ISA. The theoretical relationship of the microprogrammed ISA

and other models of meshes is formally established. General macro structures which enable a microprogrammed ISA to efficiently simulate an ISA with an (almost) arbitrary instruction set and communication power are also given. This enables an efficient and very flexible microprogrammed ISA implementation of a high-level ISA programming model.

Chapter 5 demonstrates how 'optimal' ISA program compression can be made very efficient for moderate to large-scale ISA systems, by using the *wavefront* as a basis for program compression. This has the added practical advantage of being very compatible with high-level ISA languages, which are also *wavefront*-based.

Chapter 6 gives a formal semantic definition of the microprogrammed ISA, together with a practical proof method for (microprogrammed) ISA programs written in *wavefront*-based languages. The method yields compact and rigorous proofs, whose size compares favourably with the proofs for corresponding uniprocessor programs. Both the semantics and the proof method can be easily extended to deterministic and synchronous meshes using *wavefront*-based languages.

Chapter 7 applies program compression techniques to Processor Arrays. The concept of *wavefront interleaving* is developed, which increases flexibility and/or decreases program memory size. An evaluation and analysis shows that these program compression techniques can reduce program memory area and the $O(n)$ delays due to program loading.

Chapter 8 draws conclusions from this work.

Acknowledgements

I wish to thank most of all Dr Heiko Schröder for his excellent supervision in the course of my research, especially for his practical suggestions, constructive criticisms, and his encouragement. I wish also to thank my advisor Professor Richard Brent for his valuable advice and many helpful suggestions.

I am very grateful to the following people: Dr Iain Macleod, for his many clarifications and recommendations on the text of this thesis, Dr Patrick Lenders, for his collaboration and constructive criticism contributing towards Chapter 6, and all the staff and students of the Computer Sciences Laboratory, for their friendship over the last three years.

I also thank my parents Atis and Yvonne for their encouragement and support throughout my whole education, as well as my wife Mayumi for her love and support during this degree.

Finally I gratefully acknowledge the generous financial support of the Australian Government for this degree.

Abstract

To date, control structures for mesh-connected networks have only been adequately developed for SIMD meshes or small-scale meshes. With the recent improvements in VLSI, control structures for large-scale, powerful models of meshes must now be considered.

From the context of confident and efficient programmability, this thesis develops suitable control structures for large-scale, communication register-based Instruction Systolic Array (ISA) and Processor Array (PA) meshes. These structures all match the (systolic) mesh's suitability for VLSI, have a modest area overhead, and can significantly improve overall mesh performance, reduce overall mesh hardware and/or improve mesh flexibility.

Control structures to implement program compression are developed; these structures permit the efficient storage and loading of mesh programs.

Various approaches for program compression are developed for $n \times n$ ISAs. These approaches can also be applied to SIMD meshes. The *wavefront*-based approach is not only compatible with high-level ISA languages, but also achieves generality, extremely high program compression rates and a high program loading performance with only a modest $O(n)$ area control structure overhead.

Moreover, combining this approach with microprogramming techniques is shown to increase the ISA's flexibility on algorithms using nontransmittent data. For this purpose, a generalization of the ISA model, called the microprogrammed ISA, is developed. It permits abstraction from the ISA's instruction set and forms an important concept for high-level ISA programming. Techniques developed for the microprogrammed ISA demonstrate how to efficiently simulate, on a basic orthogonally-connected mesh, algorithms based on more powerful array topologies. A formal definition of the microprogrammed ISA is given using weakest precondition semantics. From this, a proof technique is developed which enables convenient manual verification for microprogrammed ISA programs written in diagonal-based languages. The semantics and proof method can be extended to

other models of synchronous, communication register-based meshes.

The application of program compression techniques to $n \times n$ Processor Arrays is shown to reduce the cell program memory area while also reducing program loading delays. These delays are $O(n)$ due to mismatches between the propagation of the loading and execution 'activities' of PA programs. A *vertical interleaving* technique is introduced which can further reduce cell program memory area and improve PA flexibility with a small control logic overhead. These techniques are particularly suitable for fine to medium-grained PAs.

This thesis develops various practical aspects of the ISA, and demonstrates that using a simple cell design, the ISA can efficiently implement large classes of algorithms. The contributions reported here reduce the attractiveness of the SIMD mesh model since they mitigate the ISA model's relative disadvantages of extra program input bandwidth and the critical nature of its instruction granularity, while making the ISA model even more powerful. They also improve the programmability aspect of fine to medium-grained Processor Arrays, making them more attractive for constructing large-scale parallel architectures.

Contents

Preface ii

Acknowlegements v

Abstract vi

Contents viii

List of figures xv

List of tables xix

1 Introduction **1**

1.1 Why meshes for parallel computing? 3

1.1.1 Fine-grained or coarse-grained meshes? 5

1.1.2 Limitations to mesh size 6

1.1.3 Approaches for meshes 8

1.2 The Instruction Systolic Array 9

1.2.1 The structure of the ISA 10

1.2.2 Systolic concepts and terminology 12

1.3 Themes 13

1.4 Reading guide 15

2 General ISA Design Issues **18**

2.1 Introduction 18

2.2 Case study: the boolean ISA, BISA 19

2.2.1 Instruction set and communication modes 20

2.2.2 Some boolean algorithms on the BISA 24

2.2.2.1 The Red Squares program 25

2.2.2.2 The matrix multiplication program 27

2.2.2.3 The pattern matching program 29

2.2.2.4 Transitive closure program 34
2.2.2.5 Associative memory lookup program 37
2.2.2.6 Median finding program 38
2.2.3 Limitations of the BISA 43
2.3 ISA instruction granularity 44
2.4 Internal memory size 46
2.5 Matching problem size to the array size 47
2.5.1 Small matrix sizes on large ISAs 48
2.5.2 Partitioning: large matrix sizes on smaller ISAs 49
2.5.2.1 LSGP partitioning for the Red Squares program . 49
2.5.2.2 LPGS partitioning 50
2.5.2.3 Comparison of partitioning methods for the ISA . 52
2.6 Data interfaces for the ISA 53
2.6.1 Existing mesh data interfaces 54
2.6.2 Data interface properties 56
2.6.3 A partitionable instruction systolic data buffer design . . . 58
2.7 Conclusions 62
2.A Appendix: The BISA prototype, bISA 63
2.A.1 General description 63
2.A.2 Informal description of the instruction set 64
2.B Appendix: LSGP partitioning for the Red Squares program 65

3 Program Compression on the Instruction Systolic Array 68
3.1 Introduction 68
3.2 Program compression — related work and concepts 71
3.2.1 Data compression on meshes 73
3.2.2 Program compression on existing meshes 74
3.3 Motivations for compressing ISA Programs 75
3.4 Program compression using the SISA 79
3.5 Program compression using microprogramming 81
3.5.1 Implementation of microprogramming 82
3.5.2 A microprogrammed transitive closure program 83

3.6 Program compression using Subroutining 85
3.7 Program Compression using ISAC 87
3.7.1 Examples of ISAC coding 88
3.7.2 Definition of ISAC 90
3.7.3 Matrix restorers for ISAC 92
3.8 Evaluation of program compression methods 95
3.8.1 Compression rates 95
3.8.2 Hardware overheads 96
3.9 Application to other mesh architectures 98
3.10 Conclusion 99
3.A Appendix: Implementation of ISAC 102
3.A.1 ISAC table traversal algorithm 102
3.A.2 ISAC implementation of general loops 103
3.A.3 ISAC implementation of divide-and-conquer programs . . . 104

4 The Microprogrammed Instruction Systolic Array 107

4.1 Introduction 107
4.1.1 The microprogrammed ISA model 109
4.1.2 ISAs vs. SIMD meshes and microprogramming 110
4.1.3 Motivations for microprogramming the ISA 112
4.2 Examples of microprogramming the ISA 113
4.2.1 The LCS algorithm 113
4.2.2 Transitive closure algorithm revisited 116
4.3 Implementation of the microprogrammed ISA 118
4.3.1 Implementation of variable length queues 119
4.4 The microprogrammed ISA in relation to other meshes 120
4.4.1 Establishing the relationship 121
4.4.2 The microprogrammed ISA timing rules 123
4.4.3 Simulation of ISA programs 123
4.5 Givens Rotations: a microprogrammed ISA case study 130
4.5.1 Description of the systolic algorithm 131
4.5.2 ISA implementation using enhanced communications . . . 133

4.5.3 Timesharing for the four output register mode 135
4.5.4 Timesharing for the one output register mode 137
4.5.5 Optimizing the period 137
4.5.6 Factoring out the sine-cosine generations 141
4.6 Simulating an arbitrary instruction granularity ISA 141
4.6.1 Constraints on the macros' internal structure 142
4.6.2 A macro structure to emulate high instruction granularity 143
4.6.3 Microprogrammed Dynamic Time Warping 147
4.6.4 Extensions for the microprogrammed emulation of ISAs . . 148
4.7 Conclusions 150
4.A Appendix: Variable length microinstruction queues for VLSI . . . 153

5 Program Compression by ISA Diagonals **155**
5.1 Introduction 155
5.2 Definition of ISADL 161
5.2.1 Constraints on ISADL for efficient program compression . 163
5.2.2 Example: row reversal revisited 165
5.3 Implementation of ISADL 167
5.3.1 The diagonal sequencer 167
5.3.2 The diagonal restorer 168
5.3.2.1 Diagonal fetch/shift logic 168
5.3.2.2 The diagonal loading logic 169
5.3.3 Optimizing program loading and storage area 171
5.3.4 Representation of ISADL programs for loading 174
5.4 Extensions of ISADL 175
5.4.1 Relaxing the ISADL constraints 175
5.4.1.1 Nested loops: Constraint 1 175
5.4.1.2 Divide-and-conquer programs: Constraint 2 . . . 176
5.4.1.3 'Flat' ISA programs: Constraint 3 180
5.4.2 Combining ISADL with the SISA 182
5.4.3 Incorporating subroutines into ISADL 183
5.4.4 Combining ISADL with microprogramming 183

5.5 Evaluation of ISADL 185
5.5.1 ISADL requirements for various ISA algorithms 185
5.5.2 ISADL memory and I/O bandwidth requirements 186
5.5.3 Conclusions 189
5.A Appendix: Boolean matrix algorithms in ISADL 191
5.A.1 Red Squares program 192
5.A.2 Matrix multiplication program 192
5.A.3 Pattern match program 193
5.A.4 Transitive closure program 193
5.A.5 Associative memory lookup program 194
5.A.6 Median finding program 195
5.A.7 ISADL macro rules 196
5.B Appendix: Implementation of ISADL 196
5.B.1 The diagonal load algorithm 196
5.B.2 Transforming 'flat' ISA programs 200
5.B.3 Binary recursive programs in ISADL 200

6 A Micro-level Semantics for the Microprogrammed ISA 203
6.1 Introduction 203
6.2 Syntax of an ISA microlanguage 206
6.3 Semantics for the ISA microlanguage 208
6.4 A programmer's view of ISA program verification 211
6.5 A wavefront-based proof method 213
6.5.1 Example: the LCS program 216
6.5.2 Validity of the proof method 218
6.5.3 Interpretation of the communication register time-stamps . 221
6.5.4 Incorporating ISADL into the method: a semantics for ISADL 222
6.6 Using the proof method 223
6.6.1 The LCS program revisited 225
6.6.2 The Red Squares program: simple, uniform iterations . . . 225
6.6.3 The matrix input program: non-uniform iterations 227

6.6.4 The matrix output program: non-uniform iterations 228
6.6.5 The row swap program: complex iterations 230
6.7 Discussion and conclusions 231
6.7.1 Discussion of the proof method 232
6.7.2 Future research on mesh program verification 235

7 Program Compression for Processor Arrays 237
7.1 Motivations 237
7.1.1 Simple examples of PAC programs 240
7.1.2 Program compression on existing PAs 243
7.1.3 Alternative approaches for PAC 247
7.2 Implementation of PAC 249
7.2.1 Definition of the PAC constructs 249
7.2.2 A PAC program interface 250
7.2.3 Transformation into PACl 251
7.2.4 Implementation of PACl 254
7.2.5 Array synchronization using PAC 256
7.3 PAC on more complex PA programs 257
7.3.1 Matrix multiplication 257
7.3.2 Matrix transpose program 258
7.3.3 Warshall's transitive closure algorithm in PAC 261
7.3.4 Kung's transitive closure algorithm in PAC 264
7.3.5 Simulating skewed matrix input on a PA 265
7.4 Extensions to PAC 267
7.4.1 Implementation of wavefront interleaving 267
7.4.1.1 Simulation of an ISA using horizontal interleaving 270
7.4.1.2 Matrix transpose program revisited 271
7.4.2 Extensions to PACl 272
7.4.3 Widening the domain of PAC 272
7.5 Evaluation of PAC 274
7.5.1 Load-time vs. run-time evaluation of cell position-dependent code 274

7.5.2 Program loading and execution: serial vs. overlapped . . . 277
7.5.2.1 Calculation of the PCU table buffering factor . . 280
7.5.2.2 Harmonizing program loading and execution wavefronts: an efficient concept for program loading . 281
7.5.3 Conclusions regarding a viable domain for PAC 283
7.A Appendix: Definition of PAC 286
7.B Appendix: The PAC transitive closure program compressed 287

8 Conclusions **289**

Bibliography **294**

List of Figures

1.1 A 3×4 instruction systolic array (ISA), with program, comprised of instruction part (top), and selector part (left), of period 6 . . . 11

2.1 Sub-programs for program RedSquares on a 4×4 BISA 26
2.2 Program MatMult for a 3×5 ISA 29
2.3 Program MatMult for a 3×5 BISA 30
2.4 Sub-programs of program Match^4 on a 4×4 BISA 32
2.5 Initial data configuration for transitive closure program on a 4×4 BISA 35
2.6 Transitive closure program on a 4×4 BISA 36
2.7 Instruction part for (the sub-programs of) program AssocMem for $m = 3$ for a 9×9 BISA 39
2.8 Instruction part for (the sub-programs of) program MedFind for a 4×4 BISA 41
2.9 Initial data buffer and ISA configuration for column-major LPGS scheduling of the transitive closure program for partition (i', j') . . 51
2.10 An area-efficient ISA data interface, for a 4×4 ISA 55
2.11 A VLSI data interface for a sorting chip (mesh) 56
2.12 Instruction systolically programmable data interface for a 3×3 ISA 59
2.13 Programming the western data buffer channel for loading a matrix into a 4×4 ISA 61
2.14 LSGP partitioning of the RedSquares program around cell $(1, 1)$ for $\kappa = 4$ 66

3.1 Ringshift program for a 6×6 ISA 71
3.2 C registers before and after the ringshift program for a 6×6 ISA 71

3.3 A 10×10 ISA program interface 77
3.4 Ringshift program for a 6×6 SISA 80
3.5 Instruction decode tables on a 4×4 ISA 83
3.6 Transitive closure algorithm (one pass) on a 4×4 ISA 84
3.7 ISA/SISA programs for rotation 85
3.8 SISA program for vertical and horizontal rotation by d positions . 86
3.9 Transitive closure program for an $n \times n$ ISA 87
3.10 LoadMat program for a 4×4 ISA 88
3.11 RowRev program for a 4×4 ISA 90
3.12 Tabular encoding of '$((a\ b)^3\ c)^2\ (d)^4$' 93
3.13 Tables in matrix restorer cell j for divide-and-conquer row reversal programs 105

4.1 Microprogrammed LCS program on a 3×3 μISA with $\lambda = 4$, $\mu = 3$, $\sigma = 2$ and $\bar{\eta} = 1$ 115
4.2 Transitive closure program using timesharing of the C register for a 3×3 normal μISA with $\mu = \sigma = 5$ and $\bar{\eta} = 1$ 117
4.3 Simulation relationships between various models of processor meshes, with their wavefront parameter restrictions. 122
4.4 A $(4,2)$ wavefront μISA program simulating an ISA program (instruction part) 125
4.5 A $(3,3)$ wavefront μISA program simulating a $(2,2)$ wavefront μISA program 127
4.6 Givens Rotations block G_i on an enhanced communication ISA (normal μISA) 134
4.7 Givens Rotations block G_i on a μISA with two-way timesharing of eastern output register (with $\mu_1 = \mu_2 = 2$) 136
4.8 Givens Rotations block on a μISA with three-way timesharing of the single output register (illustrated for $\mu_1 = \mu_2 = 2$) 138
4.9 Optimized Givens Rotations block G_i on a normal μISA leaving result matrix in row-major order (with $\mu_1 = \mu_2 = 2$) 140
4.10 Macro structure for ISA emulation using timeharing 146

4.11 Dynamic Time Warping program on a normal μISA using a single output register C, with $\lambda = 8, \mu = 5$ and $\bar{\eta} = 3$ 149
4.12 Variable length queue corresponding to $\mu\ (\sigma) = 4 + 1 = 5$ 154

5.1 LoadMatD program, instruction part (selectors are '1's) 159
5.2 TransClos program, instruction part (selectors are similar) 160
5.3 RotH_d program, instruction part (selectors are '1's) 161
5.4 Implementation (initial state) for RotH_4 on 8×8 ISA 164
5.5 RotH'_{2^k} program (selectors are '1's) 166
5.6 Diagonal restorer for RotH'_8 on an 8×8 ISA, time steps 1-4 . . . 167
5.7 Single iteration of RotH_8 program illustrating a *diagonal pattern* requiring $N_{\text{p}} = 3$. 172
5.8 RotH'_8 program with single diagonal pattern (named d_1) 173
5.9 Program DivConq, showing the spliced diagonal technique 177
5.10 'Flat' LoadMat program, instruction part 181
5.11 'Forward' triangle merge (over width d) instructions for an $n \times n$ ISA 200

6.1 Execution of simplified Red Squares program around 'typical' cell $(i, j) = (2, 2)$ for a 4×4 ISA 212
6.2 Normalization Lemma: visualization of $(\forall_{i,j} s1_{ij}^{k_{ijt}})$ around cell (i, j) for the LCS program 220
6.3 UnLoadMat program, instruction part, on a 4×4 ISA 229

7.1 Instructions executed at each cell for $(\text{Compute}A_k)^2$ program for a 3×3 PA. 241
7.2 Instructions executed at each cell for programs with cell-position dependent iterations for a 3×3 PA. 243
7.3 SymEigen program on a 5×5 PA 244
7.4 PAC program interface for a 4×4 PA 251
7.5 MatMult program for a 3×5 PA 258
7.6 Execution of MatTrans1 program on a 5×5 PA, time steps 1-9 . 259
7.7 MatTrans1 program for a 5×5 PA 260

7.8 Horizontal communication phase of Transitive Closure program on one row of a 5×5 PA 262
7.9 Simulating input of A'_i for the Transitive Closure program on row i of a 5×5 PA 266
7.10 Number of superfluous instructions/cell for program MatTrans1 on a 5×5 PA 272
7.11 Program interface harmonizing program loading and execution wavefronts on a 4×4 PA 282

List of Tables

2.1 Comparison of LSGP and LPGS partitioning methods for the ISA 52

3.1 Program compression ratios for ISA with $n = 8, w_i = 8$ 97

3.2 Hardware overheads for program compression methods (in terms of static storage bits per diagonal restorer cell) for a $2^5 \times 2^5$ ISA with $w_i = 8$ 98

4.1 Timing rules for a (μ, σ) wavefront microprogrammed ISA 124

5.1 ISADL parameter values for various ISA programs 187

5.2 Diagonal restorer cell hardware requirements 188

7.1 Load-time vs. run-time PAC program evaluation 276

Chapter 1

Introduction

The advent of Very Large Scale Integration (VLSI) technology has provided vast improvements in digital computer performance over the last two decades. Somewhat paradoxically, an ever increasing need for more computing power continues, since VLSI has allowed the processing of increasingly larger sets of data and has opened new computationally-intensive fields of research. Since the switching times of transistors are now decreasing only moderately, large-scale parallelism will be the only way to gain substantial speedup, and this has become feasible with the availability of low-cost, high-density VLSI devices coupled with computer-aided VLSI design facilities. Thus, an important problem is to design large-scale parallel computer architectures and corresponding algorithms that are realizable in present and near future technology, ie. VLSI and Wafer Scale Integration (WSI).

At present, the applications demanding such computing power for the most part fall into two categories: scientific/engineering computations requiring the use of supercomputers, and digital signal processing. Real-time processing is often important for the latter, and desirable for the former, placing a strict lower limit on acceptable computer performance. Flexibility (or programmability) is essential for the former and is often essential for the latter, eg. real-time vision processing systems which call hundreds of subroutines during normal use [10, p1]. A problem lies in designing computer architectures which can implement (large classes of) these applications with acceptable performance.

The mesh-connected network (also referred to more simply as a *mesh* or an *array*) [22, 26] is a promising solution to the compound problem of designing flexible, high performance computer architectures that are also suited to large-scale VLSI and WSI implementation. A mesh-connected network is a two dimensional array of identical processing elements, in which any processing element can directly communicate only with its physically adjacent neighbours.

However, even for meshes, the general parallel programming issue of confident[1] and yet efficient programmability remains. This issue is one of the most outstanding problems for parallel computing today [39][2, p2] [27, p1][26, p417] and can be divided into the following aspects:

- what high level language concepts, including abstraction from parallel architecture details, should be used to ease programmability?
- what *control structures*, ie. hardware inside and at the periphery of the mesh, are required to support flexible and efficient programmability?
- what testing and verification techniques should be used to determine the correctness of parallel programs?

Since few large-scale parallel architectures with significantly more flexibility than the SIMD approach have been built, the design of efficient control structures for systems of such size is only just being considered.

This thesis deals with the above issue from the point of view of *communication register-based* meshes (defined in Section 1.1), concentrating on the *control structures* aspect. It thus examines the question of how to design meshes to have high flexibility, with minimal overhead in terms of time and area, so that large-scale (ie. massively parallel) meshes may be used to very competitively implement supercomputing and digital processing applications in the future.

Specifically, *control structures* refer to hardware components manipulating and determining the flow of an architecture's control (program) information. However, in a broader sense, *control structures* may refer to hardware components supporting any (high-level language) programming feature.

[1] ie. parallel programs can be written and expected to exhibit their intended performance.

Section 1.1 discusses the merits of the mesh as a candidate for the most promising semi-general to general-purpose architecture for massively parallel computation. Section 1.2 introduces the Instruction Systolic Array (ISA), a programmable mesh used extensively in subsequent chapters. Section 1.3 discusses the themes of this thesis and Section 1.4 gives a reading guide.

1.1 Why meshes for parallel computing?

The subject of this thesis is mesh-connected networks, in particular orthogonally-connected meshes, such as the ISA. These meshes have interesting possibilities for control structures, and yet have an efficient VLSI implementation. A mesh of m rows and n columns will be called an $m \times n$ array.

Included here in the concept of a mesh is a torus (a mesh in which the boundaries 'wrap-around' to meet each other). This is because a torus can also be efficiently implemented in VLSI.

The *communication register-based* model for mesh communication is assumed here. In this model, a mesh processing element or *cell* can communicate with adjacent cells only by reading their communication registers (see Section 1.2 for more details). This model is simple, read/write conflict-free and can model the higher-level message-passing concept of communication. Thus, it is sufficiently general for the discussion of mesh control structures. It is also convenient to assume synchronous communication is used, although most concepts presented in this thesis can be adapted for asynchronous communication.

This thesis is concerned mainly with fine- to medium-grained meshes because they are (currently) better candidates for large-scale parallelism, and because their programmability issues (and hence control structures) are simpler.

Before discussing the relative merits of meshes over other parallel architectures, the need for large scale parallelism should be reviewed. Quinn [44, pp18-21] gives a listing of popular arguments against the merits of large-scale parallelism, and provides rebuttals for these arguments.

Only one such argument, Amdahl's law [44, pp45-47], is of particular significance to this thesis. Amdahl's law implies that even a small fraction f, $0 \leq f \leq 1$,

of sequential operations can limit the overall speedup afforded by parallel computation. For a parallel computer of p identical processors, it states that the speedup S over a one-processor system is limited by:

$$S \leq \frac{1}{f + (1-f)/p} \tag{1.1}$$

While this law is useful in exploring whether a particular algorithm is suitable for parallelization, it does not imply that large-scale parallelism is not worthwhile, since parallelism enables computations to operate on larger and larger data sizes n (in proportion to p), and for many computations $f^{-1} = \Omega(n^a)$, where $a > 0$.

Amdahl's law has an important corollary: that any successful massively parallel architecture needs a powerful (host) processor which is capable of extremely fast sequential computations. Such a host processor is implicitly assumed for the meshes discussed in this thesis.

Meshes have an advantage over other candidates for massively parallel architectures in that they exploit all of the VLSI architectural design principles [26, p10] such as modularity, balance between I/O and computation (see also [25]), simple and regular data/control paths, and localized/reduced interconnections. This last principle is important since VLSI communication is restrictive, and is expected to become more restrictive with the scaling down of VLSI devices [40, pp35-37]. As long as this situation remains, meshes will continue to be a parallel architecture of great importance.

For currently available commercial large-scale parallel computers, the main rival of meshes is the hypercube architecture. A hypercube of N cells requires $\log N$ steps for communication between arbitrary cells, whereas a square, two-dimensional torus of N cells requires $\sqrt{N}$ steps. For $N \leq 256$, it can be seen that in terms of communication, the hypercube can give at most twice the performance (in terms of number of steps) and so its extra complexity over the torus is difficult to justify. For larger values of N, the delay and cost associated with the long wires of the hypercube, not to mention its lack of expandability (each node of the hypercube requires $\log N$ connections) has led Fujitsu to choose the torus over the hypercube for its CAP array processor [19], and similarly the new generation

of Intel parallel machines[2] are mesh-based.

More powerful architectures than the hypercube (such as shared memory networks) suffer even more seriously from these drawbacks when they are made large-scale.

The above arguments justify choosing the mesh as a promising candidate for large scale parallelism, and hence as the subject for this thesis. Section 1.1.1 discusses the issue of fine-grained meshes vs. coarse-grained meshes, and Section 1.1.2 discusses how large a scale can we expect meshes to be implemented in current and near-future technology. Section 1.1.3 discusses various models for meshes that are relevant to this thesis.

1.1.1 Fine-grained or coarse-grained meshes?

Having focused on mesh architectures, what granularity of the mesh cells should be chosen? This issue is open, but it can be at least said that fine- to medium-grained meshes potentially offer more computing power and occupy a significant niche in supercomputing and digital signal processing applications.

The finer the mesh, the more raw computing power is available, I/O limitations permitting. The argument runs as follows: while for a given cost (area), a coarse-grained mesh's smaller number of cells may be compensated by its cells having more powerful instructions, the fine-grained mesh's cells are smaller and therefore have a shorter instruction cycle. A similar argument influenced the Connection Machine design to use fine-grained bit processors, even for typical use on 8 bit or 32 bit data [14, p50].

However, the finer the granularity, in general, the lower the flexibility. Thus, for example, high-level programming constructs which simulate various computing topologies and algorithms requiring large amounts of local storage in each cell may be difficult or impossible to implement efficiently on a fine-grained mesh. Large amounts of local storage may also reduce I/O limitations of a mesh [25].

Generally, the applications intended for a mesh determine its granularity. Large classes of applications, particularly in the field of digital signal processing,

[2]These machines can simulate hypercube and other topologies in software.

require only fine granularity. Implementing these applications on coarse-grained architectures can be wasteful in terms of area.

1.1.2 Limitations to mesh size

How large an $n \times n$ mesh (using local control and data paths, such as the Instruction Systolic Array) can (feasibly) be constructed? This section gives an overview of this issue, and concludes that meshes whose dimensions are of the order of a thousand or more may indeed be feasible, provided the I/O bandwidth between the mesh and external (mass) storage can be made sufficiently high. Factors possibly limiting (systolic) mesh size include:

- an $o(n)$ I/O bandwidth[3] (between mesh and external data memory).
  For matrix computations, the typical period is $\Theta(n)$. If only an $o(n)$ I/O bandwidth could be afforded for a mesh, the time to take to load an $n \times n$ matrix would exceed $\Omega(n)$, dominating the computational period. This means that increasing the array beyond a certain size will result in no more overall speedup.

  More generally, Amdahl's law states that in an algorithm has a fraction f of sequential operations (eg. performed by the host machine or an I/O device), the maximum speedup achievable is f^{-1} (see equation (1.1)), even with unlimited mesh size. However, mesh applications using systolic algorithms on $n \times n$ matrices typically have $\Omega(n^3)$ mesh operations, and might in the worst case have $O(n^2)$ sequential operations (ie. for loading the matrices using an $O(1)$ I/O bandwidth), so that $f^{-1} = \Omega(n)$. In such a case, the overall speedup increases with at least the square root of the number of processors, rather than the number of processors, so that increasing mesh size eventually will have a diminishing return.

  However, it may still be that a $\Theta(n)$ I/O bandwidth can feasibly (although perhaps expensively) be engineered, in which case the array size would not be limited.

[3]See also Section 2.4. Note that '$o(n)$' means '$O(n)$ but not $\Theta(n)$'.

- global clock skew constraints.

  Systolic systems, exploiting the efficiency of local signal propagation, still rely on a global signal: the clock pulse. For large meshes, the skew of a global clock signal may become a limiting factor [9][26, p301]. However, hybrid (local/global) synchronization schemes can avoid this problems for (two-dimensional) meshes [9], and also meshes can be implemented using the more expensive asynchronous communication as is used by Wavefront Array Processor (WAP) meshes [26] and asynchronous counterparts of the ISA [20, ch.6]. In either case, clock skew need not be a serious limitation to mesh size.

- circuit board packaging constraints.

  Packaging technology requires that the circuit boards making up a parallel computer be packed into a three-dimensional volume. In a large mesh, made up of a large number of circuit boards, configuring these boards so that logical nearest-neighbour connections are always implemented by short wires may be difficult. However, by choosing a (physically) cylindrical or toroidal configuration for the circuit boards, as is done for the PAX (mesh) computer [56, p64], this problem can be overcome.

Wafer Scale Integration (WSI) is seen as a technology particularly suited to meshes [26, pp384-385][36], due to their local interconnection property. This is because in such levels of integration, long wires have propagation delays proportional to at least their length unless they are driven by large, power-inefficient fanout buffers [45]. Either way, long wires extending across a wafer would incur a high cost, making architectures such as hypercubes less suitable for WSI. Problems of clock skew are also reduced using WSI. WSI allows an order of magnitude increase in chip packing densities, so that 512×512 or even 1024×1024 meshes can (feasibly) be constructed. However, major technical difficulties such as power dissipation and fault tolerance still need to be resolved for WSI to become practical.

1.1.3 Approaches for meshes

One approach for a mesh model suited to VLSI is the concept of the systolic array, introduced by Kung and Leiserson [22]. A systolic array is a synchronous system consisting of a large number of simple processing elements (cells) where one or more streams of data are shifted through with constant speed and direction. Each of the cells is capable of executing one function only on the data it receives from its direct neighbours.

This leads to a significant drawback, ie. systolic arrays lack flexibility and are often referred to as "algorithms cast in silicon". On the other hand, their advantages include simplicity of cells, regularity of layout and locality of communication. This makes the systolic array suitable for VLSI implementation and results in extremely fast processing.

Examples of systolic arrays are the well known hexagonal array for multiplication of band-matrices [37, pp69-74], a linear systolic array for pattern matching [12] and a mesh-connected systolic array for sorting [28]. All these architectures look quite different and are only adequate to the special algorithm they realize.

Other approaches, more flexible than the systolic array, include:

- the SIMD mesh [26, p147][17], in which a single instruction is broadcast to all cells, which may be masked according to the cell's position and/or status registers. This approach, while simple and regular, requires non-local propagation of control signals, and so is not well suited to VLSI. They are also not convenient for implementing the propagation of *skewed* matrices required by many systolic algorithms.

- the *Instruction Systolic Array* (ISA) [21], which is also a systolic architecture where the cells are programmable and instructions are provided to them in a systolic manner. Thus it is possible to execute different programs on ISAs. More flexibility is added by a masking mechanism which combines the stream of instructions with an orthogonal stream of selectors (a boolean matrix), so that an instruction is only executed if it meets a selector bit of value '1' (see Figure 1.1).

The architectural concept of the ISA (presented in Section 1.2) preserves the advantages of the systolic array such as simplicity, regularity and locality, whereas its main disadvantage, its lack of flexibility, is overcome.

- the Processor Array (PA), in which each mesh cell is independently programmable (ie. a MIMD mesh). This is the most flexible of all models, and machines such as the CMU Warp processor [1] and CMU PSC computer [10] can successfully implement a large variety of systolic (and other) algorithms. However, this approach sacrifices simplicity for flexibility, with cells requiring individual program storage.

- the Wavefront Array Processor (WAP) [26, Ch.5], which combines dataflow computing with systolic arrays. WAPs uses asynchronous communication which, while having a considerable overhead, eliminates the need for a global clock (required by systolic arrays). This makes WAPs attractive on very large scale systems. WAPs may be programmed like the ISA (except that instruction propagation is now asynchronous), but are currently implemented using asynchronous Processor Arrays, such as Imnos Transputer arrays.

1.2 The Instruction Systolic Array

The ISA is a new architecture for parallel computation which meets the requirements of VLSI [21] and is capable of efficiently executing a large variety of parallel algorithms, while having very simple cell control structures.

An ISA is capable of executing a large variety of different algorithms (see [35, Ch.5-6] for examples), even if every processor can execute only a few different instructions. Thus the ISA-concept leads to greater flexibility then the systolic array concept. While it has somewhat less flexibility than the Processor Array, the ISA requires much smaller area per cell and therefore can provide a larger degree of parallelism on a fixed area.

In [21] it has been shown that the ISA concept is equivalent to the (MIMD) Processor Array and furthermore that arbitrary systolic arrays can be implemented on ISAs without loss of efficiency. Since then many different instruction

systolic arrays have been designed [49, 42, 32, 51] and some of them are under way to be realized in hardware.

1.2.1 The structure of the ISA

The structure of an $m \times n$ *Instruction Systolic Array* is depicted in Figure 1.1. The cells in the ISA have a very simple control unit and no storage space for programs. Instructions are *pumped* through the columns of the array of cells from top to bottom. This matrix of instruction codes is called the *instruction part.* In addition, selectors ('0' or '1') are pumped through the rows of the array, from left to right. The pumping of instructions and selectors is implemented by very simple control structures (ie. shift registers). This matrix over $\{0,1\}$ is called the *selector part.* A '0' causes every cell in its row to stay passive (not identical to the 'NoOp'-instruction). A '1' causes every cell in its row to execute the instruction that has been provided to it.

Every cell has some local memory including a designated *communication register.* The array is synchronized by a global clock, and the execution of every instruction is assumed to take the same time.

Communication between cells is done in the following way. Each cell can read information from its four direct neighbours. To avoid read/write conflicts, the execution of instructions is done in two non-overlapping phases. If a cell needs data from one of its four direct neighbours, it reads that neighbour's communication register during the first phase of the execution of an instruction. Thus at most five cells can read from a communication register simultaneously. During the second phase of the execution of an instruction, every cell writes into its own communication register or its own internal registers. The open ended data links of the cells at the boundaries of the array are used for external input and output of data.

Let $\mathcal{I}$ be the set of instructions the cells can execute, including a 'NoOp' instruction which does not change a cell's memory contents. An ISA program P of period K consists of an instruction part, which is a sequence '$p^1, p^2, \ldots, p^K$' of n-tuples over $\mathcal{I}$, and also a selector part, which is a sequence '$s^1, s^2, \ldots, s^K$' of

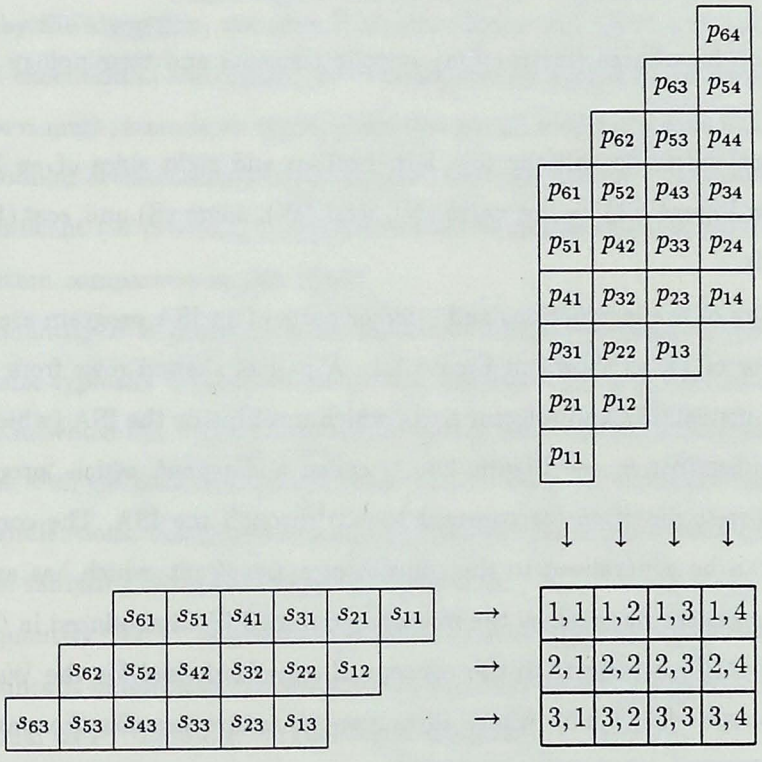


Figure 1.1: A 3×4 instruction systolic array (ISA), with program, comprised of instruction part (top), and selector part (left), of period 6

m-tuples over $\{0,1\}$. For $1 \leq i \leq m$, $1 \leq j \leq n$ and $1 \leq k \leq K$, selector s_{ki} and instruction p_{kj} are available at cell (i,j) at time $t = k+(i-1)+(j-1)$ (see Figure 1.1), where t is the global clock beat starting as the first instruction enters ISA cell $(1,1)$. Thus, the execution time T of program P is $T = (m-1)+(n-1)+K$ (instruction cycles).

1.2.2 Systolic concepts and terminology

This section introduces (instruction) systolic concepts and terminology used in this thesis.

It is convenient to call the top, left, bottom and right sides of an ISA (as depicted in Figure 1.1) as the *north* (N), *west* (W), *south* (S) and *east* (E) sides respectively.

The rows of the instructions and selector parts of an ISA program are *skewed* (by a factor of 1), as shown in Figure 1.1. A pair of skewed rows from an ISA program's instruction and selector parts which meet inside the ISA (which share the first subscript, k; see Figure 1.1) is called a *diagonal*, which 'propagates' in a south-west direction (at constant speed) through the ISA. The concept of diagonal can be generalized to the concept of a *wavefront*, which has arbitrary skew, for architectures such as the microprogrammed ISA introduced in Chapter 4. This is still less general to the concept of *wavefront* used for the Wavefront Array Processor [26, Ch.5], whose asynchronous nature permits the wavefronts to 'bend' due to local timing irregularities.

Transmittent data [26, p118] is data that is passed (systolically) unchanged through a mesh. *Nontransmittent* data, strictly speaking, is data that is updated as it passes through a mesh. However, when the mesh is an ISA, *nontransmittent* will usually refer only to such data when it is flowing southwards or eastwards, in pace with the instructions or selectors. It turns out that one of the main advantages of the ISA over the SIMD array is its superior handling of nontransmittent data.

Contraflow occurs in a mesh when streams of (transmittent or nontransmittent) data flow in opposite directions.

A (parallel) algorithm is said to be *compute bound* [25] when the the order of the basic computational operations (asymptotically) exceeds that of the basic input/output operations. Otherwise, it is called *I/O bound.* There is a limit to the degree of parallelism that can be efficiently employed with I/O bound algorithms.

Standard notations and concepts for asymptotic complexity of computer algorithms is used throughout this thesis. The relevant quantities are the area A required by the algorithm, the time T of execution of the algorithm, and the period P of execution of the algorithm[4]. For algorithms suited for implementation on an $n \times n$ mesh, bounds on these quantities are generally given as functions of n. For modular architectures such as meshes, the quantities A, P and AP are the most significant (in practice) measures of algorithm performance, and are used for algorithm comparison in this thesis.

The quantity P is generally more significant than T because in practice the problem size typically exceeds the mesh size, and algorithm partitioning methods are used, in which the overall time and period is determined by the algorithm's period on a single partition (see Section 2.5.2). Also, in real-time signal processing applications, the period is generally the most important, determining the maximum sampling rate of the sensory input data.

The quantity $(AP)^{-1}$ is generally regarded as the "figure of merit" for systolic devices, since it determines the architecture's performance (throughput, which is proportional to P^{-1}) for a given cost (area of silicon).

1.3 Themes

This thesis addresses the issue of confident and efficient programmability for large-scale, fine- to medium-grained meshes, concentrating on the control structures aspect. This is done with respect to an underlying *wavefront* programming model, which is motivated by the (instruction) systolic concept and elaborated in Chapters 4 and 7. In terms of the wavefront model, SIMD mesh and ISA programs use

[4] P is the minimum amount of time between consecutive executions of the algorithm. P never exceeds T.

fixed velocity wavefronts, microprogrammed ISA programs use wavefronts with a limited range of velocities, whereas Processor Array programs use combinations of arbitrary velocity wavefronts. The wavefront intuitively corresponds to the (simulated, in the case of the Processor Array) propagation of control information, ie. the propagation of a 'computational activity' or a 'sweep', through a mesh. Thus, the wavefront programming model has a natural relationship with the control structures required to support it. To demonstrate that this model is appropriate for high-level mesh languages and mesh program verification techniques is one aim of this thesis.

More specifically, this thesis considers which control structures, of modest hardware costs (compared with the mesh's overall system hardware costs) can significantly:

1. reduce overall mesh system hardware.
   eg. reduce the size of external and internal mesh program memory.
2. increase overall mesh performance.
   eg. minimize delays associated with loading new programs.
3. increase mesh flexibility.
   eg. support a more general mesh programming model.

Thus, a sub-theme of this thesis is to develop control structures which can significantly enhance the Instruction Systolic Array and (fine- to medium-grained) Processor Arrays, as these meshes efficiently support the wavefront programming model.

As well as the control structures (which include the program interface) for the mesh itself, those of its data interface must also be considered. These must make the data interface sufficiently programmable to unburden the mesh from low-level functions such as the shuffling and formatting of data (and hence enhance the mesh's overall performance).

Since each mesh cell may be independently programmable, the overall programs for large-scale meshes are potentially large. As a result, issues of how to externally store and fetch these programs so as not to degrade mesh performance

must be considered, taking into account the fact that I/O bandwidth is already a limiting factor for such parallel architectures. In practice, this can be resolved by the use of *program compression*, which can achieve the advantages of Items 1 and 2 above.

The program compression-related concepts of (ISA) *microprogramming* and wavefront *interleaving* can achieve the advantages of Item 3 above.

However, in the case of the wavefront programming model, the control structures aspect interacts with the high-level language and verification aspects of confident and efficient programmability.

Microprogramming can be also seen as a concept facilitating high-level mesh programming, since it provides abstraction from the instruction set and communication details. It thus extends the the wavefront programming model to the 'macro'-level.

Languages for program compression may be related to high-level mesh programming languages, which also requires the concise expression of mesh programs. This particularly applies to wavefront-based languages.

Applying the wavefront concept to verification techniques for meshes can also provide compact, ie. 'compressed', proofs of program correctness, with a structure reflecting that of the programs written in a wavefront-based program compression language. Where verification techniques also involve semantic definitions, they can contribute in another way to the issue of confident programmability by specifying the (actual) behaviour of mesh programs.

1.4 Reading guide

In the design of control structures for high-performance meshes, area efficiency is a crucial goal. Thus, it is important to consider the 'in-practice' requirements of mesh programmability, stemming from real-life examples (which are often non-trivial), so that extra area overhead is not introduced through unnecessary generality, while important special cases can still be implemented. Also, for programmable meshes, the average case performance is more important in practice than the more easily determined worst case performance. Such consider-

ations should be reflected in the design, analysis and evaluation of these control structures. Insofar as it is feasible, these requirements are considered in this thesis.

Furthermore, the control structures proposed here all extensively utilize the systolic concept, which yields hardware-efficient but often esoteric designs. This, combined with the requirements of the preceding paragraph, necessarily makes the subject matter of this thesis somewhat detailed and difficult to read.

Through the case study of the boolean ISA, Chapter 2 introduces the ISA concepts and issues which motivate the central themes of this thesis. It also presents the design, with suitable control structures, for an ISA data interface. A basic understanding of these ISA programming and design issues will be useful for the reading of the remaining chapters.

Chapter 3 is the key chapter for the program compression theme of this thesis, and presents four program compression methods for the ISA. Sections 3.5, 3.6 and 3.7 respectively introduce Chapters 4, 5 and 7.

Chapter 4 develops the microprogrammed ISA as an extension of the ISA model. Its basic concepts are required for a reading of its semantic modelling in Chapter 6. The microprogrammed ISA concepts of the *wavefront* and the macro are used in Chapter 7.

Chapter 5 develops wavefront-based program compression for practical implementation on a (microprogrammed) ISA. Chapter 6 gives a semantic definition of the microprogrammed ISA from which is developed a wavefront-based proof method — this effectively gives a semantic definition for wavefront-based (microprogrammed) ISA languages. Chapter 7 applies program compression techniques to Processor Arrays.

Chapters 5 to 7 are not required for the reading of any other core chapters of this thesis. In particular, Chapters 5 and 6 discuss rather involved low-level details, and may be omitted.

In summary, Chapters 3, 4 and 7 contain the central results of this thesis, and are the most important to read.

In each of the following chapters, the introductory section contains a more

detailed reading guide for that specific chapter. Also, where necessary, reading guides are given for the major sections of the chapters.

Chapter 2

General ISA Design Issues

2.1 Introduction

The Instruction Systolic Array (ISA), introduced in Section 1.2, is a flexible model of parallel computation suitable for efficient VLSI implementation [35, 21]. However, many papers on the ISA consider single problems or at most small classes of problems, and the issues of designing a general-purpose instruction set and suitable program and data interfaces for the ISA are left largely undeveloped[1]. Thus, the potential flexibility of the the ISA has yet to be demonstrated. These issues must be dealt with if the ISA is to attain its full versatility as a model that can be used for general-purpose matrix computations.

For this chapter, except where specified otherwise, the following program and data interface models for an $n \times n$ ISA are used (cf. Figure 2.10):

- The ISA data interface has data buffers (each of area $\Theta(n^2)$) at each of its (NEWS) sides connected to the respective communication registers of the cells of the boundaries of the ISA, eg. the western input communication registers on the ISA's western edge can read from the western data buffer and the western data buffer can store the values of the output communication registers of the ISA's western edge. These buffers are assumed to act as either queues or stacks and to be sufficiently programmable to interact

[1] An ISA instruction set suited for sorting algorithms and a design of an ISA data interface has been proposed by Lang [35, Ch.3].

with the ISA in the desired way.

- The ISA program interface has buffers storing instructions to its north (the *instruction buffer*), and selectors (the *selector buffer*) to its east.

Section 2.2 describes a boolean ISA, BISA, which is used as a reference point for discussion for the rest of this thesis. The concept of the BISA, developed by the author, is of interest in its own right, being one of the first ISA cell designs intended to implement a wide variety of matrix algorithms and having flexible modes of communication.

In Section 2.2.2, examples of BISA programs are given, which also provide motivation for concepts such as microprogramming and program compression, which are dealt with in detail in the subsequent chapters. From considering the BISA, a discussion of the instruction granularity of the ISA is given in Section 2.3. The overall memory size of an ISA is discussed in Section 2.4. The crucial problem of matching the problem size to the mesh size is addressed in Section 2.5. From there, the design of ISA data interfaces is developed in Section 2.6, which includes a proposal of a partitionable mesh data interface design, with control structures enhancing overall mesh performance. The development of an ISA program interface is deferred to Chapter 3. Conclusions are given in Section 2.7.

The original results of this chapter are mainly confined to Sections 2.2 and 2.6.3. Other parts of this chapter build largely on existing works.

For reading subsequent chapters of this thesis, a familiarity of the basic programming concepts and notatons of Section 2.2 will be useful. This is because a BISA-like instruction set is used to express the ISA and Processor Array programs appearing in this thesis. Section 2.3 provides motivation to Chapter 4; whereas Sections 2.4 and 2.6 are relevant to Chapter 3.

2.2 Case study: the boolean ISA, BISA

Boolean ISA implementations for graph algorithms have already appeared in the literature [32, 48, 49, 50]. These implementations are based on an ISA cell

designed for that particular application. This section presents the design of a boolean ISA (BISA) sufficiently powerful to efficiently implement these and other algorithms, with capabilities for bitwise arithmetic.

A boolean ISA is chosen for this case study for the sake of simplicity of design. It is sufficient for illustrating most ISA design and programming principles. Moreover, boolean ISAs have a particularly high ratio between the instruction and data lengths, and hence they can especially benefit from the program compression techniques introduced in Chapter 3.

Section 2.2.1 gives the instruction set and describes the flexible communication modes of the BISA cells. A variety of new algorithms is then given for the BISA in Section 2.2.2, justifying the design decisions taken for the BISA. The limitations for the BISA are given in Section 2.2.3. A prototype version of BISA, designed for easy VLSI implementation, and yet still capable of implementing all but one of algorithms of Section 2.2.2, is given in Appendix 2.A.

2.2.1 Instruction set and communication modes

The BISA is reasonably modest in terms of its instruction set, communication capabilities and internal storage. However, these are sufficient to enable the BISA to implement most boolean ISA algorithms.

The BISA has an instruction cycle based on a two phase clock, whose phases are denoted by ϕ_1, ϕ_2 respectively. On ϕ_1, a BISA cell reads the operands (possibly including one of the communication registers of its four neighbouring cells) for the current instruction. On ϕ_2, a BISA cell writes the result of the instruction to its destination (possibly its own communication register).

The instructions for an array of BISA cells are sent south through the array via 16-bit shift registers. While a 16-bit instruction code seems rather large, it has a RISC-like format which is very easy to decode. The efficiency of this scheme for the ISA is justified in Section 2.2.3. The selectors bits are passed east across the array via 1-bit shift registers.

A BISA cell has the following 16 memory locations, which are allocated addresses from 0 to 15 in order of their presentation:

- Four registers, (C'_S, C'_E, C'_N, C'_W), which may be used as output communication registers or internal registers, according to which communication mode the BISA is currently in.

- Four 'accumulator' registers (A, B, A', B').

- Four ring-shift registers (R_1, R_2, R_3, R_4) which are bit arrays of length $l_R = O(\log n)$. These registers may be rotated in either direction, and can be used for bitwise integer arithmetic, manipulating non-numeric data and as 'secondary' storage. Only one, say the 0th, element of these registers is directly accessible.

- Four read-only input communication registers (C_W, C_N, C_E, C_S). These store the previous instruction cycle's value of the appropriate output communication register of the respective (west, north, east, south) neighbouring cells.

For the cells on the ISA boundaries, 'neighbouring cells' includes the appropriate data buffer cells.

The BISA has two output communication register modes:

1. One output register mode.
   C'_S (here denoted C) is read by all four neighbours of the cell, while C'_E, C'_N, C'_W (now denoted D, E, F respectively) are used as internal registers. This mode is used in the ISA literature, and is efficient for most ISA algorithms.

2. Four output register mode.
   Here C'_S, C'_E, C'_N and C'_W are read by the cell's south, north, north and south neighbours respectively. This mode is more powerful, although not always as convenient, and is useful when different sets of data are passed in different directions simultaneously.

This last mode is useful to overcome the 'single communication register' bottleneck which can limit mesh performance.

The syntactic form of a generic BISA instruction is (where immediately below, '[...]' represents an optional field):

$$[C(\text{src}),]\,dst \;\leftarrow\; dst\;\theta\;src \;\; [\text{if } cond] \;\; [;(dst)^{\text{r\_op}}] \;\; [;(src)^{\text{r\_op}}]$$

where:

$$\begin{aligned}
\theta &\in \{\text{id}, \overline{\text{id}}, \wedge, \overline{\wedge}, \vee, \overline{\vee}, \oplus, \overline{\oplus}\} \\
dst &\in \{\text{C}'_\text{S}, \text{C}'_\text{E}, \text{C}'_\text{N}, \text{C}'_\text{W}, \text{A}, \text{B}, \text{A}', \text{B}', \text{R}_1[0], \text{R}_2[0], \text{R}_3[0], \text{R}_4[0]\} \\
src &\in \{\text{C}'_\text{S}, \text{C}'_\text{E}, \text{C}'_\text{N}, \text{C}'_\text{W}, \text{A}, \text{B}, \text{A}', \text{B}', \text{R}_1[0], \text{R}_2[0], \text{R}_3[0], \text{R}_4[0], \text{C}_\text{N}, \text{C}_\text{W}, \text{C}_\text{S}, \text{C}_\text{E}\} \\
cond &\in \{\text{A} = 1, \text{B} = 1\} \\
\text{r_op} &\in \{\text{`}+\text{'}, \text{`}-\text{'}\} \\
C(s) &= \begin{cases} \text{addr}^{-1}(\text{addr}(s) \bmod 16) & \text{if in one-output mode} \\ \text{addr}^{-1}(\text{addr}(s) \bmod 4) & \text{if in four-output mode} \end{cases}
\end{aligned}$$

Here 'addr' is the mapping between register names and their respective addresses; note that the output communication registers have addresses from 0 to 3. Note also that $C(s)$ always selects a register that is being currently used as an output communication register.

The semantics of the above generic instruction is reasonably self-evident, except for the optional ring-shifting fields of the form '$(R)^{\text{r\_op}}$' , where $R \in \{R_0, R_1, R_2, R_3\}$, which have the following meanings:

$$\begin{aligned}
(R)^{+} &: (R[i] \leftarrow R[(i+1) \bmod l_\text{R}] \text{ , for } 0 \le i < l_\text{R}) \\
(R)^{-} &: (R[i] \leftarrow R[(i-1) \bmod l_\text{R}] \text{ , for } 0 \le i < l_\text{R})
\end{aligned}$$

These registers can be implemented efficiently in VLSI in a similar fashion to two-phase VLSI stacks (see [40, pp71-75]).

The BISA instruction set has the following features:

- an easily decoded 16-bit RISC instruction format.
- support for the boolean operations of bit copy, 'and', 'or', 'exclusive or', and their negations.
- an optional store to the communication register (corresponding to the source register), as well as the destination register. This simple and easily imple-

mented feature helps to alleviate the 'communication register bottleneck' of such types of meshes.

- a data-dependent masking feature (used also for ISA instruction set of [35, Ch.3], in which the result of the instruction is stored only if the specified accumulator (A or B) is set. This can considerably improve the flexibility of the ISA.

- an optional rotation of any ring-shift operand of the instruction. The semantics suggests the rotation occurs after the execution of the main part of the instruction.

To illustrate the utility of this instruction set, a list of commonly used instructions, using convenient shorthands that will be used throughout the rest of this thesis, is given below. The list is put in two sections, corresponding to each of the BISA's communication modes:

$A \leftarrow A$	$[A \leftarrow A \,\mathrm{id}\, A]$		(no-operation)
$C \leftarrow \overline{C}$	$[C \leftarrow C \,\overline{\mathrm{id}}\, C]$		(negate C)
$C \leftarrow 0$	$[C \leftarrow C \oplus C]$		(clear C)
$R \leftarrow (R = C)$	$[R \leftarrow R \overline{\oplus} C]$		(does R match with C?)
$C, A \leftarrow A\, C_S$	$[C, A \leftarrow A \wedge C_S]$		
$C \leftarrow C_W$, if $A = 1$			(read from west if A is set)
$B \leftarrow R_1;\ R_1^+$	$[B \leftarrow R_1[0];\ R_1^+]$		
			(copy $R_1[0]$ to B then +ve rotate R_1)
$R_2 \leftarrow C_N;\ R_2^-$	$[R_2[0] \leftarrow C_N;\ R_2^-]$		
			(copy C_N to $R_1[0]$ then -ve rotate R_2)

$C'_E, B \leftarrow C_W$		(note: here $C(C_W) = C'_E$)
$C'_S, A \leftarrow A + C_S$	$[C'_S, A \leftarrow A \vee C_S]$	(note: here $C(A) = C'_S$)

To support bit-wise integer operations, a 'half-add' instruction (used to perform counting) is included:

$$\mathrm{HA}(S, C): \quad (S, C) \leftarrow (S \oplus C, SC)$$

in which typically the sum bit S would be the 0th element of a ring-shift register. Although the half-add instruction can be implemented by a sequence of three already existing BISA instructions, a direct hardware implementation is worthwhile if bitwise integer arithmetic is regularly used. A 'full-add' instruction can

then be implemented as follows:

$$\mathrm{FA}(S, A, C): \quad \mathrm{HA}(S, A); \mathrm{HA}(S, C); C \leftarrow C \vee A$$

The use of bit-wise arithmetic on the BISA is illustrated in Section 2.2.2.6.

Similarly, bit-wise comparisons can be performed by a sequence of existing BISA instructions (eg. performing a bitwise minimum operation on two integers requires approximately six BISA instructions per bit). If greater efficiency is required, specialist bitwise comparison instructions may be added to the instruction set.

2.2.2 Some boolean algorithms on the BISA

This section presents some BISA implementations of boolean systolic algorithms taken from a variety of problem areas. These utilize most of BISA's instruction set. A program for solving the Red Squares problem is a typical boolean ISA application. The matrix multiplication and transitive closure programs demonstrate how complex communication patterns can be emulated on the one output communication register mode of the BISA, although in the case of matrix multiplication, the four output communication register mode is superior. The pattern matching and associative memory search programs are examples of how non-numeric problems can be implemented bitwise on a boolean ISA. The median finding program demonstrates how to use BISA's ring shift registers for bitwise counting.

For the purpose of reading the rest of this thesis, the Red Squares and matrix multiply programs (and to a lesser extent, the transitive closure and pattern match programs) are important. Examples of simpler ISA programs can be found in Chapter 3.

The presentation of the ISA programs in this section is based on a simple, informal notation to specify the sequencing of the main sub-programs, together with the standard matrix representations of the ISA sub-programs. This arrangement is necessary for the handling of the more complex programs. While the matrix representation of ISA programs is cumbersome and only specifies the program for

a fixed ISA size, its use is adequate for this section. More concise and general notations for expressing ISA programs are presented in the subsequent chapters of this thesis.

2.2.2.1 The Red Squares program

The Red Squares problem is formulated as follows:

> Given an $n \times n$ boolean matrix A, compute S, the size of the largest square of 1's (largest 'red square') in A.

For implementing this algorithm on a boolean ISA, a unary representation of S is convenient. An equation giving the kth bit, $0 \leq k < n$, of such a unary representation is:

$$S_k = \bigvee_{i=0}^{n-1} \bigvee_{j=0}^{n-1} A_{i,j}^k \tag{2.1}$$

where:

$$\begin{aligned} A^0 &= A \\ A'^k_{i,j} &= \begin{cases} 0 & \text{if } j = n \\ (A^k_{i,j} A^k_{i,j+1}) & \text{otherwise} \end{cases} \\ A^{k+1}_{i,j} &= \begin{cases} 0 & \text{if } i = n \\ (A'^k_{i,j} A'^k_{i+1,j}) & \text{otherwise} \end{cases} \end{aligned} \tag{2.2}$$

where $0 \leq k < n$ and $1 \leq i, j \leq n$. Inspecting equation (2.2), it is evident that $A^k_{i,j}$ is true if and only if there is a 'red square' of size $k+1$ in A having its upper left corner in position (i, j). Hence, S_k is true if and only if there is a red square of size $k+1$ in A. S_k can be efficiently computed on an ISA using the following recursive formula:

$$\begin{aligned} S1'^k_{i,j} &= \begin{cases} A^k_{i,j} & \text{if } j = 1 \\ A^k_{i,j} \vee S1'^k_{i,j-1} & \text{otherwise} \end{cases} \\ S1^k_i &= \begin{cases} S1'^k_{i,n} & \text{if } i = 1 \\ S1'^k_{i,n} \vee S1^k_{i-1} & \text{otherwise} \end{cases} \\ S_k &= S1^k_n \end{aligned} \tag{2.3}$$

where $0 \leq k < n$ and $1 \leq i, j \leq n$. Equations (2.2) and (2.3) suggest the following algorithm:

RedSquares:

ComputeS_k;
ComputeA_k;
ComputeS_k } **repeated $n-1$ times**

where each of the $n-1$ iterations here corresponds to one of the $n-1$ values of k of equation (2.2). A pictorial representation of these sub-algorithms is presented in Figure 2.1.

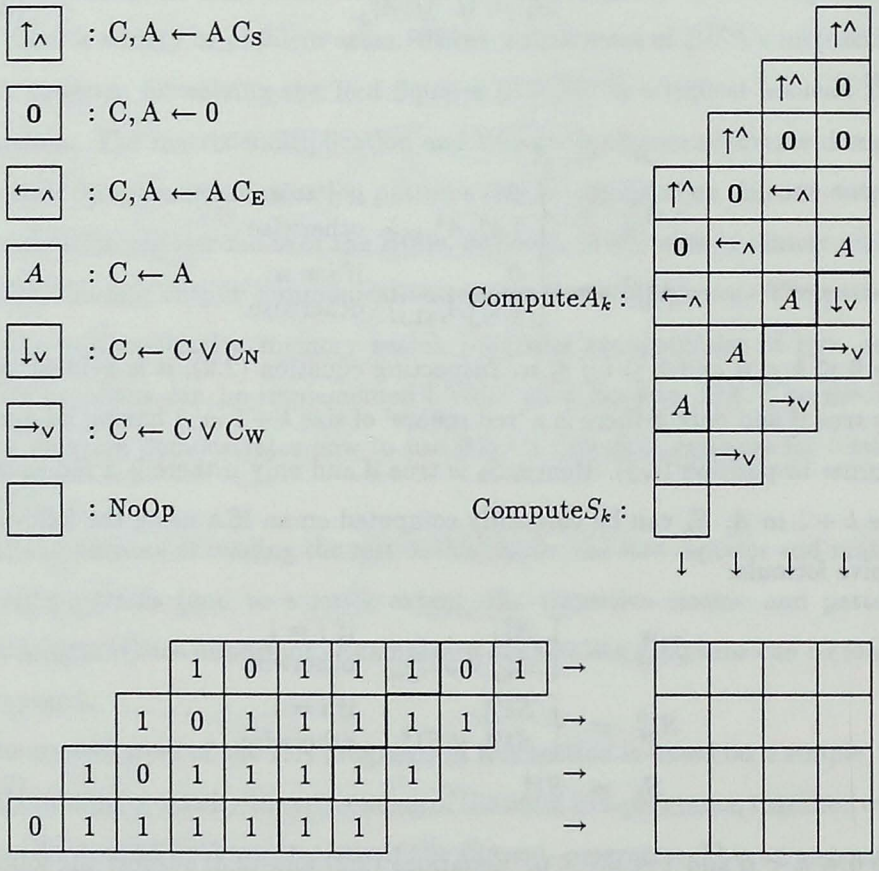


Figure 2.1: Sub-programs for program RedSquares on a 4×4 BISA

Assume that initially the A^0 matrix is stored in both the respective A and C registers of an $n \times n$ BISA using a one output communication register mode[2].

The ComputeS_k sub-algorithm is derived from equations (2.3): its kth iteration in algorithm RedSquares computes S^k. For its first diagonal, the C register of cell (i,j) already contains the value of $A^k_{i,j}$, so that cell (i,j), for $2 \leq j \leq n$, executes the '$\rightarrow_\vee$' instruction to compute $S1'^k_{i,j}$, whereas cell $(i,1)$ need only perform a no-operation. Similarly, cell (i,n), for $2 \leq i \leq n$, executes the '$\downarrow_\vee$' instruction to compute $S1^k_i$, whereas cell $(1,n)$ need only perform a no-operation. The nth column of the southern data buffer is assumed to read the value of S_k.

The ComputeA_k sub-algorithm is derived from equations (2.2): its kth iteration computes A^{k+1}. Firstly, the C register of cell (i,j) is loaded with $A^k_{i,j}$. Then, cell (i,j), for $1 \leq j < n$, executes the '$\leftarrow_\wedge$' instruction to compute $A'^k_{i,j}$ by reading from the east. This in turn requires cell $(i,j+1)$ to have completed loading its C register with $A^k_{i,j+1}$; hence this instruction must be preceded by a no-operation. Cell (i,n) clears its C register, using a '**0**' instruction. Similarly, cell (i,j), for $1 \leq i < n$, executes the '$\uparrow_\wedge$' instruction to compute $A^{k+1}_{i,j}$ by reading from the south. This in turn requires cell $(i+1,j)$ to have completed computing $A'^k_{i,j+1}$: here again, a preceding no-operation is required. This time, the no-operation is implemented by a '**0**' instruction meeting with a 0 selector (for rows 1 to $n-1$). However, on row n, this '**0**' instruction meets with a 1 selector, and clears the C registers there.

2.2.2.2 The matrix multiplication program

This program illustrates how an ISA algorithm, with a complicated (three-way) communication pattern can be implemented on a BISA using two of its output communication register modes. Computing the product of $n \times n$ boolean matrices:

$$C = AB$$

can be implemented easily on an $n \times n$ array of BISA cells. In [31], such an algorithm is presented where A (B) is passed east (south) through the ISA from

[2]Chapter 3 presents the simple ISA algorithm which loads a matrix into the ISA (see also [51]).

the west (north) data buffer and C_{ij} is accumulated in the respective C registers of the ISA. This algorithm is efficient for the BISA, since the matrix C can be computed and unloaded in $5n$ instruction cycles[3].

However, in [57, pp189-192], an algorithm computing C on a $n \times (2n-1)$ systolic array is presented. In this algorithm, A (B^t, the transpose of B) is passed east (west) through the ISA from the west (east) data buffer, and C is accumulated southwards through the ISA (see Figure 2.2(a)). This algorithm has the disadvantage of requiring extra processors and leaving the result matrix, C, in a scrambled form. However, it presents a challenge for BISA implementation. A straightforward ISA implementation is presented in Figure 2.2(b). This requires a powerful 'M' instruction to perform the computation:

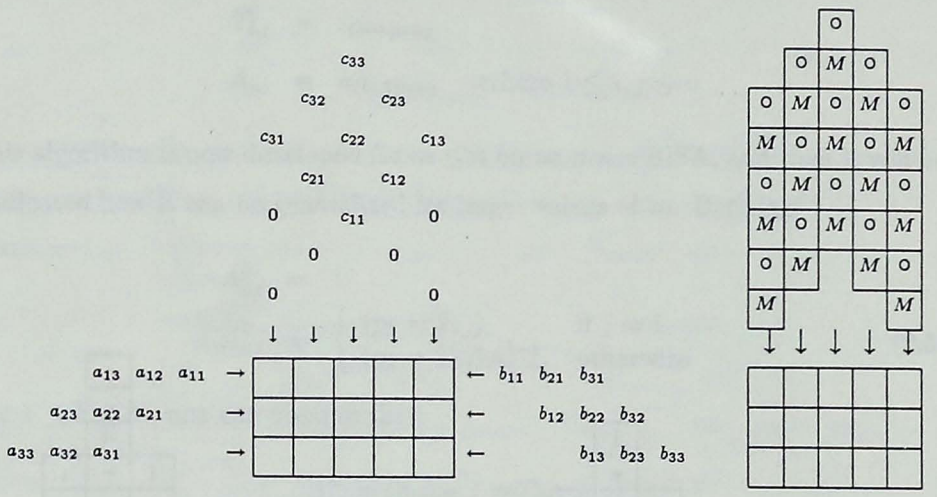
$$M: \quad C'_W \leftarrow C_E; C'_E \leftarrow C_W; C'_S \leftarrow C_N + C_E C_W$$

using the four output communication register mode. Note that the instruction matrix has an inverted 'vee' shape, and the ISA selectors are not used. The 'diagonals' and 'anti-diagonals' of no-operations (blank instructions) are inserted so that the 'diagonals' of 'M' instructions (passing A east) and the 'anti-diagonals' of 'M' instructions (passing B^t west) can coincide. This is the ISA equivalent to inserting corresponding diagonals and anti-diagonals of zeros in the initial value of the C matrix for the systolic array implementation.

The 'M' instruction is much too large to be implemented on a BISA directly[4], but this surprisingly has a compensating factor. A consecutive 'M' and no-operation instruction sequence can be replaced by an east-west communication instruction sub-sequence and a compute/southward communication instruction sub-sequence, as is shown in Figure 2.3. For the one output register mode, the former sub-sequence requires two instructions more than that for the four output register mode. Note that in either case, the 'diagonal' and 'anti-diagonals' of the east-west communication parts coincide naturally. Thus the no-operation

[3]A variant, in which A ([the initial value of] C) is passed east (south) through the ISA from the west data (north) buffer and B is stored in the respective B registers of the BISA. This variant, including the loading of B, has a period of only $4n$.

[4]Chapter 4 discusses how such instructions can be effectively microprogrammed in ISAs such as the BISA.



(a) data flow (b) ISA program

Figure 2.2: Program MatMult for a 3×5 ISA

instructions are no longer required. However, the skew of these 'diagonals' now exceeds one, which is rather irregular for the ISA[5]. The four output register mode program has a period of $4n$, optimal for the BISA provided matrix loading and unloading are taken into account. The one output register mode program has a period of $6n$.

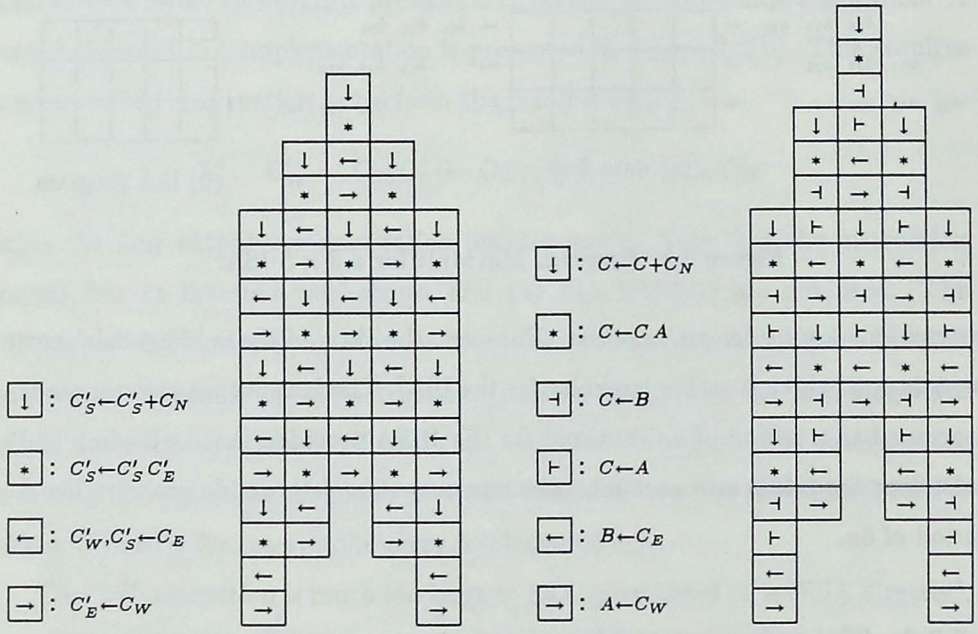
2.2.2.3 The pattern matching program

The pattern matching program efficiently solves on a boolean ISA the following non-numeric (textual problem):

> Given some boolean text $t_{1..n^2}$, and a pattern $p_{1..m}$, where $m = \Omega(n)$ and $m < n^2/2$, find the position of all occurrences of p in t, ie. find $a_{1..n^2}$ such that:

$$a_i = \begin{cases} 0 & \text{if } 1 \leq i < m \\ (t_{(i-m+1)..i} = p) & \text{if } m \leq i \leq n^2 \end{cases} \qquad (2.4)$$

[5]However, this is quite natural for the more general microprogrammed ISA introduced in Chapter 4.



(a) four output mode (b) one output mode, 2 iterations only

Figure 2.3: Program MatMult for a 3×5 BISA

Given the row-major matrix representations of t and a:

$$\begin{aligned} T_{i,j} &= t_{(i-1)n+j} \\ A_{i,j} &= a_{(i-1)n+j} \qquad \text{where } 1 \leq i,j \leq n \end{aligned}$$

this algorithm is now developed for $m \leq n$ on an $n \times n$ BISA, and then it will be indicated how it can be generalized for larger values of m. Defining:

$$\begin{aligned} A^0_{i,j} &= 1 \\ A^k_{i,j} &= \begin{cases} (p_k = T_{i,j}) & \text{if } j = 1 \\ (p_k = T_{i,j}) A^{k-1}_{i,j-1} & \text{otherwise} \end{cases} \end{aligned} \tag{2.5}$$

for $1 \leq k \leq m$, one can observe that:

$$A^k_{i,j} = (p_{(k-l)..k} = T_{i,(j-l)..j})$$

where $l = \min(j,k) - 1$. Hence:

$$A_{i,j} = A^m_{i,j} A'_{i-1,j} \tag{2.6}$$

where:

$$A'_{i-1,j} = A^{m-j}_{i-1,n}$$

The algorithm assumes that initially in cell (i,j) the C register is set and the B register stores $T_{i,j}$. It also assumes that p is stored in row 1 of the west data buffer (to emerge in order $p_1, p_2, p_3, \ldots, p_m$). The algorithm consists of the following sequences of sub-algorithms which are given in Figure 2.4:

Match^m:

Match$'$ } **repeated** m **times**;

ReadE$_m A'$;

Combine$A^m A'$;

The Match$'$ sub-algorithm is in turn composed of the three smaller sub-programs:

Match$'$:

ShiftEA^{k-1};

ScatterSEp_k;

ComputeA^k

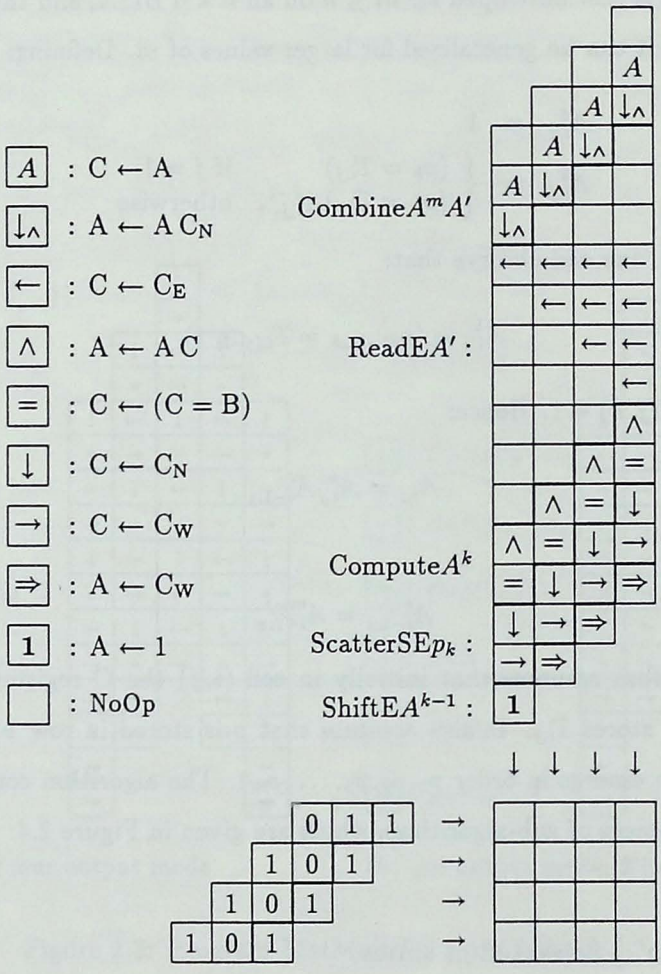


Figure 2.4: Sub-programs of program Match^4 on a 4×4 BISA

(selectors are 1's unless otherwise stated)

Consider the kth iteration of Match′. ShiftEA^{k-1} shifts $A_{i,j}^{k-1}$ east one unit from cell (i,j) to $(i,j+1)$, for $1 \leq j < n$. On this iteration, the east data buffer is assumed to store the contents of the C register of cell (i,n), which contains $A_{i,n}^{k-1}$. ScatterSEp_k broadcasts p_k east and south through the array. ComputeA^k performs the computation of equation (2.5), leaving $A_{i,j}^k$ in the C register of cell (i,j). After the execution of Match′, the C registers concatenated with the east data buffer contents (in order of arrival) of row i contain the values:

$$A_{i,1}^m \quad A_{i,2}^m \quad \cdots \quad A_{i,n}^m \quad A_{i,n}^{m-1} \quad A_{i,n}^{m-2} \quad \cdots \quad A_{i,n}^0$$

The Read$\mathrm{E}_m A'$ sub-algorithm reads the $n \times m$ matrix A' from the east data buffer (which is assumed to be behaving as a queue), so that the computation of equation (2.6) can then be performed by sub-program Combine$A^m A'$. Here, the north data buffer is assumed to contain zeroes to be read by Combine$A^m A'$ on row 1 of the BISA.

For a general value of m, ie. $m = qn + r, 1 \leq r \leq n$, assume that p initially resides in the east data buffer as follows (rightmost values to emerge first):

$$\begin{array}{lcccc}
\{\mathrm{row}_1 :\} & p_n & \cdots & \cdots & p_1 \\
\{\mathrm{row}_2 :\} & p_{2n} & \cdots & \cdots & p_{n+1} \\
 & \vdots & & & \vdots \\
\{\mathrm{row}_{q+1} :\} & & p_{qn+r} & \cdots & p_{qn+1}
\end{array}$$

The pattern match algorithm now consists of q sequential instances of program Match^n followed by Match^r (except that sub-program ScatterSEp_k is selected for the k'th row, where k' is number of the instance). The result matrix lies in the C registers of the BISA at the completion of Match^r.

The period of this algorithm, for $m = \Omega(n)$, is $O(n)$, a $\Theta(n^2)$ speedup of an equivalent serial algorithm. Note that this problem is compute-bound for $m = O(1)$, and hence is not suitable for parallel implementation for such small m. This algorithm can be extended for pattern matching of strings comprising of $b = O(1)$ bits, eg. for bytes $b = 8$, by holding all b bits of each element of T locally in a BISA cell (ie. in ring shift register R_1, provided $l_{\mathrm{R}} \geq b$) and by modifying sub-program Match′ to perform a b-bit broadcast and comparisons.

2.2.2.4 Transitive closure program

The systolic transitive closure algorithm of Kung-Gubias-Thompson [57, pp189-197] for an $n \times n$ boolean matrix A can be implemented on an $n \times n$ BISA. This algorithm requires a two-way communication pattern, which, because of the computations involved, presents a challenge because of BISA's simple instruction set. The initialization step of this algorithm is:

$$\begin{aligned} A^0_{ij} &= 0 \\ {A'}^0_{ik} &= A_{ik} + \delta_{ik} \\ {A''}^0_{kj} &= A_{kj} + \delta_{kj} \end{aligned}$$

for $1 \leq i.j,k \leq n$, where δ_{ij} is 1 if $i = j$, and 0 otherwise. One pass of the algorithm is expressed as:

$$\begin{aligned} A^k_{ij} &= A^{k-1}_{ij} + {A'}^{j-1}_{ik} {A''}^{i-1}_{kj} \\ {A'}^{j}_{ik} &= \begin{cases} {A'}^{j-1}_{ik} & \text{if } i \neq k \\ A^k_{ij} & \text{if } i = k \end{cases} \\ {A''}^{i}_{kj} &= \begin{cases} {A''}^{i-1}_{kj} & \text{if } j \neq k \\ A^k_{ij} & \text{if } j = k \end{cases} \end{aligned}$$

for $1 \leq i,j,k \leq n$. After three consecutive passes of this algorithm, the transitive closure of A is in the matrix corresponding to A^n.

