Regular Datapaths on Field-Programmable Gate Arrays

Vom Fachbereich für Mathematik und Informatik
der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Inform. Andreas Koch

Eingereicht am 09.07.1997
1. Referent: Prof. Ulrich Golze
2. Referent: Prof. Rolf Ernst
Mündliche Prüfung am 02.09.1997

For Anja.

Abstract

Field-Programmable Gate Arrays (FPGAs) are a recent kind of programmable logic device. They allow the implementation of integrated digital electronic circuits without requiring the complex optical, chemical and mechanical processes used in a conventional chip fabrication. FPGAs can be embedded in traditional system designflows to perform prototyping and emulation tasks. In addition, they also enable novel applications such as configurable computers with hardware dynamically adaptable to a specific problem.

The growing chip capacity now allows even the implementation of CPUs and DSPs on single FPGAs. However, current design automation tools trace their roots to times of very limited FPGA sizes, and are primarily optimized for the implementation of random glue logic. The wide datapaths common to CPUs and DSPs are only processed with reduced performance.

This thesis presents Structured Design Implementation (SDI), a suite of specialized tools coordinated by a common strategy, which aims to efficiently map even larger regular datapaths to FPGAs. In all steps, regularity is preserved whenever possible, or restored after disruptive operations were required.

The circuits are composed from parametrizable modules providing a variety of logical, arithmetical and storage functions. For each module, multiple target FPGA-specific implementation alternatives may be generated in both gate-level netlist and layout views.

A floorplanner based on a genetic algorithm is then used to simultaneously choose an actual implementation from the set of alternatives for each module, and to arrange the selected module implementations in a linear placement. The floorplanning operation optimizes for short routing delays, high routability, and fit into the target FPGA.

In addition, the coarse granularity of an FPGA as compared to a gate array (large logic blocks instead of small transistors as building blocks) necessitates a compaction phase to avoid inefficiencies. Floorplanning takes this into account by grouping modules amenable to compaction, and prepares for a merging of their functions across module boundaries.

For each set of compactable modules, structure extraction and regularity analysis phases search for a common regular bit-sliced structure across all modules in the set. The new master-slices thus discovered are then processed using conventional logic synthesis and technology mapping techniques, reducing both area and delay over their pre-compaction levels.

Since the originally generated module layout is invalidated by the compaction operation, the mapped logic blocks in each compacted master-slice have to be re-placed in a regular manner. This microplacement operation is performance-driven, and optimizes delay, control signal routing and slice abutment across master-slice boundaries. The compacted modules are then reassembled from the microplaced master-slices according to the structural information extracted previously.

The result is the efficient mapping of a regular bit-sliced datapath architecture to a regular bit-sliced layout. Practical experiments show delay reductions of up to 33% as compared to layouts produced by conventional tools. The exploitation of regularity during processing also reduces CAD runtimes by up to 78%.

Kurzfassung

Field-Programmable Gate-Arrays (FPGAs) sind eine noch junge Art von programmierbaren Logikbausteinen. Sie erlauben die Implementierung von integrierten Digitalschaltungen ohne die komplizierten optischen, chemischen und mechanischen Prozesse, die normalerweise für die Chipfertigung erforderlich sind. FPGAs können im Rahmen konventioneller Entwurfsmethoden zu Emulationszwecken und Prototyp-Aufbauten herangezogen werden. Sie erlauben aber auch völlig neue Anwendungen wie rekonfigurierbare Computer, deren Hardware dynamisch an ein spezielles Problem angepaßt werden kann.

Die gewachsene Chip-Kapazität erlaubt nun sogar die Implementierung von CPUs und digitalen Signalprozessoren (DSPs) auf einem einzelnen FPGA. Die Leistungsfähigkeit der entstandenen Schaltungen wird jedoch durch die zur Zeit erhältlichen CAD-Werkzeuge limitiert, da diese noch auf stark beschränkte FPGA-Größen ausgerichtet sind und primär der platzsparenden Verarbeitung unregelmäßiger Logik dienen. Die breiten Datenpfade in Bit-Slice-Struktur, die den Kern vieler CPUs und DSPs darstellen, werden nur suboptimal behandelt.

Diese Arbeit stellt Structured Design Implementation (SDI) vor, ein System von spezialisierten CAD-Werkzeugen, die auch größere reguläre Datenpfade effizient auf FPGAs abbilden. In allen Verarbeitungsschritten wird dabei die bestehende Regularität soweit wie möglich erhalten oder nach regularitätsvernichtenden Operationen wiederhergestellt.

Zur Schaltungseingabe steht eine Bibliothek von allgemeinen Modulen aus den Bereichen Logik, Arithmetik und Speicherung bereit. Diese können durch Belegung verschiedener Parameter wie Bit-Breiten und Datentypen an aktuelle Anforderungen angepaßt werden. Für jedes der Module können unterschiedliche Implementierungsalternativen in Form von Gatternetzlisten oder Layouts generiert werden.

Ein Floorplanner, basierend auf einem genetischen Algorithmus, wählt anschließend, bei gleichzeitiger linearer Plazierung der Module, für jedes Modul die günstigste Alternative aus. Dabei wird in Hinsicht auf kurze Leitungsverzögerung, gute Verdrahtbarkeit und Einpassung in das Ziel-FPGA optimiert.

Die grobe Granularität von FPGAs im Vergleich zu konventionellen Gate-Arrays (große Logikblöcke statt feiner Transistoren) erfordert eine Kompaktierung, um Ineffizienzen zu vermeiden. Dazu werden während des Floorplanning geeignete Module zusammen plaziert und die Verschmelzung ihrer Funktionen über Modulgrenzen hinweg vorbereitet.

Aus jeder Gruppe von zu verschmelzenden Modulen wird nun eine modulübergreifende reguläre Bit-Slice-Struktur extrahiert und diese auf Regularitäten hin untersucht. Die auf diese Weise bestimmten neuen Master-Slices werden anschließend mittels konventioneller Logiksynthese- und Technologieabbildungsverfahren in Bezug auf Flächenbedarf und Verzögerungszeit optimiert.

Da diese Operationen das ursprünglich generierte Layout ungültig machen, müssen die Logikblöcke in den optimierten Master-Slices wieder neu plaziert werden. Diese Mikroplazierung zielt auf die Wiederherstellung eines regulären Layouts hin und optimiert dabei die Signalverzögerungen, die Verdrahtung von Slice-übergreifenden Steuerleitungen und die Anreihbarkeit der Slices. Die kompaktierten Module werden dann entsprechend der vorher extrahierten Struktur aus den mikroplazierten Master-Slices neu aufgebaut.

Das Ergebnis dieser Vorgehensweise ist die effiziente Abbildung eines regulären Datenpfades auf ein reguläres Layout unter Erhaltung der Bit-Slice-Struktur. Praktische Experimente haben eine Verminderung der Schaltungsverzögerung um bis zu 33% im Vergleich zu konventionell berechneten Lösungen ergeben. Die konsequente Ausnutzung der Regularität führt auch zu einer Verkürzung der CAD-Rechenzeiten um bis zu 78%.

Acknowledgments

I would like to thank my thesis advisor Prof. Ulrich Golze, known to me since my first semester at Braunschweig, for many fruitful discussions, and giving me the freedom to explore and develop a broad range of CAD techniques. He also provided the hard- and software infrastructure indispensable for the success of my work.

Furthermore, I am grateful to Prof. Rolf Ernst for acting as a co-referee for this thesis.

The legibility of the text was markedly improved by considering the comments offered by Andrea Gondring and Ulf Bahrenfuss.

PARAMOG was made possible by the efforts of Holger Sadewasser and Jens Dittmer. It was their perseverance in reverse-engineering the XC4000 FPGA that allowed the exploitation of highly chip-specific structures during module generation.

The implementation of my SDI tool set was enabled by the free access to high-quality tools from other institutions. The well-documented and robust UCB SIS was used as framework for integrating my own functionality, as well as for providing logic optimization and technology mapping operations. For further experiments, Jason Cong (FlowMap) and Klaus Eckl (TOS-TUM) made their technology mapping tools available. Peter Barth supplied the OPBDP solver used as a first step of the hybrid ILP solving approach.

I am indebted to my parents for laying the groundwork that allowed me to successfully complete this and many of the other endeavors I have undertaken thus far.

The moral support by Anja Teske proved to be invaluable in the hectic final phase of this thesis' gestation.

Contents

1 Introduction **1**

2 Overview of Structured Design Implementation **11**
- 2.1 Xilinx XC4000 FPGAs 12
  - 2.1.1 Logic Block Architecture 12
  - 2.1.2 Routing Architecture 14
- 2.2 Structured Design Entry 17
- 2.3 Target Topology 19
  - 2.3.1 Datapath Topology 19
  - 2.3.2 Chip Topology 24
- 2.4 Module Generation 26
- 2.5 Module Selection and Floorplanning 28
- 2.6 Compaction 29
  - 2.6.1 The Need for Compaction 31
  - 2.6.2 Soft- and Hard-macros 31
  - 2.6.3 Preserving Module Placement 32
  - 2.6.4 Extracting and Exploiting Regularity 33
  - 2.6.5 Logic Optimization and Mapping 35
- 2.7 Microplacement 35
  - 2.7.1 Congestion Handling 36
  - 2.7.2 Pre-placement Activities 36
  - 2.7.3 Regularizing Logic Blocks 38
  - 2.7.4 Two-Phase Placement 40
- 2.8 Design Integration 45

3 Module Generators and Library **47**
- 3.1 Previous Work 47
  - 3.1.1 Generating Modules for Macro-Cells 48
  - 3.1.2 Generating Modules for FPGAs 48
  - 3.1.3 Module Templates 49
- 3.2 Anatomy of an SDI Module 49
  - 3.2.1 Classes of Regularity 50
  - 3.2.2 Geometric Hierarchy 50
  - 3.2.3 Structural Modules 52
- 3.3 Module Generation in SDI 53
  - 3.3.1 Paramog Architecture 53
  - 3.3.2 Design Cycle 53

3.3.3 Module Parameters 54
3.3.4 Design Alternatives 54
3.3.5 Generator Output 57
3.3.6 XC4000-specific Features 58
3.3.7 Implementation Details 59

4 Module Selection and Floorplanning 63
4.1 Optimization by Genetic Algorithms 63
4.2 Problem Description 63
4.3 Solution Representation 64
4.4 Genetic Crossover Operators 64
4.4.1 Uniform Crossover 65
4.4.2 One-Point Crossover 65
4.4.3 Two-Point Crossover 66
4.5 Genetic Mutation Operators 66
4.5.1 Allel Mutation 67
4.5.2 Position Mutation 67
4.5.3 Translocation Mutation 67
4.5.4 Reversal Mutation 68
4.6 Genetic Inversion Operator 68
4.7 Multi-Criteria Evaluation 68
4.7.1 Net Delays 69
4.7.2 Compactibility 69
4.7.3 Fit into Target FPGA 70
4.7.4 Routability 70
4.8 Selection 70
4.8.1 Elite Selection 71
4.8.2 Expected Value Selection 71
4.8.3 Fitness Selection 72
4.8.4 Random Selection 72
4.9 Parameters and Dynamic Fuzzy-Control 72
4.10 Capabilities and Limitations 73

5 Fundamentals for Compaction and Microplacement 75
5.1 Basics 75
5.2 Structure and Behavior of Digital Circuits 76
5.2.1 Hierarchy 76
5.2.2 Regularity 78
5.2.3 Grouping Bits 79
5.2.4 Structure of Circuits: Network Skeleton 80
5.2.5 Behavior of Circuits: Network 83
5.2.6 Master-Slices and Slices 85
5.2.7 V-Zones: Multi-Iteration Circuits 89
5.2.8 Stacks: Multi-v-zone Structures 91
5.2.9 H-Zones: Multi-Stack Structures 93
5.2.10 Modules: Multi-h-zone Structures 95

5.2.11 Datapaths: Multi-Module Structures 96
5.2.12 Flattening the Hierarchy 97
5.3 Topological and Geometrical Layout 100
5.3.1 Representing Geometrical Layout 100
5.3.2 Representing Topological Layout 101
5.3.3 Describing Regular Bit-Sliced Layouts 101

6 Regular Compaction 105
6.1 Finding Optimizable Areas 105
6.2 Flattening the Subdatapath 108
6.3 Structure Extraction 108
6.3.1 Requirements on Master-Slice Candidates 110
6.4 Regularity Analysis 112
6.4.1 Building Terminal Graphs 114
6.4.2 Constrained Isomorphism 116
6.5 Logic Optimization and Mapping 119
6.5.1 Tool Integration 120
6.5.2 Pre- and Post-Compaction Isomorphism 120
6.6 Summary and Relations between Structures 122
6.7 Effects on Placement 123

7 Microplacement 125
7.1 Vertical Topological Placement 125
7.2 ILP for Horizontal Geometrical Node Placement 127
7.2.1 Determining the Placement Area 128
7.2.2 Node Placement 129
7.2.3 Control Signal Routing 129
7.2.4 Critical Path Segment Delay Computation 133
7.2.5 Maximal Critical Path Delay 135
7.2.6 Vertical Inter-omS Alignment 136
7.2.7 Objective Function 138
7.3 Efficiently Solving 0-1 ILPs 139
7.3.1 Preprocessing and Constructive Enumeration 139
7.3.2 Pruned Branch-and-Bound 139
7.3.3 Capabilities and Limitations 140
7.4 Heuristic for Horizontal Geometrical Node Placement 140
7.4.1 Ensemble-Based Annealing 140
7.4.2 Optimization Cost Function 140
7.4.3 Capabilities and Limitations 141
7.5 ILP for Vertical Geometrical Node Placement 141
7.5.1 Node Placement 142
7.5.2 Vertical Distance in CLBs 142
7.5.3 Recognizing Linear Horizontal Placement 143
7.5.4 Recognizing Horizontally Abutting Cells 144
7.5.5 Recognizing Vertically Abutting Cells 144
7.5.6 Vertical SM Distance in a Single Column 146

7.5.7 Arc-Based Unit-to-Cell Assignment 146
7.5.8 SM Distance in Adjacent Columns 147
7.5.9 Computing Net Delay in SMs 148
7.5.10 Computing Path Delay in SMs 149
7.5.11 Computing Maximal Critical Path Delay 149
7.5.12 Objective Function 150
7.5.13 Solving the Vertical Microplacement 0-1 ILP 150
7.6 Handling Sequential Elements 150
7.6.1 Placing Bound Flip-Flops 151
7.6.2 Placing Floating Flip-Flops 152
7.7 Design Integration 153

8 Experimental Results 155
8.1 Tools Used 156
8.2 Generic 16-bit Datapath 157
8.2.1 Circuit 157
8.2.2 Processing 157
8.2.3 Performance 158
8.2.4 Comments 160
8.3 74181-based 32-bit ALU 160
8.3.1 Circuit 160
8.3.2 Processing 160
8.3.3 Performance 160
8.3.4 Comments 163
8.4 Address Generator for DES Encryption 163
8.4.1 Circuit 163
8.4.2 Processing 165
8.4.3 Performance 166
8.4.4 Comments 166
8.5 Logic Unit of RISC CPU 166
8.5.1 Circuit 166
8.5.2 Processing 166
8.5.3 Performance 169
8.5.4 Comments 169
8.6 Discussion 169

9 Summary and Future Work 173

Bibliography 176

Abbreviations 188

Index 190

List of Figures

1.1 Conceptual FPGA architecture 5
1.2 Logic block architectures 7
1.3 Routing architectures 8

2.1 SDI overview 11
2.2 Xilinx XC4002 FPGA 13
2.3 Xilinx XC4000 configurable logic block 14
2.4 XC4000 switch matrix 14
2.5 XC4000 single and double length lines 15
2.6 Connecting to long lines 16
2.7 Signal delays on different routing resources 16
2.8 Enlarged section of Figure 2.7 17
2.9 Example datapath providing simple arithmetic functions 18
2.10 Classic datapath structures 19
2.11 Folding modules jutting out of the placement area 20
2.12 Extents of datapaths and modules 21
2.13 Matched and mismatched bit-slice pitch 22
2.14 Examples for BPLB values 23
2.15 On-chip topology 24
2.16 SDI topology as used on the Sparxil processor 25
2.17 Regular structure of a module 26
2.18 Examples for module layouts 27
2.19 Multiple BPLB values in a single datapath 29
2.20 Compaction 30
2.21 Wasted space in a module-based layout 31
2.22 Hard-macros as boundaries of compaction areas 32
2.23 Sample datapath segment 33
2.24 H-zones and v-zones in a stack 35
2.25 Floorplan context of an unplaced, compacted module 36
2.26 LUT pin arrangement and configuration 37
2.27 Back-annotation of timing into optimized master-slice 38
2.28 XC4000 CLB and corresponding regular cells 39
2.29 Interchangeable netlist cells in CLBs 39
2.30 Cell-based placement matrix 40
2.31 Horizontal placement model 41
2.32 Vertical placement model with example TTN routing lengths d_{SM} 43
2.33 Real CLB routing structure (a) and abstract model (b) 44

2.34 Design integration 45

3.1 Geometric regularities in a module 51
3.2 Circuit and underlying module topology 52
3.3 Paramog architecture 54
3.4 Physical extents of a module and signal alignments 55
3.5 Layout styles for shift registers 56
3.6 Effects of different pin assignments for logic function $ab + c$ . . 57
3.7 Effects of different pin assignments for FFs 58
3.8 Basic tile for layout generation 59
3.9 Sample layout for a 6x6-bit multiplier 61

4.1 Decaying population size during optimization 73
4.2 Increasing mutation rate after reaching local optimum 73

5.1 (a) Graph hierarchy T and (b) hierarchy tree T_G 77
5.2 Regularity tree R_G. 79
5.3 Network skeletons 83
5.4 Iterating a master-slice to obtain slices 86
5.5 Logical completeness and abutment in an alternately folded unit 88
5.6 v-zone tree with root N_v, showing hierarchy and regularity . . . 90
5.7 From v-segment to h-zone: hierarchy and regularity relations . 94
5.8 Regularity and hierarchy in a datapath 97
5.9 Flattening a v-zone into v-segments 98
5.10 Geometrical (a) and topological layout (b) with representations 100
5.11 Hierarchical bottom-up layout of a datapath 104

6.1 Applying Algorithm 3 108
6.2 Flattening, structure extraction and regularity analysis 109
6.3 Creating new primary ports in MSCs 112
6.4 Increased precision of terminal graphs 113
6.5 Effects of logic processing 121
6.6 Isomorphic correspondence between terminals on N_f, $\mathfrak{M}$, and N_o 122

7.1 Post-compaction vertical topological re-placement 127
7.2 Control signal connectivity via VLL 132
7.3 Vertical inter-optimized master-slice alignment 137
7.4 Computing the set of nodes to align 138
7.5 Computing maximal critical path delay dsm 150
7.6 Placing bound and floating flip-flops 151

8.1 Single bit-slice of the example circuit 157
8.2 Placement and routing solely by PPR 158
8.3 SDI placement with PPR routing 159
8.4 32-bit 74181-based ALU implemented with XACT 161
8.5 32-bit 74181-based ALU implemented with SDI 162
8.6 32-bit 74181-based ALU implemented with XACT on XC4003 . 164
8.7 Bit-slice of address generator for DES encryption 165

8.8 UFC-A address generator implemented with XACT 167
8.9 UFC-A address generator implemented with SDI 168
8.10 SRISC logic unit implemented with XACT 170
8.11 SRISC logic unit implemented with SDI 171

List of Tables

2.1 SDI module library overview 27

3.1 Current list of LPM modules 47

8.1 Performance of generic 16-bit datapath 159
8.2 Performance of 74181-based 32-bit ALU 161
8.3 Logic processing statistics for 32-bit ALU 163
8.4 Performance of UFC-A address generator 166
8.5 Performance of 32-bit SRISC logic unit 169

List of Algorithms

1 Deriving geometrical layout 102
2 Find largest contiguous soft subdatapaths of datapath D 106
3 Determine datapath D_Q for the the module sequence $Q \in C_D$ . . 107
4 Finding raw master-slice candidates 110
5 Refining an rMSC into an MSC 111
6 Create a terminal label 115
7 Building terminal graphs 116
8 Isomorphism constraints on terminal labels 117
9 Isomorphism constraints on terminal labels in TTNs 118
10 Label-based test for constrained isomorphism 118
11 Regularity analysis 119
12 Restoring vertical topological placement after compaction 126
13 Determining the placement area per omS 128
14 Generating constraints for input control signal VLL routing . . 131
15 Generating constraints for output control signal VLL routing . 132
16 Generating constraints for all segments on critical paths 135
17 Generating constraints for vertical TTN delay in SMs 148
18 Generating constraints for path delay in SMs 149
19 Computing switch matrix distances between arbitrary nodes . . 152

1 Introduction

In the dynamic world of modern electronics, the counterpart to the mythical quest for the holy grail is the quest for a vorpal sword to fight the all-too-real wyrm of increasing design complexity. Unfortunately, such a wondrous weapon has not been discovered yet, and the noble quest has turned more into the search for a better mousetrap. While the abovementioned wyrm will remain undaunted by the later, the lifes of countless engineers and circuit designers in the trenches will be eased by each gradual refinement to such basic an implement.

This work will describe one such improvement: A set of CAD tools, and a strategy for their use, to efficiently realize regular datapaths on field-programmable gate arrays. Despite being far from the wished-for dragonslayer (and more in the mousetrap league), the effects of this highly specialized method are quite appreciable when considered in a wider context.

With increasing chip complexities, the requirement for ever-higher performance, and steadily narrowing time-to-market windows, traditional design techniques are becoming more and more difficult to apply successfully.

Traditional quality control methods, such as simulation-based validation, become increasingly cumbersome to use. Especially when the complexity of the individual chip, which even today already encompasses dozens of millions transistors, is eclipsed by the larger complexity of multi-chip *systems*. E.g., current telecommunications systems require simulation patterns on the order of 10^{11} of vectors [Qds96a] to support high confidence in the design. This huge number of vectors, combined with the changing requirement of running fewer, but longer simulations (to handle the growing complexity of implemented algorithms), instead of many shorter simulations, can no longer be processed in reasonable timespans. Even if an effort this massive is undertaken for a single chip, it usually cannot take system-level effects into account: Subtle interactions between multiple chips, especially with regard to rarely occurring boundary conditions, can usually only be discovered when observing an actually running system, instead of using human-devised test patterns [Qds96b]. The traditional breadboard-based prototyping using discrete components is often prohibited by the circuit or system complexity, or minimal real-time response requirements, however. Thus, another approach to the validation problem has to be found.

The quickly moving market also demands a hitherto unprecedented flexibility. Often, systems must be extremely adaptable to follow changing requirements such as upgrades of telecommunications standards, or new peripheral devices. While general purpose CPUs or DSPs could provide the maximum flexibility (all functionality implemented in malleable software), very fast processors are needed to meet the high performance requirements. Unfortunately, the considerable chip and system costs for these processors often precludes their use for all but very high-end applications.

For various applications, even the fastest available general purpose processors cannot provide the desired performance. Examples include particle detection and analysis in high-energy physics [NZKK94], or DNA sequence matching [BuAK96]. While an *application-specific integrated circuit* (ASIC) could fulfill the performance requirements, future changes can often only be anticipated by including a multitude of user-programmable parameters and operating modes on the chip. In addition to increasing the complexity of the basic design even more, this approach relies on the foresight and expertise of the designers to extrapolate all future usage variations. In case of an unforeseen variation, the ASIC becomes useless, or can only be used with considerable effort, performance degradation, or an expensive redesign.

In quite a few cases, *programmable logic*, especially in the form of field-programmable gate arrays, can be a solution to the problems sketched above. The efficient application of FPGAs, especially their support by novel optimized CAD algorithms, will be at the center of this work.

The details of the programming process for FPGAs depend on the specific on-chip technology used. Common methods of configuration storage include E-PROM, EEPROM, SRAM and anti-fuses[1]. Anti-fuse, EPROM and EEPROM configurations are non-volatile, SRAM configurations are volatile in that the configuration must be battery-preserved, or be loaded from an external non-volatile medium (ROM, external CPU etc.).

Especially re-programmable FPGAs can be used to perform validation by *emulation* instead of simulation [BEKS95]. Here, a single chip (or even an entire system) is partitioned into a number of FPGAs integrated with the remaining hardware components (e.g., peripheral devices, CPUs). In this manner, the complete system and environment are actually assembled in hardware, and can be used to perform conclusive system-wide tests (e.g., boot an operating system [Qds96b]), or be observed for billions of test vectors [Qds96a]. While the emulation speed is usually far slower than the target speed of the final ASIC (by a factor of 40 in [Qds96a]), it still exceeds simulation speeds by several magnitudes. Current emulation systems have capacities of millions of gates [Qds96c], and will benefit directly from the growth in FPGA capacities.

Non-reprogrammable FPGAs (e.g., anti-fuse-based) are unsuitable for fully automatic system emulation (no in-circuit reconfigurability). However, they can be used to extend the limits of traditional bread-board based prototyping approaches by integrating large amounts of formerly discrete logic into a single device.

The field-programmability of FPGAs also provides the required flexibility

---

[1] An anti-fuse has a high-impedance state, but can be brought permanently into low–impedance by briefly applying a (relatively high) programming voltage. In this manner, it can be used to selectively establish permanent connections.

in a world of short times-to-market and changing specifications. The critical path in a product cycle will be shorter due to the removal of long foundry lead times. Furthermore, the low NRE charges for FPGA-based implementations lower the cost of short-term design alterations (*engineering change order*, ECO), and thus significantly reduce the risk when tracking non-formalized standards. E.g., [3Com95] describes the design of a 10BASE-T Ethernet interface that was begun while the standards were still being formalized. By using FPGAs, the design could closely follow all changes in the emerging standard, and be released to the general market as soon as the standardization procedures were closed. Only after satisfying the initial demand with FPGA-based interfaces was a conversion to a classical gate array (to achieve lower unit costs) performed.

In-circuit programmable FPGAs are even more flexible: They allow hardware upgrades by simply shipping a new configuration bitstream (e.g., on a diskette or by network transfer [Xili96d]) to the customers. In this manner, hardware can be maintained as flexibly as software.

The basic performance of FPGAs is far slower than that of foundry-fabricated circuits. The delay of the fundamental elements (a gate for gate arrays, a logic block for FPGAs) differs by five orders of magnitude. However, reprogrammable devices can be configured with hardware specific to a given problem. E.g., for a certain dataset, the device might be loaded with a unit multiplying by a constant “5”. Another dataset might lead to a multiplication by “42” being generated. By actually adapting hardware to low-level problem parameters, and employing specialties such as non-Von Neumann architectures, multi-stage pipelining, and problem-specific instruction sets, an FPGA-based configurable processor, also called *field-programmable custom computing machine* (FCCM), can outperform even the most powerful conventional computers on certain problems.

E.g., for DNA sequence matching, an FCCM based on an array of 248 processing elements[2] outperformed even MP-1 and CM-2 supercomputers by factors of 1344 and 7288 (respectively) [BuAK96]. A very simple FPGA-based co-processor [KoGo94], containing just three Xilinx XC4010 FPGAs and two 256K × 32 bit memory banks, can label objects in black-and-white images 6.5 times faster than a general purpose SPARC 20/71 workstation [Meye97] [Koch97b].

It is interesting to trace the development of programmable logic in a larger historical context. In contrast to traditionally fabricated chips, *programmable logic devices* (PLD) are sold as “blank” devices which can later be *personalized* with a specific design. Ideally, this process, also called *configuration* or *programming*, is performed without resorting to foundry techniques (etching, photolithography etc.). Depending on the nature of the device, configuration may be performed once or many times, using dedicated programming equipment, or by simple serial download to an already system-integrated device.

The various variants of programmable read-only memories (PROM, EPROM, EEPROM) may be viewed as PLDs: An $2^n \times m$ PROM can be used to imple-

---

[2] Each based on a Xilinx XC4010 FPGA with 256K × 16 bits of local memory.

ment m combinational functions of n variables (using the memory to hold a truth table with 2^n rows and m output columns). While the table-lookup character guarantees constant response times (without regard to the complexity of the function implemented), the memory size grows exponentially with the number of variables to evaluate, which becomes impractical for larger numbers of variables.

The next step in the evolution of PLDs was prompted by the insight that often only a small fraction of input variable combinations is relevant to the practical problem. Thus, it would suffice for the PLD to contain logic only for this subset of input combinations, and ignore the rest (*don't-cares*). This lead to the *programmable logic array* (PLA) , a device implementing combinational functions in a sum-of-products form. A PLA consists of a configurable AND-matrix connected to an also configurable OR-matrix. By appropriately programming the AND-matrix, the product terms may be composed. The program for the OR-matrix then assembles sums from the selected product terms. Since the AND-matrix is programmable (instead of the fully decoded address evaluation in a memory circuit), only the relevant input combinations are evaluated.

To reduce the fabrication complexity of blank PLAs, and increase performance by reducing delay, a variant called *programmable array logic* (PAL) does away with the programmable OR-matrix. Here, each sum is composed from a fixed number of products.

The current generation of programmable arrays are devices called *generic array logic* (GAL), based on PALs. They also include sequential elements (flip-flops and latches), and internal connections to feed outputs back into the array. In this manner, they can directly implement state machines.

While the simple two-level structure of programmable arrays, also called *simple* PLDs (SPLD) leads to short circuit delays (around 5ns for fast commercially available chips), it also limits the design complexity: With an increasing number of product terms, the internal connection network (separate from the AND/OR logic functions) grows impractically large. Thus, current SPLDs usually have an upper limit around 200 equivalent gates [BrRo96].

Classical *mask-programmable gate arrays* (MPGA) allow the efficient implementation (60ps gate delay) of circuits with up to 12 million gates [Texa97]. Their "blank" chips, also called gate array masters, consist of a matrix of prefabricated transistors. An MPGA is personalized by applying final metal layers to interconnect the transistors in the desired manner. Unfortunately, this process relies on foundry equipment, and cannot be performed "in the field". Typical turnarounds are measured in weeks and months, and non-recurring engineering (NRE) charges begin at 10,000s of dollars.

Recently, *laser programmable gate arrays* (LPGA) have become available. Their blank chips include all metal connections between transistors. A laser beam is then used to remove all extraneous connections, leaving only those specified by the netlist. While personalization of LPGAs does not rely on foundry technology, the need for a precision laser cutter also makes it impractical for "in the field" prototyping. Still, with turnarounds reduced to a few days, and NRE charges starting at under $10,000, LPGAs are very attractive for low-volume designs of up to 70k gates, with a gate delay of 300ps [Chip97].

The main advantage of MPGAs over SPLDs is the scalability of their structure: In contrast to SPLDs, logic functions and interconnect are not implemented independently of each other, but rely on the same resources (transistor matrix). Thus, when increasing the area for logic functions, the area for interconnect also increases (and vice versa).

The *field-programmable gate array* (FPGA), invented in 1985 by Xilinx Inc., combines the easy “in the field” programmability of SPLDs with the scalable logic and interconnection structure of MPGAs, allowing a currently available maximal capacity of 85k gates [Xili97], and near-term projected capacities of up to 400k gates [Acte96].

Unfortunately, the development of FPGA-specific CAD tools has not kept up with the growth in chip capacity. Many of the current tools trace their ancestry to a time, when FPGAs could only hold a thousand gates, and were primarily used to implement glue logic. When attempting to implement the more complex datapath structures common to many of today’s CPUs and DSPs in one of these larger FPGAs, the traditional tools are often overtaxed. The resulting circuits are often inefficient in terms of area use and performance. This work will describe a suite of CAD tools, and a strategy coordinating their use, specialized for efficiently mapping datapaths onto FPGAs.

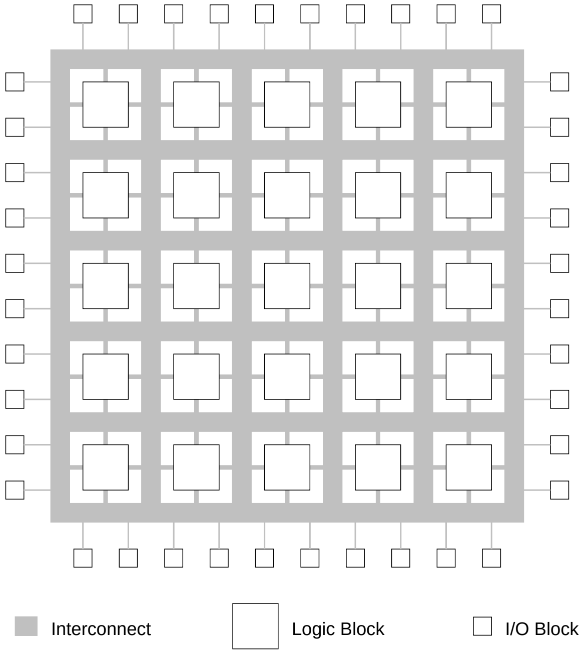


Figure 1.1: Conceptual FPGA architecture

Figure 1.1 shows the fundamental elements of a conceptual FPGA [BFRV92]. It consists of a number of programmable *logic blocks*, interconnected by a programmable *interconnection network*. Programmable *input/output blocks* at the periphery of the chip allow for chip-external communication.

The implementation of a circuit on an FPGA consists of three main steps:

1. The circuit netlist is *partitioned* into individual logic blocks (each of which can hold only a relatively small part of the logic). The result is a netlist of logic block configurations.