To implement this algorithm on the BISA, the matrix A^k can be stored internally in the BISA, whereas the other two matrices can be passed systolically east and south through the BISA, and be updated as they are passed. Thus, initially, ${A'}^0$ (${A''}^0$) is stored in the west (north) data buffer, and A^0 can be stored in the BISA, as is shown in Figure 2.5. The algorithm then consists of n repetitions; on the kth repetition, cell (i,j) reads ${A'}^{j-1}_{ik}$ (${A''}^{i-1}_{kj}$) from cell $(i,j-1)$ (cell $(i-1,j)$) and A^{k-1}_{ij} from its own A register. It then computes A^k_{ij}, storing it in its own A register, and makes available ${A'}^{j}_{ik}$ (${A''}^{i}_{kj}$) in its communication register [at separate times] to be read upon the next repetition by cell $(i,j+1)$ (cell $(i+1,j)$). The matrix ${A'}^n$ (${A''}^n$) can then be returned from the east (south) data buffer to the west (north) data buffers for the next pass.

The BISA program to implement a single pass is given in Figure 2.6, using the one output communication register mode. For the sake of brevity, the 'C', '$\Rightarrow$'

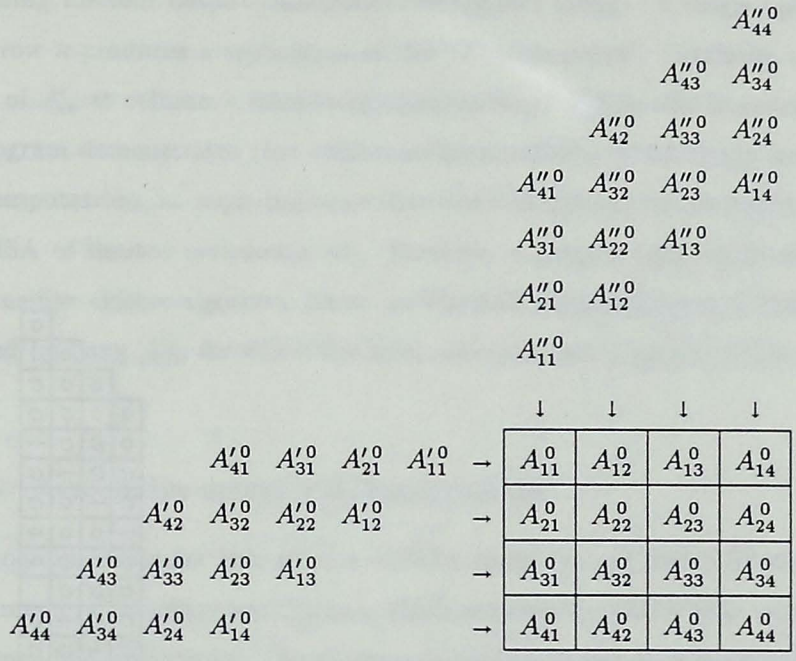


Figure 2.5: Initial data configuration for transitive closure program on a 4×4 BISA

and 'o' macros each represent a sequence of three BISA instructions. Consider the kth iteration of this program for cell (i, j). For columns $j \leq k$, $A'^{j-1}_{ik} = A'^0_{ik}$ is passed east by a diagonal of k '$\rightarrow$' instructions. At column $j = k$, A'^k_{ik} receives a new value. Columns $j > k$ must wait for this updated value before performing their computations, with $A'^{j-1}_{ik} = A'^k_{ik}$ being passed to the remaining columns by a diagonal of $(n - k)$ '$\Rightarrow$' 'instructions'. For rows $i \leq k$, $A''^{i-1}_{kj} = A''^0_{kj}$ is passed south by the '$C \leftarrow C_N$' component of the first 'C' 'instruction', which meets with a diagonal of k '1' selectors. At row $i = k$, A''^k_{kj} receives a new value. Rows $i > k$ must wait for this updated value before performing their computations, with $A''^{i-1}_{kj} = A''^k_{kj}$ being passed to the remaining rows by the '$C \leftarrow C_N$ component of the second 'C' 'instruction' which meets a diagonal of $(n - k)$ '1' selectors. The period for a single pass is $13n$ on the BISA.

Here, the instruction and selectors themselves, as opposed to systolic control bits, determine where to update the A' and A'' matrices. Since BISA's instruction set is limited, a repetition cannot be implemented by a single BISA instruction

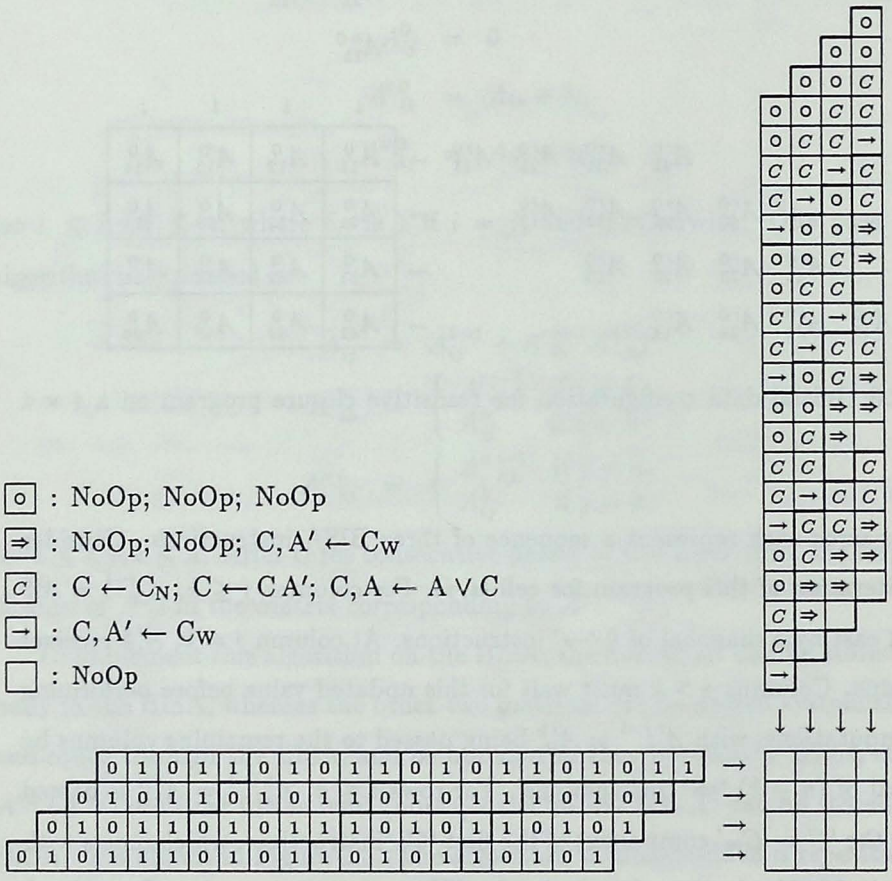


Figure 2.6: Transitive closure program on a 4×4 BISA

(even using the four output communication register mode). A single update of A''_{kj} at row k produces a replication of the 'C' 'instruction'. Similarly, a single update of A'_{ik} at column k introduces no-operations. While this is undesirable, this program demonstrates that communication patterns occasionally involving large computations, ie. requiring more than one instruction, can be implemented on an ISA of limited instruction set. However, a simpler (and more efficient) ISA transitive closure algorithm based on Warshall's algorithm has already been proposed by Lang [32], for which the total algorithm has a period of $11n$ on the BISA.

2.2.2.5 Associative memory lookup program

This program illustrates how an $n \times n$ BISA array can perform a bitwise associative memory matching and lookup, which requires the BISA cells to perform data-dependent operations. The program is parameterized by a word length m, where $m = \Omega(n)$ and m divides n. The program performs the following task:

> Given the $(n^2/m) \times m$ *key* matrix K, an $(n^2/m) \times m$ *lookup* matrix L and an $m \times 1$ pattern vector p, return, if it exists, the value $L_{i'}$ where (i' is the largest integer such that) $K_{i'} = p$.

Since $m = \Omega(n)$, each word in the lookup or key matrices is distributed over m BISA cells. Thus, the program assumes that K is stored bitwise in row major order in the array, ie. $K_{i',1}, \ldots, K_{i',m}$, where $1 \leq i' \leq n^2/m$, is stored in the respective A' registers of cells $(i, j_1), \ldots, (i, j_m)$, where $i = \lfloor \frac{m(i'-1)}{n} \rfloor + 1$ and $j_k = m(i' - 1) \bmod n + k$. A similar case exists for L, except that it is stored in the B' registers of the BISA. For the sake of simplicity, the program also assumes that the pattern vector p is stored in each row of the western data buffer[6]. The program outputs whether the lookup is successful in column $n-1$ of the south data buffer. It also outputs the lookup value (if any) in the last m columns of the south data buffer. The program uses BISA's one output communication register mode and may be expressed by the following sequence of sub-programs:

[6] A simple $O(m)$ BISA program can be found to read in p from a single row of the west data buffer, and then write it to all rows.

AssocMem:

Scatter$p$;

Match;

ScatterMatch;

GatherRowsMatch;

GatherArrayMatch;

GatherRows$L$;

GatherArray$L$

These sub-programs do not use the ISA selectors and their instruction matrices are illustrated for $m = 3$ and $n = 9$ in Figure 2.7. The most interesting features of this program are contained in sub-program GatherRows$L$; only this sub-program will be explained in detail.

Consider the case of key $K_{i'}$, $1 \leq i' \leq n^2/m$. The first five sub-programs place the result of the match ($K_{i'} \neq p$) in the A registers of cells (i, j_1) to (i, j_m) and reset the B register of a cell iff a match has already occurred in that row.

Sub-program GatherRows$L$ gathers bitwise across each row the (rightmost) lookup value corresponding to a matched key. On its kth iteration, $1 \leq k \leq m$, the C register of cell (i, j_k) is loaded with $L_{i',k}$. If $K_{i'} = p$, then the A register of cell (i, j_k) is 0 and the '$\underline{\rightarrow}^{A}$' instruction is not executed, allowing $L_{i',k}$ to be read by the cell to the east. Otherwise, this instruction is executed and the kth bit of any matched lookup values from the west is sent eastwards.

These lookup values are gathered row-wise in the C registers of their respective cells; sub-program GatherArray$L$ gathers them column-wise in a similar way, using the '$\downarrow_B$' instruction. The lookup value $L_{i'}$ of the largest i' such that $K_{i'} = p$ can be then read by the last m columns of the southern data buffer.

The period of the algorithm is $O(m)$, an $O(n^2/m)$ speedup on an equivalent serial algorithm.

2.2.2.6 Median finding program

An efficient ($O(L \log L)$) systolic implementation of finding the median matrix M, using an $L \times L$ window, where $L = 2k' + 1$, of the boolean $n \times n$ image matrix

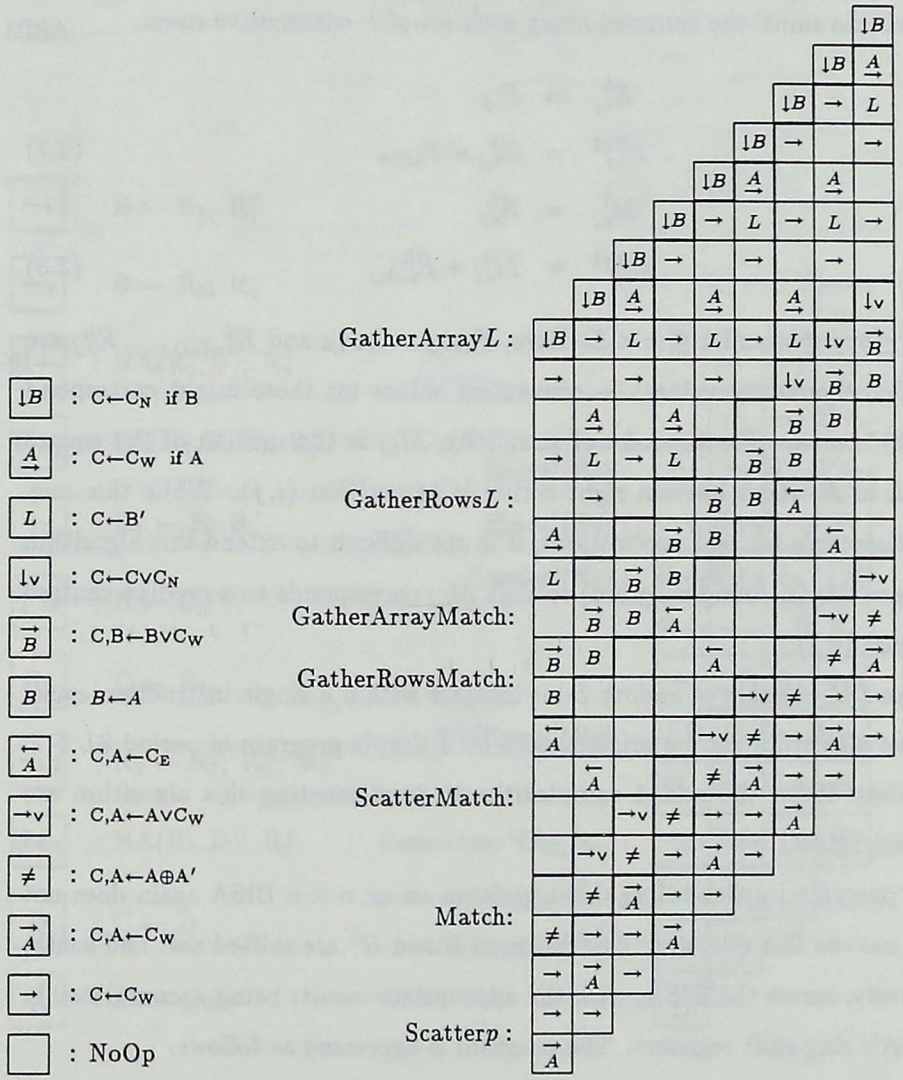


Figure 2.7: Instruction part for (the sub-programs of) program AssocMem for $m = 3$ for a 9×9 BISA

P can be derived from the following set of equations:

$$M_{i,j} = (M^L_{i,j} > L^2/2)$$

The image matrix is first summed along each set of L consecutive columns, and these 'column sums' are summed along each set of L consecutive rows:

$$\begin{aligned} R^1_{i,j} &= P_{i,j} \\ R^{k+1}_{i,j} &= R^k_{i,j} + P_{i,j-k} \qquad (2.7) \\ M^1_{i,j} &= R^L_{i,j} \\ M^{k+1}_{i,j} &= M^k_{i,j} + R^L_{i-k,j} \qquad (2.8) \end{aligned}$$

where $1 \leq i, j \leq n$ and $1 \leq k < L$. Here, $P_{i,2-L}, \ldots, P_{i,0}$ and $R^L_{2-L,j}, \ldots, R^L_{0,j}$ are unspecified 'boundary values' — convenient values for these might correspond to a 'grey' colour. For the sake of simplicity, $M_{i,j}$ is the median of the square of side L in P whose bottom right corner is at position (i, j). While this may be unsatisfactory for real applications, it is not difficult to extend this algorithm (and hence the following program) so that $M_{i,j}$ corresponds to a median centred on position (i, j).

On an ISA capable of adding L-bit integers within a single instruction cycle, the above algorithm can be implemented by a simple program of period $3L + 2$. For boolean ISAs, the added complexities in implementing this algorithm are illustrated below.

The program implementing this algorithm on an $n \times n$ BISA again does not need to use the ISA selectors. The matrices P and R^L are shifted east and south, respectively, across the BISA, with the appropriate results being accumulated in the BISA's ring-shift registers. The program is expressed as follows:

MedianFind:

$\left.\begin{array}{l} \mathrm{AddPixel_W}^{1(k)}(\mathrm{R_1}); \\ \mathrm{ResetCopy_W}^{1(k)}(\mathrm{R_1}, \mathrm{R_2}) \end{array}\right\}$ **repeated for** $k := 1$ **to** $L - 1$;

$\left.\begin{array}{l} \mathrm{AddRow_N}^{L'}(\mathrm{R_1}, \mathrm{R_2}); \\ \mathrm{Carries}^{1(k)}(\mathrm{R_2}); \\ \mathrm{Reset}^{L'+1(k)}(\mathrm{R_2}); \\ \mathrm{Reset}^{L'}(\mathrm{R_1}) \end{array}\right\}$ **repeated for** $k := 1$ **to** $L - 1$

where $l(x) = \lceil \log(x+1) \rceil$ and $L' = l(L-1)$. Here, the '**repeated for** $k := 1$ **to** $L-1$' construct means that $L-1$ repetitions of the associated sub-programs are performed, with the kth repetition being dependent on the value of k. The instruction parts of these sub-program are presented in Figure 2.8 for a 4×4 BISA.

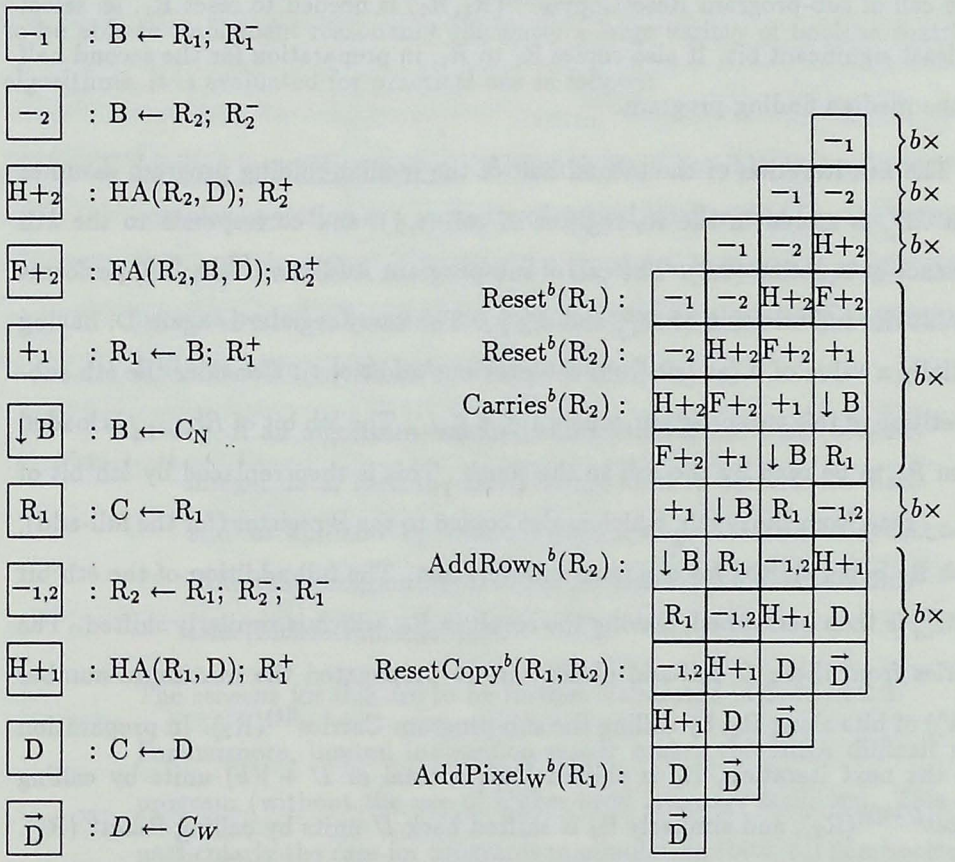


Figure 2.8: Instruction part for (the sub-programs of) program MedFind for a 4×4 BISA

The program uses the one output register mode of the BISA. It assumes that initially the pixel $P_{i,j}$ is stored in both the C register and the current bit of the R_1

register of cell (i,j). Consider the execution of this program on cell (i,j). The call of $\text{AddPixel}_{\text{W}}^{\text{l}(k)}(\text{R}_1)$ implements the kth instance of equation (2.7). Here, $P_{i,j-k}$ is read from the west and stored in the carry register D. It is then transferred to C in preparation for the next iteration. Then, $\text{l}(k)$ half-add instructions are performed to implement the addition of a this pixel to register R_1 (which currently contains an integer not exceeding k). Upon each half-add, R_1 is shifted so that the carry can be propagated as far as it can possibly go (in this case, $\text{l}(k)$ bits). The call of sub-program $\text{ResetCopy}_{\text{W}}^{\text{l}(k)}(\text{R}_1, \text{R}_2)$ is needed to reset R_1, ie. select its least significant bit. It also copies R_1 to R_2, in preparation for the second half of the median finding program.

The kth iteration of the second half of the median finding program assumes that $M_{i,j}^k$ is stored in the R_2 register of cell (i,j), and corresponds to the kth instance of equation (2.8). The call of sub-program $\text{AddRow}_{\text{N}}^{L'}(\text{R}_1, \text{R}_2)$ performs the bitwise full addition of $M_{i,j}^k$ and $R_{i-k,j}^L$. The carry register is again D, having initially a value of 0 (as left from the previous additions). Consider the bth sub-repetition of this sub-program, where $1 \leq b \leq L'$. The bth bit of $R_{i-k-1,j}^L$ is loaded from R_1 to be read by the cell to the south. This is then replaced by bth bit of $R_{i-k,j}^L$ read from the north, which is also copied to the B register (for the full-add), with R_1 being shifted for the next sub-iteration. The full-addition of the bth bit of $M_{i,j}^k$ is then performed, leaving the result in R_2, which is similarly shifted. The carries from these L' full-add cycles can be propagated the maximum number $(\text{l}(k))$ of bits along R_2, by calling the sub-program $\text{Carries}^{\text{l}(k)}(\text{R}_2)$. In preparation for the next iteration, R_2 is shifted back a total of $L' + \text{l}(k)$ units by calling $\text{Reset}^{L'+\text{l}(k)}(\text{R}_2)$, and similarly R_1 is shifted back L' units by calling $\text{Reset}^{L'}(\text{R}_1)$.

The final step in the median finding algorithm, to determine whether $M_{i,j}^L > L^2/2$, is left undeveloped. It can be implemented by broadcasting $L^2/2$ through the BISA, and performing a bitwise comparison.

The period of this program is $O(L \log L)$, an $O(n^2/\log L)$ speedup of the equivalent algorithm executed on a serial machine supporting full integer addition. However, since the BISA is performing these operations bitwise, the BISA implementation of image median finding is inefficient for small n.

2.2.3 Limitations of the BISA

The BISA is a boolean ISA of moderate capacity, being capable of performing bitwise integer and symbolic processing, and hence has application in implementing simple graph and pixel image processing algorithms. For example, it is capable of implementing systolic graph algorithms such as the transitive closure, shortest path, minimum spanning tree, bridge and cut point algorithms proposed for the ISA by Schimmler and Schröder [49]. While the BISA has been demonstrated to be able to implement reasonably efficiently a large variety of boolean matrix algorithms, it is evaluated for practical use as follows:

1. Limited instruction power. Although it is possible to implement a complex operation as a sequence of several smaller ones (eg. the bitwise 'full-add' operation of Section 2.2.1), and sometimes it might even be area-period efficient to do so, some systolic algorithms require the BISA to have a high instruction granularity.

   eg. if an algorithm required some nontransmittent (bitwise) integer data, then the BISA would need to perform an 'half-add' or 'full-add' operation within a single instruction cycle. Note that program MedFind of Section 2.2.2.6 uses transmittent (bitwise) integer data.

   The reasons for this are to be further elaborated in Section 2.3.

   Furthermore, limited instruction power makes the BISA difficult to program (without the use of higher-level language facilities). This is particularly the case for programs manipulating (bitwise) non-boolean data, such as the MedFind program.

2. Fair communication capabilities. The BISA has a novel instruction set that allows various useful modes of output communication registers, and permits data to be stored to a communication register and non-communication register simultaneously. This alleviates the 'communication register bottleneck' of this type of processor array, and these communication capabilities are sufficient for the class of algorithms

intended for BISA.

3. Insufficient on-cell storage. For general purpose boolean processing, it is useful to have a larger storage capacity (ie. of the order of 10^2 bits). Not all of these need be random access, and ring shift registers provide reasonably efficient means of 'secondary' storage. This is because algorithm partitioning (see Section 2.5.2) may require considerably large sets of data from other partitions to be stored, so BISA's on-cell storage may need to be expanded to meet these requirements.

At present, the BISA uses a 16-bit instruction. The extra pin count potentially arising from this can be avoided by the following technique. It is envisaged that many BISA cells can be fitted on a single chip, and that the decoding (from a *compressed* instruction, ie. a 4-bit, 'macro') of the instruction stream for each column of BISA cells need be done only once at the top (north end) of the column. Four bit 'macros' are deemed sufficient since the decoding scheme can be flexible (ie. different for each program), and most boolean ISA programs require less than 2^4 different instructions. This should result in a reduction of the overall area of the BISA, while keeping its pin count low. This idea is further developed for the microprogrammed ISA introduced in Chapter 3.

From a $350\lambda \times 350\lambda$ nMOS layout of the prototype BISA cell (called bISA, see Appendix 2.A), we estimate that an nMOS layout for the BISA would require not more than $1000\lambda \times 1000\lambda$ area. With a 2.5μm nMOS technology, a 4×4 BISA array would then fit onto a 1cm^2 area chip. Since a BISA cell would have a high instruction rate, and would require 5 pins per column (by compressing instructions into 4 bit macros, as mentioned above) pin count should not present a serious limitation to BISA implementation. However, with 1μm nMOS technology, pin count could impose a limitation on the BISA.

2.3 ISA instruction granularity

The main advantage of the ISA over SIMD arrays is that it is superior in handling nontransmittent data. Hence, by incorporating more sophisticated control

structures, effort should be taken to implement nontransmittent data efficiently on the ISA.

In Section 2.2.2, nontransmittent data is used in four of the BISA programs given there. In program RedSquares (Section 2.2.2.1), the variable $S1_i'^k$ is updated as it passes across row i, accumulating A_{ij}^k as is passes cell (i,j). In program MatMult (Section 2.2.2.2), the variable C_{ij} accumulates the product of $A_{ik}B_{kj}$ as it passes row k. In sub-program GatherRows$L$ of program AssocMem (Section 2.2.2.5), the lookup values are passed westwards along each row, and are updated when passing cells that had matched the lookup values. All these updates could be achieved by a single BISA instruction, so their BISA implementation is efficient. Program TransClos (Section 2.2.2.4), updates (in a more complex fashion) and passes westward the matrix A', making the program bulky and difficult to read. If it was necessary to pass integer data through the BISA[7], and update it (eg. add values to it) as it passes, the BISA would have to implement the 'full-add' instruction directly.

The handling of nontransmittent data sometimes requires high instruction granularity, ie. a relatively large amount of computation and inter-cell communication can be completed within a single instruction cycle.

A high instruction granularity enables the *nontransmittent* data to keep pace with the instructions that manipulate it. Either such highly specialized instructions and a high inter-cell I/O bandwidth must be directly implemented (which is costly, particularly if these are only occasionally used), or the basic ISA architecture must be altered to effectively implement them (as proposed in Chapter 4). For the former case, the requirements of the nontransmittent data intended to be used on the ISA determines its instruction granularity, with its associated communication capabilities.

Flexible modes of output communication registers in an ISA are always useful, since they have an efficient implementation and this flexibility alleviates the 'communication register bottleneck'. If a single mode needs to be chosen for an ISA, the most powerful mode required should be chosen.

---

[7]The median finding program of Section 2.2.2.6 could have been reformulated this way.

2.4 Internal memory size

Another important factor in ISA cell design is the amount of internal memory in each ISA cell. In the case of the BISA, each cell has slightly more internal memory than is required by the programs of Section 2.2.2. While systolic algorithms generally require a modest amount of data storage in each cell, an ISA should have considerable internal memory if it needs to meet some of the following requirements:

1. to implement more complex functions such as multiplication, reciprocal or sine-cosine evaluation by the use of lookup tables stored in the internal memory of each cell [26, pp457-458]. Provided high accuracy is not essential, the use of lookup tables is probably the most efficient method of implementing functions not easily supported by the ISA instruction set directly.
2. to support LSGP partitioning (see Section 2.5) of a $\kappa n \times \kappa n$ problem size on an $n \times n$ ISA, it may be necessary for the cell internal memory to have $O(\kappa^2)$ capacity.
3. to reduce the I/O traffic of the ISA in situations where its I/O bandwidth is insufficient to match its computation rate[8]. This is explained in detail below.

A standard technique to overcome the I/O limitations of a computer architecture is to incorporate a large local memory to reduce its overall external I/O traffic [25]. This can be successful for solving problems in which each output depends on many inputs. In the case of matrix algorithms, such problems are therefore unlikely to be I/O-bound[9] and hence are suitable for parallel implementation [25]. It is reasonable to assume that the I/O bandwidth between an $n \times n$ ISA and its data buffers is $\Theta(n)$. The ISA's computational bandwidth of $\Theta(n^2)$ is balanced with its I/O bandwidth of only $\Theta(n)$ matrix computations (such as

[8]This is not needed if the ISA data buffers have a large memory capacity instead.

[9]Examples of I/O bound problems are matrix-vector multiplication and linear system solving.

matrix multiplication and triangularization) by the overall ISA internal memory of $\Theta(n^2)$ area [25]. Hence, for these matrix computations, an ISA cell's internal memory can be independent on the array size.

However, for applications such as implementing d-dimensional grid problems on an ISA, the overall internal memory of the ISA (and its data buffers) should increase by a factor of n^d.

Furthermore, the I/O bandwidth between the ISA system (the ISA combined with its data buffers) and external data sources should also be considered. In real-time signal processing applications, it can be assumed that $\Omega(n)$ sensing devices exist and can send data to the ISA system in parallel [58, p296]. In this case, the bandwidth can be considered $\Theta(n)$, and the comments above apply to the overall ISA system. However, if the ISA system communicates with an external mass data storage, as is the case in most parallel computer systems, it is conservative to assume that the I/O bandwidth between them is $o(n)$. This is because a $\Theta(n)$ I/O bandwidth requires that the mass storage is 'truly parallel', which could incur great expense in the design and implementation of the storage, not to mention overheads in the organization of data for parallel access. Note that the parallel mass storage in most existing parallel architectures,

> eg. the 32-bank Data Vault of the 2^{16} processor element Connection Machine CM-1, which accesses each bit of 32-bit words in parallel,

is not considered 'truly parallel', since the parallelism that they provide in data access is constant. Here, even for matrix computations, the overall memory of an ISA system must be large enough to reuse as much data as possible in order to reduce external I/O traffic.

2.5 Matching problem size to the array size

In the ISA programs of Section 2.2.2, the ISA is assumed to have a size matching that of the matrices used by the program. In typical scientific applications, computations may use data matrices whose dimensions could range from hundreds to tens of thousands. Any ISA used for such applications must however have fixed

dimensions of the order of hundreds, possibly as much as a few thousand (see Section 1.1.2). Hence, for any particular application, the matrix size and the ISA size are likely to mismatch, typically with the former exceeding the the latter. This presents a crucial problem with the practical usage of array processors in general. If the data matrices are too small for the ISA size, the design of the ISA interface can alleviate the loss of ISA utilization (Section 2.5.1). If the data matrices are too large for the ISA size, they can be *partitioned* into smaller units which match the array size (Section 2.5.2). Both of these techniques have important consequences in the design of ISA programs and control structures for ISA cells and data interfaces. These issues are related to the important problem of balancing a parallel computer architecture's computational and input/output bandwidths (as discussed in Section 2.4).

2.5.1 Small matrix sizes on large ISAs

While this situation is less common in practice, it still warrants some consideration since it is conceivable that data matrices whose dimensions are of the order of a hundred could be efficiently run on an ISA whose dimensions are of the order of a thousand. A simple solution, suited to the ISA data interface proposed by Lang (see Section 2.6.1), is to use the north-westernmost sub-array matching the matrix size, but this can result in poor overall ISA utilization. This can be avoided by being able to partition the ISA into sub-arrays of smaller dimensions, which can either operate in a pipelined fashion [26, p508] or independently. To utilize partitioning into sub-arrays, the following features are required:

- that each sub-array be individually programmable.
- that each sub-array has an effectively independent data interface.
- that ISA programs are parameterized for each possible sub-array size.

While at first glance it might seem difficult, an ISA system architecture, consisting of the data interface of Section 2.6.3 together with the program interfaces for the 'optimal' program compression methods of Chapter 3, can efficiently implement these features.

2.5.2 Partitioning: large matrix sizes on smaller ISAs

An $n \times n$ ISA can operate on matrices of size $\kappa n \times \kappa n$, $\kappa \geq 1$, using either the Locally Serially Globally Parallel (LSGP) or the Locally Parallel Globally Serial (LPGS) partitioning methods [26, pp374-376] (these are also referred to as partitioning using *coalescing mappings* and *cut-and-pile mappings*, respectively [43]). In LSGP partitioning, ISA cell (i,j) is allocated the (i,j)th sub-matrix after the matrix has been partitioned into $\kappa \times \kappa$ sub-matrices. Then, all ISA cells (in parallel) perform the required operations on each element of its $\kappa \times \kappa$ sub-matrix in some serial order. In LPGS partitioning, the matrices are partitioned into κ^2 $n \times n$ sub-matrices. Each of these sub-matrices in turn is operated on, in some appropriate order, the results of which are later combined to give an overall result.

It is essential that the LPGS and/or LSGP partitioning methods can be successfully developed for commercial applications of the ISA. These methods are discussed in more detail below. For reading Section 2.6, an appreciation of the LPGS method is important.

2.5.2.1 LSGP partitioning for the Red Squares program

In general, the LSGP method requires extra control structures[10] and a large internal memory in each ISA cell, but is efficient otherwise. The control structure overhead might be thought too great for fine-grained ISAs; however, surprisingly, LSGP partitioning can usually be implemented on an ISA with very little control structure overhead.

Finding a scheduling for the LSGP partitioning that is compatible with the way instructions and selectors flow through the ISA is a difficult problem. The example program of Appendix 2.B is sufficient to demonstrate that, provided the nontransmittent data of an ISA program is updated with associative and commutative operations, and there is no contraflow in the other data used by the program, then such a scheduling can indeed be found with very low control

[10] eg. A partitioning factor of κ requires mechanisms to pass instructions/selectors at the rate of one cell per κ cycles — cf. a (κ, κ) wavefront μISA (see Chapter 4).

structure overhead.

Appendix 2.B further demonstrates that, provided its ring-shift registers are large enough, LSGP partitioning can even be implemented on the BISA. However, this incurs a considerable expense in programming effort.

Alternatively, microprogramming techniques similar to those presented in Chapter 4 can provide reasonably modest control structures to support more general and more easily programmable LSGP partitioning.

2.5.2.2 LPGS partitioning

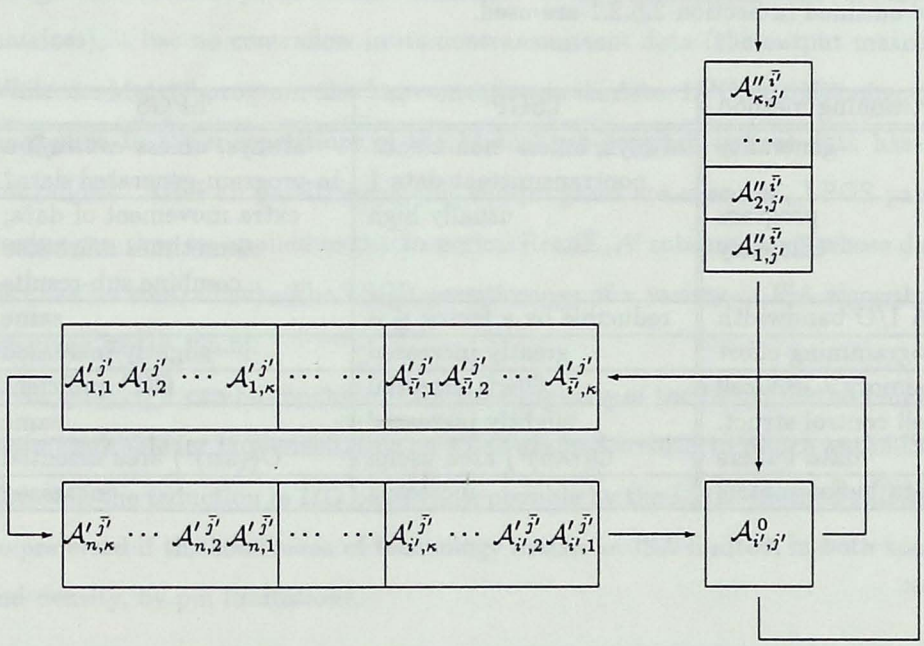
The LPGS method is suitable for ISA programs for which data flows in two orthogonal directions, since then it is easy to schedule the order of execution of the κ^2 $n \times n$ sub-matrices. As an example, consider the LPGS partitioning of the transitive closure program of Section 2.2.2.4 for $\kappa n \times \kappa n$ data on an $n \times n$ ISA. Upon breaking the κn repetitions of this program down into κ stages, for the k'th stage, $0 \leq k' \leq \kappa$, the matrix $A^{k'n}$ is partitioned into:

$$\left(\begin{array}{c|c|c|c} \mathcal{A}^{k'}_{1,1} & \mathcal{A}^{k'}_{1,2} & \cdots & \mathcal{A}^{k'}_{1,\kappa} \\ \hline \mathcal{A}^{k'}_{2,1} & \mathcal{A}^{k'}_{2,2} & \cdots & \mathcal{A}^{k'}_{2,\kappa} \\ \hline \vdots & \vdots & & \vdots \\ \hline \mathcal{A}^{k'}_{\kappa,1} & \mathcal{A}^{k'}_{\kappa,2} & \cdots & \mathcal{A}^{k'}_{\kappa,\kappa} \end{array}\right)$$

ie. the (i,j)th element of $\mathcal{A}^{k'}_{i',j'}$ is given by $A^{k'n}_{i'n+i,\, j'n+j}$, where $0 \leq i', j', k' \leq \kappa$ and $1 \leq i, j \leq n$. Similarly, the (i,j)th element of the nontransmittent sub-matrix $\mathcal{A}''^{i'}_{k',j'}$ ($\mathcal{A}'^{j'}_{i',k'}$) is defined to be $A''^{i'n}_{k'n+i,\, j'n+j}$ ($A'^{j'n}_{i'n+i,\, k'n+j}$).

An efficient LPGS scheduling is to execute all κ stages of the (i',j')th partition consecutively. This requires the matrix $\mathcal{A}^0_{i',j'}$ to be loaded into the ISA, and $\mathcal{A}''^{i'-1}_{k',j'}$ ($\mathcal{A}^{j'-1}_{i',k'}$) to be passed southwards (eastwards) through the ISA, for $1 \leq k' \leq \kappa$. A column-major scheduling for the partitions satisfies this requirement, together with FIFO queues (data buffers) as is shown in Figure 2.9.

Even though an ISA program may have *contraflow*, LPGS partitioning may still be applied. By combining the result matrices in a non-trivial way, Lang gives an efficient LSGP partitioning for the ISA implementation of Warshall's transitive closure algorithm [32]. Also, an equivalent algorithm may exist that has no contraflow, eg. the MatMult program of Section 2.2.2.2 may be replaced



(note: $\bar{i}' = i' - 1$ and $\bar{j}' = j' - 1$)

Figure 2.9: Initial data buffer and ISA configuration for column-major LPGS scheduling of the transitive closure program for partition (i', j')

by more efficient ISA matrix multiplication programs that have no contraflow [31].

2.5.2.3 Comparison of partitioning methods for the ISA

The LSGP and LPGS methods of partitioning are compared for the ISA in summary form in Table 2.1. For the LSGP method, it is assumed that the movement of the ISA instructions and selectors are not modified, and techniques such as those outlined in Section 2.5.2.1 are used.

partitioning method	LSGP	LPGS
generality	always, unless 'non-assoc.' nontransmittent data †	always, unless *contraflow* in program-generated data‡
program efficiency	usually high	extra movement of data; sometimes must also combine sub-results
ISA I/O bandwidth	reducible by a factor $\leq \kappa$	same
programming effort	greatly increased	slightly increased
memory / ISA cell	$\Theta(\kappa^2)$ essential	$\Theta(1)$ sufficient
cell control struct.	slightly increased	same
data buffers	$O((\kappa n)^2)$ area useful	$O((\kappa n)^2)$ area essential
data buffer control	increased	increased

notes:

- 'always', 'same' and 'increased' are qualifications relative to an ISA using no partitioning.

† this means the update of the nontransmittent data must be by an associative and commutative operation; also there must be no contraflow in the other data.

‡ in this case only, *nontransmittent* means data that is updated as it passes through the ISA in *any* direction.

Table 2.1: Comparison of LSGP and LPGS partitioning methods for the ISA

For the ISA programs of Section 2.2.2, the transitive closure program cannot

be partitioned using the LSGP method, since its nontransmittent data (the matrices A' and A'') is updated by a non-associative operator. A similar situation exists for the microprogrammed ISA programs introduced in Chapter 4. Also, when a boolean ISA is used to perform bitwise integer arithmetic, the limitations of cell memory severely restrict the LSGP method.

All ISA programs of Section 2.2.2 can be partitioned using the LPGS method, as can also the microprogrammed ISA programs introduced in Chapter 4. Even though the MatMult program has contraflow in its transmittent data (the input matrices), it has no contraflow in its nontransmittent data (the output matrix). While the Match^n program also has contraflow in its data, LPGS partitioning can be applied to the n repetitions of the Match$'$ sub-program (whose data has no contraflow). After all partitions of this sub-program are executed, LPGS partitioning can then be applied to the $2n$ period Read$\mathrm{E}_n A'$ sub-program (whose data also has no contraflow). The LSGP partitionings of a variety of ISA algorithms are given in [35, Ch.6].

In general, it can be concluded that the simplicity of the LPGS method makes it more suitable for implementation on a fine-grained architecture such as the ISA. However, the reduction in I/O bandwidth possible by the LSGP method still may be preferred if the conditions of technology makes an ISA limited, in both speed and density, by pin limitations.

2.6 Data interfaces for the ISA

A programmable data interface for meshes (eg. the ISA), which includes the data buffers and their 'front ends' (to external storage devices) is extremely important in providing temporary storage and 'balancing' the array architecture [26, pp362-368] (see Section 2.4). In order to enhance overall mesh performance, the control structures for appropriate data interfaces must be carefully designed.

In general, the data interface's memory can be regarded as the ISA system's cache memory, since it can be easily engineered to have an $\Theta(n)$ I/O bandwidth[11]

[11]The ISA registers are the fastest memory, with a $\Theta(n^2)$ bandwidth, whereas that of external storage is the slowest.

with the ISA. Thus, over a long series of computations, the management of large amounts of data in the data buffers is important to minimize external I/O. Ideally, the memory capacity of the data buffers and the ISA should be large enough so that the ISA system can be regarded as 'intelligent memory' (as is done for the Connection Machine [14]), ie. the crucial data structures (matrices) are resident in the ISA system for the whole duration of their creation and use by the ISA. While this approach seems costly in terms of area, it is important in reducing the major limitation of scalable parallel architectures such as the ISA: their external I/O bandwidth. The data buffer design is also important in supporting LPGS (and LSGP) partitioning methods. Thus the data buffer's programmability, capacity and ability to move data are important considerations in the design of an ISA system.

This section examines the data interface for the ISA, and hence deals with part of an important problem: how to integrate an ISA into an overall computing system, in such a way that the array's high computational bandwidth can be utilized. The control (program) interface for the ISA is dealt with in Chapter 3. A programmable data interface itself requires a control interface; however, the data interface designs proposed here can be programmed using the instruction systolic paradigm, and hence their control interfaces are similar to that of the ISA. All principles here discussed for the ISA extend to any meshes, so that their application is quite general. Section 2.6.1 describes existing mesh data interfaces. The desirable properties of ISA interfaces are then outlined in Section 2.6.2. From this, a partitionable and instruction systolically programmable data buffer design is proposed in Section 2.6.3. This design can be easily adapted for any mesh using skewed matrix input/output, and is a useful application of the instruction systolic concept.

2.6.1 Existing mesh data interfaces

While much work has been published on mesh (including the ISA) algorithms and design, the problem of the design for a general, programmable data interface has been left largely undeveloped. However, in most systolic algorithms, the flow

of data over the array's boundaries plays a crucial role in the algorithm.

A simple and area-efficient ISA data interface, based on that proposed by Lang [35, pp28-30], is shown in Figure 2.10. The *data front-end* unit performs

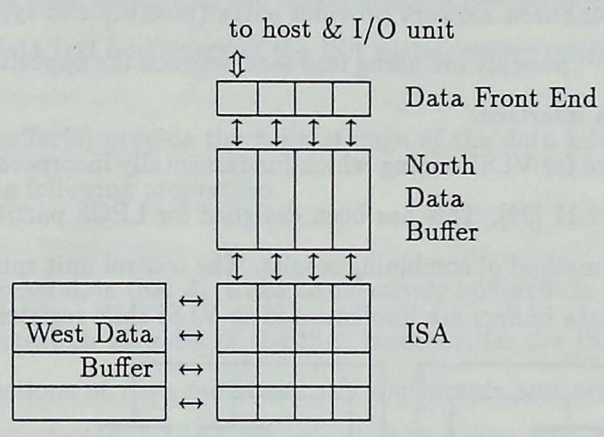


Figure 2.10: An area-efficient ISA data interface, for a 4×4 ISA

functions such as data formatting and data up/down loading. $\Theta(n^2)$ area data buffers for the north and west sides are used, which may be programmed to behave as arrays of either queues or stacks. A write to (read from) a data buffer queue/stack may occur when the adjacent ISA cell writes into (reads from) the appropriate output (input) communication register. Output from (input to) the western side of the array is simulated by shifting east (broadcasting west) the data — such operations can be done in unit period on the ISA. Thus, this data interface requires very simple control structures, and makes efficient use of the ISA's high internal I/O bandwidth.

However, this design suffers from the disadvantage that the ISA system is not partitionable into sub-arrays, and from the fact that feedback occurs via the ISA itself, resulting in a loss of period. To keep this loss to one instruction per feedback operation, output communication register modes more elaborate than those of the BISA, and/or special cell hardware features, are required. To be convinced of this, the reader may try to implement feedback for the $S1$ and S data of program RedSquares on a BISA Section 2.2.2.1). A less serious disadvantage of this design is that the *data front end unit* cannot communicate directly with the ISA, resulting in extra host/ISA communication latency.

A general mesh system architecture is proposed by S.Y. Kung [26, pp362-368]. This has a similar data interface to the above design, with a single *interface unit*, made of a single data buffer and the data front end unit, being used for the mesh. An *interconnection network* provides extra (possibly non-systolic) modes of communication[12], possibly including feedback between the opposite edge of the mesh and the data interface.

A data interface for VLSI sorting which fundamentally incorporates feedback is given in Figure 2.11 [29]. This has been designed for LPGS partitioning using a sort-split-merge method of combining results. The control unit splits partitions of data, and the data buffers are implemented as VLSI shift registers.

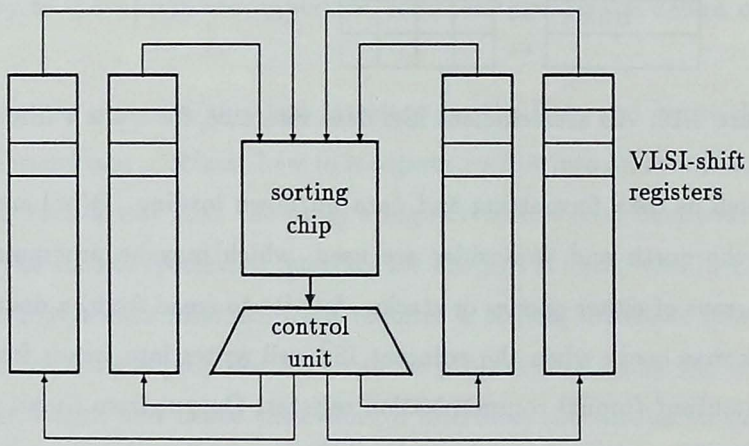


Figure 2.11: A VLSI data interface for a sorting chip (mesh)

2.6.2 Data interface properties

Dividing the data interface into the data front end unit and the data buffer(s), as suggested in Section 2.6.1, the data front end unit should be able to perform the following functions:

- support bus protocols etc. for communication with external storage.
- store a few $n \times n$ matrices, which are currently being downloaded or uploaded into the ISA system. This storage would need to be both accessible

[12]eg. the row/column broadcasting of the ICL-DAP [17, pp246-247]

serially for communication with external storage, and accessible in parallel for communication with the ISA system.

- to provide data formatting and decompression (see Section 3.2.1) to alleviate the data I/O bottleneck of the ISA system where possible.

The data buffer(s) provide the main storage of the data interface, and they should have the following properties:

1. be general, in that data can be effectively buffered (in a FIFO or LIFO manner) on all sides of the ISA. However, for the ISA, the preferred directions of data movement are southwards and eastwards, so that primarily data is input into the north and west sides, and output from the south and east sides.

2. be expandable, so that a large ISA system can be made by simply adding on a suitable number of ISA sub-system modules. Similarly, for the efficient implementation of small data size problems on a large ISA system, the ability to partition the ISA system into smaller independent or pipelined ISA sub-systems is desirable.

3. use fixed length wires only.

4. be able to support LPGS (and possibly also LPGS) partitioning to any partitioning factor of κ required by the ISA's applications. This requires $O(\kappa^2)$ storage locations for each of the n^2 data buffer cells. Also, for LPGS partitioning, the data buffers must be able to simulate feedback queues of lengths such a n, κn and $\kappa^2 n$ (cf. Figure 2.9).

5. to have control structures that can efficiently support these capabilities, and be able to support all I/O operations the ISA might reasonably require.

However, a tradeoff exists between the benefits of these properties and the area that they require.

2.6.3 A partitionable instruction systolic data buffer design

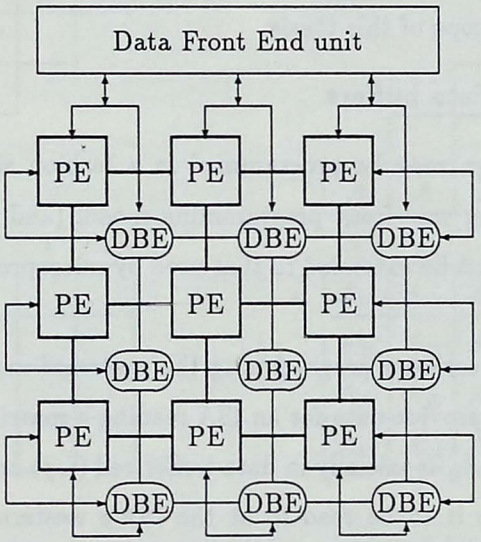
A partitionable instruction systolic data buffer design is presented here which satisfies the data buffer properties of Section 2.6.2. The design supports skewed input of matrices into the ISA, such as is required for most systolic algorithms. The design has simple control structures which aim to maximize the overall ISA performance.

The design can be derived by 'folding' the two data buffers of Figure 2.10 into one, as is shown in Figure 2.12. Each element in this data buffer (DBE) contains a cell having several (at least four, one for each direction) independent data channels. For the sake of simplicity, each data channel is assumed to move one word of data one unit in the appropriate direction each ISA cycle. At any time step, the data buffer cell is assumed to be able to communicate one word of data (either or both ways) between its $\Theta(\kappa^2)$ word RAM and any of its data channel cells. The interaction of each data channel with the data buffer can be programmed separately.

Figure 2.12 gives a planar layout of the data buffer design (note that this 'cylindrical' topology can be laid out more efficiently than is suggested by this figure). By incorporating a direct link between each ISA cell and its adjacent data buffer cell, this layout may also enable matrices to be transferred between the data buffer and the ISA in unit time, as well as support LSGP partitioning.

A more area-efficient 2-planar layout, with the PE on the upper plane and the data buffer on the lower, is also feasible (a 2-planar circuit board configuration is used for the PAX array computer [56, p64]). This layout requires that the area of the ISA cells and DBEs each must match exactly, possibly requiring in turn that the ISA cells expand their internal memories.

It is evident that this design satisfies the data buffer properties of Section 2.6.2, and that the data front end unit can communicate directly with the ISA. Feedback queues of length $\alpha n, \alpha > 0$, can be implemented by reading from the data channel into memory location x and then writing from memory location $x + \alpha$ into the data channel, at each data buffer cell. The next such read/write



('PE' denotes a ISA processing element; 'DBE' denotes a data buffer cell)

Figure 2.12: Instruction systolically programmable data interface for a 3×3 ISA

operation is similar except x is replaced by $x + 1$.

This data buffer design has however some drawbacks. While it is easy to program matrix row and/or column reversal, matrix transposition cannot (easily) be performed by the data buffers. Also, since the design requires at least four independently programmable data channels, the data buffer memory may need to support parallel read and write operations, a significant overhead. The ISA and data buffer system have a cylindrical topology, whose one-planar layout results in some wastage of area.

It would be interesting to make a detailed area-time comparison between this period-efficient design and the area-efficient design presented earlier. This is however beyond the scope of this thesis.

Programming the data buffers

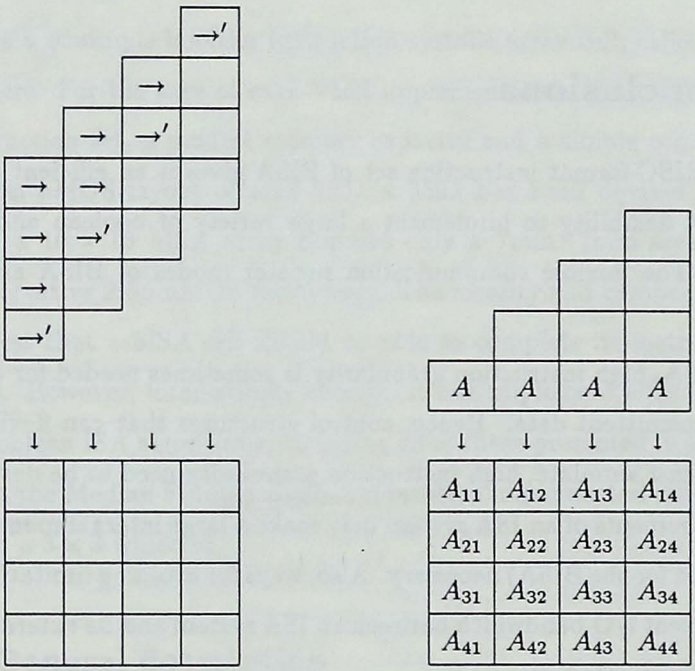
The data buffer design may be programmed in a fashion similar to the ISA, except that underlying wavefront programming model (and hence the buffer's control structures) must be extended to that used by microprogrammed ISA (see Section 4.1.1).

To illustrate this, consider programming the westward-moving data channel of the data buffers to provide data for an ISA reading a matrix A from the west. Assume that element A_{ij} is initially in data buffer cell (i, j) and that the current ISA program requires it to be read in at the ISA's western edge by the t_jth instruction diagonal, where $t_j = t_0 + j\lambda$. Here, the ISA program reads once from the west every $\lambda \geq 1$ cycles. Since the data data channel takes $j+1$ steps to reach is ISA cell $(i, 1)$ from data buffer cell (i, j), A_{ij} must be loaded at the diagonal corresponding to:

$$t_j - (j + 1) = t_0 - 1 - j(\lambda - 1)$$

This is illustrated in Figure 2.13 with $\lambda = 1$ for an ISA program loading a matrix loading from the west. This program loads A_{ij} into the A register of ISA cell (i, j), and is different from other ISA load matrix programs (see Section 3.7.1, [51]) which would load A_{ij} into cell $(i, n - j + 1)$.

The data buffer program here is also a skewed matrix, which is advanced n steps ahead of the corresponding ISA program. To load sub-matrices, an ISA-like



(a) ISA program (no selectors used) (b) synchronized data buffer program

('→' denotes 'C ← C_W'; '→′ denotes 'A ← C_W' and 'A' executed at data buffer cell (i, j) loads A_{ij} into the data channel)

Figure 2.13: Programming the western data buffer channel for loading a matrix into a 4×4 ISA

selector mechanism can be used to select the required rows, and the columns of the data buffer channel can be selected by the instruction stream[13]. Note that the 'skew' of the A instructions in Figure 2.13(b) is $\lambda - 1 = 0$. For $\lambda > 1$, in order for the selector bits to properly meet such lines of loading instructions, they need to propagate westward one unit every $\lambda - 1$ (rather than 1, as in the ISA) cycles. A method for implementing variable speed selector bits is given in Chapter 4.

2.7 Conclusions

The simple RISC-format instruction set of BISA gives it an efficient cell design and sufficient flexibility to implement a large variety of boolean and symbolic algorithms. The flexible communication register modes of BISA are a useful feature.

For the ISA, high instruction granularity is sometimes needed for algorithms using nontransmittent data. Hence, control structures that can flexibly implement, or at least simulate, high instruction granularity need to be developed.

The requirements of an ISA system may make a large internal memory (larger than that used for the BISA) necessary. Also, ways for avoiding limitations caused by an insufficient I/O bandwidth between an ISA system and its external memory need to be investigated.

LPGS partitioning is recommended when the data size exceeds the ISA size, since it is efficient and sufficiently flexible in practice. However, LSGP partitioning may be used instead, and can be efficiently implemented on most ISA programs with a surprisingly small control structure overhead.

It is important for a mesh's data interface to be designed to enhance overall mesh performance as much as possible. To meet this end, it must support LSGP partitioning (if the data size is too large) and should be partitionable (if the data size is too small). The (ISA) data interface design proposed in this chapter achieves these properties using very simple control structures such as those used by the ISA itself. The design of matching ISA program interfaces

[13]Thus, the data buffer channels may be programmed more efficiently by implementing them as an SISA [31], a special case of the ISA introduced in Chapter 3.

now needs to be addressed. An issue for future research is to investigate whether this period-efficient design is more area-period efficient than the standard area-efficient designs.

2.A Appendix: The BISA prototype, bISA

The features a of simple boolean instruction systolic array cell, called bISA, are presented here. For the sake of easy VLSI implementation, the bISA has a very simple instruction set, a modest memory capacity and a simple communication protocol. An nMOS layout of area $350\lambda \times 350\lambda$ has been devised for a bISA cell. Thus, a 16×16 bISA array requires only a 7mm×7mm area of silicon, with a conservative 2.5μ nMOS technology. The locality and compactness of the layout suggest that a bISA cell should be able to complete its instruction cycle within 10ns. However, interestingly enough, it can implement efficiently a large variety of boolean ISA algorithms, including all of those presented in section 2.2.2 (except that the Median Finding algorithm must be implemented differently, and is limited to a 3×3 window).

2.A.1 General description

The bISA has an instruction cycle based on a two phase clock, whose phases are denoted by ϕ_1, ϕ_2 respectively, as for the BISA (see Section 2.2.1).

The instructions for an array of bISA cells are sent south through the array via 9-bit shift registers. The selectors for instructions are passed east across a bISA array via 1-bit shift registers.

The bISA has the following memory:

1. A single communication register (C), as for the BISA. This register is one of the 'accumulator' operands of an instruction.
2. An 'accumulator' (A).
3. A register (R).

4. An 8-element stack (S), which can be used as extra storage. The stack can also be used for unary counting, by being 'cleared', and then using a conditional store instruction (ie. store only if the A register holds a high value) from the A register. Note that the stack has an undefined effect on underflow. This stack has the role of the more sophisticated ring-shift registers of the BISA.

5. Four read-only input communication registers (C_W, C_N, C_E, C_S), as for the BISA.

2.A.2 Informal description of the instruction set

The bISA instructions are here represented by their semantic effects; the form of these effects are given by:

$$dst \leftarrow \theta_1\ src \qquad \text{if } cond$$
$$dst \leftarrow\ dst\ \theta_2\ src \quad \text{if } cond$$

where:

$$\theta_1 \in \{\text{id}, \text{not}\}$$
$$\theta_2 \in \{=, \neq, \text{and}, \text{nand}, \text{or}, \text{nor}\}$$
$$cond \in \{\text{true}, \text{A} = 1\}$$

and either:

$$dst \in \{\text{C}, \text{A}, \text{R}, \text{S}\}$$
$$src \in \{\text{C}, \text{A}\}$$

or:

$$dst \in \{\text{C}, \text{A}\}$$
$$src \in \{\text{C}, \text{A}, \text{R}, \text{S}, C_W, C_N, C_E, C_S\}$$

Such an instruction set can be represented using a nine-bit RISC instruction format. It can be seen that the bISA shares many of BISA's simpler instructions. The bISA can execute 'conditional store' instructions (by choosing $cond = (A = 1)$), in which the result of the instruction is stored only if the A register contains a high value (1). If S is accessed to evaluate the result of the instruction, a *pop*

operation is performed on S, with the bottom element of S being used to compute the result. If *dst*=S, and *cond* evaluates to **true**, the result of the instruction is *push*ed onto S.

This informal description is concluded with a list of commonly used instructions:

$A \leftarrow A$	(no-operation)
$C \leftarrow (C \neq C)$	(clear C)
$R \leftarrow (R = C)$	
$C \leftarrow (C \wedge C_W)$	
$A \leftarrow S$	(pop from S onto A)
$S \leftarrow A$ if $A = 1$	(push from A onto S if $A = 1$)

The last example is used for unary counting.

2.B Appendix: LSGP partitioning for the Red Squares program

The RedSquares program of Section 2.2.2.1 can be efficiently implemented with LSGP partitioning on a BISA with no extra control overheads, provided that the BISA ring shift register length $l_R = \kappa^2$. The RedSquares program is of interest because its kth repetition, $1 \leq k < n$, features both *nontransmittent* data (eg. $S1'^k$ of equation (2.2)) and also the shifting of other data (eg. A^k and A'^k) west and north. Since the nontransmittent $S1'^k$ is updated with a commutative, associative operation (boolean disjunction), and only two orthogonal directions are required for shifting data, the LSGP partitioning method can be easily scheduled (eg. using, for northward and westward shifting, a 'forward' row-major or column major scheduling).

For the sake of simplicity, it will be assumed first that some of the registers (ie. A, B) of the ISA cells are $\kappa \times \kappa$ 'sub-matrix registers', whose rows (columns) are accessed via the current value of the index register i (index register j). These index registers may be incremented or reset in parallel during an ISA instruction. Figure 2.14 demonstrates the partitioning and an appropriate scheduling for the LSGP partitioning of the RedSquares program. The scheduling gives the time each element in the global matrix is updated, and assumes that $\kappa + 1$ instruc-

tions are required to update each row[14]. Note that the flow of ISA instructions [selectors] requires that each element of the sub-matrices of cell $(1,2)$ [cell $(2,1)$] be updated one time unit after the corresponding element of the sub-matrix of cell $(1,1)$ is updated. This scheduling is different from the non-partitioned (ie. $\kappa = 1$) ISA scheduling, where the (i,j)th element is updated $i+j-2$ steps after the $(1,1)$th element.

11	12	13	14	15	16	$\cdots$
21	22	23	24	25	26	$\cdots$
31	32	33	34	35	36	$\cdots$
41	42	43	44	45	46	$\cdots$
51	52	53	54	55	56	$\cdots$
61	62	63	64	65	66	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

(a) partitioning of sub-matrices

1	2	3	4	2	3	$\cdots$
6	7	8	9	7	8	$\cdots$
11	12	13	14	12	13	$\cdots$
16	17	18	19	17	18	$\cdots$
2	3	4	5	3	4	$\cdots$
7	8	9	10	8	9	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

(b) 'forward' row-major scheduling

Figure 2.14: LSGP partitioning of the RedSquares program around cell $(1,1)$ for $\kappa = 4$

The partitioning of the first two diagonals of the ComputeA^{k+1} sub-program (which form A'^k), for ISA columns 1 to $n-1$, is for this scheduling:

$$\left.\begin{array}{l} i := 1; j := 1; \\ \mathrm{C} \leftarrow \mathrm{A}[i,j]; \\ \text{Update}A'; \\ \mathrm{A}[i,j] \leftarrow \mathrm{A}[i,j]\,\mathrm{C_E};\ \ i := i+1; j := 1; \end{array}\right\} \textbf{repeated } \kappa \textbf{ times}$$

where UpdateA' is given by:

$$\mathrm{A}[i,j] \leftarrow \mathrm{A}[i,j]\,A[i,j+1];\ \ j := j+1;\ \ \} \textbf{ repeated } \kappa - 1 \textbf{ times}$$

Here, if the C register of cell (1,2) is loaded at times (1, 6, 11, 16) with ($A^k_{1,5}$, $A^k_{2,5}$, $A^k_{3,5}$, $A^k_{4,5}$) in turn, these values are read by cell (1,1) at times (4, 9, 14, 19) in order to compute ($A'^k_{1,4}$, $A'^k_{2,4}$, $A'^k_{3,4}$, $A'^k_{4,4}$) respectively. The period for this code, which performs the κ^2 updates, is only $(\kappa^2+\kappa+1)$. The code for partitioning the last two diagonals of ComputeA^{k+1} is only slightly more complicated.

[14]An extra instruction is needed for communication reasons.

Similarly, the code for the first diagonal of the ComputeS^k sub-program is given by:

$$\left.\begin{array}{l} i := 1; j := 1; \\ \mathrm{C} \leftarrow \mathrm{A}[i,j]; \\ \mathrm{Update}S1'; \\ \mathrm{C} \leftarrow \mathrm{C} \vee \mathrm{C_W}; i := i+1; j := 1; \end{array}\right\} \textbf{repeated } \kappa \textbf{ times}$$

where Update$S1'$ is given by:

$$\mathrm{C} \leftarrow \mathrm{C} \vee \mathrm{A}[i,j]; j := j+1; \ \} \textbf{ repeated } \kappa - 1 \textbf{ times}$$

Here, if the C register of cell (1,1) is loaded at times (5, 10, 15, 20) with ($S1'^k_{14}$, $S1'^k_{24}$, $S1'^k_{34}$, $S1'^k_{44}$) in turn, these values are read by cell (1,2) at times (6,11,16,21) in order to compute ($S1'^k_{1,8}$, $S1'^k_{2,8}$, $S1'^k_{3,8}$, $S1'^k_{4,8}$) respectively. Note that the summation for $S1'^k_{1,n}$ is now taken in the order:

$$(S1'^k_{1,1} \vee \ldots \vee S1'^k_{1,4}) \vee (S1'^k_{1,5} \vee \ldots \vee S1'^k_{1,8}) \vee \ldots \vee (S1'^k_{1,n-3} \vee \ldots \vee S1'^k_{1,n})$$

relying on the associative property of boolean disjunction. The code for the last diagonal of Compute$S1^k$ is similar.

Note that the i and j registers here are incremented (rather than decremented) because $\mathrm{A}[i,j]$ is updated using $\mathrm{A}[i,j+1]$ and $\mathrm{A}[i+1,j]$, ie. data is being shifted to the north and west directions. If, for example, data is to be shifted north and east, the scheduling would change so that the i register is incremented and the j register is decremented.

Since the implementation of 'sub-matrix registers' and their accessing is complex, particularly for boolean ISA, it is outlined how LSGP partitioning can be implemented using the ring-shift registers of the BISA. Consider converting the UpdateA' sub-program above by using the $\mathrm{R_1}$ and $\mathrm{R_2}$ ring-shift registers (assumed to each contain exactly $l_\mathrm{R} = \kappa^2$ bits) to replace the A 'sub-matrix register'. $\mathrm{R_1}$ stores the sub-matrix in row-major order (the same order as the scheduling) and $\mathrm{R_2}$ is a copy of $\mathrm{R_1}$. Using $\mathrm{R_1}$ to access $\mathrm{A}[i,j]$, $\mathrm{A}[i,j+1]$ can be accessed by advancing $\mathrm{R_2}$ one bit relative to $\mathrm{R_1}$. After all κ^2 elements of $\mathrm{R_1}$ are updated, it is necessary to again copy $\mathrm{R_1}$ into R_2. Surprisingly, the use of the ring-shift registers here increases the period by a factor of less than two.

Chapter 3

Program Compression on the Instruction Systolic Array

3.1 Introduction

The architectural concept of the Instruction Systolic Array (ISA), presented in Section 1.2, preserves the advantages of the systolic array such as simplicity, regularity and locality, whereas its main disadvantage, the lack of flexibility, is overcome. Handling programs for instruction systolic arrays introduces various new problems in the area of programming languages and compiler construction. Since ISA provides an independent stream of instructions for each column, these programs tend to be large. This raises two questions concerning ISA design: will I/O (pin) limitations become the bottleneck for ISAs? Where and in which form will the programs be stored such that the process of fetching the program does not delay the computation?

This suggests the use of program compression to overcome the loss of overall ISA performance due to I/O limitations and reduce ISA system program storage hardware. Compressing programs requires a control structure mechanism (called a *matrix restorer*) to restore the compressed version of the ISA program into its original (matrix) form. This must occur in a synchronous manner such that each column receives one instruction per time unit, with the *matrix restorers* themselves being linear instruction systolic arrays. It is convenient to distin-

guish between two forms of program compression: *vertical* program compression, in which a sequence of instructions to be executed at a particular processing element is compressed, and *horizontal* program compression, in which similar instruction sequences over a range of processing elements are compressed into a single, generalized instruction sequence.

While these techniques are specifically designed for the ISA, they also have some application to SIMD meshes, Wavefront Array Processors and MIMD meshes (Chapter 7 describes the application of program compression techniques to MIMD meshes). Related works on program compression are referred to Section 3.2. These include relevant (matrix) data compression techniques (Section 3.2.1) and existing methods of program compression for non-ISA architectures (Section 3.2.2). The motivations for ISA program compression are given in Section 3.3.

In this chapter, four different methods of program compression for ISA programs are presented. The first method (explained in detail in Section 3.4) involves a variation of the concept of the instruction systolic array (the *single instruction systolic array* (SISA)), reducing its flexibility and/or performance for a minority of programs. It involves also a slight simplification of the ISA concept and there is no additional hardware necessary for its implementation.

The second method (Section 3.5) carries the concept of microprogramming over to the ISA, ie. macro instructions (representing short ISA subprograms) are defined and ISA programs are expressed in terms of these macros. Here additional control structures are needed to translate macros into ISA readable form (short ISA programs). This method has the advantage that it allows the implementation of complex instructions on an ISA having only a simple cell design, increasing the ISA's flexibility.

The third method consists of introducing language concepts allowing the specification of subprograms and loops. This method is described in Section 3.6. It uses the ISA diagonal or wavefront as a fundamental unit of description, which turns out to be appropriate for program compression. This idea is further elaborated in Chapter 5. Program restoration firstly consists of replacing names of subprograms by corresponding ISA programs and, in the case of loops, replicating

the body of the loop as often as specified. The diagonals also need to be restored from their compressed form. Here, the matrix restorers serve as an interpreter. This method has the advantage that it makes ISA programs more readable.

The fourth method (presented in Section 3.7) makes use of the regularity of ISA program matrices. Regular expressions of a specific form are used to specify ISA programs and a matrix restorer, suitable for VLSI implementation, is designed to restore these expressions into ISA program matrices. The third and fourth methods gain by far the best compression rates but are also the most expensive in control structure overheads.

An evaluation of these methods, in terms of both compression rate and hardware overhead, for a moderate sized ISA is given in Section 3.8. The application of these methods to other mesh architectures is outlined in Section 3.9. Conclusions are given in Section 3.10.

This chapter is a revised and extended version of a paper of the same title co-authored with Schröder [53]. It introduces the concept of *program compression* as a technique to improve ISA (and other meshes') performance. Sections 3.2, 3.3, 3.5, 3.7, 3.8, 3.9, and 3.10, together with the implementation issues of 3.6, are the author's work.

For reading this chapter, the sections describing program compression methods are largely independent, with common program compression concepts being introduced in Section 3.3. However, the concepts introduced in Sections 3.5, 3.6 and 3.7 form the basis of Chapters 4, 5 and 7 respectively.

The ISA instruction set assumed for this chapter is similar to that described for the boolean ISA, BISA, (Section 2.2), except that it can support the appropriate data types (integer or boolean) required by the particular example programs given in this chapter. For simplicity, one output communication register (C) mode of BISA is mainly used for these programs.

A simple ISA program to demonstrate program compression techniques is the ringshift program (see Figure 3.1). This program performs a ringshift of the C registers of the columns and rows of a $n \times n$ ISA. The execution time of the program is $n + 4$ clock cycles and its period is 4. Figure 3.2 shows the contents

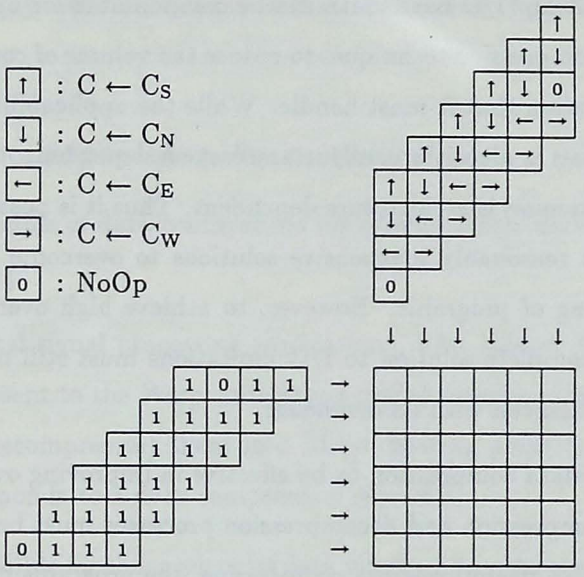


Figure 3.1: Ringshift program for a 6 × 6 ISA

of the C registers of the ISA before and after execution of the program of Figure 3.1.

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

8	9	10	11	12	7
14	15	16	17	18	13
20	21	22	23	24	19
26	27	28	29	30	25
32	33	34	35	36	31
2	3	4	5	6	1

Figure 3.2: C registers before and after the ringshift program for a 6 × 6 ISA

3.2 Program compression — related work and concepts

As discussed in Section 2.5, a parallel architecture's high computational bandwidth can be limited by an inadequate I/O bandwidth. This is serious because current technology allows the former to grow very large, especially for scalable

parallel architectures such as meshes, whereas the latter is difficult to increase in proportion [25]. A low I/O bandwidth can be compensated for by applying compression and decompression techniques to reduce the volume of control (program) and data information that it must handle. While the applicability of data compression techniques is algorithm and/or application dependent, that of program compression techniques is architecture dependent. Thus it is possible to come up with general but reasonably inexpensive solutions to overcome I/O limitations due to the loading of programs. However, to achieve high overall architecture performance, a complete solution to I/O limitations must still be found, whose performance is balanced with its overheads.

For program/data compression to be effective in improving overall mesh performance, the compression and decompression processes must be relatively fast. Fortunately for the case of program compression, the programs can be originally represented and stored in compressed form, and the decompression can be easily done in parallel by the matrix restorers.

However, the case for data compression is different. Let IO denote the I/O bandwidth between an $n \times n$ mesh and its external environment. Data compression would only be required if the I/O bandwidth is limiting the mesh performance, ie. $IO = o(n)$. Now, the time taken to load/unload an $n \times n$ matrix is $\Theta(n^2/IO)$ and hence the time taken to compress/decompress this matrix must be $O(n^2/IO)$. If the compression must be performed serially (requiring time $\Theta(n^2)$), it can only be worthwhile if either $IO = O(1)$ or the matrix was originally stored (externally) in compressed form. For the latter case, if the matrix was produced by the mesh, the mesh should itself perform the compression (in parallel).

An alternative to compression techniques for alleviating the bottleneck due to loading data (and programs) in/out of a mesh is to employ *recycling*. Recycling involves increasing the internal storage of the mesh (and its data interface) so that as much data (or programs) as possible is loaded once and then reused when needed (as discussed in Section 2.4). Provided the data is re-used a large number of times, ie. more than a constant number of times, the overall I/O traffic can be considerably reduced. This is a more general solution to I/O limitations, but is

expensive in terms of hardware and has the disadvantage that for some algorithms the internal memory must increase faster than the number of processing elements [25].

3.2.1 Data compression on meshes

The simplest example of data compression for meshes comes under the category of data formatting:

> eg. for digital signal processing applications, 8-bit integer data from sensors are sent to the Warp computer's (data) Interface Unit, which converts (decompresses) them into 32-bit floating point [1, p1530]. This corresponds to a data compression rate of 4.

Similarly, it is possible for the mesh or its data interface to generate matrices made of repetitions of a few simple patterns (which can be represented in a compact form),

> eg. produce null matrices, produce random (ie. grey) matrices for image processing applications, fill in 'gaps' in sparse matrices and recover symmetric matrices from their lower triangular parts.

which can sometimes effectively achieve a significant degree of compression. Some of these decompressions can be achieved by the mesh itself in linear time, which is sufficiently fast to achieve overall speedup provided $IO = o(n)$. Less trivial examples of this nature include multiplication of arbitrary matrices with Hankel and Toeplitz matrices (which can be represented by vectors; their matrix form is restored using "bus expanders" just before entering the mesh [59, pp33-38]). Also, in some neural network applications, which often have efficient mesh implementations [26, pp269-280], there may be sufficient symmetries in the network to apply compression techniques to the corresponding synaptic strength matrices.

Data compression can also be effectively achieved in some situations by reformulating the algorithm to take advantage of data redundancies, eg. mesh implementations of band matrix operations [26, pp177-178][43]. However, this does not always lead to an efficient mesh implementation.

More general data compression techniques such as data transforms and vector quantization [26, pp573-578] cannot generally be employed since the compression/decompression algorithms are too expensive and may introduce unacceptable distortions in the data. However, text compression techniques such as run length encoding and a simple form of vector quantization (which corresponds to microprogramming) are sufficiently simple to be applicable to program compression.

3.2.2 Program compression on existing meshes

This section examines the extent to which existing meshes incorporate program compression techniques. Note that microprogramming techniques, even on uniprocessors, also achieve some degree of program compression.

The Connection Machine (CM) [14] is normally regarded as a SIMD hypercube rather than an SIMD mesh. However, the design decisions made for program loading apply equally to an SIMD mesh of similar scale and granularity. Since the CM is comprised of fast bit processors, program compression is applied to the SIMD instruction stream in the form of microprogramming so that the CM host can "amplify" its program I/O bandwidth to keep up with the CM [14, pp88-89] The CM implements masking via status registers that can be set in each cell (depending on the cell's address and data contents). CM cell instructions are conditionally executed on these status registers.