2. Each logic block configuration is then assigned to an actual logic block on the FPGA (*placement*).

3. The interconnection network is programmed for the connectivity proscribed by the netlist (*routing*).

All of these steps are highly dependent on the actual structures (*architecture*) of logic blocks and routing network. See [BFRV92] for a detailed discussion including a statistical analysis of different architectures. The next few paragraphs will present brief overviews of general logic block and routing architectures. An actual FPGA, the Xilinx XC4000 series, will be examined in greater detail in Section 2.1.

It is the fixed FPGA architecture that distinguishes CAD tools for FPGAs significantly from those for classical standard cell or gate array technologies. In an FPGA, it will not be possible, e.g., to simply increase the width of a routing channel to handle congestion: The channel width has been fixed on the FPGA die at fabrication time, and cannot be influenced by the user. Due to the long interconnection and logic block delays, even slightly suboptimal partitioning, placement, or routing can cause a significant performance degradation. Efficient CAD tools will have to be finely tuned to exploit a given FPGA target architecture for optimal results. To this end, our back-end tools have detailed knowledge about the capabilities of logic blocks, and the intricacies of the interconnection networks. In addition, however, we follow a two-pronged approach by also specializing the front-end for the regular datapath structures we intend to implement. In this manner, we can cover precisely those areas neglected by the current general-purpose tools.

The nature and size of logic that fits into a single logic block is determined by the logic block architecture. A logic block can be as primitive as a simple transistor pair [Cros94], or be as complex as to contain an integer multiplier, two ALUs, six registers and three local memories [EbCF96]. The most common logic blocks are based on one or more k-input lookup-tables (k-LUTs) (Figure 1.2.a) [Alte95] [ATTM95] [Xili96a] [Xili96b] [Xili96c], multiplexers (Figure 1.2.b) [Acte95a] , or on a PAL-like internal structure (Figure 1.2.c) [Alte96] [AMDI96]. Often, they also contain sequential elements such as flipflops or latches (not shown in Figure 1.2). Small logic blocks (in terms of logic capacity) are called *finely granular*, otherwise they are called *coarsely granular*.

Since the signal delays within a logic block are usually far shorter than those encountered when routing a signal through the general interconnection network, many current FPGAs are coarsely granular. However, for an efficient

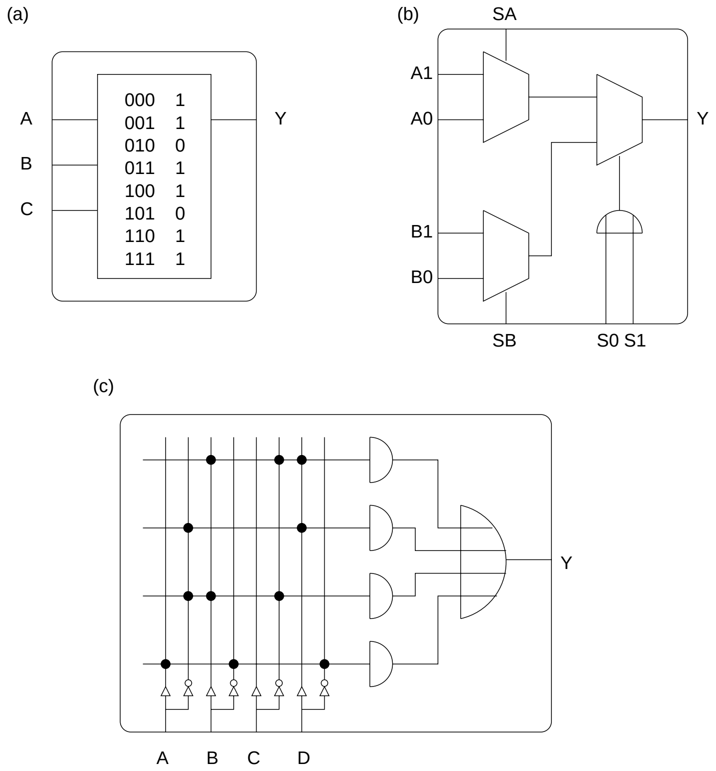


Figure 1.2: Logic block architectures: (a) look-up table, (b) multiplexer, (c) PAL

mapping to these chips, the large capacity of the logic blocks actually has to be filled. Due to their larger size (in terms of silicon area), a coarsely granular FPGA will have fewer logic blocks than a finely granular FPGA of the same gate capacity. Thus, partially filled coarsely granular blocks will waste a much larger fraction of chip gate capacity than partially filled finely granular blocks. To minimize this wastage, our system contains dedicated optimization passes aiming to maximally fill each logic block.

The routing architecture influences the performance of the FPGA-implemented circuit as well as the speed of the implementation process. A fast routing network will allow short connections between logic blocks. A very general routing network (e.g., a fully populated crossbar) will only need minimal computational effort to determine a configuration which provides the desired connectivity. More constrained architectures (e.g., connectable metal segments with varying lengths) require more complex routing algorithms.

However, the capabilities (speed and flexibility) of the routing network are limited by the silicon area available for its implementation. Thus, the design of a routing architecture is constrained by conflicting requirements:

1. **Capability**: All required connections should be routable with minimum delay. However, more flexible routing networks may consume more silicon area.

2. **Silicon area**: Since the die size of a chip is limited, a more flexible routing network consumes area possibly better used by logic blocks. However, a small but overconstrained routing architecture might hamper circuit performance (by forcing long “detour” connections), or even prevent the circuit from being routed at all (insufficient routing resources).

Note especially the last point: The number and layout of routing resources (similar to “channels” in standard cell technologies) on an FPGA is fixed at “blank” chip fabrication time. Thus, circuits to be used on this FPGA type have to get by with the resources available, or cannot be implemented at all on the given FPGA type. The full-custom and standard cell routing techniques of just increasing the channel widths in congested areas are unavailable on FPGAs, making efficient FPGA routing considerably more difficult.

While interconnection delay on FPGAs is also dependent on wire lengths and capacities, it depends primarily on the number and nature of programmable connections (anti-fuses, pass transistors, transmission gates, multiplexers), called *switches*, that the signal passes from source to sink.

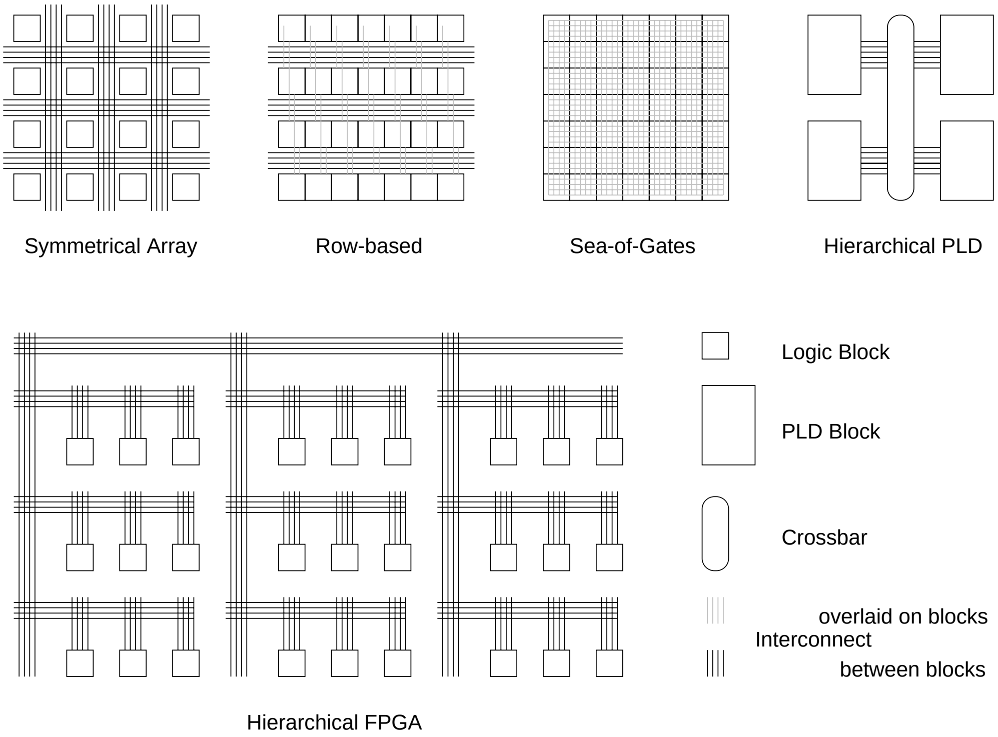

Symmetrical Array
Row-based
Sea-of-Gates
Hierarchical PLD
Hierarchical FPGA

Figure 1.3: Routing architectures

The most common routing architectures used today are shown in Figure 1.3. Assume that switches are placed at all wire intersections (but see Section 2.1). Without going into a more detailed discussion, in symmetrical arrays, row-based FPGAs and sea-of-gates FPGAs, geometrically adjacent logic blocks will have short interconnection delays (few switches in the signal path). FPGAs composed as hierarchical PLDs usually have distance-independent routing delays due to the (fully or partially populated) crossbar switch at the center of the chip. Routing delays on hierarchical FPGAs, however, are not proportional to geometrical distance. E.g., assume a source to be placed in the bottom-right logic block, and a sink in the logic block directly above it. While the logic blocks are adjacent, the signal has to cross three routing hierarchy levels (six switches) before it reaches its destination. Many FPGAs support a mix of routing architectures (e.g., a symmetrical array combined with hierarchical elements), or special features not covered by the general model. These might include dedicated resources for routing clock signals, or for distributing high fan-out signals with low skew over long distances.

As with logic block architectures, a good FPGA CAD tool needs to take the specific details of the target FPGA's interconnection network into account. Our system exploits these resources by mapping higher-level concepts (such as inter-bit-slice control signals) recognized by the front-end directly to their most efficient counterpart on the physical level.

By combining a front-end specialized for a specific class of circuits (wide regular datapaths, as needed in FCCMs and fast ASIC emulation) with a back-end optimized for the target FPGA architecture, we are able to generate faster circuits in shorter amounts of computation time. Given the abovementioned wide applicability of FPGAs in a modern VLSI design, even gradual improvements to small parts of a such a circuit can have appreciable effects on the entire system.

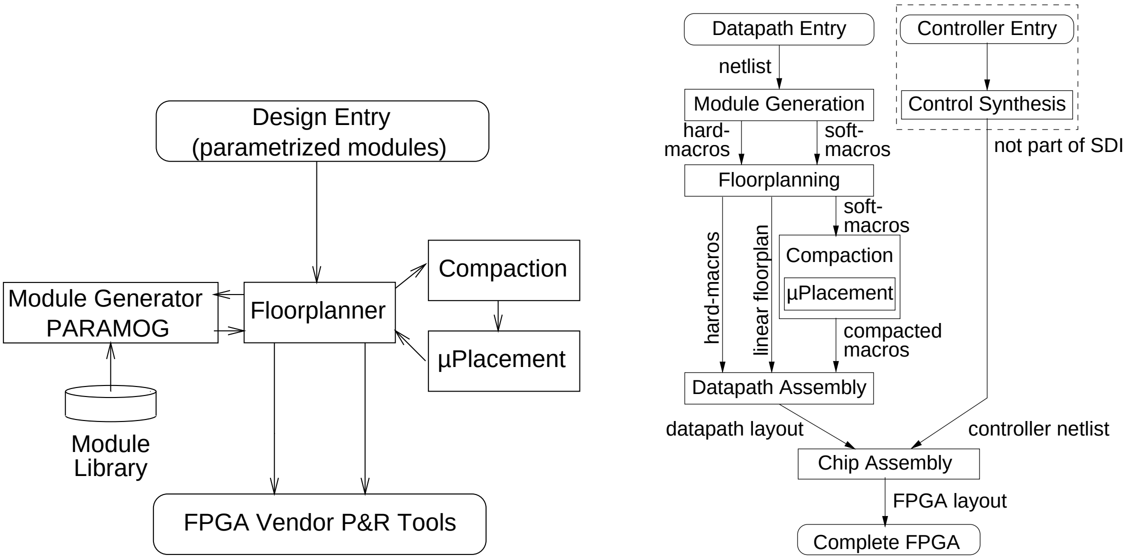
It's been said:
"Build a better mousetrap,
and the world will beat a path to your door."
But that's not the purpose of a mousetrap, is it?
– Eric S. Raymond on USENET

2 Overview of Structured Design Implementation

The points raised in Chapter 1 lead to *Structured Design Implementation* (SDI), a strategy for the efficient implementation of bit-sliced datapath structures on FPGAs. Bit-sliced architectures are called *regular* in context of SDI. The general organisation of the strategy and the design flow is shown in Figure 2.1 and outlined in the following sections.

This chapter presents an easily accessible overview of the entire system, while Chapters 3 to 7 will describe key components in greater detail or more formally.

SDI does not consist of a single tool, but a suite of specialized tools and a strategy coordinating their application. The suite combines a floorplanner, module generators, and tools for placement and global routing with minimization and technology mapping algorithms. It is thus difficult to compare it with specialized stand-alone tools that cover only part of the design implementation process, but these tools can often be integrated into SDI with minimal effort (see Section 2.6).



(a) System Architecture **(b)** Design Flow

Figure 2.1: SDI overview

2.1 Xilinx XC4000 FPGAs

To better understand the design decisions made during the development of SDI, it is helpful to examine the architecture of the currently targeted FPGA in greater detail. The Xilinx XC4000 series, one of the most popular chips, is a third generation FPGA and was introduced by Xilinx Inc. in 1990. Later revisions, such as the XC4000A, XC4000E, XC4000EX, and XC4000XL series, improve upon various details (e.g., routing facilities, synchronous on-chip memories etc.), but much of the basic design is still unchanged. The XC4085XL FPGA has a capacity of 85k gates and is currently (June 1997) the highest density device available[1]. With logic block delays of 1.3ns in the "-1" speed grade, XC4000EX-1 and XC4000XL-1 chips are also among the fastest current chips.

Figure 2.2 shows a "blank" XC4002 FPGA. The basic layout is very similar to the conceptual FPGA of Figure 1.1, with the routing architecture being a symmetrical array (Figure 1.3). The I/O blocks (IOB) are located at the sides of the array.

XC4000 FPGAs also have various special features not part of the conceptual FPGA. These include wide edge decoders, the use of logic blocks as fast on-chip 32x1 bit or 16x2 bit memories, a fast carry-propagation logic for adders and counters, and on-chip tri-state buffers (TBUF).

2.1.1 Logic Block Architecture

The logic blocks on an XC4000 FPGA are called *configurable logic blocks* (CLB). A simplified view of the internal structure is shown in Figure 2.3. Basically, a CLB consists of two 4-LUTs F and G, and a 3-LUT H. Furthermore, it contains two D-flipflops FFX and FFX. Some connections are hardwired (e.g., the H LUT accepts inputs only from the outputs of the F and G LUTs, and the H1 CLB input pin. However, others can be established at configuration time using a network of programmable multiplexers. In this manner, e.g., the output of the H LUT might be routed to the Y CLB output pin, or the DIN CLB input pin connected to the input of the FFX D-flipflop.

A single CLB may implement any two independent functions of two variables, any one function of five variables, any function of four variables together with a single of *some* functions of five variables, or some functions of up to nine variables.

Due to the large logic capacity of a CLB (as compared to transistor pairs and multiplexers), XC4000 FPGAs have a coarse granularity. Consider, e.g., the amount of CLB capacity wasted when implementing only a simple two-input AND gate in a single CLB. Thus, for the XC4000, the method of "packing" logic into CLBs with minimum wastage and delay becomes very important.

---

[1] Note that in order to obtain this much *configurable* capacity, the FPGA chip itself has 16 million transistors, three times the number in Intel's PentiumPro CPU [Xili97].