The ICL DAP [17, pp243-247] is a fine-grained SIMD mesh. The DAP similarly uses program compression on its instruction stream, using a 60 instruction-word "instruction buffer" into which the DAP host loads instruction sequences (including small loops, later to be unfolded at run time). For the DAP, the main masking mechanisms are the "row and column vectors", for which DAP cell (i, j) combines the ith bit of the row vector and the jth bit of the column vector to decide whether to execute the current instruction. This masking mechanism is similar to that of the SISA, both reducing the masking information from n^2 bits per cycle to $2n$ bits per cycle. Current implementations of the DAP have $n = 64$. Program compression then should also be applied to the row and column vectors (requiring 128 bits per cycle bandwidth), being substantially larger than the

instruction stream (requiring 32 bits per cycle bandwidth).

In summary, SIMD meshes, which inherently utilize *horizontal* program compression since their instruction streams can be used by all elements, may also require *vertical* program compression in their instruction streams. Large SIMD meshes using 'row/column vectors' (or similar mechanisms) would also require (*vertical* and *horizontal*) program compression for its masking mechanisms. In this case, the program compression techniques presented here for the ISA can be similarly applied to SIMD meshes.

The Wavefront Array Processor (WAP) is usually implemented with cells containing their own instruction memories, as for MIMD meshes [26, pp299-300]. In this case, the comments on MIMD meshes (below) apply equally to WAPs. However, WAPs could also be implemented using "instruction wavefronts" (like the ISA) [20, Sect.6.3.1], in which case the program compression techniques for the ISA could be applied to WAPs also.

MIMD meshes such as Inmos Transputer arrays [26, pp467-470] and the Warp processor [1] have medium (microprocessor) to coarse-grained processing elements (cells) with several Kwords of program memory each. These memories are thus large enough to store many programs, and thus the program I/O bottleneck can be alleviated using *recycling*. The program for each cell is coded in a compressed form (using 'loops' or run-length encodings) as in conventional uniprocessors, and further (horizontal) compression can be effectively obtained by downloading identical code segments into all cells. This can be made more powerful by using registers containing the cells 'id', ie. row and column indices, so that cells can interpret identical code segments differently. However, Chapter 7 demonstrates that the application of program compression techniques similar to ISAC can reduce the program loading times and cell program memory size.

3.3 Motivations for compressing ISA Programs

An example motivating the need for program compression techniques is given by the boolean ISA prototype, bISA. As discussed in Appendix 2.A, a conservative estimate shows that current technology is capable of implementing a 16×16 bISA

array on a chip of side 1cm, with a cycle time of 10ns. Since bISA uses 9-bit instructions, the chip has an program input bandwidth of 16000 Mbits s^{-1}. For comparison, the peak host to Warp processor data I/O bandwith (using a VME bus with DMA) is 45 Mbits s^{-1} [1, p1530]. Considering the fact that the Warp cells are each composed of hundreds of chips, this comparison demonstrates that a bISA chip alone has an unachievable program input bandwidth. This problem can only be overcome with the introduction of powerful program compression techniques[1].

Consider the program interface for an $n \times n$ ISA (similar arguments apply to an $m \times n$ ISA). This ISA is intended to execute a large number of programs, some of which are assumed to have $\Omega(n)$ period.

A straightforward implementation of the program interface is given in Figure 3.3 (the SISA would have a similar interface). Matrices representing ISA programs, some which have size $\Omega(n^2)$, are loaded from the external *instruction (selector) memory*. For sufficiently large n, these memories would need to be *parallel*, ie. access $\Theta(n)$ instructions (selectors) simultaneously, to enhance the rate of supplying programs to the ISA. To supply the ISA with exactly n instructions per cycle, both the instruction (selector) memory and its $\Theta(n)$ number of wires connecting to the ISA would need to made very fast. A more economical alternative is to employ recycling, ie. allow these programs to be a little slower (by a constant factor) and introduce an $\Omega(n^2)$ area *instruction (selector) buffer*, capable of holding several ISA programs.

A similar situation arises for the ISA data interface [35, pp28-30] (see Section 2.6), except that it is convenient to assume that *data buffers* are connected to the communication registers of all ISA boundaries (rather than just the north and west boundaries).

[1]Note that for *optimal* program compression methods, the minimum (compressed) program input bandwidth actually decreases with increasing array size. This is because the program period is usually $\Omega(n)$; hence increasing n gives more time for compressed programs to be loaded. It is a fair comment that the host to bISA array data I/O bandwidth might still be unachievable; however, for typical applications, its average external I/O rate might only be n bits per several tens of cycles.

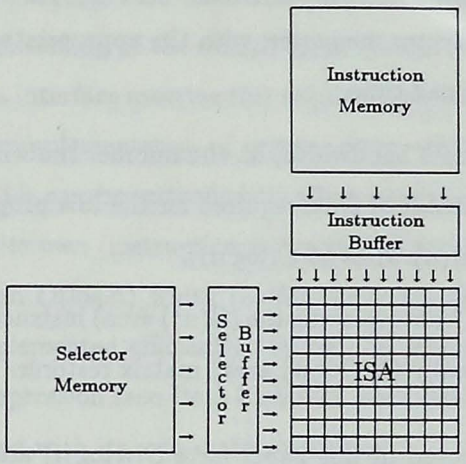


Figure 3.3: A 10×10 ISA program interface

An important consideration is the volume of information passing this interface, since systolic array, particularly ISA, computations are often limited by I/O requirements [22, 25]. When the size of a program is larger than the size of the program's data (often the case with ISAs — see Table 3.1), or when the program's data already resides within the data buffers, the overall ISA performance (speed) can be improved considerably by passing a compressed (optimally to $O(\log n)$ space[2]) representation of the program across the program interface. The compressed representation must then be restored, by a *matrix restorer*, to its full matrix form before entering the ISA cells. In any case, program compression techniques can also reduce the overall area (cost) of the ISA system in the following ways:

1. Reduction of the area of the instruction (selector) memory (optimally, from an $\Omega(n^2)$ area parallel memory to an $O(\log n)$ area serial one).

2. Obtaining a precise, parameterized representation of ISA programs.

[2]Here, $O(\log n)$ space is considered optimal since a constant $n \times n$ matrix still requires $O(\log n)$ space, to specify n, to be represented.

This means that for every required sub-array size[3] and values of other parameters[4], only one version of an ISA program need be stored in the instruction and selector memories, with the appropriate parameter(s) being resolved at load-time.

3. Reduction of the I/O bandwidth, ie. the number the wires and correspondingly the number of pins, required for the ISA program interface (optimally, from $\Theta(n)$ wires to $O(\log n)$).

4. Reduction of area from replacing the ($\Omega(n^2)$ area) instruction (selector) buffer by a (optimally $O(n \log n)$ area) matrix restorer.

All matrix restorers proposed in this paper have $O(n \log n)$ area, consuming a small part of the $\Theta(n^2)$ area of the overall ISA system. This raises the implementation question: is it more area-period efficient to overlap the loading of (the information for restoring) the next program into the matrix restorer with the execution of the current program? Consider the case of loading ISA programs having a typical period of $\Omega(n)$. The matrix restorers for such programs can be loaded in an $O(\log n)$ period. Overlapping has an advantage of the saving the $O(\log n)$ loading period, at the expense of doubling the $O(n \log n)$ area matrix restorer tables (storage). We suggest that for at least up to moderate array sizes, the constants of proportionality favour overlapping, and the implementations of program compression presented in this chapter can accommodate this. In particular, these designs efficiently support overlapping since the matrix restorer tables are used in a read-only manner. Thus, using a two-phase VLSI implementation [40, pp226-233], each table can be used (read) on the first phase and can be loaded (written) on the second phase, using a single bus.

In any case, it is possible (and desirable) to minimize the matrix restorer storage by breaking ISA programs into 'irreducible' $\Omega(n)$ sub-programs,

ie. sub-programs that are not expressed as sequences of significantly smaller, in terms of code size, $\Omega(n)$ sub-programs,

---

[3]ie. when the problem size is smaller than the array size; see Section 2.5, and [35, pp66-67].

[4]eg. the parameter d of program RotHV(d) of Section 3.6.

and loading and using 'irreducible' sub-programs in turn.

Section 2.5.1 argued that data and program interfaces should be designed to allow the partitioning of the overall mesh system into a small number of sub-systems. A data interface meeting this requirement has been presented in Section 2.6.3. With the implementation of prgram compression, the ISA program interface of Figure 3.3 can be extended to allow partitionability by providing each sub-array with its own (instruction and selector) matrix restorers. If the matrix restorers have an $O(\log n)$ width (as for the methods presented subsequently), this can be implemented efficiently. Indeed, to retain the pin count advantage of program compression (see Item 3 above), each separately packaged sub-array must be provided with its own matrix restorers.

3.4 Program compression using the SISA

The flow of control information in an ISA is asymmetric, ie. w_i-bit instruction codes move through in north-south direction while only 1-bit selectors move west-east. This asymmetry is overcome and at the same time a compression rate of approximately $w_i/2$ is achieved by the concept of the *single instruction systolic array* (SISA) [31]. In the SISA, column selectors (in addition to the row selectors) are pumped through the columns of the array while the instructions travel in diagonal wavefronts through the array, starting at the north-west corner. Now an instruction is only executed if it meets with a row selector and a column selector equal to "1".

The SISA ringshift program is shown in Figure 3.4.

While the compression rate gained by introduction of the SISA concept is only moderate, there are no additional costs involved in its implementation. The loss in flexibility paid might then be acceptable. A large variety of efficient data permutation algorithms equally efficient on the SISA and the ISA is given by Lang [35, Ch.6]. The SISA still seems to be significantly more powerful than the SIMD concept (see [31]). The reader might convince him/herself of this by trying to implement the ringshift program on an SIMD mesh.

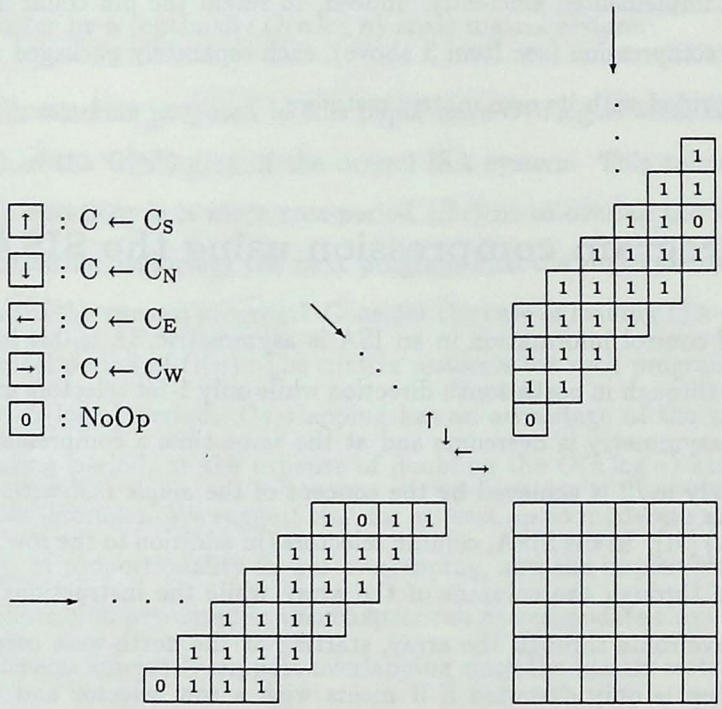


Figure 3.4: Ringshift program for a 6×6 SISA

3.5 Program compression using microprogramming

Microprogramming is a standard technique in microprocessor design [55, Ch.10], and is used in many array processing chips [26, pp454-474]. It has the benefits of simplifying processor design and reducing the communication bandwidth from program memory to the CPU (and is thus a form of program compression). For the ISA, a very simple form of microprogramming is proposed (a simplification of that given in Chapter 4 and in [34]) that still retains these advantages. By 'microprogramming', it is meant that *microcodes*, ie. sequences of $\lambda = O(1)$ w_i-bit ISA instructions are represented by single w_m-bit macro(instruction)s, where $w_m << \lambda w_i$ (see Figure 3.5(a)). For an ISA program, its microprogramming length parameter $\lambda \geq 1$ is not arbitrary: it needs to correspond to a suitable semantic unit of the program (otherwise, the number of macros, and hence w_m, will become unacceptably large). Microprogramming achieves a compression rate of $\lambda w_i / w_m$ (and can achieve the same factor in pin reduction).

A simple example of program compression using microprogramming using $\lambda = 4$ is for the ISA ringshift program (see Figure 3.1). Three macros (requiring $w_m = 2$) are used to represent the instruction sequences '$0; \leftarrow; \downarrow; \uparrow$' (for ISA column 1), '$\rightarrow; \leftarrow; \downarrow; \uparrow$' (for ISA columns 2 to $n-1$) and '$\rightarrow; 0; \downarrow; \uparrow$' (for ISA column n). For $w_i = 8$, this achieves a compression rate of 16.

Also, the microprogramming *skew* parameter μ, where normally μ divides λ, can also be introduced[5]. This gives other important advantages, important for the development of the ISA for general-purpose array processing. It enables the composition of a large range of powerful (macro)instructions from a simple (micro)instruction set. Thus, the ISA cells can be kept as small as possible, enabling both a faster clock cycle and greater parallelism (more cells can be fitted on a chip). Powerful instructions may be required by an ISA to implement some systolic algorithms efficiently (or at all). Such algorithms use nontransmittent data.

[5] An ISA without microprogramming corresponds to $\mu = 1$.

3.5.1 Implementation of microprogramming

Unlike the usual use of microprogramming, which may be called *static microprogramming*, a form of *dynamic microprogramming* is proposed here for which length, skew and contents of the microcodes can be changed between different ISA programs (or sub-programs of length $\Omega(n)$).

Instruction macros entering the ISA chip from the north are decoded into their respective sequences of λ ISA instructions by a row of *decode tables* (in this case, the *matrix restorer*) situated above the ISA cells (see Figure 3.5(b)). The microcodes for the next ISA program can be simultaneously loaded (into currently unused table locations) across the row in a systolic manner. For each macro, λ selector bits are used, having the added power that each row of an ISA can omit arbitrary instructions within a macro. The selector sequences themselves may be coded by macros, for $\lambda > w_m$, requiring a column of selector decode tables to the west of the ISA.

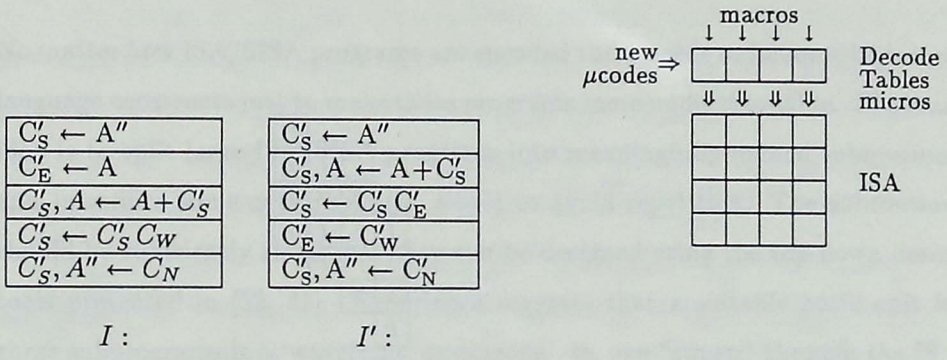
For the simple model of microprogramming proposed here, the [micro-] instructions (selectors) are 'skewed' by $\mu \geq 1$ units[6]. These skews are implemented by having a selector (instruction) queue of [variable] length μ in each ISA cell. These queues slow down the rate of the selector (instruction) streams through the ISA by a factor of μ, thus ensuring their proper co-ordination.

For the purposes of this chapter, it can be assumed that the ISA cells' input communication registers latch new values either every μ microcycles (a simpler model, more like traditional microprogramming), or every microcycle (allowing multiple data transfers in a given direction per macro).

Our experience suggests that $w_m = 4$ and a decode table length of 24 words will be sufficient for most 'irreducible' ISA sub-programs (cf. Table 3.2). Under these circumstances, the extra control logic and I/O bandwidth due to the systolic loading of the decode tables is quite modest. The area overhead for implementing the decode tables might easily be offset by the pin reduction.

While the instruction and selector queues may also be implemented efficiently

[6]The microprogrammed ISA model of Chapter 4, and also [34], allows these skews to be independent. This reduces program period in some cases.



(a) microcodes for program of sect. 3.5.2 (b) decode tables

Figure 3.5: Instruction decode tables on a 4×4 ISA

in VLSI (see Section 4.3), there is a delay penalty for dynamically variable values of μ. Upon changing from an ISA program of microprogramming skew μ to one of skew μ', where $\mu' < \mu$, the ISA must be fed $O(n(\mu' - \mu))$ diagonals of 'no-operation' microinstructions to prevent the faster moving micros of the second program from colliding with those of the first.

3.5.2 A microprogrammed transitive closure program

The algorithm of Kung et. al. [57, pp192-197] (see Section 2.2.2.4) to find the transitive closure of an $n \times n$ adjacency matrix A can be implemented on an $n \times n$ ISA having a simple (BISA-like) instruction set using microprogramming with $\lambda = \mu = 5$. The algorithm uses the matrices A, A′ and A″, initialized respectively to $\mathbf{0}, A + I$ and $A + I$, where I is the identity matrix and $\mathbf{0}$ is the zero matrix. The matrix A resides in the ISA registers of the same name, whereas the nontransmittent matrices A′ and A″ are passed east and south through the ISA (see Figure 2.5). The algorithm consists of three of these passes; and at the kth step of a pass, cell (i, j) performs the computation:

$$\begin{aligned} \mathrm{A}_{ij} &\leftarrow \mathrm{A}_{ij} \vee \mathrm{A}'_{ik}\, \mathrm{A}''_{kj}; \\ \mathrm{A}'_{ik} &\leftarrow \mathrm{A}_{ij} \qquad \text{if } j = k; \\ \mathrm{A}''_{kj} &\leftarrow \mathrm{A}_{ij} \qquad \text{if } i = k \end{aligned}$$

Here, the four-output register mode of a BISA-like instruction set is assumed,

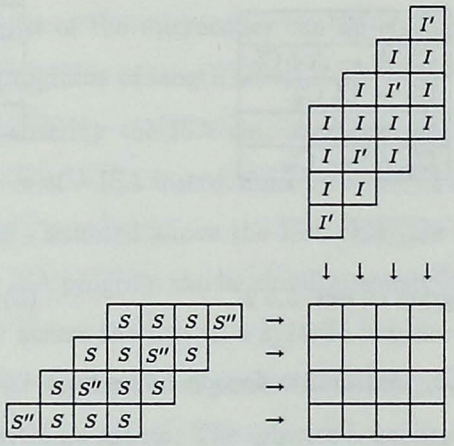


Figure 3.6: Transitive closure algorithm (one pass) on a 4×4 ISA

and ISA internal registers A′ and A″ are used as temporary storage for matrices of the same name. Only two instruction macros, I and I' (determining the final value of C'_E), and two selector macros, $S = 11111$ and $S'' = 11110$, (determining the final value of C'_S) are required (see Figure 3.5(a)). The resultant algorithm for a single pass is shown in Figure 3.6, with both the instruction and selector matrices resembling identity matrices. Consider the kth step (kth macro diagonal passing through the ISA). An I' instruction meets cell (i, j) only when $j = k$, and there updates A'_{ik}. Similarly, an S'' selector meets cell (i, j) only when $i = k$, and there updates A''_{kj}.

Without using microprogramming, the same ISA can implement this algorithm using using 13 instructions per step (see Section 2.2.2.4), by introducing rather complicated instruction replications. More interesting examples of how microprogramming enhances both the design and efficiency of ISA programs are given in Chapter 4. Microprogramming offers an efficient technique to implement such algorithms on a general purpose ISA, while allowing algorithms requiring simpler instructions to run at their maximal rate.

3.6 Program compression using Subroutining

No matter how ISA/SISA programs are encoded there needs to be some high-level language constructs just to make these programs more understandable. The main idea is to split large ISA/SISA programs into meaningfully-named subroutines, and introduce some constructs (ie. *loops*) to avoid repetition. The subroutines should be sufficiently small that they can be designed using the top-down design tools presented in [52, 41]. Experience suggests that a suitable basic unit for these subprograms is a 'wavefront' or diagonal, ie. one "sweep" through the ISA. Subroutining is capable of coding most ISA programs in $O(\log n)$ space, and optimally satisfies the requirements of Section 3.3.

For convenience, a shorter (but less readable) way of encoding $m \times n$ ISA/SISA programs is first introduced. This is similar to the Pascal-like SISA language introduced in [35, Ch.4]. Diagonals are presented as a sequence of pairs/triples $(ic, rs)/(op, cs, rs)$. Here ic and cs are n-tuples, rs is an m-tuple and op is a single instruction. The example programs of Figures 3.1 and 3.4 can then be written as shown in Figure 3.7.

Rot1H:			Rot1H:		
	$< 0(\leftarrow)^{n-1},$	$(1)^m >;$		$<\rightarrow,$ $0(1)^{n-1},$	$(1)^m >;$
	$< (\leftarrow)^{n-1}0,$	$(1)^m >$		$<\leftarrow,$ $(1)^{n-1}0,$	$(1)^m >)$
Rot1V:			Rot1V:		
	$< (\downarrow)^n,$	$0(1)^{m-1} >;$		$<\downarrow,$ $(1)^n,$	$0(1)^{m-1} >;$
	$< (\uparrow)^n,$	$(1)^{m-1}0 >$		$<\uparrow,$ $(1)^n,$	$(1)^{m-1}0 >$

Figure 3.7: ISA/SISA programs for rotation

Here names have been assigned to ISA/SISA subroutines and superscripts have been used to express diagonals. These superscripts can also be seen as a method of program compression, and at the same time parameterize the programs with respect to ISA size. Furthermore, subroutines can be used inside a "**repeat**" command: thus, an SISA program which rotates the contents of the C registers d positions vertically and horizontally can then be encoded as shown in Figure 3.8. The subroutine name is followed by a list of variables (in this case with only one entry, d, giving the distance of rotation). Also, Subroutining allows different types of parameters (variables): value parameters as introduced and

address parameters which for example allow the use of the same subroutine code for rotating the contents of different registers.

RotHV (d):

Rot1H:

$<\rightarrow,\quad 0(1)^{n-1},\quad (1)^{m}>;$

$<\leftarrow,\quad (1)^{n-1}0,\quad (1)^{m}>)$

Rot1V:

$<\downarrow,\quad (1)^{n},\quad 0(1)^{m-1}>;$

$<\uparrow,\quad (1)^{n},\quad (1)^{m-1}0>$

repeat d **times**

Rot1H;

Rot1V.

Figure 3.8: SISA program for vertical and horizontal rotation by d positions

Additional hardware is needed to implement these higher-level constructs on the ISA/SISA. The general implementation is rather complex to describe but is efficient, and is given in detail for the lower-level, but otherwise similar, language ISADL in Section 5.3. For simplicity, consider implementing Subroutining on the SISA. The $O(1)$ different row (column) selector patterns of a program are loaded into the tables of $O(n)$ area *diagonal restorers* to the west (north) of the SISA. These are controlled by an $O(\log n)$ area *diagonal sequencer* located to the north-west of the SISA. The diagonal sequencer handles functions such as subroutine and parameter substitutions and **repeat** commands, and selects the appropriate instructions and diagonal restorer table indices for each diagonal to be sent to the SISA. To ensure proper synchronization, these indices are pumped systolically through the diagonal restorers.

Consider the loading of the column selector pattern '$(1)^{n-1}0$' into the north diagonal restorer. This loading process occurs in an instruction systolic fashion, from the left (cell 1) to the right (cell n). Cells 1 and $n-1$ are first marked,[7] then all of the cells in between are marked, and the '1' selector is then loaded in the marked cells (cells 1 to $n-1$). The '0' selector is then loaded in the unmarked cells (cell n). Efficient loading of the diagonal '$\ldots(E)^{k}\ldots$' requires that $|E|$ be a

[7]The marking of cell i can be done in $O(\log i)$ steps using systolic bitwise counting.

power of 2 — in practice, this is almost always the case[8].

Alternatively, if only a few (ie. $O(1)$) types of selector patterns are ever used, they can all be stored permanently in the diagonal restorer's tables. Thus, selector patterns of length n can be represented compactly by their $O(1)$ table indices. This form of program compression is similar in concept to microprogramming except that it is horizontal rather than vertical.

To handle more general ISA programs, the **repeat** construct needs to be generalized, as is illustrated in Figure 3.9 for the microprogrammed transitive closure program of Section 3.5. In this case, the diagonal(s) corresponding to

TransClos:

for $k := 1$ **to** n **do**
$< I^{k-1}I'(I)^{n-k},\ S^{k-1}S''(S)^{n-k} >$

Figure 3.9: Transitive closure program for an $n \times n$ ISA

$k = 1$ would be initially loaded into the diagonal restorer. Upon each iteration, these would be shifted 'downstream' one unit to produce the diagonal for the next iteration. However, this scheme does not work for a few ISA programs which require a single diagonal to be shifted 'upstream' one unit for at least two consecutive steps (eg. the LoadMat program of Section 3.7). Such programs require recoding, with sometimes a decrease in efficiency (as explained in Section 5.4.1.3). For *divide-amd-conquer* programs (see Section 3.7.1), the diagonals contain expressions depending exponentially on k: hence they need to be updated in a different way, as explained in Section 5.4.1.2.

3.7 Program Compression using ISAC

ISA Compressed coding (ISAC) is a precise, parameterized, low-level language capable of coding ISA programs in $O(\log n)$ space, optimally satisfying the motivations of Section 3.3. In ISA Subroutining, compression is first applied *horizontally* (ie. compress diagonals first) and then *vertically* (ie. iterations of diagonals). In ISAC, compression can also be first applied vertically (ie. compress iterations

[8]See Section 5.3.2.2.

for instructions for particular ISA columns) and then horizontally (ie. generalize over all columns), giving a more flexible approach. ISAC is designed so that an ISAC program can be easily loaded, in a systolic fashion, into the instruction (selector) matrix restorers. For the sake of simplicity, an $n \times n$ ISA model is assumed here; however, the concepts presented are easily extended to general ISAs.

3.7.1 Examples of ISAC coding

The constructs of ISAC are introduced by some simple examples.

The LoadMat program (see Figure 3.10) loads an $n \times n$ row-major matrix from the western data buffer into the C (communication) register of an $n \times n$ ISA. The instructions for this program consist of a *descending triangle* (initial height n) of '$\rightarrow$' instructions and an *ascending triangle* (initial height 0) of '0' instructions. The selectors for this program consist of a *repetition* (height n) of diagonals consisting entirely of 1's. The ISAC encoding of the program is:

LoadMat:
$(\rightarrow)^{\triangleright \cdot n}\ (0)^{\triangleleft \cdot 0}$
$(1)^{n}$

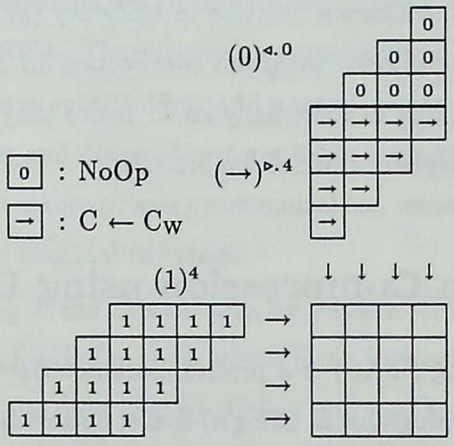


Figure 3.10: LoadMat program for a 4×4 ISA

For a microprogrammed ISA, the TransClos program (see Figure 3.6) is coded in ISAC as:

TransClos:

$(I)^{\triangleleft.0}\ I'\ (I)^{\triangleright.n-1}$

$(S)^{\triangleleft.0}\ S''\ (S)^{\triangleright.n-1}$

Here, the instruction part consists of an *ascending triangle* (initial height 0) of I macros, followed by a 'diagonal'[9] of I'' macros, followed by a *descending triangle* (initial height $n-1$) of I macros (and similarly for the selector part).

The RotHV(d) program of Section 3.6 is coded in ISAC using a *repetition* construct (of height d):

RotHV(d):

$([0\ _1|\rightarrow\ _n]\ \ [\leftarrow\ _{n-1}|0\ _n]\ \ \downarrow\ \ \uparrow\ \ )^d$

$(1\ \ \ \ \ \ \ \ \ \ 1\ \ \ \ \ \ \ \ [0\ _1|1\ _n]\ \ [1\ _{n-1}|0\ _n]\ \ )^d$

The diagonals are coded in an analagous way to the Subroutining method using the ISAC *choice* construct, eg. the first diagonal '$[0\ _1|\rightarrow\ _n]$' (meeting with a 1 selector) is equivalent to the Subroutining expression $< 0(\rightarrow)^{n-1},\ 1^n >$'.

RowRev (see Figure 3.11) is an ISA program using a *divide and conquer* strategy[10] for reversing the rows of a matrix stored in the C registers of an $n \times n$ ISA, where n is a power of 2. The program swaps adjacent blocks of width 2^{k-1} by using 2^{k-1} horizontal ring shifts over distances of $d = 2^k$, where $1 \leq k < \log n$. The program is based on a horizontal ring-shift over consecutive blocks of width $d/2$ (cf. program Rot1H of Section 3.6), coded in ISAC as:

RotH(d):

$[0\ _1|\rightarrow\ _d]\ \ [\leftarrow\ _{d-1}|0\ _d]$

$1\ \ \ \ \ \ \ \ \ \ 1$

Here, the *choice* construct '$[0\ _1|\rightarrow\ _d]$' means a '0' instruction appears in column 1, and a '$\rightarrow$' instruction appears in columns 2 to d; this pattern is then repeated across the remaining columns. Similarly, a whole diagonal of 1 selectors could be expressed as '$[1\ _n|\ _n]$', but this is abbreviated simply to '1'. The whole algorithm is expressed as:

RowRev:

$((\text{RotH}(2^k))^{2^{k-1}})^{k=1\ldots\log n}$

which introduces the *parameterized repetition* construct, '$(\ldots)^{k=l\ldots u}$', which generalizes the *repetition* construct.

---

[9] In ISAC, a 'diagonal' can have arbitrary skew, which gives ISAC extra flexibility.

[10] cf. ISA algorithms for matrix transposition [31] and sorting [46].

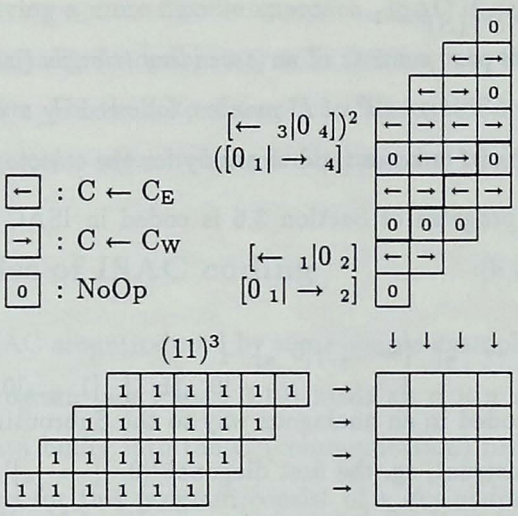


Figure 3.11: RowRev program for a 4×4 ISA

3.7.2 Definition of ISAC

Comparing ISAC to sequential programming languages, the *repetition* construct corresponds to *for* loops (with constant loop bounds), whereas the *choice* and *triangle* constructs correspond to *if* statements and *for* loops respectively, whose guards and loop bounds depend in a simple manner on the column (row) number. These constructs are now defined more formally. Let k, d, l and u denote non-negative integers. Let α, β and $\alpha(k)$ denote the instruction (selector) part of an ISAC encoding of an ISA program.

The *parameterized repetition* construct provides vertical compression, and is defined by:

$$(\alpha(k))^{k=l..u} = \begin{cases} \alpha(l)(\alpha(k))^{k=l+1..u} & \text{if } l \leq u \\ \epsilon & \text{otherwise} \end{cases} \tag{3.1}$$

where ϵ is the empty string. In this case, it may be assumed that some preprocessing mechanism evaluates the arithmetic expressions in $\alpha(k)$, for each value of k. The ordinary *repetition* construct can be defined similarly.

The *choice* and *triangle* constructs allow different streams of instructions (selectors) to be seen at each column (row) of an ISA. The former provides horizontal compression, whereas the latter provides compression in both directions. They

can be defined by introducing the concept of the *projection* of α on the jth column (row) of an $n \times n$ ISA, ie. the instruction (selector) sequence that is produced at column (row) j by α, where $1 \leq j \leq n$. This projection, denoted $p_j(\alpha)$, is defined by:

$$\begin{aligned}
p_j(\alpha) &= \alpha \quad \text{if } \alpha \text{ contains no } [\ldots], (\ldots)^{\triangleleft.l} \text{ or } (\ldots)^{\triangleright.u} \text{ constructs} \\
p_j(\alpha\,\beta) &= p_j(\alpha)\; p_j(\beta) \\
p_j((\alpha)^{\triangleright.u}) &= (p_j(\alpha))^{u-j+1} \\
p_j((\alpha)^{\triangleleft.l}) &= (p_j(\alpha))^{l+j-1} \\
p_j([\alpha_{\ k}|\beta_{\ d}]) &= \begin{cases} p_{j'}(\alpha) & \text{if } j' \leq k, \text{ where } j' = (j-1) \bmod d + 1 \\ p_{j'-k}(\beta) & \text{otherwise} \end{cases}
\end{aligned} \tag{3.2}$$

Note that $p_1(\alpha)^{\triangleleft.0}$ and $p_n(\alpha)^{\triangleright.n-1}$ produce empty sequences. Note also that the *choice* construct distributes in a fairly straightforward way over all other constructs, a property useful in optimizing compression rates.

Extensions of ISAC might also be required to increase its generality. For example, the ISA Perfect Shuffle program [51, p286] requires 'half-triangle' constructs (with the triangle height changing every second column) to be introduced. Also *vertical interleaving* may be a useful extension to ISAC (see Section 7.4).

Example: the instruction sequence for column $j = 2$ of the LoadMat program for $n = 4$ (see Figure 3.10) is given by:

$$p_2((\rightarrow)^{\triangleright.4}\ (0)^{\triangleleft.0}) = ((\rightarrow)^{4-2+1}\ (0)^{0+2-1})$$

Similarly, the instruction sequence for column $j = 3$ of the RowRev program is: for $n = 4$ (see Figure 3.11) is given by:

$$p_3([0_{\ 1}|\rightarrow_{\ 2}]\ [\leftarrow_{\ 1}|0_{\ 2}]\ ([0_{\ 1}|\rightarrow_{\ 4}]\ [\leftarrow_{\ 3}|0_{\ 4}])^2) = (0\ \leftarrow\ (\rightarrow\ \leftarrow)^2)$$

Given α, the ISAC representation of the instructions (selectors) of an ISA program of period t, the corresponding $t \times n$ instruction (selector) matrix I is recovered using the *projections*:

$$I_{ij} = (p_j(\alpha))_i \quad \text{for } 1 \leq i \leq t \text{ and } 1 \leq j \leq n \tag{3.3}$$

where $(\ldots)_i$ means selecting the ith element of the enclosed sequence. Similarly, a general transformation from I gives an equivalent (space-inefficient) ISAC encoding:

$$
\begin{array}{l}
[I_{11}\ \ 1|\ [I_{12}\ \ 1|\ \cdots\ I_{1n}\ \ 2]\ \cdots\ n] \\
[I_{21}\ \ 1|\ [I_{22}\ \ 1|\ \cdots\ I_{2n}\ \ 2]\ \cdots\ n] \\
\vdots \qquad\ \vdots \qquad\qquad\ \vdots \\
[I_{t1}\ \ 1|\ [I_{t2}\ \ 1|\ \cdots\ I_{tn}\ \ 2]\ \cdots\ n]
\end{array}
$$

3.7.3 Matrix restorers for ISAC

This section describes the hardware required to restore the matrix form for the instruction part of an ISAC encoding; similar hardware is required for the selector part. The cell PCUs of PAC (see Section 7.2) and the *diagonal sequencer* of the Subroutining method (see Section 5.3.1) have similar design features.

The parameterized repetition construct can be handled by special matrix restorer mechanisms as described in Appendix 3.A.3[11].

An ISAC encoding is loaded by passing it systolically across the linear array of cells comprising the matrix restorer. The jth cell in the matrix restorer interprets this encoding according to equations (3.2), and stores the jth projection of this encoding in an $O(\log n)$ table. Since $O(\log n)$ loading times are permissible, the most expensive operations required in this process are (bitwise) counting, table accessing and stack operations. The choice constructs are loaded in a similar way to that of the Subroutining diagonals (see Section 3.6). Repetition constructs are loaded according to the tabular encoding described below. Triangle constructs are loaded similarly, except that systolic (bitwise) counters can be used to give their position-dependent heights (see Section 7.2.3).

The tables in the matrix restorer's cells store the corresponding projection compactly in $O(1)$ length tables, utilizing *loops* projected from the repetition or triangle constructs in the original ISAC encoding. The design presented here accommodates the nesting of these loops of w levels, where w is less than the table length. Loops are required to be of the form $(Ei)^k$, where E is a sequence of instructions possibly with loops, i is an instruction and $1 \leq k \leq n$

[11] A simple solution uses a 'preprocessor', adjoining the matrix restorer, to 'unfold' any parameterized repetition constructs, according to equation (3.1) before loading the program into the matrix restorer. In practice, the 'unfolded' encoding has size $O(\log^2 n)$, requiring the matrix restorer cell's to have storage of the same order.

(note that ISA programs normally do not require $k > n$). The encoding of such loops is straightforward, and is illustrated in Figure 3.12. Note that double left parentheses are encoded by replacing the inner left parenthesis with a '$*$' in the table entry for the matching right parenthesis.

(	d	$)^4$
	c	$)^2$
$*$	b	$)^3$
(	a	

Figure 3.12: Tabular encoding of '$((a\ b)^3\ c)^2\ (d)^4$'

While loading tables for this encoding, if a loop's '$)^k$' entry is encountered, for $k \leq 0$, the body of the loop must be deleted (ie. by resuming the loading of the rest of the encoding from the location of the loop's '(' entry — this requires maintaining a w-element stack of table pointers). If the loop's '$)^k$' entry is also marked '$*$', the beginning of the rest of the encoding 'inherits' the loop's unmatched '('.

An algorithm effectively performing equation (3.3) for cell j of the matrix restorer is now described. This algorithm resembles the action of a standard microprocessor traversing its program memory (containing loops), except that it must ensure an ISA instruction is supplied every cycle (even if loop bookeeping is also performed during that step). This is essential in maintaining proper synchronization.

A pointer x traverses the table in such a manner as to produce the required instruction sequence (ie. '*abababcababcdddd*' for the table of Figure 3.12). Consider the algorithm traversing the loop $(E\ i)^k$. On the first access of the loop's '(' entry, x is pushed onto the left parenthesis position stack LPs. On the first access of the loop's '$)^k$' entry, if $k = 1$, LPs is popped and x is incremented. Otherwise $k > 1$ and k is pushed onto the counter stack CTs and x is set to top(LPs). On all the $k - 2$ intermediate accesses of the '$)^k$' entry, top(CTs) is decremented and x is set to top(LPs). On the last access, ie. top(CTs) $= 1$, CTs is popped and x is incremented. If this entry is not marked '$*$', the LPs stack is also popped.

If the loop is of the form $(i)^k$ (taking a single table entry), the situation is similar except that the LPs stack is not used, with x being left unchanged where it would have been set to top(LPs). If this entry is also marked '$*$', x must however be pushed onto LPs on its first access.

The first access of a loop's '(' entry is determined simply by the boolean flag FT, which is **true** iff x was incremented on the last cycle. The first access of a loop's '$)^k$' entry is determined by the boolean stack FTs, which records the value of FT during the last access of the loop's '(' entry.

A more formal description of this algorithm is given in Appendix 3.A.1. This method of encoding loops appears inefficient since the third ('$)^k$') field of the table will usually be blank. However, variable length table entries can be easily implemented to reduce this loss, as described in Section 3.8.2. This method of encoding appears not to be sufficiently general since loops of the form '$(E(E')^{k'})^k$' are not handled. In practice, these are sufficiently rare that recoding ISA programs to avoid them is a viable option. Extensions of the above design to handle these loops are given in Appendix 3.A.2.

Our experience suggests that 32 entries is sufficient for most 'irreducible' $\Omega(n)$ ISA sub-programs (see Table 3.2).

It might be desirable to simplify and make more flexible the implementation of ISAC proposed here. One approach for doing this would be to implement ISAC more like that of program memories in standard microprocessor design. Here, one table access per loop iteration (ie. one table entry per loop) would be devoted to loop bookeeping and would not fetch a new ISA instruction. In this way, general nested loops, subroutine calls and possibly even loops with zero iterations could be easily handled. However, to supply each column of the ISA with a new instruction every ISA cycle, the corresponding table accessing algorithm must run at two steps per ISA cycle, and a FIFO queue, able to buffer $O(\omega)$ instructions, must be inserted between the ISA and the matrix restorers. Two factors discouraged the development of this approach here: Firstly, accessing a table twice per ISA cycle is deemed unfeasible for ISAs with a very fast clock cycle, such as the boolean ISA. Secondly, considerable analysis would be required

to determine whether queue underflow[12] could be produced by a particular ISA program (queue underflow would for example be produced by the nested loop '$((i)^2)^n$').

3.8 Evaluation of program compression methods

In this chapter, four program compression methods have been proposed. For practical purposes, however, criteria are required to compare these methods in a realistic ISA system. These criteria are evaluated over a variety of 'irreducible' $\Omega(n)$ ISA sub-programs, taken from this chapter and the literature, which we believe are representative of 'irreducible' $\Omega(n)$ ISA programs in practice.

The choices for these criteria are the compression rates and the number of bits of static storage required the matrix restorer cell's tables (indicating the area overhead required in implementing a method)[13].

For comparing the methods, the ISA system chosen has dimensions $2^5 \times 2^5$ (expandable up to $2^8 \times 2^8$) and uses 8-bit instructions and data. While these dimensions are rather arbitrary, the results presented below, when combined with the asymptotic behaviour of the program compression methods, can give the reader a feeling for their relative performance.

3.8.1 Compression rates

Table 3.1 gives the program compression methods' reduction of overall I/O bandwidth. The numbers are given to two significant figures, since the exact numbers are dependent on implementation details.

[12]Queue underflow results in loss of synchronization inside the matrix restorers and in the ISA receiving the wrong program.

[13]A third criterion, an estimate of the period $O(\log n)$ required to load the matrix restorer for these ISA sub-programs is also relevant, for moderate values of n (to compare ISAC and Subroutining). However, this requires a more detailed description of the implementation given in this chapter.

For microprogramming, $\mu = 1$ except for the TransClos program (where $\mu = 5$), and for each program, a 'natural' value of the microcode length λ is chosen. The macro width w_m is set at the smallest possible length for each program, in order to optimize the compression rate.

For Subroutining and ISAC, a variable length symbol method is used to code the programs in their symbolic form. Here, a 4-bit tag is used to identify the type of symbol, eg. instruction, left/right parenthesis, etc. 8 bits are used for instructions (1 bit is used for selectors) and 8 bits are used for any counters. Thus, this coding is adequate for $n \leq 2^8$. An $O(\log n)$ area preprocessor can translate these code sequences into matrix restorer loading instructions.

As indicated by the fifth column of Table 3.1, most of the I/O of the ISA system is due to the programs, with many programs operating on matrices resident in the ISA (which were originally loaded in by programs such as LoadMat). Thus, the overall I/O traffic of the ISA system can be significantly reduced by these methods, particularly by the 'optimal' ones. Note that for increasing n to 2^8, the 'optimal' methods would improve over the others by a factor of ≈ 64.

3.8.2 Hardware overheads

Table 3.2 gives the program compression methods' overheads in hardware, in terms of the number of static memory bits required for each program in each instruction matrix restorer cell's tables. This is given for the ISA programs of Table 3.1. It gives an indication of the overall area required by each method. The other factor for hardware overhead, the complexity of the cell's control logic, is simplest for microprogramming, and most complex for ISAC. Note that the SISA has no hardware overhead.

For microprogramming, the number of bits for each decode table is given by the number of macros used ($\leq 2^{w_m}$) times λ times w_i.

For Subroutining, the simple implementation outlined in Section 3.6 is used, with one diagonal restorer entry (a 2-bit tag, used for shifting, and a w_i-bit instruction) for each different diagonal. For the divide-and-conquer programs used here, one extra instruction plus eight extra control bits per diagonal are

program	t	λ	w_m	$\frac{V_d}{V_p+V_d}$	V_p	V_p for method			
						SISA	micro.	Subr.	ISAC
RotHV($n/2$)	$2n$	4	2	0	18000	4600	2100	190	230
TransClos ◇	$5n$	5	1	2/48	46000	*'' 14000	2100	730	310
LoadMat	n	1	1	8/9	9300	2300	2100	*'' 120	100
RowRev †	$\approx 2n$	2	2	0	18000	4600	2100	390	410
Transpose †[31]	$\approx 6n$	2	3	0	55000	14000	13000	390	*' 680
Matrix Mult.[31]	$5n$	5	1	16/45	46000	8200	2100	100	130
Perfect Shuffle ‡[51]	n	2	2	0	9200	2200	4100	* 200	310

notes:

t is the program period, V_p (V_d) is program (data) I/O volume in bits, λ is the chosen microcode length, and w_m is the minimum macro width.

◇ The input data is assumed to be 1-bit here.

* The program had to be recoded for this method. '*'' ('*''') signifying the period is longer by a small constant amount (factor).

† divide-and-conquer programs, typically having a lower compression rates for Subroutining and ISAC.

‡ To simplify the coding, an extra compare-exchange row was inserted to increase the period from $n-2$ to n.

Table 3.1: Program compression ratios for ISA with $n = 8, w_i = 8$

required (see Section 5.4.1.2). This has considerable scope for optimization, as indicated in Section 5.3.3.

For ISAC, the table entries (using the scheme of Section 3.7.3) are assumed to be 24 bits each, with 4-bit tag, w_i-bit instruction and 12-bit counter fields. However, where the counter field is not used for two consecutive odd-even entries, the second entry is packed into the first's counter fields. This reduces the overall table size. For divide-and-conquer programs (see Appendix 3.A.3), the situation is similar except that the entries are 30 bits wide.

program	# static bits per cell for:		
	microprogramming	Subroutining	ISAC
RotHV($n/2$)	$3 \cdot 4w_i = 96$	$4 \cdot 10 = 40$	$3 \cdot 24 = 72$
TransClos	$2 \cdot 5w_i = 80$	$5 \cdot 10 = 50$	$9 \cdot 24 = 216$
LoadMat	$2 \cdot 1w_i = 16$	$2 \cdot 10 = 20$	$2 \cdot 24 = 48$
RowRev	$3 \cdot 2w_i = 48$	$2 \cdot 26 = 52$	$5 \cdot 30 = 150$
Transpose	$6 \cdot 2w_i = 96$	$6 \cdot 26 = 156$	$18 \cdot 30 = 540$
Matrix Mult.	$1 \cdot 5w_i = 40$	$5 \cdot 10 = 50$	$3 \cdot 24 = 72$
Perfect Shuffle	$4 \cdot 2w_i = 64$	$2 \cdot 10 = 20$	$6 \cdot 24 = 144$

Table 3.2: Hardware overheads for program compression methods (in terms of static storage bits per diagonal restorer cell) for a $2^5 \times 2^5$ ISA with $w_i = 8$

3.9 Application to other mesh architectures

For SIMD meshes, the SISA concept already exists in the row/column vectors of the ICL DAP. Microprogramming-like techniques can also be applied on the SIMD instruction stream, as is done for the Connection Machine. Subroutining and ISAC can be applied to compress the SIMD row/column vectors; however, in this case, all elements of a vector must be produced simultaneously (cf. the elements of ISA diagonals which are produced in a time-skewed fashion).

eg. On an $m \times n$ SIMD mesh, the program for unskewed matrix input from the west[14] could be expressed in ISAC-like notation as '$(1)^{\triangleright \cdot n}(0)^{\triangleleft \cdot 0}$' for the column vector program and as '$(1)^m$' for the row vector program. It can be expressed in Subroutining-like notation as:

[14] cf. the 'skewed' LoadMat program of Section 3.7.1.

for k:=1 **to** n **do**

$<\leftarrow,\ (1)^m,\ (1)^k(0)^{n-k}>$

and here, the column vector for the $(k+1)$th iteration is produced by performing a simultaneous (rather than a time-skewed, as in ISA Subroutining) right shift from that of the kth iteration. Since the shifting is simultaneous for SIMD meshes, the loss of flexibility of ISA Subroutining due to 'upstream' shifting would not carry over.

However, the loading of ISAC-like choice and triangle constructs loses efficiency if it is not done in a time-skewed (systolic) fashion (this also applies to loading of row/columns vectors if Subroutining is applied). In this case, the loading of the vectors can occur in a time-skewed fashion n time steps earlier, requiring the overlapping of program loading and execution, and hence extra buffering by the matrix restorer tables.

MIMD meshes have program memories in each of their cells and are required to be more powerful than the ISA model; hence only the most flexible method, ISAC, is expected to have application here. A two-dimensional extension of ISAC can be applied to efficiently load the cell's program memories, and to reduce their size.

3.10 Conclusion

A wide range of algorithms has been designed to be implemented on the ISA. Several ISA processors have been designed with different instruction sets. Prototypes of VLSI implementations of ISAs are under development. However, the peripheral hardware requirements (eg. the program interface) for ISAs are not settled yet. Furthermore, higher-level language concepts have to be developed in order to make the ISA concept acceptable for a wider range of users.

This chapter addresses, from a program compression viewpoint, both of these viewpoints. Part of a 'minimal' ISA high-level language is introduced in Section 3.6, and any program compression methods used must be compatible with the high-level ISA language used. Furthermore, microprogramming has potential in

abstracting from the granularity of ISA's instruction set, developing a complementary requirement for higher-level ISA programming. Some of the peripheral hardware requirements for ISAs are outlined in Section 3.3, and elaborated in Sections 3.6 and 3.7.

This chapter has demonstrated that (in particular 'optimal') program compression techniques can substantially reduce the overall I/O traffic[15] and the peripheral hardware requirements of the ISA (Sections 3.3 and 3.8.1). The latter should easily offset the hardware overhead ($O(n \log n)$ area matrix restorers) of program compression methods. This occurs even for moderate ISA sizes and when the ISA cell's instruction and data word lengths are of similar sizes.

The program compression method(s) to be chosen for an ISA system depends on its intended area of application. If program compression is critical to the overall system cost/performance, ie. if the ISA size is small or the data width is much larger than the instruction width, then the less expensive SISA or microprogramming methods are appropriate. Provided the SISA is sufficiently powerful to implement all of its intended applications, with only slightly less efficiency than the ISA, its simplicity and lack of hardware overhead makes it appropriate. Otherwise, microprogramming, with its enhanced flexibility and its generally superior compression rates, is appropriate.

If program compression is critical, an 'optimal' method (Subroutining or ISAC) should be chosen. The compression rates of either are sufficiently good that the difference between them is not significant in practice. While ISAC appears to be more flexible than Subroutining, Subroutining has less control structure overhead. Furthermore, Subroutining has an important practical advantage in that its control structures directly support a high-level ISA language. This facilitates easier automatic 'compilation' from high-level code into code to load and manipulate the matrix restorers.

For a given area of application, combining one of the 'optimal' and non-'optimal' methods might optimize compression rate, flexibility and hardware overhead. Of these, combining Subroutining with microprogramming (possibly also

[15] A corresponding pin count reduction also can be achieved, provided the matrix restorers can be packaged with the ISA itself.

with the SISA) is the most interesting. This combines high compression rates with high flexibility, with potential to enhance higher-level ISA programming, and yet still has fairly modest control structure overheads.

These methods have application in other mesh architectures. In particular, Subroutining can be applied to efficiently compress the row/column vectors for SIMD meshes, and ISAC can be applied to optimize program loading times and cell program memories for MIMD meshes. However, efficient program compression techniques mitigate the more powerful and scalable ISA's chief disadvantage over the SIMD mesh: its extra program input traffic. The other disadvantage, that instruction granularity is a crucial issue for the flexibility of the ISA, may be mitigated by microprogramming techniques. Hence, the results of this chapter make the SIMD mesh a considerably less interesting architecture than the ISA.

However, overall mesh performance may still be marred by I/O limitations due to loading data. Here, data compression techniques only have a limited applicability, and it may be necessary to pay the the high price of *recycling* and/or to develop costly parallel I/O transfers. How this can be achieved in detail is beyond the scope of this thesis.

This chapter has demonstrated how control structures (particularly for the ISA program interface) can substantially increase overall ISA performance while reducing overall ISA system costs. Further work includes investigating how the microprogrammed ISA can extend the power and flexibility of the ISA model (Chapter 4), and support a higher-level ISA programming model. Also, whether the loss of flexibility of Subroutining can be avoided in practice, and the details of its (optimized) implementation need to be examined (Chapter 5). Both of these program compression methods utilize the wavefront concept and show sufficient promise to require a formal semantic definition (Chapter 6). The application of ISAC-like techniques to MIMD meshes also has considerable potential (Chapter 7). It may be useful to simplify the implementation of ISAC, possibly along the lines of the suggestions at the end of Section 3.7.3, for this purpose.

3.A Appendix: Implementation of ISAC

3.A.1 ISAC table traversal algorithm

The algorithm for traversing the tabular representation of the ISAC projections, as outlined in Section 3.7.3, is here described informally but in more detail. This algorithm is performed independently in all ISAC matrix restorer cells. While it appears complex, it is expected that the algorithm has a reasonably efficient VLSI implementation using PLAs.

Let x be the table pointer. The algorithm begins with x initialized to the table entry corresponding to the beginning of the current program, and terminates when it reaches the entry corresponding to that corresponding to the end of the program. At each step, the instruction field of the xth table entry is sent to the adjacent ISA column. The algorithm uses the integer stack CTs, the table pointer stack LPs and the boolean stack FTs, each having length w (the maximum allowed loop nesting) and the usual 'push', 'pop' and 'top' operations. These are initialized to empty stacks. The algorithm also uses the boolean variable FT, initialized to **true**.

A single step of the algorithm is now described in terms of cases for the different types of table entries. If the xth table entry is neither a '(' or '$)^k$' entry, the algorithm performs:

$x \leftarrow x + 1$; $FT \leftarrow 1$ {advance to next entry}

If the xth table entry is a '(' entry (only), the algorithm performs:

if FT **then** {starting 1st itn. of loop}
  push(x, LPs); push(FT, FTs) {record x for next itn. of loop}
$x \leftarrow x + 1$; $FT \leftarrow 1$ {advance to next entry}

If the xth table entry is a '$)^k$' entry (only), let T^* denote whether that entry is also marked '$*$'. In this case, the algorithm performs the more complex action:

if $(\text{top}(FTs) \wedge k > 1)$ **or** $(\overline{\text{top}(FTs)} \wedge \text{top}(CTs) > 1)$ **then**
  {ending not the last of $k > 1$ itns. of loop}

	if top(FTs) **then**	{ending 1st itn.}
	push($k-1, CTs$)	{record # itns. to go}
	else	
	top(CTs) $\leftarrow$ top(CTs) -1	{update # itns. to go}
	top(FTs) $\leftarrow 0$	{record 1st itn. over}
	$x \leftarrow$ top(LPs); $FT \leftarrow 0$	{return to corresp. '(' entry}
else		{ending last itn.}
	if $\overline{\text{top}(FTs)}$ **then**	{ie. $k > 1$ - restore CTs}
	pop(CTs)	
	if T^* **then**	{top(LPs) shared by enclosing loop}
	top(FTs) $\leftarrow 1$	{at 1st itn. of an enclosing loop}
	else	{discard top(LPs)}
	pop(FTs); pop(LPs)	
	$x \leftarrow x+1$; $FT \leftarrow 1$	{advance to next entry}

If the xth table entry is both a '(' and a '$)^k$' entry, x is pushed on LPs on the 1st iteration only if T^* is true. Otherwise, the action is as for the '$)^k$' entry except that FPs and LPs are not used, with x replacing top(LPs) and FT replacing top(FPs).

3.A.2 ISAC implementation of general loops

The tabular encoding of loops for the ISAC matrix restorer cells described in Section 3.7.3 forbids loops of the form $(E'(E)^k)^{k'}$. In ISAC encodings, such loops might occasionally be required.

A simple method of extending the tabular encoding to handle most of such loops occurring in practice is now described. These loops are restricted to the form "i' $(E'(E_1\ i)^k)^{k'}$", where i and i' represent simple instructions. The outer loop counter k' can be stored with the table entry for i', and is pushed onto the CTs stack when this entry is accessed. The '$i)^k$' entry is marked to signify a double right parenthesis; when the inner loop terminates, the CTs stack is popped. At this point, top(CTs) will give the remaining number of iterations for the outer loop, and the table pointer x is either incremented or set to the '('

entry of the outer loop accordingly.

A general but more expensive solution is to code loops of the form $(E'(E)^k)^{k'}$ as $(E'(E)^k\Box)^{k'}$, where '$\Box$' is a special instruction code that is never passed to the ISA. This extension requires the table to perform two steps per ISA instruction cycle, and sends all instructions except the '$\Box$' to a queue (initially empty) of length $2w$ (the table waits if the queue is full). The queue sends an instruction to the ISA every cycle, and can be shown to never run empty.

3.A.3 ISAC implementation of divide-and-conquer programs

An efficient ISAC implementation of divide-and-conquer programs is described here. These programs can be expressed using a parameterized repetition contruct[16] such as $(\alpha(k))^{k=c..\log n}$, where c is typically in the range 1 to 3. The k-dependent expressions in $\alpha(k)$ (being initial heights of *triangle* or *repetition* constructs, or boundaries in *choice* constructs) are of the forms $d \pm c'$ or $\frac{d}{2} \pm c'$ where d is a shorthand for 2^k.

This implementation is not general but is sufficient, with a little ingenuity, to cover currently existing ISA divide-and-conquer programs including matrix transposition [31], sorting based on row-wise/column-wise odd-even transposition sort [46], 'triangle' and 'diamond' merging [46, 51], and bubblesort [51]. To illustrate this implementation, the RotH(d) sub-program of Section 3.7.1:

$$\text{RotH}(d): \quad ([0\ {}_1| \rightarrow\ {}_d]\ \ [\leftarrow\ {}_{d-1}|0\ {}_d])^{d/2}$$

is used, where RowRev= $(\text{RotH}(2^k))^{k=1..\log n}$. An equivalent sub-program, used in ISA matrix transposition [31]:

$$\text{RotH}'(d): \quad \begin{matrix}[(0\ 0)^{\triangleright .\, d/2}(\leftarrow\ \rightarrow)^{\triangleleft .\, 1} & {}_{d/2}| \\ (\leftarrow\ \rightarrow)^{\triangleright .\, d/2}(0\ 0)^{\triangleleft .\, 1} & {}_{d}]\end{matrix}$$

is also used. In this case, RotH'(d) has been 'padded' with an extra diagonal before and after, ensuring that triangle constructs never have height 0. These extra diagonals meet 0 selectors, so that they have no semantic effect.

[16] Note that in practice, only one level of parameterized repetition is ever used.

The implementation is illustrated in Figure 3.13. It is an extension of the tabular encoding of Section 3.7.3, thus requiring no more than $O(\log n)$ area per matrix restorer cell. The parameterized repetition is implemented as a repetition of height $\log n$, whose kth repetition is interpreted according to the value of $d = 2^k$. The program is preceded with an extra no-operation ('0') instruction, which is used to set up the nested loop (according to the scheme proposed in Appendix 3.A.2) and also to select the appropriate sub-program for the first repetition. The basic idea is that, for the kth repetition in matrix restorer cell j, loop counters can be generated by masking their respective values for the last repetition with $\frac{d}{2}$ (and possibly adding $\frac{d}{2}$ as well), as shown in the fifth columns. The appropriate component of the choice $[P_{\ e}|Q_{\ d}]$ can be selected by pre-testing whether $j_d > e$, where $j_d = (j-1) \bmod d + 1$, as shown in the first columns.

$j_d > 1$	$*$	$\leftarrow$	$)^0)$	$\bmod \frac{d}{2} + \frac{d}{2}$	$j_d > 1$	$*$	0	$)^0)$	$\bmod \frac{d}{2} + \frac{d}{2}$
$j_d > d-1$	(	$\rightarrow$			$j_d > d-1$	(	0		
					$j_d > 1$		0	$\log n$	

(a) using '0 $(([0_{\ 1}| \rightarrow_{\ d}]\ \ [\leftarrow_{\ d-1}|0_{\ d}])^{d/2})^{k=1..\log n}$', $d = 2^k$

$j_d > \frac{d}{2}$	$*$	$\rightarrow$	$)^{n-j})$	$\bmod \frac{d}{2}$	$j_d > \frac{d}{2}$	$*$	0	$)^{n-j})$	$\bmod \frac{d}{2}$
$j_d > \frac{d}{2}$	(	$\leftarrow$			$j_d > \frac{d}{2}$	(	0		
$j_d > \frac{d}{2}$		0	$)^{n-j}$	$\bmod \frac{d}{2}$	$j_d > \frac{d}{2}$		$\rightarrow$	$)^{n-j}$	$\bmod \frac{d}{2}$
$j_d > \frac{d}{2}$	(	0			$j_d > \frac{d}{2}$	(	$\leftarrow$		
					$j_d > \frac{d}{2}$		0	$\log n$	

(b) using '0 $(\mathrm{RotH}'(d))^{k=1..\log n}$', $d = 2^k$

Figure 3.13: Tables in matrix restorer cell j for divide-and-conquer row reversal programs

The implementation of the divide-and-conquer *choice* constructs:

$$[P_{\ 1}|Q_{\ d}],\quad [P_{\ d/2}|Q_{\ d}],\quad [P_{\ d-1}|Q_{\ d}]$$

is now described. P (Q) is loaded into the odd (even) entries of the ISAC matrix restorer cell's tables (in Figure 3.13, consecutive even-odd entries are put side

by side for the sake of readability). On each table access, the boolean condition (first column of the table) of the current (xth) entry is evaluated, to a value C, say. The table pointer x is here updated to follow either the even (P) or odd (Q) table entries according to the formula:

$$x \leftarrow 2(v \operatorname{div} 2) + C$$

where v is the value x would have been given according to the algorithm of Appendix 3.A.1.

The mask values d and $\frac{d}{2}$ are updated, using simple shift operations, upon accessing (for the last time) the last even or odd entry of the parameterized repetition construct. In this case, the value of C would be computed from the updated mask values, in order to select either P or Q for the next repetition.

To implement in ISAC the divide-and-conquer sorting programs using odd-even transposition sorting, a four-way *choice* construct must to be introduced, which can be implemented in a similar way to the above two-way construct.

Consider the implementation of the repetition and triangle constructs. At matrix restorer cell j, these give rise to loop counters having values given by expressions of the forms $d' + c$, $d' - c - 1$, $(j \pm c) \bmod d'$ and $(d' - j \pm c) \bmod d'$, where d' is either $d = 2^k$ or $\frac{d}{2} = 2^{k-1}$ and $0 \leq c' \leq d'$. Hence, the counter values for $\alpha(k)$ can be generated from:

$$\begin{aligned} d' + c &= (c) \bmod d' + d' \\ d' - c - 1 &= (n - c - 1) \bmod d' \\ (j \pm c) \bmod d' &= (j \pm c) \bmod d' \\ (d' - j \pm c) \bmod d' &= (n - j \pm c) \bmod d' \end{aligned}$$

The values of the parenthesized expressions (on the RHS of the above equations), together with two bits to specify the appropriate operation (one bit to specify whether the mask d' is either d or $\frac{d}{2}$, and one bit to specify whether to add on d') are stored in the diagonal restorer tables. This is illustrated in the last columns of the tables of Figure 3.13. These counter values must be strictly positive, which may require 'padding' with extra diagonals or the introduction of four-way divide-and-conquer *choice* constructs.

Chapter 4

The Microprogrammed Instruction Systolic Array

4.1 Introduction

Microprogramming is a standard technique for simplifying microprocessor design and reducing the input bandwidth from the program memory to the CPU [55, Ch.10][17, Ch.3]. Microprogramming also provides a higher-level abstraction from a processor's microinstruction set and detailed hardware control, enabling it to interpret several instruction sets. Microprogramming is used in many digital signal processing chips suitable for array processing and in the Inmos Transputer [26, pp467-470]. Microprogramming is also used to "amplify" the bandwidth of the SIMD instruction steam of the Connection Machine [14, p88]. In large-scale non-SIMD meshes, program input bandwidth is an even more important factor.

In Section 3.5 (see also [53, 34]), the concept of the microprogrammed ISA (μISA), was introduced, mainly from the motivation of program compression (ie. reducing program input bandwidth). In this chapter, this *dynamic* form of *vertical* microprogramming for the ISA is further developed. This technique enables the efficient emulation of almost arbitrary *instruction granularity* ISAs, as well as efficient cell design and abstraction from a fixed (ISA) instruction set. This supports higher-level programming of the ISA model, and hence reduces programming effort. In terms of the wavefront model, it supports the implementation of

macro-level wavefronts. Because of the flow of control information in a wavefront-based mesh (eg. the ISA), the implementation is not trivial; on the other hand, it extends the mesh's range of efficiently implementable programs. This makes a unique and interesting application of microprogramming.

The μISA (usually) uses fixed-length microcodes, with no internal iterations and no conditional mechanisms except the ISA selectors[1]. This leads to an area-efficient implementation suitable even for fine-grained ISAs.

Section 4.1.1 briefly describes the μISA model used for this chapter. A variant useful for emulating arbitrary instruction granularity ISAs, called the *normal* μISA, is introduced there. Section 4.1.2 discusses how microprogramming increases the ISA's advantage in power over SIMD meshes (by enhancing the ISA's handling of nontransmittent data), while mitigating the ISA's disadvantage in requiring more programming effort (due to the fact that instruction granularity is critical for the ISA). Section 4.1.3 elaborates the motivations for the μISA.

In Section 4.2, the μISA model is illustrated by the LCS algorithm (representing the class of Dynamic Time Warping algorithms) and the transitive closure algorithm (representing the class of algebraic path algorithms). An implementation of microprogramming, using simple and area-efficient control structures, is given in Section 4.3. For the sake of efficiency, these control structures must also be combined with a macro discipline to properly implement macro-level wavefronts.

In Section 4.4, the μISA model is viewed as a generalization of the ISA model, and its extra control structures are demonstrated to make it more powerful than the ISA and SIMD mesh models, in both a theoretical and practical sense. Section 4.5 illustrates the utility of the normal μISA model for the Givens Rotations algorithm (representative of the class of matrix QR factorization algorithms). General transformations which enable a normal μISA to emulate an ISA of (almost) arbitrary instruction granularity is given in Section 4.6. This enables the μISA to realize its chief application: an extension of the ISA model, increasing both its power and flexibility. Conclusions are given in Section 4.7.

[1]However, the μISA micros (eg. a signed addition instruction) might themselves be implemented by a second level of microprogramming.

The contribution of this chapter is to introduce and develop the theoretical and practical aspects of the microprogrammed ISA, which can be seen both as a generalization and an extension of the ISA concept.

For reading the remainder of this thesis, an understanding of Sections 4.1.1, 4.2 and 4.3 is important. A semantics for the microprogrammed ISA is given in Chapter 6. A generalization of the basic concepts of the microprogramming is used to describe control structures and program compression techniques for Processor Arrays in Chapter 7, especially Section 7.4.1.

4.1.1 The microprogrammed ISA model

The model for an $m \times n$ μISA is now briefly described. It is a refinement of the model proposed in [34], in which the concept of an ISA macro is more usefully defined.

The concept of the ISA diagonal can be generalized to that of the *wavefront* for the μISA, due to the introduction of the *microprogramming parameters* (μ, σ). A (μ, σ) wavefront propagates at an angle of $\tan^{-}1(\sigma/\mu)$ with respect to the north edge of the ISA, with a velocity of $1/\mu$ $(1/\sigma)$ in the south (east) direction. With this concept, a SIMD mesh uses a $(0,0)$ wavefront; an ISA uses a $(1,1)$ wavefront; a [(μ, σ) wavefront] μISA uses a (μ, σ) wavefront with $\mu, \sigma > 0$ and a Processor Array (MIMD mesh) program can use various combinations of arbitrary wavefronts (see Chapter 7).

A (μ, σ) wavefront μISA is also parameterized by the *microcode length* λ (see Figure 4.1). Thus, a *microcode* is a sequence of $\lambda = \Theta(1)$ w_i-bit (*micro*) instructions, and is represented by a single w_m-bit, where $w_m << \lambda w_i$ *macro*(instruction)[2]. The instruction microcodes themselves are skewed between neighbouring columns by an offset of σ, $1 \leq \sigma \leq \lambda$, and the corresponding selector microcodes are skewed by a factor of μ, $1 \leq \mu \leq \lambda$. Using λ individual selectors bits (*selector micros*) per macro has the added power that each row of an μISA can omit arbitrary micros within a macro.

A fourth parameter, used for the emulation of arbitrary instruction granularity

[2]Our experience suggests $w_m = 4$ is sufficient for most applications (see Table 3.1).

ISAs by μISAs, is the *macro overlap factor*, $\bar{\eta}$. The quantity $(\bar{\eta}+1)$ corresponds to the communication register *timesharing* factor, ie.:

> the (maximum) number of times an output communication register is used to transfer separate words of data within a given macro.

and is, in the best case, equal to it. The microcode length is usually related to the selector skew by the relation:

$$\lambda = \mu + \bar{\eta} \tag{4.1}$$

although to increase program compression, λ may be made larger (cf. Section 3.5 and Table 3.1).

This concept of timesharing requires that the μISA's input communication registers read new values every microcycle. The alternative, to read new values on every macrocycle, while appearing to correspond more closely to traditional microprogramming, excludes timesharing and is more difficult to implement.

The *normal* μISA has the constraint that $\sigma = \mu$. This model, with $\bar{\eta} = 0$, was proposed in Section 3.5. This model is most similar to the ISA, which is in fact a normal μISA with $\mu = 1$, and is suitable for emulating ISAs of arbitrary *instruction granularity*. Occasionally, however (see Sections 4.2.1 and 4.5.6), the non-normal μISA is slightly more efficient.

This form of microprogramming is dynamic, in that the microcode lengths, contents and skews can be efficiently changed between $\Omega(n)$ sub-programs. Indeed, for the purpose of optimization, the microcode length may vary between consecutive macro diagonals (see Sections 4.5.5 and 4.6.2). In this case, care must be taken that correct inter-macro communication is preserved. Changing the microcode skews can incur a $\Theta(n)$ delay, and hence they should be changed less frequently.

4.1.2 ISAs vs. SIMD meshes and microprogramming

The main advantage of the ISA over the SIMD mesh is that it can pass a diagonal of nontransmittent data through the mesh at constant period. This occurs

when the nontransmittent data moves in pace with an ISA program diagonal (wavefront), which moves and updates the data.

> eg. The ISA can perform in constant period, from either rows or columns, operations such as broadcasting, ring-shifting, searching, summing and minimum finding. These operations cannot be efficiently performed on a SIMD mesh.

Thus, algorithms based on matrix transitive closure or QR factorizations can be implemented in an $\Theta(n)$ period on an ISA, but appear to require a $\Theta(n^2)$ period on a SIMD mesh. Thus, the ISA is more powerful than the SIMD mesh, and more flexible than (non-programmable) systolic arrays.

However, to handle nontransmittent data in this way requires that its communication and computation must be completed in a single instruction cycle. This in turn makes the ISA's *instruction granularity* a critical quantity. This has two consequences: Firstly, an ISA's (fixed) instruction set may still be inadequate to implement systolic algorithms making heavy use of nontransmittent data. Sometimes, the efficient implementation of such algorithms requires extremely powerful and specialized instructions, ie. performing several arithmetic operations and communicating several words of data per cycle.

Secondly, unlike SIMD meshes (or uniprocessors), it is impossible to abstract from instruction granularity by simply composing sequences of instructions. This make the ISA more difficult to program, particularly when using high-level 'macro' constructs, than the SIMD mesh.

The microprogramming parameters (μ, σ) introduced in this chapter provide a way of effectively constructing high-level macros, enabling the μISA to efficiently emulate various (macro) instruction sets (with varying instruction granularities). With this abstraction of the ISA model, its handling of *nontransmittent* data becomes much more powerful, and its merit, as compared with the SIMD mesh model, is significantly enhanced.

4.1.3 Motivations for microprogramming the ISA

Microprogramming can enable the design of an efficient ISA suitable for a large application domain, in which a very large range of (powerful) instructions may be required. This is important for the development of the ISA for general-purpose matrix processing. Factors motivating dynamically microprogrammable ISAs include:

1. A simple instruction set is attractive for the ISA cell designer. This is because, I/O limitations permitting, it is useful to keep the ISA cells as small as possible. The advantages of doing this are twofold: a faster clock cycle and greater degree of parallelism (more cells can be fitted on a chip) are possible. Microprogramming permits the ISA cell designer to design a simple (micro)instruction set; then, for the current algorithm to be run on the ISA, the desired instruction set, with sufficient *instruction granularity*, can be constructed by choosing a suitable value of (μ, σ) and composing the appropriate macros.

2. The communication aspect of the high *instruction granularity* achievable by the μISA enables the *timesharing* of output communication registers (up to a factor of $\bar{\eta} + 1$). This in turn allows different kinds of meshes to be efficiently emulated using a basic mesh. For example, for the LCS algorithm of Section 4.2.1, an orthogonally-connected μISA uses microprogramming to emulate, on the macro level, a hex-connected ISA.

3. Microprogramming can achieve a program compression rate (and corresponding pin reduction) of $\lambda w_i / w_m$ (for the instruction part of the program) [53]. Motivations for program compression include reduction of the bandwidth[3] from the external ISA program memory to the ISA, and reduction of the size of the ISA external program memory (see Section 3.3).

[3]ISA program matrices are typically of $\Omega(n^2)$ size.

The importance of each of these factors depends both on what class of algorithms is required for the ISA and on the technology used.

The simple model for the μISA presented here provides the benefits of conventional microprogramming, as well as increasing the range and efficiency of algorithms using nontransmittent data that can be implemented using a fixed ISA (micro)instruction set. It does this in a such a way that the more frequently used low-instruction granularity ISA programs can be run at maximum efficiency (in terms of both area and time).

4.2 Examples of microprogramming the ISA

In Section 3.5.2, the Kung-Gubias-Thompson transitive-closure algorithm was implemented on a normal μISA using a four output communication register mode with $\lambda = \mu = 5$ (ie. no timesharing). In this section, μISA programs are given illustrating communication register *timesharing* to a factor of $\bar{\eta} + 1 = 2$. The LCS program (Section 4.2.1) illustrates the use of a non-normal μISA (in this case being slightly more efficient than the normal μISA). The transitive closure algorithm is re-implemented in Section 4.2.2 on a normal μISA using a single output communication register, with $\lambda = \mu + 1 = 6$.

4.2.1 The LCS algorithm

The Largest Common Substring (LCS) algorithm [47, 38] is an inexact string matching algorithm used to compare protein sequences in molecular biology. Given a small alphabet Σ, and two strings $a, b \in \Sigma^*$, a is defined to be similar to b iff $|\mathrm{LCS}(a,b)| > 78\%|a|$. Defining $m = |a|$ and $n = |b|$, $|\mathrm{LCS}(a,b)|$ is given by $M'_{m,n}$ using the following set of equations ($0 \leq i \leq m, 0 \leq j \leq n$):

$$\begin{aligned} M_{i,j} &= (a_i = b_j) \quad \text{if } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \\ M'_{i,j} &= \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{M_{i,j} * M'_{i-1,j-1}, M'_{i-1,j}, M'_{i,j-1}\} & \text{otherwise} \end{cases} \end{aligned} \tag{4.2}$$

where $a * b = a(b+1)$. This algorithm can be implemented in $O(1)$ period on an $m \times n$ ISA, with the matrix M' being the nontransmittent data. The implementation of [47] assumes each cell of the ISA has *two* output registers, denoted here

A and C. Its initial conditions are that the ith row of the ISA's west data buffer stores $M'_{i,0}$, the jth column of the north data buffer stores '$M'_{0,j-1}; M'_{0,j}$' and the A register of cell (i,j) stores $M_{i,j}$. The implementation consists of a uniform diagonal, selected on all rows, consisting of the instruction:

$$(\mathrm{C}, \mathrm{A}) \leftarrow (\max(\mathrm{A} * \mathrm{A_N}, \mathrm{C_N}, \mathrm{C_W}), \mathrm{C_W})$$

which gives an efficient implementation of the LCS algorithm, at the cost of an ISA with unusual communication capabilities and specialized, complex instructions.

This algorithm can be implemented on a non-normal $m \times n$ μISA, with $\mu = 3$ and $\sigma = 2$, with a simple microinstruction set, a single communication register C and an internal register A. This implementation has the same initial conditions, but uses a single communication register, which must be timeshared to a factor of $\bar{\eta}+1 = 2$ (achieved by 'overlapping' macros by $\bar{\eta}$ microcycles). The corresponding program is illustrated in Figure 4.1.

To visualize the execution of this program, the instructions (selectors) are moved down (right) one small unit per microcycle; the topmost (leftmost) instruction (selector) micro in each cell being executed at that instant. One may visualize that each ISA cell here has queues of length 3 (2) to slow the rate of the instruction (selector) micros passing through the ISA (cf. Section 4.3). Here, the value of $M'_{i-1,j-1}$ is shifted to cell $(i-1,j)$ by execution of the micro '$\mathrm{C} \leftarrow \mathrm{C_W}$', in time to be read by cell (i,j) as it executes the '$\mathrm{A} \leftarrow \mathrm{A} * \mathrm{C_N}$' micro. It may be thought that the '$\mathrm{A} \leftarrow \mathrm{A} * \mathrm{C_N}$' micro is undesirably specialized. However, it is often useful to implement ISA instructions that can be executed conditionally on the value of a register, say A, being non-zero (see Section 2.2.1). With initially $\mathrm{A} = M_{i,j} \in \{0,1\}$, the BISA-like instruction '$\mathrm{A} \leftarrow \mathrm{A} + \mathrm{C_N}$, if A' would implement the micro '$\mathrm{A} \leftarrow \mathrm{A} * \mathrm{C_N}$'.

This program can be easily converted for the four output communication register mode normal μISA (see Section 4.6.2). This increases the microinstruction skew to the 'normal' value of $\sigma = 3$, and correspondingly increases the macro overlap to $\bar{\eta} = 2$.