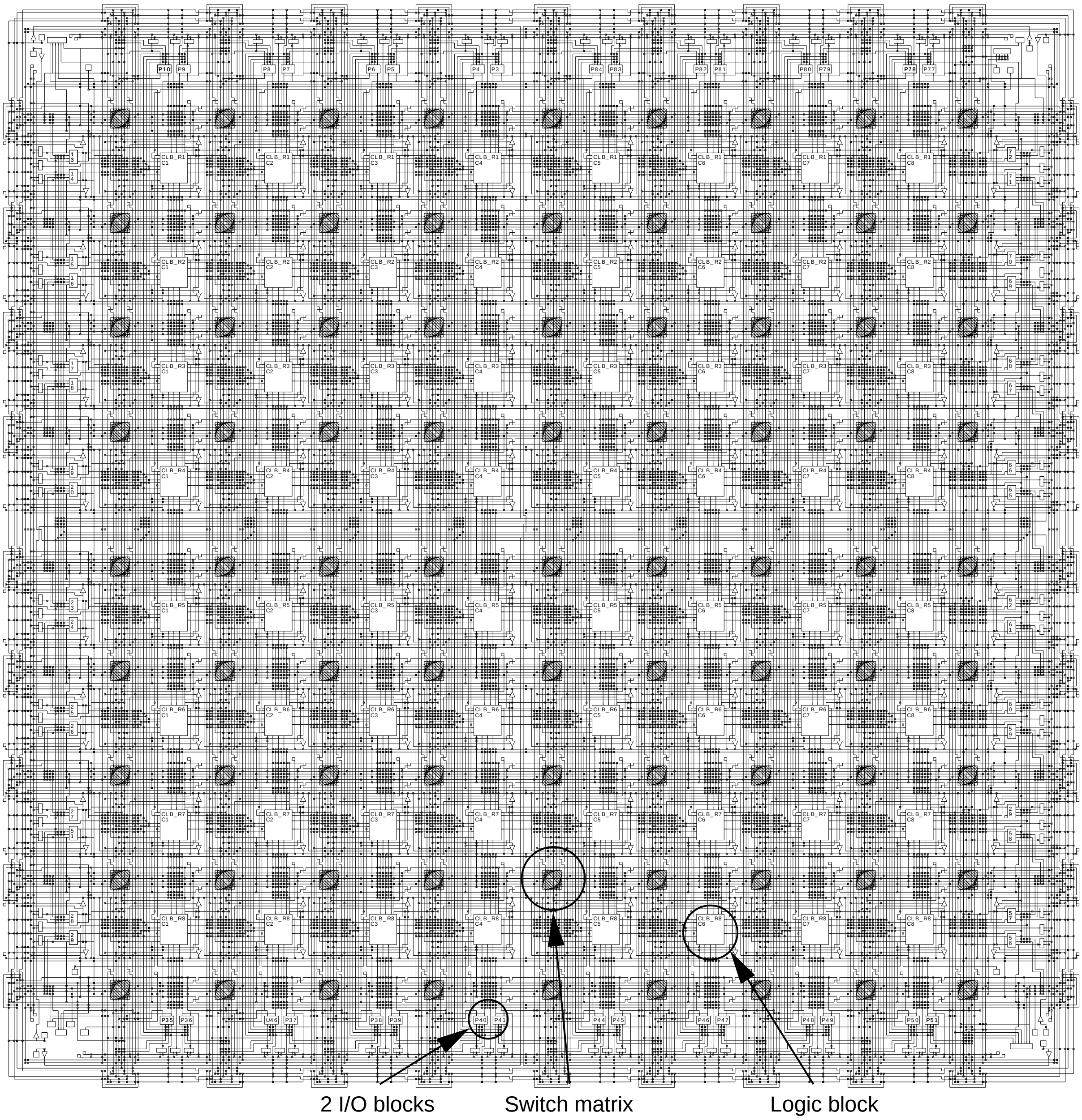


Figure 2.2: Xilinx XC4002 FPGA

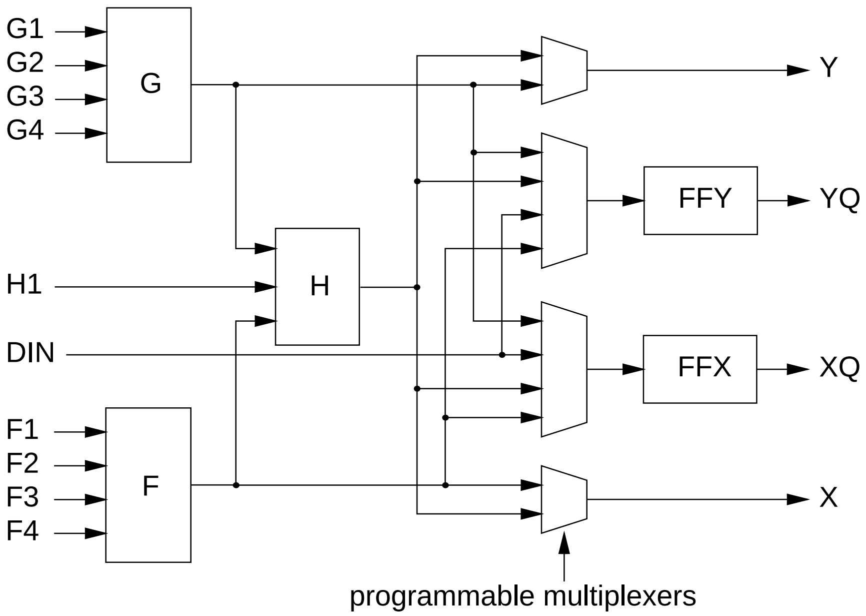


Figure 2.3: Xilinx XC4000 configurable logic block

2.1.2 Routing Architecture

As shown in Figure 2.2, an XC4000 FPGA consists of a symmetrical array of logic blocks, with metal segments of varying lengths placed between the logic blocks. Programmable connections, called *switch matrices* (SM), at the intersections of horizontal and vertical metal segments, allow the interconnection of individual segments to cross longer routing distances, possibly also routing a signal around a corner.

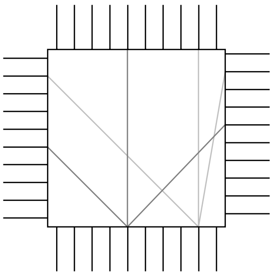


Figure 2.4: XC4000 switch matrix

Figure 2.4 shows a switch matrix and two of the possible connections. Note that, to conserve silicon area, a SM is no fully populated crossbar, but provides a more constrained connectivity: A signal entering at pin $n \in \mathbb{N}$ at side $k \in \{N, S, E, W\}$ can only be routed to pin n at the sides $\{N, S, E, W\} \backslash \{k\}$.

While a routing architecture just based on SMs and wires of a single length (one logic block) could certainly work, its performance would be limited by the increasing number of slow switches between source and sink of a net: E.g., to connect a logic block in row 1, column 1, with a logic block at row 1, column 6,

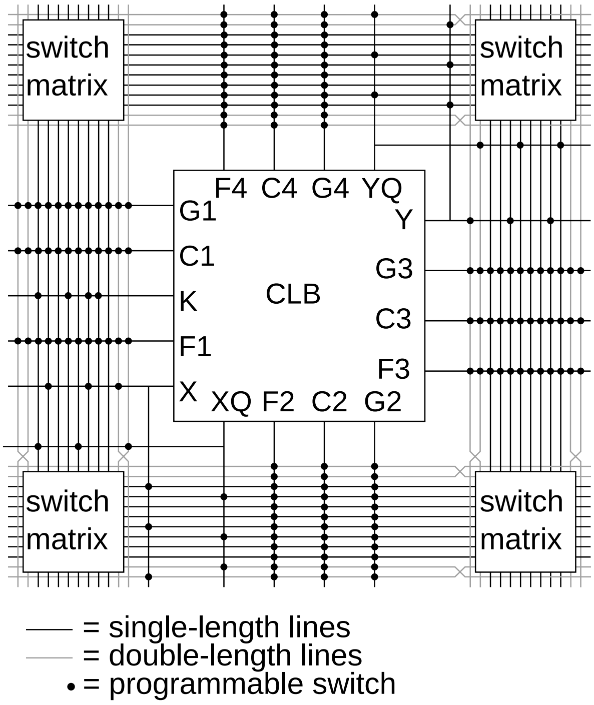


Figure 2.5: XC4000 single and double length lines

the signal would have to pass through five SMs, each with a resistance of 1-2 kΩ, and capacitance of 10-20 fF.

To counter this effect, XC4000 FPGAs use different length metal segments. *Single length lines* are only a single CLB long, while *double length lines* have a length of two CLBs. The result is shown in Figure 2.5. The CLB pins can be connected to the metal segments forming the routing network through individual programmable switches. The single length lines (shown in black) run between a pair of adjacent SMs. Double length lines (shown in grey), pass only through every *second* SM (see also Figure 2.5). While this limits their flexibility somewhat (e.g., the signal cannot turn around a corner at an arbitrary SM), it also decreases signal delay by halving the number of switches in the signal path.

For even longer distances, the XC4000 routing architecture provides *horizontal* and *vertical long lines* (HLL, VLL): Long metal segments running across half the height (VLL) and length (HLL) of the chip, unbroken by any switch matrix. For increased flexibility, two horizontal or vertical long lines may be interconnected (using programmable switches at the center of the chip) to route a signal along the entire length or height of the FPGA. Figure 2.6 shows how CLB pins can be connected to the long lines.

When a signal has to be routed to every CLB on the FPGA (e.g., clocks), *global long lines* may be used. While only four global long lines exist in the XC4000 series (also shown in Figure 2.6), they can supply a large number of sinks with very low skew.

Newer revisions such as the XC4000EX and XL extend this routing scheme

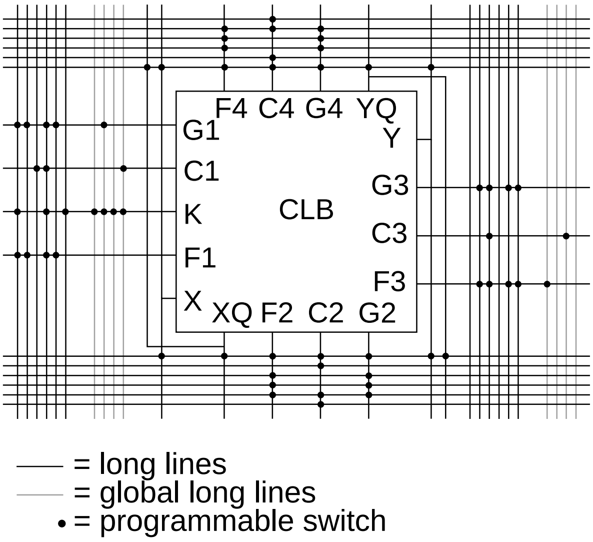


Figure 2.6: Connecting to long lines

with additional resources such as direct interconnections for very fast connectivity between adjacent CLBs, and quad and octal length lines for fast medium-range connections (an extension of the double length line concept). Furthermore, they also increase the actual number of metal segments available.

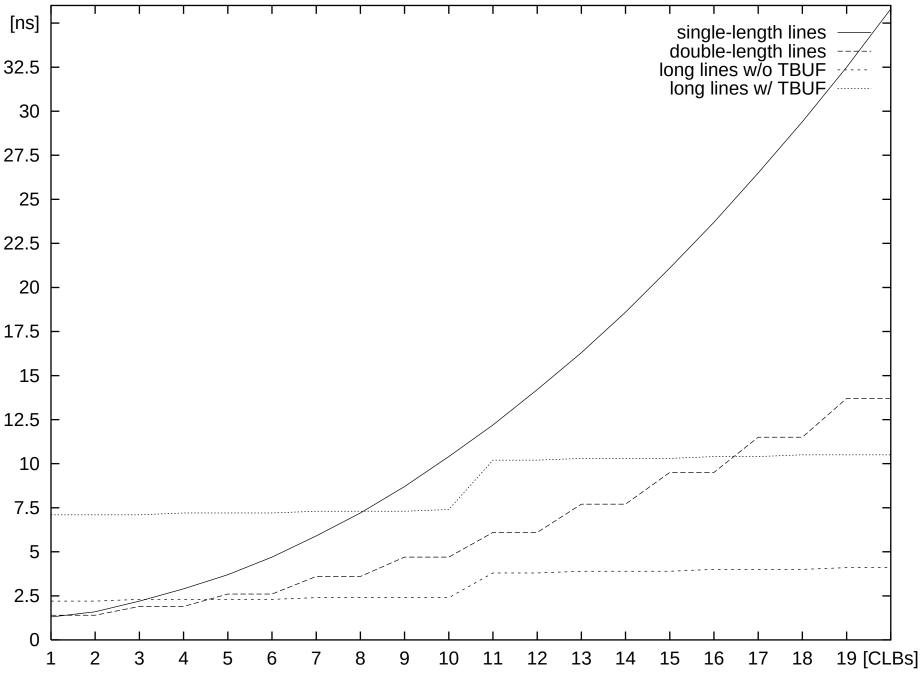


Figure 2.7: Signal delays on different routing resources

The three level routing (single, double, long lines) of the XC4000 leads to

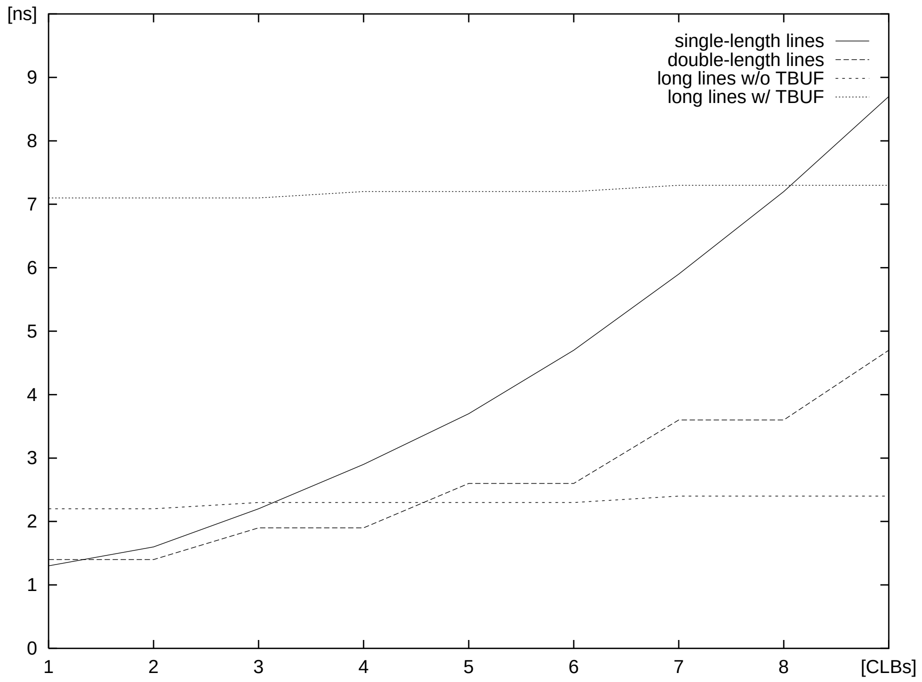


Figure 2.8: Enlarged section of Figure 2.7

the delays graphed in Figures 2.7 and 2.8[2]. The X-axis shows the total net load (fanouts) and the distance to the farthest CLB, the Y-axis the delay in ns. Note the reduced slope of the double length line vs. the single length line delay. Furthermore, long line delay is almost completely independent of routing distance and net load, and increases only when the programmable switches at the chip center are crossed (between CLBs 10 and 11 on the XC4010-5). Some of the horizontal long lines have connections to TBUF outputs for implementing on-chip tri-state busses. The capacitive loading of these outputs slows these long lines down considerably.

Returning to Figure 2.5, note the placement of input and outputs around a CLB: While inputs can be routed into the CLB from any direction (the input pins of a LUT are interchangeable), the outputs have preferred directions. The X and XQ outputs most efficiently supply sinks located below and left of the CLB, while the Y and YQ outputs should be used for sinks above and right of the CLB.

2.2 Structured Design Entry

We concentrate on implementing regular datapaths using a regular on-chip layout, and will rely on traditional methods to handle irregular circuits such as controllers. Thus, we need to derive information describing the regular (bit-sliced) structure from a given circuit description. Since SDI handles only regular structures, our input format can afford to be specialized for this do-

---

[2] Delay times were experimentally determined on an XC4010-5.

main. So instead of extracting regularity information from a generic netlist (as in [OdHN87], [ChCh93], [YuWY93], [NaBK95]), or evaluating manually annotated regularity attributes (e.g., [ATTM94]), datapaths are entered into SDI in the form of interconnected parametrized modules that encapsulate the bit-sliced structure.

Explanation 1 A *module* is a sub-circuit, described at varying degrees of abstraction (behavioral, structural, layout). *Parametrized* modules do not have a static description, but rely on a static *template* combined with dynamic property-value assignments, to create an adapted description at instantiation time. *Regular* modules have a description that follows a consistent pattern: A module consists of instances (copies with actual parameters) of masters (templates with formal parameters). A bit-sliced structure is a special case of a regular module that is also a regular array (Section 3.2.1). ■

To measure the regularity of a given structure, the *regularity index* of [Leng86] can be applied. It is defined as the ratio of actual components (instances) to the number of original templates (masters). The regularity index of common circuits is currently in the range 10 . . . 100.

In the context of this work, we will concentrate on the processing of automatically generated (Section 2.4) bit-sliced parametrized modules, just called modules for brevity. While SDI can also process non-bit-sliced modules, they will be treated as "black-boxes" and not optimized further (Explanation 5).