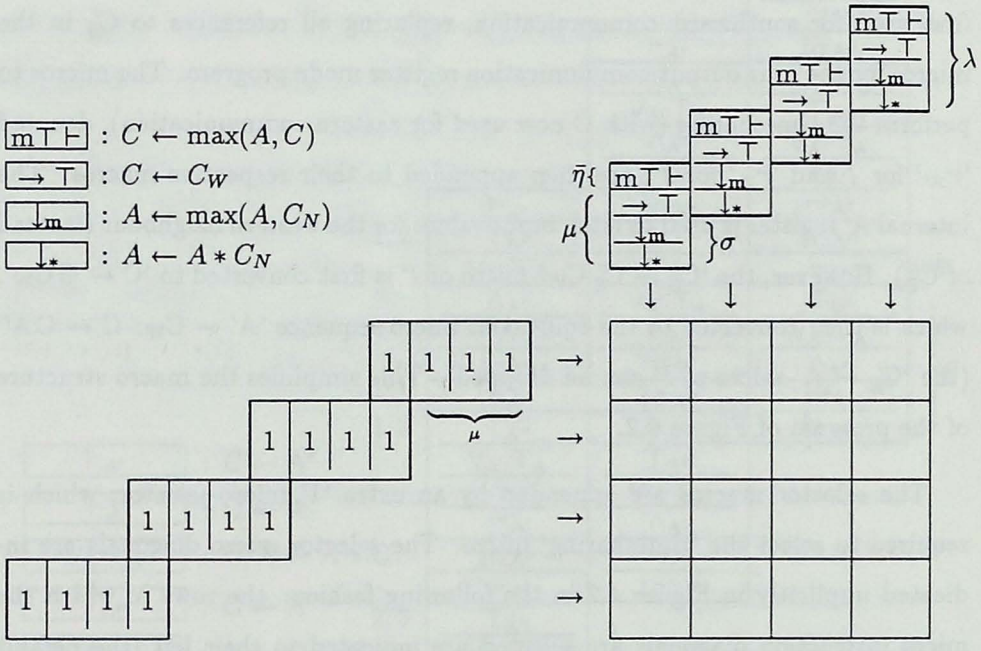


Figure 4.1: Microprogrammed LCS program on a 3×3 μISA with $\lambda = 4$, $\mu = 3$, $\sigma = 2$ and $\bar{\eta} = 1$

4.2.2 Transitive closure algorithm revisited

The normal μISA implementation of the transitive closure algorithm (see Section 3.5.2), using four output communication registers with $\lambda = \mu = 5$, can be easily transformed for implementation on a μISA using one output communication register, by timesharing this register to a factor of $\bar{\eta} + 1 = 2$. This in turn increases the microcode length to $\lambda = \mu + 1 = 6$.

Comparing with Figure 3.5(a), the instruction macros, I and I', are converted in a fairly straightforward manner, as indicated in Figure 4.2. The C register is first used for southward communication, replacing all references to C'_S in the micros for the four output communication register mode program. The micros to perform the timesharing (with C now used for eastern communication), denoted '$\vdash_{A'}$' for I and '$\vdash_{A}$' for I', are then appended to their respective macros. The internal A′ register is used to latch input values for the western neighbour (instead of C'_E). However, the '$C'_S \leftarrow C'_S\, C_W$' micro of I' is first converted to '$C \leftarrow C\, C_W$', which is then converted to the equivalent micro sequence '$A' \leftarrow C_W$; $C \leftarrow C\, A''$' (the '$C'_E \leftarrow A$' micro of I' can be dropped). This simplifies the macro structure of the program of Figure 4.2.

The selector macros are appended by an extra '1' micro-selector, which is required to select the 'timesharing' micro. The selector micro diagonals are indicated implicitly in Figure 4.2 in the following fashion: the rows in which the micro instruction diagonals are selected are indicated to their left (the default is all rows are selected). Thus, in the first macro diagonal, the third micro is selected on all rows, whereas the fifth micro is selected on rows 2 and 3 only. For the sake of brevity, this representation of μISA programs will be used henceforth.

The A'-value for cell (i, j) is read (from the west) on the 2nd microcycle, reading the value of the C register of cell $(i, j-1)$ set by the $2 + \mu - 1 = 6$th microcycle. Similarly, the A''-value for cell (i, j) is read (from the north) on the 1st microcycle, reading the value of the C register of cell $(i-1, j)$ set by the $1 + \mu - 1 = 5$th microcycle. In this way, the C register is used to efficiently pass two items of nontransmittent data.

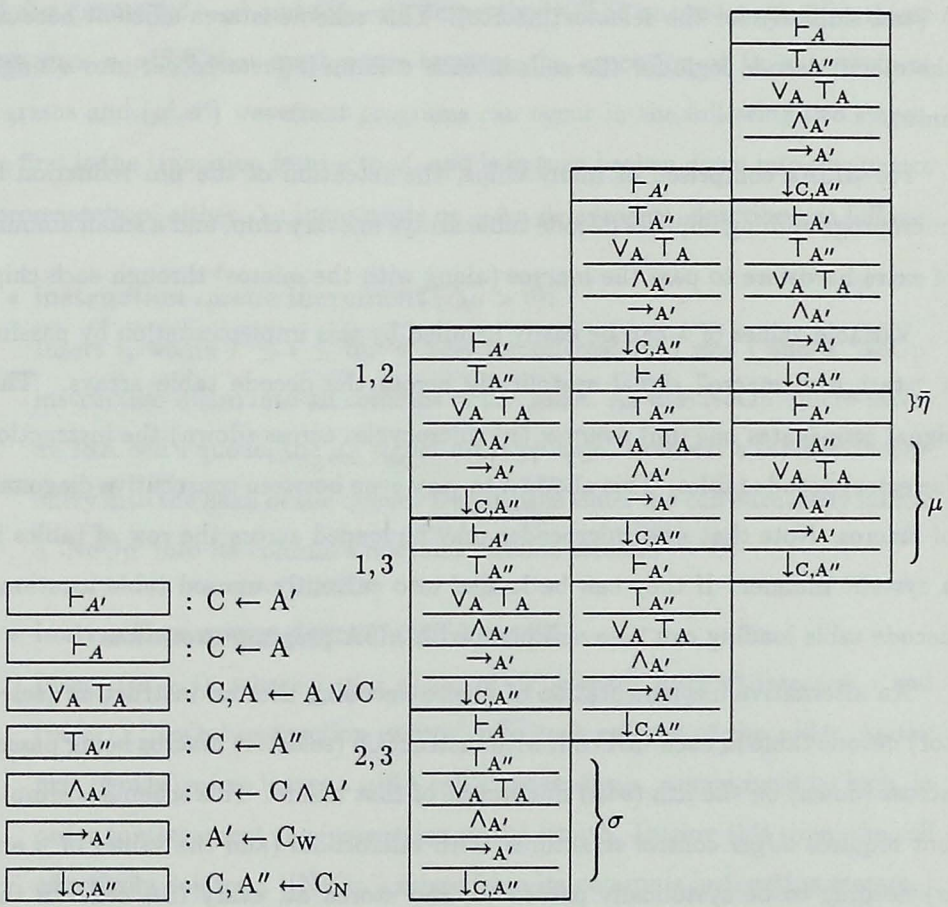


Figure 4.2: Transitive closure program using timesharing of the C register for a 3×3 normal μISA with $\mu = \sigma = 5$ and $\bar{\eta} = 1$

4.3 Implementation of the microprogrammed ISA

An implementation of decoding macros for the μISA was given in Section 3.5.1. There, it was described how a row of decode tables, connected to the north edge of a μISA, decodes instruction macros into their respective microcodes of length λ (and similarly for the selector macros). This scheme is area-efficient because the macro decode logic for the cells in each column is *factored out* into a single unit.

For μISAs comprised of many chips, the retention of the pin reduction of microprogramming requires decode table arrays in every chip, and a small amount of extra hardware to pass the macros (along with the micros) through each chip.

Variable values of λ can be easily handled by this implementation by passing a "start new macro" signal systolically across the decode table arrays. This signal propagates one unit every σ (μ) microcycles across (down) the instruction (selector) decode tables. This allows λ to vary even between consecutive diagonals of macros. Note that new microcodes may be loaded across the row of tables in a systolic manner. If they can be loaded into currently unused table locations, decode table loading can be overlapped with μISA program execution.

An alternative implementation of macro decoding uses an instruction (selector) decode table in each ISA cell, with instruction (selector) macros being passed across (down) on the μth (σth) microcycle of that macro. This scheme is simpler but requires larger control structures, with microcodes (and the values of μ and σ) needing to be systolically passed to, and stored at, every ISA cell. In this implementation, an ISA cell would pass down (across) an instruction (selector) macro during the execution of the μth (σth) of the λ microcycles of that macro.

In the standard implementation, each ISA cell has an instruction (selector) queue of length μ (σ), effectively slowing down the rate of instruction (selector) micros through the ISA by a proportional amount. They are the essential control structures for the implementation of ISA microprogramming. The detailed implementation of these queues is described below.

4.3.1 Implementation of variable length queues

It is assumed here that each μISA cell has a variable length instruction (selector) queue, which can have its length incremented/decremented within a single microcycle[4]. The systolic control signals $\mu_+, \mu_-, (\sigma_+, \sigma_-)$ respectively increment or decrement, the instruction (selector) queue. These signals themselves are assumed to propagate through the μISA through at least one of queues. Let $\Delta\mu$ and $\Delta\sigma$ denote $(\mu' - \mu)$ and $(\sigma' - \sigma)$, respectively. The queue length adjustment of an $m \times n$ μISA that must occur between the execution of (μ, σ) wavefront programs and (μ', σ') wavefront programs can occur in the following two stages. The first is the transition from μ to μ', and is in turn broken down into a sequence appropriately of either $\Delta\mu$ increments or $-\Delta\mu$ decrements, described as follows:

- **instruction queue increment** $(\Delta\mu > 0)$:
  Insert i, where $1 \leq i \leq m$, '0' selector micros[5] into row i and a 'NoOp' instruction micro into all columns of the μISA. As the 'NoOp' micro leaves an ISA cell's queue, the μ_+ signal must be high, to insert a further 'NoOp' entry into the head of the queue. During this time, the cell effectively inserts a 'NoOp' into its column's microinstruction stream.

- **instruction queue decrement** $(\Delta\mu < 0)$:
  Insert $(m - i)$, where $1 \leq i \leq m$, of '0' selector micros[6] into row i and $(m + 1)$ 'NoOp' instruction micros[7] into each column of the μISA. As the first 'NoOp' micro leaves a μISA cell's queue, the μ_- signal must be high, in order to decrement the instruction queue length. During this time, the cell effectively deletes a ('NoOp') micro from its column's instruction stream.

The second is the transition from σ to σ', and is identical to the first step with rows interchanged with columns; instructions with selectors; i, m, μ, μ' with

[4]Similar ideas can be used for the alternative implementation of macro decoding which might be capable of making arbitrary adjustments to μ and σ within a microcycle.

[5]These are used to skew the selectors by μ' units.

[6]Ditto.

[7]These are to be overtaken by the faster moving micros to come.

j, n, σ, σ'. Thus, the ISA loses an effective period of:

$$\max(\Delta\mu, -(m+1)\Delta\mu) + \max(\Delta\sigma, -(n+1)\Delta\sigma)$$

microcycles during this transition, the effective period being the number of 'NoOp' micros it has to insert into column 1 of the ISA. Note that the first diagonal of a faster (μ', σ') wavefront program must begin at least:

$$t_{\text{del}}(\Delta\mu, \Delta\sigma) = \max(0, -(m-1)\Delta\mu) + \max(0, -(n-1)\Delta\sigma) \qquad (4.3)$$

microcycles after the last diagonal of the old program, to avoid a 'collision' with it. Hence, the above algorithm for performing the transition is efficient.

It has been mentioned that the signals μ_+, μ_-, σ_+ and σ_- are pumped through the ISA at the same rate as the micro instructions (or selectors). This can be achieved by passing them through (say) the selector queues. While these are incremented, a low value of these signals must also be inserted into the queues. While these are decremented, the above scheme guarantees that the entry deleted is a '0' selector with low-valued signals. With a little care, using the above scheme, these four signals could be easily encoded into two, further reducing the hardware overheads. All of these signals can propagate through the ISA as an instruction propagates through an SISA, so they require negligible extra input pins.

The use of queues to implement microprogramming 'factors out' the more bulky decode tables from the μISA columns. Appendix 4.A proposes a variable length queue design suitable for efficient VLSI implementation with moderate ranges of μ and σ.

4.4 The microprogrammed ISA in relation to other meshes

Section 4.1.3 suggests that the extra control structures of the μISA make it a more powerful model than the ISA, and hence also than the SIMD mesh. This section justifies this claim for a (μ, σ) wavefront $n \times n$ μISA, with the establishment of theoretical relationships between the μISA and meshes from the SIMD to the MIMD classes.

For this comparison, assume that all meshes are $n \times n$ and communicate via (the same type of) communication registers, and use the same instruction sets. A Processor Array (PA) is defined to be an $n \times n$ mesh of independently programmable processors; this model can use, among other things, arbitrary wavefronts. In an Instruction Broadcasting Array (IBA) [21], instructions (selectors) are independently broadcast to each column (row) of the mesh; this model uses only a $(0,0)$ wavefront. The IBA is a generalization of a "vector-orientated" SIMD mesh [21], ie. a SIMD mesh using only row/column vector masking mechanisms (eg. the ICL DAP [17, pp243-248]).

In Section 4.4.1 the mesh 'power' hierarchy:

$$\text{PA} \Rightarrow \mu\text{ISA} \Rightarrow \text{ISA} \Rightarrow \text{IBA}$$

is established in both a theoretical (proofs of general 'efficient' simulation techniques) and a practical (asymptotic efficiency on known practical problems) sense. A model's position in this hierarchy is determined by the values (chiefly, the signs) of the wavefront parameters that it allows.

Section 4.4.2 gives timing rules for the μISA, which are used again in Section 4.6.

Section 4.4.3 demonstrates how an arbitrary μISA can efficiently simulate an ISA (this result is required to establish the above hierarchy). An intermediate result is that a (μ, μ) wavefront μISA program can be 'efficiently' simulated by a $(\mu + \delta, \mu + \delta)$ wavefront μISA, where $\delta \geq 0$. This result has significance in the practical development of the normal μISA model: from it, the delay associated with changing the wavefront parameters can be compared with that of simulation, for programs using smaller wavefront parameters.

4.4.1 Establishing the relationship

Figure 4.3 summarizes the relationship between the various models of processor meshes. The relationship '$P \Rightarrow Q$', read as "P efficiently simulates Q", denotes that an arbitrary program of period t in model Q can be simulated by a program of period $O(t)$ in model P. It can be seen at once that this relationship is

$$\begin{array}{ccccccc} \text{PA} & \Rightarrow & \mu\text{ISA} & \Rightarrow & \text{ISA} & \Rightarrow & IBA \\ (\text{any } \mu, \sigma) & & (\mu > 0, \sigma > 0) & & (\mu = \sigma = 1) & & (\mu = \sigma = 0) \end{array}$$

Figure 4.3: Simulation relationships between various models of processor meshes, with their wavefront parameter restrictions.

transitive. The relationships 'PA⇒ISA' and 'ISA⇒IBA' are established in [21]; a similar proof to that of 'PA⇒ISA' demonstrates that 'PA⇒μISA' must also hold. This is because the PA is the most powerful way that such a mesh can be programmed. It remains to show that 'μISA⇒ISA': this will be demonstrated in Section 4.4.3. All these results have practical significance, since their proofs also give general transformations from programs in the less powerful model to programs of similar efficiency in the more powerful model.

For these models, the reverse of these simulation relationships do not hold. For example, an $O(1)$ PA (ISA) program can be found that cannot be simulated in less than $\Omega(n)$ period on a ISA (IBA) [21]. Using similar ideas, it can also be shown that with $\mu > 1$ and/or $\sigma > 1$, there exists an $O(1)$ μISA program that cannot be simulated in less than $\Omega(n)$ period on the ISA. However, these counter-results lack direct practical significance since the programs used in the proofs are rather contrived.

For counter-results, the power of a mesh model is more usefully indicated by considering whether a model can implement a practical problem (asymptotically) more efficiently than another model. There appears to be no $O(1)$ period program that can perform a horizontal ringshift on an IBA or SIMD mesh, although such a program exists for an ISA (see program Rot1H, Section 3.6). There also appears to be no $O(1)$ period program than can perform a 'reverse' horizontal ringshift[8] on the ISA or μISA, although such a program exists for the PA[9]. Finally, for meshes with communication registers that can store a single integer, there appears to be no $O(1)$ period program performing the LCS algorithm operation on an ISA in constant period, although such a program is given for a $(3,2)$ wavefront μISA

[8]ie. a ringshift in the opposite direction to that used for the ISA program.

[9]This PA program would use a $(1,-1)$ wavefront, and the period is taken with respect to this wavefront.

(see Section 4.2.1). It can then be concluded that the mesh hierarchy of Figure 4.3 holds in a practical and strong sense.

In the sense of the previous paragraph, the difference in power between meshes using $(0,0)$ and $(1,1)$ wavefronts (ie. between the IBA and ISA) is greater than that between meshes using $(1,1)$ and [say] $(3,2)$ wavefronts (ie. between the ISA and the μISA). This is demonstrated by the fact that when in practice a μISA yields an asymptotically more efficient solution than an ISA, it was because the ISA was limited by inadequate instruction granularity. For this reason, the main application of the μISA is seen as a flexible and area-efficient emulator of arbitrary instruction granularity ISAs. It also indicates that there is a diminishing return in the extra μISA control structure hardware required to implement higher and higher values of μ and σ (see Section 4.3).

Combining this observation with Figure 4.3, it can also be seen that the power of a mesh model can be characterized by the types of wavefronts that it can use, with the signs (positive, negative or zero) of μ and σ being more critical than their magnitudes.

4.4.2 The microprogrammed ISA timing rules

The microprogrammed ISA timing rules define the communication part of the μISA, and are used to prove μISA simulation results. The μISA timing rules state that upon an ISA cell executing the kth micro of an ISA program, it reads the communication register of the neighbouring cell in direction d, $d \in \{\mathrm{N},\mathrm{E},\mathrm{W},\mathrm{S}\}$, left by the $(k+\delta_d(\mu,\sigma))$th micro (ie. this micro was executed on the last microcycle). Note that the first micro for column j enters the $(1,j)$th cell $(j-1)\sigma$ microcycles after the first micro enters cell (1,1). The rules are summarized in Table 4.1. Note that $\delta_d(1,1)$ gives the timing rules for an ISA, and $\delta_d(0,0)$ gives those for an IBA.

4.4.3 Simulation of ISA programs

This section shows that a μISA efficiently simulates an ISA (and hence an IBA), and that normal μISAs can efficiently simulate normal μISAs using a smaller

d :	cell (from (i,j)	$\delta_d(\mu,\sigma)$
N	$(i-1,j)$	$\mu-1$
S	$(i+1,j)$	$-\mu-1$
W	$(i,j-1)$	$\sigma-1$
E	$(i,j+1)$	$-\sigma-1$

Table 4.1: Timing rules for a (μ,σ) wavefront microprogrammed ISA

value of μ. These results use general program transformations which assumes that the μISA has an extra internal register, called D, which is not available to the ISA.

However, since the μISA is proposed to implement 'dynamic' microprogramming, the existence of such transformations are not crucial in practice. They are useful, however, when a μISA predominantly using (μ,σ) wavefront programs needs to simulate a short ISA (IBA) program, and the delay in changing the wavefront parameters exceeds that associated with the transformation. In such cases, these simple transformations can be applied (requiring only very modest control structures to dynamically perform such transformations) to improve the overall efficiency of the μISA.

Firstly, the lemmas showing that non-normal μISAs can efficiently simulate an ISA are given:

Lemma 4.1 *An arbitrary ISA program of period t can be simulated by a (μ,σ) wavefront μISA, with $\mu > \sigma \geq 1$ in period $t_{\mu,\sigma} = \mu t$.*

transformation:

> *I_{kj}, the (k,j)th element of the ISA program's instruction matrix, is used to form the (k,j)th macro (of length $\lambda = \mu$) of the μISA program's instruction matrix:*
>
> $$\mathrm{D} \leftarrow \mathrm{C_E};\ (\mathrm{NoOp})^{\mu-2};\ I'_{kj}$$
>
> *where $I'_{kj} = (I_{kj})^{\mathrm{C_E}}_{\mathrm{D}}$. S_{ik}, the (k,j)th element of the ISA program's instruction matrix, is similarly used to form the corresponding selector macro of the μISA program's selector matrix:*
>
> $$1;\ (0)^{\mu-2};\ S_{ik}$$

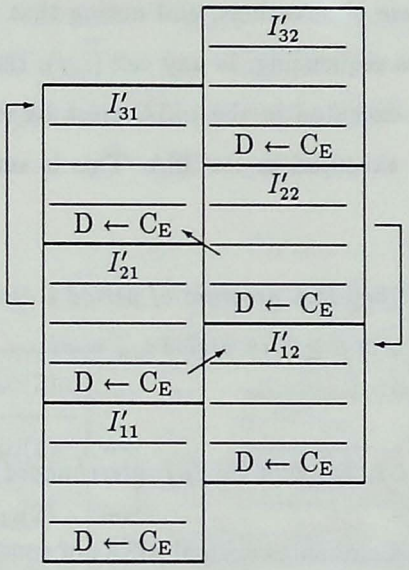


Figure 4.4: A $(4, 2)$ wavefront μISA program simulating an ISA program (instruction part)

This is illustrated for the instruction macro matrix in Figure 4.4.

proof (semi-formal):

Consider Figure 4.4, as cell $(i, 1)$ executes micro I'_{31}. The timing rules state that C_S contains the value of the communication register of cell $(i + 1, 1)$ left by the 'NoOp' preceding I'_{21}. This value must be the same as that left by the I'_{11} micro. Similarly, D contains the value of C_E at the preceding '$\leftarrow$' micro, ie. the value of the communication register of cell $(i, 2)$ left by the first 'NoOp' after I'_{12}. This value must be the same as that left there by the I'_{21} micro.

Similarly, as cell $(i, 2)$ executes the I'_{12} macro, C_W contains the value of the communication register of cell $(i, 1)$ left by the first 'D $\leftarrow$ C_E' micro after I'_{11}. This value must be the same as that left by the I'_{11} micro. Similarly again, C_N contains the value of the communication

register of cell $(i-1,2)$ left by the 'NoOp' micro preceding I'_{22}. This value must be the same as that left by the I'_{21} micro.

Generalizing these observations, and noting that the above transformation preserves sequencing, in any cell (i,j), the operand values of micro I'_{kj} when executed in the μISA must be identical to those of micro I_{kj} when executed in the ISA. This is sufficient to establish equivalence. □

Lemma 4.2 *An arbitrary ISA program of period t can be simulated by a (μ,σ) wavefront μISA, with $\sigma > \mu \geq 1$ in period $t_{\mu,\sigma} = \sigma t$.*

transformation:

(as for Lemma 4.1, but with C_E (μ) interchanged with C_S (σ))

How a normal μISA simulates normal μISAs of smaller wavefront parameters can be derived using the lemma:

Lemma 4.3 *An arbitrary (μ,μ) wavefront μISA program of period $\mu t_{\mu,\mu}$ can be simulated by a $(\mu+1,\mu+1)$ wavefront μISA in period $(\mu+1)t_{\mu,\mu}$.*

transformation:

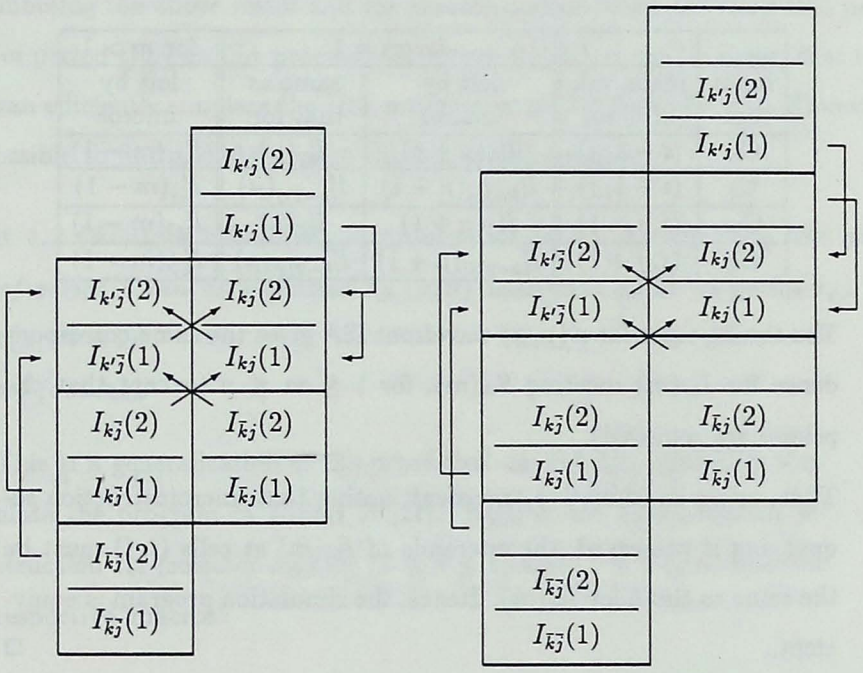
Let I_{kj} [S_{ik}] denote the (k,j)th instruction [(i,k)th selector] 'macro' of length $\lambda = \mu$ of the (μ,μ) wavefront program (for $1 \leq i,j \leq n$ and $1 \leq k \leq t_\mu$), with $I_{kj}(m)$ [$S_{ik}(m)$] denoting the the mth micro of this macro (for $1 \leq m \leq \mu$).

This is used to form I'_{kj} [S'_{ik}], the (k,j)th instruction [(i,k)th selector)] 'macro' of the simulation program simply by appending a 'NoOp' ['0'] micro to it, ie.:

$$I'_{kj} = \text{`}I_{kj}; NoOp\text{'}$$
$$S'_{ik} = \text{`}S_{ik}; 0\text{'}$$

This is illustrated for the instruction macro matrix in Figure 4.5.

proof:



(a) original program around I_{kj} (b) transformed program around I_{kj}

Figure 4.5: A $(3,3)$ wavefront μISA program simulating a $(2,2)$ wavefront μISA program

(instruction part; note: $\bar{x} = x - 1, x' = x + 1$)

Consider the meeting of $I'_{kj}(m)$ and $S'_{ik}(m)$ in cell (i,j), for $1 \leq m \leq \mu$. The following tables determines input register values read at this point. Columns 3 and 5 are determined by the timing rules for a $(\mu+1, \mu+1)$ ISA. The macros of column 4 are all 'NoOp's; hence, the communication register values are determined by the preceding micro (column 5).

		for $m = 1$		for $m > 1$
input reg.	reads val. of C in:	left by micro:	same as that for:	left by micro:
C_N	$(i-1,j)$	$I'_{kj}(\mu+1)$	$I'_{kj}(\mu)$	$I'_{k'j}(m-1)$
C_S	$(i+1,j)$	$I'_{(k-2)j}(\mu+1)$	$I'_{(k-2)j}(\mu)$	$I'_{\bar{k}j}(m-1)$
C_W	$(i,j-1)$	$I'_{k\bar{j}}(\mu+1)$	$I'_{k\bar{j}}(\mu)$	$I'_{k'\bar{j}}(m-1)$
C_E	$(i,j+1)$	$I'_{(k-2)j'}(\mu+1)$	$I'_{(k-2)j'}(\mu)$	$I'_{\bar{k}j'}(m-1)$

The timing rules for a (μ,μ) wavefront ISA gives the same correspondence for $I_{kj}(m)$ meeting $S_{ik}(m)$, for $1 \leq m \leq \mu$ (except that the primes are removed).

Thus, using an inductive argument, noting that microinstruction sequencing is preserved, the operands of $I'_{kj}(m)$ at cells (i,j) must be the same as those for $I_{kj}(m)$. Hence, the simulation program is equivalent. □

From these lemmas, a (μ,σ) wavefront μISA can be shown to efficiently simulate an ISA:

Result 4.1 (A μISA efficiently simulates an ISA) *An arbitrary ISA program of period t can be simulated by (μ,σ) wavefront μISA in period:*

$$t_{\mu,\sigma} = \max\{\mu,\sigma\}t$$

proof:

Combine the transformations of Lemma 4.1 (for the case $\mu > \sigma$), Lemma 4.2 (for the case $\mu < \sigma$), and Lemma 4.3 (applied $\mu - 1$ times, for the case $\mu = \sigma$). □

Hence, the delay of changing the wavefront parameters to $(1,1)$ and executing the ISA program exceeds the delay of this transformation when:

$$(\mu-1)n+(\sigma-1)n+t \geq \max\{\mu,\sigma\}t$$

which always occurs when:

$$t \leq n/2 \tag{4.4}$$

Combining the above result and the transformation of an arbitrary IBA program (of period t) by an ISA program (of period $3t$) [21], it can be shown that the μISA can efficiently simulate the IBA program in period $3\max\{\mu,\sigma\}t$. However, it is possible to find a slightly more efficient transformation:

Result 4.2 (A μISA efficiently simulates an IBA) *An arbitrary IBA program of period t can be simulated by (μ,σ) wavefront μISA in period $t_{\mu,\sigma} = (2\max\{\mu,\sigma\}+1)t$.*

proof (outline):

This is a generalization of the proof that an ISA can efficiently simulate the program in period $3t$ [21]. Expand the IBA program instruction I_{kj} (selector S_{ik}) for $(1 \leq k \leq t)$ and $(1 \leq j \leq n)$ into the respective 'macros':

$$\begin{aligned} I'_{kj} &= (I_{kj})^{\mathrm{C}}_{\mathrm{D}};\ (\mathrm{NoOp})^{\max\{\mu,\sigma\}-1};\ \mathrm{C} \leftarrow \mathrm{D};\ (\mathrm{NoOp})^{\max\{\mu,\sigma\}} \\ S'_{ik} &= S_{ik};\ (0)^{\max\{\mu,\sigma\}-1};\ S_{ik};\ (0)^{\max\{\mu,\sigma\}} \end{aligned}$$

The first sequence of 'NoOp's ensures that no eastward or southward communication occurs within the macro diagonals; the second ensures that the succeeding macro diagonal can read from the south or east the results of the execution of this diagonal. □

Hence the delay of changing the wavefront parameters to $(1,1)$ and executing the ISA program simulating the IBA program exceeds the delay of this transformation when:

$$(\mu-1)n+(\sigma-1)n+3t \geq (2\max\{\mu,\sigma\}+1)t$$

which always occurs again when:

$$t \leq n/2 \tag{4.5}$$

It is here conjectured that in practice a normal μISA (using a sufficiently large value of μ) can efficiently simulate an arbitrary μISA. This, and the fact that the normal μISA is most appropriate to simulate ISA programs of (unlimited) instruction granularity, makes the normal μISA preferable to be implemented in hardware. Hence, the following result is useful:

Result 4.3 (A normal μISA efficiently simulates 'smaller' normal μISAs) *For any $\mu' \geq \mu$, a (μ, μ) wavefront μISA program of period $\mu t_{\mu,\mu}$ can be simulated by a (μ', μ') wavefront μISA in period $\mu' t_{\mu,\mu}$.*

proof:

Apply $(\mu' - \mu)$ times the transformation of Lemma 4.3. □

Hence, while predominantly using (μ', μ') wavefront programs in a normal μISA, executing a (μ, μ) wavefront program may be required. Here, the delay of changing the wavefront parameters to (μ, μ) and directly executing the program exceeds the delay of this transformation when:

$$(\mu' - \mu)n + (\mu' - \mu)n + \mu t_{\mu,\mu} \geq \mu' t_{\mu,\mu}$$

which always occurs when:

$$\mu t_{\mu,\mu} \leq \mu n/2 \tag{4.6}$$

4.5 Givens Rotations: a microprogrammed ISA case study

Givens Rotations are used to triangularize matrices, and form an important step in matrix inversion. The Givens Rotations algorithm is representative of matrix QR factorization algorithms [5, pp161-165], all of which appear to require a similar (instruction) systolic implementation. A systolic algorithm for performing the Givens Rotations is found in [13], and an implementation for the Instruction Systolic Array (ISA) is found in [7]. This implementation is one of the first

demonstrations of how an ISA can handle relatively complex patterns of non-transmittent data. It also demonstrates that the ISA implementation of systolic algorithms can be considerably complicated by restricting the ISA to pass only single words of data between adjacent cells per (macro) instruction cycle.

Variations of the Given Rotations algorithm give an interesting application for the (normal) μISA, illustrating how it can simulate an ISA of very high instruction granularity (in terms of the amount of computation and communication per [macro] instruction cycle). The practical ISA implementation of Givens Rotations is thus enhanced considerably, in terms of efficiency and ease of programming, using microprogramming techniques. The most important of these is communication register timesharing, especially for one output communication register mode μISAs. Optimized variations of the algorithm also illustrate the use of variable microcode lengths. However, while the normal μISA is the easiest to program, a $(\mu, 1)$ wavefront μISA can implement the same algorithms in slightly lower period.

The Givens Rotations algorithm for the ISA requires in any case either microprogramming or an extremely (perhaps unreasonably, if the ISA is intended for other applications) powerful instruction set. For convenience, it is assumed here that all matrices are $n \times n$; however, all ideas and algorithms here can be easily extended to the more general $m \times n$ matrices.

While this section is not essential for reading the rest of this chapter, a feel for how the μISA simulates a high instruction granularity ISA (see especially Sections 4.5.3 and 4.5.5) will be useful in understanding the more general treatment of Section 4.6.

4.5.1 Description of the systolic algorithm

As in [7], the algorithm transformimg a matrix A into upper triangular form can be expressed as:

Givens:

$A^n \leftarrow A$;

for $i := 1$ **to** $n - 1$ **do**

$A^i \leftarrow A^n$;

for $j := i+1$ **to** n **do**

$A^{j+1} \leftarrow T^{ij}A^j$;

where T^{ij} is a rotation matrix to annihilate $A^j_{j,i}$ and has the form:

$$T^{ij} = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & C^{ij} & S^{ij} & & & \\ & & & -S^{ij} & C^{ij} & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix}$$

where the *cosine-sine* coefficients occupy the intersections of rows i, j and columns i, j respectively and can be computed by the the procedure call:

$$\text{generate}(A^j_{ii}, A^j_{ji}, C^{ij}, S^{ij})$$

where:

generate(x, y, c, s):

if $(x = 0)$ then $c := 0; s := 1$

else $t := y/x; c := \sqrt{1+t^2}; s := ct$

Using the form of T^{ij}, the body of the inner loop of algorithm Givens can be re-expressed as, for $1 \leq k \leq n$:

$$\begin{aligned} A^{j+1} &\leftarrow A^j; \\ A^{j+1}_{ik} &\leftarrow C^{ij}A^j_{ik} + S^{ij}A^j_{jk}; \\ A^{j+1}_{jk} &\leftarrow -S^{ij}A^j_{ik} + C^{ij}A^j_{jk} \end{aligned}$$

and since $j > i$ and the jth row of the matrix is only operated on the $(j-i)$th iteration of the inner loop, then $A^i_{jk} = A^{i+1}_{jk} = \ldots = A^j_{jk}$ and the above inner loop body can be again re-expressed as:

$$A^{j+1} \leftarrow A^j; \tag{4.7}$$

$$A^{j+1}_{ik} \leftarrow C^{ij}A^j_{ik} + S^{ij}A^i_{jk}; \tag{4.8}$$

$$A^{j+1}_{jk} \leftarrow -S^{ij}A^j_{ik} + C^{ij}A^i_{jk} \tag{4.9}$$

Note that the ith row of the result matrix is not changed after the ith iteration of the outer loop of algorithm Givens.

4.5.2 ISA implementation using enhanced communications

Based on the ideas of [7], an ISA (a normal μISA) implementation of the Givens Rotations algorithm is first given; this architecture is assumed to be capable of communicating both rotation parameters westwards during a single (macro) instruction cycle.

An ISA using the four output communication register mode is used here. However, C'_E is special in that it has two *banks*, ie. it can hold two numbers (the sine and cosine rotations). These banks are denoted here $C'_E.c$ and $C'_E.s$.

For comparison with the ISA program of [7, Sect.4], the following symbolic register names are used in this program:

$$C' = C'_E.c;\ \ S' = C'_E.s;\ \ C = C_W.c;\ \ S = C_W.s$$
$$A = C'_N;\ \ A_S = C_S;\ \ B = C'_S;\ \ B_N = C_N$$

and the program uses the 'sub-macros':

$$\begin{aligned} CS &: \text{generate}(B_N, A, C', S') \\ CS' &: \text{generate}(B, A_S, C', S') \\ R &: B \leftarrow (C'B_N + S'A) \\ R' &: A \leftarrow (-S'B + C'A_S) \\ \Rightarrow &: C' \leftarrow C;\ \ S' \leftarrow S \end{aligned}$$

The program assumes the input matrix A initially resides in row-major order in the (symbolic) A registers of the ISA. It consists of the sequential blocks $G_1; G_2; \ldots; G_{n-1}$, similar to the program of [7, Fig.4], but each block is much simpler and shorter. Block G_i corresponds to the body of the outer loop of algorithm Givens and operates on rows 1 to n_i (where $n_i = n - i + 1$) and columns n_i to n of the ISA only, and is illustrated in Figure 4.6. Block G_i assumes that

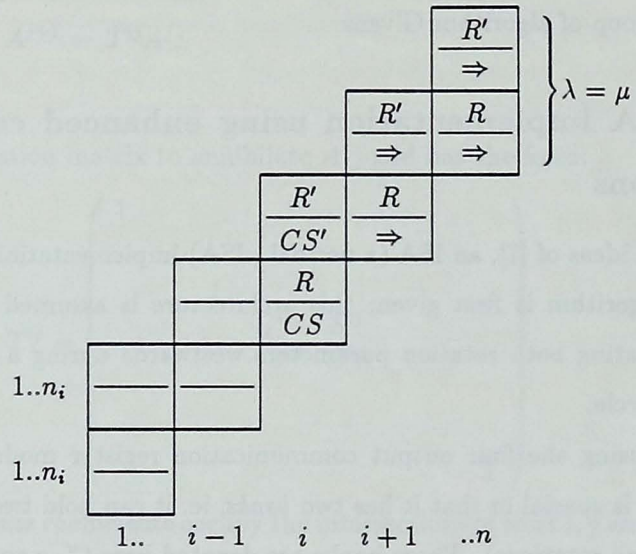


Figure 4.6: Givens Rotations block G_i on an enhanced communication ISA (normal μISA)

the last n_i rows of A^i reside in row-major order in the A registers of the first n_i rows of the ISA. After its execution, the first i rows of the rotated matrix lie in the B-registers of rows n_i to n of the ISA, ie. in a 'row-reversed' row-major order.

Consider the execution of block G_i in ISA cell (i', k), where $1 \leq i' \leq n - i + 1$ and $i \leq k \leq n$. Assume that initially the A register contains A^i_{jk}, and B_{N} reads A^j_{ik}, where $j = i' - i$. The (first) 'CS' sub-macro on column $k = i$ then performs the procedure call generate$(A^i_{ji}, A^j_{ii}, C^{ij}, S^{ij})$. The results of this call are broadcast east along row i'. The 'R' sub-macro corresponds to equation (4.8), so that the B register gets set to A^{j+1}_{ik}. On row 1, the B_{N} registers are assumed to be held at 0, so that here the 'R' sub-macro effectively performs the assignment '$B \leftarrow A$'. The 'R'' sub-macro corresponds to equation (4.9), setting the A register to $A^{j+1}_{(j+1)k}$, since A_{S} reads the value of $A^i_{i(j+1)}$. Thus, this sub-macro shifts the remaining matrix north one unit.

As was suggested possible in [7, Sect.9], each block G_i requires only two instruction diagonals. In a practical normal μISA implementation, the generate and rotate sub-macros might themselves be broken down into $\mu_1, \mu_2 \geq 1$ micros,

respectively, so that the microcode length here is given by $\lambda = \mu_1 + \mu_2$, (with $\lambda = \sigma = \mu$). For the normal μISA implementation, restrictions must be placed in the microcode structure so that the ISA is properly simulated. In this case, it is sufficient that each communication register is not modified before some fixed microcode position (say m_1), and is read by micros (at microcode positions $m_2 \leq m_1$) in the appropriate neighbouring cell (see Section 4.6).

This version is a significant improvement on the period of the corresponding μISA implementation without using such enhanced communications or timesharing [7, Fig.6] which would require $11(\mu_1 + \mu_2)$ microinstructions per block. This is because the most powerful instruction there also involves both a *sine-cosine* generation and a rotation.

4.5.3 Timesharing for the four output register mode

The above algorithm assumes that the eastern output communication register has two banks which can be simultaneously read. In practice, such a feature may require an undesirably high inter-cell I/O bandwidth, but can be obviated provided the 'CS', 'CS''' 'micros' can be broken down into $\mu_1 \geq 2$ microcycles, and by *timesharing* C'_{E} to a factor of $\bar{\eta} + 1 = 2$. While this maintains the microinstruction skew at $\mu = \mu_1 + \mu_2$, the microcode length is slightly increased to $\lambda = \mu + \bar{\eta}$.

C' is now bound to C'_{E}, S' is bound to a fresh (non-communication) register, and the '$\Rightarrow$' sub-macro is redefined to be the sequence of μ_1 micros:

$$\Rightarrow \ : \ C' \leftarrow \mathrm{C_W}; \ S' \leftarrow \mathrm{C_W}; \ \text{No-Op}^{\mu_1 - 2}$$

Correspondingly, each R and R' sub-macro must be appended with the 'overlap' micro to output the *sine* value (which enables the timesharing of C'_{E}):

$$\mathrm{C}'_{\mathrm{E}} \leftarrow S'$$

The corresponding program is given in Figure 4.7. Note that columns $j < i$ execute 'NoOp's for block G_i, and are omitted from this Figure.

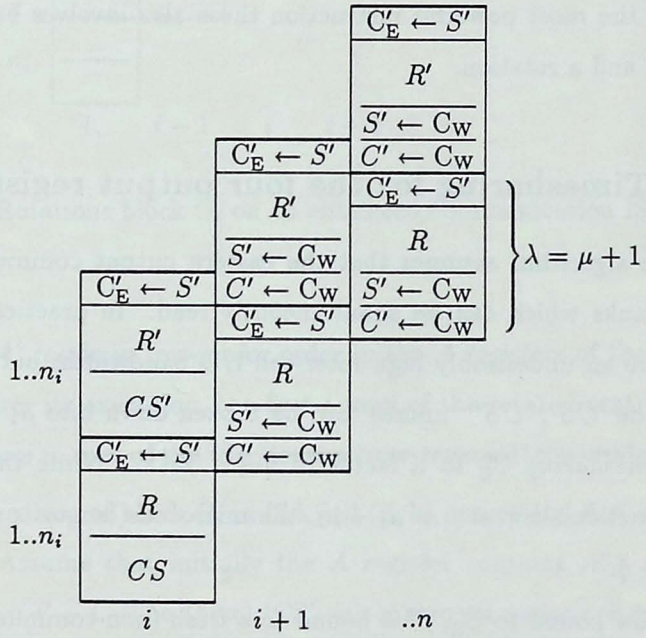


Figure 4.7: Givens Rotations block G_i on a μISA with two-way timesharing of eastern output register (with $\mu_1 = \mu_2 = 2$)

4.5.4 Timesharing for the one output register mode

Here, the normal μISA has its single output communication register C timeshared by a factor of three (passing the values of (B, C', S') on the first macro, and (C', S', A) on the second. This requires a macro overlap of $\bar{\eta} = 2$ and a longer macro length of $\lambda = (\mu_1 + \mu_2 + 2) + \bar{\eta}$.

The program is similar to that of Figure 4.7. However, the symbolic registers B, A, C', B_N and A_S are now bound to fresh non-communication registers. The macros of the first (second) diagonal are preceded by a micro to read the required value into B_N (into A_S), and are appended with the 'overlap' micro sequence "C $\leftarrow$ B; C $\leftarrow$ C'; C $\leftarrow$ S'" ("C $\leftarrow$ C'; C $\leftarrow$ S'; C $\leftarrow$ A"). The case for the second macro diagonal is different because northward/westward communication must occur after southward/eastward communication for this style of macro structure (see item (a) of Section 4.6.2).

Block G_i of the program is illustrated in Figure 4.8. Note that after a cell executes 'C $\leftarrow$ B', the new B-value can be read into the southern neighbouring cell as it executes the '$B_N \leftarrow C_N$' micro (at distance $\mu - 1$ below). Similarly, after a cell executes the 'C $\leftarrow$ A' micro, the new A-value is ready to be read into the northern neighbouring cell as it executes the $(\mu + 1)$th micro of block G_{i+1} (at distance $\mu + 1$ above). Providing that the first four micros of G_{i+1} do not modify the C register (a reasonable assumption, if $\mu_1 \geq 2$), the new A-value can still be read into the northern neighbouring cell either two or four microcycles later, as it executes the '$A_S \leftarrow C_S$' micro of G_{i+1}.

4.5.5 Optimizing the period

Equations (4.8) and (4.9) require only one *sine-cosine* generation per rotation. By not performing the northward shift in sub-macro R', ie. by redefining:

$$R' \ : \ A \leftarrow -SB_N + CA$$

the second sine-cosine generation of the algorithm of Section 4.5.2 can be eliminated, with the rows of the rotated matrix being output (lowest rows first) from the B registers of row n of the ISA into the southern data buffer. Here, the four

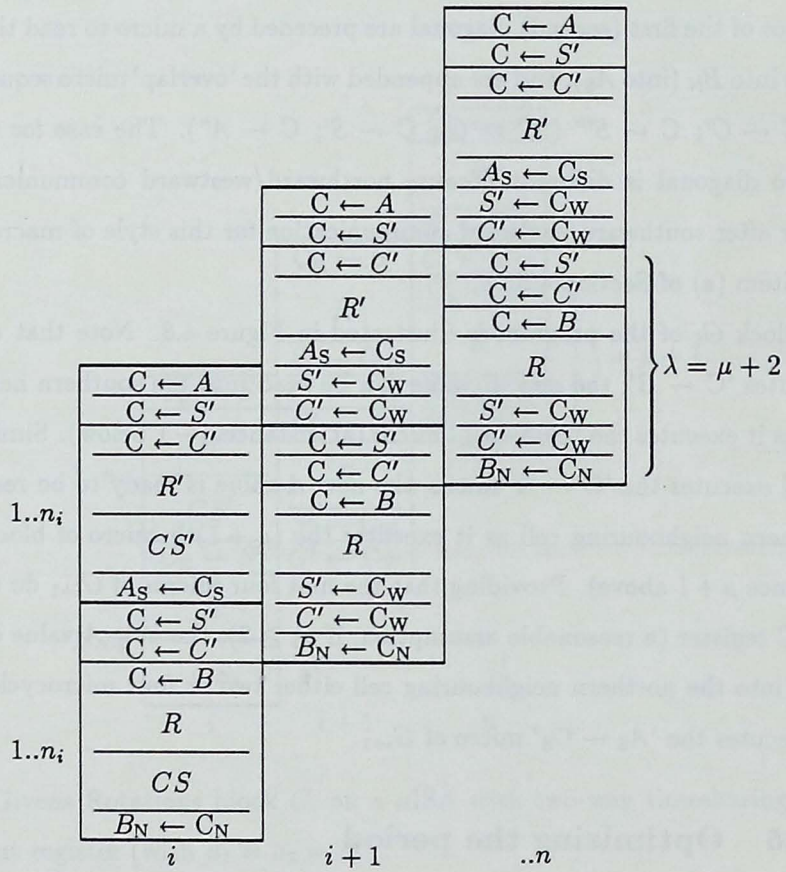


Figure 4.8: Givens Rotations block on a μISA with three-way timesharing of the single output register (illustrated for $\mu_1 = \mu_2 = 2$)

output communication register mode is assumed with the symbolic A register being bound to a fresh non-communication register but otherwise the symbolic register assignments given for the $\bar{\eta} = 1$ algorithm of Section 4.5.3 hold.

Block G_i now operates on all rows (but only rows n_i to n perform useful rotations), relying on the fact that the (inactive) rows 1 to $n-i$ have their A registers cleared by the preceding blocks. Hence, with B_N for row 1 held at 0, the B registers for rows 1 to $n-i$ will retain their zero values by the R instruction. Block G_i operates on columns i to n, as before.

To optimize the period, ie. minimize the number of microcycles per block, the G_i block could consist of a macro diagonal with '$CS;R$' (for column $j = i$) and '$\Rightarrow;R$' (for columns $j > i$) followed by a shorter macro diagonal with R'' (for columns $j \geq i$). The first macro diagonal is of length $\lambda_1 = \mu + 1$ with $\mu = \mu_1 + \mu_2$, and the second is of length $\lambda_2 = \mu_2$. This is safe because there is now no communication between consecutive blocks and because the second macro diagonal now performs no communication. This scheme has no contraflow, making it easily partitionable (see 2.5.2.2).

Alternatively, the result matrix could be shifted upwards 'on the fly', so that the result matrix is left in row-major order in the μISA after the execution of rotation blocks G_1 to G_n (with the extra block G_n being required to perform the last upward shift)[10]. The result matrix is stored in the symbolic register B', which has the binding:

$$B' = \mathrm{C}'_{\mathrm{N}}$$

The program for block G_i is illustrated in Figure 4.8, with the rows selecting each sub-macro diagonal being indicated to their left. Column k of the southern data buffer reads A^n_{ik} after cell (k, n) executes the last micro of the 'R' sub-macro; this is read back $\bar{\eta} > 0$ microcycles later into the B' register of cell (n, k) by the '$\uparrow_{B'}$' micro. In cell $(n - i', k)$, the '$\uparrow_{B'}$' micro reads into B' the value of $A^n_{(i-i')k}$ from cell $(n - i' + 1, k)$ (which was put there by block G_{i-1}).

[10]Such 'on the fly' shifting of nontransmittent data is required for μISAs using data interfaces proposed by Lang (see Figure 2.10).

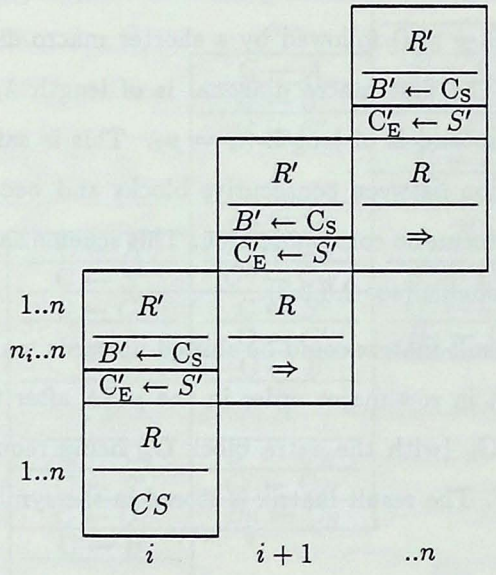


Figure 4.9: Optimized Givens Rotations block G_i on a normal μISA leaving result matrix in row-major order (with $\mu_1 = \mu_2 = 2$)

4.5.6 Factoring out the sine-cosine generations

A final variation of the Givens Rotations algorithm is to confine the expensive hardware to perform the *sine-cosine* generations to a linear array connected to (the western edge of) column 1 of the ISA (note that having the *sine-cosine* array *connected* to, as opposed to *part of*, column 1 of the ISA costs an extra microcycle in communication latency). Thus, the program now shifts west the A-values at the end of each block.

Unfortunately, this westward shifting creates a delay of 2σ microcycles between when cell (i, j) produces its own A-value and when it receives the A-value from its eastern neighbour. For the normal μISA implementations, $\sigma \geq \mu_1 + \mu_2$, making this delay large.

However, all variations on the Givens Rotations algorithm can be implemented on a $(\mu, 1)$ wavefront μISA (with slightly lower period). This μISA could then implement the westward shift with only a $2\sigma = 2$ microcycle delay.

Alternatively, a normal μISA using data interleaving could efficiently implement this westward shift. In this case, the μISA would execute each block G_i on two independent matrices in turn, with the first matrix's A-value being read from the east after G_i is executed on the second.

4.6 Simulating an arbitrary instruction granularity ISA

One of the main applications of a (normal) μISA (of modest microinstruction granularity) is its area-time efficient and flexible simulation of an ISA of high instruction granularity, or, in more general terms, supporting a macro-level wavefront programming model. The implementation of Section 4.3 is too general to support this directly: it is more efficient to implement the extra contraints required in software rather than hardware.

Section 4.6.1 gives constraints on the structure of macros used to efficiently simulate, ie. *emulate*, ISAs in this sense. Based on these rules, Section 4.6.2 proposes a macro structure using an appropriate choice of the microcode skew and

macro overlap to emulate high instruction granularity. While such a structure is not always optimal (although generally, it is close to being optimal), such a discipline makes the μISA significantly simpler to program. It is also easy to efficiently emulate more powerful topologies such as hex-connected ISAs on the μISA. Such macro structures have been used in the normal μISA programs presented already in this chapter; a final example, a variation on the Dynamic Time Warping algorithm, is derived using this macro structure (Section 4.6.3). More powerful extensions for this macro structure, using enhanced communication register nodes and data delay queues, are outlined in Section 4.6.4.

While this section is not required for the reading for the rest of this thesis, an intuitive understanding of it is helpful in understanding the timing and programming of the μISA.

4.6.1 Constraints on the macros' internal structure

Consider the sequence of (macro) diagonals (of lengths λ'', λ' and λ respectively):

$$D''; \; D'; \; D$$

passing through a (normal microprogrammed) ISA, with a particular value v to be read (between micro positions r_f and r_l, with $1 \leq r_f \leq r_l \leq \lambda$) by D.

If v is read from the west or north [south or east] neighbouring cell, it must reside in the communication register of that cell as it executes D [D''] (during micro positions w_f to w_l, where $1 \leq w_f \leq \lambda$ [w''_f to w''_l, where $1 \leq w''_f \leq \lambda''$]). Now (w_l+1) $[w''_l+1]$ corresponds to the position of the first micro that overwrites this communication register; thus $w_l > \lambda$ $[w''_l > \lambda'']$ is possible.

Using the microprogramming timing rules (Section 4.4.2), the required constraints on r_f and r_l for v to be read are given by, for reading from the north/west and south/east directions respectively:

$$w_f - \mu + 1 \leq r_f \leq r_l \leq w_l - \mu + 1 \tag{4.10}$$

$$\mu + 1 - (\lambda' + \lambda'' - w''_f) \leq r_f \leq r_l \leq \mu + 1 - (\lambda' + \lambda'' - w''_l) \tag{4.11}$$

A useful special case of these rules, adopted in the μISA programs presented here, is when v is read once only (ie. $r_l = r_f$).

Example: correctness of the microprogrammed transitive closure program.

For program TransClos, (see Section 3.5.2 and Figure 3.5(a)), no communication registers are timeshared ($\lambda = \mu$).

Here, all macros are constructed so that the value of C'_S (C'_E) is read at a fixed micro position $r_f = 1$ ($r_f = 2$); this value is not overwitten until the micro at position $w'_l = 1$ ($w'_l = 2$) in the next macro is executed. Now, by the definition of w_l, $w_l - \lambda + 1 = w'_l$, and $r_l = r_f \leq w'_l$ and hence the inequality:

$$r_l \leq w_l - \mu + 1$$

of equation (4.10) holds. Note that the constraint $w_f - \mu + 1 \leq r_f$ automatically holds for $\lambda = \mu$, since here $1 \leq r_l, w_f \leq \mu$.

This demonstrates that the macro structure of the TransClos program correctly emulates the corresponding ISA program implementing each macro as a single instruction.

With the leftmost two inequalities of equations (4.10, 4.11) reduced to equalities, the macro structure allowing timesharing is proposed in the following section.

4.6.2 A macro structure to emulate high instruction granularity

This section considers the normal μISA emulation of an ISA program in which ($\eta = \bar{\eta} + 1$) values are communicated during a single instruction. This can be achieved by using a disciplined macro structure, which usually gives a solution close to the optimal period. This structure makes use of the macro overlap parameter $\bar{\eta}$, with arbitrary computation granularity achieved by the parameter μ. Consider again the macro sequence '$D''; D'; D$' of Section 4.6.1. The macro structure (for D) is given by the micro sequence:

$$R_1; \ldots; R_\eta;\ C_1; \ldots; C_c;\ W_1; \ldots; W_\eta$$

with $\lambda = \mu + \bar{\eta}$ and $\mu = \eta + c + 1$. The values $v_1, \ldots, v_\eta$ are placed in the communication registers by micros $W_1, \ldots, W_\eta$; these values can be read in only by the $R_1, \ldots, R_\eta$ micros[11].

If reading from the south or east is used, it is also required that $R_1, \ldots, R_\eta$ do not modify any communication register read in by a north or west cell (so as not to overwrite a value placed there by the previous macro). Otherwise, it is possible to allow an optimization with $c < 0$: this means that the last $(-c)$ R-micros coincide with the first $(-c)$ W-micros (and hence are of the form: "communication register" $\leftarrow f$("input register") — see the first example below).

Examples.

- The LCS program (see Section 4.2.1) can be implemented on a normal μISA with the slightly longer macro length $\lambda = \mu + 2 = 5$. Here $\eta = 2$ and $c = -1$, and the macro (using the four output communication register mode for the sake of clarity) has the structure:

$$\begin{aligned} R_1; R_2 &= \mathrm{A} \leftarrow \mathrm{A} * \mathrm{C_N};\ \mathrm{A} \leftarrow \max(\mathrm{A}, \mathrm{C_N}) \\ R_3/W_1 &= \mathrm{C'_S} \leftarrow \mathrm{C_W} \\ W_2; W_3 &= \mathrm{C'_S}, \mathrm{A} \leftarrow \max(\mathrm{A}, \mathrm{C'_S});\ \mathrm{C'_E} \leftarrow \mathrm{A} \end{aligned}$$

  This macro structure is shown in Figure 4.10(c).

- For the transitive closure program of Section 4.2.2, $\lambda = \mu + 1 = 6$, $\eta = 2$ and $c = 2$. Here, the macro I' has the structure:

$$\begin{aligned} R_1; R_2 &= \mathrm{C}, \mathrm{A''} \leftarrow \mathrm{C_N};\ \mathrm{A'} \leftarrow \mathrm{C_W} \\ C_1; C_2 &= \mathrm{C} \leftarrow \mathrm{C} \wedge \mathrm{A'};\ \mathrm{C} \leftarrow \mathrm{C} \vee \mathrm{A} \\ W_1; W_2 &= \mathrm{C} \leftarrow \mathrm{A''};\ \mathrm{C} \leftarrow \mathrm{A} \end{aligned}$$

  and similarly for the macro I. A similar macro structure is shown in Figure 4.10(a).

[11] Northward or westward communication occurs over two macros; in this case, a macro may have a different number of read/write macros. For simplicity, the above macro structure assumes otherwise; however, these ideas can be easily extended to cover this case.

- The Givens rotation program of Figure 4.8 uses $\eta = 3$ and $c = \mu_1 - 2 + \mu_2$. The first macro of block G_i on columns $j > i$ has the structure:

$$\begin{aligned} R_1; R_2; R_3 &= B_{\mathrm{N}} \leftarrow \mathrm{C_N};\ C' \leftarrow \mathrm{C_W};\ S' \leftarrow \mathrm{C_W} \\ C_1; \ldots; C_c &= (\mathrm{NoOp})^{\mu_1 - 2};\ R \\ W_1; W_2; W_3 &= \mathrm{C} \leftarrow B;\ \mathrm{C} \leftarrow C';\ \mathrm{C} \leftarrow S' \end{aligned}$$

Various cases for this macro structure are illustrated in Figure 4.10. This structure is particularly suited to reading several values from the north or west, since it is assured that for each value $r_l = r_f = w_f - \mu + 1$ (cf. equation (4.10)). For reading from the south or east, timeharing occurs less often in practice. Two cases must then be considered:

(a) no timesharing of the northward or westward communication registers (cf. Figure 4.10(a,c)). In this case, the preceding macro D' can have the usual length of $\lambda' = \lambda$, providing its first η (read) micros do not overwrite the north or east communication registers. This means that $w''_l \geq \lambda'' + \eta$, allowing (cf. equation (4.11)):

$$r_f = r_l \leq 2 \leq \mu + 1 - (\lambda' + \lambda'' - w''_l)$$

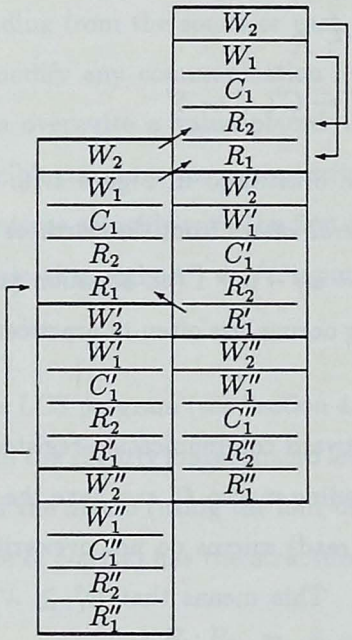
and hence in this case, the value communicated north should be read by either R_1 or R_2 and be written on the last micro of D''.

(b) timesharing of northward or eastward communication registers. Here, the preceding macro D' is shortened and hence cannot read from the north or west (in practice, D' is often a 'NoOp' or it performs no communication; see Figure 4.10(b)). Setting r_f to be the corresponding macro read position for w_f, ie. $r_f = \eta - (\lambda'' - w''_f) + 1$, and $w''_l = w''_f$, equation (4.11) requires λ' to be:

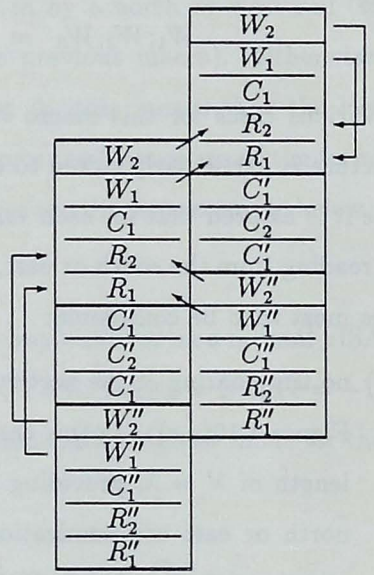
$$\lambda' = \mu - \eta = c + 1$$

It can be seen from Figure 4.10(a,b) how a hex-connected ISA can be emulated using timesharing techniques with $\eta = 2$.

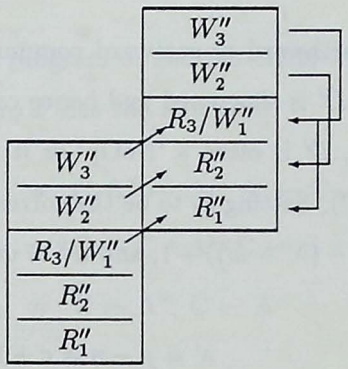
A methodology for using the μISA to emulate an ISA of arbitrary instruction granularity is then:



(a) reading from N/W with $\eta = 2, c = -1$



(b) reading from S/E with $\eta = 2, c = 1$



(c) reading from N/W with $\eta = 2, c = 1$

Figure 4.10: Macro structure for ISA emulation using timeharing

1. write the ISA program, assuming arbitrary instruction granularity.
2. independently translate the ISA diagonals into macros (in terms of a fixed μISA microinstruction set) in the above format. From the largest macro, determine the value of μ for the program.
3. using this value of μ, 'pad' the macros (if necessary), so that a macro communicating η values (to the south or east) will have length $\lambda = \mu + \eta - 1$. Macros performing no communication can be of any length unless preceding a macro reading several values from the south or east.

These last two steps are amenable to automation, and hence this technique can help in programming for the ISA model by abstracting from its instruction set. While some padding (with 'NoOp's) is required, this is still more area-time efficient than designing ISA cells with various large and complex instructions which are only used a small fraction of the time.

4.6.3 Microprogrammed Dynamic Time Warping

The macro structure of 4.6.2 is now applied to a two-dimensional systolic implementation of the Dynamic Time Warping problem with local continuity constraints [26, pp583-584]. This problem is in the same class as the LCS problem of Section 4.2.1; however, it requires a much higher instruction granularity.

This problem uses the $m \times n$ match matrix M (produced by a cartesian product of element-wise matches from two test patterns of lengths m and n) and the integer weights $w_1, \ldots, w_5$. It can be formulated as finding $M'_{m,n}$ where for $-1 \leq i \leq m$ and $-1 \leq j \leq n$:

$$M'_{i,j} = \begin{cases} 0 & \text{if } i \leq 0 \text{ or } j \leq 0 \\ \min \left\{ \begin{array}{l} M'_{i-2,j-1} + w_1 M_{i-1,j} + w_2 M_{i,j}, \\ M'_{i-1,j-1} + w_3 M_{i,j}, \\ M'_{i-1,j-2} + w_4 M_{i,j-1} + w_5 M_{i,j} \end{array} \right\} & \text{otherwise} \end{cases}$$

While this problem appears to require a 'super hex'-connected ISA, it can easily be implemented on a rectangular mesh ISA having three communication registers (denoted C, W and NW) and internal registers A, NNW, and NWW (initialized to $w_3 M_{i,j}$, $w_1 M_{i-1,j} + w_2 M_{i,j}$ and $w_4 M_{i,j-1} + w_5 M_{i,j}$ respectively). The instruction

performing this computation of $M_{i,j}$ in cell (i,j) is then given by:

$$\begin{aligned} \mathrm{W} &\leftarrow \mathrm{C_W}; \\ \mathrm{NW} &\leftarrow \mathrm{W_N}; \\ \mathrm{C,A} &\leftarrow \min\{\mathrm{NW_N} + \mathrm{NNW},\ \mathrm{W_N} + \mathrm{A},\ \mathrm{NW_W} + \mathrm{NWW}\} \end{aligned}$$

This can be emulated on an $m \times n$ normal μISA (with $\mu = 5$) using a macro diagonal of length $\lambda = 8$ and macro overlap $\bar{\eta} = 3$. Here, the registers W and NW are now internal registers, with C remaining a communication register. A timesharing factor of $\bar{\eta} + 1 = 4$ arises since the values $M'_{i-2,j-1}, M'_{i-1,j-2}, M'_{i-j,j-1}$ and $M'_{i,j-1}$ must be read into registers NNW, NWW, NW and W respectively. The program's macro diagonal is given in Figure 4.11 (all selectors are 1's).

The macro structure is a special case of the $c = 0$ structure:

$$R_1; R_2; R_3; R_4; W_1; W_2; W_3; W_4$$

with W_1 and W_2 only modifying register A, and leaving $M_{i-1,j-1}$ in the C register (set there by micro R_3) to be read by the south and east cells during micros R_1 and R_2 respectively (see Figure 4.11). Note also that R_3 reads the value of the C register of the north neighbour, which was set there by W_3.

A similar macro can be used to set up $M_{i,j}, M_{i-1,j}$ and $M_{i,j-1}$ in cell (i,j): this also requires the communication of four values. A non-communicating macro (of unrestricted length) would then be required to initialize NNW, NNW and A.

It remains to be said that a preliminary ($\lambda = 12$) solution was derived at once (by the author) from applying the macro structure of Section 4.6.2. A principle of ordering micros used was that the values of most distant origin should be read in first. From this, some optimizations were easily applied. As a comparison, the much simpler LCS program of Section 4.2.1 was derived in much longer time (by the author), without using this technique of macro structuring.

4.6.4 Extensions for the microprogrammed emulation of ISAs

So far, techniques for implementing communication register timing on the μISA have assumed a BISA-like instruction set and communication register modes.

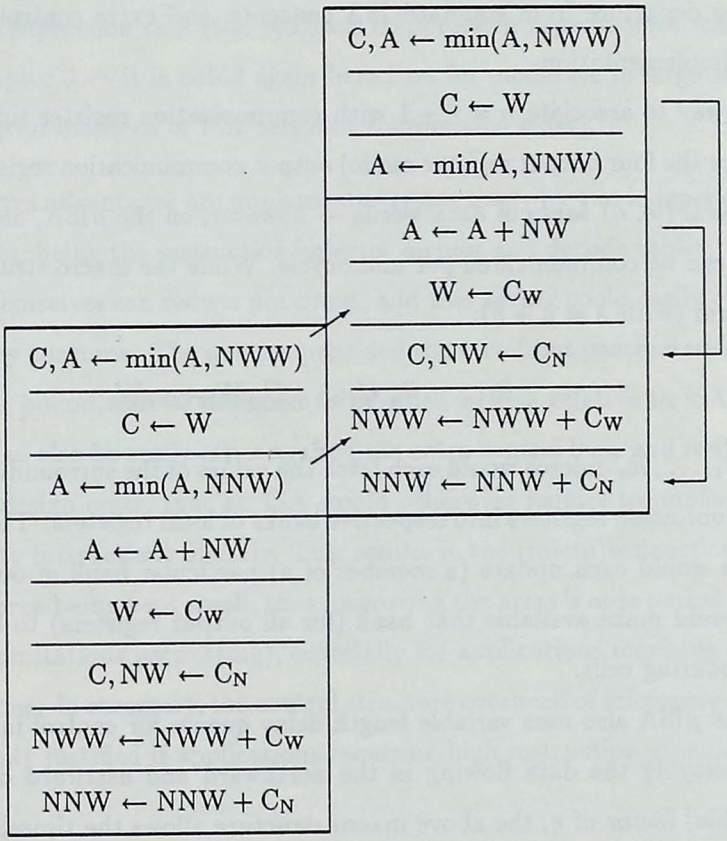


Figure 4.11: Dynamic Time Warping program on a normal μISA using a single output register C, with $\lambda = 8, \mu = 5$ and $\bar{\eta} = 3$

This has resulted in the value η being associated with the number of values read in per macro rather than the timesharing factor of communication registers (these coincide for the one output register mode). Also, it results usually in a small loss in efficiency in shuffling data in and out of communication registers. The macro structure of Section 4.6.2 permits easy timesharing for data flowing southwards and eastwards, but is not so convenient for data flowing in the other direction. This section outlines how the μISA can be extended to overcome these problems. These extensions were not developed earlier in this chapter since they require a further departure from standard ISA concepts and extra control structures for their implementation.

A way to associate $\eta = \bar{\eta} + 1$ with communication register timesharing is to use (for the four output register mode) output communication registers comprised of several (ie. η) *banks* of data words — however, on the μISA, only one bank of these can be communicated per microcycle. While the macro structure is still of the form (with $\lambda = \mu + \bar{\eta}$):

$$R_1; \ldots; R_\eta; C_1; \ldots; C_c; W_1; \ldots; W_\eta$$

the '$R_1, \ldots, R_\eta$' micros would each latch the values of the surrounding neighbours' communication registers into respective banks of local registers. The '$W_1, \ldots, W_\eta$' micros would each update (a member of a) particular bank of output registers: this would make available that bank (for all output registers) to be read by the neighbouring cells.

The μISA also uses variable length delay queues for control information. By also delaying the data flowing in the northward and eastward directions by a (variable) factor of η, the above macro structure allows the timesharing for data flowing in any direction. This will yield a cleaner and more powerful programming methodology for the ISA model using the μISA.

4.7 Conclusions

The ISA is a considerable improvement in terms of flexibility over the systolic array model, but still lacks flexibitity in applications using nontransmittent data

which may require high instruction granularity. This can be solved by using the dynamic form of vertical microprogramming proposed in this chapter. The dynamic aspect gives the μISA great flexibility, in that tailor-made, optimized microcodes can be chosen for each individual program. Vertical microprogramming is suitable because it is the simplest and most area-efficient method of microprogramming. While retaining the advantages of traditional microprogramming, the microprogrammed ISA has the extra advantages of increasing the effective instruction granularity (and hence the power) of the ISA and of achieving a constant program compression rate (the program compression aspects were mainly discussed in Chapter 3 — it is noted again here that for moderate to large array sizes, more powerful methods of ISA program compression exist).

These two extra advantages are implemented respectively by two independent control structures, being the instruction/selector queues and decode tables. The decode tables themselves can reduce pin count, and this saving could easily offset the area that they consume. The queues, provided the wavefront parameters have a constant upper bound, can be designed to be small in area relative to ISA cell size. This area can also be easily offset against the extra control logic and routing, not to mention design costs, that an ISA would otherwise require to implement coarse granularity instructions directly. This results in the (micro)instruction set and overall cell area being kept small, thus improving the array's area-period performance (I/O limitations permitting), especially for applications requiring only simple instructions. In summary, the control structure overhead of microprogramming can be easily justified if applications requiring high instruction granularity are used.

The μISA can be regarded as a new model of mesh, and has been shown to be more powerful than the ISA (and hence the SIMD mesh). Here, 'more powerful' means that an arbitrary ISA program can be transformed into an equivalent μISA program whose period is of the same order. Similar observations were shown to apply for the normal μISA simulating a normal μISA program of smaller wavefront parameters. These transformations also have practical significance in that for short programs, applying these transformations can be more efficient than

changing the wavefront parameters (incurring a small $O(n)$ delay) and executing the original program directly.

However, the main application of the μISA is as an emulator of arbitrary instruction granularity ISAs, ie. supporting a macro-level wavefront programming model. For this purpose, the μISA is restricted to the *normal* μISA, using the macro structure discipline of Section 4.6, which has been demonstrated to give efficient implementations of algebraic path algorithms based on the Kung-Gubias-Thompson systolic transitive closure algorithm, Dynamic Time Warping algorithms, and matrix QR factorization algorithms. On an ISA lacking the powerful and specialized instructions required, these algorithms become more complicated and longer, if they can be efficiently implemented at all. There is then considerable scope for worthwhile application of the μISA.

This macro structure discipline permits the timesharing of communication registers (and hence abstraction from the communication part of instruction granularity). This discipline is well developed for data flowing in the south and east directions; however, its development (possibly requiring extra control structure support) for data flowing in the opposite directions is an issue for future research. From this discipline, a methodology for macro-level programming of the ISA is outlined — this involves deriving the program for an ISA program of ideal instruction granularity, and then using this discipline to derive the macro structure for a fixed microinstruction set μISA. The first of these stages especially is an issue for future research.

However, non-normal μISA implementations, with more freedom in their macro structure, may yield a slightly more efficient solution. This represents a tradeoff between programming efficiency and programming effort.

A direction for future work on the μISA is to investigate the use of data delay queues. Such queues are already used in systolic architectures such as the CMU Warp Processor [1, pp1524-1525], and are expected to improve the power and flexibility of the μISA model. They also should simplify the emulation of the ISA when efficient timesharing of data flowing north and west is required: here the macro overlap parameter η would give the lengths of the northward and westward

data queues.

Another direction for future research is to extend the μISA concepts, particularly the macro structure of Section 4.6, to more general meshes capable of supporting the macro-level wavefront programming model, such as Processor Arrays. This is partially examined in Chapter 7. Since an orthogonally-connected μISA has been shown to efficiently simulate programs using more general topologies[12] (on the macro level), it can be concluded that a general-purpose mesh should use (wavefront) microprogramming techniques on a basic orthogonal-connection topology to simulate more powerful (and expensive but only occasionally used) topologies.

The microprogrammed ISA is an extension to the ISA which increases the ISA's relative merit over SIMD meshes. It can be seen as a further step towards making the ISA model suitable for large-scale, general-purpose matrix computations. However, for this purpose, it also is necessary to implement 'optimal' (wavefront-based) program compression methods on the μISA — this is the concern of the next chapter.

4.A Appendix: Variable length microinstruction queues for VLSI

A variable length queue design, suitable for efficient VLSI implementation, is proposed here. This design is suitable for moderate ranges of μ and σ, ie. $1 \leq \mu, \sigma \leq 10$ (this would be sufficient to make a ten-fold increase in the μISA's effective instructional power). This design consists of a shift register (each cell consisting of a pair of gated inverters), as is shown in Figure 4.12. The half cells, H_1 and H_2 ensure that the minimum delay through the whole structure is 1 microcycle. To decrement the queue length, the gates of the lowest 'unopened' cell (Q_4 in Figure 4.12) are opened; and conversely to increment the queue length. This is implemented by a 1-bit stack adjoining the queue: a '1' in a stack cell 'opens' the adjoining queue cell. Thus a queue decrement corresponds to a *pop* on the stack

[12]eg. hex-connected arrays, in the case of Dynamic Time Warping algorithms.

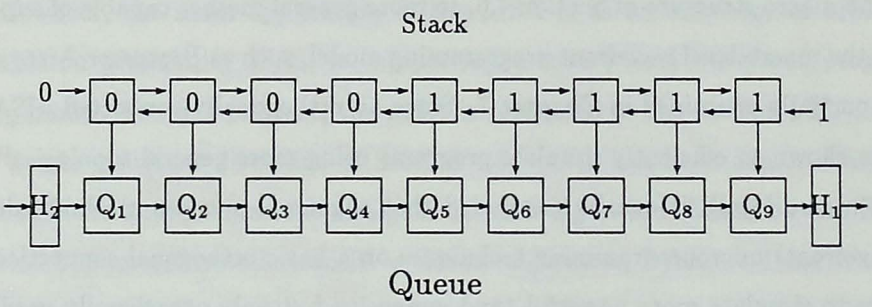


Figure 4.12: Variable length queue corresponding to $\mu\ (\sigma) = 4 + 1 = 5$

(with a '1' underflow); and conversely for queue increments.

The delay δT through a series of δ such cells cannot be ignored. For nMOS technology, using minimum size (8:1) inverters, and an estimated stray capacitance ratio[13] of 1.5, $T \approx 1.5(1+8+1+1)\tau = 16.5\tau$ [40, pp10-12], where τ is the minimum sized switch delay. Since the arrangement of Figure 4.12 requires the delay δT to be within a half microcycle, which could be as low as 100τ for a fast μISA, the maximum value of δ would be 6, short of the target of 9. This could be rectified by moving the half-cell H_2 to below the cell Q_3.

Providing the queue could be overlaid onto the passive instruction (selector) path through a μISA cell, it would add negligible area to the layout. The main increase in area would be due to the stack, but this would still be much less than a full decode table. Hence, provided large values of $\mu\ (\sigma)$ are not required, this implementation of variable length queues for microprogramming requires only a small hardware overhead.

[13]The ratio is low, since the cells can be put very close together

Chapter 5

Program Compression by ISA Diagonals

5.1 Introduction

Motivations to compress ISA programs include reduction of the critical host to ISA input bandwidth and overall reduction in the ISA system hardware [53] (see Section 3.3). These factors are particularly important for large ISAs composed of fine- to medium-grained cells. Chapter 3 discussed the SISA method of ISA program compression [31], which has a very simple implementation but achieves only a constant factor of program compression and reduces the flexibility of the ISA. Other methods, such as ISAC, can 'optimally' compress essentially all ISA programs. ISAC can achieve this performance of compression since it exploits the few different types of regularities found in ISA program matrices. ISAC has a drawback, however, in that it requires $O(n \log n)$ area control structures for its implementation.

The ISA is an architecture suited to performing computations expressed as sequences of diagonals or wavefronts. For such computations, the diagonal is an appropriate 'semantic unit' (see Chapter 6) for a high-level ISA language. The Subroutining program compression method of Section 3.6 gives such a language (similar to the ISA language proposed by Lang [35, Ch.4]), having an 'optimal' program compression rate and showing potential for a very efficient implemen-

tation. Also, the advantages of microprogramming the ISA, as demonstrated in Chapter 4, make it essential that the easy compatability of this implementation with ISA microprogramming be developed. However, still in question is the flexibility of this language in terms of on which ISA programs it can efficiently implement program compression. This chapter investigates the issues of the implementation and flexibility of the Subroutining method, choosing a more primitive representation of Subroutining that is convenient for discussing program compression concepts.