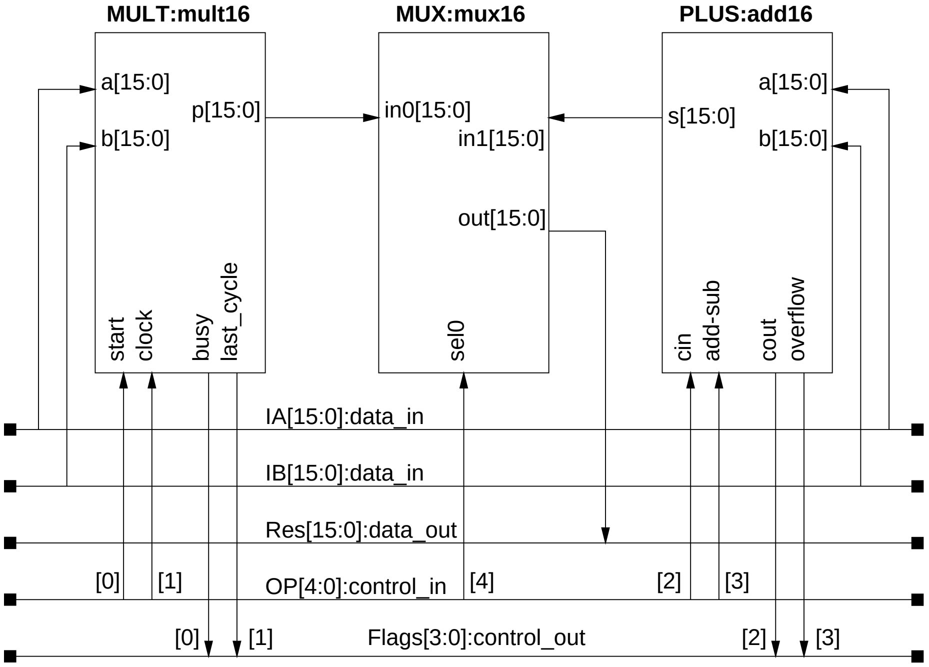


Figure 2.9: Example datapath providing simple arithmetic functions

Designs are expressed in the SDI netlist format *SNF*, which is a textual netlist of module declarations, module instantiations, and interconnections. It also associates values with module parameters such as bus widths, data types, and optimization requests (speed vs. area). Furthermore, SNF allows to clearly differentiate between data and control signals. A typical example for a datapath expressible in SNF is the fragment shown in Figure 2.9. It calculates either the sum, the difference, or the product of two data operands IA and IB and puts the results on an data output bus Res. The specific operation performed and additional information (e.g., carry initialization) is entered through the control input bus OP. Status information monitoring the progress of the datapath (e.g., busy) or arithmetic flags are made available to the outside on the control output bus Flags. A detailed description of the format can be found in [Putz95a].

In this manner, information on the logical structure of the circuit is passed down from design entry to the placement and routing tools, and does not have to be reconstructed from an unstructured, possibly flattened netlist. The preserved regular structure can then be used to optimize the circuit as well as the internal operation of the electronic design automation (EDA) algorithms.

2.3 Target Topology

2.3.1 Datapath Topology

In holding with the classical approaches (e.g., [CNSD90], [BeGr93], [Raba85], [GrPe97], [Shro82]), SDI assumes that a datapath is laid out as a linear placement of modules, with each module consisting of stacks of bit-slices (Figure 2.10). Within the datapath, all modules usually have a consistent stacking direction for the bit-slices from least-significant bit (LSB) to most-significant bit (MSB). However, this direction may change locally to accomodate *folded* modules that are too tall for the placement area (Figure 2.11). In holding with the traditional approach, control and data flows are generally orthogonal to each other.

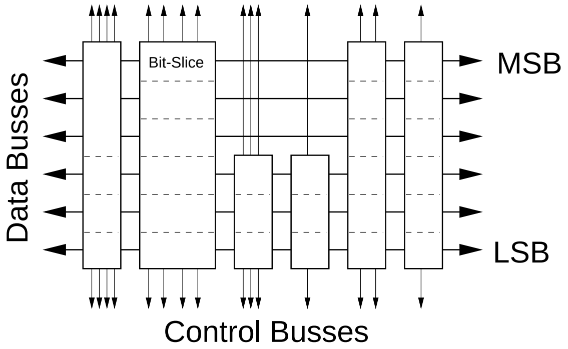


Figure 2.10: Classic datapath structures

When describing the layout and topology of datapaths or modules, it is useful to differentiate between an unplaced datapath *logical circuit*, and its placed *physical layout*. Placement is differentiated into the *geometrical place-*

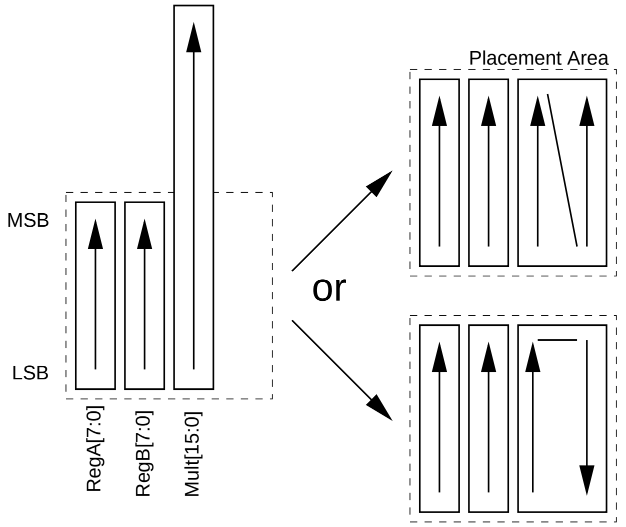


Figure 2.11: Folding modules jutting out of the placement area

ment of a layout at an explicitly specified location on the die, and the *topological placement* of a layout relative (“above”, “left-of”, etc.) to another.

To this end, we introduce the quantities of *width*, *height*, and *length*. Figure 2.12 illustrates their relationship.

Explanation 2 The logical *width* of a datapath or module circuit is the maximum number of differently significant bits processed in parallel. *Height* and *length* refer to the size of the bounding box of the physical datapath layout, with height being measured in the direction of width. Height and length are expressed in technology-dependent units, with logic blocks (LB) being most appropriate for FPGAs. ■

The restriction “differently significant” in the explanation of width serves to more clearly describe the logical structure of arithmetic units in the datapath. For example, while a 32-bit adder with operands A[31:0] and B[31:0] has 64 input bits, each two of these input bits will have the same significance in the input words (e.g., A[0] and B[0], A[1] and B[1], etc.). Since the data flows in the datapath should be correctly aligned (usually at the LSB, as in Figure 2.10), the physical datapath layout is directly influenced by this logical characteristic.

In full-custom and macro-cell technologies, cell placement and routing are only constrained by design rules. Even when targetting sea-of-gates masters, the basic blocks (e.g., transistors or transistor pairs) on such masters are often highly symmetrical (often 4-way, sometimes even 8-way [GrSt94]). Thus the orientation of the datapath, and the directions for bit-slice stacking and signal flow area are usually only determined by external pin-constraints and independent of the base substrate.

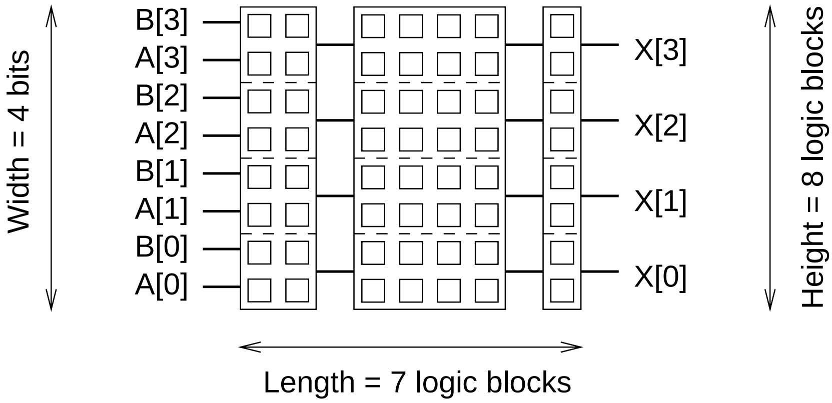


Figure 2.12: Extents of datapaths and modules

Suitable FPGA Architectures

In stark contrast, any kind of layout on FPGAs has to take the fixed underlying architecture of the FPGA chip into account. The freedom for specifying an FPGA architecture is highly limited by the large amount of chip area consumed for implementing the programming facilities for logic and routing (see [BFRV92]). For this reason, symmetrical blocks are quite rare for FPGAs and do not exceed 4-way symmetry even when they are actually implemented [Atme94], [Algo92].

When designing any kind of physical design software for FPGAs, the composition of the logic blocks and the structure of the routing network have to be considered carefully.

Since SDI aims at laying out regular circuits in a regular manner, it is directly influenced by the regularity of the underlying FPGA architecture. Thus, the architecture most suitable for our application would have highly regular and homogeneous logic blocks and routing resources.

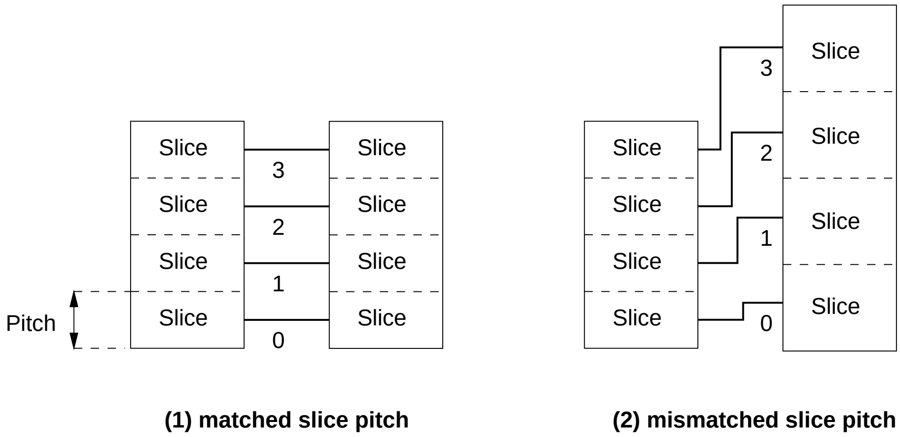
SDI does not depend on the core elements of the logic blocks (K-LUTs, MUXes). However, it relies on a regular logic-block structure. Single-output blocks, such as those found on Actel ACT [Acte95a], Xilinx XC6200 [Xili96b] are ideally suited to SDI. Coarse-grained multi-output blocks, such as AT&T ORCA [ATTM95], and Xilinx XC4000 CLBs, can be decomposed into regular single-output blocks (Section 2.7.3). However, this becomes difficult for fine-grained multi-output blocks such as QuickLogic pASIC2 [Quic95] or Atmel AT6000 [Atme94]. Here, a library-based technology mapping could assemble the multi-output blocks, each of which would then be placed as a unit. Architectures that directly couple logic-block functionality to placement locations, e.g., Pilkington/Motorola TS-series tiles [Pmel96], would have to carefully trade-off placement regularity vs. wasted space due to partially filled tiles. They could be better exploited using hierarchical clustering placement methods.

FPGAs with a row-oriented topology often lack the routing resources for two orthogonal signal flows. Architectures with sets of tightly interconnected logic-blocks and hierarchical routing, such as the Altera FLEX10K series [Alte95], are better exploited by clustering placement methods [CuoL96] than by our regular-array approach. Applying our methods to chips with routing through a universal interconnection matrix would not yield much of an improvement: these FPGAs dedicate much of their die area to the cross-bar switch, which provides an almost constant-delay routing independent of circuit placement [ACCK96]. However, with an increasing number of logic blocks the size of the cross-bar (even if not fully populated) consumes too much area to be practical for all but highly specialized applications. Thus, our requirements are currently best fulfilled by matrix-structured FPGAs.

We chose to target the Xilinx XC4000 family, since it provides a homogeneous routing structure (matrix) as well as logic-blocks that are easily regularized. The recent XC6200 family, which was not available in time to be included in our research, would also fulfill the criteria. Due to the fine logic-block granularity, it would also be less dependent on the compaction step (Section 2.6).

Bit-Slice Pitch

As with full-custom and macro-cell-based datapaths, a fourth quantity becomes relevant when aiming for a regular layout. The *pitch* of a bit-slice is the height of a single slice (Figure 2.13). For a regular layout, all bit-slices in modules in a datapath should balance the pitch. Otherwise, routing becomes more difficult and congestion increases (Figure 2.13.(2)). This becomes especially critical on FPGAs, where the channel width is fixed on the die and signals are delayed by the programmable routing elements.



(1) matched slice pitch

(2) mismatched slice pitch

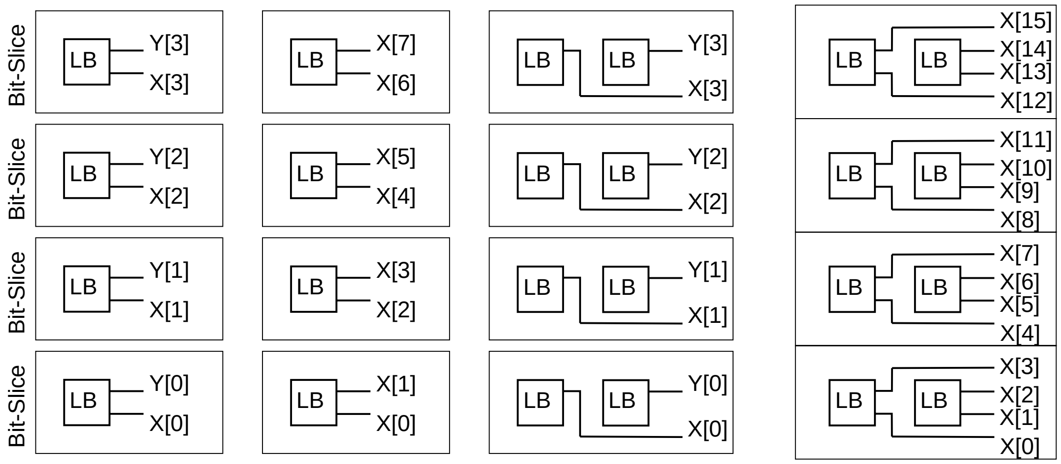
Figure 2.13: Matched and mismatched bit-slice pitch

However, for FPGAs, it is useful to extend the classical concepts of bit-slice and pitch to accomodate the coarse granularity of logic blocks compared to full-custom/macro-cell, or even gate arrays and sea-of-gates. Since LBs often have multiple outputs, even a slice having only the height of single LB can conceivably compute multiple different functions at once. Thus, this minimal slice can process more than a single bit.

In a similar manner, we extend the concept of pitch from the simple height measure of a bit-slice to the number of bits with different significances actually processed per LB height (*BPLB*)[3].

Explanation 3 *Bits per logic-block-height* (BPLB) is a bit-slice specific quantity. It is defined as the number of outputs with different bit significances on a given bit-slice divided by the height of this bit-slice in logic blocks. ■

Figure 2.14 gives some examples for different BPLB values. It assumes LBs with two independent inputs (as on the XC4000, Section 2.1.1). E.g., in Figure 2.14.a each bit-slice is one LB high and has two outputs, but both with the same significance. Since multiple outputs with the same bit-significance do not influence the BPLB, (a) processes 1 BPLB. Furthermore, with a long bit-slice that contains multiple LBs in a single row, the BPLB value can exceed the number of independent outputs of a LB (Figure 2.14.d). As with pitch, BPLB should stay as constant as possible across an entire datapath for maximal performance.



(a) 1 BPLB (b) 2 BPLB (c) 1 BPLB (d) 4 BPLB

Figure 2.14: Examples for BPLB values

Applying these concepts to our concrete target FPGA family, the Xilinx XC4000 series of chips, we determine the following specifics: The dataflow of the hard-carry logic (Section 2.1) requires a vertical stacking of bit-slices inside a module, with the LSB at the bottom and the MSB at the top. The modules themselves will be placed horizontally, leading to a topology as shown in Figure 2.15. The orthogonal signal flows can be efficiently implemented in this arrangement: The (possibly high fan-out) control signals are efficiently routed on VLLs, while data signals mainly use the various horizontal routing resources. By using VLLs, control signals can be distributed with minimum

---

[3] Previous papers refer to this quantity as bits-per-CLB *BPC*

skew along the entire height of the chip, making them easily accessible to all bit-slices in a module.

2.3.2 Chip Topology

After establishing the topology for the datapath itself, we now have to determine where to place it on the chip. This is most critical for FPGA-based systems with a fixed pin-out (like the Sparxil-processor [KoGo94]). When the pin-out of the FPGA can remain variable (e.g., when a system-wide programmable routing network based on FPICs is employed), the chip pin-out itself can be adjusted to accomodate a floating placement of the datapath section of the whole circuit.