The *ISA Diagonal Language* (ISADL) introduced here is a low-level, diagonal-based ISA language. Its constructs are equivalent to the lower-level constructs of Subroutining, so that all program compression properties of ISADL carry over to Subroutining. A subset of ISADL, sufficient to implement 'optimal' program compression for most ISA programs (of our experience), has an implementation 'factoring out' much of ISAC's control structures. ISADL is then proposed here as a practical compromise between the SISA and ISAC concepts, in which the range of algorithms intended to be run on a particular ISA determines the control structure overhead. Hence, in this chapter, elegance is sacrificed for implementation efficiency and more attention to detail is given.

The concept of ISADL is based on an observation similar to that motivating the SISA: that the diagonals of ISA programs have a surprisingly simple structure. Two approaches for implementing ISADL arise:

- pass sufficient information across the *diagonal restorer* to completely construct each diagonal as it enters the ISA (cf. the SISA). This has the advantage of requiring very little storage and being very powerful, at the expense of a high I/O bandwidth (ie. several instructions and $\log n$-bit integers would need to be processed and passed for the more complex diagonals).

- load the (few) diagonal (pattern)s for an $\Omega(n)$ subprogram into the *diagonal restorer* before use, and then perform (systolic) operations on these to restore the diagonals as they are sent to the ISA. This has the advantages of reducing the I/O bandwidth and simplifying the logic of the diagonal restorer, at the expense of extra storage.

Thus, a tradeoff arises between the internal storage and I/O bandwidth of an ISADL implementation. For simplicity, ISADL is formulated favouring higher internal storage, and modifications are then given to reduce this at the expense of extra I/O bandwidth.

Some simple examples of ISADL programs are given later in this section. Section 5.2 gives a formal definition of ISADL, in terms of the ISA program matrices that it represents. A sub-language of ISADL, having an efficient program compression implementation, is defined in Section 5.2.1. Section 5.3 gives an overview of the control structures required to implement program compression for this sub-language. These control structures are broken down into two components: an $O(\log n)$ area *diagonal sequencer*, and an $O(n)$ area linear (instruction systolic) array called the *diagonal restorer*. This allows more general ISADL programs, microprogramming and subroutines to be easily incorporated (Section 5.4), as well as easy ISA expandability. Section 5.4.1.3 also discusses how ISADL programs that are difficult to implement program compression on can be recoded, or replaced by equivalent programs. Thus, Sections 5.3 and 5.4 combined describe how ISADL program compression can be made practical. The success of this is evaluated in Section 5.5, which estimates the hardware overhead and program loading performance of ISADL for a typical range of ISA programs. The compression rate of ISADL is 'optimal', being essentially the same as that of the Subroutining program compression method (see Table 3.1), and need not be further discussed in this chapter. Conclusions are given in Section 5.5.3.

This chapter's contribution is to develop, in considerable detail, 'optimal' program compression for diagonal-based ISA languages, so that their implementation will be highly beneficial and in every way feasible for a moderate to large-scale ISA system. If the reader is already convinced that this is possible, this chapter need not be read in detail. An understanding of the basic concepts and expression of ISADL (from this section) will also be useful, but not essential, for the reading of Chapter 6, in which proof methods for diagonal-based ISA languages are presented. The basic implementation concepts of ISADL (from Section 5.2.1) are briefly considered for the implementation of program compression for processor

arrays in Chapter 7.

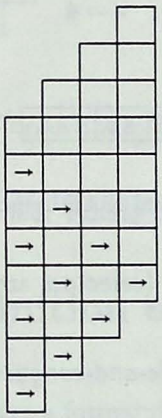
Examples of ISADL encodings

The basic concepts and features of ISADL are illustrated here. The LoadMatD program[1] of Figure 5.1(a) loads an $n \times n$ matrix from the west data buffer into the respective C registers of the ISA. The instruction part of the program consists of $2n$ diagonals, the odd ones containing both '$\rightarrow$' and 'NoOp' (o) instructions. These are separated by a *boundary*, initially at column n, which shifts left between successive iterations. In ISADL, an instruction repeated across successive columns in a diagonal need only be specified by the positions of its (left and right) boundaries and the instruction itself. This provides *horizontal* program compression, and leads to the encoding of Figure 5.1(b). Note that the bottom diagonal has a 'NoOp' (o) sub-diagonal whose left and right boundaries are equal, in this case signifying that there are $4 - 4 = 0$ repetitions of the 'o' instruction ending at column 4. This is done so that this sequence of (8) diagonals can be easily expressed as repetitions of a small sequence of (2) 'generalized' diagonals, as shown in Figure 5.1(c). This provides *vertical* program compression. The passive (even) diagonals are required for the implementation of ISADL, as discussed in Section 5.3. The ISADL encoding of the general LoadMat program is given in Figure 5.1(d).

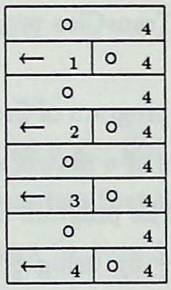
For the sake of conciseness, the notations $x' = x + 1$ and $\bar{x} = x - 1$ are used in this chapter.

TransClos, a pass of the microprogrammed implementation of the Kung-Gubias-Thompson transitive closure algorithm of Section 3.5.2, has a similar encoding. Its instruction matrix resembles a skewed identity matrix, and can also be regarded as n iterations of a diagonal in which the boundaries between the I and I' macros shift right one unit each iteration (the boundaries at the kth iteration are at $k-1$ and k, for $1 \leq k \leq n$). Here, the leftmost sub-diagonal has an implicit left boundary of 0. The instruction matrix and the ISADL representation are given in Figure 5.2.

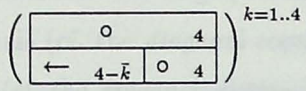
[1] cf. the program LoadMat of Section 3.7.1.



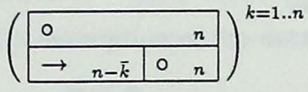
(a) matrix for a 4×4 ISA



(b) ISADL encoding of matrix, $n = 4$

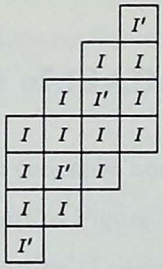


(c) introducing iteration, for $n = 4$

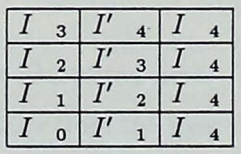


(d) general ISADL encoding

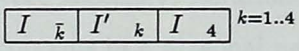
Figure 5.1: LoadMatD program, instruction part (selectors are '1's)



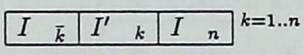
(a) matrix for a 4×4 ISA



(b) ISADL representation of matrix, $n = 4$



(c) introducing iteration, $n = 4$



(d) general ISADL encoding

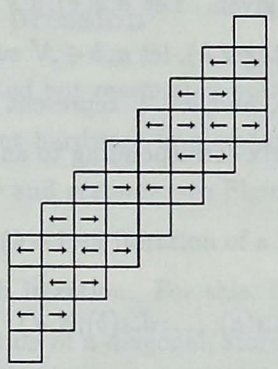
Figure 5.2: TransClos program, instruction part (selectors are similar)

The RowRev program of Section 3.7.1 uses a *divide-and-conquer* strategy[2] for reversing the rows of a matrix stored in the C registers of an $n \times n$ ISA, where n is a power of 2. The program uses a sub-program, RotH_d, which swaps adjacent blocks of width $\frac{d}{2}$ by using $\frac{d}{2}$ horizontal ringshifts over a distance of d. This accomplishes a row reversal when repeated for $d = 2, 4, \ldots, n$. This results in a surprisingly efficient algorithm to reverse the rows of a matrix, and is expressed in ISADL as:

$$\mathrm{RowRev} \;:\; (\mathrm{RotH}_{2^k})^{k=1..\log n}$$

introducing a generalization of the *repetition* construct. The RotH_d sub-program consists of two diagonals, which have boundaries that do not shift and a basic pattern of *recurrence width* d, repeated across the n columns of the ISA (it is assumed here that d divides n). Its instruction matrix and ISADL encodings are illustrated in Figure 5.3. Note that $(\ldots)^x$ is a shorthand for $(\ldots)^{l=1..x}$, emphasizing that the diagonals' boundaries are not shifted between iterations.

[2]This is also used in ISA implementations of sorting [46, 51].



(a) matrix, for an 8×8 ISA, $d = 4$

←	3	o	4
o	1	→	4
←	3	o	4
o	1	→	4

(b) ISADL encoding of matrix, $d = 4$ and implicitly $n = 8$.

$$\left(\begin{array}{|c|c|c|c|} \hline \leftarrow & d-1 & o & d \\ \hline o & 1 & \rightarrow & d \\ \hline \end{array} \right)^{\frac{d}{2}}$$

(c) general ISADL encoding

Figure 5.3: RotH_d program, instruction part (selectors are '1's)

5.2 Definition of ISADL

This section gives a formal definition of ISADL for an $n \times n$ ISA, which not only defines the language in terms of the ISA program matrices that it represents, but also provides a specification of its implementation of Section 5.3. This section also introduces the main concepts of ISADL that are required for its efficient implementation. In particular, the description of an ISADL program can be broken down into two largely orthogonal concerns: a description of the sequence of diagonals (cf. the *diagonal sequencer*) and a description of the details of each diagonal (cf. the *diagonal restorer*).

In this section, the following notation is used. $\mathcal{N}$ denotes the set of natural numbers, with $m, n \in \mathcal{N}$. The set of total mappings from a set N to the set X is denoted $N \rightarrow X$. Thus, $x\_s \in (1..n \rightarrow X)$ is a sequence of n elements of X and could represent a linear array, whereas $M \in (1..m \rightarrow (1..n \rightarrow X))$ could represent an $m \times n$ matrix. $<>$ denotes the empty sequence, $< x >$ denotes a unit sequence, and x_s^n denotes a sequence composed of n repetitions of the sequence x_s. For $x\_s, x\_s' \in (1..n \rightarrow X)$ and $i \in 1..n$, '$x\_s\ x\_s'$' denotes x_s *concatenated* with x_s' and $x\_s[i]$ is an alternative notation for $x\_s(i)$, emphasizing x_s is interpreted as an array. The set of all strings over a set D_s is denoted D_s^*, with ϵ denoting

the empty string. $\mathcal{I}$ is the set of ISA instructions [or selectors].

The formal definition of ISADL can now be given. Let $d\_s(k), d\_s'$ denote ISADL encodings (possibly depending on some integer k), let $a, b \in \mathcal{N}$ such that $a \leq b$, let D represent a single ISADL diagonal, and let ';' represent tabular concatenation. The ISA instruction [selector] matrix corresponding to an ISADL encoding d_s is given by mat_gen(d_s), where:

$$\begin{aligned} \text{mat\_gen}(\epsilon) &= <> \\ \text{mat\_gen}(d\_s(k)^{k=a..b}; d\_s') &= \text{mat\_gen}(d\_s(a); \ldots; d\_s(b); d\_s') \\ \text{mat\_gen}(D; d\_s') &= < \text{diag\_seq}(D, n) > \text{mat\_gen(d\_s)} \end{aligned} \quad (5.1)$$

Given that $i \in \mathcal{I}$, $n' \in 1..n$, $(n_1, \ldots, n_l)$ are integer expressions whose values satisfy $0 = n_0 \leq n_1 \leq n_2 \leq \ldots \leq n_l \leq w \leq n$ where n_l divides w, $D_1, \ldots, D_l$ are ISADL (sub-) diagonals, ISADL diagonals reduce to sequences of n instructions [selectors] as is given by:

$$\begin{aligned} \text{diag\_seq}(i, w) &= < i >^w \\ \text{diag\_seq}(\boxed{D_1 \;\; n_1 \mid \ldots \mid D_l \;\; n_l}, w) &= (\text{diag\_seq}(D_1, n_1 - n_0) \\ &\quad \ldots \text{ diag\_seq}(D_l, n_l - n_{l-1}))^{w/n_l} \end{aligned} \quad (5.2)$$

Equation (5.2) converts ISADL diagonals into a similar form to that used by the Subroutining program compression method. Example: referring to Figure 5.3(b),

$$\text{diag\_seq}(\boxed{\circ \;\; 1 \mid \rightarrow \;\; 4}, 8) = (< \circ > (<\rightarrow>)^3)^{8/4} = < \circ \rightarrow\rightarrow\rightarrow \circ \rightarrow\rightarrow\rightarrow >$$

These equations also give implicitly the allowed syntax of ISADL. Since these equations are reasonably simple, the examples of Section 5.1 are deemed to provide adequate illustration. Note that for the first iteration of the diagonal of program TransClos, $n_1 = 0$, so here no I instruction appears to the left of the I' instruction. Apart from a few similar cases, diagonals whose boundaries 'cross over' (see section 5.2.1) are best excluded from ISADL programs. The recurrence width n_l (and hence the array size n) is required to be a power of 2.

An arbitrary $m \times n$ matrix can be encoded using the tabular form of ISADL, with at most n entries in each row, and at most m different rows. In practice, these encodings can be represented in a constant space, due to natural redundancies in ISA program matrices.

5.2.1 Constraints on ISADL for efficient program compression

A restricted but reasonably powerful subset of ISADL is now described that has an efficient hardware implementation, the chief components being the *diagonal sequencer* and *restorer* (see Figure 5.4). The restrictions here make it possible to form the $(k+1)$th iteration of a ISADL diagonal from a simple systolic operation on its kth iteration. For this, it is sufficient that the *diagonal restorer*, on the kth iteration of a diagonal, stores the diagonal's current instruction sequence (in linear array form), with the elements to be updated for the $(k+1)$th iteration being appropriately marked. An updated element derives its value from its immediate left or right neighbouring element. This enables an extremely efficient and scalable implementation which also forms the basis of the extended versions of Section 5.4.

Consider the ISADL iteration $(\ldots D(k)\ldots)^{k=1..d}$. To allow the easy systolic construction of $D(k+1)$ from $D(k)$, where the diagonal $D(k)$ has boundary expressions dependent on k, $D(k)$ must satisfy the following constraints:

1. $D(k)$ is enclosed by no other iterations.
2. boundary expressions in $D(k)$ are of the forms c, $s-k$ or $a+k$, where c, s, a are expressions independent of k. Note that for

$$D(k) = \boxed{\cdots}\boxed{D'_{\ s-k}}\boxed{\cdots}$$

   any boundary expressions inside D' implicitly inherits the '$-k$' term from the $s-k$ boundary (and similarly for the $a+k$ boundary. See equation (5.2)).
3. if $D(k)$ has boundary expressions of the form $s-k$, the corresponding loop must contain at least two diagonals[3].

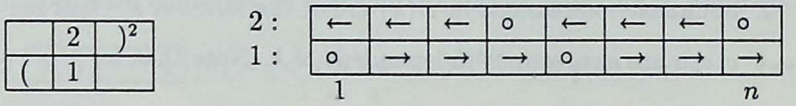
For this ISADL iteration to be considered 'well formed', it is also required that the boundary expressions of $D(k)$ do not 'cross over' for any $k \in 1..d$. ie. the

[3]cf. the LoadMatD program of Section 5.1.

sets $\{c\}, (a+1..a+d)$ and $(s-d..s-1)$ are pairwise disjoint. Exceptions are permitted for the sake of convenience for $c = n_l = s-1$ [$c = 1 = a+1$], where n_l has its meaning as in equation (5.2), eg. programs LoadMatD with $c = n$ [TransClos with $c = 1$] and RotH'_d with $c = n_l = 2d$. In these cases, the instruction 'hidden' by boundary 'cross overs' on the first iteration must be available from the immediate right [left]. In practice, other cases can be avoided by substituting equivalent loops.

The initial value $D(1)$ of the diagonal $D(k)$ is loaded into the *diagonal restorer*, with instructions on the boundaries being augmented with an appropriate '+' and/or '−' marker[4]. The restorer's boundaries (0th and $n+1$th elements) are given respectively by the instructions of the 1st and nth instructions $D(2)$ — these are required if $a+1 = 1$ or $s-1 = n$.

Each diagonal is given an index in the range $1..N_\mathrm{d}$, $N_\mathrm{d} > 0$. The respective sequences of diagonal indices to be produced (which may be augmented by extra information) is determined by the *diagonal sequencer*, which traverses a table storing the run-length encoding of the required sequences of diagonal indices and lookup tables. A visualization of these components for the program RotH_4 (cf. Figure 5.3) on an 8×8 ISA is given in Figure 5.4.



(a) diagonal sequencer (b) diagonal restorer

Figure 5.4: Implementation (initial state) for RotH_4 on 8×8 ISA

The diagonal sequencer's state is $d\_s \in D\_s^*$, where $D\_s = 1..N_\mathrm{d}$. The current

[4]In a few cases, both may be required. eg. by the above rule, for the diagonal:

$$D(k) = \boxed{\cdots \; s-k} \; \boxed{i_1 \;\; 1} \; \boxed{i_2 \;\; 2} \; \boxed{s+k} \; \boxed{\cdots}$$

the instruction at the $s-1$ boundary is marked '−' (to copy i_1 left upon each iteration of k) and those at the $s-1+1$ and $s-1+2$ boundaries are marked '+' (to copy i_1 and i_2 right). In this case, however, the rule must be extended so that the instruction at the $s-1+1$ boundary is *also* marked '−', (to copy i_2 left), cf. the *triangle merger* program of Appendix 5.B.2.

state of the diagonal restorer is $d\_r \in (0..N_d \rightarrow (1..n+1 \rightarrow D\_r))$, where $D\_r = \mathcal{I} \times \mathcal{P}(\{+,-\})$. The (i,j)th element of the instruction (selector) matrix produced by the combined states (d_s, d_r) is given by:

$$\text{mt\_gen}(d\_s, d\_r)[i][j]$$

where, for $d\_s, d\_s' \in D\_s^*$; $d\_r, d\_r' \in (1..N_d \rightarrow (0..n+1 \rightarrow D\_r))$; $k \in \mathcal{N}$ and $d \in D\_s$:

$$\text{mt\_gn}(\epsilon, d\_r) = <> \tag{5.3}$$

$$\text{mt\_gn}(d\_s^1\ d\_s', d\_r) = \text{mt\_gn}(d\_s\ \ d\_s', d\_r) \tag{5.4}$$

$$\text{mt\_gn}(d\_s^{k+1}\ d\_s', d\_r) = \text{mt\_gn}(d\_s\ \ d\_s^k d\_s', d\_r) \tag{5.5}$$

$$\text{mt\_gn}(d\ d\_s, d\_r) = < \text{apply}(d\_r[d]) > \text{mt\_gn}(d\_s, d\_r \oplus \{(d, \text{next}(d\_r[d]))\}) \tag{5.6}$$

where '$\oplus$' denotes *functional override*, and for each $j \in 0..n+1$:

$$\text{apply}(d))[j] = \text{ins}[j] \tag{5.7}$$

$$\text{next}(d))[j] = (\text{ins}'[j], \text{bdp}'[j])$$

$$\text{ins}'[j] = \begin{cases} \text{ins}[j+1] & \text{if } `-' \in \text{bdp}[j], \text{ and } j \leq n \\ \text{ins}[j-1] & \text{if } `+' \in \text{bdp}[j-1], \text{ and } j \geq 1 \\ \text{ins}[j] & \text{otherwise} \end{cases} \tag{5.8}$$

$$\text{bdp}'[j] = \begin{cases} \text{bdp}[j+1] & \text{if } `-' \in \text{bdp}[j+1] \text{ and } j \leq n \\ \{`+'\} & \text{if } `+' \in \text{bdp}[j-1] \text{ and } j \geq 1 \\ \phi & \text{otherwise} \end{cases} \tag{5.9}$$

where $(\text{ins}[j], \text{bdp}[j]) = d[j]$. The diagonal sequencer is responsible for unfolding run-length encodings (equations (5.3-5.5)) and supplying the diagonal restorer with the correct sequence of diagonal data (equation (5.6)). The diagonal restorer performs the corresponding operations on its diagonals (equations (5.8, 5.9)), and supplies the ISA with the current diagonal (equation 5.7).

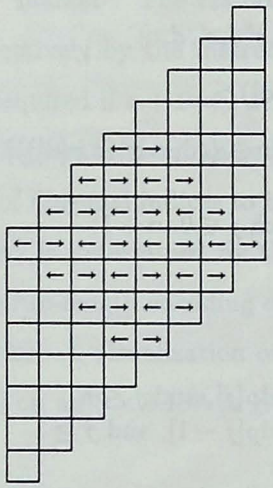
5.2.2 Example: row reversal revisited

Equations (5.3-5.9) are illustrated here by an alternative *divide-and-conquer* ISA row-reversal program, RowRev'. This program now uses the sub-program RotH'_{2^k}, which swaps odd-even adjacent blocks of width 2^{k-1} by using the horizontal

ringshift from the $(2^{k-1}-\bar{l})$th element of the left block to the $(2^{k-1}-\bar{l})$th element of the right block, repeated for $l = 1, \ldots, 2^{k-1}$. Thus, the ISADL encoding is given by:

$$\text{RowRev}' \;:\; (\text{RotH}'_{2^k})^{k=1..logn}$$

The RotH'_{2^k} sub-program is coded using two diagonals, as shown in Figure 5.5(c).



(a) matrix, for $k = 3, n = 8$

o	0	←	4	o	8
o	1	→	5	o	8
o	1	←	5	o	8
o	2	→	6	o	8
o	2	←	6	o	8
o	3	→	7	o	8
o	3	←	7	o	8
o	4	→	8	o	8

(b) ISADL encoding of this matrix

$$\left(\begin{array}{|c|c|c|c|} \hline o & \frac{d}{2}-l & \leftarrow & d-l \;\; o \;\; d \\ \hline o & \frac{d}{2}-\bar{l} & \rightarrow & d-\bar{l} \;\; o \;\; d \\ \hline \end{array} \right)^{l=1..\frac{d}{2}}$$

(c) general ISADL encoding, $d = 2^k$

Figure 5.5: RotH'_{2^k} program (selectors are '1's)

The *diagonal restorer* state corresponding to the encoding of Figure 5.5(c), for $k = 2$ and $n = 8$, is given in Figure 5.6. This indicates the first four diagonals produced (cf. the bottom four rows in Figure 5.5(b)). Note that the *diagonal sequencer* state is identical to that of Figure 5.4, with the first diagonal being indexed on odd time steps and the second on even time steps. At time step 3, cells 4 and 7 receive the instructions from cells 5 and 8, respectively, because cells 4 and 7 had a '–' (left-shifting boundary point marker) at time 1. The markers themselves are also shifted left.

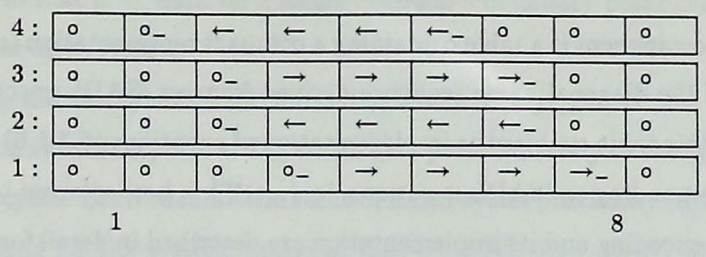


Figure 5.6: Diagonal restorer for RotH'_8 on an 8×8 ISA, time steps 1-4

5.3 Implementation of ISADL

This section describes the implementation of ISADL program compression, giving the basic design of the diagonal sequencer (Section 5.3.1) and diagonal restorer (Section 5.3.2). This basic design was assumed for the implementation of the Subroutining program compression method in Sections 3.6. An extension, significantly reducing *diagonal restorer* table area, is given in Section 5.3.3; it does however have the disadvantage of increasing *diagonal restorer* I/O bandwidth, which may need to be moderated by introducing further techniques.

The basic ideas presented in these sections are important for the reading of the rest of this chapter. An outline of an internal representation of ISADL programs, directly suitable for the loading the diagonal sequencer and restorer, is outlined in Section 5.3.4.

5.3.1 The diagonal sequencer

This section describes the design of the diagonal sequencer, which is very similar to a cell of the *matrix restorer* of the ISAC method of program compression (see Section 3.7.3). The function of the diagonal sequencer is to provide the *diagonal restorer* with the correct sequence of diagonal names (and possibly other information) so that it can produce the correct sequence of diagonals for the current program to be sent to the ISA. In practice, the diagonal sequencer is expected to be packaged separately from the ISA and the *diagonal restorer*. It requires $O(\log n)$ area, thus allowing for easy expandability of design in the overall

system.

Its main component is a table containing a compact representation (run-length encoding) of the diagonal name sequence derived from an ISADL encoding. The structure of this table enables the implementation of equations (5.3-5.6), the main feature corresponding to ISADL iterations, ie. $(\ldots)^{k=1..c}$, here referred to as *loops*. The tabular encoding and its implementation are described in detail for the ISAC method in Section 3.7.3 (see especially Figure 3.12); the only difference here is that diagonal names (from the set D_s) are used instead of ISA instructions, and that only one (as opposed to n) of these tables is used, so that area-efficient optimizations are not crucial here.

The extensions to ISADL of Sections 5.3.3 and 5.4 requires only that the diagonal name field of the table be augmented with extra information, and that, for nested loops with variable upper bounds, the diagonal sequencer has some means of updating the table's *loop counter* fields.

5.3.2 The diagonal restorer

Here is described a design for an ISADL diagonal restorer which is adequate for the constraints of Section 5.2.1. It corresponds to the *systolicization* of equations (5.7-5.9). The main data structures for diagonal restorer cell j are the tables INS and BDP (each of size N_{d}), where $(\mathrm{INS[d]}, \mathrm{BPD[d]}) \in (\mathcal{I} \times P(\{+, -, \text{''}\})$ correspond to $d\_r(d)_j, \mathrm{d} \in 1..N_{\mathrm{D}}$. The two cell components described below communicate through these tables, and at any time, are assumed to operate on disjoint partitions of them. Extensions to this design, to increase its generality, will be given in later sections.

5.3.2.1 Diagonal fetch/shift logic

This part of the diagonal restorer is responsible for fetching the instructions for the current diagonal to be sent to the ISA, and for performing any *shift* operations (ie. $+, -$) on it. Since the diagonal is sent to the ISA in a time skewed fashion, the fetch/shift operations must be *systolicized* [37, pp27-32], from lower numbered columns first. This is possible due to Constraint 3 of Section 5.2.1,

which ensures that a '−' shift on a single diagonal's boundary point cannot occur on two consecutive steps.

The diagonal fetch/shift logic for cell j, where $1 \leq j \leq n$, is now described. The cell has a systolic input $\mathrm{d} \in D_s$, which gets passed directly to the $(j+1)$th cell on the next cycle. Let d'' denote the value of d two cycles previously (a 'lookahead' of 2 is required for the systolicization of the '−' shift). $\mathrm{INS}'_-, \mathrm{INS}'_+$ are registers used to shift diagonal boundaries, with INS_- denoting the INS'_- register of cell $(j+1)$, and INS_+ denoting the INS'_+ register of cell $(j-1)$ (a similar convention applies for $\mathrm{BDP}'_+, \mathrm{BDP}'_-, \mathrm{BDP}_+, \mathrm{BDP}_-$). The shifting of diagonal boundaries is implemented by the following steps, performed at each ISA cycle. The instruction part is given by (cf. equation (5.8)):

$$\begin{pmatrix} \mathrm{INS}[\mathrm{d}''] \\ \\ \mathrm{INS}'_- \\ \\ \mathrm{INS}'_+ \end{pmatrix} \leftarrow \begin{pmatrix} \begin{pmatrix} \mathrm{INS}_- & \text{if } \text{`}-\text{'} \in \mathrm{BDP} \\ \mathrm{INS}_+ & \text{if } \text{`}+\text{'} \in \mathrm{BDP}_+ \end{pmatrix} \\ \begin{pmatrix} \mathrm{INS}_- & \text{if } \mathrm{d} = \mathrm{d}'' \\ \mathrm{INS}[\mathrm{d}] & \text{if } \mathrm{d} \neq \mathrm{d}'' \end{pmatrix} \\ \mathrm{INS}[\mathrm{d}''] \end{pmatrix}$$

Note that the test (d=d″) is identical in all diagonal restorer cells, and can be performed by the diagonal sequencer, with the result being passed systolically through the diagonal restorer. Similarly, the BDP part is given by (cf. equation (5.9)):

$$\begin{pmatrix} \mathrm{BDP}[\mathrm{d}''] \\ \\ \mathrm{BDP}'_- \\ \\ \mathrm{BDP}'_+ \end{pmatrix} \leftarrow \begin{pmatrix} \begin{pmatrix} \mathrm{BDP}_- & \text{if } \text{`}-\text{'} \in \mathrm{BDP}_- \\ \{\text{`}+\text{'}\} & \text{if } \text{`}+\text{'} \in \mathrm{BDP}_+ \\ \phi & \text{otherwise} \end{pmatrix} \\ \begin{pmatrix} \mathrm{BDP}_- & \text{if } \mathrm{d} = \mathrm{d}'' \\ \mathrm{BDP}[\mathrm{d}] & \text{if } \mathrm{d} \neq \mathrm{d}'' \end{pmatrix} \\ \mathrm{BDP}[\mathrm{d}''] \end{pmatrix}$$

The instruction sent to the corresponding column of the ISA is given by (cf. equation (5.7)):

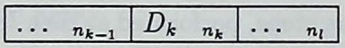
$$\mathrm{INS}[\mathrm{d}'']$$

5.3.2.2 The diagonal loading logic

A simple boolean processor is sufficient to set the L (load) bit which determines whether a diagonal restorer cell loads in its INS tables with the current load instruction value (which is passed across the restorer systolically). If the E (stack

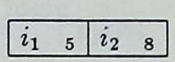
is also set at a cell, the BDP field (default value ϕ) is also loaded from the current input. The loading operation reflects the recursive structure of an ISADL diagonal, as is given by equations (5.2). The assumption that the array width n is a power of 2, as are also any *recurrence widths* (ie. n_l of equations (5.2)) within a diagonal, is essential for this component.

The design here is based the small boolean stack E, and a boolean variable E' for each cell in the diagonal restorer. The idea of the algorithm can be obtained by considering loading the ISADL diagonal:



It is assumed that the E' (E stack top) bit is set only at cells where this diagonal begins (ends), being in this case at cell 1 (cell n). This E stack top is pushed onto the E stack. If $n_l/n > 1$, E' is set high at cells $j|j \equiv 1(\text{modulo } n_l)$. To load sub-diagonal D_k, it is (inductively) assumed that the E' bit is set only at positions $j|j \equiv n_{k-1}+1(\text{modulo } n_l)$. The E stack top is then set only at positions $j|j \equiv n_k(\text{modulo } n_l)$, ie. one E bit is set $n_k - n_{k-1}$ units after every set E' bit. If D_k is an instruction, the L bit is set between consecutive high E'-E pairs (ie. in cells $j|j \equiv \delta(\text{modulo } n_l)$ where $n_{k-1} < \delta \leq n_k$) and the appropriate instruction is loaded. Otherwise, the procedure repeats recursively. Afterwards, the E' bit is set in cells immediately following the set E bits, ie. at cells $j|j \equiv n_k + 1(\text{modulo } n_l)$, so that the next sub-diagonal can be loaded. After all l sub-diagonals are loaded, the E stack is popped. The 0th and $(n+1)$th cells are special (requiring a single L bit, which can be set/reset as required) and are loaded separately.

Example: consider $n = 16$ and loading the ISADL diagonal



For this program, $n_1 = 5$ and $n_l = 8$, where $l = 2$. Initially, the 16 element diagonal restorer has its E' (E stack top) bit set only at cell 1 (cell 16). Then, the E' bit is set at cells $j|j \equiv 1(\text{modulo } 8)$. The top of E is then set in all cells after the first set E' (ie. in all cells). To select the cells $j|j \equiv 5(\text{modulo } 8)$, the

following bit operations are applied to the top of E:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	leave odd 1's
1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	leave odd 1's
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	leave even 1's

Here, odd [even] 1's only have been left at step $r, 1 \leq r \leq \log n_l$, according to whether the $(\log n_l + 1 - r)$th bit of $n_1 - 1 = 4$ is 1 [is 0].

The full algorithm, with optimizations, is presented in Appendix 5.B.1.

5.3.3 Optimizing program loading and storage area

Typically, the (initial values of) diagonals of an ISA program are slight variations of a very few generic *diagonal patterns*. For example, many ISA programs consists of several near-uniform diagonals, each with different instructions. In such cases, it is inefficient for the diagonal restorer to load and store each slightly different diagonal separately. This section introduces the concept of *diagonal patterns*, which reduce the diagonal restorer's loading time and internal storage at the expense of its input bandwidth. This bandwidth can be reduced using the concept of 'macros'.

The central idea is to load the diagonal restorer with a *diagonal pattern* which represents several similar (initial values of) diagonals. Upon execution, the diagonal sequencer supplies the necessary information to the diagonal restorer so that it can reduce the *diagonal pattern* to the required diagonal. This concept is recommended for ISADL implementation, and is used for the evaluation of Section 5.5.

The concept of *diagonal patterns* is introduced via the *instruction variables*, $\tilde{\mathcal{I}} = \{\tilde{\mathcal{I}}_1, \ldots, \tilde{\mathcal{I}}_{N_\mathrm{p}}\}$, where $N_\mathrm{p} \geq 0$ and $\tilde{\mathcal{I}} \cup \mathcal{I} = \phi$, The RotH_d program can be considered to have a single *diagonal pattern*, which for each of its diagonals, an $\tilde{\mathcal{I}} \rightarrow \mathcal{I}$ lookup table can be used to reconstruct the diagonal from the pattern, as is shown in Figure 5.7.

The implementation of *diagonal patterns* requires the following simple modifications:

- instead of storing just the diagonal (pattern) name in each entry of its table,

$\leftarrow$ 7	o 8
o 1	$\rightarrow$ 8

(a) ISADL encoding (diagonal names annotated)

$\leftarrow$ 1	$\leftarrow$ 7	o 8
o 1	$\rightarrow$ 7	$\rightarrow$ 8

(b) 'merging' diagonals to have identical boundaries

$\tilde{\mathcal{I}}_1/\leftarrow, \tilde{\mathcal{I}}_2/\leftarrow, \tilde{\mathcal{I}}_3/\mathrm{o}$	
$\tilde{\mathcal{I}}_1/\mathrm{o}, \tilde{\mathcal{I}}_2/\rightarrow, \tilde{\mathcal{I}}_3/\rightarrow$	

$\tilde{\mathcal{I}}_1$ 1	$\tilde{\mathcal{I}}_2$ 7	$\tilde{\mathcal{I}}_3$ 8
$\tilde{\mathcal{I}}_1$ 1	$\tilde{\mathcal{I}}_2$ 7	$\tilde{\mathcal{I}}_3$ 8

(c) lookup tables for each instance of the single-*diagonal pattern* version

Figure 5.7: Single iteration of RotH_8 program illustrating a *diagonal pattern* requiring $N_\mathrm{p} = 3$.

the diagonal sequencer now also stores a 'shift-suppress' bit (default value 0; see below for its use) and the pattern's lookup tables. Upon accessing this entry, the diagonal sequencer sends all this information to the diagonal restorer.

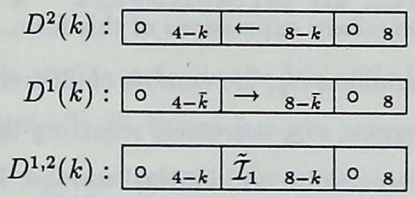
- the diagonal restorer fetches the entry according to the current diagonal (pattern) name: if the entry is in $\tilde{\mathcal{I}}$, the corresponding entry in the lookup table is sent to the ISA, otherwise the entry is sent to the ISA. If the 'shift-suppress' bit is 1, the INS and BDP tables in the diagonal restorer cells are not updated (but otherwise operate as described in Section 5.3.2.1).

This concept can be extended to diagonals whose boundary expressions vary (over iterations). Consider the program RotH'_8 (cf. Figure 5.5), which can be expressed as:

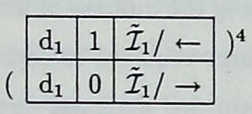
$$(D^1(k); D^2(k))^{k=1..4}$$

where $D^1(k)$ and $D^2(k)$ are defined in Figure 5.8(a). Observing that the boundaries for $D^2(k)$ and $D^1(k+1)$ are identical, these can be merged into a single

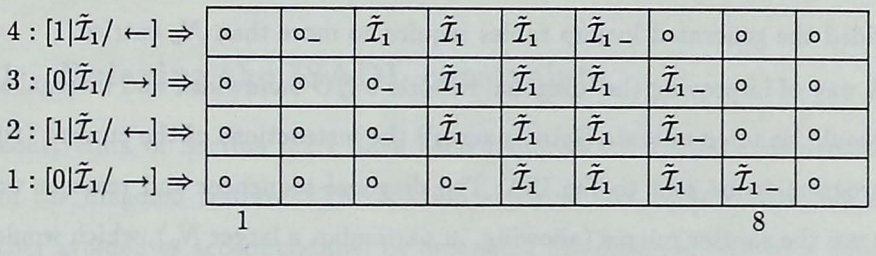
diagonal pattern $D^{1,2}(k)$, defined also in Figure 5.8(a). The implementation of these is illustrated in Figure 5.8(b,c) (cf. Figure 5.6).



(a) diagonals of RotH'_8 made into a *diagonal pattern*



(b) diagonal sequencer state



(c) diagonal restorer state, time steps 1-4

Figure 5.8: RotH'_8 program with single diagonal pattern (named d_1)

In general, upon forming diagonal patterns over the iteration of at least r diagonals:

$$(\ldots; D^1(k); \ldots; D^2(k); \ldots; D^r(k); \ldots)^{k=1..d}$$

there may be a particular r_0, where $1 \leq r_0 \leq r$, in which the diagonals:

$$D^{r_0+1}(k), \ldots, D^r(k), D^1(k+1), \ldots, D^{r_0}(k+1)$$

are most easily merged into a single diagonal pattern. For this diagonal pattern, the 'shift-suppress' bit is 1 in the diagonal sequencer's table for entries corresponding to D^r for $r \neq r_0$, and only at D_{r_0} are the diagonal pattern's boundaries shifted.

The merging of ISADL diagonals into diagonal patterns can also extend the types of boundary expressions possible. In particular, it is sometimes possible to implement diagonals whose boundary expressions are of the forms '$c, c_- - rk, c_+ +$

rk' provided r of them can be merged into a single diagonal pattern (all of these require a 0 'shift suppress' bit). Other diagonals may be merged into this pattern, and these require a 1 'shift suppress' bit. However, we have not yet encountered any examples of ISA programs that require boundary expressions of this form.

By increasing the number of instruction variables, N_p, the number of different diagonal patterns required to represent a program, N_d, decreases, reflecting the tradeoff between the diagonal restorer's I/O bandwidth and internal storage. In a real implementation, N_p would be fixed to a small value, eg. $N_p \leq 4$, and the groups of diagonals to be merged into a pattern can be chosen by hand. The diagonal patterns for each of these groups can then be (automatically) generated, provided the generated lookup tables require no more than N_p entries.

A way of improving the diagonal restorer's I/O bandwidth and internal storage would be to enumerate (into 'macros') the instructions of the current ($\Omega(n)$) subprogram to be sent to the ISA. The diagonal sequencer and restorer would then use the shorter micros (allowing, in particular, a larger N_p), which would be decoded using (dynamically loadable) tables before entering the ISA cells. The overall storage may not always be reduced, but the overall I/O bandwidth would be reduced, since the decode tables, reloaded every $\Omega(n)$ cycles, require only a modest I/O bandwidth.

5.3.4 Representation of ISADL programs for loading

Using the concepts outlined in Section 3.3, an $O(\log n)$ space representation of the ISADL program would be stored in an external ISA program memory. This representation should enable efficient loading of the diagonal sequencer and restorer tables with their initial values. A more or less direct representation of the diagonal sequencer table could be stored in the external program memory. A simple, compressed representation of the $O(\log n)$ length diagonal load logic's instruction sequence (together with its inputs) for the diagonal restorer could be stored in the external memory. To load the diagonal restorer tables, a simple 'front end' to the diagonal restorer (with buffering) would be required to perform these expansions. Since these details should not create any difficulties, they are

not further elaborated here.

5.4 Extensions of ISADL

This section extends the basic ISADL implementation of Section 5.3 to cover practically all known features required by ISA programs. These extensions are of two forms: relaxing the constraints on ISADL required by the implementation of Section 5.3.2 (these are used in the evaluation of ISADL in Section 5.5.2), and combining ISADL with other forms of program compression mentioned in Chapter 3.

5.4.1 Relaxing the ISADL constraints

The constraints of Section 5.2.1 have been required for a simple implementation of the diagonal restorer. This section explains how these constraints can be either avoided or accommodated by extending the diagonal restorer's design. The extensions of Constraints 1 and 2 also have application in incorporating the higher-level aspects of Subroutining into ISADL.

5.4.1.1 Nested loops: Constraint 1

A diagonal whose boundary expressions are dependent on k, the index of some enclosing loop, has been constrained to not be enclosed inside any other loops. This section shows how this constraint can be relaxed. Consider an ISADL program P_{k_1,k_2} whose boundaries may depend on the integers k_1, k_2 (and no other loop indices). The ISADL program $(\ldots(P_{k_1,k_2})^{k_1=1..d_1}\ldots)^{k_2=1..d_2}$ can be implemented in two ways:

- upon every iteration of the outer (k_2) loop, reload the diagonals corresponding to P_{1,k_2}. This is feasible if d_1 is sufficiently large, ie. $d_1 = O(n)$, and the reloading could be done either by the host or an extended diagonal sequencer.

- initially store the diagonals corresponding to $P_{1,1}$ in a separate area of the diagonal restorer (eg. the SD and l/r tables of Section 5.4.1.2). Consider

the k_2th iteration of the outer loop, for the diagonal d_{k_2} of P_{1,k_2}. Just before the first use of d_{k_2} (in the inner loop), it is copied to the main tables of the diagonal restorer; then d_{k_2+1} is produced systolically from d_{k_2} in the fashion described in Section 5.3.2.1. This approach would at most duplicate the storage required by the diagonal restorer, and only very slightly affect its I/O bandwidth.

However, if P_{k_1,k_2} is independent of k_1, a simplification can be made. The diagonals for $P_{1,1}$ are initially loaded into the diagonal restorer. Consider the k_2th iteration of the outer loop. On all but the last iterations of the k_1 loop, the diagonal sequencer always sends a 'suppress-shift' bit of '1' to the diagonal restorer. On the last iteration, the normal 'shift-suppress' bit values are sent to the diagonal restorer to produce the diagonals for P_{1,k_2+1}.

5.4.1.2 Divide-and-conquer programs: Constraint 2

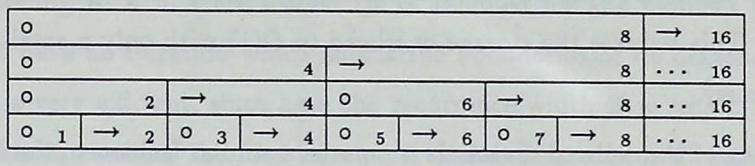
Constraint 2 restricts boundary expressions in an ISADL diagonal to be of the forms c, $s-k$ or $a+k$ where k is the index of an enclosing iteration. In practice, this needs to be relaxed only to cover cases where the boundary expressions depend exponentially on k, as occurs in *divide-and-conquer* programs, eg. program DivConq[5] of Figure 5.9.

These ISADL programs are generally of the form[6] $(P(k))^{k=l..u}$, with $1 \leq l < u \leq \log n$, which consists of patterns repeated over recurrence widths of 2^k. The boundary expressions of $P(k)$ are of the forms c, $2^k - c$, $2^{\bar{k}} \pm c$, where $c \geq 0$ is an expression independent of k. Typically $P(k)$ also contains a nested loop whose upper bound is $\Theta(2^k)$; this detail affects only the design of the diagonal sequencer. Approaches for implementing such ISADL programs mainly affect the diagonal restorer and include:

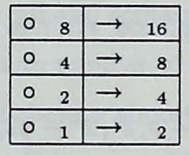
- reload the diagonal restorer upon each iteration of k, as described in Section 5.4.1.1. This is simple and inexpensive, but has the drawback that for small

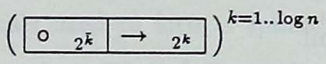
[5]This program is taken from the first repetition of the first diagonal of the $(\mathrm{RotH}'_{2^k})^{k=1..\log n}$ program and is only used to illustrate this technique.

[6]cf. $(\mathrm{RotH}'_{2^k})^{k=1..\log n}$, see Figure 5.3.



(a) 'expanded' ISADL program, $n = 16$





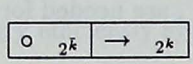
(b) usual ISADL encoding, $n = 16$ (c) introducing iteration

Figure 5.9: Program DivConq, showing the spliced diagonal technique

k, $P(k)$ may be too short to load the diagonals for $P(k+1)$.

- load all $(u - l + 1)$ versions of the boundaries into the diagonal restorer before execution. This is the most general approach, and is feasible since $u \leq \log n$, but has the disadvantage of making the diagonal restorer's cells have $O(\log n)$ area.

- introduce the technique of *spliced diagonals*, which involves separating the ISADL diagonals of $P(k)$ into sub-diagonals (which are independent of k) on either side of all boundaries of the four forms $a, 2^k - s, 2^{\bar{k}} - s2, 2^{\bar{k}} + a2$, (where $a, a2 > 0$ and $s, s2 \geq 0$) together with the recurrence and *half-recurrence* widths (of forms $2^k, 2^{\bar{k}}$). These are collectively called the *splicing boundaries.*

  eg. the diagonal  of the DivConq program (see Figure 5.9) has two such sub-diagonals, being uniform diagonals of 'o' or '→' instructions. It has the boundaries $2^k - 0$ and $2^{\bar{k}} - 0$.

The (initial values of the) diagonals of $P(k)$ are then produced by combining the sub-diagonals using the splicing boundaries in the obvious way.

The approach here is load the splicing boundaries of $P(1)$ only, and from their kth version, produce the $(k+1)$th version. This can be achieved by

simple bit operations on bit-arrays which represent the splicing boundaries. This reduces the storage overhead to $O(1)$ with only a small loss of generality.

Loading of the spliced diagonals is done in a similar fashion to the loading of ISADL diagonals, as described in Section 5.3.2.2.

The last approach is deemed the most promising, and will now be described in further detail.

The splicing boundaries of the forms $a, 2^k - s, 2^k, 2^{\bar{k}} - s2, 2^{\bar{k}} + a2, 2^{\bar{k}}$, are stored respectively in the bit-arrays $\mathrm{l}, \mathrm{r}, \mathrm{i}, \mathrm{l2}, \mathrm{r2}, \mathrm{i2} \in (1..n \rightarrow (1..N_s \rightarrow \{0,1\}))$, where $N_s \leq N_d$ gives the subset of diagonals that can be formed by the spliced diagonal technique. In this case, a '1' in an array cell means a boundary (of the appropriate form) occurs there. In the case of 'l2' and 'r2', 'phantom' boundaries corresponding to $(2^{\bar{k}} - s2)$ and $(a2)$ are also inserted to enable efficient updating of these bit-arrays.

eg. for the DivConq program with $n = 16$, the splicing boundaries of interest for the diagonal $\boxed{\circ \quad 2^k \mid \rightarrow \quad 2^k}$ (having diagonal name d_0 say) are illustrated below. Here, $\mathrm{r}[\mathrm{d}_0]$ and $\mathrm{i}[\mathrm{d}_0]$ correspond to the expressions $(2^k - 0)$ and (2^k), and are visualized by a single array, where '[' (']') represents a '1' in $\mathrm{r}[\mathrm{d}_0]$ ($\mathrm{i}[\mathrm{d}_0]$) only:

$k = 2:$ 0 0 0 [] 0 0 0 [] 0 0 0 [] 0 0 0 []
$k = 3:$ 0 0 0 0 0 0 0 [] 0 0 0 0 0 0 0 []

A similar convention is used now to visualize $\mathrm{l2}[\mathrm{d}_2]$ and $\mathrm{i2}[\mathrm{d}_2]$, which here correspond to the expressions $(2^{\bar{k}} - 0)$ and $(2^{\bar{k}})$. Here 'phantom' boundaries of $\mathrm{l2}[\mathrm{d}_2]$, written as '[', are needed for the formation of the next iteration's diagonal only:

$k = 2:$ 0 [] 0 [] 0 [] 0 [] 0 [] 0 [] 0 [] 0 []
$k = 3:$ 0 0 0 [] 0 0 0 [] 0 0 0 [] 0 0 0 []

From inspection, increasing k requires the deleting of every other '[ ]' pair.

The technique is now more generally described. Consider again the program $(P(k))^{k=l..u}$. Before its execution, the sub-diagonals and splicing boundaries

corresponding to $k = l$ are loaded (it is assumed for the moment that $P(k)$ does not have an iteration which affects the boundaries of its diagonals). This loading is very efficient, since here the recurrence width 2^l is small. Upon the kth iteration, consider the *first* time a particular diagonal (or *diagonal pattern*, see Section 5.3.3) having index d_0, is about to be used in diagonal restorer cell j. Assume that this cell also contains the table $\mathrm{SD} \in (1..\bar{N}_{\mathrm{sd}} \rightarrow (1..N_{\mathrm{s}} \rightarrow \mathcal{I}))$. Here, N_{sd} is the maximum number of sub-diagonals used to form a spliced diagonal; in practice, one of these is almost always a uniform 'NoOp' diagonal, which does not have to be explicitly stored in SD. The diagonal's jth element is set to the s_jth sub-diagonal (counted from the left) of d_0 at cell j, as follows:

$$
\begin{aligned}
\mathrm{INS}[\mathrm{d}_0] &\leftarrow \mathrm{SD}[\mathrm{d}_0][s_j] \\
s_0 &= 0 \\
s_j &= \begin{cases} 0 & \text{if } \mathrm{i}[\mathrm{d}_0][j] = 1 \\ s_{j-1} + (\mathrm{l}[\mathrm{d}_0][j] \vee \mathrm{r}[\mathrm{d}_0][j]) & \text{otherwise} \\ \qquad + (\widetilde{p2_j}\,\mathrm{l2}[\mathrm{d}_0][j] \vee p2_j\,\mathrm{r2}[\mathrm{d}_0][j]) & \end{cases} \\
p2_j &= (\mathrm{parity}(\mathrm{i2}[\mathrm{d}_0][1..j-1]) = 1)
\end{aligned}
$$

where $\tilde{x}$ denotes the negation of x. The variable $p2_j$ is used to ignore the 'phantom' boundaries in the l2 and r2 tables. The splicing boundaries are simultaneously updated for the next (ie. $(k+1)$th) iteration by deleting every odd (or even) entry [this doubles the distances between corresponding entries]:

$$
\begin{aligned}
(\mathrm{l}[\mathrm{d}_0][j], \mathrm{r}[\mathrm{d}_0][j], \mathrm{i}[\mathrm{d}_0][j]) &\leftarrow (\mathrm{l}[\mathrm{d}_0][j] \wedge \widetilde{p_j},\ \mathrm{r}[\mathrm{d}_0][j] \wedge p_j,\ \mathrm{i}[\mathrm{d}_0][j] \wedge \widetilde{p_j}) \\
p_j &= (\mathrm{parity}(\mathrm{i}[\mathrm{d}_0][1..j-1]) = 1) \\
(\mathrm{l2}[\mathrm{d}_0][j], \mathrm{r2}[\mathrm{d}_0][j], \mathrm{i2}[\mathrm{d}_0][j]) &\leftarrow (\mathrm{l2}[\mathrm{d}_0][j] \wedge p2_j, \mathrm{r2}[\mathrm{d}_0][j] \wedge \widetilde{p2_j}, \mathrm{i2}[\mathrm{d}_0][j] \wedge \widetilde{p2_j})
\end{aligned}
$$

These operations can be very efficiently systolicized (cf. Section 5.3.2.1).

For programs of the form $(P(k))^{k=u..l}$, where $u > l$, these operations on the splicing boundaries need only be reversed. This can be achieved by shifting a copy of these bit arrays the appropriate distance, and then recombining the copy with the original. Such operations can also be easily systolicized if $P(k)$ contains at least $2^{\bar{k}}$ iterations of each diagonal (usually the case in practice), with data being shifted one unit on each such iteration. Since such programs have not been found necessary in practice, their implementation is not further developed.

If $P(k)$ contains an iteration of the form $(P'(k,o))^{o=1..l(k)}$, the *boundary point* operations (cf. the variable BDP of Section 5.3.2.1) must also be considered. This can be done by annotating an extra field, denoted '.B', of type $\mathcal{P}(\{+,-\})$ to the splicing boundaries l, r, l2 and r2. The boundary point operation in diagonal restorer cell j for diagonal d_0 is formed as follows:

$$\mathrm{BDP}[d_0] \leftarrow (B_1(l[d_0][j]) \cup B_1(r[d_0][j])) \cup (B_{\widetilde{p2}_j}(l2[d_0][j]) \cup B_{p2_j}(r2[d_0][j]))$$

$$B_b((\mathrm{bit},\mathrm{bdp})) = \begin{cases} \mathrm{bdp} & \text{if } b \wedge \mathrm{bit} = 1 \\ \{\} & \text{otherwise} \end{cases}$$

Example: the $(\mathrm{RotH}'_{2^k})^{k=1..\log n}$ program has identical splicing boundaries as the DivConq program, except that l2 and r are annotated with '−' markers. For $k = 3$, these can be visualized as:

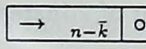
r :	0	0	0	0	0	0	0	[⁻]	0	0	0	0	0	0	0	[⁻]
l2 :	0	0	0	[⁻]	0	0	0	[⁻]	0	0	0	[⁻]	0	0	0	[⁻]

To reduce storage, it is possible to merge the 'l' and 'r' (also 'r2' and 'l2') arrays together, since the splicing boundaries represented by each should never overlap. These were modelled separately here to simplify the presentation. Also, the $i[1..N_d][1..n]$ and $i2[1..N_d][1..n]$ arrays could also be merged together, and can in practice be shared by all diagonals that are constructed using splicing boundaries. Thus, the extra hardware required for implementing splicing boundaries is reasonably modest.

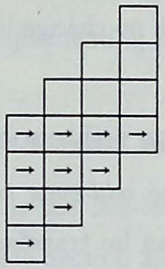
One possibility would be to implement all of ISADL using spliced diagonals, ie. shift bit-patterns, rather than ISA instructions, to implement diagonal boundaries of the form $c \pm k$. This might result in a reduction of hardware and the different mechanisms required to implement ISADL, but would have complicated and obscured many of the ideas presented here.

5.4.1.3 'Flat' ISA programs: Constraint 3

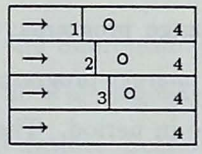
If in $(P_k)^{k=1..d}$, P_k consists of a single diagonal which has a boundary expression of the form $s - k$, the diagonal restorer cannot easily generate $P_1, \ldots, P_d$. Solutions to this problem include transforming the 'flat' ISA programs into semantically equivalent (or at least sufficiently similar) ISA programs, or by including new (and

relatively expensive) mechanisms into the diagonal restorer. This is illustrated in the LoadMat program of Figure 5.10, where $P_k =$ 

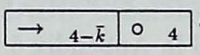
. While the



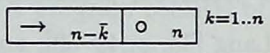
(a) matrix, for a 4 × 4 ISA



(b) ISADL encoding of matrix



(c) introducing iteration, $n = 4$



(d) general ISADL encoding

Figure 5.10: 'Flat' LoadMat program, instruction part

k'th diagonal can be produced by a left shift on the kth diagonal, (see Figure 5.10(b)), the overall timing requires, as the nth row of the program matrix enters the ISA, that all columns of the diagonal restorer simultaneously send a new instruction to the ISA. For iterations of r diagonals, this phenomenon also occurs if there is a boundary of the form $c - rk$ (only of interest when the r diagonals are merged into a single diagonal pattern). Such simultaneous changes cannot be achieved by simple systolic shifting.

In general, a 'flat' ISA program (ie. a ($> O(1)$) section of a row of the program's matrix forms a boundary between two different sorts of instructions) must be handled in one of the following ways:

- be replaced by a sufficiently similar non-'flat' program. For example, the LoadMat program can be replaced by the program of Figure 2.13, which loads (in period n) in row-reversed order, a matrix from the west data buffer into the ISA.

- be transformed into an equivalent non-'flat' program. For the 'flat' ISA

programs encountered so far, such transformations can be found with at most a factor of 2 increase in period. eg. Loading/unloading a matrix from the west is done by inserting an extra diagonal (cf. Figures 5.10 and 5.1). However, programs loading/unloading a matrix from the east (requiring period $2n$) are transformed similarly but with no change in the period (see Appendix 5.A.3).

The other known class of 'flat' ISA programs includes some of the comparison-exchange programs, eg. the row-wise *odd-even transposition sort* and *triangle merger* programs [46], which can be transformed with negligible increase in period. These transformations are achieved by 'skewing' the comparison-exchanges, and inserting two empty 'anti' diagonals. This method is preferred, presumably on the grounds of better programming style, by Lang [35, pp49–59], and is illustrated in Appendix 5.B.2.

- be handled using an idea similar to the spliced diagonal technique introduced in Section 5.4.1.2. Consider the kth diagonal of the ISADL encoding of the LoadMat program of Figure 5.10(c). The sub-diagonal left (right) of the '$n - \bar{k}$' boundary is composed of '$\rightarrow$' ('o') instructions, and is given index d_1 (d_0), say. Upon executing the kth iteration of this diagonal, a counter is passed systolically across the restorer:

  > its initial value is $4 - \bar{k}$, and at each cell, it is decremented (ie. at cell j, it has value $4 - \bar{k} - \bar{j}$)

  If this counter is (not) greater than 0, the d_1th (d_0th) diagonal is selected.

  This idea can be easily extended to more general examples of 'flat' ISA programs, but has the disadvantage of making the diagonal restorer's cells dependent on $\log n$ (the size of the largest counters needed).

5.4.2 Combining ISADL with the SISA

ISADL is used for program compression on both the ISA instruction and selector matrices similarly. For the SISA [31], the instruction matrix is replaced by a 'top'

selector matrix and a single instruction stream. The latter can be implemented by inserting an extra field (for the instruction stream) in the 'top' selector matrix's diagonal sequencer: this instruction field is sent to the SISA instead of the 'top' diagonal restorer. Since the diagonal restorers need only store and manipulate bit information, implementing ISADL on the SISA is extremely efficient.

5.4.3 Incorporating subroutines into ISADL

ISADL incorporates the low-level concepts of the Subroutining program compression method (see Section 3.6), such as *iterations* of diagonals, even when the diagonals are dependent on the *index* of the iteration. Section 5.A illustrates how the meaningful naming of (parameterized) sub-programs is naturally incorporated into ISADL. Generally, if an $\Omega(n)$ ISA program consists of many calls of a sub-program, this can be implemented in ISADL as follows:

- load the diagonal sequencer in advance with all diagonal information required, excepting diagonals (diagonal patterns) that can be easily constructed from previous ones (as is done for iterations and divide-and-conquer programs).

- implement subroutines in the diagonal sequencer's table, in a similar fashion to normal computer memory. The essential features of this would be to update (ie. $\pm 1, \times 2, /2$) and to reset (ie. using small stacks) loop indices and to send diagonal update/reset commands to the diagonal restorer (some of these have already been modelled). Useful features would include organizing the diagonal sequencer's table in a fashion similar to implementing subroutines in RAM, with a mechanism for subroutine return.

These extensions are illustrated in the ISADL implementation of binary recursive ISA programs (see [48, Sect.5]) in Appendix 5.B.3.

5.4.4 Combining ISADL with microprogramming

A (μ, σ) wavefront microprogrammed ISA (see Sections 4.1.1 and 4.3) has a skew between neighbouring columns of selectors of $\mu \geq 1$ and a skew between neigh-

bouring columns of instructions of $\sigma \geq 1$. It has *micro*(instruction)*s* of width w_i, and sequences of λ *micros* may be coded by a *macro* of width $w_m < w_i$. Two approaches can be taken to use ISADL on a microprogrammed ISA:

- use *macros* in the diagonal restorer, which are decoded by a row of *decode tables* before entering the μISA. This can reduce the diagonal restorer I/O bandwidth by a factor of $w_m/w_i\lambda$, and hence the diagonal restorer would be able to afford a larger value of N_p.

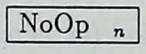
- use *micros* in the diagonal restorer. This eliminates the need for *decode tables*, and the diagonals of an ISA program are generally more simply expressed on the *micro* level than on the *macro* level. The loss of the program compression due to microprogramming is unimportant since ISADL provides much more powerful program compression.

Although it is important to reduce the I/O bandwidth of the diagonal restorer, the second approach will be developed here. This is because λ must generally be a factor of the number of *micros* within every iteration in an ISADL program (normally the case when a *macro* represents a meaningful semantic unit, but otherwise a value of $\lambda > 2$ could seldom be used). Since the μISA would be intended to run a range of such programs, the diagonal restorer would still need sufficient I/O bandwidth to cope with the lowest value of λ (which is unlikely to exceed 2).

Consider the diagonal restorer for the top (ie. instruction) part of the ISA program, which must implement a skew between neighbouring columns of $\sigma \geq 1$. The design given in Section 5.3.2 must now be generalized from $\sigma = 1$ to handle larger values of σ. All systolic information passed downstream must now be buffered by queues of length σ. The systolic diagonal index input, d, must now have a lookahead, d″, of $\sigma + 1$. Hence, in diagonal restorer cell j, d″ would refer to the entry entering cell j's diagonal index queue, and the 'd' would refer to the entry at the tail of the queue on the previous ISA microcycle.

Since a small range of the values of σ would be adequate for most microprogrammed ISA systems, these queues can be efficiently implemented by variable length shift registers (cf. the selector queues of the μISA cells in Section 4.3.1).

Changing the queue lengths can be done incrementally, and to increment σ, a diagonal index corresponding to the diagonal



should be inserted in the diagonal index queue.

The additional storage of these queues might require reducing the size of N_p, and hence that of the lookup tables required by introducing diagonal patterns (Section 5.3.3), since these tables must also be queued. The diagonal restorer of the left (selector) part of the microprogrammed ISA is similar, except that μ is interchanged for σ. The design of the diagonal sequencer is unaffected.

5.5 Evaluation of ISADL

Since ISADL is here proposed as a practical method for ISA program compression, this section examines the requirements of ISADL on the ISA algorithms presented in this chapter (including Appendix 5.A), giving the values of the ISADL (diagonal restorer) parameters N_d and N_p in each case. It then examines the hardware requirements of ISADL due to these (and other) parameters. In both these cases, the merging of diagonals into diagonal patterns (Section 5.3.3) enhances the efficiency of ISADL's implementation. An overall conclusion for ISADL can then be given.

5.5.1 ISADL requirements for various ISA algorithms

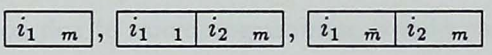
Table 5.1 gives the requirements of ISADL for the various ISA programs presented earlier in this chapter, those listed in Table 3.2, and those presented in Appendix 5.A. Except for programs MedFind and Match^n, these programs are of 'irreducible' code size (in the sense of Section 3.3) and hence realistically represent the volume of information that must be held in the diagonal restorer at any one time.

For a given (optimally chosen) value of N_p, the number of *instruction variables* used to form *diagonal patterns*, the table gives the values of N_d, the number of

diagonal (patterns) of each program. This can be compared with the program's value of N_{d0}, which is N_d evaluated for $N_p = 0$. The reduction of the static memory from implementing diagonal patterns is given by N_d/N_{d0}.

The table also gives L_d, the period (ie. number of diagonal restorer loading instructions) for loading these N_d diagonals into the diagonal restorer. L_d is hoped to be generally less than the array size n, here assumed to be a power of 2, and at least less than the period of the program itself.

In evaluating L_d, the fully optimized version of the algorithm of Appendix 5.B.1 is used. The main results are that, assuming the E' (E) bit is set at cells $j|j \bmod m = 1$ ($j|j \bmod m = m$), diagonals of the forms:



can be loaded in periods 1, 2 and 2 respectively. Such diagonals are the most common in ISA programs, so that the loading algorithm is generally very efficient.

It can then be concluded that allowing a small value of N_p can significantly reduce N_d and keep L_d small, indeed smaller than the array width (program period) for $n \geq 32$ ($n \geq 16$) for all of the above programs. However, ISA programs can be devised with worse loading performance, and in these cases a moderate array size n is required to make $n \geq L_d$.

5.5.2 ISADL memory and I/O bandwidth requirements

Section 5.5.1 gives some feeling as to the values of the ISADL parameters N_d, N_p, N_s and N_{sd} required by practical implementations. This section estimates the hardware required by the implementation of the diagonal restorer of Section 5.3 in terms of its principal components, the (static) memory and the inter-cell I/O bandwidth. The latter is important since it corresponds directly to the pin count of the diagonal restorer. The cost of the diagonal sequencer's hardware is relatively small, since it has $O(\log n)$ area.

Table 5.2 gives the cost of the diagonal restorer's hardware in terms of these components, where w_i is the width (in bits) of the 'control codes' of the diagonal restorer. This has been given for the basic implementation of ISADL with its

program	notes	N_{d0}	N_p	N_d	period, t	L_d	$\min n \mid t \geq L_d$
LoadMat	$*$	2	0	2	$2n$	4	2
TransClos	$*$	1	0	1	n	3	4
RowRev	o	2	0	2	$\approx 2n$	14	8
RowRev$'$	o$*$	2	1	1	$\approx 2n$	11	8
RotHV$(n/2)$		4	2	2	$2n$	4	2
Transpose	o$*$	6	2	3	$\approx 6n$	24	4
Matrix Mult.		5	1	1	$5n$	1	1
Perfect Sh.	‡	2	0	2	n	$2\log n + 5$	16
RedSquares		8	1	2	$7n + 2$	4	1
MedFind	•	10	1	1	$\geq 4L \log L$	1	1
Match^n	$*$	11	1	3	$7n + 6$	6	8
AssocMem	$*$•	3	2	2	$2m + 2$	$\approx \log m + 6$	4 if $m \geq 4$
MatMult	$*$†	6	1	1	$6n$	3	1
TransClos$'$	$*$	7	1	1	$13n$	3	1

*: in these programs, diagonal restorer cells 0 or $(n + 1)$ also need to be loaded, taking an extra cycle for each such operation.

o: these are *divide-and-conquer* programs, in which L_d is given by the periods to load the *splicing boundaries* (for the first iteration only) and the corresponding sub-diagonals (these use the algorithm for loading ordinary ISADL diagonals). They require $N_s \leq 2$ *spliced diagonals* and $N_{sd} = 2$ sub-diagonals.

• the period of these programs is dependent on another parameter (ie. L or m) assumed to be $\Omega(n^r), 0 < r \leq 1$, and not exceeding n.

†: the $\sigma = 3$ version of MatMult is used here.

‡ the ISADL encoding of the Perfect Shuffle program is given in Appendix 5.B.2.

Table 5.1: ISADL parameter values for various ISA programs

feature	I/O bandwidth		static memory		assoc. logic
	variable	# bits	variable	# bits	
diagonal fetch/ shift logic	d	$\log N_d$			
	INS_, INS_+	$2 \times w_i$	INS	$w_i N_d$	R/W access
	BDP_, BDP_+	2×2	BDP	$2N_d$	R/W access
diagonal loading logic	h	1	L, E'	2×1	bit opns.
	instr'n, etc.	$w_i + 2 + \log N_d$	E	4	
	opcodes	5			
diagonal patterns	lookup tables	$N_p w_i$	INS	N_d added	
	'suppress' bit	1			
Divide -and Conquer	$p, p2$	2×1	l/r, l2/r2	$2 \times 3N_s$	bit opns.
	s	$\log N_{sd}$	i/i2	N_s	$+1 \pmod{N_{sd}}$
			SD	$N_s \bar{N}_{sd} w_i$	R/W access
micro-prog.	all cell inputs must be buffered in shift registers of length $\sigma - 1$				
total # bits	$(N_p + 3)w_i + 2 \log N_d + \log N_{sd} + 15$		$(N_d + N_s \bar{N}_{sd})w_i + 3N_d + 7N_s + 6$		

Table 5.2: Diagonal restorer cell hardware requirements

recommended extensions of Sections 5.3 and 5.4. Note that nested loops (Section 5.4.1.1) can be implemented using the SD and l/r tables of the divide-and-conquer hardware (assuming that these tables are loaded by the diagonal load logic). Note that for the diagonal fetch/shift logic, the variables $INS'_+, INS'_-, BDP'_+, BDP'_-$ and d'' are refreshed every cycle, so they need not be implemented using static memory storage. For each row, the hardware cost (in bits) is annotated with the associated variables; the reader might need to refer back to the relevant section to recall their functions. The logic associated with updating these variables is given in the rightmost column.

Example: For the set of algorithms used in this chapter (see Table 5.1), sufficient values of the above parameters are:

$$N_d = 4, N_p = 2, N_s = 4, \bar{N}_{sd} = 1$$

Substituting these values in the totals of Table 5.2 gives a total I/O bandwidth of $\approx (5w_i + 20)$ bits, and a total static memory of $\approx (8w_i + 45)$ bits. For a diagonal restorer cell for the selector part of the ISA program, $w_i = 1$, so that ≈ 25 bits I/O bandwidth and ≈ 55 bits of static memory are required. For a

cell for the instruction part, w_i can be reduced to ≈ 4 (using microprogramming-like techniques; see Section 5.3.3), so that ≈ 40 bits I/O bandwidth and ≈ 75 bits of static memory are required. These figures are approximate since this evaluation may have ignored some (hopefully minor) considerations, and because the implementation has plenty of room for application-specific optimizations.

For the appropriate programs of Table 5.1, substituting the value of N_{d0} with N_d in the number of bits of the INS and BDP tables of Table 5.2 gives the "static memory per program" column for Subroutining[7] in Table 3.2 (with $w_i = 8$). For *divide-and-conquer* programs, the l/r, l2/r2 and i/i2 tables are also counted (with $N_s = N_{d0}$, $\bar{N}_{sd} = 1$).

Table 5.1 indicates that the loading of the diagonal restorer is generally so fast that overlapping the loading of a new program's diagonals while restoring a current program's diagonals should not be necessary. However, if overlapping is required, two versions of each of the diagonal restorer's tables are required, almost doubling the static storage requirement.

5.5.3 Conclusions

This chapter has proposed a low-level, diagonal-based ISA language, called ISADL. This language is general except that the array size and any program *recurrence widths* are powers of 2 (not a serious limitation in practice). Since it essentially a low-level version of ISA Subroutining, the results of this chapter settle the issues of whether the Subroutining program compression method is sufficiently flexible and of how to optimize its implementation. It also demonstrates that its implementation can be easily extended for compatability with ISA microprogramming.

While one can devise ISA programs which can be compressed efficiently in ISADL but not in ISAC (and vice versa), one can say that the flexibility of the ISAC and ISADL implementations are both sufficiently high to cover practically all known features of ISA programs. An exception to this occurs with 'flat' ISA programs (see Section 5.4.1.3), which present difficulties for ISADL. We have

[7]Table 5.1 assumed that Subroutining is implemented in the same way as ISADL except that diagonal patterns are not used, ie. $N_p = 0$.

indicated, however, that these programs can be avoided in practice.

In terms of the cost of implementation, ISADL is interesting in that the *matrix generator* can be factored into two components, an $O(\log n)$ area diagonal sequencer, and an $n \times O(1)$ diagonal restorer. Thus, theoretically ISADL has hardware cost of $O(n)$, asymptotically better than ISAC's area cost of $O(n \log n)$. For realistic values of n, eg. $n = 2^8$, the hardware costs also favour ISADL, by what is estimated to be at least order of magnitude, so that ISADL is strongly recommended as the most practical method of program compression for the ISA. ISADL also shows a high performance in program loading time, and thus is well suited for smaller ISA sizes. It also has practical advantages in that (higher-level constructs of) Subroutining, as well as other forms of ISA program compression, can be efficiently incorporated into ISADL.

Thus, in practice, ISADL is concluded to be a superior program compression method to ISAC.

Much detail of the implementation of ISADL has been given in this chapter. This will enable an ISA system designer to realistically evaluate the cost, in terms of design and hardware, of incorporating ISADL into an ISA system. We conclude, from the results of Section 5.5.2, that ISADL would be cost effective for ISAs with an array size $n \geq 16$, and increasingly cost effective for larger values of n. The diagonal restorer cells are estimated to have an area comparable to a boolean ISA (BISA) cell, as described in Section 2.2.

The implementation of the ISADL diagonal restorer is interesting in that operations over $\log n$ bit integers (required by ISAC) has been replaced by bitwise operations, and that it utilizes fully the systolic concept. It has, however, a drawback: to incorporate a new program compression feature into ISADL may require a new mechanism. Thus, a 'Rolls Royce' implementation may be rather complicated. A mitigating factor is that the mechanism might only involve the diagonal sequencer, as was the case for incorporating subroutines in ISADL. The $O(1)$ area of the diagonal restorer cell makes an ISA system implementing ISADL very easily expandable.

One reason for the success of ISADL in program compression is that is de-

scribes ISA programs on what is normally a very natural level: the diagonal. This is demonstrated in Chapter 6, where ISADL is combined with a diagonal-based ISA program proof method. This proof method gives very compact proofs for ISA programs, and also an indirect semantics for ISADL. This has an important practical advantage: that high-level ISA languages should also be based on the diagonal, giving a very simple compilation into ISADL. Appendix 5.A shows how by adding simple macro facilities, the presentation of ISADL programs can be improved.

Future research on ISADL includes examining a specific ISA application, and examining how its programs can be expressed in ISADL for program compression implementation. This is because details of implementation (and hence their cost) depend strongly on the set of ISA program features that need be incorporated. Also a convenient representation of ISADL programs for diagonal sequencer and restorer loading needs to be decided — this involves translation from a high-level (diagonal-based) ISA language, with some optimization heuristics. Particular attention here needs to be given to creating *diagonal patterns.*

Wavefront-based program compression, combined with ISA microprogramming, is both feasible and highly beneficial for improving the ISA model for flexible, large-scale matrix computations.

5.A Appendix: Boolean matrix algorithms in ISADL

This appendix illustrates the utility of ISADL for coding the boolean ISA algorithms of Section 2.2.2. These examples illustrate how typical ISA program are coded in ISADL. They also illustrate how the ISADL encodings give a precise and parameterized expression of these programs (not possible using the standard matrix representation of ISA programs). To improve the readability of these encodings, naming of sub-programs in ISADL encodings is introduced.

The sub-programs are then expanded as simple macros (the rules are given in Section 5.A.7). This suggests how ISADL could be used as a basis for a higher-

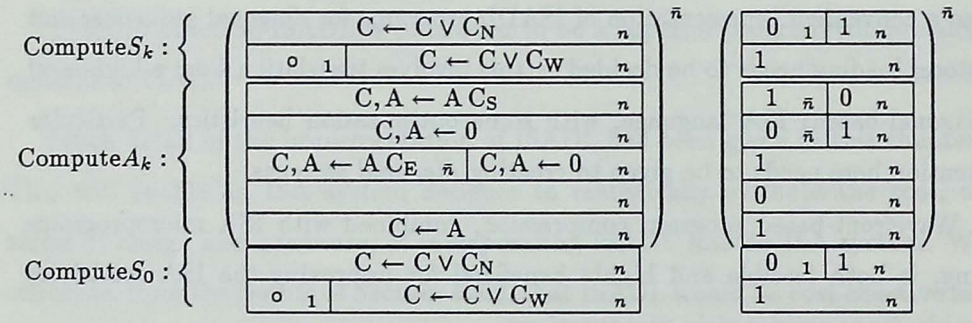
level ISA language. These codings are also used in the evaluation of ISADL in Section 5.5.1.

5.A.1 Red Squares program

The Red Squares program, finding the largest square of 1's in an $n \times n$ boolean matrix, is coded, for an $n \times n$ ISA as:

$$\text{RedSquares}: \left(\begin{array}{|c|}\hline \text{Compute}S_k \\ \hline \text{Compute}A_k \\ \hline \text{Compute}S_0 \\ \hline \end{array} \right)^{k=1..\bar{n}}$$

where the sequencing occurs bottom-up. This expands to the full ISADL encoding (instructions in the centre and selectors to the right with subprogram names annotated to the left):



5.A.2 Matrix multiplication program

This program is for an $n \times m$ ISA, where $m = 2n - 1$, and is of interest since one input matrix is broadcast east, the other is broadcast west, and the result matrix is moved south. This three-way movement is achieved by combining a SE moving wavefront (ie. a regular diagonal) with a SW moving wavefront. Fortunately, this wavefront combination can be expressed in terms of regular diagonals as (instruction part; all selectors are 1):

$$\text{MatMult}: \left(\begin{array}{|c|}\hline \text{NoOp}' \\ \hline \text{M} \;\; \text{NoOp} \\ \hline \end{array} \right)^{k=1..n} \left(\begin{array}{|c|}\hline \text{NoOp}' \\ \hline \text{NoOp} \;\; \text{M} \\ \hline \end{array} \right)^{k=1..\bar{n}}$$

The 'M' instruction is used in four output register mode, and has the effect:

$$(C'_S, C'_E, C'_W) \leftarrow (C_N + C_W * C_E, C_W, C_E)$$

whereas the NoOp and NoOp′ macros have no effect. For the one-output register mode, with a microprogrammed instruction diagonal restorer having $\sigma = 3$, the above instructions become the sub-programs:

M :

$C \leftarrow C + C_N$	1
$C \leftarrow C * A$	1
$C \leftarrow B$	1
$C \leftarrow A$	1
$B \leftarrow C_E$	1
$A \leftarrow C_W$	1

NoOp :

o	1
o	1
o	1
o	1
o	1
o	1

with NoOp′ becoming the empty macro. The boundary '1' in the above sub-programs indicates that the instructions have a recurrence width of 1, ie. repeat across every column, in the contexts where they are used.