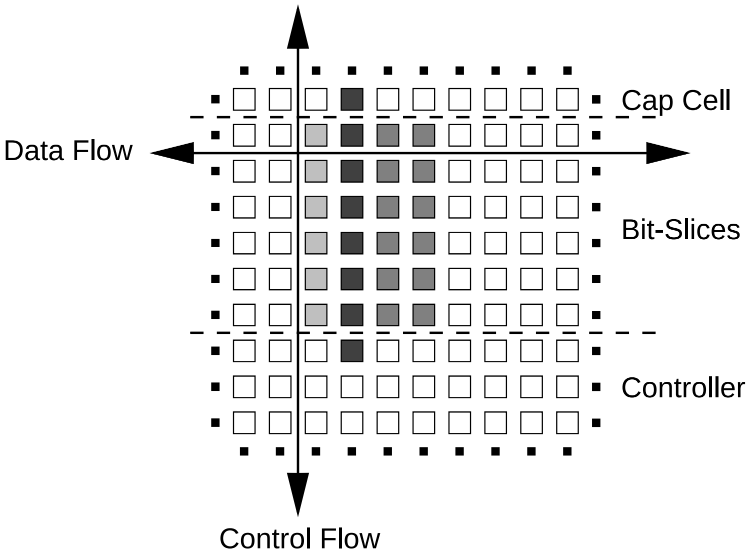


Figure 2.15: On-chip topology

The chip topology targeted by SDI is characterized by a fixed threepartite layout (Figure 2.15). The large middle section holds the regular part of the datapath. This part consists of the horizontal arrangement of modules, each composed of vertically stacked bit-slices. The area below the datapath is intended to hold the controller, whose irregular logic is not processed by SDI. A small area above the regular section can hold irregularities in the modules such as *cap cells*, e.g., the processing of overflow and carry bits in a signed adder. Such irregularities may also reach below the datapath baseline into the controller section, e.g., the initialization of the carry chain below the LSB of the adder mentioned above. After datapath placement, all of the remaining chip area may be used for implementing the controller through conventional methods (the XACT tool suite for Xilinx FPGAs).

The dimensions of the three areas are application-specific and must be manually designated by the user. In the case of FPGAs with a fixed pin-out on a printed circuit board (PCB), the area heights are primarily determined by the interconnection pattern on the PCB, which in turn depends on the width requirements for external busses.

The SDI chip topology can accommodate multiple datapaths on a single FPGA, arranged horizontally across the chip, as long as their total length remains smaller than the FPGA length. To allow access to the external pads and communication between otherwise independent datapaths, horizontal long-distance connections, which are available in many FPGA architectures, can be used. For XC4000 FPGAs, this suggests the use of HLLs. They are usually not allocated for simple inter-module communication (HLLs are inefficient over short distances), but are very effective for chip-wide communication (e.g., access to the external pads). The XC4000 also has a more unusual feature in that each of the HLLs can be split by switching off a pass-transistor in the middle. This can be employed to good advantage for the implementation of two completely independent datapaths, with each using the full number of HLLs per channel in its chip half.

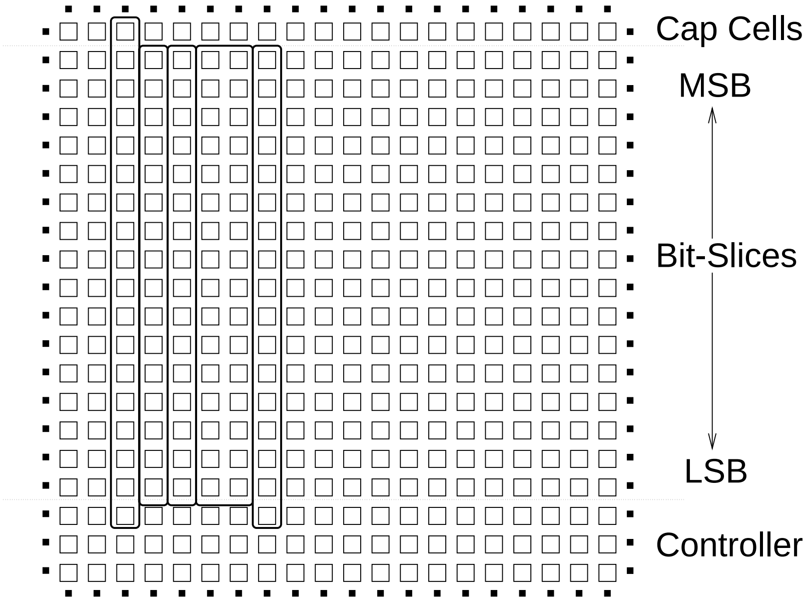


Figure 2.16: SDI topology as used on the Sparxil processor

This base topology was already considered for the hardware design and PCB layout of the Sparxil processor system. The 20x20 matrix of the XC4010 FPGAs (Figure 2.16) is vertically organized with one CLB as cap cell, 16 CLBs for the datapath, and 3 CLBs initially reserved for the controller. The pads for the data busses are locked on the left and right sides of the FPGAs, while control signals are locked to the top and bottom sides. Thus, the PCB layout mirrors the on-chip layout.

In this manner, the topology allows the area-efficient implementation of 32-bit operations in modules that are only a single CLB long. By fully using the datapath area height of 16 CLBs, and processing 2 BPLB (possible with XC4000 CLBs, Section 2.1.1), 32-bit functions can be computed directly.

2.4 Module Generation

The parametrized module generator Paramog [Ditt95], [Sade95] is responsible for generating individual modules in SDI. While these modules are already composed in a regular manner by vertically stacking bit-slices, it is useful to introduce another level of regularity to capture relations between bit-slices.

Explanation 4 A *master-slice* (MS) is an independent sub-circuit that is instantiated one or more times into a module as bit-slice. A *zone* is a contiguous part of a module which contains only bit-slices that are instances of a single master-slice. Multiple instances of the master-slice are stacked vertically. ■

The relationship between bit-slices (also called slices), master-slices, and zones is shown in Figure 2.17.

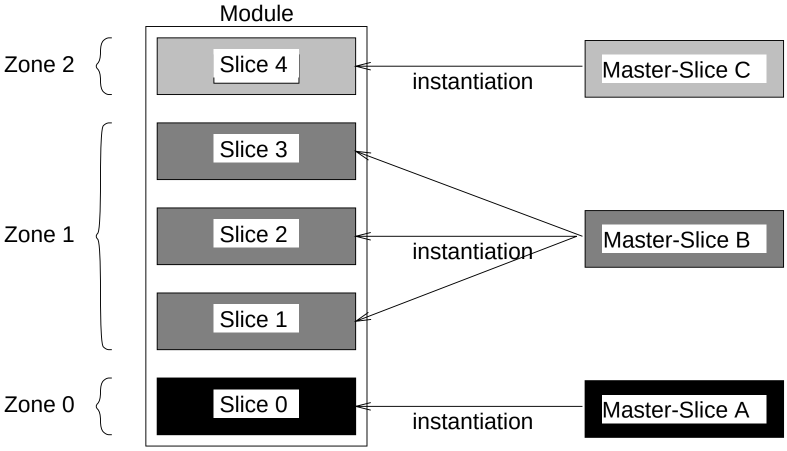


Figure 2.17: Regular structure of a module

Paramog currently supports the functions shown in Table 2.1. New modules can be added by building on the general framework of C++ classes.

For each of these modules, Paramog can offer various layout alternatives with different BPLB values and module heights, folding modules if the height of the placement area is exceeded (Figure 2.18).

Among the parameters supported by Paramog are:

- function
- operands with widths and datatypes
- result width and datatype
- optimization preference (delay vs. area)
- BPLB value for the module slices
- maximum module height in logic blocks

Class	Functions	Comment
Logic	NOT, AND, NAND, OR, NOR, XOR, XNOR, MUX	
Shift and Rotate	LSHIFTA, RSHIFTA	arithmetic shifts
	LSHIFTL, RSHIFTL	logical shifts
	LROT, RROT	
Storage	REG, RAM, ROM	
	CONST	literals
Arithmetic	ABS	
	COMPL1	one's complement
	COMPL2	two's complement
	INCDEC, ADDSUB, MULT	
Comparison	COMPARE	
Counter	COUNT	

Table 2.1: SDI module library overview

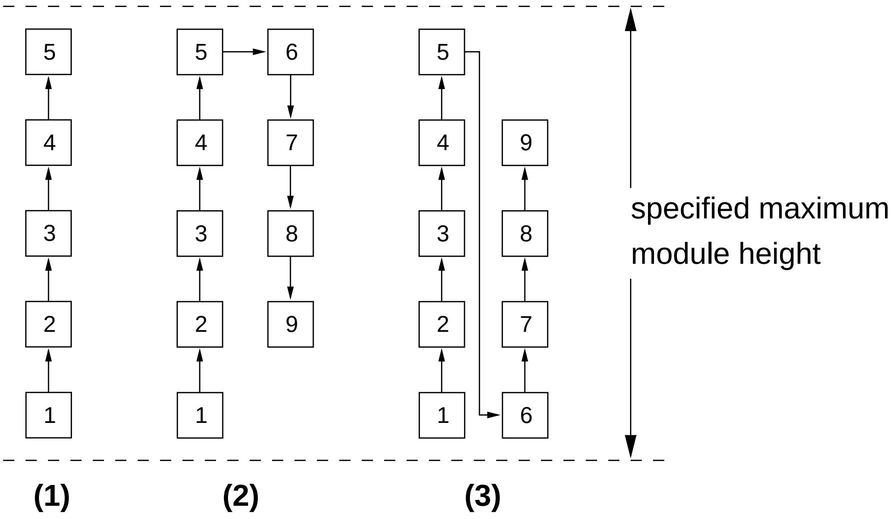


Figure 2.18: Examples for module layouts: (1) unfolded, (2) alternate folding, (3) unidirectional folding

Paramog responds with a set of topology alternatives, each having different BPLB values or delay vs. area tradeoffs. On request, Paramog generates an actual layout for a selected topology alternative. These module layouts are already placed and routed. FPGA specifics such as dedicated hardware and different routing resources are considered: for the XC4000 chips, Paramog can employ the hard-carry logic as well as on-chip memory blocks and distinguishes between single-length, double-length and long lines for module-internal routing.

However, it operates only at the layout and structural netlist levels. Unlike more complex systems like LORTGEN [BrMR94], Paramog does not have an extensive internal knowledge base and does not evaluate the quality of the proposed implementations or perform architectural optimizations. Also, it does not perform data type propagation as in X-BLOX [Xili94a]. To a certain degree, these tasks are handled by the floorplanning component of SDI (Section 2.5).

2.5 Module Selection and Floorplanning

Since Paramog offers various layout alternatives for a given module, a concrete layout has to be selected for each module instance. For an optimal choice, the layout alternatives for all modules in the datapath have to be considered (to equalize their BPLB values, Explanation 3). Furthermore, layout selection and module placement are also interdependent. E.g., assume the situation shown in Figure 2.19, in which it was advantageous to support two different BPLB values in a single datapath. This might occur, when certain tightly interconnected modules on the critical path obtain their best performance with topologies different from the rest of the datapath. Due to the loose coupling with the other modules and the local scope, the routing delay penalty associated with BPLB matching might be acceptable here.

The FloorPlanner component [Putz95a] [Bode97] of SDI is reponsible for simultaneously evaluating module layout and placement alternatives. Initially, it obtains the available layout topologies for all module instances in the datapath by querying the generators. Then, it begins to linearly place instances in the regular region of the FPGA. However, instead of simply generating different orderings of module instances and evaluating them in terms of wire length (as in the classical linear placement problem [Sung83], [SaCh94]), different concrete layouts (as offered by Paramog) are selected and evaluated in context of the linear placement.

FloorPlanner heuristics are based on a fuzzy-controlled [NaKK94] genetic algorithm (GA) [Gold89], and thus consider multiple different layout choices and placements simultaneously. The fuzzy-controller is responsible for dynamically adapting the parameters of the genetic algorithm (e.g., population size, mutation rate, etc.) in order to improve the GA's performance but prevent premature convergence to a local optimum.

SDI can afford to use this computationally expensive, but powerful algo-

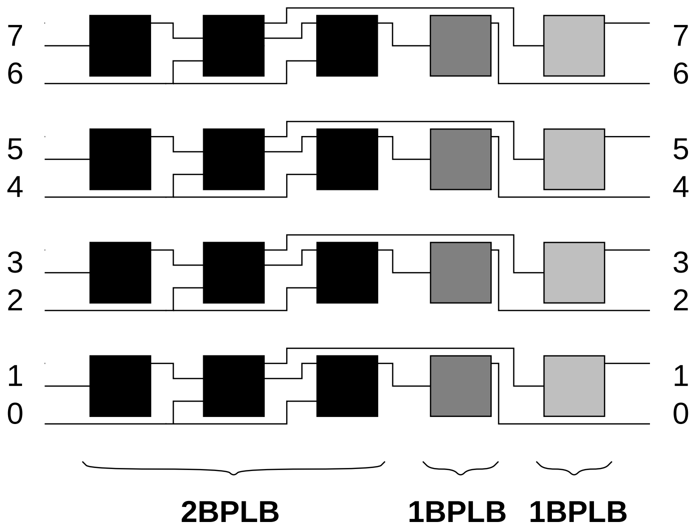


Figure 2.19: Multiple BPLB values in a single datapath

rithm: The solution space consists only of placing complete module instances (usually less than two dozens on current FPGAs) and selecting their configuration (around three to four after BPLB normalization). This problem size is far more manageable than that of an algorithm processing a netlist of basic gates (todays FPGAs easily have complexities of 10,000s of gates). The exploitation of the regular structure allows the efficient use of a GA. Alternatively, other approaches base on integer linear programs, or simulated annealing could be employed.

The fitness function of FloorPlanner mainly considers the following factors in its search for a high-quality chip layout: the uniformity of the instance BPLB values, the wire length and the compactibility of adjacent modules (Section 2.6). The fitness is not only computed as a simple weighted sum, which could lead to a sub-optimal solution [EsKu96], but also by true multi-criteria evaluation.

2.6 Compaction

For traditional standard-cell and gate-array technologies, the process of laying out the datapath would end here, and routing could commence. However, the coarse block-granularity of FPGA logic blocks as compared to standard-cells and sea-of-gates transistors suggests an additional *compaction* phase to reduce area and delay inefficiencies (Section 2.6.1). Area and delay reductions are achieved by locally applying classical logic synthesis and technology mapping algorithms. However, this compaction step should not disrupt the regular circuit structure (Sections 2.6.3 and 2.6.4). The steps of this phase are outlined in Figure 2.20.

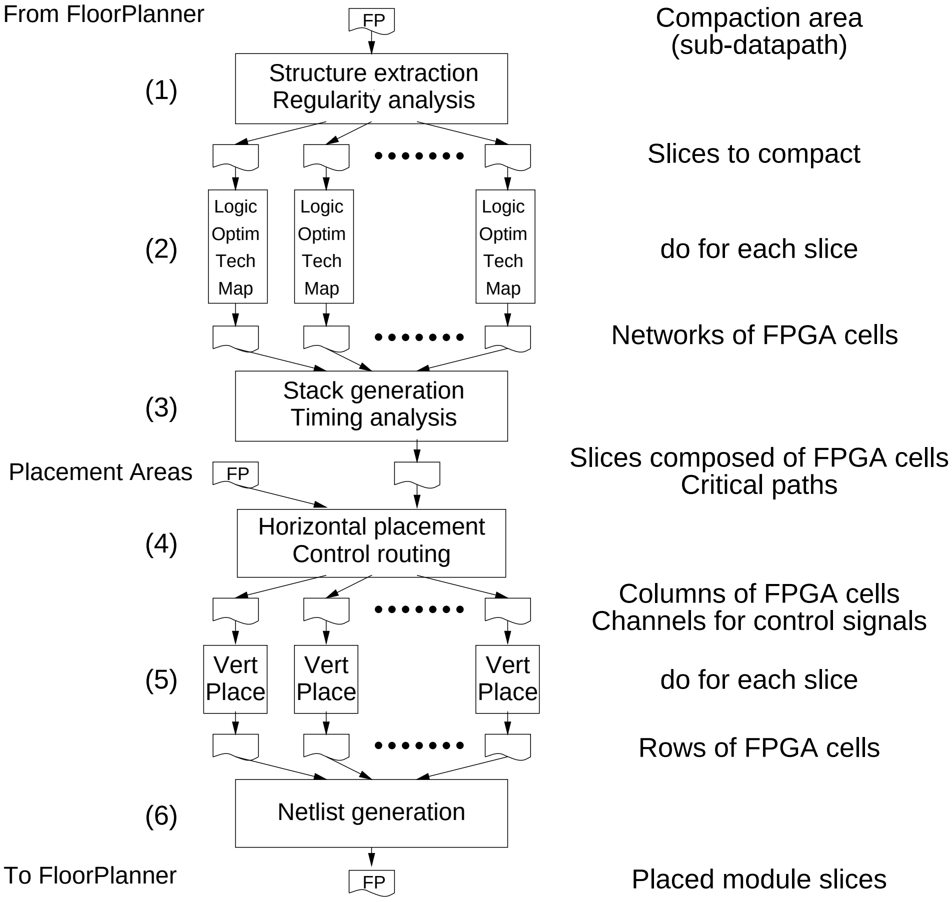


Figure 2.20: Compaction

2.6.1 The Need for Compaction

To illustrate the need for compaction, consider the following premises: The datapath layout is assumed to consist of a linear placement of regularly generated modules. Since a module is always at least one logic block long (most module generators, Paramog included, cannot generate module fragments), partially utilized blocks inside each module waste area and speed. The size of the wasted area and the loss in speed increase with the logic capacity of a single FPGA logic block and the number of modules in the datapath.

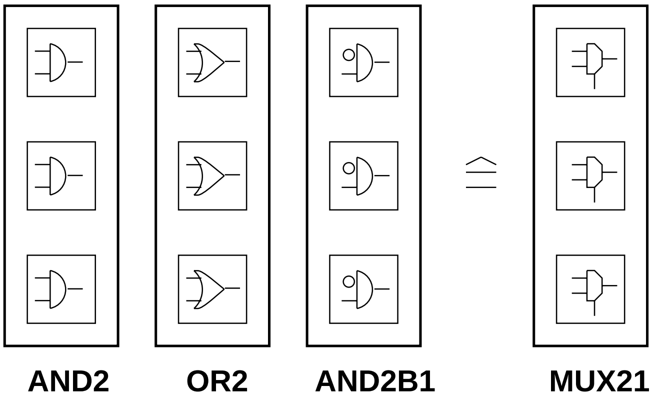


Figure 2.21: Wasted space in a module-based layout

Figure 2.21 is an extreme example for such a scenario: The 3-bit wide datapath contains three regular modules AND2, OR2, and AND2B1, implementing the functionality of a 3-bit wide 2-1 multiplexer. However, even assuming fine-grained logic blocks on the FPGA (e.g., Actel ACT logic modules, Atmel AT6000, or Xilinx XC6200 cells), the function MUX21 can be implemented in a single logic block per bit. Thus, this sample datapath wastes 2/3 of its area (length of 3 LBs vs. 1 LB) and only runs at 1/2 the speed (2 levels vs. 1 level of LB delay) of the single block solution. This situation becomes worse with coarser-grained blocks such as the K-LUTs found, e.g., in Xilinx XC3000/XC4000 or AT&T ORCA FPGAs.

The compaction process merges adjacent modules to better utilize the logic blocks, but leaves the regular bit-sliced structure of the modules as well as the general topology of the datapath intact.

2.6.2 Soft- and Hard-macros

Compaction is not performed indiscriminately on all modules in the datapath. We distinguish between soft and hard-macros.

Explanation 5 A *soft-macro* is a module that contains only simple combinational and sequential logic that is tractable using conventional logic synthesis tools, and which has a very simple internal topology. A *hard-macro* is a module that uses chip-specific features, like on-chip memory or carry-chains, that are intractable by conventional logic synthesis, or has a highly irregular or very complex internal structure laid out carefully to take maximum advantage of the FPGA block and routing topologies. ■

It is the responsibility of the module generators to declare a generated macro either hard or soft. In order to avoid a performance deterioration for hard-macros, they are not compacted, but passed unchanged. However, hard-macros serve the important role of marking boundaries between sets of soft-macros to be compacted separately (Section 2.6.3).

2.6.3 Preserving Module Placement

Compaction respects the original floorplan of the datapath in that the ordering of hard-macros and regions of soft-macros in the linear placement remains constant. In this manner, we avoid increases in critical path routing delay. These could be caused by inadvertently merging logic implemented in two widely spaced regions of the datapath. Note that this does not preclude optimization at the architectural level *before* employing SDI. However, once SDI is engaged, replicated logic is only eliminated if doing so does not disturb the module ordering of the linear floorplan.

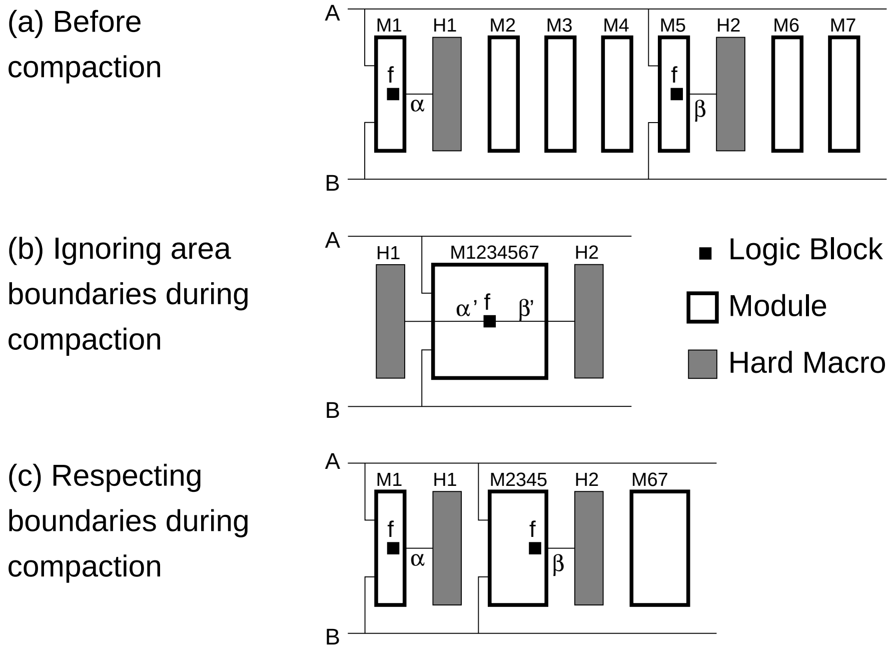


Figure 2.22: Hard-macros as boundaries of compaction areas

Figure 2.22 shows an example for this reasoning. The datapath consists of soft-macros $M_1, \ldots, M_7$ and hard-macros H_1, H_2. FloorPlanner has calculated the linear placement shown in Figure 2.22.a. The function f is realized twice at separate locations in the datapath: once in M_1 and once in M_5. Note the relative routing lengths α and β between the outputs of the LBs realizing f and their sinks in H_1 and H_2. Consider the layout that would be obtained when foregoing the original floorplan during compaction (Figure 2.22.b): All soft-macros in the datapath have been compacted together, yielding a combined module $M_{1234567}$. While this compacted module has reduced area (and most likely delay) as compared to the sums of its original constituent modules, the wire lengths have degraded: Even when assuming an optimal placement, with the LB calculating f in the middle between its two sinks, the post-compaction wire lengths α' and β' are most likely still increased over their pre-compaction equivalents, slowing down the whole datapath.

When the boundaries marked by the hard-macros are respected (Figure 2.22.c), area is traded for speed: Compaction occurs now separately for the sets of soft-macros $\{M_1\}$, $\{M_2, \ldots, M_5\}$ and $\{M_6, M_7\}$. The resultant compacted modules M_1, M_{2345}, and M_{67} are larger than $M_{1234567}$, but the wire lengths do not increase over their pre-compaction values.

2.6.4 Extracting and Exploiting Regularity

After determining the separate sets of soft-macros to be compacted, each of these sets is now processed separately (potentially in parallel). As the first step, the regular bit-sliced structure across all modules in a set is extracted to reduce run-times for the following compaction steps.

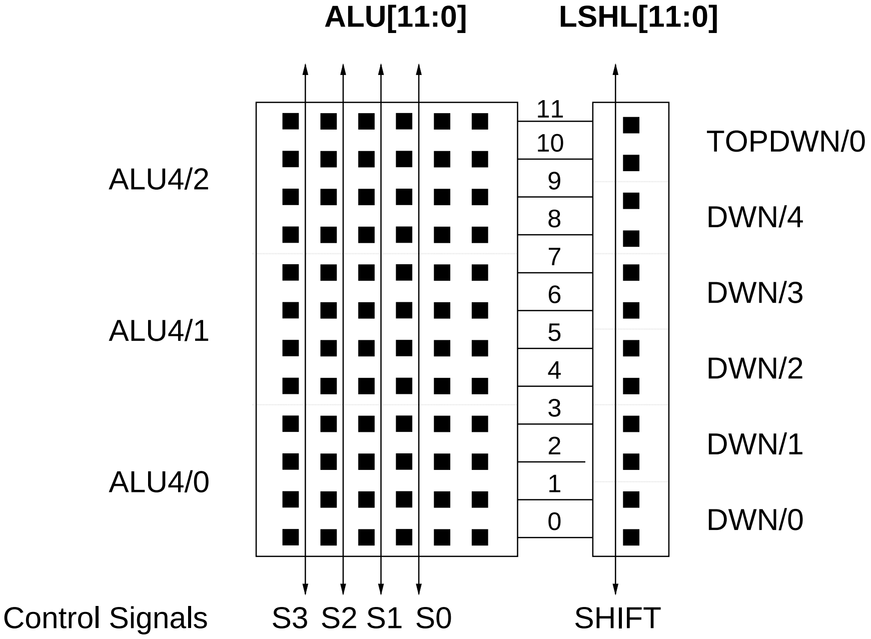


Figure 2.23: Sample datapath segment

Figure 2.23 shows a sample datapath segment consisting of two soft-macros, a 12-bit ALU connected to a 12-bit logical-shift-left register. The ALU has a single zone of logic which is composed by stacking the three instances ALU4/0, ALU4/1, and ALU4/2 of the master-slice ALU4. Each of the slices is 4 LB high and processes 4 differently significant bits, yielding a value of 1 BPLB. The shift-register has two zones of logic. The bottom zone contains 5 instances of the master-slice DWN, each providing a simple downward-shifting functionality. The top zone contains a single instance of the master-slice TOPDWN, which also shifts in a zero into its MSB when the shifter is activated by the SHIFT control signal. Each of the shifter slices also has a 1 BPLB topology (height of 2 LB, processing 2 differently significant bits).

While the partitioning of the datapath into independent module sets already leads to smaller sub-problems, their size can be reduced even further by considering the bit-sliced regularity in datapath modules: Since the compaction operation aims to preserve the bit-sliced structure and homogeneous pitch of the datapath, it can only be performed *horizontally*. Thus, the datapath segment under compaction can now be searched horizontally *across* soft-macro boundaries for areas of recurring logic that are replicated vertically. The areas obtained in this manner will form the base for the master-slices of the compacted modules.

E.g., the datapath segment in Figure 2.23 has two such areas: The first one contains a single instance of the master-slice ALU4, and two instances of the master-slice DWN. It occurs stacked twice at the bottom of the datapath segment. The second area contains a single instance each of the master-slices ALU4, DWN, and TOPDWN. It occurs once at the top of the datapath segment. We refine our terminology to better describe these relationships:

Explanation 6 An *h-zone* is obtained by collecting logic blocks along a horizontal scanline running across the layout. Since this scanline crosses module boundaries, the h-zone can include logic blocks originating in the master-slices of multiple horizontally adjacent modules. Thus, the h-zone can be described as a set of its constituent master-slices. For readability reasons, information about interconnections will be omitted at this level (but see Section 2.7.2). A *v-zone* is a vertical stacking of bit-slices obtained by instantiating an h-zone one or more times. A v-zone is described with a tuple (hz, n), where hz is an h-zone and n is the number of times it is iterated.[4] A *stack* $(v_1, v_2, ..., v_n)$ is an upward sequence of v-zones. ■

Figure 2.24 illustrates the relationship between h-zones, v-zones, and a stack. Note that a v-zone is a refinement of the zone concept (Explanation 4). Applied to the example of Figure 2.23, we obtain the h-zones (ALU4, DWN, DWN) and (ALU4, DWN, TOPDWN). Since the first h-zone occurs twice, it forms the v-zone ((ALU4,DWN,DWN),2). The second h-zone occurs only once, leading to the v-zone ((ALU4, DWN, TOPDWN),1). The complete datapath segment can thus be described by the stack (((ALU4,DWN,DWN),2), ((ALU4, DWN, TOPDWN),1)).

Further compaction operations are now performed once for each occurence of the h-zones, called *merged master-slice*, and iterated as required by the stack. Since most soft-macros have few different h-zones (the usual maximum is around 3), but with relatively large replication counts (8 and 16 are common

---

[4] For improved consistency, this usage of h-zone and v-zone has been changed as compared to [Sade95], [Ditt95].

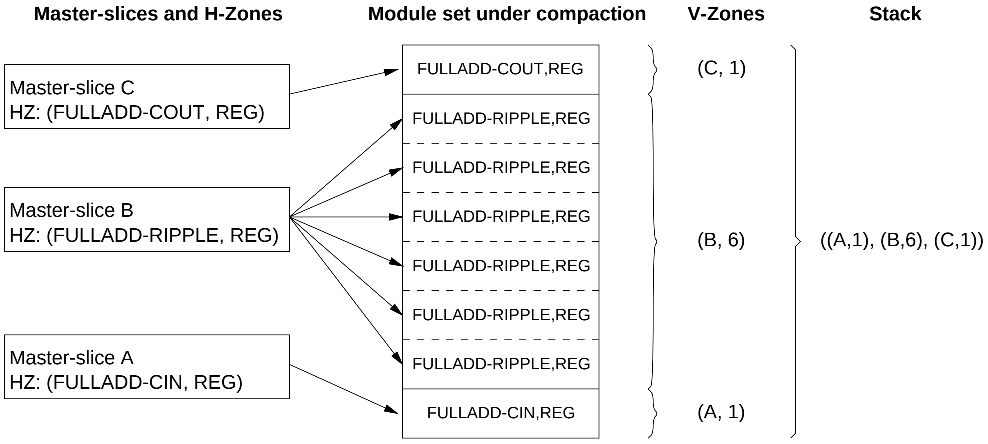


Figure 2.24: H-zones and v-zones in a stack

values), the problem size can often be reduced by an order of magnitude. In the example in Figure 2.24, the savings are less impressive: the logic of the h-zone (ALU4,DWN,DWN) is only compacted once and the result is duplicated, yielding a reduction to 2/3s of the original problem size.

2.6.5 Logic Optimization and Mapping

The merged master-slices extracted in the previous step from the module set under compaction form the basis for the optimized master-slices of the compacted module. Since the boundaries between modules were dissolved during structure extraction, the following steps merge the logic of separate modules. Since this merging occured only horizontally, the basic sliced structure (stack) of the compacted module remains intact. Because the merged master-slices are independent of each other, they could also be processed in parallel at this step.

Conventional compaction steps used for full-custom and macro-cell technologies are based on 1-D or 2-D geometric operations (e.g., [Marp90]) under design rule constraints. In contrast, module compaction for FPGAs is achieved by applying classical logic synthesis and mapping algorithms to every merged master-slice. Each of the resulting optimized master-slices contains the functionality of all the originally separate master-slices in the h-zone, but with reduced area and delay. The optimized master-slices are then used as master-slices for the compacted module.

Since the general strategy is independent of the details of this step, it can directly profit from any research advances in optimization and mapping.

2.7 Microplacement

With the original regular placement lost, the mapped network for each compacted master-slice has to be re-placed in a manner consistent with the base topology (Section 2.3). This step of the strategy is termed *microplacement*, since it is performed separately for each compacted module in context of the initial linear floorplan. The context, usually provided by adjacent hard-macros or FPGA I/O pads, determines the locations of the inputs and outputs of the compacted module (Figure 2.25).

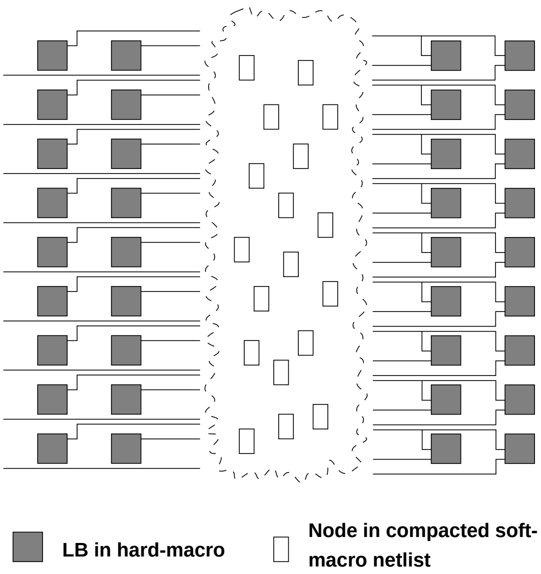


Figure 2.25: Floorplan context of an unplaced, compacted module

As before, the placement operations can be performed independently for each module, and possibly in parallel.

2.7.1 Congestion Handling

Except for the allocation of VLLs (Section 2.1.2) to vertical control signals, microplacement does not attempt to balance track density between routing channels to limit congestion. This is feasible for many circuits, because the pins on an LB are often interchangeable to a wide degree. Architectures like the XC4000, with all of the LUT inputs logically equivalent to each other, have the greatest flexibility in this regard. Figure 2.26.a shows the arrangement of pins around an XC4000 CLB (for a single LUT) and possible pin-assignments for the realization of the function $f(a, b) = a + b$ (in (b)).

Thus, the pin-assignment and routing steps following SDI (Section 2.8) can relieve congestion by locally swapping pins to less dense channels and adjusting the LUT configuration accordingly.

2.7.2 Pre-placement Activities

To maximize the performance of the compacted module, microplacement is timing-driven, minimizing the critical path delay. Thus, the critical paths through the compacted module have to be determined first.