5.A.3 Pattern match program

The Pattern Match program, with the pattern length $m \leq n$, illustrates how a 'flat' (load matrix from the east) subroutine, ReadEA', is skewed (see Section 5.4.1.3). It is expressed in ISADL as:

Match^{m} :

Combine$A^m A'$
ReadEA'
$\left(\right.$ ComputeA^k
ScatterSEp
ShiftEA^{k-1} $\left.\right)^{k=1..m}$

with the sub-programs given by (selector diagonals only given if not trivial):

Combine$A^m A'$: {

$C, A \leftarrow A\ C_N$	n

ReadEA' : $\left(\right.$

o	n		
o	$n-k$	$C \leftarrow C_E$	n

$\left.\right)^{k=1..m}$

ComputeA^k :

$A \leftarrow A\ C$	n
$C \leftarrow (C = B)$	n

ScatterSEp :

$C \leftarrow C_N$	n
$C \leftarrow C_W$	n

ShiftEA^{k-1} : {

$A \leftarrow 1$	1	$A \leftarrow C_W$	n

0	1	1	n
1	1	0	n

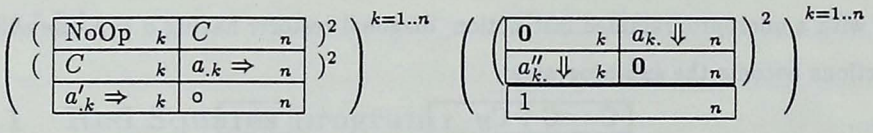
For $m = qn + r$, where $0 \leq r < n$, this program is written:

$\left(\right.$ Match^{r} / Match^{n} $\left.\right)^{q}$

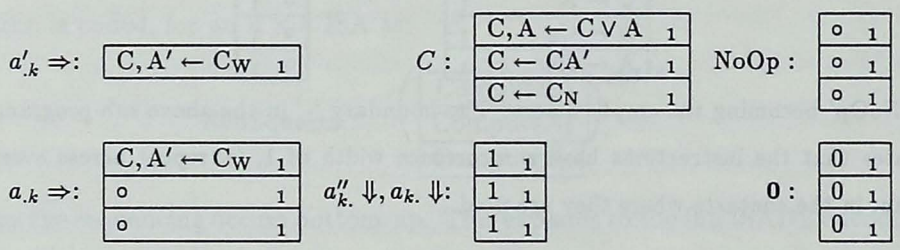
5.A.4 Transitive closure program

The non-microprogrammed transitive closure program, TransClos′, is coded in ISADL, with a rather interesting nested loop structure (differing for the instruction and selector

parts), as:



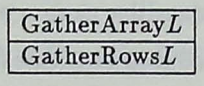
which uses the ISADL sub-programs (indices appearing in the sub-program names here are only annotations to improve readability):



To understand the ISADL encoding directly, consider the execution of iteration k along row i of the ISA. Columns $1..k$ receive the old value of a'_{ik} from $\boxed{a'_{.k} \Rightarrow \ k \mid \cdots}$, which broadcasts it east from cell $(i, 0)$ (row i of the western ISA boundary). Columns $k'..n$ receive the updated value of a_{ik} from $\boxed{\cdots \ k \mid a_{.k} \Rightarrow \ n}$, which broadcasts it east from cell (i, k). This value (being the updated a'_{ik}) is produced by executing (the last instruction of) C at cell (i, k). Now consider the execution of iteration k along column j of the ISA. Rows $1..k$ receive the old value of a''_{kj} from C (1st sub-iteration) meeting $\boxed{a''_{k.} \Downarrow \ k \mid \cdots}$, which broadcasts it south from cell $(0, j)$ (row j of the northern ISA boundary). Rows $k'..n$ receive the updated value of a_{kj} from C (2nd sub-iteration) meeting $\boxed{\cdots \ k \mid a_{k.} \Downarrow \ n}$, which broadcasts it south from cell (k, j). This value (being the updated a''_{kj}) is produced by executing (the last instruction of) C (1st sub-iteration) at cell (k, j).

5.A.5 Associative memory lookup program

Here a pattern $p[1..m]$ is searched for in a key table $K[1..n^2/m][1..m]$ (with associated lookup table $L[1..n^2/m][1..m]$) stored in row major order in an $n \times n$ ISA (with $m|n$). The matching lookup pattern(s) $L[i]$ are then gathered, ready to be output on the southern side of the ISA. This program, called AssocMem, has the most varied and complex diagonal patterns out of all ISA programs of this chapter. Its selectors are again 1's and its instruction part, for the last two sub-programs of the AssocMem program of Section 2.2.2.5, is coded in ISADL as:



The sub-programs expand to:

$$\text{GatherArray}L: \left\{ \boxed{C \leftarrow C_N,\ \text{if B}}_{n} \right.$$

$$\text{GatherRows}L: \left\{ \left(\begin{array}{|l|l|l|} \hline C \leftarrow C_W & C \leftarrow C_W,\ \text{if A} & C \leftarrow C_W \\ \hline \text{o} & C \leftarrow B' & \text{o} \\ \hline \end{array}_{\bar{k}\ \ k\ \ m} \right)^{k=1..m}$$

5.A.6 Median finding program

Finding the median of an $L \times L, L = 2K+1$ window of an $n \times n$ image (boolean matrix) is implemented here using a boolean ISA performing bitwise addition using ring shift registers R_1 and R_2 (at least of size $2 \log L$ bits) The overall program is divided into two main stages, summing the L pixels to the west, and adding the L (row) sums to the north:

$$\text{MedFind}: \left(\begin{array}{|l|} \hline \text{Reset}^{\lceil \log L \rceil}(R_1) \\ \hline \text{Reset}^{\lceil \log k'L \rceil}(R_2) \\ \hline \text{Carries}^{\lceil \log k' \rceil}(R_2) \\ \hline \text{AddRowSum}_N^{\lceil \log L \rceil}(R_1, R_2) \\ \hline \end{array} \right)^{k=1..L} \left(\begin{array}{|l|} \hline \text{ResetCopy}^{\lceil \log k' \rceil}(R_1, R_2) \\ \hline \text{AddPixel}_W^{\lceil \log k' \rceil}(R_1) \\ \hline \end{array} \right)^{k=1..L} \boxed{R_1 \leftarrow A}_{n}$$

The selectors for each of these sub-programs are 1's; the instruction parts are given by:

$$\text{Reset}^b(R_1): \{ \quad \left(\boxed{B \leftarrow R_1; R_1^-}_{n} \right)^b$$

$$\text{Carries}^b(R_2) \{ \quad \left(\boxed{\text{HA}(R_2, D)\ R_2^+}_{n} \right)^b$$

$$\text{AddRowSum}_N^b(R_1, R_2): \left\{ \left(\begin{array}{|l|} \hline \text{FA}(R_2, B, D); R_2^+ \quad n \\ \hline R_1 \leftarrow B; R_1^+ \quad n \\ \hline B \leftarrow C_N \quad n \\ \hline C \leftarrow R_1 \quad n \\ \hline \end{array} \right)^b \right.$$

$$\text{ResetCopy}^b(R_1, R_2): \{ \quad \left(\boxed{R_2 \leftarrow R_1; R_1^-; R_2^-}_{n} \right)^b$$

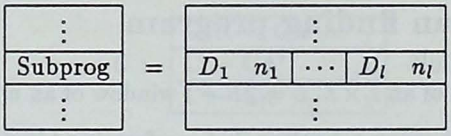
$$\text{AddPixel}_W^b(R_1): \left\{ \left(\begin{array}{|l|} \hline \text{HA}(R_1, D); R_1^+ \quad n \\ \hline C \leftarrow D \quad n \\ \hline D \leftarrow C_W \quad n \\ \hline \end{array} \right)^b \right.$$

Recall that $\text{HA}(R, C)$ denotes $(R, C) \leftarrow (R \oplus C, RC)$, $\text{FA}(S, B, C)$ can be implemented by the sequence '$\text{HA}(R, B); \text{HA}(R, C); C \leftarrow C \vee B$' and that appending an instruction with '$;R^+$' ('$;R^-$') indicates that the ring-register operand R_1 is shifted forward (back) after execution of that instruction (and similarly for R_2). This program is interesting for the purpose of ISADL implementation: some nested loops have their bounds dependent on the logarithm of the outer loop index. This requires an extra capability in the ISADL diagonal sequencer.

5.A.7 ISADL macro rules

In Appendix 5.A.1-5.A.6, various 'macro'-type conventions were introduced to improve the readability of ISADL programs. These conventions are defined semi-formally here.

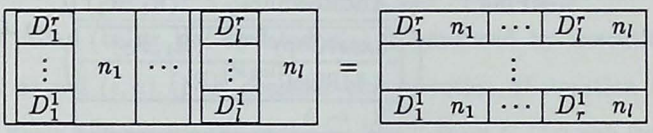
1. Sub-program macro rule: if Subprog= $\boxed{D_1 \;\; n_1 \;|\; \cdots \;|\; D_l \;\; n_l}$, then:



This is used to expand macros in programs ReqSquares, MedFind, Match^m and AssocMem, and is a tidy way to sequence sub-program calls inside tables.

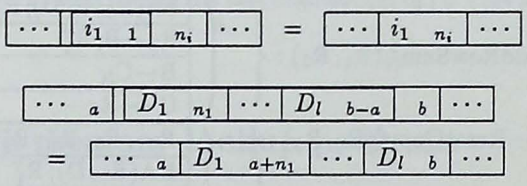
2. Nesting ISADL sub-programs within a diagonal:

This can occur if all sub-programs have the same length (in the following case r):



This is used to expand macros in programs RedSquares, MedFind and Match (with $l = 1$); and in programs MatMult and TransClos′ (with $l = 2$).

3. Nested sub-diagonal simplification:



This is used to simplify macro expansions in programs MatMult and TransClos′.

5.B Appendix: Implementation of ISADL

5.B.1 The diagonal load algorithm

The algorithm setting the L bit, used to selectively load ISADL diagonals into diagonal restorer cells, is described here. The algorithm is based the small boolean stack E

(where the context is clear, E also denotes the E stack top), and a boolean variable E' for each cell in the diagonal restorer.

Loading the initial value of a component of an ISADL diagonal requires that the E' (E) bit is set in the cells where that component begins (ends). To load a whole diagonal, assume that the E' (E) bit is high only at cell 1 (cell n). If the component extends over w columns, let a consecutive high E' and high E pair, separated by w units, be denoted 'E'-$_w$-E'. Here, w corresponds to the w of equation (5.2), and can be counted in $(\log w - 1)$ steps using the systolic input bit h, which travels across the diagonal restorer with the loading logic instruction codes. If the component is a simple instruction, the L bit is set between E'-$_w$-E pairs, and the instruction is loaded. Otherwise, the process is repeated recursively, and if the component has recurrence width n_l where $n_l = qw$ and $q > 1$, q E'-$_{n_l}$-E must be first set between each E'-$_w$-E. Since this process occurs in a time-skewed (instruction systolic) fashion across the diagonal restorer, the systolic bit h can be used to efficiently detect if the current cell is between an E'-$_w$-E.

The following algorithm gives the required diagonal loading logic instruction sequence for cells 1 to n (the value of h read in at cell 1 is put in brackets ([0] is default), and meta-instructions are in **bold**). Note that $n_0 = 0$ and $\tilde{E}$ denotes the negation of the 2nd top element of E.

$\underline{d\_l}(i, w) =$

$(L, h) \leftarrow (h \vee E', E' \vee h\bar{E})$ (* load i*)

$\underline{d\_l}($ | D_1 n_1 | D_2 n_2 | ... | D_l n_l | $, w) =$

$E \leftarrow \text{push}(E)$

if $n_l < w$ **then** (* put w/n_l E'-$_{n_l}$-Es inside current E'-$_w$-E *)

$\underline{\text{new}}(E, n_l - 1, \log n_l)$ (* set every $\bar{n}_l$th $(\text{mod} n_l)$ E after high E'*)

$(E', h) \leftarrow (E' \vee h, E\tilde{E})$

(* set E' if a high E (and not the last high E) in previous cell*)

for $k := 1$ **to** l **do**

$\underline{\text{new}}(E, n_k - n_{k-1} - 1, \lceil \log(n_k - n_{k-1}) \rceil)$

(* set every $(n_k - n_{k-1})$th $(\text{mod} 2^{\lceil \log(n_k - n_{k-1}) \rceil})$ E after high E'*)

$(E, h) \leftarrow (Eh, E' \vee h\bar{E})$ (* select first of these after a high E'*)

$\underline{d\_l}(D_k, n_k - n_{k-1})$

if $k < l$ **then** (* set E' to E of the previous cell*)

$(E', h) \leftarrow (h, E)$

else (* set E' only in 1st cell (for next diag.) *)

$(E', h) \leftarrow (h, 0)$ [1]

$E \leftarrow \text{pop}(E)$

$\underline{\text{new}}(E, \delta, l) =$

$(E, h) \leftarrow (E' \vee h, E' \vee h)$ (* set E high if after any high E'*)

for $k := 1$ **to** l **do**

$(E, h) \leftarrow (Eh, E \oplus h)$ $[\bar{\delta}_k]$ (* select odd (even) Es if $\delta_k = 0$ (1)*)

where δ_k denotes the kth bit of δ (1st bit is least significant). $\underline{new\_E}(\delta)$ sets E high at intervals of $2^{\lfloor \log \delta \rfloor + 1}$ and an offset of δ, bit from the last high E'. The correctness of this procedure relies on the fact that the distance between consecutive high $E' - E'$ pairs is always a power of 2, and is divisible by n_l.

Optimizations

It is necessary for at least small number of diagonals to be loaded within a period of $\Omega(n)$ ISA instruction cycles. Since the period of the above algorithm is $O(\log n)$, determined by calls to the $\underline{\text{new}}$ procedure, these optimizations reduce the number of these calls. These optimizations require negligible extra hardware for their implementation.

Since in this case the corresponding E values have already been calculated, the call:

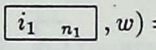
$$\underline{\text{new}}(E, n_l - n_{l-1} - 1)$$

can be replaced by the instruction:

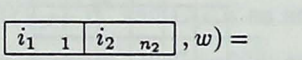
$$E \leftarrow \text{pop}(E)$$

after inserting an extra '$E \leftarrow \text{push}(E)$' instruction immediately before the **for** loop.

Diagonals that often occur in ISA programs are uniform, except possibly that one edge instruction is different. Here, useful optimizations would be:

$\underline{d\_l}($  $, w) =$

$(L, h) \leftarrow (E' \vee h, E' \vee h\bar{E})$ (* load i_1 *)

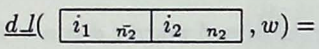
$\underline{d\_l}($  $, w) =$

if $n_l < w$ **then**

$\vdots$ (* as above *)

$(L, h) \leftarrow (h, E' \vee h\bar{E})$ (* load i_2 everywhere *)

$L \leftarrow E'$ (* load i_1 at left end *)

$\underline{d\_l}($  $, w) =$

if $n_l < w$ **then**

$\vdots$ (* as above *)

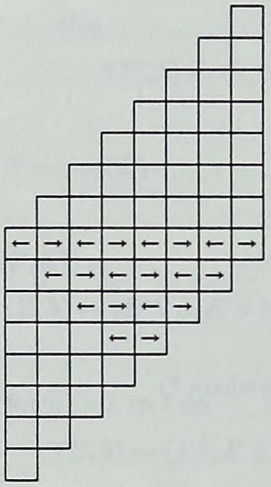
$(L, h) \leftarrow (h, E' \vee h\bar{E})$ (* load i_1 everywhere *)

$L \leftarrow E$ (* load i_2 at right end *)

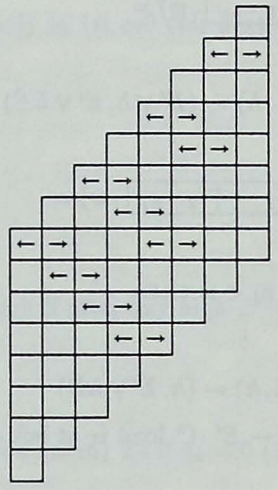
There are further optimizations that can be taken, such as, when loading several diagonals of the same recurrence width m say, calculating once and then re-using the E'-$_m$-E pairs for each such diagonal. Further speedup can be achieved by making the systolic input h into a bit vector of size l_0, where $1 \leq l_0 \leq \log n$. The <u>new</u> procedure can now select a high E at intervals of 2^{l_0} during a single instruction, resulting in a factor l_0 speedup.

5.B.2 Transforming 'flat' ISA programs

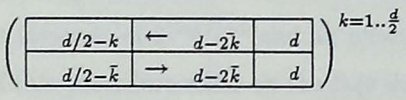
This appendix illustrates the efficient transformation of the *odd-even transposition sort* and *triangle merger* programs [46] outlined in Section 5.4.1.3. These transformations are achieved by 'skewing' the comparison-exchanges, and inserting two empty 'anti' diagonals. This is illustrated for the 'forward' *triangle merger* (over width d) program of Figure 5.11 (all selectors are 1's; the case for the 'reverse' *triangle merger* is similar).



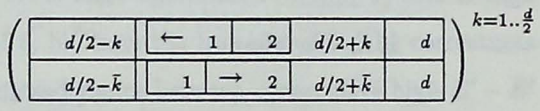
(a) original matrix, $d = n = 8$



(b) transformed matrix $d = n = 8$



(c) original ISADL encoding



(d) transformed ISADL encoding

Figure 5.11: 'Forward' triangle merge (over width d) instructions for an $n \times n$ ISA

5.B.3 Binary recursive programs in ISADL

To illustrate incorporating more sophisticated Subroutining concepts into ISADL (Section 5.4.3), this appendix demonstrates how binary recursive ISA programs can be

implemented in ISADL. This also incorporates the concepts of *diagonal patterns* (Section 5.3.3) and *spliced diagonals* (Section 5.4.1.2). Iteration is a form of linear recursion which can be efficiently implemented in ISADL providing the differences between successive iterations are 'acceptable'. By unfolding binary recursion into linear recursions, binary recursive ISA programs (which typically incorporate some *divide-and-conquer* strategies) can be coded efficiently in ISADL. Consider the ISADL program $P(n)$:

$$\begin{aligned} P(1) &= P \\ P(k+1) &= A_{2^k}; P(k); B_{2^k}; P(k); C_{2^k} \end{aligned}$$

where A_n, B_n, C_n are $O(n)$ ISA subroutines (usually, at least one of them is $\Theta(n)$). An example of a binary recursive ISA algorithm is the algorithm for finding all cut-points of a graph of [48, Sect.6], which has $A_{2^k} = \text{PUSH}; (\text{INC}; \text{ROT}_{2^{k+1}})^{2^k}$, $B_{2^k} = (\text{INC}; \text{ROT}_{2^{k+1}})^{2^k}$ and C_{2^k} is an empty program.

As an example, $P(8)$ unfolds to (semicolons omitted):

$$\begin{array}{l} A_4(A_2(A_1\ P\ B_1\ P\ C_1) \\ B_2(A_1\ P\ B_1\ P\ C_1)C_2) \\ B_4(A_2(A_1\ P\ B_1\ P\ C_1) \\ \ B_2(A_1\ P\ B_1\ P\ C_1)C_2)C_4 \end{array}$$

which is rearranged to:

$$\begin{array}{l} (A_4\ A_2\ A_1)\ P \\ (B_1\ P\ C_1\ B_2\ A_1\ P) \\ (B_1\ P\ (C_1\ C_2)B_4(A_2\ A_1)P) \\ (B_1\ P\ C_1\ B_2\ A_1\ P) \\ B_1\ P\ (C_1\ C_2\ C_4) \end{array}$$

Thus, $P(n)$ is then coded in ISADL as follows:

$$\begin{array}{l} (A_{2^k})^{k=b_{n/2}..1}; P; \\ (B_1; P; (C_{2^k})^{k=1..b_i}; B_{2^{b_i+1}}; (A_{2^k})^{k=b_i..1}; P)^{i=1..n/2-1}; \\ B_1; P; (C_{2^k})^{k=1..b_{n/2}} \end{array}$$

where the *binary coefficient* b_i is given by the lowest set bit in i's binary representation:

$$b_i = \min\{j : 0 \leq j < \log n | i \bmod 2^j = 1\}$$

An efficient ISADL implementation of this requires that:

- each diagonal of A_{2^k} can be merged with a diagonal of C_{2^k} into a diagonal pattern, and any diagonal of B_{2^k} (or $B_{2^{k+1}}$) can be merged into one of these. Thus, all diagonal patterns for these subroutines initially correspond to $2^{b_{n/2}}$, and the patterns are varied during the nested (k) iterations as is done in Section

5.4.1.2. If the diagonals of A_{2^k} or B_{2^k} have their boundaries dependent on 2^k, this in turn requires that A_{2^k} is $\Theta(2^k)$ (to accommodate duplicating splicing boundaries by 2^{k-1} shifts), and C_n is $\Omega(1)$ (to accommodate removing odd/even splicing boundaries).[8] This is sufficient to ensure the correct boundaries are available in the diagonal restorer at any time.

- the diagonal sequencer be able to generate the sequences $1..b_i$ ($b_i..1$) and dynamically allocate the values of any loop bounds (usually $2^k, 2^k-1$) within A_k, B_k and C_k. It must also command the *diagonal restorer* to change its *splicing boundaries* appropriately.

[8]This can always be achieved by inserting a few diagonals of 'NoOp's in C_n, which can be efficiently merged with any diagonal on A_n or B_n. This will not increase the time complexity of $P(n)$.

Chapter 6

A Micro-level Semantics for the Microprogrammed ISA

6.1 Introduction

Parallel computing presents many challenges to the programmer, who must now consider issues of synchronization and communication as well as computation. As a result of this, the debugging of parallel programs can be extremely difficult, due to non-determinism in parallel program execution and the fact that uniprocessor debugging techniques (eg. tracing variables) disturb the timing of parallel execution and often yield unmanageable volumes of data [39, pp12-15]. This has lead to an interest in program verification techniques for parallel computers, in which the correctness of a program can be established (at least in part) at coding time.

Chapters 3, 4 and 5 have demonstrated that the microprogrammed ISA and wavefront-based ISA program compression can enhance considerably the efficiency and flexibility of implementing the wavefront programming model on fine-grained meshes. Both of these concepts have also touched upon the high-level language aspect of the issue of confident and efficient programmability. This chapter examines the verification aspect of this issue for microprogrammed ISA programs written in a wavefront-based language (ie. ISADL).

While both of these concepts are based on the intuitive wavefront mesh pro-

gramming model, it may furthermore be asked whether this model can be given a formal semantic basis. An affirmative answer to this question is important for two reasons: Firstly, it would give more reason to believe that the wavefront model is appropriate for developing high-level language concepts, and not just efficient control structures. Secondly, it would provide a reference point to confident programmability by defining what is the actual behavior of a wavefront-based program, so that it may be compared with its intended behavior. This chapter seeks such an answer for the features of the wavefront model that can be supported by the microprogrammed ISA.

The microprogrammed ISA (μISA), being a new model of parallel computation, needs to be formally defined, and the well-developed weakest precondition semantics [6] is an appropriate choice. Fortunately, since the μISA is synchronous, deterministic and relatively simple, its semantic modelling and associated program verification techniques are straightforward, as compared with more general models of parallel computation. This chapter gives a weakest-precondition semantics for the μISA, which is modelled at the micro-level, for the sake of simplicity. However, it can easily be extended to the macro level, since the same principles apply to macros as to micros.

The ISA, and in particular the μISA, are relatively difficult to program, as compared to SIMD meshes and uniprocessors. One approach to simplify the programming of the μISA was given in Section 4.6 — this was a discipline for programming the μISA at the macro level. Another approach is to compensate for programming difficulties by developing useable program verification (ie. proof) methods for the μISA, in order that the μISA can be used confidently in practice. Hence, this chapter develops a useable μISA program proof method based on the weakest precondition semantics.

The microprogrammed ISA semantics of [34] is based on the concept of global predicates to specify the pre/post-conditions of the ISA. It is also based on a 'serialization principle', in which during a global time snapshot over the array, the operations at each component are carried out in an arbitrary, serial order. This leads to a simple and elegant semantics, which is amenable to mechanized

and manual program verification.

However, for *wavefront* architectures such as the μISA (Chapter 4) global time snapshots are not a convenient frame of reference, so that the direct application of the semantics for program verification is difficult. Hence, this chapter also develops a proof method based on the semantics that is convenient for the verification of μISA programs expressed in diagonal or wavefront-based languages such as ISA Subroutining (Section 3.6) and ISADL (Chapter 5). This can be achieved by combining two concepts:

1. describing the state of the array as a 'predicate array' of the states of its components, and then to *parallelize* the weakest precondition operation. ie. the weakest precondition of the 'predicate array' due to a global snapshot (array) of operations or instructions is expressed in terms of the weakest precondition of each operation component applied to its respective predicate component. Communication effects, including boundary conditions, are modelled by a technique called *communication register time-stamps* (CRTS).

2. incorporating the diagonal or wavefront as the basic semantic unit, which requires a *time-skewed* (rather than a global time) frame of reference. This frame of reference follows the wavefront as it propagates through the array.

The method has an interesting parallel with program compression, since it can compress a (expanded) proof using the semantics directly in an analogous way to that in which (the matrix form of) an ISA program can be compressed by ISADL. This method is intended to be as far as possible compatible with the programmer's (intuitive) view of the μISA.

This proof technique may be used in the following ways:

- to verify that a μISA program implements a given series of recursion equations/boundary conditions.

- to derive the (weakest) series of recursion equation/boundary conditions that a μISA program implements.

- to be used to optimize microcodes of a given μISA program (in conjunction with some semi-automated 'assistant' system).

Sections 6.2 and 6.3 give extended versions of the syntax and semantics of the μISA microlanguage of [34]. Section 6.4 describes the programmer's view of the μISA, outlining accordingly how a program verification technique can approximate the programmer's intuition. Using most of these ideas, Section 6.5 develops the wavefront-based proof technique. It also establishes various results to demonstrate the validity and generality of the method with respect to the semantics. Section 6.6 illustrates the method on some non-trivial examples, and introduces the use of boundary conditions and invariants. Section 6.7 discusses the relative merits of this technique, and its possible extensions.

This chapter gives a semantics for the μISA (and hence ISA) model, forming an important step in the development of this model. The proof method also effectively gives a semantics for the ISADL or Subroutining programming languages. The CRTS technique of the proof method also models general boundary conditions, essential for systolic algorithms, which the semantics itself does not provide.

Sections 6.2 and 6.3 are largely based on Dr Lender's semantic modelling of the μISA from [34]; the rest of this chapter is the author's own work.

This chapter is not essential reading for the rest of this thesis, and only a reader interested in semantics or program verification is likely to be interested in the (perhaps unavoidably complicated) details. However, the reader might still find it rewarding to obtain a feeling for the semantics and proof method by looking at the examples in Sections 6.3, 6.4 and 6.5.1.

6.2 Syntax of an ISA microlanguage

The syntax of a microprogramming language for an $m \times n$ microprogrammed ISA is given here. This language is also wavefron-based, but for the sake of a convenient semantic definition, differs from the Subroutining or ISADL languages already encountered. The language's syntax uses the BNF notation, along with

the following notations: '$[\ ]_k$' denotes k repetitions of the enclosed material; ';' denotes sequential composition. A program consisting of $K > 0$ micro diagonals has the syntax:

$$
\begin{aligned}
\text{< program >} &::= \text{< line of stat >< line of sel >} \\
&\quad [;\text{< line of stat >< line of sel >}]_{K-1} \\
\text{< line of stat >} &::= \text{< stat1 >;< stat2 >} \quad [,\text{< stat1 >;< stat2 >}]_{n-1} \\
\text{< stat1 >} &::= \text{< inputregs >} \leftarrow \text{< exp1s >} \mid \text{SKIP} \\
\text{< stat2 >} &::= \text{< accregs >} \leftarrow \text{< exp2s >} \mid \text{SKIP} \\
\text{< inputregs >} &::= \text{< inputreg >} [,\text{< inputreg >}] \\
\text{< accregs >} &::= \text{< accreg >} [,\text{< accreg >}] \\
\text{< exp1s >} &::= \text{< exp1 >} [,\text{< exp1 >}] \\
\text{< exp2s >} &::= \text{< exp2 >} [,\text{< exp2 >}] \\
\text{< inputreg >} &::= \textit{a local input register,} \\
&\quad \textit{not a communication register} \\
\text{< accreg >} &::= \textit{a local accumulating register} \\
\text{< exp1 >} &::= \textit{expression involving any register} \\
\text{< exp2 >} &::= \textit{expression not involving a neighbour's} \\
&\quad \textit{communication registers} \\
\text{< line of sel >} &::= \text{< sel >} \quad [,\text{< sel >}]_{m-1} \\
\text{< sel >} &::= 0 \mid 1
\end{aligned}
$$

So far, the syntax is consistent with that of [34], and it encompasses simultaneous assignments. Further restrictions on the simultaneous assignments allowed are given in the semantics of the next section. The syntax will now be more closely defined in terms of traditional ISA models which use single output communication register per cell [21]. In this chapter, the μISA cells are assumed to have four input registers, one communication register, two accumulating registers, support boolean conjunction and disjunction, and support integer addition, multiplication, and maximum. The extra syntax rules are then:

$$\text{< accreg >} ::= \text{A} \mid \text{B} \mid \text{C}$$

$$
\begin{aligned}
< \text{inputreg} > &::= \text{N}|\text{S}|\text{E}|\text{W} \\
< \text{exp1} > &::= \text{C}_\text{N}|\text{C}_\text{S}|\text{C}_\text{E}|\text{C}_\text{W} \\
< \text{exp2} > &::= < \text{reg} > | < \text{reg} >< \text{OP} >< \text{reg} > | \max(< \text{reg} >, < \text{reg} >) \\
< \text{reg} > &::= < \text{accreg} > | < \text{inputreg} > \\
< \text{OP} > &::= \wedge | \vee | + | *
\end{aligned}
$$

In these rules, $\text{C}_\text{N}, \text{C}_\text{E}, \text{C}_\text{W}, \text{C}_\text{S}$ are the communication (C) registers of the north, east, west and south neighbours. To more realistically model the ISA, it is also required that any 'inputreg' (eg. N) used in 'stat2' (eg. $\text{C} \leftarrow \text{C} * \text{N}$) of a statement must also be updated in the corresponding 'stat1' (eg. $\text{N} \leftarrow \text{C}_\text{N}$). This ensures that a value read into an 'inputreg' cannot be used on subsequent cycles (and will not be introduced into the corresponding precondition).

6.3 Semantics for the ISA microlanguage

This approach for defining a semantics of the ISA microlanguage is based on the 'weakest precondition' operator (cf. [6]). For the sake of convenience, a program P having K diagonals (ie. period of K) using a (μ, σ) wavefront on an $m \times n$ μISA, is given in the *wavefront* form:

$$
\begin{aligned}
P &= \mathop{;}_{k=1,K} (lstat_k\ lsel_k) \\
&= \mathop{;}_{k=1,K} \left(\mathop{,}_{j=1,n} (stat1_{jk}; stat2_{jk}) \quad \mathop{,}_{i=1,m} (sel_{ik}) \right)
\end{aligned}
$$

where "$\mathop{;}_{l=a,b} P(l)$" intuitively denotes '$P(a); P(a+1); \ldots; P(b)$' and "$\mathop{,}_{l=a,b} P(l)$" textually represents the diagonal composed of the elements '$P(a),\ldots,P(b)$'.

P is first syntactically transformed into a series of global 'snapshot' instruction matrices, using the microprogrammed ISA transformation $k_{ijt} = t - \mu(i-1) - \sigma(j-1)$, where the index t denotes the time step, and the indices i and j denote the position of the processing elements:

$$
P' = \mathop{;}_{t=1,T} \left(\mathop{;}_{i=1,m} \left(\mathop{;}_{j=1,n} P1_{ijt} \right) \right); \left(\mathop{;}_{i=1,m} \left(\mathop{;}_{j=1,n} P2_{ijt} \right) \right)
$$

where, for the microprogrammed ISA[1], $T = K + \mu(m-1) + \sigma(n-1)$, and, for $k = k_{ijt}$:

$$P1_{ijt} = \begin{cases} PE_{ij}.stat1_{jk} & \text{if } 1 \leq k \leq K \wedge sel_{ik} = 1 \\ \text{SKIP} & \text{otherwise} \end{cases}$$

$$P2_{ijt} = \begin{cases} PE_{ij}.stat2_{jk} & \text{if } 1 \leq k \leq K \wedge sel_{ik} = 1 \\ \text{SKIP} & \text{otherwise} \end{cases}$$

This amounts to the postulate that $\text{wp}(P, Q) = \text{wp}(P', Q)$ for all Q. As is proved in [34, Sect.6.1] (using the fact that at time t within P', communication register access and update over the array occurs in the two non-overlapping phases $(;_i\,;_j P1_{ijt})$ and $(;_i\,;_j P2_{ijt})$ in P'), the order of execution along the i, j indices is irrelevant. In this sense, it can be said that this semantics embodies a 'serialization principle'. For this reason, it is necessary to decompose the instructions into two phases. The meaning of $PE_{ij}.\text{stat1}_{jk}$ is: 'statement stat1_{jk} in processing element PE_{ij} at time step $t = k + \mu(i-1) + \sigma(j-1)$'. The semantics of the different mechanisms is, for $L > 0$:

$$\begin{aligned} \text{wp}(S_1; S_2, Q) &= \text{wp}(S_1, \text{wp}(S_2, Q)) \\ \text{wp}(\text{PE}_{\text{ij}}.\text{SKIP}, Q) &= Q \\ \text{wp}(PE_{ij}.(\text{reg}^1, \ldots, \text{reg}^L \leftarrow \text{exp}^1, \ldots, \text{exp}^L), Q) &= Q^{\text{reg}^1_{ij}, \ldots, \text{reg}^L_{ij}}_{\text{exp}^1{}_{ij}, \ldots, \text{exp}^L_{ij}} \end{aligned}$$

where:

- $Q^{\text{r}^1,\ldots,\text{r}^L}_{\text{e}^1,\ldots,\text{e}^L}$ is the predicate Q where all occurrences of the variables $\text{r}^1, \ldots, \text{r}^L$ have been simultaneously replaced by $\text{e}^1, \ldots, \text{e}^L$, respectively. The variables must be distinct.

- $\text{reg}^l_{ij}, 1 \leq l \leq L$ is the register (input or accumulating) reg^l of PE_{ij} (for example A_{ij}).

- $\text{exp}^l_{ij}, 1 \leq l \leq L$ is the expression exp^l calculated in PE_{ij}. Thus every occurrence of 'reg' is replaced by reg_{ij} (for example, 'A' is replaced by

[1]Generally, T represents the last time when P is in the array, ie.: $T = \max\{t | k_{ijt} = K, 1 \leq i \leq m, 1 \leq j \leq n\}$.

A_{ij}), every occurrence of C_N is replaced by $C_{i(j-1)}$, every occurrence of C_E is replaced by $C_{(i+1)j}$, every occurrence of C_W is replaced by $C_{(i-1)j}$, and every occurrence of C_S is replaced by $C_{i(j+1)}$.

This semantics is in fact suitable for any deterministic, communication register-based mesh for which program control is data independent. For this purpose, a similar microlanguage should be chosen, with a corresponding transformation into 'global' snapshot instruction matrices.

Example — the LCS program

The LCS program (see Section 4.2.1; a similar program is found in [34, Sect.7.1]) has $K = 4, \mu = 3, \sigma = 2$ and is written syntactically as:

$$\begin{aligned}
\mathrm{LCS} \;=\; & \left(\underset{j=1,N}{,} \; (\mathrm{N} \leftarrow \mathrm{C_N}; \mathrm{A} \leftarrow \mathrm{A} * \mathrm{N}) \;\; \underset{i=1,M}{,} \; (1) \right) \\
& \left(\underset{j=1,n}{,} \; (\mathrm{N} \leftarrow \mathrm{C_N}; \mathrm{A} \leftarrow \max(\mathrm{A}, \mathrm{N})) \;\; \underset{i=1,m}{,} \; (1) \right) \\
& \left(\underset{j=1,n}{,} \; (\mathrm{W} \leftarrow \mathrm{C_W}; \mathrm{C} \leftarrow \mathrm{W}) \;\; \underset{i=1,m}{,} \; (1) \right) \\
& \left(\underset{j=1,n}{,} \; (\mathrm{SKIP}; \mathrm{C} \leftarrow \max(\mathrm{C}, \mathrm{A}) \;\; \underset{i=1,m}{,} \; (1) \right)
\end{aligned}$$

For a 3×3 ISA, with '$*, \rightarrow, \downarrow$, m' denoting '$\mathrm{A} \leftarrow \mathrm{A} * \mathrm{N}$, $\mathrm{C} \leftarrow \mathrm{W}$, $\mathrm{A} \leftarrow \max(\mathrm{A}, \mathrm{N})$, $\mathrm{C} \leftarrow \max(\mathrm{C}, \mathrm{A})$' respectively, the corresponding 'snapshot' matrices $(P2_{ij})_t$ for $1 \leq t \leq 6$ are (in row-major order, i, j subscripts suppressed, and 'SKIP' is default entry):

*		

↓		

→	*	

m	↓	
*		

	→	
↓		

	m	
→	*	

The final snapshot occurs at $T = 14$ with only 'm' in cell $(3, 3)$ of the ISA. From this, the reader can visualize the transformed program LCS′.

6.4 A programmer's view of ISA program verification

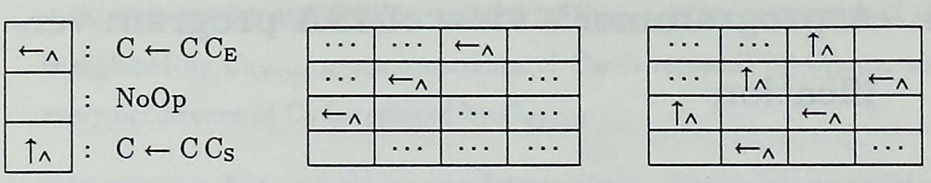
This section illustrates how the programmer might reason about the μISA (ISA) during the development of a program. This reasoning is consistent with wavefront concepts. It is desired that a proof method mirrors this reasoning process as much as possible, because this makes the proof method easier to use. Thus, the priority here is to make a proof method practical, even at the expense of some of its 'theoretical elegance'.

The example used to illustrate ISA programmer's reasoning is a simplified version of the Red Squares program (see Section 2.2.2.1). The program implements on an $n \times n$ ISA the following set of recursion equations (for $1 \leq i, j \leq n$ and $1 \leq k < n$):

$$\begin{aligned} A'^{k}_{i,j} &= A^{k}_{i,j} A^{k}_{i,j+1} \\ A^{k+1}_{i,j} &= A'^{k}_{i,j} A'^{k}_{i+1,j} \end{aligned}$$

where, for the sake of simplicity, the boundary conditions are left unspecified. These equations suggest a program consisting of $n-1$ uniform repetitions, in which the kth repetition at typical cell (i,j) begins with the value of $A^{k}_{i,j}$ in its communication register. Hence, when cell $(i, j+1)$ similarly has ready $A^{k}_{i,j+1}$, cell (i,j) reads this value to produce $A'^{k}_{i,j}$. Then, when cell $(i+1, j)$ has similarly computed $A'^{k}_{i+1,j}$, cell (i,j) reads this value to produce $A^{k+1}_{i,j}$. The associated ISA wavefronts, passing through a typical cell (i,j), are illustrated in Figure 6.1.

To verify that such a scheme correctly implements the recursion equations, the programmer considers the 'state', ie. the relevant values of the (C) registers, around cell (i,j) – these are annotated in braces below — as the respective



(a) for the '$\leftarrow_\wedge$' wavefront (b) for the '$\uparrow_\wedge$' wavefront

Figure 6.1: Execution of simplified Red Squares program around 'typical' cell $(i,j) = (2,2)$ for a 4×4 ISA

sequence of wavefronts pass through it:

$$
\begin{array}{l}
\{\mathrm{C}_{ij} = A^k_{i,j}\} \\
\mathrm{NoOp}; \\
\{\mathrm{C}_{ij} = A^k_{i,j}\}\ \{\mathrm{C}_{i(j+1)} = A^k_{i,j+1}\} \\
\mathrm{C} \leftarrow \mathrm{C}\,\mathrm{C}_\mathrm{E}; \\
\{\mathrm{C}_{ij} = A^k_{i,j} A^k_{i,j+1} = A'^k_{i,j}\} \\
\mathrm{NoOp}; \\
\{\mathrm{C}_{ij} = A'^k_{i,j}\}\ \{\mathrm{C}_{(i+1)j} = A'^k_{i+1,j}\} \\
\mathrm{C} \leftarrow \mathrm{C}\,\mathrm{C}_\mathrm{S}; \\
\{\mathrm{C}_{ij} = A'^k_{i,j} A'^k_{i+1,j} = A^{k+1}_{i,j}\}
\end{array}
$$

Here, the programmer begins with the assumption that C_{ij} contains the correct starting value $A^k_{i,j}$; then ($\sigma = 1$) NoOp's are required to ensure that the corresponding assumption holds in cell $(i, j+1)$ before forming $A'^k_{i,j}$, by reading $\mathrm{C}_{i(j+1)}$. Similar reasoning holds for the next step.

The above reasoning may be thought of as an informal or a 'programmer's verification' of the Red Squares program. The annotations may be thought of as being under the scope of some kind of universal quantification over the subscripts. In the SIMD version of this algorithm (using a $(\mu, \sigma) = (0, 0)$ wavefront, and hence NoOp's are no longer required), this quantification corresponds to global time snapshots over the array, so that verification is more straightforward. However, in the ISA version, quantification occurs in a 'time-skewed' sense in which the predicate describing the state of a cell holds when the respective wavefront passes through it. This concept is more difficult to formalize.

A fair comment is that since the programmer reasons about how the 'state' around a typical array cell changes as execution proceeds, the weakest precondition semantics can never be fully appropriate to base such a proof method on.

This is because weakest precondition semantics reasons in the reverse direction. While this gives the semantics elegant theoretical properties, it widens the gap between formal and intuitive reasoning about programs. Such problems aside, a wavefront-based proof method will now be introduced, which otherwise attempts to mirror this intuitive 'programmer's verification'. This can be achieved by using the concept of a time-skewed 'state' around a typical array cell, with the 'relevancy' being modelled by the selective introduction of communication register time-stamps[2].

6.5 A wavefront-based proof method

This section presents a proof method based on the microprogrammed ISA semantics. The method is suitable for manual proofs. This section proper introduces the method and its main concepts, which are illustrated by a simple example in Section 6.5.1. That the method is indeed valid with respect to the μISA semantics is demonstrated by the results of Section 6.5.2 — this section is not essential for further reading. Section 6.5.3 gives an interpretation for the CRTS introduced by the method under which it yields the *weakest* or most general solutions — this is important for the intuitive understanding of the method. Section 6.5.4 shows how ISADL is incorporated into the method (giving an effective semantics for ISADL) — however, this can more easily be understood merely be following the examples of Section 6.6.

The semantics considers a general program P for an $m \times n$ μISA of period K and microprogramming parameters μ and σ. The first micro of P enters the ISA a time $t = 1$ and the Kth micro leaves the μISA at time $t = T$, where $T = K + \mu(m-1) + \sigma(n-1)$. Convenient shorthands used here are '$\bar{y}$' denoting '$y-1$', and 'y'' denoting '$y+1$', 'i,j' denoting '$i : 1 \leq i \leq m, j : 1 \leq j \leq n$', and $Q^{a..b}$ denoting $(Q^a \wedge Q^{a+1} \wedge \ldots \wedge Q^b)$.

The method is based on evaluating the (weakest) precondition at time t, Q_t, for $1 \leq t \leq T$, of the μISA due to the execution of the k_{11t}th to k_{mnt}th wavefronts

[2]cf. the the proof of the Red Squares program in Section 6.6.2 in which the time stamp $C_{ij}^{k.1}$ ($C_{ij}^{k.3}$) corresponds to the constant $A_{i,j}^k$ ($A_{i,j}'^k$) of this section.

of P, where:

$$k_{ijt} \;=\; t - \mu\bar{i} - \sigma\bar{j} \tag{6.1}$$

The method requires the ISA postcondition $Q = Q_{T'}$ and also the preconditions Q_t, for $1 \leq t \leq T$, to be in a *normal form*:

$$Q_t = (\forall_{i,j} s_{ij}^{k_{ijt}}) \tag{6.2}$$

where $s_{ij}^{k_{ijt}}$ is *normalized*, ie. it refers to the registers of no ISA cell except cell (i,j).

Thus, the precondition conjunct, $s_{ij}^{k_{ijt}}$ of cell (i,j) in Q_t is determined by the corresponding conjunct, $s_{ij}^{k_{ijt'}}$, in $Q_{t'}$ and the instruction, '$P1_{ijt}; P2_{ijt}$', executed in cell (i,j) at time t. In this process, $s_{ij}^{k_{ijt}}$ may need to *normalized* using the (unnormalized) precondition conjuncts corresponding to the neighbouring cells. If cell (i,j) is on the edge of the μISA, boundary conditions must also be considered.

The method consists of the three steps:

1. *normalize* the postcondition $Q_{T'}$, yielding $s_{ij}^{k_{ijT'}}$.
2. with the definitions, for $k = k_{ijt}$:

$$s_{ij}^{k} \stackrel{\text{def}}{=} s_{ij}^{k_{ijT'}} R_{ij}^{k..k_{ijT'}} \quad \text{if } k > K \tag{6.3}$$

$$s_{ij}^{k} \stackrel{\text{def}}{=} \text{wp}(P_{ij}^{k}, s_{ij}^{k'}) R_{ij}^{k} \quad \text{if } 1 \leq k \leq K \tag{6.4}$$

$$s_{ij}^{k} \stackrel{\text{def}}{=} s_{ij}^{k_{ij1}} R_{ij}^{k..0} \quad \text{if } k < 0 \tag{6.5}$$

$$R_{ij}^{k} \stackrel{\text{def}}{=} \begin{cases} (\mathrm{C}_{ij}^{k} = \mathrm{C}_{ij}) & \text{if } P1_{\bar{i}jt}, P1_{i\bar{j}t}, P1_{i'jt} \text{ or } P1_{ij't} \text{ contain } \mathrm{C}_{ij} \\ \textbf{true} & \text{otherwise} \end{cases} \tag{6.6}$$

$$P_{ij}^{k} \stackrel{\text{def}}{=} (P1_{ijt}; P2_{ijt})_{\mathrm{C}_{\bar{i}j}^{k+\mu}, \mathrm{C}_{i\bar{j}}^{k+\sigma}, \mathrm{C}_{i'j}^{k-\mu}, \mathrm{C}_{ij'}^{k-\sigma}}^{\mathrm{C}_{\bar{i}j}, \mathrm{C}_{i\bar{j}}, \mathrm{C}_{i'j}, \mathrm{C}_{ij'}} \tag{6.7}$$

   evaluate successively s_{ij}^{k}, for $1 \leq k \leq K$.
3. identify the (weakest) precondition as Q_1.

Here, C_{ij}^{k} for $1 \leq i \leq m, 1 \leq j \leq n$ and $1 \leq k \leq K$ are the *communication register time-stamps* (CRTS) introduced into the R_{ij}^{k} conjuncts[3]. The CRTS

[3]These are introduced only when necessary for *normalization*, ie. $R_{ij}^{k_{ijt}}$ is true if cell (i,j) is read at time t — see equation (6.6). Note that if equation (6.6) presents difficulties, theoretical or practical, R_{ij}^{k} can be simply redefined to be $(C_{ij}^{k} = \mathrm{C}_{ij})$.

generally denote 'fresh variables', ie. variables not appearing elsewhere in the current universe of discourse, particularly in the postcondition $Q_{T'}$. If C_{ij}^{k} does appear in the postcondition, an extra constraint is imposed on the C register of cell (i,j) at time t where $k_{ijt} = k$.

Section 6.5.2 establishes that the method is valid, ie. that $Q_1 \Rightarrow \mathrm{wp}(P', Q^{T'})$, and that Q_t, for each $1 \leq t \leq T$, is in *normal form.* In Section 6.5.3, an interpretation is given for the CRTS under which $Q_1 = \mathrm{wp}(P', Q^{T'})$.

The communication phase (timing) of the μISA instruction cycle is explicitly modelled by the substitutions of CRTS in $\mathrm{wp}(P1_{ijt}; P2_{ijt},\ s_{ij}^{k_{ijt'}})$ in equation (6.8). It is convenient to move these substitutions inside the definition of $P_{ij}^{k_{ijt}}$ in equation (6.7). The computation phase is modelled by applying the *weakest precondition* operator. Thus, the splitting of a μISA instruction into two explicit phases is no longer necessary, a significant conceptual and practical simplification. For the use of this method, the syntax for the μISA microlanguage can then be simplified by redefining:

$$\begin{aligned} &< \text{line of stat} > \ ::= \ < \text{stat2} > [, < \text{stat2} >]_{n-1} \\ &< \text{exp2} > \ ::= \ \text{an expression involving any register} > \\ &< \text{reg} > \ ::= \ < \text{accreg} > |\mathrm{C_N}|\mathrm{C_S}|\mathrm{C_E}|\mathrm{C_W} \end{aligned}$$

Equation (6.3) [(6.5)] states effectively that the precondition [postcondition] of P at cell (i,j) is given by s_{ij}^{1} [$s_{ij}^{K'}$], and that only what occurs between these states is significant. This corresponds to the intuitions of the μISA programmer, who visualizes P as a series of *wavefronts* moving through the μISA and considers the values of the registers of cell (i,j) as each wavefront passes through it. Thus, the padding of μISA programs on either side with large numbers of 'NoOp' wavefronts is made implicit by the proof method.

Note that in practice, P will be written so that cell (i,j) is read at time t only if $1 \leq k_{ijt} \leq K'$, so that $R_{ij}^{k_{ij1}..0} = \mathbf{true}$ [$R_{ij}^{K'..k_{ijT'}} = R_{ij}^{K'}$].

So far, the microprogrammed ISA semantics are adequate for constant boundary conditions. A simple way to generalize the semantics for arbitrary boundary conditions is to *time-stamp* the boundary (input) registers, so that at time t, the boundary registers are denoted by (the constants): $C_{0j}^{k_{0jt}}, C_{i0}^{k_{i0t}}, C_{m'j}^{k_{m'jt}}, C_{in'}^{k_{in't}}$

respectively, where $1 \leq i \leq m, 1 \leq j \leq n$. This notation is convenient, since in for example $P_{ij}^k = (\mathrm{C}_{ij} \leftarrow \mathrm{C}_{i\bar{j}}^{k+\sigma})$ (which corresponds to the μISA instruction 'C $\leftarrow$ $\mathrm{C_W}$'), $\mathrm{C}_{i\bar{j}}^{k+\sigma}$ denotes a boundary register for $j = 1$, and a CRTS otherwise. The derived preconditions generally impose some constraints on these constants, from which boundary conditions can be deduced.

For specifying output boundary conditions, the CRTS C_{i1}^k, C_{1j}^k, C_{in}^k and C_{jn}^k, for some appropriately chosen $1 \leq i \leq m, 1 \leq j \leq n$ and $1 \leq k \leq K$, may appear in the postcondition. This imposes some extra constraints on the corresponding communication registers at the corresponding times. As these CRTS are fixed externally to the proof method, so that they should be interpreted differently from other CRTS (see Section 6.5.3).

It may be asked why the transformation $k_{ijt} = t - \mu\bar{i} - \sigma\bar{j}$ was chosen here for the microprogrammed ISA. The reason is that the inverse transformation was used to form $(P1_{ijt}; P2_{ijt})$ of Section 6.3, and hence that P_{ij}^k is determined by '$stat1_{jk}; stat2_{jk}$' and sel_{ik} from the *wavefront*:

$$\left(\underset{j=1,n}{\text{,}} \; (stat1_{jk}; stat2_{jk}) \; \underset{i=1,m}{\text{,}} \; (sel_{ik}) \right)$$

in the program P. From the observation that *wavefronts* are always reasonably simple (this is the motivation of ISADL), the description of P_{ij}^k will also be reasonably simple. This leads to compact proofs of μISA programs, as is demonstrated in Section 6.6. Other transformations of the form $k_{ijt} = t - f(i,j)$ may be chosen but are unlikely to be of any intuitive or practical significance.

6.5.1 Example: the LCS program

The LCS algorithm is an inexact string matching algorithm with an efficient microprogrammed ISA implementation (see Section 4.2.1), with $K = 4, \mu = 3$ and $\sigma = 2$. The selectors of this program are all 1's, so that P_{ij}^k, for $1 \leq k \leq K$, are given simply by evaluating each of the four uniform instruction diagonals at cell (i,j) and replacing any external register references with their respective

CRTS (or *time-stamped* boundary registers):

k	P_{ij}^k
4	$\mathrm{C}_{ij} \leftarrow \max(\mathrm{C}_{ij}, \mathrm{A}_{ij})$
3	$\mathrm{C}_{ij} \leftarrow \mathrm{C}_{i\bar{j}}^5$
2	$\mathrm{A}_{ij} \leftarrow \max(\mathrm{A}_{ij}, \mathrm{C}_{\bar{i}j}^5)$
1	$\mathrm{A}_{ij} \leftarrow \mathrm{A}_{ij} * \mathrm{C}_{\bar{i}j}^3$

A normalized postcondition is chosen:

$$Q = \forall_{i,j}(M'_{ij} = \mathrm{C}_{ij})$$

The method will find the relationship between the constant matrix M' and the initial values of the μISA's registers and boundary conditions. This is achieved by deriving the intermediate conditions s_{ij}^4 to s_{ij}^1, noting R_{ij}^5 and R_{ij}^4 are non-trivial:

$$\begin{aligned}
s_{ij}^5 &= (M'_{ij} = \mathrm{C}_{ij})(\mathrm{C}_{ij}^5 = \mathrm{C}_{ij}) \\
&= (M'_{ij} = \mathrm{C}_{i\bar{j}}^5 = \mathrm{C}_{ij}) \\
s_{ij}^4 &= \mathrm{wp}(\mathrm{C}_{ij} \leftarrow \max(\mathrm{C}_{ij}, \mathrm{A}_{ij}),\ s_{ij}^5) \\
&= (M'_{ij} = \mathrm{C}_{i\bar{j}}^5 = \max(\mathrm{C}_{ij}, \mathrm{A}_{ij}))\ (\mathrm{C}_{ij}^4 = \mathrm{C}_{ij}) \\
s_{ij}^3 &= \mathrm{wp}(\mathrm{C}_{ij} \leftarrow \mathrm{C}_{i\bar{j}}^5,\ s_{ij}^4) \\
&= (M'_{ij} = \mathrm{C}_{i\bar{j}}^5 = \max(\mathrm{C}_{i\bar{j}}^5, \mathrm{A}_{ij})\ (\mathrm{C}_{ij}^4 = \mathrm{C}_{i\bar{j}}^5) \\
s_{ij}^2 &= \mathrm{wp}(\mathrm{A}_{ij} \leftarrow \max(\mathrm{A}_{ij}, \mathrm{C}_{\bar{i}j}^5),\ s_{ij}^3) \\
&= (M'_{ij} = \mathrm{C}_{i\bar{j}}^5 = \max(\mathrm{C}_{i\bar{j}}^5, \mathrm{A}_{ij}, \mathrm{C}_{\bar{i}j}^5)\ (\mathrm{C}_{ij}^4 = \mathrm{C}_{i\bar{j}}^5) \\
s_{ij}^1 &= \mathrm{wp}(\mathrm{A}_{ij} \leftarrow \mathrm{A}_{ij} * \mathrm{C}_{\bar{i}j}^4\ s_{ij}^2) \\
&= (M'_{ij} = \mathrm{C}_{i\bar{j}}^5 = \max(\mathrm{C}_{i\bar{j}}^5, \mathrm{A}_{ij} * \mathrm{C}_{\bar{i}j}^4, \mathrm{C}_{\bar{i}j}^5)\ (\mathrm{C}_{ij}^4 = \mathrm{C}_{i\bar{j}}^5) \\
&= (M'_{ij} = \mathrm{C}_{i\bar{j}}^5 = \max(\mathrm{C}_{i\bar{j}}^5, \mathrm{A}_{ij} * \begin{bmatrix} \mathrm{C}_{0j}^4 & \text{if } i = 1 \\ \mathrm{C}_{\bar{i}j}^5 & \text{if } i > 1 \end{bmatrix}, \mathrm{C}_{\bar{i}j}^5))\ (\mathrm{C}_{ij}^4 = \mathrm{C}_{i\bar{j}}^5)
\end{aligned}$$

In this last step, $\mathrm{C}_{\bar{i}j}^4$, for $1 \leq \bar{j} < n$ was eliminated using the equations $(\forall_{i,j}\mathrm{C}_{ij}^4 = \mathrm{C}_{i\bar{j}}^5)$. By matching here $\mathrm{C}_{i\bar{j}}^5$ with the result matrix M'_{ij}, the LCS program correctly implements the LCS algorithm given by equations (4.2) provided the following initial/boundary conditions hold:

$$\begin{aligned}
\mathrm{A}_{ij} &= M_{ij} \\
\mathrm{C}_{0j}^5 &= M'_{0j}
\end{aligned}$$

$$C^5_{i0} = M'_{i0}$$
$$C^4_{0j} = M'_{0\bar{j}}$$

6.5.2 Validity of the proof method

In this section, the validity of the proof method, using equation (6.8), is given. Equations (6.3-6.5) are derived, for the sake of convenience, from the equivalent expression for the precondition components:

$$s^{k_{ijt}}_{ij} = (\mathrm{wp}(P1_{ijt}; P2_{ijt},\ s^{k_{ijt'}}_{ij})^{C_{ij},\ C_{i\bar{j}},\ C_{i'j},\ C_{ij'}}_{C^{k+\mu}_{ij}, C^{k+\sigma}_{i\bar{j}}, C^{k-\mu}_{i'j}, C^{k-\sigma}_{ij'}})\ R^{k_{ijt}}_{ij} \tag{6.8}$$

where $1 \leq i \leq m, 1 \leq j \leq n, 1 \leq t \leq T$ and $k = k_{ijt}$. This assertion is justified as follows. For $k > K$ or $k < 1$, $P1_{ijt} = P2_{ijt} =$ SKIP, and hence the substitutions can have no effect on the *normalized* predicate $s^{k'}_{ij}$. For $1 \leq k \leq K$, $s^{k'}_{ij}$ is again *normalized*, so that the substitutions can be 'brought inside' $\mathrm{wp}(P1_{ijt}; P2_{ijt},\ s^{k'}_{ij})$ to form P^k_{ij}.

By identifying $s1^{k_{ijt}}_{ij}$ with $\mathrm{wp}(P1_{ijt}; P2_{ijt},\ s^{k_{ijt'}}_{ij})R^{k_{ijt}}_{ij}$, the following lemma is useful for *normalizing* the preconditions generated by the proof method:

Lemma 6.1 (Normalization Lemma) *given* $1 \leq t \leq T$*:*

$$(\forall_{i,j} s1^{k_{ijt}}_{ij}) = (\forall_{i,j} (s1^{k_{ijt}}_{ij})^{C_{ij},\ C_{i\bar{j}},\ C_{i'j},\ C_{ij'}}_{C^{k_{ijt}}_{ij}, C^{k_{i\bar{j}t}}_{i\bar{j}}, C^{k_{i'jt}}_{i'j}, C^{k_{ij't}}_{ij'}})$$

where:

$$\begin{aligned}
s1^{k_{\bar{i}jt}}_{\bar{i}j} &\Rightarrow (C_{\bar{i}j} = C^{k_{\bar{i}jt}}_{\bar{i}j}) && \text{, if } i > 1 \text{ and } (s1^k_{ij} \text{ contains } C_{\bar{i}j}) \\
s1^{k_{i\bar{j}t}}_{i\bar{j}} &\Rightarrow (C_{i\bar{j}} = C^{k_{i\bar{j}jt}}_{i\bar{j}}) && \text{, if } j > 1 \text{ and } (s1^k_{ij} \text{ contains } C_{i\bar{j}}) \\
s1^{k_{i'jt}}_{i'j} &\Rightarrow (C_{i'j} = C^{k_{i'jt}}_{i'j}) && \text{, if } i > m \text{ and } (s1^k_{ij} \text{ contains } C_{i'j}) \\
s1^{k_{ij't}}_{ij'} &\Rightarrow (C_{ij'} = C^{k_{ij't}}_{ij'}) && \text{, if } j < n \text{ and } (s1^k_{ij} \text{ contains } C_{ij'})
\end{aligned}$$

proof:

With $k = k_{ijt}$ and $k^- = k_{(i-1)jt}$, consider $i > 1$ and s^k_{ij} containing $C_{\bar{i}j}$, ie. communication from the north in cell (i,j) (note: s^k_{1j} could

only contain the boundary register C^k_{0j} value instead of $C_{\bar{i}j}$). Here, the relevant factors of $(\forall_{i,j} s^{k_{ijt}}_{ij})$ are:

$$\begin{aligned} s1^{k^-}_{\bar{i}j}\, s1^{k}_{ij} &= s1^{k^-}_{ij}(C_{\bar{i}j} = C^{k^-}_{\bar{i}j}) s1^{k}_{ijt} \\ &\quad \text{by the assumption } s1^{k^-}_{ij} \Rightarrow (C_{\bar{i}j} = C^{k^-}_{\bar{i}j}) \\ &= s1^{k^-}_{ij}(C_{\bar{i}j} = C^{k^-}_{\bar{i}j})(s1^{k}_{ijt})^{C_{\bar{i}j}}_{C^{k^-}_{\bar{i}j}} \quad \text{property of predicates} \\ &= s1^{k^-}_{ij}(s1^{k}_{ijt})^{C_{\bar{i}j}}_{C^{k^-}_{\bar{i}j}} \quad \text{again by the assumption} \end{aligned}$$

If s^k_{ij} does not contain $C_{\bar{i}j}$, the above equation still holds. By repeating this procedure in all directions, all four substitutions can be made to s^k_{ij}. Note the condition $s^k_{ij} \Rightarrow (C_{ij} = C^k_{ij})$ is preserved by these substitutions. By iterating over all (i,j) (in arbitrary order), the result is established. □

For the microprogrammed ISA, with $k = k_{ijt}$, then $k+\mu = k_{\bar{i}jt}, \ldots, k-\sigma = k_{ij't}$, so that the CRTS substituted here are

$$C^{k+\mu}_{\bar{i}j}, C^{k+\sigma}_{i\bar{j}}, C^{k-\mu}_{i'j}, C^{k-\sigma}_{ij'}$$

The Normalization Lemma is illustrated for the LCS program (cf. Section 6.5.1, s^3_{ij} with part (a), and s^2_{ij} with part (b)) in Figure 6.2.

For normalized postconditions, the microprogrammed ISA semantics can now be *parallelized* in the following sense:

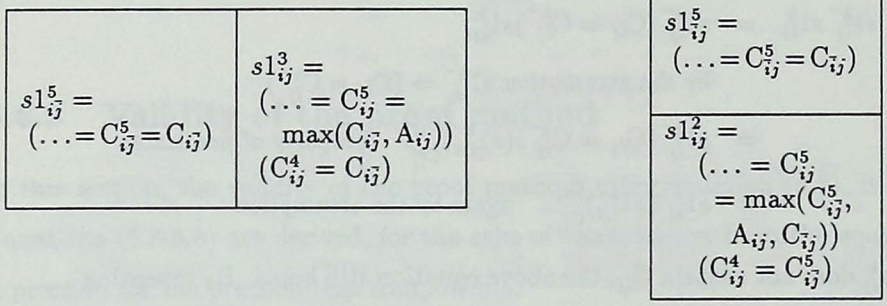
Result 6.1 (Parallelization result) *given t, $1 \le t \le T$, and provided for all $1 \le i \le m, 1 \le j \le n$ that $s^{k_{ijt}}_{ij}$ is defined as in equation (6.8) and $s^{k_{ijt'}}_{ij}$ is* normalized, *then $s^{k_{ijt}}_{ij}$ is also* normalized *and:*

$$(\forall_{i,j} s^{k_{ijt}}_{ij}) \;\Rightarrow\; \mathrm{wp}((;_{i,j} P1_{ijt}; P2_{ijt}),\ (\forall_{i,j} s^{k_{ijt'}}_{ij}))$$

proof:

Now $s^{k_{ijt}}_{ij}$ is normalized, since $s^{k_{ijt'}}_{ij}$ and R^k_{ij} are normalized, and the only register references outside cell (i,j) in $(P1_{ijt}; P2_{ijt})$ are eliminated by the subsequent substitutions. Also:

$$\mathrm{wp}((;_{i,j} P1_{ijt}; P2_{ijt}),\ (\forall_{i,j} s^{k_{ijt'}}_{ij}))$$



(a) $k_{ijt} = 3, s1^3_{ij}$ containing $C_{i\bar{j}}$ from $(P1_{ijt}; P2_{ijt}) = (C_{ij} \leftarrow C_{i\bar{j}})$

(b) $k_{ijt} = 2, s1^2_{ij}$ containing $C_{\bar{i}j}$ from $(P1_{ijt}; P2_{ijt}) = (A_{ij} \leftarrow \max(A_{ij}, C_{\bar{i}j}))$

Figure 6.2: Normalization Lemma: visualization of $(\forall_{i,j} s1^{k_{ijt}}_{ij})$ around cell (i,j) for the LCS program

$$
\begin{aligned}
&= (\forall_{i,j} \mathrm{wp}((;_{i,j} P1_{ijt}; P2_{ijt}),\ s^{k_{ijt'}}_{ij})) && \text{property of wp}\\
&= (\forall_{i,j} \mathrm{wp}((P1_{ijt}; P2_{ijt}),\ s^{k_{ijt'}}_{ij})) && \text{since } s^{k_{ijt'}}_{ij} \text{ is normalized}\\
&\Leftarrow (\forall_{i,j} \mathrm{wp}((P1_{ijt}; P2_{ijt}),\ s^{k_{ijt'}}_{ij}) R^{k_{ijt}}_{ij}) && \text{property of predicates}\\
&= (\forall_{i,j} (\mathrm{wp}((P1_{ijt}; P2_{ijt}),\ s^{k_{ijt'}}_{ij}) R^{k_{ijt}}_{ij})^{C_{\bar{i}j},\ C_{i\bar{j}},\ C_{i'j},\ C_{ij'}}_{C^{k_{\bar{i}jt}}_{\bar{i}j}, C^{k_{i\bar{j}t}}_{i\bar{j}}, C^{k_{i'jt}}_{i'j}, C^{k_{ij't}}_{ij'}})\\
&\quad \text{by the Normalization Lemma, applicable due to defn. of } R^k_{ij}\\
&= (\forall_{i,j} (\mathrm{wp}(P1_{ijt} P2_{ijt}),\ s^{k_{ijt'}}_{ij})^{C_{\bar{i}j},\ C_{i\bar{j}},\ C_{i'j},\ C_{ij'}}_{C^{k_{\bar{i}jt}}_{\bar{i}j}, C^{k_{i\bar{j}t}}_{i\bar{j}}, C^{k_{i'jt}}_{i'j}, C^{k_{ij't}}_{ij'}} R^{k_{ijt}}_{ij})\\
&\quad \text{since } R^{k_{ijt}}_{ij} \text{ is normalized}\\
&= (\forall_{i,j} s^{k_{ijt}}_{ij}) && \text{by equation (6.8)}
\end{aligned}
$$

□

Inductively applying this result over time, together with the definition of P', establishes the validity of the proof method with respect to the original μISA semantics:

Result 6.2 (Validity result)

$$(\forall_{i,j} s^{k_{ij1}}_{ij}) \Rightarrow \mathrm{wp}(P',\ (\forall_{i,j} s^{k_{ijT'}}_{ij}))$$

6.5.3 Interpretation of the communication register time-stamps

The validity of the CRTS technique of the proof method has just been demonstrated. Intuitively, the CRTS introduced by the method, ie. those not already appearing in the original postcondition, denote some (not yet known) values residing in the μISA's communication registers at particular time instants, and the proof method eventually determines these values. At this stage, they may or may not add any real information to the precondition; in the latter case, it is desirable to eliminate them from the precondition (eg. it is desirable to eliminate altogether C^4_{ij}, for $1 \leq i \leq m$ and $1 \leq j \leq n$, in $(\forall_{i,j} s^1_{ij})$ for the LCS program of Section 6.5.1). A formal interpretation of the CRTS is given here, which forms justification for their introduction and subsequent elimination. Under this interpretation, the proof method does in fact yield the *weakest* precondition.

The interpretation runs as follows. Consider some μISA cell (i_0, j_0) at time step t, with $1 \leq t \leq T$ and $k_0 = k_{i_0 j_0 t}$. At time t, $C_{i_0 j_0}$ is assumed to have a definite (but possibly unknown) value[4], so that the conjunct $(\exists C^k_{i_0 j_0} . R^{k_0}_{i_0 j_0 t})$, where $R^{k_0}_{i_0 j_0}$ (either **true** or $(C_{i_0 j_0} = C^{k_0}_{i_0 j_0})$), evaluates to **true**. Assume that $C^{k_0}_{i_0 j_0}$ does not appear in the postcondition; hence, it cannot appear in any $s^k_{ijt'}$, so that with $s1^{k_{ijt}}_{ij} \stackrel{\text{def}}{=} \text{wp}((P1_{ijt}; P2_{ijt}), s^{k_{ijt'}}_{ij})$ (cf. equation (6.8)):

$$\begin{aligned} (\forall_{i,j} s1^{k_{ijt}}_{ij}) &= (\forall_{i,j} s1^{k_{ijt}}_{ij})(\exists C^k_{i_0 j_0} . R^{k_0}_{i_0 j_0 t}) \\ &= (\exists C^k_{i_0 j_0} . (\forall_{i,j} s1^{k_{ijt}}_{ij} R^{k_0}_{i_0 j_0 t})) \end{aligned}$$

Under the interpretation of C^k_{ij} being under implicit universal quantification (when not appearing in the postcondition), R^k_{ij} always evaluates to **true**, so that, by inspecting the proof of Result 6.1:

$$(\forall_{i,j} s^{k_{ijt}}_{ij}) = \text{wp}((;_{i,j} P1_{ijt}; P2_{ijt}),\ (\forall_{i,j} s^{k_{ijt'}}_{ij}))$$

Hence, under this interpretation of the CRTS, the proof method yields the weakest precondition.

[4] cf. VLSI, where a memory cell can similarly be assumed to have a definite value during the 'read phase' of any instruction cycle.

This interpretation can also be used to eliminate CRTS no longer contributing any real information to the precondition. Consider that at some time $1 \leq i \leq t$, a conjunct of the form $(C^{k_0}_{i_0 j_0} = E)$ appears in $(\forall_{i,j} s^{k_{ij} i}_{ij})$ (which is implicitly quantified over $C^{k_0}_{i_0 j_0}$), where E contains no μISA registers. If $C^{k_0}_{i_0 j_0}$ appears in no other conjuncts, the scope of its quantification can be brought inside to give $(\exists C^{k}_{i_0 j_0}.C^{k}_{i_0 j_0} = E)$, which similarly evaluates to **true** and can be dropped. Note that if E did contain any μISA registers, dropping this conjunct may result in loss of the normalization property, and the proof method could not proceed further.

Note that any constants appearing in the postcondition $Q^{T'}$ and the boundary registers $C^{k}_{i0}, C^{k}_{0j}, C^{k}_{in'}, C^{k}_{m'j}$ should not be interpreted as being implicitly existentially quantified, since these are fixed externally to the weakest precondition derivation. In any case, a precondition derivation seeks to express the postcondition constants in terms of the boundary registers and the values of the μISA registers during the precondition: to existentially quantify any of these would render the precondition meaningless.

6.5.4 Incorporating ISADL into the method: a semantics for ISADL

ISADL is a language based on the diagonal or wavefront as a semantic unit for a μISA program. For describing ISADL implementations, the semantics of a ISADL diagonal is given in terms of instruction sequences; for the purpose of this method, its semantics is given in the form of a 'guarded expression', ie. a series of instructions, each guarded by (mutually exclusive) predicates which are functions of the cell position, (i, j). The guarded expression form of the instruction part of a ISADL diagonal D on an $m \times n$ μISA is given by:

$$\text{d\_ge}(D, \textbf{true}, j, 0, n)$$

where, with $x \bmod_1 y = (x-1) \bmod y + 1$,:

$$\text{d\_ge}(\boxed{D_1 \quad n_1 \mid D_2 \quad n_2 \mid \ldots \mid D_l \quad n_l}, G, \mathcal{J}, l, h)$$

$$= \text{d\_ge}(D_1, G \wedge (l < \mathcal{J} \leq h),\ (\mathcal{J} - l) \bmod_1 n_l, 0, n_1)$$

$$\text{d\_ge}(D_2, G \wedge (l < \mathcal{J} \leq h),\ (\mathcal{J} - l) \bmod_1 n_l, n_1, n_2) \ldots$$

$$\text{d\_ge}(D_l, G \wedge (l < \mathcal{J} \leq h),\ (\mathcal{J} - l)\text{mod}_1 n_l, n_{l-1}, n_l)$$

$$\text{d\_ge}(\text{ins}, G, \mathcal{J}, l, h) \quad = \quad \text{“}PE_{ij}.\text{ins}\ ,\ _{G \wedge (l < \mathcal{J} \leq h)}\text{”} \tag{6.9}$$

where 'PE_{ij}.ins' is the instruction 'ins' with any register references evaluated for cell (i, j), as is described in Section 6.2. The corresponding selector diagonal D' is similarly converted into "guarded expression" form by:

$$\text{d\_ge}(D', \textbf{true}, i, 0, M)$$

and the diagonal's operation (effective instruction) on cell (i, j) is given by the 'Cartesian product' of these two guarded expressions. This, after evaluating register references at cell (i, j) and making the appropriate substitutions of equation 6.7), gives P_{ij}^k. Equation (6.9) may be understood by studying the examples of Section 6.6. Note that the conditions '$0 < i \leq m, 0 < j \leq n$' are implicitly assumed and can be dropped.

6.6 Using the proof method

One of the primary objectives of the proof method is to give compact proofs of μISA programs with a minimum of bookeeping. The essential part of the proof method involves evaluating and rearranging the intermediate component preconditions s_{ij}^k, for $1 \leq k \leq K$, using equation (6.4). From applying the proof method to the LCS program in Section 6.5.1, two observations can be made to reduce bookeeping:

1. introducing the C_{ij}^5 in the postcondition to give the component $s_{ij}^5 = (M'_{ij} = \text{C}_{ij}^5 = \text{C}_{ij})$ is unnecessary to maintain *normalization* in the precondition components $s_{\bar{i}j}^3$ and $s_{i\bar{j}}^2$ (which 'read' C_{ij} from C_{ij}^5). This is because M'_{ij}, an expression containing no μISA registers, could be used instead of C_{ij}^5 to normalize the other components.

   In general, if s_{ij}^k is 'read' by a neighbouring component[5] $s_{\bar{i}\bar{j}}^{\tilde{k}}$, and $(s_{ij}^k \Rightarrow (\text{C}_{ij} = C_{ij}^k))$, where C_{ij}^k is an expression containing no μISA registers

[5] In this case, $k - \tilde{k} = k_{ijt} - k_{\bar{i}\bar{j}t}$.

(eg. M'_{ij}), then R^k_{ijt} can be defined as **true** provided $\mathcal{C}^k_{ij}$ is used instead of C^k_{ij} (eg. to form $P^{\bar{k}}_{ij}$).

Also, when the postcondition component $s^{K'}_{ij}$ is 'read', postcondition components of the form '$(\mathrm{C}^{K'}_{ij} = \mathrm{C}_{ij})\ldots$' may be used. Here, $\mathrm{C}^{K'}_{ij}$ is not to be interpreted as under implicit universal quantification (see Section 6.5.3).

2. when $R^k_{ij} = (\mathrm{C}^k_{ij} = \mathrm{C}_{ij})$, it is convenient to redefine s^k_{ij} to the equivalent form:

$$s^k_{ij} \stackrel{\text{def}}{=} (\mathrm{wp}(P^k_{ij}, s^{k'}_{ij}))^{\mathrm{C}_{ij}}_{\mathrm{C}^k_{ij}} R^k_{ij}$$

This helps to reduce the conjuncts in s^k_{ij} containing μISA register references (the *live* conjuncts) from the others (the *static* conjuncts, eg. conjunct $(\mathrm{C}^4_{ij} = \mathrm{C}^5_{ij})$ in s^2_{ij} of Section 6.5.1). The *static* conjuncts are not affected by the weakest precondition operations.

This distinction is useful since static conjuncts can be separated, manipulated[6], and used to simplify the live conjuncts at any later stage. Once in the appropriate form (see Section 6.5.3), they can then be dropped from the precondition Q_1.

For presentation of the proofs required by this method, a tabular presentation with P^k_{ij} and s^k_{ij} (live and static conjuncts) is concise and convenient. With the above observations, the static conjuncts of $s^{k'}_{ij}$, need not be explicitly 'carried' to define s^k_{ij}: they need be written only once. Enclosing static conjuncts in brackets ([...]) signifies that they must be included in the precondition.

In the remainder of this section, the proof method is applied to four simple microprogrammed ISA programs, and one more complicated one. The first program has a constant period, whereas the others are iterative and require the induction of *invariants*, the simplicity of which reflects the uniformity of their ISADL encodings. These examples illustrate the derivation of boundary conditions and communication effects in various directions. Of these examples, the first two are the simplest and capture the main ideas of the proof method.

[6]This, together with inducing the *invariants*, are the parts of the proof method most difficult to mechanize.

6.6.1 The LCS program revisited

The proof of the LCS program (Section 6.5.1) is now given using the tabular presentation described above. The ISADL encoding for the instructions of this program is given by (corresponding state numbers, k, annotated to the left):

4	$C \leftarrow \max(C, A)$	n
3	$C \leftarrow C_W$	n
2	$A \leftarrow \max(A, C_N)$	n
1	$A \leftarrow A * C_N$	n

The selectors are all 1's. With M'_{ij}, for $0 \leq i \leq m$ and $0 \leq j \leq n$, being used in place of C^5_{ij} (the CRTS and the north/west boundary input registers corresponding to state $k = 5$), a normalized postcondition is chosen:

$$Q = (\forall_{i,j}(M'_{ij} = C_{ij}))$$

The method derives the intermediate conditions s^4_{ij} to s^1_{ij}, using P^k_{ij} which is derived from the corresponding ISADL diagonal according to equations (6.7,6.9):

k	P^k_{ij}	s^k_{ij}	
		live	static
5		$M'_{ij} = C_{ij}$	
4	$C_{ij} \leftarrow \max(C_{ij}, A_{ij})$	$M'_{ij} = \max(C^4_{ij}, A_{ij})$ $C^4_{ij} = C_{ij}$	
3	$C_{ij} \leftarrow M'_{ij}$	$M'_{ij} = \max(C^4_{ij}, A_{ij})$	$C^4_{ij} = M'_{ij}$
2	$A_{ij} \leftarrow \max(A_{ij}, M'_{ij})$	$M'_{ij} = \max(C^4_{ij}, A_{ij}, M'_{ij})$	
1	$A_{ij} \leftarrow A * C^4_{ij}$	$M'_{ij} = \max(C^4_{ij}, A_{ij} * C^4_{ij}, M'_{ij})$	
1		$M'_{ij} = \max(M'_{ij}, A_{ij} * \begin{cases} C^4_{0j}, & i=1 \\ M'_{ij}, & 1<i\leq n \end{cases}, M'_{ij})$	

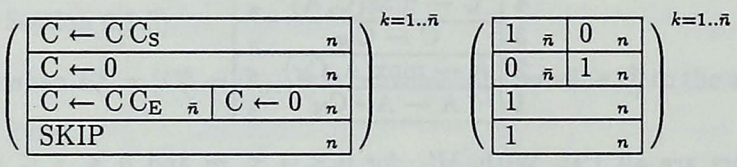
At the last stage, the static conjunct of s^4_{ij} is used to eliminate C^4_{ij} (for $i > 1$) and C^4_{ij} from s^1_{ij}. This static conjunct can then be dropped from the final precondition.

6.6.2 The Red Squares program: simple, uniform iterations

The Red Squares program[7] calculates at the kth step, where $1 \leq k < n$, the matrix $C^{k.1}$, where $C^{k.1}_{ij}$ is set iff there is a square of 1's of size k in the $n \times$

[7] cf. program RedSquares of Section 2.2.2.1; the program here, for the sake of simplicity, has the ComputeS_k sub-program removed, and hence need not use the A register.

n boolean matrix $\mathrm{C}^{k.1}$ whose left-upper corner is in position (i,j). Its (μ)ISA implementation uses $K = 4\bar{n}, \mu = 1$ and $\sigma = 1$. Applying the proof method on this program illustrates how the CRTS, together with the concept of static conjuncts, can simplify the *invariant* and hence the proof. The ISADL encoding for the instructions of this program is given by:



Introducing a convenient notation '$k.a = 4k + a$', a normalized postcondition is chosen, as before:

$$Q = (\forall_{i,j} s_{ij}^{n.1}) \ , \text{ where } s_{ij}^{n.1} = (\mathrm{C}_{ij}^{n.1} = \mathrm{C}_{ij})$$

which, reflecting the uniform iterations in the ISADL encoding, is generalized in a uniform way to the give the (live part of) the invariant predicate, for $1 \leq i, j, k \leq n$:

$$s_{ij}^{k.1} \stackrel{\text{def}}{=} (\mathrm{C}_{ij}^{k.1} = \mathrm{C}_{ij})$$

The method is now used to establish that this invariant is maintained, ie. the kth diagonal, operating on $s_{ij}^{k'.1}$, yields $s_{ij}^{k.1}$, for each $1 \leq k \leq \bar{n}$. Relationships between the CRTS $\mathrm{C}^{k'.1}$ and $\mathrm{C}^{k.1}$ are derived in this process.

l	P_{ij}^l	s_{ij}^l	
		live	static
$k'.1$	...	$\mathrm{C}_{ij}^{k'.1} = \mathrm{C}_{ij}$	
$k.4$	$\mathrm{C}_{ij} \leftarrow \mathrm{C}_{ij}\mathrm{C}_{i'j}^{k.3}$, $0<i\leq\bar{n}$	$\mathrm{C}_{ij}^{k'.1} = \begin{cases} \mathrm{C}_{ij}\mathrm{C}_{i'j}^{k.3} & , 0<i\leq\bar{n} \\ \mathrm{C}_{ij} & , i=n \end{cases}$	
$k.3$	$\mathrm{C}_{ij} \leftarrow 0$, $i=n$	$\mathrm{C}_{ij}^{k.3} = \mathrm{C}_{ij}$	$\mathrm{C}_{ij}^{k'.1} = \begin{cases} \mathrm{C}_{ij}^{k.3}\mathrm{C}_{i'j}^{k.3} & , 0<i\leq\bar{n} \\ 0 & , i=n \end{cases}$
$k.2$	$\mathrm{C}_{ij} \leftarrow \mathrm{C}_{ij}\mathrm{C}_{ij'}^{k.1}$, $0<j\leq\bar{n}$ $\mathrm{C}_{ij} \leftarrow 0$, $j=n$	$\mathrm{C}_{ij}^{k.3} = \begin{cases} \mathrm{C}_{ij}\mathrm{C}_{ij'}^{k.1} & , 0<j\leq\bar{n} \\ 0 & , j=n \end{cases}$	
$k.1$		$\mathrm{C}_{ij}^{k.1} = \mathrm{C}_{ij}$	$\mathrm{C}_{ij}^{k.3} = \begin{cases} \mathrm{C}_{ij}^{k.1}\mathrm{C}_{ij'}^{k.1} \\ !, \ 0<j\leq\bar{n} \\ 0 & , j=n \end{cases}$

In the table, a notation for conditionals serves as a convenient shorthand; in the P_{ij}^l column, the default value is 'SKIP'. Note that at step $k.3$ (step $k.1$), the

static conjunct is produced by substituting C_{ij} with $C_{ij}^{k,3}$ (with $C_{ij}^{k,1}$) at the earliest possible opportunity. From the static conjuncts, the relationship of the values of the C registers between successive iterations is derived implicitly at each step. The precondition corresponds to the invariant with $k = 1$, stating that at each μISA cell contains the corresponding element of an initial matrix $C^{1.1}$, and the set of (weakest) recurrence equations of the static conjuncts gives the relationship between the final matrix $C^{n.1}$ and the initial matrix $C^{1.1}$.

6.6.3 The matrix input program: non-uniform iterations

The LoadMat program (see Section 3.7.1) loading a matrix from the western data buffer to the C registers of an ISA, uses $K = n, \mu = 1$ and $\sigma = 1$. Applying the proof method on this program illustrates the use of the CRTS for deriving input boundary conditions. The ISADL encoding for the instructions of this program is given by:

$C \leftarrow C_W \quad _{n-\bar{k}}$	SKIP $_n$	$^{k=1..n}$

The selectors are all 1's.

A normalized postcondition is chosen, as before:

$$Q = (\forall_{i,j} s_{ij}^{n'}) \ , \text{ where } s_{ij}^{n'} = (C_{ij}^{n'} = C_{ij})$$

which, reflecting the ISADL encoding, needs to be generalized to the invariant (component) condition, for $1 \leq i, j \leq n$ and $1 \leq k \leq n'$:

$$s_{ij}^{k} \stackrel{\text{def}}{=} (C_{ij} = \left\{ \begin{array}{ll} C_{ij}^{k} & , \ 0<j\leq n-\bar{k} \\ C_{ij}^{n'} & , \ n-\bar{k}<j\leq n \end{array} \right\})(\ C_{i0}^{n'-\bar{j}} = C_{ij}^{n'} \quad , \ 0<j\leq n-\bar{k} \ )$$

This invariant can be induced by applying the method for a few cases (eg. $k = n-1$, $k = n-2$) on Q. Note that during iteration k, only the registers in cells $(i,1)$ to $(i, n-k)$ are actually read (hence only the CRTS $C_{i1}^{k'}, \ldots, C_{i(n-k)}^{k'}$ are introduced).

The method is now used to establish that this invariant is maintained, ie. the kth diagonal, operating on $s_{ij}^{k'}$, yields s_{ij}^{k}, for each $1 \leq k \leq n$, to establish when and where the matrix $C^{n'}$ enters the ISA.

l	P^l_{ij}	s^l_{ij}	
		live	static
k'	$\ldots$	$C_{ij} = \begin{cases} C^{k'}_{ij} & , 0<j\leq n-k \\ C^{n'}_{ij} & , n-k<j\leq n \end{cases}$	$\ldots$
k	$C_{ij} \leftarrow C^{k'}_{i\bar{j}}$ $, 0<j\leq n-\bar{k}$	$\left(C^{k'}_{i\bar{j}} = \begin{cases} C^{k'}_{ij} & , 0\leq j\leq n-k \\ C^{n'}_{ij} & , j=n-\bar{k} \end{cases}\right)$ $(C_{ij} = C^{n'}_{ij} \quad , n-\bar{k}<j\leq n)$ $(C_{ij} = C^{k}_{ij} \quad , 0<j\leq n-\bar{k})$	
k		$C_{ij} = \begin{cases} C^{k}_{ij} & , 0<j\leq n-\bar{k} \\ C^{n'}_{ij} & , n-\bar{k}<j\leq n \end{cases}$	$C^{k'}_{i0} = C^{k'}_{ij}$ $, 0\leq j\leq n-\bar{k}$ $\left[C^{n'-\bar{j}}_{i0} = C^{n'}_{ij} \quad , j=n-\bar{k}\right]$

In the last step, a recurrence equation has been solved, expressing $C^{k'}_{i0}$ in terms of the postcondition matrix $C^{n'}$. The precondition corresponds to $k = 1$, and from here, the boundary conditions can be derived from the retained static conjuncts:

$$C^{n'-\bar{j}}_{i0} = C^{n'}_{ij} \quad , 0 < j \leq n$$

6.6.4 The matrix output program: non-uniform iterations

The UnLoadMat program (see Figure 6.3), unloading the $n \times n$ matrix A from the C register of the ISA into the eastern data buffer, uses $K = \bar{n}, \mu = 1$ and $\sigma = 1$. Applying the proof method on this program illustrates the use of the CRTS for deriving output boundary conditions. The ISADL encoding for the instructions of this program is given by:

$SKIP \quad n-k$	$C \leftarrow C_W \quad n$	$k=1..\bar{n}$

The selectors are all 1's.

A postcondition is chosen with the CRTS on the east ISA boundary being set to some (unspecified) matrix A:

$$Q = (\forall_i Q_i^{1..n}) \text{ , where for } 1 \leq k \leq n\text{: } Q^k_i = (C^k_{in} = A_{i,n-\bar{k}})$$

$Q^{k'}_i$ is interpreted as follows: after the execution of the kth diagonal of program UnLoadMat, the value of $A_{i,n-k}$ is read by the east data buffer from the C register of cell (i, n). This postcondition is unusual in the sense that all of its conjuncts are static, since they express output boundary conditions. In a similar fashion to

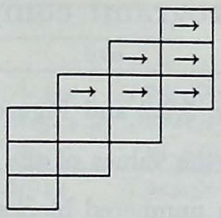


Figure 6.3: UnLoadMat program, instruction part, on a 4×4 ISA

('$\rightarrow$' denotes '$C \leftarrow C_W$')

the LoadMat program, live conjuncts are introduced into the invariant predicate, for $1 \leq i, j, k \leq n$:

$$s_{ij}^{k} \stackrel{\text{def}}{=} \left(C_{ij} = \begin{cases} A_{i,j} & 0<j\leq n-k \\ C_{ij}^{k} & n-k<j\leq n \end{cases} \right) Q_i^{1..k}$$

Note that during iteration k, only the registers in cells $(i, n-k)$ to (i, n) are actually read by other ISA cells or the east data buffer (hence only the CRTS $C_{i(n-k)}^{k'}, \ldots, C_{in}^{k'}$ are introduced).

The method is now used to establish that this invariant is maintained, ie. the kth diagonal, operating on $s_{ij}^{k'}$, yields s_{ij}^{k}, for each $1 \leq k \leq n$, to establish where the matrix A resides before the program is executed:

l	P_{ij}^{l}	s_{ij}^{l}	
		live	static
k'	$\ldots$	$C_{ij} = \begin{cases} A_{i,j} & 0<j\leq n-k' \\ C_{ij}^{k'} & n-k'<j\leq n \end{cases}$	$[Q_i^{1..k}]$ $(C_{in}^{k'} = A_{i,n-k})$
k	$C_{ij} \leftarrow C_{ij}^{k'}$, $n-k<j\leq n$	$C_{ij} = \begin{cases} A_{i,j} & 0<j\leq n-k' \\ C_{ij}^{k'} & j=n-k \end{cases}$ $(C_{ij}^{k'} = C_{ij}^{k'}$, $n-k<j\leq n)$ $(C_{ij} = C_{ij}^{k}$, $n-k<j\leq n)$	$(C_{in}^{k'} = A_{i,n-k})$
k		$C_{ij} = \begin{cases} A_{i,j} & 0<j\leq n-k \\ C_{ij}^{k} & n-k<j\leq n \end{cases}$	$C_{ij}^{k'} = A_{i,n-k}$, $n-k\leq j\leq n$

In the last step, a recurrence equation has been solved, expressing $C_{ij}^{k'}$ in terms of the postcondition matrix A. The precondition corresponds to s_{ij}^{1}:

$$C_{ij} = A_{i,j}$$

6.6.5 The row swap program: complex iterations

The RotH'_d program (see Section 5.2.2 and Figure 5.5), parameterized by $d = 2^l (1 \leq l < logn)$, interchanges the values of the C registers in corresponding positions between odd and even numbered blocks of width d. It presents the greatest challenge so far to the proof method, with non-uniform iterations based on exchange steps. Exchanging values between adjacent ISA cells is surprisingly easy to express in terms of the wavefront concept, as this example illustrates.

The program uses $K = 2d, \mu = 1$ and $\sigma = 1$. The ISADL encoding for the instructions of this program is given by:

$$\left(\begin{array}{|l|l|l|} \hline \mathrm{SKIP}\ _{\bar{k}} & \mathrm{C} \leftarrow \mathrm{C_E}\ _{d+\bar{k}} & \mathrm{SKIP}\ _{2d} \\ \hline \mathrm{SKIP}\ _{k} & \mathrm{C} \leftarrow \mathrm{C_W}\ _{d+k} & \mathrm{SKIP}\ _{2d} \\ \hline \end{array} \right)^{k=1..d}$$

The program's selectors are all 1's.

Similarly, with now '$k.a = 2k + a$', a normalized postcondition is chosen, as before:

$$Q = (\forall_{i,j} s_{ij}^{d'.1}) \ , \text{where } s_{ij}^{d'.1} = (A_{ij} = \mathrm{C}_{ij})$$

The approach taken here is to express any introduced CRTS in terms of A, and eliminate them from the invariant. By again applying the method on Q for a few iterations, the invariant (component) condition is induced, for $1 \leq i \leq m, 1 \leq j \leq n$ and $1 \leq k \leq d'$:

$$s_{ij}^{k.1} \stackrel{\text{def}}{=} (\mathrm{C}_{ij} = \left\{ \begin{array}{ll} A_{i,j+d} & , 0 < j_0 \leq d - \bar{k} \\ A_{i,j+\bar{k}-d} & , \text{d-}\bar{k} < j_0 \leq 2d - \bar{k} \\ A_{i,j} & , 2d - \bar{k} < j_0 \leq 2d \end{array} \right\})$$

where $j_0 = (j - 1) \bmod 2d + 1$. The method is now used to establish that this invariant is maintained. The CRTS $\mathrm{C}_{ij}^{k.2}$ for $d - k \leq j_0 \leq 2d - k$, and $\mathrm{C}_{ij}^{k.1}$ for $d - \bar{k} \leq j_0 \leq 2d - \bar{k}$, are introduced by the method and can be later eliminated.

l	P^l_{ij}	s^l_{ij}	
		live	static
$k'.1$	$\ldots$	$C_{ij} = \begin{cases} A_{i,j+d} & , 0<j_0\leq d-k \\ A_{i,j+k-d} & , d-k<j_0\leq 2d-k \\ A_{i,j} & , 2d-k<j_0\leq 2d \end{cases}$	
$k.2$	$C_{ij} \leftarrow C^{k.1}_{ij'}$, $d-k<j_0\leq 2d-k$	$(C_{ij} = A_{i,j+d} \quad , 0<j_0\leq d-k)$ $(C^{k.1}_{ij'} = A_{i,j+k-d} \quad , d-k<j_0\leq 2d-k)$ $(C_{ij} = A_{i,j} \quad , 2d-k<j_0\leq 2d)$ $(C_{ij} = C^{k.2}_{ij} \quad , d-k<j_0\leq 2d-k)$	
$k.2$		$C_{ij} = \begin{cases} A_{i,j+d} & , 0<j_0\leq d-k \\ C^{k.2}_{ij} & , d-k<j_0\leq 2d-k \\ A_{i,j} & , 2d-k<j_0\leq 2d \end{cases}$	$C^{k.1}_{ij} = A_{i,j+\bar{k}-d}$, $d-\bar{k}<j_0\leq 2d-\bar{k}$
$k.1$	$C_{ij} \leftarrow C^{k.2}_{ij}$, $d-\bar{k}<j_0\leq 2d-\bar{k}$	$(C_{ij} = A_{i,j+d} \quad , 0<j_0\leq d-k)$ $(C_{ij} = C^{k.2}_{ij} \quad , j_0=d-\bar{k})$ $\left(C^{k.2}_{ij} = \begin{cases} C^{k.2}_{ij} & , d-\bar{k}<j_0\leq 2d-k \\ A_{i,j} & , j_0=2d-\bar{k} \end{cases}\right)$ $(C_{ij} = A_{i,j} \quad , 2d-\bar{k}<j_0\leq 2d)$ $(C_{ij} = C^{k.1}_{ij} \quad , d-\bar{k}<j_0\leq 2d-\bar{k})$	
$k.1$		$C_{ij} = \begin{cases} A_{i,j+d} & , 0<j_0\leq d-\bar{k} \\ A_{i,j+\bar{k}-d} & , d-\bar{k}<j_0\leq 2d-\bar{k} \\ A_{i,j} & , 2d-\bar{k}<j_0\leq 2d \end{cases}$	$C^{k.2}_{ij} = A_{i,2d-\bar{k}+(j-j_0)}$, $d-k<j_0\leq 2d-\bar{k}$

In this process, the static conjuncts have been manipulated (index renumbering and recurrence equation solving), and in the last step, they have been used to eliminate $C^{k.2}_{ij}$ from the live conjuncts. The invariant gives the precondition $s^{k.1}_{ij}$:

$$C_{ij} = \begin{cases} A_{i,j+d} & , 0<j_0\leq d \\ A_{i,j-d} & , d<j_0\leq 2d \end{cases}$$

verifying that the swap has indeed occurred. While this proof is rather complicated, it must be remembered that this program is indeed rather difficult to understand.

6.7 Discussion and conclusions

This chapter has given a formal description of the syntax and semantics for microprogrammed ISAs, which, for $\mu = \sigma = 1$, gives also the semantics of ordinary ISAs (see also [33]). This is extended from the semantics of [34] in that multiple assignments have been included. This semantics is based on a 'serialization principle' and it enables formal proofs of correctness to be given for μISA programs, and it can be readily mechanized. The semantics demonstrates how the weak-

est precondition semantics, developed for sequential architectures, can be easily extended to any synchronized parallel architecture.

Most importantly, the semantics is conceptually simple and theoretically sound. However, to make this semantics useful, the wavefront-based proof method was derived from it. The key point of the method is that it is practically useful, enabling the compact and manageable verification of μISA programs. Thus, for the case of the microprogrammed ISA, all aspects of the issue of confident and efficient programmability have now been addressed.

The proof method reflects the parallel and pipelined nature of the μISA. It generates proofs with a structure reflecting the program's ISADL encoding (ie. reflecting program compression). The method, coupled with equation (6.9), gives an effective semantics for wavefront-based μISA programs (in this case ISADL programs). The convenience of ISADL for use with the proof method indicates that for (μ)ISA programs, the wavefront is the most natural and intuitive 'semantic unit', and hence indicates that the wavefront model is appropriate for developing high-level (ISA) language concepts (cf. ISA Subroutining). While the proof method at least partially reflects the programmer's intuitions, its basic ideas can be used to develop other wavefront-based program verification techniques.

At a first glance, the microprogrammed ISA seems a difficult architecture to program. However, the semantics and wavefront-based proof method, since they can be applied as easily to each, indicate that the μISA is not (on the low level) significantly more complex than the ISA or even the SIMD mesh. Furthermore, macro-level μISA programs may be verified by the semantics or the proof technique, by simply extending the communication register structure (see Section 4.6).

6.7.1 Discussion of the proof method

The proof method is general: the only restriction is that the postcondition be *normalized*, which, in our experience, has always been satisfiable. The method has been shown to be theoretically valid, and, with the interpretation of any introduced CRTS being under (implicit) existential quantification, also yields the

weakest precondition. The proofs generated by the method were presented in an (experimental) tabular form: this eliminated the unnecessary duplication of information.

The proof method introduces a minimum of CRTS, which were necessary to keep the *normalization* property. This has the bonus that arbitrary boundary conditions can be handled in a uniform way. It also has the advantage of reducing the size of the *live* conjuncts of the precondition, as was done for the Red Squares program of Section 6.6.2. However, the proofs sometimes may still require some careful (subscript) bookeeping — it is fair to say that formal program verification is hard to make simple.

The proof method may however appear somewhat inelegant. After trying hard to find a more elegant and yet as useable proof method based on weakest precondition semantics, it remains an open issue as to whether such a proof method can be found.

The proof method compares favourably in terms of proof size with the corresponding weakest precondition-based proofs for uniprocessor programs. This is partly due to the implicit quantification for μISA programs replacing the explicit (nested) loops of the corresponding sequential programs. Compared with a proof technique based more directly in the ISA semantics (cf. the proof, adequate for only constant boundary conditions, of the LCS program in [34, Sect.7.2]), the method yields more rigorous and shorter proofs. The latter is because much of the bookeeping is done by the results of Section 6.5.2, and that the ISA instructions need no longer be broken down into two steps, due to the *normalization* property being maintained.

The proof method captures some of the intuitive μISA programming ideas (see Section 6.4), with the idea of explicit array cell 'states' being modelled (for the communication registers) by the CRTS technique, with the normalization property enabling global array predicates to be separated into conjuncts corresponding to array cells. It also hides the counter-intuitive aspects of the semantics, which requires 'padding' of μISA programs and splitting instruction into two phases. However, the concept of cell 'states' is not completely modelled by the method,

and, more seriously, the proof method reasons in the reverse direction to the programmer (since it is based on the weakest precondition semantics). The latter makes induction of invariants, and their proofs, more difficult and confusing than they need be.

The proof method appears to be mechanizable, except for operations, such as solving recurrences and for inducing *invariants*. In such cases, intelligence is required, but this is a price that must always be paid for compact, parameterized proofs. A semi-automated assistant system would form a reasonable compromise. In general, mechanization of proofs requires a similar degree of intelligence just to keep their outputs readable, and often such a proof is not convincing unless it is independent of the array size, or at least uses a large (fixed) array size.

The proof method can be adapted to other deterministic, communication register-based mesh architectures, with a 'snapshot' semantics as given by the transformed program P' of Section 6.3, by simply modifying the k_{ijt} transformation. The results of Sections 6.5.2 and 6.5.3 require only that this transformation be of the form $k_{ijt} = t - f(i,j)$. For example, an SIMD mesh would use the trivial transformation $k_{ijt} = t$ (ie. $\mu = \sigma = 0$) and and a 'flat' wavefront ISA would use $k_{ijt} = t - (i-1)$ (ie. $\mu = 1$ and $\sigma = 0$). A more interesting example is the rectangular (MIMD) mesh for computing symmetric eigenvalues of [3] which would use $k_{ijt} = t - |i-j|$ (see Section 7.1.1). Generally, a program written for a MIMD mesh will have to have some simple k_{ijt} transformation if it is to be readily understood.

The proof method uses a transformation from a more convenient (abstract) time-skewed frame of reference (using essentially equivalent predicates) to a global (concrete) frame of reference. In this sense, it has an interesting parallel with the abstract data type refinement technique [16], which has found application in the verification of VLSI circuits for pipelined computers [8]. A direct application of such techniques might also be suitable for the μISA.

6.7.2 Future research on mesh program verification

Synchronous, deterministic and simple models of meshes, such as the μISA, are amenable to relatively simple semantic modelling and verification methods, as compared with other models of parallel computation. Issues of synchronization are much simpler, while non-determinism, shared variables, deadlock and livelock do not occur. This makes simple models such as the ISA and μISA attractive, at least in the short term.

The application of program verification techniques on these types of meshes is expected to be more important and more successful than for uniprocessors:

- more important, because subtle timing effects in the μISA make it more difficult to program[8].

- more successful, since the μISA is programmed in terms of short subroutines (which are convenient for proof methods), has simplicity of programming features (no data-dependent control and cells only perform simple assignments, while I/O can be handled in a uniform way) and has also implicit quantification over array subscripts (simplifying proof bookeeping).

Thus, it can be argued that it is important to develop intuitive and useable program verification methods for such architectures.

The proof method presented in this chapter was made less intuitive and useable than desired, due to the fact that the weakest precondition semantics works from the postcondition to the precondition, whereas it is natural to reason in the opposite (forward) direction. This has limited the popularity of weakest precondition semantics in general. However, one of the earliest versions of axiomatic semantics reasoned in the forward direction [11], and a topic for future research would be to develop the wavefront-based proof method on this semantics. Also, the CRTS technique might be introduced more naturally into this semantics.

It may be commented that the microprogrammed ISA semantics transforms an ISA program P in a natural *wavefront* form into a sequence P' of 'snapshot'

[8]The RedSquares program of Section 2.2.2.1 had a bug, surviving several months, exposed by applying the proof method on it.

programs, and the proof method then effectively reverses this transformation. A topic for future research might then be to give a direct wavefront-based semantics, using the intuitive ideas of the proof method. The proof method has indicated that such a semantics is both feasible and worthwhile, but to give such a semantics is beyond the scope of this thesis.

Chapter 7

Program Compression for Processor Arrays

7.1 Motivations

Previous chapters have developed control structures supporting program compression and microprogramming techniques which improve the overall efficiency, reduce the overall hardware and increase the flexibility of the (microprogrammed) ISA. This architecture supports a basic wavefront programming model and has no cell program memory; however, it has reasonably efficient implementations of most array algorithms. This chapter examines to what extent these techniques can be applied to the Processor Array (PA) to achieve the same advantages. The PA[1] has a (minimal) program control unit (PCU), which in turn is expected to enable it to support an *extended* wavefront programming model. The PA is intended to be fine- to medium-grained, and has the following properties:

- the PA cell's PCU (program) memory has $O(\log n)$ area (required to efficiently store $O(n)$ iterations of instruction sequences). Such an area upper bound is used in practice, but it requires that the PA programs can be efficiently compressed *vertically*. Thus, arbitrary MIMD mesh programs cannot be directly executed on this architecture without having to reload

[1]This is slightly less powerful (in theory) than the Processor Array of Section 4.4, which effectively has a PCU of unlimited size.

each cell's program memory every $\Omega(\log n)$ cycles.

- the PA has limited data-dependent operations. For example, an instruction can be masked using the cell's status bits, which may be set in a data-dependent way. Section 7.4 discusses how this condition may be efficiently relaxed.

- the PA is communication register-based, with a fixed (nearest neighbour) interconnection pattern. The *timesharing* of processes is not permitted over the PA. Also, the LSGP partitioning scheme (see Section 2.5.2) is not to be used on the PA, since this scheme complicates the structure of PA programs. These assumptions are made mainly for the sake of simplicity.

The control logic required for the PCU includes decrement operations and zero equality tests over $\log n$-bit integers (a description for a suitable PCU can be found in the ISAC *matrix generator* cell of Section 3.7.3). In return for the expense of the PCU control structure, we desire that (the program compression method for) the PA can implement a wider programming model than can the μISA. This model is called the *extended wavefront* model[2], whose programs can use:

1. (μ, σ) wavefronts of arbitrary (but constant) μ and σ. Thus, the PA should be able to simulate, with no time loss, any efficiently compressible SIMD or microprogrammed ISA program. Furthermore, the PA should be more efficient than the μISA for algorithms requiring row-dependent operations.

2. *partitioned* wavefronts, in which different wavefront programs are executed in each component of a partitioning of the PA (eg. the symmetric eigenvalue program of Section 7.1.1).

3. *interleaved* wavefronts, in which the overall program is expressed as a *superposition* of a few different wavefront programs (see Section 7.4.1).

[2]For PAs, the concept of a *wavefront* is extended slightly in that a wavefront may be interpreted differently as it passes through the array.

To our current knowledge, this class includes all PA programs that are useful in practice.

The control structures for loading programs from this model are required to be suitable for large-scale PAs and to have modest area overheads, ie.:

- the program loading mechanism is systolic and a minimum (ideally $O(\log n)$) of PA cycles are lost during program loading.

- the loading process requires operations no more expensive than those used by the PCU, ie. decrements/comparisons with zero over $O(\log n)$-bit integers, and stack/memory accessing.

The Processor Array Compressed language (PAC) is a low-level PA programming language implementing, with these requirements, program compression for the extended wavefront programming model on the PAs described above. Section 7.1.1 introduces some examples motivating PAC. It also introduces the notation of PAC, which is similar to that of ISAC (see Section 3.7). Section 7.1.2 discusses the extent to which program compression methods are implemented on existing PAs. This leads to the discussion for the alternative implementations of PAC in Section 7.1.3. Section 7.2 defines the PAC notation, describes a translation of PAC into a slightly lower-level language, called PACl, and describes an efficient implementation of PACl. After considering the more complicated examples of extended wavefront programs in Section 7.3, extensions to PAC are proposed in Section 7.4. Section 7.5 gives a cost analysis of the alternative implementations of PAC and discusses the application domain for which PAC is considered viable for PAs.

The contribution of this chapter is to apply the program compression concepts developed earlier in this thesis to Processor Arrays. This exposes general issues of program loading for PAs (Sections 7.1.3 and 7.5) and the need for PA control structures to implement *interleaving* (Sections 7.3 and 7.4.1). The basic concepts of PAC can be understood from Sections 7.1.1 and 7.2.

7.1.1 Simple examples of PAC programs

This section gives some simple examples motivating PAC and introducing its notation. They also motivate the requirements of program compression for Processor Arrays.

The (ComputeA_k)$^{n-1}$ program (a simplification of the RedSquares program of Section 2.2.2.1) can be expressed as the sequence of instructions executed at each PA cell. Figure 7.1(a) gives a program using an ISA (ie. a $(1,1)$) wavefront, where activity propagates one unit south and one unit east each time step. Note that the cell instruction sequences are executed from the bottom up. This is almost identical to the corresponding ISA program except for the '**0**' instruction in the bottom row. Figure 7.1(b) illustrates the program for a SIMD (ie. a $(0,0)$) wavefront, where activity begins simultaneously in all cells. Note that this program has a lower period than any equivalent program on ISA or SIMD meshes (whose masking consumes extra instruction cycles). This example then illustrates the advantage in efficiency of PAC since it does not require any (cell position-dependent) masking capability.

The ISA wavefront program is written in PAC as (order of execution is left-to-right):

$$(\circ\ [\leftarrow\ _2|\mathbf{0}\ _3]\ \circ\ \left[\frac{\uparrow\ _2}{\mathbf{0}\ _3}\right])^2$$

for a 3×3 PA, and for an $m \times n$ PA:

$$(\circ\ [\leftarrow\ _{n-1}|\mathbf{0}\ _n]\ \circ\ \left[\frac{\uparrow\ _{m-1}}{\mathbf{0}\ \ _m}\right])^{n-1}$$

The SIMD wavefront program is the same, except that the 'NoOp' ('$\circ$') instructions are omitted.

This program introduces the *repetition* construct '$(P)^k$', read as:

> cells (i,j) (in the current context) execute k repetitions of the program P (note that k may be a restricted function of i and j),

the *column selection* construct '$[P\ _r|Q\ \ _n]$', read as:

> columns $j|0 < j \leq r$ execute program P and columns $j|r < j \leq n$ execute program Q,

$\uparrow)^2$ $\circ$ $\leftarrow$ $(\circ$	$\uparrow)^2$ $\circ$ $\leftarrow$ $(\circ$	$\uparrow)^2$ $\circ$ $\mathbf{0}$ $(\circ$
$\uparrow)^2$ $\circ$ $\leftarrow$ $(\circ$	$\uparrow)^2$ $\circ$ $\leftarrow$ $(\circ$	$\uparrow)^2$ $\circ$ $\mathbf{0}$ $(\circ$
$\mathbf{0})^2$ $\circ$ $\leftarrow$ $(\circ$	$\mathbf{0})^2$ $\circ$ $\leftarrow$ $(\circ$	$\mathbf{0})^2$ $\circ$ $\mathbf{0}$ $(\circ$

(a) for ISA wavefront ($\mu = \sigma = 1$)

$\uparrow)^2$ $(\leftarrow$	$\uparrow)^2$ $(\leftarrow$	$\uparrow)^2$ $(\mathbf{0}$
$\uparrow)^2$ $(\leftarrow$	$\uparrow)^2$ $(\leftarrow$	$\uparrow)^2$ $(\mathbf{0}$
$\mathbf{0})^2$ $(\leftarrow$	$\mathbf{0})^2$ $(\leftarrow$	$\mathbf{0})^2$ $(\mathbf{0}$

(b) for SIMD wavefront ($\mu = \sigma = 0$)

('$\circ$' denotes 'NoOp', '$\leftarrow$' denotes '$C \leftarrow CC_E$', '$\mathbf{0}$' denotes '$C \leftarrow 0$' and '$\uparrow$' denotes '$C \leftarrow CC_S$')

Figure 7.1: Instructions executed at each cell for (ComputeA_k)^2 program for a 3×3 PA.

and the *row selection* construct '$\left[\begin{array}{c} P \quad s \\ \hline Q \quad m \end{array}\right]$', read as:

> rows $i | 0 < i \leq s$ execute program P, and rows $i | s < i \leq m$ execute program Q.

The selection constructs allow the construction of partitions of the PA (program).

Often, it is necessary to have different numbers of repetitions in different PA cells. This is illustrated in Figure 7.2. With $\mu = 1$, the LoadMat program is equivalent to the program of the same name in Section 3.7.1. Program ISAtoSIMD creates a SIMD wavefront from an ISA wavefront by delaying cell (i, j) by $(m - i + n - j)$ cycles[3]. This type of program is very important in PAC.

The LoadMat program is coded in PAC as '$(\rightarrow)^{3-j+1}(\mathrm{o})^{j-1}$' for a 3×3 PA, and as '$(\rightarrow)^{n-j+1}(\mathrm{o})^{j-1}$' for an $m \times n$ PA. In ISAC, this program is coded as '$(\rightarrow)^{\triangleright.n}(\mathrm{o})^{\triangleleft.0}$'; however, for PAC, a more general and readable form is convenient. The ISAtoSIMD program is coded in PAC as '$(\mathrm{o})^{3-i}(\mathrm{o})^{3-j}$' for a 3×3 PA, and as '$(\mathrm{o})^{m-i}(\mathrm{o})^{n-j}$' for an $m \times n$ PA. Note that this last is equivalent to '$(\mathrm{o})^{m-i+n-j}$', which is a less preferable form since the exponent can exceed n (and so is harder to store in the PA cell's PCUs).

A more interesting program is SymEigen, the $n \times n$ PA implementation of the symmetric eigenvalue algorithm (full systolic version) of [3, pp75-80]. This program can be viewed as wavefronts moving outwards from the main diagonal (see Figure 7.3(c)-(e)). It can then be expressed by partitioning the PA into a lower triangular, a main diagonal and an upper triangular parts, which have $(1, -1), (0, 0)$ and $(-1, 1)$ wavefronts respectively. If the PA is assumed to be initially using a SIMD wavefront, delaying cell (i, j) by $|i - j|$ cycles will achieve this effect. This is demonstrated in Figure 7.3(a), with one 'sweep' of the eigenvalue computation illustrated in Figure 7.3(b). L, D and U are rather complex instruction macros, the details of which are irrelevant here. In Figure 7.3(c)-(e), the instructions executed at each cell at time steps 1,4 and 5 of this program are given (the subscripts annotated to the instructions indicate which repetition produced them).

[3] A converse program exists to create an ISA wavefront from a SIMD wavefront by delaying cell (i, j) by $i + j - 2$ cycles.

$(\rightarrow)^3$	$\circ$ $(\rightarrow)^2$	$(\circ)^2$ $\rightarrow$
$(\rightarrow)^3$	$\circ$ $(\rightarrow)^2$	$(\circ)^2$ $\rightarrow$
$(\rightarrow)^3$	$\circ$ $(\rightarrow)^2$	$(\circ)^2$ $\rightarrow$

(a) LoadMat program ($\sigma = 1$)

$(\circ)^2$ $(\circ)^2$	$(\circ)^2$ $\circ$	$(\circ)^2$
$\circ$ $(\circ)^2$	$\circ$ $\circ$	$\circ$
$(\circ)^2$	$\circ$	

(b) ISAtoSIMD program ($\mu = \sigma = 1$)

('$\circ$' denotes 'NoOp' and '$\rightarrow$' denotes '$C \leftarrow C_W$')

Figure 7.2: Instructions executed at each cell for programs with cell-position dependent iterations for a 3×3 PA.

In PAC, the complete program is expressed as:

$$\text{SymEigen}: \begin{matrix} [(\circ)^{i-j}{}_{i-1}| & {}_i|(\circ)^{j-i}{}_n] \\ [(L\ \circ\ \circ)^n{}_{i-1}| & (D\ \circ\ \circ)^n\ {}_i|(U\ \circ\ \circ)^n{}_n] \end{matrix}$$

noting that the first *column selection* is equivalent to '$(\circ)^{|i-j|}$'. The second *column selection* can be read as:

> cells $(i,j)|j < i$ execute $(L\ \circ\ \circ)^n$, cells $(i,j)|j = i$ execute $(D\ \circ\ \circ)^n$ and cells $(i,j)|j > i$ execute $(U\ \circ\ \circ)^n$.

Allowing the *boundaries* in the *column selection* to depend on the row position i enhances the power of PAC, by creating programs for non-rectangular partitionings of a PA.

7.1.2 Program compression on existing PAs

This section examines the extent to which existing PAs incorporate program compression techniques. A discussion of how (fine-grained) SIMD meshes use program compression was given in Section 3.2.2; here, the discussion concentrates on (coarser-grained) MIMD meshes. From this discussion, the requirements and advantages of PAC can be more clearly seen.

	$\circ$	$(\circ)^2$	$(\circ)^3$	$(\circ)^4$
$\circ$		$\circ$	$(\circ)^2$	$(\circ)^3$
$(\circ)^2$	$\circ$		$\circ$	$(\circ)^2$
$(\circ)^3$	$(\circ)^2$	$\circ$		$\circ$
$(\circ)^4$	$(\circ)^3$	$(\circ)^2$	$\circ$	

(a) delays to set up wavefronts part

$(D')^5$	$(U')^5$	$(U')^5$	$(U')^5$	$(U')^5$
$(L')^5$	$(D')^5$	$(U')^5$	$(U')^5$	$(U')^5$
$(L')^5$	$(L')^5$	$(D')^5$	$(U')^5$	$(U')^5$
$(L')^5$	$(L')^5$	$(L')^5$	$(D')^5$	$(U')^5$
$(L')^5$	$(L')^5$	$(L')^5$	$(L')^5$	$(D')^5$

(b) actual eigenvalue computation part ($D' =$ '$D \circ \circ$', etc.)

D_1	$\circ$	$\circ$	$\circ$	$\circ$
$\circ$	D_1	$\circ$	$\circ$	$\circ$
$\circ$	$\circ$	D_1	$\circ$	$\circ$
$\circ$	$\circ$	$\circ$	D_1	$\circ$
$\circ$	$\circ$	$\circ$	$\circ$	D_1

(c) instructions at $t = 1$

D_2	$\circ_1$	$\circ_1$	U_1	$\circ$
$\circ_1$	D_2	$\circ_1$	$\circ_1$	U_1
$\circ_1$	$\circ_1$	D_2	$\circ_1$	$\circ_1$
L_1	$\circ_1$	$\circ_1$	D_2	$\circ_1$
$\circ$	L_1	$\circ_1$	$\circ_1$	D_2

(d) instructions at $t = 4$

$\circ_2$	U_2	$\circ_1$	$\circ_1$	U_1
L_2	$\circ_2$	U_2	$\circ_1$	$\circ_1$
$\circ_1$	L_2	$\circ_2$	U_2	$\circ_1$
$\circ_1$	$\circ_1$	L_2	$\circ_2$	U_2
L_1	$\circ_1$	$\circ_1$	L_2	$\circ_2$

(e) instructions at $t = 5$

Figure 7.3: SymEigen program on a 5×5 PA

The Inmos Transputer [18] can be easily configured as a medium-grained $n \times n$ PA (microprocessor sized granularity, with large PCUs). The program loading algorithm simply loads programs in depth-first order over the n^2 nodes of this Transputer network. Thus, no special effort appears to be taken to incorporate program compression on (small-scale) Transputer networks.

The Cellular Array Processor (CAP) [19] is a scalable (micro-processor sized granularity) PA capable of implementing in software various communication topologies. The cells are also sophisticated microprocessors. The architecture has the capability to efficiently down-load (broadcast) identical programs to its cells, using its "command bus". For the CAP, the PAX computer [56, p90] and various other message-passing architectures [2, pp104-106], a *cellprogram*

> ie. a 'formula' for a generic cell program (taking the cell's address or 'id' as a parameter)[4]

is written in a high-level language. After compilation, this can be effectively passed to all cells: different cells can interpret the cellprogram differently according to their address 'id'. Such architectures implicitly incorporate some program compression, and are expected to have reasonably efficient program loading.

The Warp processor [1] is a coarse-grained linear systolic array. Its cells use a very long (272-bit) instruction word (VLIW) which makes it advantageous to use a PCU even for SIMD-like or linear ISA-like programs. Otherwise, to send a new VLIW instruction to a cell every cycle would result in an unacceptably high inter-cell I/O bandwidth. The Warp langauge W2 describes Warp programs in terms of the *cellprogram* concept mentioned above [27, Sect.4.1]. The Warp machine can down-load identical instructions to each its its cells using a 'serial-chain' (ie. systolic program loading) and its PCUs are very large (several K instruction words).

Generally, PAs can efficiently perform serial down-loading and execution of *cellprograms*, but usually make little further effort to utilize program compression. The reasons for this are summarized as follows:

[4] cf. the PAC programs of Section 7.1.1, which take as parameters the cell row and column addresses i and j.

- the array sizes of existing PAs are still small, so that even loading cells individually is still viable (eg. as for Transputer meshes).

- the cells are sophisticated (microprocessor size or larger) so that loading times are generally dominated by program compile times (from a host computer).

  eg. the **Matrix multiply** algorithm for a 10 cell Warp array takes 1.7 minutes to compile (using an optimizing compiler) compared with a 25ms execution time for 100×100 matrices [4, p271]. The Transputer presents a similar situation.

  Also, the cell PCUs of these PAs are typically large (several K instruction words), enabling many programs to be stored simultaneously in each cell (thus the reloading of programs need occur less often).

- the programs themselves are typically computation-bound and operate on large data sizes only.

  eg. the **Matrix multiply** algorithm for a 10 cell Warp array should take $25 \times .09 = 2.25$ms for a 30×30 matrix. With a peak Warp instruction loading rate of one Warp instruction per 67μs [1, p1529], only ≈ 30 instructions could be loaded during this time. Since this program uses partitioning and requires pipeline initialization and optimization, this time may be insufficient to load the **Matrix Multiply** program. Also the program startup time of 5ms is significant compared to this execution time.

  Thus, it can be seen that these arrays have sufficiently fast program loading times for programs which use large data sizes and/or are repeated for many different data sets.

Considering the reasons mentioned above, it is not too surprising that few details are published on PA program loading mechanisms and performance. Cell-programs can be downloaded reasonably efficiently on most of these architectures, possibly with a capability to select single (or ranges of) PAs for separate loading.

Provided the program loading mechanisms are systolic, as for the Warp array, scalability presents no particular problem to program loading. Thus, it might be thought that the implicit program compression achieved by the cellprogram concept is sufficient. However, in the next section, the possible overheads of such an approach are examined, and from this the advantages of introducing explicit program compression concepts can be seen. Also, for these PAs, program loading times can still be significant compared with program execution times, resulting in a loss of overall PA performance. The overhead in both time and area in program loading can, without too much effort, be made very small. This applies more particularly to finer-grained PAs than to those discussed in this section.

7.1.3 Alternative approaches for PAC

As demonstrated in Section 7.1.2, an efficient way to both express and (systolically) download PA programs is to use the cellprogram concept. This can be incorporated by the following alternative approaches for the implementation of PAC, which are analysed in more detail in Section 7.5:

1(a). load a cellprogram for the complete array into each cell and interpret it at run-time.

This interpretation is done via registers in each cell which contain the cell address or 'id' (which has to be initialized by some other mechanism). This approach is efficient provided these cellprograms are fairly simply interpreted[5]. It does however, present the following disadvantages:

- the interpretation in general requires operations (eg. $O(\log n)$-bit additions, comparisons, etc.), which results in a loss of PA cycles. This is considerable for PA implementations of the more complex programs (cf. the PA equivalent of the ISA program RotH'_d of Section 7.2.3) and cannot easily be avoided. This effect is made more serious since for PA algorithms, the number of overall PA cycles lost is determined by the worst case loss of any of its cells.

[5]This approach appears to be taken for SIMD machines, which use cell addresses to determine whether to mask instructions, and most existing PAs.

– the cellprogram may be considerably more bulky than the program that any PA cell actually needs to execute (cf. the SymEigen program of Section 7.1.1 with the corresponding programs executed in each cell in Figure 7.3), often resulting in a substantial and unnecessary increase in the size of the cell PCU.

1(b). load the cellprogram for the complete PA into each cell and interpret it at load-time.

This approach avoids the disadvantages of the former approach, but possibly introduces a more complex PA cell design, since the 'program load' mode of PA cell operation would be more specialized. A variant of this is to interpret the cellprogram dynamically while loading it, and this is suitable to be combined with approach 2(b) mentioned below.

2(a). serial program loading and execution, ie. load the next program after executing the current program.

This approach is taken for most PAs, and requires that program loading and execution occur sequentially within (each cell of) the PA. It has the disadvantage that the PA is delayed since the loading of the next program has to wait for the execution of the current program to cease in the PA (and the execution of the next program must wait until its loading has ceased).

2(b). overlapped program loading and execution, ie. load the next program while executing the current program.

This approach avoids the loss of PA cycles of approach 2(a), at the expense of extra cell hardware. It requires that the cell PCU tables be capable of storing at least two programs simultaneously. It also requires separate logic to load the cell PCUs with a new program while the current program is being executed. This in turn requires dedicated circuitry to perform the cellprogram interpretation (eg. $O(\log n)$-bit adder, small stack etc.). This can be mitigated partially by the fact that the $O(\log n)$ size cellprograms can generally be loaded over $\Omega(n)$ cycles (being the period of the current PA program), so that the additions etc. could be performed bit-serially.

Coupled with dynamic interpretation of the cellprogram during loading, this is the approach recommended for PAC, since it enables the interpretation of the (PAC) cellprogram to be expressed in terms of only increment/decrement and zero equality operations (over $\log n$-bit integers). These operations are no more expensive than those required to control the cell PCUs. How this can be done is explained in Section 7.2. This method may however result in a small loss of generality.

The general objective of this chapter is to present an efficient program compression method for a PA model which is substantially more powerful than the microprogrammed ISA. The choice to have an $O(\log n)$ area PCU in each cell enables an ISAC-like program compression method. However, generalizing the ISADL concept, which requires only an $O(1)$ area PCU, might result in a more efficient solution. Unfortunately, to extend ISADL to have (μ, σ) wavefronts with μ or σ negative would approximately double the I/O bandwidth required, and $(0, \sigma)$ or $(\mu, 0)$ wavefronts would remain impossible. Also, different direction wavefronts in different partitions of a PA would be hard to implement using the ISADL concept. In short, to have a more powerful PA model than the microprogrammed ISA, the extra area of the PCUs for an ISAC-like method is acceptable.

7.2 Implementation of PAC

This section formally defines PAC and describes a transformation into a lower-level language, called PACl. Both the definition and the transformation impose constraints on PAC programs. Efficient and simple control structures for the implementation for PACl are then described, and it is then indicated how PAC *delay programs* provide quite general PA program synchronization.

This section is important for the reading of Section 7.4.2.

7.2.1 Definition of the PAC constructs

The PAC notation, sharing similarities with ISAC and ISADL, can be defined very simply. The *projection* of a PAC program P onto the PA cell (i_0, j_0), for

$1 \leq i_0 \leq m$ and $1 \leq j_0 \leq n$, is given by finding the projection onto the j_0th column of the projection of P onto the i_0th row of the PA:

$$\mathrm{C}_{j_0 n}(\mathrm{R}_{i_0 m}(P\,(i/i_0))\,(j/j_0))$$

The substitutions (i/i_0) and (j/j_0) evaluate the *repetition* counters and *selection* boundaries at cell (i_0, j_0).

Since the PAC notation is fairly obvious and the definitions of the row and column projection functions, R and C, are bulky, the formal definition of PAC is put in Appendix 7.A. Note, however, that PAC allows the column index to be present in a row selection construct (but not conversely); for this reason, the projection is performed first on the rows and then on the columns. Note also that the row and column selection constructs distribute over all PAC constructs in a fairly obvious way, a property useful in manipulating PAC programs.

An arbitrary PA program of period t represented by the $m \times n$ matrix P (where P_{ij} is the program to be executed at cell (i, j)) can be coded in PAC as:

$$\left[\begin{array}{cccccccc}
[P_{11} & 1| & P_{12} & 2| & \cdots & P_{1n} & n] & 1 \\
\hline
[P_{21} & 1| & P_{22} & 2| & \cdots & P_{2n} & n] & 2 \\
\hline
\vdots & & \vdots & & & \vdots & & \vdots \\
\hline
[P_{m1} & 1| & P_{m2} & 2| & \cdots & P_{mn} & n] & m
\end{array}\right]$$

Such an expression in general requires an $O(n^2 t)$ loading period, with an $O(t)$ area cell PCU. In practice, however, the loading period and the cell PCU area need only be $O(\log n)$. How this is achieved will be elaborated below.

7.2.2 A PAC program interface

An efficient PAC program interface is proposed in Figure 7.4. A PAC program P is passed (from the top) through an $m \times 1$ *column generator*, the i_0th row of which evaluates $P_{i_0} = \mathrm{R}_{i_0,m}(P\,(i/i_0))$. This gets passed through (from the left) the i_0th row of an $m \times n$ *column generator* array, the j_0th column of which evaluates $P_{ij} = \mathrm{C}_{j_0,n}(P_i\,(j/j0))$. The column generator array is embedded into an $m \times n$ PA. The row and column generator combined give a *matrix generator* (see Chapter 3). Note that the asymmetry of the system reflects the definitions of R and C (see equations (7.9-7.10)).

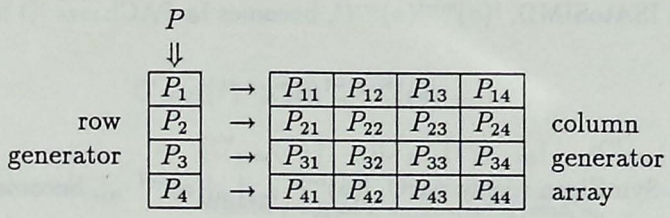


Figure 7.4: PAC program interface for a 4×4 PA

This program interface is to be loaded systolically in a fixed velocity wavefront. As shown in Figure 7.4, the ISA wavefront, with $\mu = \sigma = 1$, is chosen. This is because a fixed velocity wavefront for the program loading reduces hardware costs. Of these, wavefronts with $|\mu| = |\sigma| = 1$ are the cheapest to implement. It is also useful to match the program loading wavefronts with those of as many of the PA's program as possible (this reduces the cell PCU size — see Section 7.5). Since the ISA can efficiently implement most PA algorithms, the majority of programs would probably use the $(1, 1)$ wavefront.

The row and column generators can efficiently evaluate the required projections by updating and passing systolic counters (which are used to evaluate row or column position dependent loop indices and repetition constructs). This counting operation can be performed bitwise if necessary, and so a $(1, 1)$ program loading wavefront can be easily implemented.

Note that this program interface can easily allow the PA system to be *partitionable* into a small number of sub-arrays, by simply providing row generators for each desired sub-array (see Section 2.5.1).

7.2.3 Transformation into PACl

PAC requires to be transformed to a lower-level language, called PACl, for direct, efficient loading. PACl is basically a two-dimensional extension of ISAC (see Section 3.7). This transformation replaces expressions dependent on i and j in a PAC program with 'recipies' telling the row and column generator arrays how to compute their values during program loading. It is illustrated by the following simple examples:

- Program ISAtoSIMD, '$(\circ)^{m-i}(\circ)^{n-j}$', becomes in PACl:

$$(\circ)^{\triangleright_i \cdot m-1}(\circ)^{\triangleright_j \cdot n-1}$$

- Program SymEigen (delay part), $[(\circ)^{i-j}\ {}_{i-1}|\ {}_{i}|(\circ)^{j-i}\ {}_{n}]$, becomes in PACl:

$$[(\circ)^{\triangleleft_i \cdot \triangleright_j \cdot 0}\ {}_{\triangleleft_i \cdot 0}|\ {}_{\triangleleft_i \cdot 1}|(\circ)^{\triangleleft_j \cdot 1}\ {}_{n}]$$

  Similarly, the converse delay program, $[(\circ)^{n-i+j}\ {}_{i-1}|(\circ)^{n}\ {}_{i}|(\circ)^{n+i-j}\ {}_{n}]$, becomes in PACl:

$$[(\circ)^{\triangleright_i \cdot \triangleleft_j \cdot n}\ {}_{\triangleleft_i \cdot 0}|(\circ)^{n}\ {}_{\triangleleft_i \cdot 1}|(\circ)^{\triangleright_j \cdot n}\ {}_{n}]$$

- Program RotH'_d of Section 5.2.2 can be coded[6] into PAC as:

$$[\,(\circ\ \circ)^{d+1-j \bmod_1 d}(\rightarrow\ \leftarrow)^{j \bmod_1 d}\ {}_{d}|\ (\rightarrow\ \leftarrow)^{d+1-j \bmod_1 d}(\circ\ \circ)^{j \bmod_1 d}\ {}_{2d}]$$

  This is translated to:

$$[\,(\circ\ \circ)^{\triangleright_j \cdot d}(\rightarrow\ \leftarrow)^{\triangleleft_j \cdot 1}\ {}_{d}|\ (\rightarrow\ \leftarrow)^{\triangleright_j \cdot d}(\circ\ \circ)^{\triangleleft_j \cdot d}\ {}_{2d}]$$

Here, for a PACl program component P with an expression operated on by $\triangleleft_i$ ($\triangleright_i$), the row generator increments (decrements) the expression's corresponding systolic counter at all rows where P is selected. This counter is reset to its initial value at the rows corresponding to the beginning of each selection. A similar situation exists for $\triangleleft_j$ ($\triangleright_j$) for each row of the column generator array.

The PACl equivalent of a PAC program P, for m and n being powers of two[7], is given by:

$$\mathrm{R}'_{1m}(\mathrm{C}'_{1n}(P))$$

For PAC programs $P_1, \ldots, P_l$, integer expressions $e, n_1, \ldots, n_{l-1}$ (which possibly contain i), and integers n_l and n' where n_l divides n', the non-trivial parts of the

[6] For simplicity of expression in PAC, an extra 'diagonal' has been inserted at the beginning and end of this program.

[7] Otherwise, the expression $\mathrm{R}'_{1m'}(\mathrm{C}'_{1n'}(P))$, where $m' = 2^{\lceil \log m \rceil}$ and $m' = 2^{\lceil \log m \rceil}$, should be used instead.

definition of C′ are:

$$\begin{aligned} & C'_{j_0 n'}([P_1 \ _{n_1} | P_2 \ _{n_2} | \ldots | P_l \ _{n_l}]) \\ &= [C'_{j_1 n_l}(P_1) \ _{n_1} | C'_{j_2 n_l}(P_2) \ _{n_2} | \ldots | C'_{j_l n_l}(P_l) \ _{n_l}] \\ & \qquad \text{where } j_1 = j_0 \bmod_1 n_l, \ j_2 = (j_0 + n_1) \bmod_1 n_l, \ \ldots \\ C'_{j_0 n'}((P)^{e+c(j)}) &= (C'_{j_0 n'}(P))^{\mathrm{op}_j(c,n').\, e + c(j_0)} \end{aligned} \tag{7.1}$$

The definition of R′ is similar for the row selection and repetition constructs (noting that no expressions may now contain j) except that also, for integers $n_1, \ldots, n_l$ $(n_l | m')$:

$$\begin{aligned} R'_{i_0,m'}([P_1 \ _{n_1 + r_1(i)} | P_2 \ _{n_2 + r_2(i)} | \ldots | P_l \ _{n_l}]) = & \ [\, R'_{i_0 m'}(P_1) \ _{\mathrm{op}_i(r_1,m').n'_1} | \\ & \ R'_{i_0 m'}(P_2) \ _{\mathrm{op}_i(r_2,m').n'_2} \\ & \ | \ \ldots \ | \, R'_{i_0 m'}(P_l) \ _{n_l}] \\ & \text{where } n'_k = n_k + r_k(i_0), \ 1 \le k < l \end{aligned} \tag{7.2}$$

For the purposes of efficient implementation, the function op_j (op_i) is defined over only very restricted functions [ie. $c(j), r_1(i)$, etc. of equations (7.1-7.2) are of restricted forms]. This constrains how expressions in PAC programs can depend on j (i) for the implementation of PAC program compression. Currently, op_j is defined (op_i is identically defined):

$$\mathrm{op}_j(c, n') = \begin{cases} \triangleleft_j & , \text{if } c(j) = j \bmod_1 2^k \text{ and } 2^k | n' \\ \triangleright_j & , \text{if } c(j) = -j \bmod_1 2^k \text{ and } 2^k | n' \\ \bullet & , \text{if } c(j) = 0 \end{cases} \tag{7.3}$$

with '$\bullet$' as the default value. This definition is extended in Section 7.4.2. The requirement that 2^k divides n' for a PAC sub-program P, repeating over intervals of n' consecutive columns and containing $c(j) = \pm j \bmod_1 2^k$, ensures that $c(j) = c(j + n')$ and hence P is the same at each interval. In practice, it is not useful to lift this requirement.

For example, with $m = n$, $f(i) = i \bmod_1 n$ $f'(i) = -i \bmod_1 n$, the conversion of the SymEigen program (delay part) into PACl occurs in the following stages:

$$\begin{aligned} & R'_{in}(C'_{1n}([\, (\circ)^{f(i)+f'(j)} \ \ldots \ _{f(i)} | \, (\circ)^{f(j)+f'(i)} \ _n])) \\ &= R'_{1n}([\, C'_{1n}((\circ)^{f(i)+f'(j)}) \ \ldots \ _{f(i)} | \, C'_{f(i)n}((\circ)^{f(j)+f'(i)}) \ _n]) \\ &= R'_{1n}([\, (\circ)^{\triangleright_j .\, f(i)+f'(1)} \ \ldots \ _{f(i)} | \, (\circ)^{\triangleleft_j .\, f(i)+f'(i)} \ _n]) \end{aligned}$$

$$= \mathrm{R}'_{1n}([\,(\mathrm{o})^{\triangleright_j \cdot -1+f(i)} \quad \ldots \quad {}_{f(i)}|\,(\mathrm{o})^{\triangleleft_j \cdot 0} \quad {}_n])$$

$$= [\,(\mathrm{o})^{\triangleleft_i \cdot \triangleright_j \cdot f(1)-1} \quad \ldots \quad {}_{\triangleleft_i \cdot f(1)}|\,(\mathrm{o})^{0\,\triangleleft_j} \quad {}_n]$$

$$= [\,(\mathrm{o})^{\triangleleft_i \cdot \triangleright_j \cdot 0} \quad \ldots \quad {}_{\triangleleft_i \cdot 1}|\,(\mathrm{o})^{\triangleleft_j \cdot 0} \quad {}_n]$$

Similarly, with $g(j) = j \bmod_1 d$ and $g'(j) = -j \bmod d$ (assuming d divides n), the PAC version of the RotH$'_d$ program (see Section 5.2.2) is translated into PACl as follows:

$$\mathrm{C}'_{1n}([\,(\mathrm{o}\ \mathrm{o})^{d+1+g'(j)}(\rightarrow\ \leftarrow)^{g(j)} \quad {}_d|\,(\rightarrow\ \leftarrow)^{d+1+g'(j)}(\mathrm{o}\ \mathrm{o})^{g(j)} \quad {}_{2d}])$$

$$= [\,\mathrm{C}'_{1,2d}((\mathrm{o}\ \mathrm{o})^{d+1+g'(j)}(\rightarrow\ \leftarrow)^{g(j)}) \quad {}_d|\,\mathrm{C}'_{d+1,2d}((\rightarrow\ \leftarrow)^{d+1-g'(j)}(\mathrm{o}\ \mathrm{o})^{g(j)}) \quad {}_{2d}]$$

$$= [\,(\mathrm{o}\ \mathrm{o})^{\triangleright_j \cdot d+1+g'(1)}(\rightarrow\ \leftarrow)^{\triangleleft_j \cdot g(1)} \quad {}_d|\,(\rightarrow\ \leftarrow)^{\triangleright_j \cdot d+1-g'(d+1)}(\mathrm{o}\ \mathrm{o})^{\triangleleft_j \cdot g(d+1)} \quad {}_{2d}]$$

$$= [\,(\mathrm{o}\ \mathrm{o})^{\triangleright_j \cdot d+1-1}(\rightarrow\ \leftarrow)^{\triangleleft_j \cdot 1} \quad {}_d|\,(\rightarrow\ \leftarrow)^{\triangleright_j \cdot d+1-1}(\mathrm{o}\ \mathrm{o})^{\triangleleft_j \cdot 1} \quad {}_{2d}]$$

While the conversion into PACl seems to restrict the form of PAC programs, in practice it usually takes only a little manipulation of a PAC program to put it in PACl-convertible form. As an example, the PAC program '$(\mathrm{o})^{n-|i-j|}$' need only be re-arranged into '$[(\mathrm{o})^{n-i+j} \ {}_i|(\mathrm{o})^{n+i-j} \ {}_n]$'.

7.2.4 Implementation of PACl

This section outlines the loading of PACl programs into the row and column generator cells, describing the required control structures.

The row (column) generator loads row (column) selection constructs in exactly the same way as the ISADL *diagonal restorer* loads [the initial value of] an ISADL diagonal (cf. Section 5.3.2.2). This is because these constructs are very similar in structure to an ISADL diagonal: the only difference is that the arguments of the PAC selector constructs may be long programs, whereas those of an ISADL diagonal are only sub-diagonals. This results in the the boolean variables L and E' of the ISADL diagonal loading algorithm being replaced by short boolean stacks (of the same name) for its PAC adaptation. Hence, for loading P_k for the PAC column selection:

$$[\ldots {}_{n_{k-1}}|P_k \ {}_{n_k}|\ldots {}_n]$$

the PACl loading algorithm would then set top(L) only in columns n_{k-1} to n_k, and top(E') only in column n_{k-1}. The implementation of the PACl algorithm requires

a modest amount of hardware (a small boolean processor is sufficient). The performance of the ISADL algorithm was shown to be very good in general (see Tables 5.1 and 5.2); this performance should carry over to its PAC adaptation.

The evaluation of '$[P_1 \ \ _\delta | \ldots \ \ _n]$' across a PA row requires a bit to be set in its δth column, to mark the boundary between the cells loading P_1 and the others. This marking can be done by passing systolic counters bitwise across the row, taking $O(\log \delta)$ steps. These counters are also used to evaluate the $\lhd_i, \ldots, \rhd_j$ expressions of PACl.

Consider the loading of a PACl repetition construct of the form $(i\ P)^{\lhd_j \cdot c}$ into a cell of the column generator array, where c in an integer, i is an instruction and P is a PACl sub-program. If, in this cell, $\text{top}(L) = 0$, the cell passively passes the construct to the cell to the right. Otherwise, the cell loads this construct in the following way. If $\text{top}(E') = 1$, let v denote c; otherwise v denotes the current value of the previous column's (systolic) counter. v is loaded into the *counter* field of the current entry of the cell's PCU. The cell sets its (systolic) counter to $v + 1$ (to be read by the cell to the right). The *instruction* field of the current PCU is loaded with i, and this entry is marked '(', signifying the beginning of a repetition. Then the sub-program corresponding to P is loaded, the last instruction of which is marked as ')'. A similar case exists for the PACl sub-program $(i\ P)^{\rhd_j \cdot c}$ except that the row passes on $v - 1$ to the cell to the right, instead of $v + 1$.

The row generator loads these constructs, together with the column selection constructs, in a similar way. As an example, consider loading $(P)^{\rhd_j \cdot \lhd_i \cdot c}$ into a row (cell) of the row generator[8]. If $\text{top}(L) = 0$, the row passes on $(P)^{\rhd_j \cdot \lhd_i \cdot c}$ unchanged. Otherwise, the row loads this sub-program in the following way. If $\text{top}(E') = 1$, let v denote c; otherwise v denotes the current value of the previous row's (systolic) counter. The cell sets its (systolic) counter to $v+1$ (to be read by the succeeding row). The cell loads $(P)^{\rhd_j \cdot v}$ (with P processed in a similar way) into the adjacent row of the column generator array.

[8]Equations (7.1-7.2) would have generated $(P)^{\lhd_i \cdot \rhd_j \cdot c}$ rather than $(P)^{\rhd_j \cdot \lhd_i \cdot c}$. However, for the purpose of loading, the former expression is more convenient.

7.2.5 Array synchronization using PAC

In the microprogrammed ISA, the implementation of (μ, σ) wavefronts, and hence array synchronization, is achieved using instruction and selector queues. In the PA, (partitioned) (μ, σ) wavefronts can be implemented by simply executing (partitioned) PAC delay programs (see Section 7.1.1).

The loading of PACl programs proceeds using an ISA wavefront. It is also easy to implement an initial program start signal which also propagates with the same wavefront. However, the PA cell's PCUs may be easily instructed to begin execution of the next program immediately after executing the first program (the second program can be assumed to be loaded by this stage). If the first program has a period appropriately depending on the cell position (i, j), the second program can use any simple wavefront. Figure 7.2(b) demonstrates such a program which transforms an ISA wavefront into an SIMD wavefront. In general, the following synchronization result holds for PAC:

Result 7.1 (Simple wavefront PA synchronization result) *If an $m \times n$ PA is currently using a (μ, σ) wavefront, 'executing' the delay program $\delta(\mu' - \mu, \sigma' - \sigma)$ causes it to use a (μ', σ') wavefront, where:*

$$\delta(\Delta\mu, \Delta\sigma) = \left\{ \begin{array}{ll} ((\mathrm{o})^{\Delta\mu})^{i-1} & \text{if } \Delta\mu \geq 0 \\ ((\mathrm{o})^{-\Delta\mu})^{m-i} & \text{if } \Delta\mu < 0 \end{array} \right\} \left\{ \begin{array}{ll} ((\mathrm{o})^{\Delta\sigma})^{j-1} & \text{if } \Delta\sigma \geq 0 \\ ((\mathrm{o})^{-\Delta\sigma})^{n-j} & \text{if } \Delta\sigma < 0 \end{array} \right\}$$

proof:

Let $\Delta\mu = \mu' - \mu$ and $\Delta\sigma = \sigma' - \sigma$.

Assume that initially the PA uses a (μ, σ) wavefront. Here, the first instruction of the program $\delta(\Delta\mu, \Delta\sigma)$ is executed in cell (i, j) μ steps [σ steps] after it is executed in cell $(i-1, j)$ [cell $i, j-1$)].

If $\mu' \geq \mu$, then this program has a period $\mu' - \mu$ cycles longer in cell (i, j) than in cell $(i-1, j)$. Thus, after executing $\delta(\Delta\mu, \Delta\sigma)$, the next program's first instruction is executed in cell (i, j) $\mu + \Delta\mu = \mu'$ steps after it is executed in cell $(i-1, j)$.

Otherwise, this program has a period $\mu' - \mu$ cycles longer in cell $(i-1, j)$ than in cell (i, j). Thus, after executing $\delta(\Delta\mu, \Delta\sigma)$, the next

program's first instruction is executed in cell (i,j) $\mu - (-\Delta\mu) = \mu'$ steps after it is executed in cell $(i-1,j)$.

A similar argument establishes that after executing $\delta(\Delta\mu, \Delta\sigma)$, the next program's first instruction is executed in cell (i,j) σ' steps after it is executed in cell $(i,j-1)$. □

This result can be simply extended for partitioned wavefront programs. It also gives the delay associated with changing the wavefront parameters — this has already been discussed in Section 4.3. Thus, the PAC model has a simple, efficient and versatile method of providing PA array synchronization for the class of PA algorithms that it can implement.

7.3 PAC on more complex PA programs

This section gives more complex examples of PA programs, with their PAC encodings. These programs use interleaved wavefronts[9] and they motivate the PAC extensions of Section 7.4.

Here, the concept of wavefront *interleaving* is introduced. In some cases, these programs can be easily expressed in PAC without using (explicit) wavefront interleaving concepts; in other cases, this is impractical. This concept is also useful for providing 'separation of concerns' in systolic algorithms, and hence may be a useful tool for the derivation of such programs. These examples also require extensions to be made to PACl so that they can be loaded into the PA.

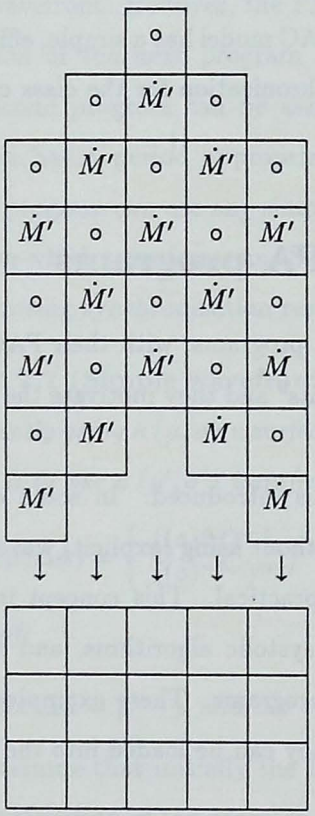
7.3.1 Matrix multiplication

The MatMult program of Section 2.2.2.2 [57, pp189-192] for an $m \times (2m-1)$ PA can be easily coded in PAC. The PAC program is shown diagramatically in Figure 7.5. It can be thought of a $(1,1)$ wavefront of M' instructions (passing input data east) combined with a $(1,-1)$ wavefront of $\dot{M}$ instructions (passing input data west). These wavefronts combine to give the $\dot{M}'$ instructions (passing

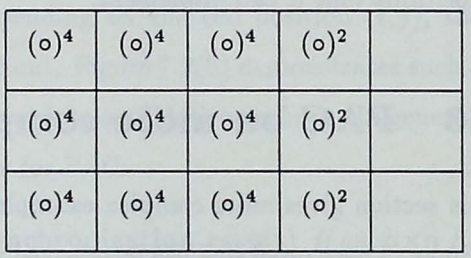
[9]Hence, they cannot be implemented as efficiently (or at all) on a microprogrammed ISA.

input data both ways), as is shown in Figure 7.5(a). This distinction between the different types of M instructions is only conceptual, to demonstrate that in some cases wavefront interleaving in PAC can be performed very easily. The MatMult program is coded in PAC as:

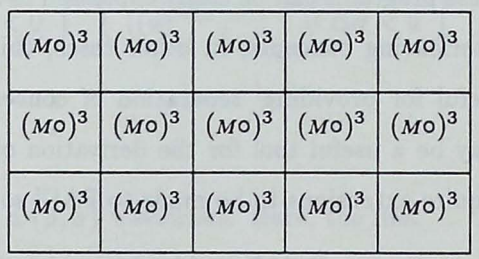
$$\text{MatMult} : [(o\ o)^m\ {}_m|\ (o\ o)^{2m-j}\ {}_{2m-1}]\ (M\ o)^m$$



(a) ISA program ($M = M' = \dot{M} = \dot{M}'$)



(b) delay part, from ISA wavefront



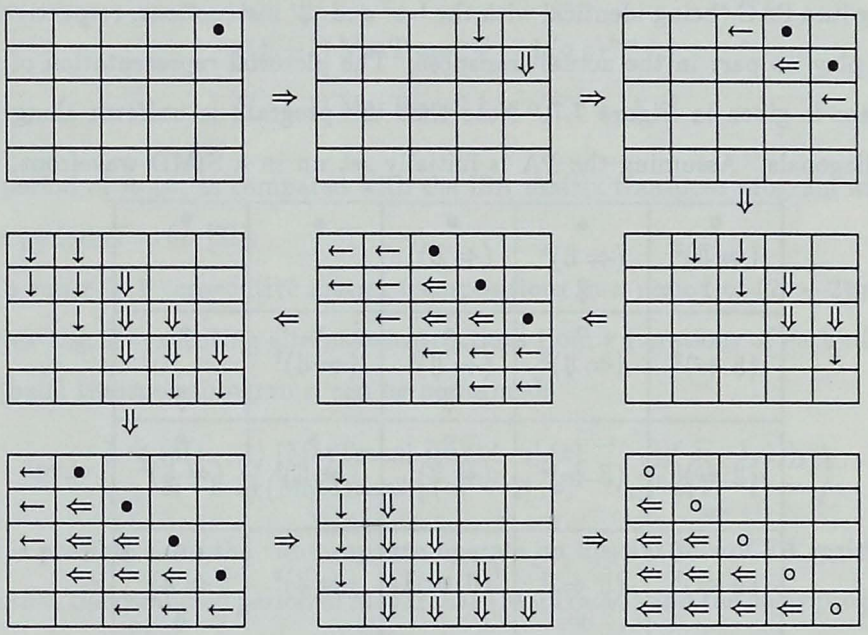
(c) computation part

Figure 7.5: MatMult program for a 3×5 PA

7.3.2 Matrix transpose program

The systolic matrix transpose algorithm of Ullman [57, pp199-201] can be implemented on an $n \times n$ PA with only orthogonal connections. The cell PCUs enable

the simulation of variable speed systolic control bits propagating through the PA. The program to perform the transpose of the upper half of a matrix, called program MatTrans1, is indicated by Figure 7.6, which gives the instructions executed at each (global) time instant.



('$\leftarrow$' and '$\Leftarrow$' denote '$C'_S \leftarrow C_E$'; '$\downarrow$' and '$\Downarrow$' denote '$C'_E \leftarrow C_N$'; '$\bullet$' denotes '$C'_S \leftarrow A$' and 'o' denotes a No-operation)

Figure 7.6: Execution of MatTrans1 program on a 5×5 PA, time steps 1-9

The program assumes that initially an $n \times n$ matrix A resides in the respective A registers of the PA. The program can be visualized as $(n-1)$ south-west moving $(1,-1)$ 'wavefronts'[10]. Consider the kth wavefront, $1 \leq k \leq n-1$. This becomes active at time step $2k-1$ (on the PA's kth anti-diagonal, see Figure 7.6), when the wavefront consists of '$\bullet$' instructions. These instructions load $A_{i,j}$ into the C'_S register of cell (i,j), where $j = n+i-k$. On time steps $2(k+t')$, where $t' \geq 0$, the wavefront consists of '$\Downarrow$' instructions which shift $A_{i,j}$ into the C'_W register of cell $(i+t'+1, j-t')$, where $j = n+i-k$. On time steps $2(k+t')+1$, where

[10]These wavefronts are unusual, since they are interpreted differently at different anti-diagonals.

$t' \geq 0$, the wavefront consists of '$\Leftarrow$' instructions which shift $A_{i,j}$ into the C'_S register of cell $(i+t'+1, j-t'-1)$, where $j = n+i-k$. The program stops at $t' = n-k-1$, with $A_{i,n+i-k}$ residing in the C'_S register of cell $(i+n-k, j-n+k)$, where $j = n+i-k$, ie. in cell $(n+i-k, i)$.

Note that the '$\rightarrow$' and '$\downarrow$' instructions of Figure 7.6 exist only to simplify the coding into PAC (being identical with the '$\Rightarrow$' and '$\Downarrow$' instructions, respectively). They play no part in the actual transpose. The pictorial representation of the program is given in Figure 7.7. Note that the program is uniform along the anti-diagonals. Assuming the PA is initially set up in a SIMD wavefront, the

$\circ$ $(\Leftarrow \Downarrow)^2$	$\bullet$ $(\Leftarrow \Downarrow)^1$ $\Downarrow$	$\bullet$ $(\Leftarrow \Downarrow)^1$	$\bullet$ $\Downarrow$	$\bullet$
$(\Downarrow \Leftarrow)^2$	$\circ$ $(\Leftarrow \Downarrow)^2$	$\bullet$ $(\Leftarrow \Downarrow)^1$ $\Downarrow$	$\bullet$ $(\Leftarrow \Downarrow)^1$	$\bullet$ $\Downarrow$
$(\Downarrow \Leftarrow)^1$ $\Leftarrow$	$(\Downarrow \Leftarrow)^2$	$\circ$ $(\Leftarrow \Downarrow)^2$	$\bullet$ $(\Leftarrow \Downarrow)^1$ $\Downarrow$	$\bullet$ $(\Leftarrow \Downarrow)^1$
$(\Downarrow \Leftarrow)^1$	$(\Downarrow \Leftarrow)^1$ $\Leftarrow$	$(\Downarrow \Leftarrow)^2$	$\circ$ $(\Leftarrow \Downarrow)^2$	$\bullet$ $(\Leftarrow \Downarrow)^1$ $\Downarrow$
$\Leftarrow$	$(\Downarrow \Leftarrow)^1$	$(\Downarrow \Leftarrow)^1$ $\Leftarrow$	$(\Downarrow \Leftarrow)^2$	$\circ$ $(\Leftarrow \Downarrow)^2$

Figure 7.7: MatTrans1 program for a 5×5 PA

delay program to set up program MatTrans1 is $\delta(1,-1) = (\circ)^{i-1}(\circ)^{n-j}$, and the computation part is given by considering the lower, diagonal, and upper parts of the PA separately:

$$\begin{aligned} \text{MatTrans1} \;:\; & [\,(\Leftarrow)^{(n+j-i) \bmod 2}(\Downarrow \Leftarrow)^{(n+j-i) \operatorname{div} 2}{}_n] \\ & (\Leftarrow \Downarrow)^{(n-1) \operatorname{div} 2} \circ {}_i| \\ & (\Downarrow)^{(n-1+i-j) \bmod 2}(\Leftarrow \Downarrow)^{(n-1+i-j) \operatorname{div} 2} \bullet {}_n] \end{aligned}$$

Note that the period of MatTrans1 in cell (i,j) is $n - |i-j|$. The delay program to restore the PA to a SIMD wavefront is $[\,{}_i|\,(\circ\,\circ)^{j-i}\,{}_n]$.

Note that PACl has to be extended to accommodate expressions such as '$(n-1+i-j)\,\mathrm{div}\,2$'; this will be dealt with in Section 7.4.2. The program to transpose the lower half of the matrix, which shall be called MatTrans2, is defined similarly and uses a $(-1,1)$ wavefront. These programs can be executed serially to perform a complete transpose, ie.:

$$\delta(1,-1)\ \text{MatTrans1}\ [\quad {}_{i}|\ (\circ\ \circ)^{j-i}\ {}_{n}]$$
$$\delta(-1,1)\ \text{MatTrans2}\ [\ (\circ\ \circ)^{i-j}\ {}_{i-1}|\ {}_{n}]$$

in a period of $\approx 4n$, as compared with the ISA matrix transpose program which has a period of $\approx 6n$ [31].

To run $r \geq 1$ consecutive matrix transpositions in a period of $(2r+2)n$, an interleaving, ie. executing alternate instructions from r iterations of each of the two (half) transpose programs, can be performed:

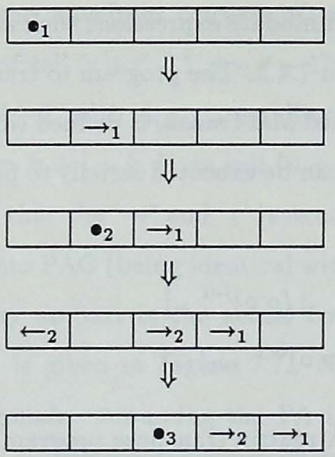
$$\text{int\_lv}\left(\begin{array}{l} <(\delta(1,-1)\ (\text{MatTrans1}\ [(\circ)^{i-j}\ {}_{i}|\ (\circ)^{j-i}\ {}_{n}])^{r}\ \delta(-1,1)), \\ \quad \delta(-1,1)\ (\text{MatTrans2}\ [\ (\circ)^{i-j}\ {}_{i}|\ (\circ)^{j-i}\ {}_{n}])^{r}\ \delta(1,-1)> \end{array}\right)$$

This is possible since the two programs operate on disjoint sets of PA registers. Note that the combined period of MatTrans1 (MatTrans2) and the delay program '$[(\circ)^{i-j}\ {}_{i}|\ (\circ)^{j-i}\ {}_{n}]$' is n, hence maintaining the wavefronts set up by $\delta(1,-1)$ $(\delta(-1,1))$.

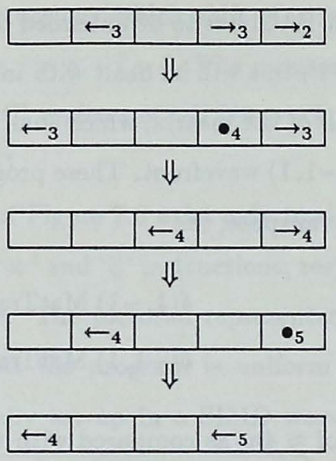
7.3.3 Warshall's transitive closure algorithm in PAC

Warshall's Transitive Closure algorithm can be implemented simply and efficiently on an $n \times n$ ISA using ring-shifting [32], which can be converted very easily into an equivalent PAC program. Here, it can be indicated how to implement this on a $n \times n$ PA without performing any ring-shifts. The purpose of this is to illustrate how a PA program whose 'wavefronts' have a time-dependent 'source' can be implemented in PAC (providing they do not collide).

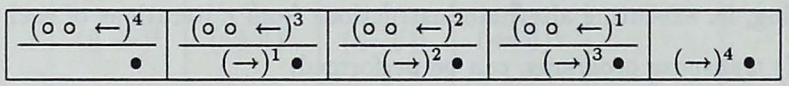
The structure of the PA program consists of three phases for each of the n iterations: a horizontal communication, a vertical communication and a local computation phase. Correspondingly, it is convenient to use a horizontal communication register C_H (read by west and east neighbours), a vertical communication



(a) time steps 1-5



(b) time steps 6-10



(c) PCUs of rows (for initially $\sigma = 1$)

('•' denotes '$C_H \leftarrow A$', '→' denotes '$C_H \leftarrow C_W$' and '←' denotes '$C_H \leftarrow C_E$'. Subscripts denote the source columns of the 'wavefronts')

Figure 7.8: Horizontal communication phase of Transitive Closure program on one row of a 5×5 PA

register C_V (read by north and south neighbours) and a local register A. The horizontal communication for a row of the PA implementation of this algorithm is illustrated in Figure 7.8.

For the kth repetition of the horizontal communication phase, column k sends an eastward moving ($\sigma = 1$) wavefront of '→' instructions and a westward moving ($\sigma = -1$) wavefront of '←' instructions from column k. Since these wavefronts never collide, it is possible to express their combination by inserting 'no-operations' (two 'o' instructions), as is shown in Figure 7.8(c). However, this program requires that initially $\sigma = 1$; because the period in column j is $3n - 2j$, the program then leaves the PA with a skew of $\sigma = -1$. The PAC encoding is, for the n repetitions of the horizontal phases:

$$\text{HorBroadCast}: \quad (\rightarrow)^{j-1} \bullet (\text{o o} \leftarrow)^{n-j}$$

and similarly, for the n repetitions of the vertical phases:

$$\text{VertBroadCast}: \quad (\downarrow)^{i-1} \bullet' (\circ\circ \uparrow)^{n-i}$$

with '$\downarrow$' denoting '$C_V \leftarrow C_N$' '$\uparrow$' denoting '$C_V \leftarrow C_S$' and '$\bullet'$' denoting '$C_V \leftarrow A$'. The n repetitions of the computation phase are coded simply as:

$$(C)^n$$

where 'C' denotes '$A \leftarrow A + C_H C_V$' (which probably would be implemented by a macro, ie. a sequence of smaller instructions). Defining '$\Uparrow$' ('$\Leftarrow$') to be the macros '$\circ\circ \uparrow$' ('$\circ\circ \leftarrow$'), the n repetitions of the whole program can be easily expressed as an interleaving of the repetitions of each of its phases:

$$\text{int\_lv}(< (\rightarrow)^{j-1} \bullet (\Leftarrow)^{n-j}, \quad (\downarrow)^{i-1} \bullet' (\Uparrow)^{n-i}, \quad (C)^n >)$$

The wavefront interleaving operation takes one macro from each of the programs of its argument list in turn (thus it is important that the macros are not expanded before the interleaving is performed). It is possible to express this program in PAC without using explicit wavefront interleaving (see Section 7.3.4 for a similar example). However, in this case, the explicit use of interleaving has resulted in the elegant expression of an otherwise complex communication pattern of the algorithm.

The ISA implementation of Warshall's transitive closure algorithm (which uses ring-shifting) [32], has a period of $10n$. Assuming a similar instruction set on a PA (except that two communication registers are used instead of one; this extra feature does not improve the ISA program's period), the 'C' macro requires only two instructions. Here, the above PA implementation, when appended with a $\delta(2,2)$ delay program to restore the $(1,1)$ wavefront, has a period of $8n$. Moreover, this implementation has a complementary version which requires the PA to be initially in a $(-1,-1)$ wavefront and leaves it in a $(1,1)$ wavefront. In this version, the activity begins in cell (n,n) and ends at cell $(1,1)$. The average period of running both versions consecutively is only $6n$.

7.3.4 Kung's transitive closure algorithm in PAC

On an $n \times n$ PA, a natural expression of the Kung-Gubias-Thompson transitive closure algorithm [57, pp192-198] for an $n \times n$ matrix A can be given using PAC. The microprogrammed ISA algorithm is given in Section 3.5.2. Recall from Section 2.2.2.4 that the algorithm consists of three passes; a single pass is described as follows. The initialization phase sets up the matrices $\mathrm{A}^0 = \mathbf{0}$ and $A'^0 = A''^0 = A + I$, where $\mathbf{0}$ is the zero matrix and I is the identity matrix. These are updated in cell (i, j) for the kth iteration, where $1 \leq k \leq n$, as follows:

$$\begin{aligned}
\mathrm{A}_{ij}^{k} &\leftarrow \mathrm{A}_{ij}^{k-1} + \mathrm{A}_{ik}^{\prime j-1}\mathrm{A}_{kj}^{\prime\prime i-1}; \\
\mathrm{A}_{ik}^{\prime j} &\leftarrow \begin{cases} \mathrm{A}_{ij}^{k} & \text{if } \mathrm{k} = \mathrm{i}; \\ \mathrm{A}_{ik}^{\prime j-1} & \text{otherwise} \end{cases} \\
\mathrm{A}_{kj}^{\prime\prime i} &\leftarrow \begin{cases} \mathrm{A}_{ij}^{k} & \text{if } \mathrm{k} = \mathrm{j}; \\ \mathrm{A}_{kj}^{\prime\prime i-1} & \text{otherwise} \end{cases}
\end{aligned}$$

Identifying A_{ij}^{k} with the A register of cell (i, j) on iteration k, and the nontransmittent variable $A_{ik}^{\prime j}$ ($A_{kj}^{\prime\prime i}$) with the C'_E (C'_S) register of cell (i, j) during the kth iteration, and the above operations with the macros:

$$\begin{aligned}
C &: \mathrm{A} \leftarrow \mathrm{A} + \mathrm{C_W C_N} \\
\Rightarrow' &: \mathrm{C}'_\mathrm{E} \leftarrow \mathrm{A} \qquad \Rightarrow : \mathrm{C}'_\mathrm{E} \leftarrow \mathrm{C_W} \\
\Downarrow'' &: \mathrm{C}'_\mathrm{S} \leftarrow \mathrm{A} \qquad \Downarrow : \mathrm{C}'_\mathrm{S} \leftarrow \mathrm{C_N}
\end{aligned}$$

the pass, using a $(5, 5)$ wavefront[11], is simply expressed as:

$$\mathrm{int\_lv}(< (C)^n,\ ((\Rightarrow)^{j-1}\ \Rightarrow'\ (\Rightarrow)^{n-j}),\ ((\Downarrow)^{i-1}\ \Downarrow''\ (\Downarrow)^{n-i}) >)$$

since at cell (i, j), only $\mathrm{A}_{ik}^{\prime j}$ ($\mathrm{A}_{kj}^{\prime\prime i}$) is updated with an '$\Rightarrow'$' ($\Downarrow''$) instruction, ie. the updating occurs only on the $k = i$th ($k = j$th) iteration.

The expression '$(\Rightarrow)^{j-1}\ \Rightarrow'\ (\Rightarrow)^{n-j}$', combined with $(C)^n$, has the same structure as the instruction part of the ISA version of this algorithm. It also has a similar function: to update the A and A′ matrices. The expression '$(\Downarrow)^{i-1}\ \Downarrow''\ (\Downarrow)^{n-i}$' has the same structure as the selector part of the ISA version: its function is to update A″. Note that on a PA, the updating of C'_S is done by a single (micro)instruction[12] at intervals of $\mu = \lambda = 5$ microcycles (ie. once

[11] This assumes that the C macro has length 3.

[12] It requires two micros, with the use of selector bits, on a microprogrammed ISA.

every iteration). The microprogramming timing rules of Section 4.4.2 ensures that upon the kth iteration, when cell (i,j) accesses C_N, the C'_S register of cell $(i-1,j)$ still stores A''^{i-1}_{kj}.

Note the similarities between this expression and the program of Section 7.3.4. Thus, the use of explicit interleaving has lead to a very straightforward PAC implementation. It is also possible to eliminate the explicit interleaving at the loss of the clarity and the compactness of the PAC encoding:

$$\begin{aligned}
&\text{int_lv}(< (C)^n,\ ((\Rightarrow)^{j-1}\ \Rightarrow'\ (\Rightarrow)^{n-j}),\ ((\Downarrow)^{i-1}\ \Downarrow''\ (\Downarrow)^{n-i}) >) \\
&\quad = [\ (\mathcal{C})^{j-1}\mathcal{C}'(\mathcal{C})^{i-j-1}\mathcal{C}''(\mathcal{C})^{n-i}\ _{i-1}| \\
&\qquad (\mathcal{C})^{i-1}\mathcal{C}'''(\mathcal{C})^{n-i}\ _{i}| \\
&\qquad (\mathcal{C})^{i-1}\mathcal{C}''(\mathcal{C})^{j-i-1}\mathcal{C}''(\mathcal{C})^{n-j}\ _{n}] \\
&\qquad \text{where } \mathcal{C} = \text{`}C\ \Downarrow\ \Rightarrow\ \text{'},\ \mathcal{C}' = \text{`}C\ \Downarrow\ \Rightarrow'\ \text{'} \\
&\qquad \text{and } \mathcal{C}'' = \text{`}C\ \Downarrow''\ \Rightarrow\ \text{'},\ \mathcal{C}''' = \text{`}C\ \Downarrow''\ \Rightarrow'\ \text{'}
\end{aligned}$$

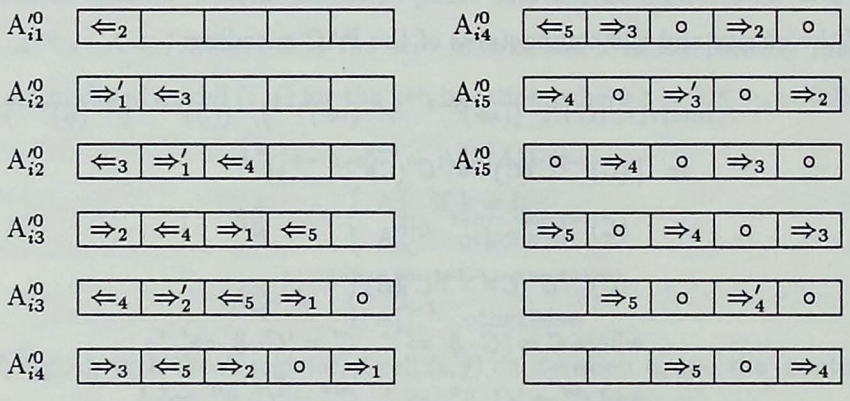
This example is already sufficiently complex to make it difficult to illustrate pictorially. In Appendix 7.B, the $\mathcal{C},\ldots,\mathcal{C}'''$ macros are rewritten to have length $\lambda = \mu = 4$ with the same instruction set as the $\lambda = 5$ microprogrammed ISA transitive closure program of Section 3.5.2. This reduction in period is due to the fact that on a PA, updating C'_S in cell (i,j) can be done here in one (micro) instruction only, whereas the ISA requires two.

7.3.5 Simulating skewed matrix input on a PA

Efficient implementation of the transitive closure algorithm of Section 7.3.4 requires the PA to have a two-dimensional torus topology. This is required to return the nontransmittent matrices A'' (A') to the left column (top row) of the PA for the second and third passes. For such a PA, it is efficient to assume that these matrices are initially resident in the PA [57, p207], ie. A''^0_{ij} (A'^0_{ij}) is initially in the C'_W (C'_N) register of cell (i,j), and then to simulate their (skewed) input into the PA.

For this topology, it is convenient to assume that the PA is buffered on its west (north) side by a column (row) of registers (or queues of length 1; cf. the

ISA data buffers proposed in Section 2.6), which read the contents of the adjacent C'_N (C'_W) registers on every PA (micro)instruction cycle. Figure 7.9 illustrates the respective time snapshots for simulating the input of row i of the matrix A'^0. These are combined with respective instructions for updating row i of A for the transitive closure program of Section 7.3.4.



(a) time steps 1-6 (b) time steps 6-10

('$\Leftarrow$' denotes '$C'_W \leftarrow C_E$', with '$\Rightarrow'$' and '$\Rightarrow$' as defined in Section 7.3.4. Subscripts denote the column index of A'_i. The contents of row i of the PA's west data buffer are annotated to the left)

Figure 7.9: Simulating input of A'_i for the Transitive Closure program on row i of a 5×5 PA

The program for row i of the PA uses a $(0,1)$ wavefront and is expressed in PAC as the interleaving of the westward and eastward communicating wavefront sub-programs:

$$\begin{aligned} \text{SkewMatInput} \quad &: \quad \text{int\_lv}(< (\Leftarrow)^{n-j}(\text{o})^j, \ (\Rightarrow)^{j-1} \Rightarrow' (\Rightarrow)^{n-j} >) \\ &= \quad [\,(\Leftarrow \Rightarrow)^{j-1} \Leftarrow \Rightarrow' (\Leftarrow \Rightarrow)^{n-2j}(\text{o} \ \Rightarrow)^j \ _{n/2}| \\ &\qquad (\Leftarrow \Rightarrow)^{n-j}(\text{o} \ \Rightarrow)^{2j-1}\text{o} \ \Rightarrow' (\text{o} \ \Rightarrow)^{n-j} \ _n] \end{aligned}$$

Note that expressions of the form $(\ldots)^{n-2j}$ and $(\ldots)^{2j-1}$ require extensions to the implementation of PACl. The corresponding program for A'^0 is similar, and uses a $(1,0)$ wavefront. The interleaving of these programs (with also $(C)^n$) implements

the first pass of the Kung-Gubias-Thompson transitive closure algorithm with simulated input of the matrices A′ and A″. However, to express the program for this pass without explicit interleaving is extremely cumbersome, and hence is inefficient for loading and storing in the PA. To efficiently implement this program on a PA, the hardware implementation of wavefront interleaving is necessary.

7.4 Extensions to PAC

Motivated by the examples of Section 7.3, this section gives some simple and relatively inexpensive extensions to PAC to increase its power and flexibility. The most important of these is called *wavefront interleaving*, which allows among other things a PA using PAC to efficiently simulate an ISA. Wavefront interleaving borrows many concepts of microprogramming (see Chapter 4), and correspondingly has two forms: *vertical* interleaving, with the unit of interleaving being an instruction macro (as illustrated in Section 7.3); and *horizontal* interleaving, with the unit of interleaving being a parallel operation within an instruction. Horizontal interleaving can be used to generalize the ISA masking mechanism, and hence be used for easy and efficient (microprogrammed) ISA simulation. Vertical interleaving can be used to simulate horizontal interleaving.

7.4.1 Implementation of wavefront interleaving

Section 7.3 gave examples of PA programs which were conveniently expressed as a vertical interleaving of two or more wavefront programs. Sometimes, this effect could be implicitly encoded in PAC; at other times, the PAC encoding without explicit interleaving became large, complex and inefficient to implement. This section formally defines vertical interleaving and indicates how it may be efficiently implemented on a $O(\log n)$ area cell PCU. It should be noted that the need to implement wavefront interleaving is not due to any limitations of PAC, but is due to the requirement for efficient loading of the $O(\log n)$ area cell PCUs. Hence, any PA programming language and the PA hardware itself should support wavefront interleaving if algorithms such as those in Section 7.3 need to

be implemented.

Consider the PAC programs $P_1, \ldots, P_l$, which are coded using fixed length macros of lengths $\lambda_1, \ldots, \lambda_l$ respectively. The *vertical interleaving* (executing one macro from each in turn) of these programs, being coded using macros of length:

$$\lambda = \lambda_1 + \ldots + \lambda_l$$

is defined in terms of its component program at the arbitrary $m \times n$ PA cell (i_0, j_0):

$$\begin{aligned} \mathrm{C}_{j_0,n}(\mathrm{R}_{i_0,m}(\mathrm{int\_lv}(< P_1, \ldots, P_l >))) &= \mathrm{i\_l}(< \mathrm{C}_{j_0,n}(\mathrm{R}_{i_0,m}(P_1)), \ldots, \\ &\quad < \mathrm{C}_{j_0,n}(\mathrm{R}_{i_0,m}(P_l)) >) \end{aligned} \tag{7.4}$$

With ϵ denoting an empty program, 'ins' denoting a PA instruction macro, $k \geq 1$ denoting an integer, and $P'_1, \ldots, P'_l$ denoting component programs, the component of a vertical interleaving is defined as:

$$\begin{aligned} \mathrm{i\_l}(< \epsilon, \epsilon, \ldots, \epsilon >) &= \epsilon \\ \mathrm{i\_l}(< (P)^0 P'_1, P'_2, \ldots, P'_l >) &= \mathrm{i\_l}(< P'_1, P'_2, \ldots, P'_l >) \\ \mathrm{i\_l}(< (P)^k P'_1, P'_2, \ldots, P'_l >) &= \mathrm{i\_l}(< P(P)^{k-1} P'_1, P'_2, \ldots, P'_l >) \\ \mathrm{i\_l}(< \mathrm{ins}\ P'_1, P'_2, \ldots, P'_l >) &= \mathrm{ins}\ \mathrm{i\_l}(< P'_2, \ldots, P'_l, P'_1 >) \end{aligned} \tag{7.5}$$

This definition implicitly assumes that all programs have the same period. The overall wavefront parameters of an interleaved program are generally chosen *a priori*; if the sub-programs themselves use different wavefronts, then the appropriate delay programs should be incorporated within them.

Example. For column 3 of an 8×8 PA, the vertical interleaving of the simulated matrix input program with the horizontal component of the transitive closure program of Section 7.3.5 is given by:

$$\begin{aligned} &\mathrm{i\_l}(< (\Leftarrow)^5(\circ)^3,\ (\Rightarrow)^2 \Rightarrow' (\Rightarrow)^5 >) \\ &= \mathrm{i\_l}(<\Leftarrow (\Leftarrow)^4(\circ)^3,\ (\Rightarrow)^2 \Rightarrow' (\Rightarrow)^5 >) \\ &= \Leftarrow \mathrm{i\_l}(<\Rightarrow (\Rightarrow)^1 \Rightarrow' (\Rightarrow)^5,\ (\Leftarrow)^4(\circ)^3 >) \\ &= \Leftarrow \Rightarrow \mathrm{i\_l}(< (\Leftarrow)^4(\circ)^3,\ (\Rightarrow)^1 \Rightarrow' (\Rightarrow)^5,\ >) \end{aligned}$$

$$
\begin{aligned}
&= (\Leftarrow \Rightarrow)^2 \ \mathrm{i\_l}(< (\Leftarrow)^3 (\mathrm{o})^3,\ \Rightarrow' (\Rightarrow)^5,\ >) \\
&= (\Leftarrow \Rightarrow)^2 \ \Leftarrow \Rightarrow' \ \mathrm{i\_l}(< (\Leftarrow)^2 (\mathrm{o})^3,\ (\Rightarrow)^5 >) \\
&= (\Leftarrow \Rightarrow)^2 \ \Leftarrow \Rightarrow' \ (\Leftarrow \Rightarrow)^2 \ \mathrm{i\_l}(< (\mathrm{o})^3,\ (\Rightarrow)^3, >) \\
&= (\Leftarrow \Rightarrow)^2 \ \Leftarrow \Rightarrow' \ (\Leftarrow \Rightarrow)^2 \ (\mathrm{o} \ \Rightarrow)^3
\end{aligned}
$$

While it is possible to express vertical interleaving in terms of existing PAC constructs (cf. Section 7.3.4), a general formula to do so would be complicated and yield surprisingly large PAC programs. Instead, hardware support for wavefront interleaving is proposed; this reduces the size of the cell PCUs and enables more efficient program loading.

The control structures required to implement vertical interleaving in the cell PCUs are surprisingly modest. Recall that the PCU has a table T to store programs, and its own working variables (eg. a table pointer, loop counter and table pointer stacks, and boolean flags). To interleave l sub-programs (eg. $l = 2, 4$), each program is loaded into a separate partition of T. l versions of the PCU working variables are then required, which are selected one at a time by a new PCU variable L. Upon beginning the execution of the interleaved program, L is set to 1. After executing a macro form the first sub-program, L is set to 2, and this continues, with the operation $L \leftarrow (L + 1) \bmod_1 l$ being performed after a macro in the current sub-program is executed. This process continues until $L = l$ and the lth sub-program has terminated, at which time the whole interleaved program is assumed to be terminated.

The macros of the interleaved programs may be implemented as macros, and be decoded in a similar fashion to the macros for the microprogrammed ISA (see Section 3.5.1). This may reduce cell PCU size in some applications. Alternatively, the sub-programs could be stored in T on the micro level, with each PCU (microinstruction) entry being augmented by a "end macro" marker. After an entry with this marker set is accessed, the execution of the next sub-program begins. This scheme has the advantage that the macros may have variable lengths.

It is also useful to use *horizontal interleaving* which combines the parts of an instruction

eg. the instruction proper and a selector bit (to determine whether

the instruction is actually performed, as in the ISA)

into a whole instruction. This is particularly useful in reducing code size if the PA's cells use long instruction words composed of several overlapped operations (as for the CMU Warp processor, which uses *software pipelining* [27, ch. 5]). As for its microprogramming counterpart, *horizontal interleaving* is more expensive to implement, since here interleaving l sub-programs would require l replications of the PCU control logic, as well l individually accessible PCU tables. However, horizontal interleaving is still useful, and two examples to illustrate this are given below.

Horizontal interleaving can in general be simulated by vertical interleaving, but the proof of this is beyond the scope of this chapter. Here, care must be taken when an interleaved sub-program uses a variable updated by another on the same macrocycle — in such cases, copying of such variables before their update is required. The simulation of the horizontal interleaving of l sub-programs will multiply the required wavefront parameters μ and σ of the original program by a factor of l.

This form of simulation is illustrated by a PA using two-way vertical interleaving simulating a (μ, σ) wavefront microprogrammed ISA (see Section 7.4.1.1). For each microprogrammed ISA diagonal, each PA cell uses one instruction to set a dummy register to the result of the diagonal's instruction, followed by a conditional store (on whether the diagonal's corresponding selector bit is set) of the dummy register into the destination of the instruction. This scheme doubles the program period, as well as the wavefront parameters μ and σ.

7.4.1.1 Simulation of an ISA using horizontal interleaving

It has been shown that an arbitrary PA program of period t can be expressed in PAC and, in $O(t)$ stages, this program can be loaded and executed in a PA with $O(\log n)$ area PCUs. Since a microprogrammed ISA program may be efficiently simulated by a PA (in the same fashion as an ISA program may be simulated by a PA [21]), a PA loaded by PAC can be said to simulate an arbitrary microprogrammed ISA. However this is unsatisfactory because performing many short

load-execute stages may result in large delays (see Section 7.5).

To offset the expense of the PCUs, it is necessary that the PAC-loaded PA be considerably more powerful than the microprogrammed ISA. In this chapter, many examples of PA algorithms have been given which cannot be implemented as efficiently (or at all) on a microprogrammed ISA. It then remains to show that 'in practice' any microprogrammed ISA program can be simulated efficiently using PAC. 'In practice' means that the microprogrammed ISA program can be coded in ISAC: this definition is justified by the high flexibility of ISAC (see Section 3.7).

This simulation uses the 'interleaving' of the top and left programs (TP, LP) of the ISA program, which can be assumed to be coded in ISAC. Their translation into PACl is straightforward, giving sub-programs TP' and LP', which are loaded into the PA. The horizontal interleaving of $< TP', LP' >$ performs the simulation, with TP' containing the instructions proper, and LP' containing the respective selector bits.

7.4.1.2 Matrix transpose program revisited

The matrix transpose program of Section 7.3 required the introduction of superfluous instructions to simplify the PAC encoding. These are denoted '$\downarrow$' and '$\leftarrow$' in Figure 7.6, and play no part in the actual transpose. By applying horizontal interleaving on:

$$\delta(1,-1)\ \text{MatTrans1}\ [\ {}_{i}|\ (\circ\circ)^{j-i}\ {}_{n}]$$

with the 'selector bit ' program '$(\mathbf{0})^{n+i-1-j}$ S1', where S1 is a yet to be determined program of period $n-i+j$, the superfluous '$\downarrow$' and '$\rightarrow$' instructions can be eliminated.

To determine S1, it should be noted that each PA cell (i,j) executing MatTrans1 executes a number n_{ij} of '$\downarrow$' and '$\rightarrow$' instructions first, and then executes the other instructions (which perform the actual transpose). By inspection of Figure 7.6, the n_{ij} coefficients may be determined, and they are given in Figure 7.10. From generalizing this figure, one can see that:

$$n_{ij} = \begin{cases} n+1-i-j & \text{if } 1 \leq j \leq n-i-1 \\ i+j-n-2 & \text{if } n-i-1 < j \leq n \end{cases}$$

ie. S1 must de-select the first n_{ij} instructions in cell (i,j) and select the $(n-i+j-n_{ij})$ remaining ones:

$$\text{S1}: \quad [\,(\mathbf{0})^{n+1-i-j}(\mathbf{1})^{2j-1}\ {}_{n-i-1}|\,(\mathbf{0})^{i+j-n-2}(\mathbf{1})^{2n+2-2j}\ {}_{n}]$$

4	3	2	1	0
3	2	1	0	0
2	1	0	0	1
1	0	0	1	2
0	0	1	2	3

Figure 7.10: Number of superfluous instructions/cell for program MatTrans1 on a 5×5 PA

7.4.2 Extensions to PACl

Sections 7.3.2 and 7.3.5 give PAC sub-programs of the forms $(\ldots)^{(n-i+j)\,\mathrm{div}\,2}$ and $(\ldots)^{n-2j}$, respectively. These can be translated into PACl by extending the current definition of C′ and op_j (see Section 7.2.3):

$$\mathrm{C}'_{j_0,n'}((P)^{(e+c(j))\,\mathrm{div}\,2}) = (\mathrm{C}'_{j_0,n'}(P))^{/_2\cdot\,\mathrm{op}_j(c,n').\,e+c(j_0)} \tag{7.6}$$

$$\mathrm{op}_j(c,n') = \begin{cases} \triangleleft_{2j} & \text{if } c(j) = 2j \bmod_1 2^k \text{ and } 2^k|n' \\ \triangleright_{2j} & \text{if } c(j) = -2j \bmod_1 2^k \text{ and } 2^k|n' \end{cases} \tag{7.7}$$

The operation '$/_2$' is implemented in the column generator array only: it instructs the respective cells to divide by 2 the current systolic counter, and can be efficiently implemented. The operation '$\triangleleft_{2j}$' ('$\triangleright_{2j}$') instructs the cell to perform the increment (decrement) of the current systolic counter from the second least [as opposed to the least] significant bit.

Depending on the range of programs to be executed on the PA, other such extensions may be required. It is likely that efficient implementations can be found for these also.

7.4.3 Widening the domain of PAC

As stated in Section 7.1, PAC assumes that the PA cells have limited data-dependent operations. It is permissible that the PA cells can mask an instruc-

tion according to the values of its status registers, which may be set in a data-dependent way. To extend this, only a (conditional) goto instruction would be required: this would set the PCU table pointer to the lookup value of a symbolic address (provided the appropriate cell status register is set). A 'symbolic address' is a marker at a point annotated inside the PAC program: at load time, when a PA cell encounters this marker, the current PCU table loading address would be stored in some lookup table. At run time, the lookup table would translate symbolic addresses into absolute PCU table addresses[13].

Hence, adding data-dependent features into the PA causes only a small overhead in the implementation of PAC. However, it should be remembered that this would not be necessary for most systolic algorithms.

The other main limitation of PAC is that, for the sake of program compression, PAC disallows iterations of the form '$(P(k))^{k=1..c}$', where $P(k)$ is a PAC subprogram dependent on k.

eg. for

$$P(k) = \begin{cases} P_1\ P_2 & \text{if } k \leq c/2 \\ P_1'\ P_2 & \text{if } k > c/2 \end{cases}$$

'$(P(k))^{k=1..c}$' would be encoded in PAC as '$(P_1\ P_2)^{c/2}(P_1'\ P_2)^{c-c/2}$'. Such codings have already been used in the PA transitive closure programs of Section 7.3.

To reduce cell PCU area, these effects can be implemented using a PA register as a counter storing the current value of k, and regarding this iteration as a data-dependent program. This incurs an overhead in program execution time.

Alternatively, interleaving could be sometimes used to simulate this effect more efficiently:

eg. for

$$P(k) = \begin{cases} P_1\ P_2 & \text{if } k \leq c/2 \\ P_1'\ P_2 & \text{if } k > c/2 \end{cases}$$

then '$(P(k))^{k=1..c}$' can be encoded as: 'int_lv($< (P_1)^{c/2}(P_1')^{c-c/2},\ (P_2)^c >$)'.

[13]In general, the absolute address would be dependent on the cell position; thus it cannot be determined before load time.

7.5 Evaluation of PAC

This section analyses in further detail alternatives for the implementation of PAC (see Section 7.1.3). These alternatives amount to two design decisions: (i) whether, in a PA cell, to evaluate any cell position dependent expressions in a program at load-time or at run-time; and (ii) whether to overlap program loading and program execution in a PA. These analyses extend to any PA requiring efficient program loading. Conclusions are then made regarding a viable PA domain for PAC.

7.5.1 Load-time vs. run-time evaluation of cell position-dependent code

As mentioned in Section 7.1.2, a choice in PA design is to load a *cellprogram* and evaluate its position dependent features either at load-time, or at run-time. To perform these evaluations, the latter choice requires extra instructions which may affect the synchronization of the PA.

To perform this comparison, several assumptions about a PA performing the evaluation at run-time must be made. Firstly, the (i,j)th cell of an $m \times n$ PA is assumed to have registers (of the same name) containing the values i, j, m and n. The column selection construct[14]:

$$[P_1 \ _{n_1} | P_2 \ _{n_2} | \ \ldots \ | P_l \ _n]$$

is assumed to be implemented as:

if $j \not\leq n_1$ **then goto** L_2
  $(\mathrm{o})^{l-2}$
  <code for P_1 > &**goto** L
L_2 : **if** $j \not\leq n_2$ **then goto** L_3
  $(\mathrm{o})^{l-3}$
  <code for P_2 > &**goto** L

[14]The implementation becomes more complex if the recurrence width of the construct is less than n.

$\vdots$

L_l : <code for P_2 > &**goto** L

L : ...

Here, 'P&**goto** L' means the last instruction of P also performs an unconditional **goto** to the code at label L. If P is of the form $(P')^k$, then this **goto** is performed only on the kth iteration of P'. To maintain proper synchronization between cells executing different sub-programs within the selection construct, no-operations have to be inserted. A similar implementation applies for the row selection construct.

Assume that the cell position dependent repetition construct '$(P_1)^{f(i,j)}$' is implemented by instructions evaluating $f(i,j)$, and that these instructions have the syntax '$R := R$ op E' where R is a register (or the PCU counter stack top), E is either a register or a fixed-length constant and 'op' includes integer addition, subtraction, absolute value, division (by a power of 2) and modulus (of a power of 2). If $f(i,j) > 0$, an instruction places the value of $f(i,j)$ on the counter stack top of the cell PCU, and P_1 is executed $f(i,j)$ times. Otherwise, the same instruction performs a **goto** past the construct.

The comparison of the two methods for a cross-section of the PAC programs of this chapter is given in Table 7.1. The second column gives section references from this chapter where the respective PAC programs can be found. The program's size (the maximum number of entries in any cell's PCU tables) for the evaluate at load-time and at run-time methods is given in columns 4 and 5, respectively. A similar situation exists for the program period (the maximum period of the program in any PA cell) in columns 6 and 7.

It should be noted that the results of Table 7.1 favour the load-time method in that programs with relatively large dependence on the cell position are given in the table. On the other hand, the assumed implementation of the run-time method uses a very specialized instruction set chosen to optimize its performance.

The load-time method is superior in terms of saving cell PCU size, as demonstrated in Table 7.1, because of two factors:

- the code for the whole PA is stored in each cell PCU (accounting for the

PAC encoding	sect. ref.	code size		period	
		load	run	load	run
RedSquares : $([\leftarrow\ _{n-1}\|\mathbf{0}\ _n]\ \left[\begin{array}{c}\uparrow\ _{m-1}\\ \hline \mathbf{0}\ _m\end{array}\right])^{n-1}$	7.1.1	2	$4+2$	$2(n-1)$	$4(n-1)$
$\left[\begin{array}{l}[(\leftarrow\uparrow)^{n-1}\ _{n-1}\|(\leftarrow \mathbf{0})^{n-1}\ _n]\ _{m-1}\\ \hline [(\leftarrow \mathbf{0})^{n-1}\ _{n-1}\|(\mathbf{0}\ \mathbf{0})^{n-1}\ _n]\ _m\end{array}\right]$	*	2	$8+3$	$2(n-1)$	$2n+2$
ISAtoSIMD: $(\circ)^{n-j}(\circ)^{m-i}$	7.1.1	2	$2+4$	$m+n-2$	$m+n+2$
SymEigen (delay part): $[(\circ)^{i-j}\ _{i-1}\|\ _i\|(\circ)^{j-i}\ _n]$	7.1.1	1		$n-1$	
$(\circ)^{\|i-j\|}$	*		4		$n+2$
SymEigen (compute part): $[(L\circ\circ)^n\ _{i-1}\|(D\circ\circ)^n\ _i\|(U\circ\circ)^n\ _n]$	7.1.1◇	3λ	$9\lambda+3$	$3n\lambda$	$3n\lambda+2$
MatMult: $[(\circ\circ)^m\ _m\|(\circ\circ)^{2m-j}\ _n]\ (M\circ)^m$	7.3.1◇	4λ	$6\lambda+7$	$4\lambda m$	$4\lambda m+4$
MatTrans: $[(\Leftarrow)^{(n-i+j)\bmod 2}$ $(\Leftarrow\Downarrow)^{(n-i+j)\operatorname{div}2}\ _{i-1}\|$ $(\Leftarrow\Downarrow)^{n\operatorname{div}2}\ \circ\ _i\|$ $(\Downarrow)^{(n-1+i-j)\bmod 2}$ $(\Downarrow\Leftarrow)^{(n-1+i-j)\operatorname{div}2}\ \bullet\ _n]$	7.3.2†	4	$10+14$	n	$n+9$
HorBroadcast: $(\rightarrow)^{j-1}\ \bullet\ (\circ\circ\leftarrow)^{n-j}$	7.3.3	4	$4+4$	$3n-2$	$3n+2$
SkewMatInput: $[(\Leftarrow\Rightarrow)^{j-1}\Leftarrow\Rightarrow'$ $(\Leftarrow\Rightarrow)^{n-2j}(\circ\Rightarrow)^j\ _{n/2}\|$ $(\Leftarrow\Rightarrow)^{n-j}(\circ\Rightarrow)^{2j-1}$ $\circ\Rightarrow'(\circ\Rightarrow)^{n-j}\ _n]$	7.3.5†	8	$16+13$	$2n$	$2n+7$

notes:

* the program of the row above is here recoded to suit the run-time method's period.

◇ (large) macros of length $\lambda \geq 1$ are used here.

† some effort has been made to optimize the run-time method's codings. The run-time method's code size and period should be treated as approximate.

Table 7.1: Load-time vs. run-time PAC program evaluation

first term in column 5).

- the code to perform the evaluation must also be stored (accounting for the second term in column 5).

For the range of the programs given in Table 7.1, the combination of these factors requires the load-time method's PCU table size to be about a third of that of the run-time's method.

If the PA instruction set does not support powerful conditional **goto** operations, such as those required for the above implementation of the run-time method, the run-time method incurs a substantial overhead. On the other hand, if the load-time method cannot use the PA cell's ALUs to perform the loading, the load-time method requires extra integer arithmetic circuitry, which also might be a considerable overhead. This can be mitigated if the load-time method is used in conjunction with overlapped program load and execution (see the next section), in which case there is sufficient time for all arithmetic operations to be performed bit-serially.

Both methods have approximately the same performance in total load-execute time, unless the load-time method overlaps the loading of a new program with the execution of the current program, in which case it generally has a small advantage (as indicated by the difference between columns 6 and 7 of Table 7.1).

7.5.2 Program loading and execution: serial vs. overlapped

This section analyses the load-execute performance of PAs with respect to whether the program load and execution phases should be executed serially or be overlapped. The former is superior in area; the latter is superior in period. The analysis is then given with respect to the area-period (AP) measure. Assume that the program loading propagates according to a fixed, say a $(1, 1)$, wavefront. The overlapped load-execute method requires extra area in the cell PCU tables, to buffer $(b + 1) \geq 2$ programs at once; however, it has the advantage that the PA can be performing useful computations virtually 100% of the time. Also, the

serial load-execute method incurs a serious delay when the program execution wavefront differs from the program loading wavefront.

To compare the two methods on an $n \times n$ PA, only $\Omega(n)$ programs using (μ, σ) wavefronts will be considered. Our experience is that these programs can be assumed to have a period not less than[15]:

$$t_{\min} = \max\{|\mu|, |\sigma|, 1\}n$$

These programs are also made to be 'irreducible'[16] where an 'irreducible' program is a program that is not expressible as a sequence of two sub-programs, each of period no less than $t_{\min}$ and each requiring significantly less coding space than the program itself.

This is possible because any application to be run on the PA can broken down into a sequence of 'irreducible' programs, which may be loaded and executed separately. Let A_{tab} denote the area of a PCU table that is just large enough to store any such 'irreducible' program, ie. A_{tab} gives the area of the smallest PCU table that the PA can use.

First consider the PA executing a sequence of several programs, all of which use a (μ, σ) wavefront. The following analysis derives for this case a sufficient value of b and an AP comparison of the methods. This value of b can then be shown to be sufficient to cover the other cases.

Let the current program to be loaded on the PA be Q. Assume that the array size n is sufficiently large that the loading period for the program Q, $t_{\text{ld}} = O(\log n)$, satisfies $t_{\text{ld}} \leq t_{\min}$.

Serial program loading and execution, by definition, requires that the program loading wavefront for the program Q does not pass through any PA cell while it is still executing the current program. This delay is equivalent to executing the delay program $\delta(1-\mu, 1-\sigma)$. After Q has been loaded, a complementary

[15]This assumption can be justified as follows. Such programs in practice consist of at least n basic operations. A value of μ such that $\mu > |\sigma| \geq 0$ is only required if north-south communication occurs between the $(k+\mu)$th instruction at row i and the $(k+1)$th instruction at row $i+1$, for some k. Since this communication would form part of one of the n basic operations, at least μn instructions are required. Similar reasoning exists for $-\mu > |\sigma| > 0$ and $|\sigma| \geq |\mu| > 0$.

[16]cf. Section 3.3.

delay, equivalent to executing $\delta(\mu-1,\sigma-1)$, is required. This is because the execution wavefront for Q must not pass through any cell still loading Q. Thus, the combined delay is approximately $t_{\text{del}} = (|\mu-1|+|\sigma-1|)n$. The total period for the load-execute cycle for Q is:

$$P = t_{\text{ld}} + t_{\text{del}} + t_Q$$

Let the total area of a PA cell operating in this way be A; the cell uses a 'minimal' PCU table of area A_{tab}.

Since it is assumed that $t_{\text{ld}} \leq t_{\text{min}}$, a PA using fully overlapped program loading and execution performs the load-execute cycle of Q in period:

$$P' = t_Q$$

For this method, since the PCU table must be able to hold $(b+1) \geq 2$ programs simultaneously (and assuming that the extra control structures required by the overlapped method is negligible), the cell area is $A' = A + bA_{\text{tab}}$.

Overlapped program load-execution is more AP efficient (ie. $A'P' \leq AP$) provided that:

$$\begin{aligned}
\frac{A'-A}{A} &\leq \frac{P-P'}{P} \\
\frac{bA_{\text{tab}}}{A} &\leq \frac{t_{\text{del}}+t_{\text{ld}}}{t_Q} \\
\frac{A_{\text{tab}}}{A} &\leq \frac{f_Q}{b}\frac{|\mu-1|+|\sigma-1|}{\max\{|\mu|,|\sigma|,1\}} + \frac{t_{\text{ld}}}{bt_Q}
\end{aligned} \tag{7.8}$$

where $f_Q = t_{\text{min}}/t_Q$, ie. $f_Q \leq 1$, and the calculation of Section 7.5.2.1 gives a conservative value of $b = 4$. However, in practice, provided occasional delays due to program loading can be tolerated, smaller values of b may be permissible. Here, the overlapped program load-execution is always superior on coarse-grained PAs, where $A_{\text{tab}} << A$. It is also superior on finer-grained PAs if on average either of the following conditions hold:

1. when f_Q is relatively close to unity, ie. P has a period not many times the minimum period t_{min} and a $(1,1)$ wavefront is not often used (in this case, $(|\mu-1|+|\sigma-1|)/(\max\{|\mu|,|\sigma|,1\}) \geq 1$, and is on average 2).

2. the array size n is sufficiently small so that t_{ld}, where $t_{\text{ld}} \leq t_{\text{min}}$, is still of comparable size to t_Q (this situation can exist for current commercial PAs — see Section 7.1.2).

Now, consider the case of executing the program sequence '$Q; Q'$', where program Q [Q'] uses a (μ, σ) wavefront [a (μ', σ') wavefront]. By Result 7.1, the delay program $\delta(\mu' - \mu, \sigma' - \sigma)$ must be effectively executed between the executions of Q and Q'. Assume that (for the overlapped method) this delay program is loaded with Q, and that the value of A_{tab} has been adjusted (only very slightly) to take this into account. A similar analysis reveals that the overlapped load-execution method does not require a larger value of value of b than it does for the previous case. However, the serial load-execute method must effectively execute the generally longer delay program:

$$\delta(\mu - 1, \sigma - 1); \delta(1 - \mu', 1 - \sigma')$$

This analysis does not consider programs using partitioned or interleaved wavefronts. Also, the serial load-execute method might be more AP efficient if it could similarly buffer $(b' + 1) \geq 2$ programs in its PCU[17]. Loading $(b' + 1)$ programs consecutively reduces the loading delays of this method by a factor of approximately b'. However, this analysis shows that b' must be less than b if the serial load-execute method is to be more AP efficient.

7.5.2.1 Calculation of the PCU table buffering factor

The buffering factor b of the PCU table for the overlapped load-execute method is calculated as follows.

If the PA begins executing (in one of its corner cells) the program Q at time $t = 0$, cell (i, j) begins executing P at time:

$$t_{\text{ex}}(i, j) = |\mu|(\text{sgn}_\mu(i-1) + (1 - \text{sgn}_\mu)(n - i)) + |\sigma|(\text{sgn}_\sigma(j-1) + (1 - \text{sgn}_\sigma)(n - j))$$

where $\text{sgn}_x = 1$ if $x > 0$ and $\text{sgn}_x = 0$ otherwise. The loading of Q, performed in period t_{ld}, must be completed before the execution of Q begins in any cell. The

[17]This is used in commercial PAs — see for example [24, p3], which indicates b' is of the order of tens for the Warp processor.

loading must then begin in cell (i,j) at time:

$$t_{\text{ld}}(i,j) = -t_{\text{ld}} - (\text{sgn}_{\mu} + \text{sgn}_{\sigma})n + i - 1 + j - 1$$

The maximum difference between these times is given by:

$$\begin{aligned} & \max\{i,j \mid t_{\text{ex}}(i,j) - t_{\text{ld}}(i,j)\} \\ &= t_{\text{ld}} + |\mu - 1|(n-1) + \text{sgn}_{-\mu} + |\sigma - 1|(n-1) + \text{sgn}_{-\sigma} \\ &\leq t_{\text{ld}} + |\mu - 1|n + |\sigma - 1|n \end{aligned}$$

noting that, for $\mu > 0$ ($\mu \leq 0$), the maximum difference occurs at row $i = m$ (row $i = 1$), and similarly for σ.

During this time, there must be sufficient space in the PCU tables to store the program being executed and the b programs awaiting execution or being loaded. To cover the worst case, it must be assumed that the program being executed and the programs awaiting execution are of the minimal period t_{min}, so that:

$$\begin{aligned} b &= \lceil \frac{t_{\text{ld}} + |\mu - 1|n + |\sigma - 1|n}{\max\{|\mu|, |\sigma|, 1\}n} \rceil \\ &\leq 1 + \lceil \frac{|\mu - 1| + |\sigma - 1|}{\max\{|\mu|, |\sigma|, 1\}} \rceil \end{aligned}$$

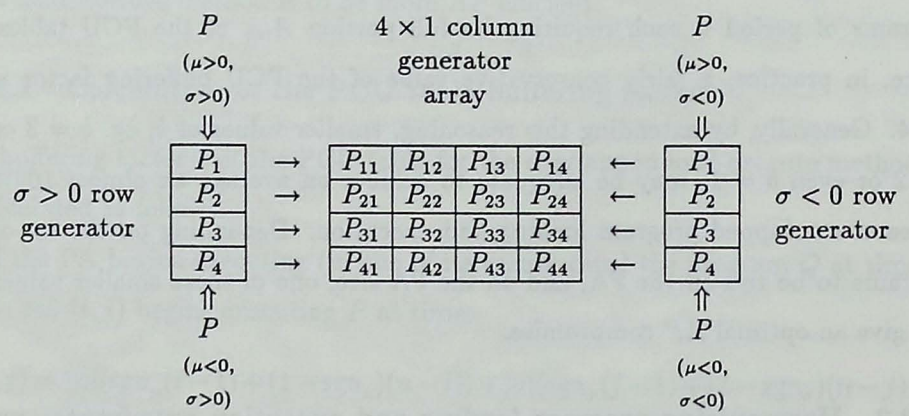
The second term is bounded above by 3 in all cases except $\mu = \sigma = -1$. For $\mu = \sigma = -1$, the second term is 4; however, in practice, b need not be calculated to cover this case, since $t_{\text{min}} = n$ and it is very unlikely to have four consecutive programs of period n each requiring its full portion A_{tab} of the PCU tables. Hence, in practice, a fairly conservative value of the PCU buffering factor is $b = 4$. Generally, by extending this reasoning, smaller values of b, eg. $b = 3$ or $b = 2$ or even $b = 1$, may be adequate to achieve on average an almost 100% degree of overlapped program loading and execution. Depending on the set of programs to be run on the PA, and on the PA size, one of these smaller values may give an optimal AP compromise.

7.5.2.2 Harmonizing program loading and execution wavefronts: an efficient concept for program loading

The main drawback with the area efficient serial program load-execute method is that it causes delays because of the mismatch between the program loading and

execution wavefronts. In combining the variable-wavefront microprogrammed ISA with ISADL (see Section 5.4.4), this problem was avoided by inserting queues in the ISADL diagonal restorer, so that no mismatch occurred. In this case, architectural considerations were simplified by the fact that $\mu, \sigma > 0$. This idea can still be extended to PAC, but the generality of wavefronts required for PAC introduces some architectural complexities. However, in most cases, it combines the program loading efficiency of the overlapped program load-execute method, with much of the area efficiency of the serial load-execute method.

Figure 7.11 gives an overview of a program interface which harmonizes the program loading wavefronts with the program execution wavefronts, and uses the serial program load-execute method. A program using a (μ, σ) wavefront is loaded into the PA from one end of one the two row generators, depending on the signs of μ and σ (these must also be non-zero). Thus the row generators (rows of the column generator array) are bi-directional linear arrays. They are implemented exactly as in Section 7.2.2, except that their inputs are buffered by program loading queues of length $|\mu|$ ($|\sigma|$) and the direction of input is determined by the sign of μ (σ). Note that the column generator array is embedded into the PA (cf. Figure 7.4). The cost of this program interface is mainly in its added



(note: $P_{i_0} = \mathrm{R}_{i_0,4}(P\ (i/i_0))$; $P_{i_0 j_0} = \mathrm{C}_{j_0,4}(P_{i_0}\ (j/j_0))$, for $1 \leq i_0, j_0 \leq 4$))

Figure 7.11: Program interface harmonizing program loading and execution wavefronts on a 4×4 PA

complexity: it must be capable of loading programs from four different points (as compared with one for the PAC program interface of Figure 7.4). There may be a serious cost also in that the I/O across each row of the column generator array is now bi-directional: this may in turn double the PA cell pin count due to program loading. The cost of the variable length program loading queues required can be kept small (cf. Section 4.3.1).

This interface can also efficiently load programs with interleaved wavefronts, by performing the loading in an appropriately interleaved manner. However, this in turn requires every row generator cell (column generator array cell) to have an extra set of input pins and an extra loading queue for each interleaved sub-program having a different value of μ (σ). The PA program examples of Sections 7.3 and 7.2 indicate that only two different wavefronts are normally required for PA wavefront interleaving: thus this overhead may not be too serious. However, this interface cannot load without delay any program using $(0, \sigma)$ or $(\mu, 0)$ wavefronts, nor any program using different wavefronts over PA partitions, such as the SymEigen program of Section 7.1.1. Thus, the choice of a program interface for efficient PA loading is a tradeoff between flexibility and efficiency.

Note that this concept can also be applied to the overlapped program load-execute method, to reduce (in most cases) the buffering factor b to 1. Thus, provided the extra pins it requires is not critical, harmonizing the program loading and execution wavefronts is a useful concept in efficient PA program loading.

7.5.3 Conclusions regarding a viable domain for PAC

This chapter has proposed PAC as a low-level PA language designed for the efficient implementation of program compression for systolic (in a broad sense) programs expressible in terms of the extended wavefront programming model (to our knowledge, this includes all PA programs that are useful in practice). After translation into a lower-level form, the efficient loading of PAC programs into 'minimal' $O(\log n)$ area cell PCUs is supported by modest control structures. PAC also has scope for further extensions. PAC provides a means of efficient and general PA program synchronization. However PAC suffers from a weakness:

unlike ISADL, the wavefront is not an explicit feature of the language. Hence, PAC might not be a convenient basis for a higher-level PA language, nor be particularly amenable to verification techniques (see Chapter 6).

Although PAC has specific constructs, most of the principles involved with its implementation can be extended to any PA language designed for efficient program loading. For this purpose, the essential features are constructs for selecting ranges of rows and columns, (the fast loading of) cell-position dependent loop counters and the wavefront interleaving construct. The latter is related to ISA microprogramming. In the case of *vertical* interleaving, PA programming power is increased in return for a very modest control structure overhead. *Horizontal* interleaving may be used to reduce PA cell code size (and hence cell PCU size) for *software pipelined* programs [27, Ch.5] in PAs implementing two-level pipelining. Wavefront interleaving is proposed here as a useful feature to be implemented in advanced PA architectures.

A reasonably efficient method for program loading, used by some existing commercial PAs, is to express a PA program as a *cellprogram*, which is loaded to the PA cells and any cell-position dependent expressions are evaluated at run-time. The program loading and execution are typically performed serially, and the considerable delays due to program loading and startup, coupled with a large PA cell grain size, has led to the cell PCUs being large, capable of storing many (ie. the order of tens of) programs.

However, for larger-scale and/or finer-grained PAs, these program loading methods may prove inefficient. These PAs would implement subroutine-sized matrix algorithms, for which typically a cell PCU capable of storing 64 instructions would be adequate [10, p3][58, p299][24, p4]. For these PAs, it is area and period-efficient to use a PCU capable of holding at most a few such programs, and to employ efficient program loading techniques. Here, it has been shown that evaluating a program's cell-position dependent expressions at load-time can reduce cell PCU size by a typical factor of 3, at the expense of relatively modest contol structures (in the case of PAC, a fairly modest bitwise processor suffices). We have shown that for a fixed wavefront program loading direction, overlapping

program loading and execution is period-efficient, but results in an increase in the size of the PCU tables by a factor of $(b+1)$, where $2 \leq b \leq 4$. This is generally area-period efficient when the area required to store a single ('irreducible') subprogram in the cell PCU divided by the whole cell area is at most b^{-1}. However, at the expense of some generality, harmonizing the program load and execution wavefronts can improve the area-time efficiency of either method, provided pin count is not a limiting factor to the PA area. All these considerations are important in any form of efficient PA program loading.

PAC may be viably implemented on these PAs, with the suggested implementation evaluating cell-position dependent expressions dynamically at load-time, and overlapping program loading and execution. For the PCUs not to form a substantial fraction of the cell area, each cell would require a comparable area for local data storage and at least implement instructions supporting efficient bit-wise arithmetic operations. For large-scale $n \times n$ PAs, an $o(n)$ host-PA I/O bandwidth requires a considerable amount of local memory in each PA cell (or in each PA data buffer cell) to offset the mismatch between the PA's I/O and computational bandwidths [25]. This determines the minimum granularity of the PA suitable for PAC implementation (finer-grained PAs may be best implemented as microprogrammed ISAs). Thus, PAC is targeted for MIMD PAs of finer than microprocessor-sized granularity with reasonably simple instruction sets. These PAs would implement array algorithms as sequences of matrix subroutines, as required for applications such as vision processing, where hundreds of these may be routinely used [10, p1].

Future research on PAC includes developing *wavefront interleaving* as a programming concept, and investigating the incorporation of LSGP partitioning into PAC. Also, the application of the macro structure of Section 4.6 should be extended to PAs; a particular challenge here is for the case of wavefront interleaving.

In summary, this chapter has demonstrated that compression and microprogramming techniques can improve overall performance, area efficiency and flexibility of large-scale, relatively simple Processor Arrays.

7.A Appendix: Definition of PAC

This appendix gives a definition of a PAC program in terms on its projection onto arbitrary cells of a PA.

For integers $i', m', m_1, \ldots, m_l$ and k (where $1 \leq i' \leq m' \leq m$ and the *row selection boundaries*[18] satisfy the constraints $1 \leq m_1 \leq m_2 \leq \ldots \leq m_l$ and $m_l | m'$), the PA instruction 'ins' and the PAC programs $P_1, \ldots, P_l$, the PAC row projection function R is defined by the equations,

$$\begin{aligned}
\mathrm{R}_{i'm'}(P_1\ P_2) &= \mathrm{R}_{i'm'}(P_1)\ \mathrm{R}_{i'm'}(P_2) \\
\mathrm{R}_{i'm'}(\ \mathrm{ins}\ ) &= \mathrm{ins} \\
\mathrm{R}_{i'm'}((P_1)^k) &= (\mathrm{R}_{i'm'}(P_1))^k \\
\mathrm{R}_{i'm'}\left(\left[\begin{array}{c|c} P_1 & m_1 \\ \hline P_2 & m_2 \\ \hline \vdots & \\ \hline P_l & m_l \end{array}\right]\right) &= \begin{cases} \mathrm{C}_{i_1 m_l}(P_1) & \text{if} \quad 0 < i' \leq m_1 \\ \mathrm{C}_{i_2 m_l}(P_2) & \text{if} \quad m_1 < i' \leq m_2 \\ & \vdots \\ \mathrm{C}_{i_l m_l}(P_l) & \text{if } m_{l-1} < i' \leq m_l \end{cases}
\end{aligned} \tag{7.9}$$

where $i_1 = i' \bmod_1 m_l$, $i_2 = (i' - m_1) \bmod_1 m_l$, ..., $i_l = (i' - m_{l-1}) \bmod_1 m_l$

where $x \bmod_1 y = (x-1) \bmod y + 1$. For PAC row projections $P'_1, \ldots, P'_l$ (ie. PAC programs containing no *row selections* and no occurrences of the column number i), and integers $j', n', n_1, \ldots, n_l$ and k (where $1 \leq j' \leq n' \leq n$ and the *column selection boundaries* satisfy the constraints $1 \leq n_1 \leq n_2 \leq \ldots \leq n_l$ and $n_l | n'$), the PAC column projection function C is given similarly by:

$$\begin{aligned}
\mathrm{C}_{j'n'}(P'_1\ P'_2) &= \mathrm{C}_{j'n'}(P'_1)\ \mathrm{C}_{j'n'}(P'_2) \\
\mathrm{C}_{j'n'}(\ \mathrm{ins}\ ) &= \mathrm{ins} \\
\mathrm{C}_{j'n'}((P'_1)^k) &= (\mathrm{C}_{j'n'}(P'_1))^k \\
\mathrm{C}_{j'n'}(\left[\begin{array}{c|c|c|c} P'_1 & n_1 & P'_2 & n_2 \end{array} \cdots \begin{array}{c|c} P'_l & n_l \end{array}\right]) &= \begin{cases} \mathrm{C}_{j_1 n_l}(P'_1) & \text{if} \quad 0 < j' \leq n_1 \\ \mathrm{C}_{j_2 n_l}(P'_2) & \text{if} \quad n_1 < j' \leq n_2 \\ & \vdots \\ \mathrm{C}_{(j_l n_l}(P_l) & \text{if } n_{l-1} \leq j' < n_l \end{cases}
\end{aligned} \tag{7.10}$$

[18] In P, these boundaries must be integers. Were they integer expressions dependent on j, the evaluation of $\mathrm{R}_{i_0,m}(P(i/i_0))$ could not proceed. However, the *column selection* boundaries can be integer expressions dependent on i, since these can be evaluated before $\mathrm{C}_{j_0,n}(\ldots)$ is evaluated.

where $j' \bmod_1 n_l$, $j_2=(j'-n_1) \bmod_1 n_l$, ..., $j_l=(j'-n_{l-1}) \bmod_1 n_l$

These equations are virtually a higher-level form of a two-dimensional extension to the definition of ISAC. Note that for *divide-and-conquer* programs, a *parameterized repetition* construct needs to be introduced to PAC as it was introduced to ISAC. For the sake of simplicity, this feature is omitted here.

Example 1: the SIMD wavefront RedSquares program at cell $(1,3)$ for a 3×3 PA is given by (cf. Figure 7.1(b)):

$$\begin{aligned} C_{33}(R_{13}(([\rightarrow\ _2|\mathbf{0}\ _3]\ \left[\begin{smallmatrix}\uparrow & 2\\ \mathbf{0} & 3\end{smallmatrix}\right])^2)) &= C_{33}(([\rightarrow\ _2|\mathbf{0}\ _3]\ R_{13}(\left[\begin{smallmatrix}\uparrow & 2\\ \mathbf{0} & 3\end{smallmatrix}\right]))^2) \\ &= (\mathbf{0}\ \uparrow)^2 \end{aligned}$$

Example 2: for the delay part of the SymEigen program on a 5×5 PA (cf. Figure 7.3(a)), the program loaded at cell $(4,2)$ is given by:

$$\begin{aligned} & C_{25}(R_{45}([(\circ)^{i-j}\ _{i-1}|\ _i|(\circ)^{j-i}\ _5])\ (i/4))\ (j/2)) \\ &= C_{25}([(\circ)^{4-j}\ _3|\ _4|(\circ)^{j-4}\ _5])\ (j/2)) \\ &= C_{25}([(\circ)^2\ _3|\ _4|(\circ)^{-2}\ _5]) \\ &= (\circ)^2 \end{aligned}$$

7.B Appendix: The PAC transitive closure program compressed

The PAC encoding of the transitive closure program of Section 7.3.4 is:

$$\begin{aligned} & [\ (\mathcal{C})^{j-1}\mathcal{C}'(\mathcal{C})^{i-j-1}\mathcal{C}''(\mathcal{C})^{n-i}\ _{i-1}| \\ & (\mathcal{C})^{i-1}\mathcal{C}'''(\mathcal{C})^{n-i}\ _i| \\ & (\mathcal{C})^{i-1}\mathcal{C}''(\mathcal{C})^{j-i-1}\mathcal{C}''(\mathcal{C})^{n-j}\ _n] \end{aligned}$$

It is possible to rewrite the $\mathcal{C},\ldots,\mathcal{C}'''$ macros to have length $\mu = \sigma = 4$ with the same instructions as the $\mu = \sigma = 5$ microprogrammed ISA transitive closure program of Section 3.5.2. This illustrates the gain in efficiency due to the extra flexibility of a PA over the microprogrammed ISA. The rewritten macros are

(lower micros are executed first)

$\mathcal{C}$:

$A \leftarrow A + B$
$B \leftarrow BC'_S$
$B, C'_E \leftarrow C_W$
$C'_S \leftarrow C_N$

$\mathcal{C}'$:

$C'_E, A \leftarrow A + C'_E$
$C'_E \leftarrow C'_E C'_S$
$C'_E \leftarrow C_W$
$C'_S \leftarrow C_N$

$\mathcal{C}''$:

$C'_S, A \leftarrow A + C'_S$
$C'_S \leftarrow C'_S C'_E$
$C'_E \leftarrow C_W$
$C'_S \leftarrow C_N$

$\mathcal{C}'''$:

$C'_E \leftarrow A$
$C'_S, A \leftarrow A + C'_S$
$C'_S \leftarrow C'_S C_W$
$C'_S \leftarrow C_N$

The correctness of this program can be established using the microprogramming timing rules of Section 4.4.2, with $\mu = \sigma = 4$. For iteration k of the algorithm, cell (i, j) reads the value v of the C'_S register of cell $(i-1, j)$ only on (micro)cycle 1, (just) before cell $(i-1, j)$ first updates it for iteration $k+1$. Since the final update for iteration k of this register must have occurred by the previous microcycle, it can be established (providing the rest of the program is correct) inductively that v is the value of A'_{kj} after it has passed through row $i-1$. A similar argument applies to the reading of the C'_E register of cell $(i, j-1)$.

Chapter 8

Conclusions

Seemingly insatiable demands for higher and higher performance in the supercomputing and real-time signal processing areas force the required degree of parallelism to a larger scale. The questions of appropriate high-level programming languages, processor architectures and processor granularity for parallel computing are still open. However, as long as the current constraints of VLSI and WSI remain, the mesh-connected network of processing elements will be widely used, even if only as an underlying architecture for implementing more general interconnection topologies.

The key issue of confident and efficient programmability has been addressed here mainly from the aspect of control structures. As advances in VLSI and WSI make the production of larger-scale meshes feasible, control structures based on uniprocessor or small-scale system concepts will prove inadequate and will have to be redesigned. The control structures proposed herein are designed for the (microprogrammed) Instruction Systolic Array and Processor Array models of mesh. They share these meshes' suitability for VLSI in terms of regularity, locality of data and control, modularity and expandability. They can be designed with modest area overheads for large-scale and fine- to medium-grained meshes, and can substantially improve their overall cost, performance and programmability.

Detailed conclusions regarding control structures and other related aspects of mesh programmability are given in the concluding sections of the preceding chapters. Considering this work as a whole, several main conclusions emerge:

- **Efficient, partitionable mesh interfaces are feasible**

Partitionable mesh data interfaces (ie. data buffers), whose control structures support partitioning and data formatting/ordering, can enhance the period efficiency of mesh systems. From the control structures developed for ISA and PA program compression, area-efficient and partitionable program interfaces can be designed.

- **Program compression enhances the ISA**

Program compression techniques are important for efficient storage and loading of mesh programs. Because of their amply high compression rates, the 'optimal' program compression methods (Subroutining and ISAC) substantially reduce the overall I/O bandwidth (and pin count) of an ISA system, while reducing its hardware costs. Vector-orientated SIMD meshes can benefit similarly from these methods. The non-'optimal' program compression methods can increase the simplicity (SISA) and flexibility (microprogramming) of the ISA model. Combining 'optimal' and non-'optimal' techniques to give an effective combination of these advantages should be considered for any medium to large-scale ISA system.

Of the 'optimal' methods, the wavefront-based method of Subroutining/ISADL should be chosen for the (microprogrammed) ISA, on the grounds of its demonstrated high effective flexibility in implementing program compression, surprisingly modest control structure overhead, compatibility with both high-level ISA languages (which are also wavefront-based) and ISA microprogramming, and very high program loading performance.

- **Microprogramming the ISA offers many advantages**

The dynamic form of vertical microprogramming used by the (microprogrammed) ISA forms an important step in developing the ISA model for general-purpose matrix computations. Having this motivation, the microprogrammed ISA is theoretically and practically demonstrated to be more powerful (ie. flexible) than the SIMD mesh or (non-microprogrammed) ISA, making it a unique application of microprogramming. With modest control structures, the microprogrammed

ISA can emulate an arbitrary instruction granularity ISA, which leads to a programming methodology for the ISA model which abstracts from instruction set details.

Efficient emulation of programs based on more general topologies (ie. hex-connected arrays) using an orthogonally-connected mesh is possible using these microprogramming techniques. For general-purpose matrix computations, use of such emulations is more area-period efficient than implementing these (expensive and occasionally used) topologies directly.

The microprogrammed ISA can be given a simple formal definition using the weakest precondition semantics. Practical verification of wavefront-based microprogrammed ISA programs is made possible by a wavefront-based proof method, which was developed from the microprogrammed ISA's semantic definition. The compactness of the proofs of these programs compares favourably with their uniprocessor counterparts, and both the semantics and the proof method can be easily extended to more general wavefront-based mesh programs.

- **Program compression enhances fine- to medium-grained Processor Arrays**

Although Processor Arrays implicitly incorporate program compression using the *cellprogram* concept, the explicit use of (ISAC-based) program compression concepts can significantly reduce program loading delays and program memory area for these meshes. An area and period-efficient alternative to relying on a large cell program memory, capable of storing many programs, is to use efficient program loading techniques on a 'minimal' cell program memory. This can be enhanced by performing load-time evaluation of cell position dependent expressions and overlapping program loading and execution. This improves the attractiveness of constructing finer-grained (and hence larger-scaled) Processor Arrays.

A medium-level programming language called PAC supports this alternative using control structures of modest area and using logic no more complicated than the cell program memories themselves. *Vertical (wavefront) interleaving*, again requiring only modest control structures, can be incorporated in PAC and is capable of significantly improving the flexibility of a Processor Array.

- **Broader issues for program compression should be considered**

While explicit program compression techniques can be developed for non-mesh architectures, they are most important for instruction systolic architectures, such as the (microprogrammed) ISA. This is because a systolic flow of control information means that (i) program compression is not implicitly incorporated through the *cellprogram* concept; (ii) the control structures implementing program compression can be 'factored out'[1]; and (iii) the architecture can be made larger-scaled due to its finer granularity. However, as any parallel architecture becomes large scale, with thousands or more cells, efficient program storage and loading, and hence explicit program compression concepts, must be considered.

While program compression techniques offer an inexpensive solution to the reduction of program input bandwidth, achieving a high overall performance requires a solution to the problem of a high data I/O bandwidth. A general solution to this problem is beyond the scope of this thesis, but is likely to be expensive. Even in this case, program compression techniques can significantly improve the cost-effectiveness of a parallel architecture.

- **The ISA is superior to the SIMD mesh**

For the case of a boolean ISA, even a simple ISA instruction set can implement a wide range of algorithms. The ISA's disadvantages over the SIMD mesh — the critical nature of its instruction granularity and its extra program input bandwidth — are mitigated by area-efficient control structures derived from the (abstraction aspect of) microprogramming and 'optimal' program compression techniques. Microprogramming also enhances the ISA's chief advantage (apart from the locality of its control paths) over the SIMD mesh: its superior handling of nontransmittent data. With the development of high-level programming facilities, which have begun with ISA Subroutining and microprogramming, the ISA/SISA model will be in all ways superior to the SIMD mesh.

[1] ie. one for each control path, rather than one for each of the more numerous processing elements.

- **The wavefront programming model is appropriate for meshes**

The underlying wavefront programming model is useful in discussing mesh control structure issues, and can be supported by efficient mesh control structures[2]. Macro-level wavefronts can then be efficiently supported by imposing software constraints. The success of the microprogrammed ISA's wavefront-based proof method demonstrates that the wavefront is a suitable 'semantic unit'. This, combined with the use of Subroutining as a high-level ISA language, indicates that the wavefront model can provide a suitable basis for mesh high-level languages and verification techniques.

The wavefront model is based on the systolic array approach, which enables very high performance, hardware-efficient implementations that are unfortunately rather esoteric, ie. expensive in terms of 'brainware'. Thus, the development of high-level programming and abstraction concepts is essential for the widespread use of architectures based on this approach[3]. The similarities between the semantic modelings of the non-systolic (in terms of control flow) SIMD mesh and the microprogrammed ISA indicate that such abstraction concepts should be possible. The wavefront model may well prove to be useful in this respect.

In summary, this thesis has examined the issue of confident and efficient programmability, with its aspects of high-level languages, control structures and program verification, for fine- to medium-grained, communication register-based and deterministic meshes, using the wavefront programming model. These meshes' high performance potential makes them attractive candidates for large-scale parallel computer systems, but I/O limitations, lack of flexibility and difficulty of programming have been perceived as critical impediments to their wider application. Taken together, the contributions reported herein represent a significant step towards overcoming these limitations.

[2]Instruction/selector queues, in the case of the (microprogrammed) ISA, and PAC delay programs, in the case of PAs, support basic wavefronts. PAC selection constructs and vertical interleaving support partitioned and interleaved wavefronts, respectively.

[3]The W2 language of the Warp processor uses such abstraction concepts [27, Ch.3].

Bibliography

[1] Annaratone M. et al., *The Warp Computer: Architecture, Implementation and Performance*, IEEE Transactions on Computers, Vol. C-36, No. 12, pp 1523–1537, December 1987.

[2] Boyle J. et al., *Portable Programs for Parallel Processors*, Holt, Reinhart and Winston, Inc., 1987.

[3] Brent R.P., Luk F.T., *The Solution of Singular-value and Symmetric Eigenvalue Problems on Multiprocessor Arrays*, SIAM J. Sci. Stat. Comput., Vol. 6, No. 1, pp 69–84, January 1985.

[4] Bruegge B. et al., *Programming Warp*, proc. IEEE COMPCON Spring 1987, pp 268–271, February 1987.

[5] Chen M., Yao K., *On Realizations of Least-Squares Estimation and Kalman Filtering by Systolic Arrays*, in: Moore W. et al. (eds), *Systolic Arrays*, pp 191–200, Adam-Hildiger, 1987.

[6] Dijkstra E.W., *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*, Communications of the ACM, Vol. 18, pp 453–457, August 1975.

[7] Dittrich A., Schmeck H., *Givens Rotation on an Instruction Systolic Array*, Bericht 8602, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1987.

[8] Fisher A.L., *Verification of VLSI Circuits* (lecture), Computing Sciences Laboratory, Australian National University, December 1989.

[9] Fisher A.L., Kung H.T., *Synchronizing large VLSI Processor Arrays*, IEEE Transactions on Computers, Vol. C-34, No. 8, pp 734–740, August 1985.

[10] Fisher A.L., Hung H.T., Sarocky K., *Experience with the CMU Programmable Systolic Chip*, technical report CMU-CS-85-16, Department of Computer Science, Carnegie-Mellon University, 1985.

[11] Floyd R.W.,*Assigning Meanings to Programs*, proc. SIAM Mathematical Aspects of Computer Science, Vol. 19, pp 19–32, 1967.

[12] Foster, M.J. and Kung, H.T., *The Design of Special-purpose VLSI Chips*, IEEE Computer, Vol. 13-1, pp 26–40, 1980

[13] Gentleman W.M., Kung, H.T., *Matrix Triangularization by Systolic Arrays*, Proc. SPIE Symp. Vol. 298, Real-Time Signal Processing IV, pp 19–26, 1981.

[14] Hillis, D., *The Connection Machine*, MIT Press, 1986.

[15] Hoare C.A.R., *Communicating Sequential Processes*, Communications of the ACM, Vol. 21, pp 666-677, August 1978.

[16] Hoare C.A.R. et al., *Data Refinement Refined*, Programming Research Group, Oxford University, May 1985.

[17] Iliffe J.K., *Advanced Computer Design*, Prentice-Hall, 1982.

[18] Inmos Ltd., *Inmos Transputer Development System 2.0: Software Tools*, Inmos Ltd., 1986.

[19] Ishii M, Goto G., Hatano Y., *Cellular Array Processor CAP and its Application to Computer Graphics*, FUJITSU Scientific and Technical Journal Vol. 23 No. 4, pp 379–390, December 1987.

[20] Krishnamurthy V., *Instruction Systolic Arrays for Exact Parallel Linear Algebraic Computation*, PhD thesis, University of Waikato, 1988.

[21] Kunde M., Lang H.W., Schimmler M., Schmeck H., Schröder H., *The Instruction Systolic Array and its Relation to other Models of Parallel Computers*, in: Feilmeier M. et al. (eds), *Parallel Computing 85*, pp 491–497, North Holland, 1986.

[22] Kung H.T., *Why Systolic Architectures*, IEEE Computer, Vol. 15, No. 1, pp 37-46, 1982.

[23] Kung, H.T., Leiserson, C.E., *Systolic Arrays (for VLSI)*, in: Duff I.S., Stewart G.W. (eds), Symposium on Sparse Matrix Computations, 1978.

[24] Kung H.T., Menzilcioglu O., *Warp: A Programmable Systolic Array Processor*, Proc. SPIE Symp., Vol 495, *Real Time Signal Processing VII*, August 1984.

[25] Kung H.T., *Memory Requirements for Balanced Computer Architectures*, Proc. IEEE 13th Annual Symp. on Computer Architecture, pp 49–54, June 1986.

[26] Kung S.Y., *VLSI Array Processors*, Prentice-Hall, 1988.

[27] Lam M., *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, 1989.

[28] Lang, H.W., Schimmler, M., Schmeck, H., Schröder, H., *Systolic Sorting on a Mesh-connected Network*, IEEE Trans. on Computers, Vol. C-34, pp 652–658, 1985

[29] Lang H.W., Schimmler M., Schmeck H., Schröder H., *A Method for Realistic Comparisons of Sorting Algorithms for VLSI*, Bericht 8406, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1985.

[30] Lang H.W., *The Instruction Systolic Array, a Parallel Architecture for VLSI*, Bericht 8502, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1985.

[31] Lang H.W., *ISA and SISA: Two Variants of a General Purpose Systolic Array Architecture*, Proc. Second Int. Conf. on Supercomputing, Vol. I, pp 460-465 1987.

[32] Lang H.W., *Transitive Closure on the Instruction Systolic Array*, Bericht 8718, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1987.

[33] Lenders P., Schröder H., *A Semantics for Instruction Systolic Arrays*, submitted to: Journal of Parallel and Distributed Computing, 1989.

[34] Lenders P., Schröder H., Strazdins P., *Microprogramming Instruction Systolic Arrays*, proc. MICRO-22, 22nd Conference on Microprogramming, Dublin, August 1989.

[35] Lang H.W., *Das befehlsystoliche Porzessorfeld*, PhD thesis, Universität Kiel, 1989.

[36] Leighton F.T., Leiserson C.E., *Wafer Scale Integration of Systolic Arrays*, proc. IEEE COMPCON, pp 297–311, 1982.

[37] Leiserson, C.E., *Area-efficient VLSI Computation*, MIT Press, Cambridge, MA, 1982.

[38] Lyall A. et al., *Implementation of Inexact String Matching on the ICL DAP*, in: Feilmeier M. et al. (eds), *Parallel Computing 85*, pp 235-240, North Holland, 1986.

[39] McGraw J.R., Axelrod T.S. *Exploiting Multiprocessors: Issues and Options*, article in: Babb R.G. (ed), *Programming Parallel Processors*, pp 7–25, Addison-Wesley, 1988.

[40] Mead C., Conway L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[41] Moldovan, D.I., *On the Design of Algorithms for VLSI Systolic Arrays*, Proceedings of the IEEE, Vol. 71, No. 1, pp 605–612, 1983

[42] Murthy V.K., Schröder H., *Systolic Arrays for Parallel G-Inversion and Finding Petri Net Invariants*, *Parallel Computing* 11 (1989), pp 349-359, 1989.

[43] Navarro J.J, Llaberia J.M., Valero M., *Partitioning: an Essential Step in Mapping Algorithms into Systolic Array Processors*, IEEE Computer, Vol. 20, No. 1, pp 77-89, July 1987.

[44] Quinn M.J., *Designing Efficient Algorithms for Parallel Computation*, McGraw-Hill, 1987.

[45] Ramanchandran W., *On Driving Many Long Wires in a VLSI Layout*, proc. IEEE COMPCON, pp 369-378, 1982.

[46] Schimmler, M., *Fast Sorting on the Instruction Systolic Array*, Bericht 8709, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1987.

[47] Schimmler M., *private communications*, January 1988.

[48] Schimmler M., Schröder H., *Finding all Cut-Points on the Instruction Systolic Array*, Bericht 8717, Institut für Informatik und Praktische Mathematik, Universität Kiel, November 1987.

[49] Schimmler M., & Schröder H., *Optimal Graph Algorithms on the Single Instruction Systolic Array*, Computer Sciences Laboratory, Australian National University, April 1988.

[50] Schimmler M., Schröder H., *A simple Systolic method to find all Bridges in Undirected Graphs*, Workshop in Graph Theoretic Concepts in Computer Science, Amsterdam, May 1988.

[51] Schmeck H., *A Comparison-based Instruction Systolic Array*, in: Cosnard M. et al. (eds), *Parallel Algorithms and Architectures*, pp 281–292, Elsevier Science, 1986.

[52] Schröder, H., *Top-down Design of Instruction Systolic Arrays for Polynomial Interpolation and Evaluation*, Journal of Parallel and Distributed Computing, Vol. 6, pp 692-703, 1988.

[53] Schröder H., Strazdins P., *Program Compression on the Instruction Systolic Array*, proc. Australian Computer Science Conferenece ACS-12, pp 76–87, February 1989.

[54] Simon H.D., *Supercomputers: Architecture, Algorithms and Applications* (lecture course), CSIRO, Canberra, September 1989.

[55] Stone H. (ed), *Introduction to Computer Architecture*, SRA Books, 1980.

[56] Tsutomu H., *The PAX Computer*, Addison-Wesley, 1985.

[57] Ullman J.D., *Computational Aspects of VLSI*, Computer Science Press, 1984.

[58] Ward C, and Davie E., *The Application and Development of Wavefront Array Processors for Advanced Front-end Signal Processing Systems*, in: Moore W. et al. (eds), *Systolic Arrays*, pp 295–302, Adam-Hildiger, 1987.

[59] Whitehouse H.J., Speisser J.M., Bromley K., *Applications of Concurrent Processor Array Technology*, in: Kung S.Y. et al. (eds), *VLSI and Modern Signal Processing*, pp 25–41, Prentice-Hall, 1986.