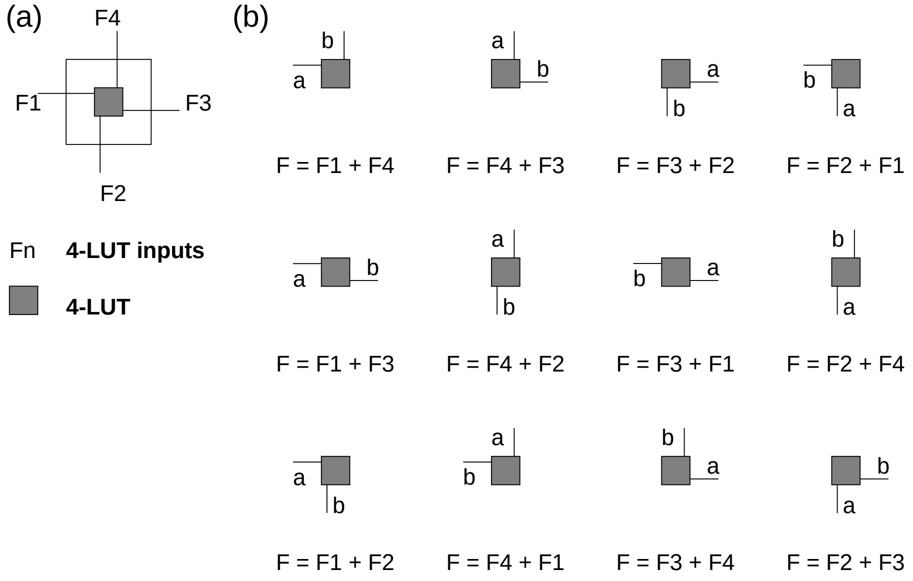


Figure 2.26: Pin-arrangement around a LUT (a) and equivalent pin-assignments with LUT configurations for $f(a, b) = a + b$ (b)

Initially, the compacted module has to be assembled from its optimized master-slices, which are instantiated according to the stack. Vertical inter-slice and control signals are then connected as specified by the module netlist.

The compacted module is then delay-traced, and the arrival and required times of inter-slice nets are back-annotated into their master-slices. For input ports, the arrival time becomes the latest time at which their signal arrives at an instance of the slice. For output ports, the required time becomes the earliest time the signal is required in an instance. Figure 2.27 shows an example of this process: A 3-bit ripple-carry adder is assembled from three instances of the FULLADD master-slice. When the entire module is delay-traced, the arrival and required times are set to the extreme values over all instances of the master-slice.

Next, each of the optimized master-slices is delay-traced separately, honoring the back-annotated constraints. The slice-local critical paths discovered by this method form the base for all further delay calculations.

While the extreme timing constraints obtained through back-annotation are not accurate enough to estimate the precise criticality of an inter-slice path through its master-slices, they can be used to determine paths that are critical at all (having timing slacks ≤ 0).

The final result is a list of critical paths for each optimized master-slice, sorted in order of ascending length. Note that multi-terminal nets are decomposed into one or more sequences $((n_1, n_2), (n_2, n_3), \ldots, (n_{m-1}, n_m))$ of *two-terminal nets* (TTN) . Each of the tuples (a, b) describes a TTN connecting a source a with a sink b. The timing-driven component of microplacement uses these sequences to minimize wire lengths on critical paths.

Another aspect that has to be resolved before placement is the placement

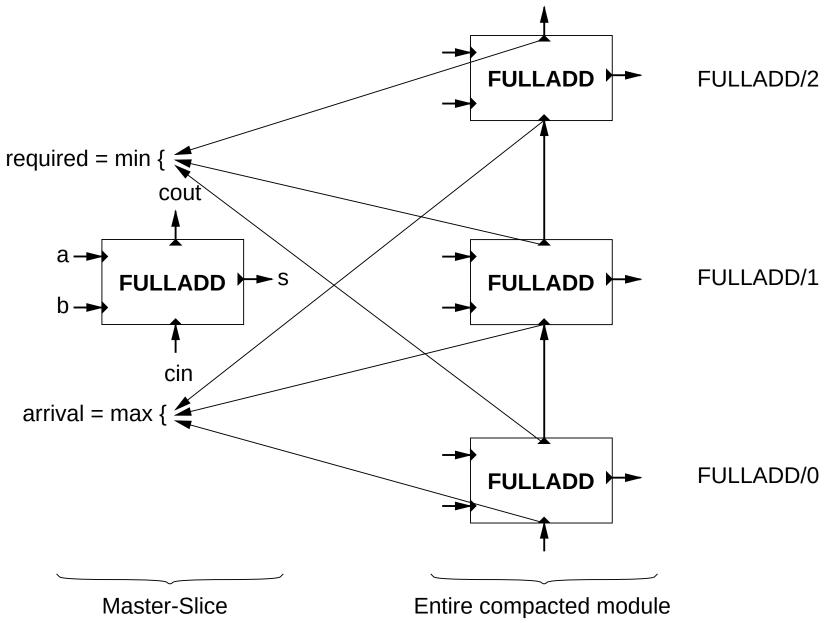


Figure 2.27: Back-annotation of timing for cin and cout into optimized master-slice

area for each slice. Due to the direct dependency between slice height and module pitch (measured in BPLB), the compactor with its datapath segment-local view cannot determine a height that ensures a homogeneous pitch across the entire datapath. Thus, this information has to be provided by the floorplanner, which can assess the best topology for the entire datapath.

In addition, the floorplanner also supplies data describing the geometric context around the compacted module, including the locations of external I/O ports, hard-macros, other compacted areas, or the FPGA pads.

2.7.3 Regularizing Logic Blocks

Microplacement aims at a regular bit-sliced layout of the compacted module. However, while recent FPGA architectures tend to have highly regular LBs (e.g., Altera FLEX 10K [Alte95], Xilinx XC5000/XC6200 [Xili96a] [Xili96b]), established chips, like the Xilinx XC4000 currently targeted by SDI, often have some pecularities.

Consider the XC4000 CLB (Section 2.1.1) shown in Figure 2.28.a. It contains two 4-LUTs named F and G, a 3-LUT named H and 2 flip-flops. The connections between these elements are determined by programmable multiplexers. However, since the H-block is not independent of the 4-LUTs (with regard to both in- and outputs), the CLB itself is inherently irregular. The usual approach to handle these irregularities is to match the nodes of the K-LUT-based netlist (with $K \leq 4$) together and pack matched nodes into the 3- and 4-LUTs in a CLB. Further processing then takes place at the CLB level [BaCM92], [MSBS91b].

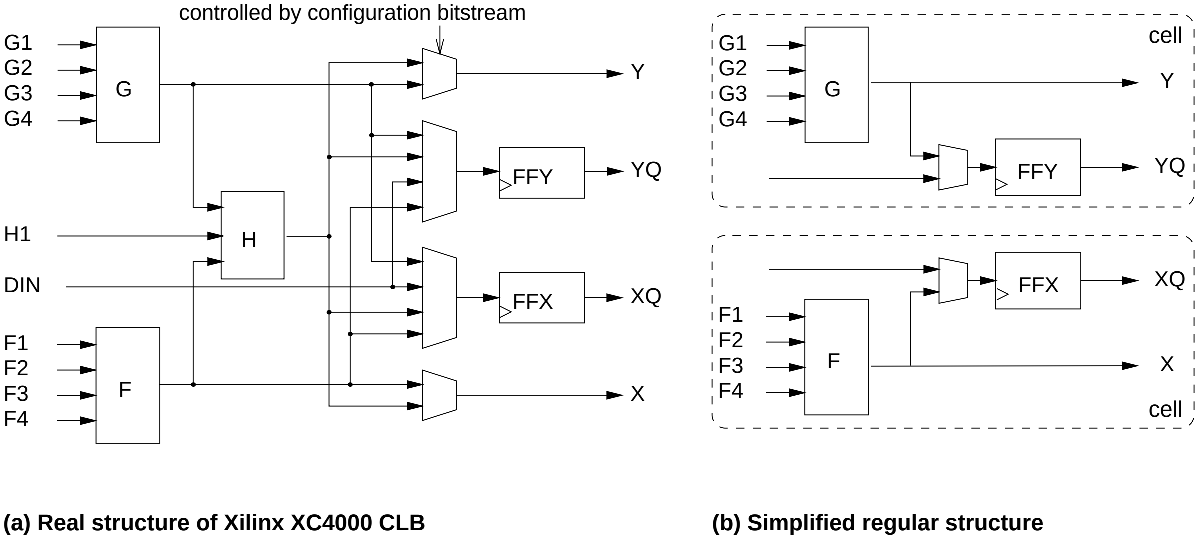

controlled by configuration bitstream
G1
G2
G3
G4
G
H1
H
DIN
F1
F2
F3
F4
F
FFY
FFX
Y
YQ
XQ
X
cell
(a) Real structure of Xilinx XC4000 CLB
(b) Simplified regular structure

Figure 2.28: XC4000 CLB and corresponding regular cells

Our solution emphasizes regularity by treating each of the irregular CLBs as two simplified, but completely regular cells (Figure 2.28.b).

Explanation 7 An SDI *cell* is the smallest regular unit of logic supported by the LB architecture of a given FPGA. In this context, "regular" means, that any two cells of the mapped netlist can be exchanged between LBs. ■

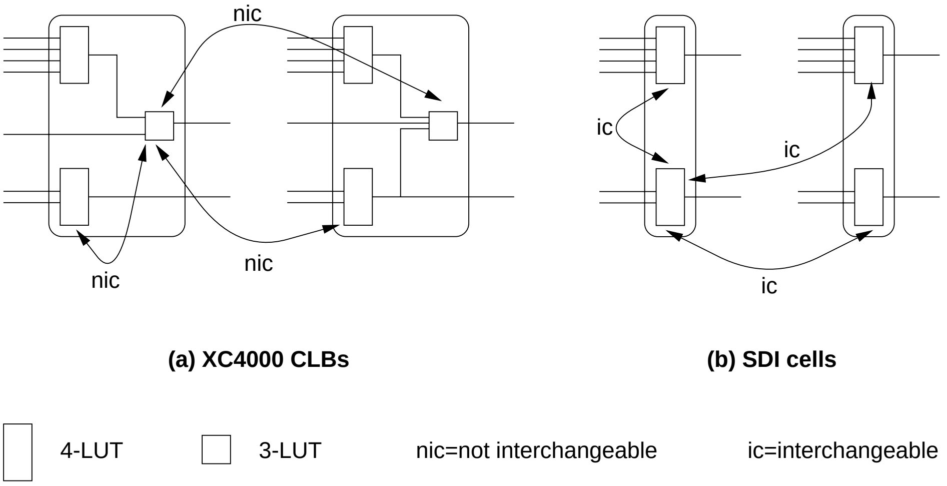


Figure 2.29: Interchangeable netlist cells in CLBs

Figure 2.29 shows an example of some of the restrictions imposed by the original CLB (a) and the complete interchangeability of netlist nodes and LUTs (inside as well as between different LBs) for cells (b).

Each of the two cells corresponding to an XC4000 CLB includes a 4-LUT and a flip-flop, which is also externally accessible. Thus, the cell concept sacrifices the H-blocks for independent flip-flop access.

Due to the cell concept, the usual match-and-pack step is not applicable to SDI. The nodes of the mapped netlist are viewed as cells and assigned to CLBs only during placement.

2.7.4 Two-Phase Placement

To exploit regularity to reduce run-time, microplacement is partitioned into two phases with differing objectives, both operating on a cell-based placement matrix (Figure 2.30).

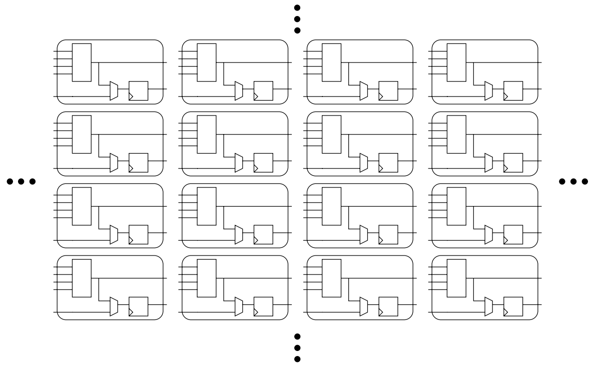


Figure 2.30: Cell-based placement matrix

The first phase aims primarily at optimizing the vertical signal flow typical for datapaths. Vertical signals include control signals spanning the entire height of the module, as well as inter-slice connections (e.g., carry-chains or shifter data). Since vertical signals may span multiple slices, all optimized master-slices of a module have to be considered simultaneously.

With all inter-master-slice dependencies now resolved, the next phase narrows its scope to a single optimized master-slice, allowing a parallel processing of all optimized master-slices. This second phase is purely timing-driven, minimizing interconnection delays.

Phase 1: Horizontal Placement

In the first phase, cells are assembled into columns, fixing their x-coordinates while the y-ccordinates are left floating. Here, the placer strives to

1. assign cells to the columns of the placement area in order to minimize the number of VLLs used for control signal routing.
2. allow vertical inter-slice nets in adjacent slices of the stack to be routed by abutment.
3. minimize the maximum routing length on critical paths.

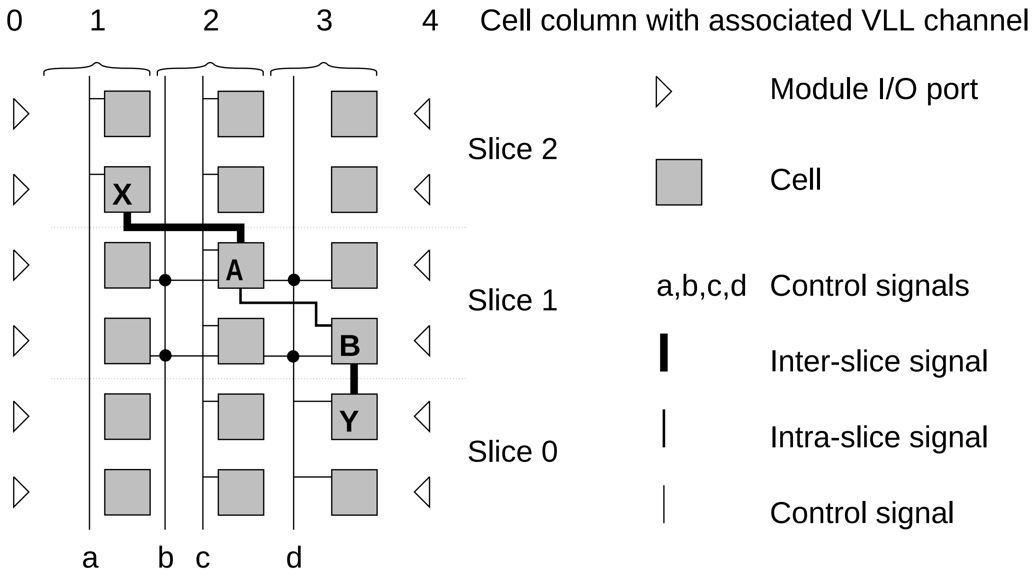


Figure 2.31: Horizontal placement model

This phase uses the placement model shown in Figure 2.31. It consists of columns of cells separated by vertical channels for control routing. For the XC4000, each of the vertical channels contains 10 VLLs (the example in Figure 2.31 uses a maximum of 2 VLLs per channel). The vertical channel associated with each cell column is assumed to lie left of the column. A control signal routed in channel k is available to cells in columns k (left of the channel) and $k-1$ (right of the channel). Note that for control routing purposes, the channel directly to the right of the module's placement area is also considered to be available. Control signals can be replicated and routed in multiple channels (not shown in the example), if necessary. Thus, the number c_{VLL} of VLLs used for control routing can be greater than the number c of control signals in the compacted module.

In addition to aligning cells along common control lines, the placer also tries to minimize the critical path delay d_{max}. This includes routing non-control signals between slices by abutment when possible. E.g., in Figure 2.31 the TTN (B, Y) was routed by abutment. I/O ports are assumed to be placed at the left or right (depending on floorplan context, Figure 2.25) border of the module. In the example, ports are located at columns 0 and 4.

For delay minimization, the wiring delay d_{AB} of a two-terminal net (A, B) is modelled as the simple horizontal distance $|x_A - x_B|$ between the two cells A, B, where x_Q is the column of cell Q. However, this metric becomes increasingly inaccurate with growing cell height H. Since the vertical distance is not known during this phase, it is currently approximated as $\lfloor H/4 \rfloor$. This assumption is based on the XC4000 topology with a maximum of one switch matrix for 4 cells (4-LUTs) in a 4×1 area (Section 2.1). Thus, d_{AB} becomes $|x_A - x_B| + \lfloor H/4 \rfloor$. Without an estimation, the model would try to minimize the wiring delays by mistakenly preferring the vertical over the horizontal direction. The layouts generated in this manner are measurably worse in terms of delay than those with the proposed estimation. This impreciseness of the approximation can be justified with the intent of the compactor to process flat bit-slices instead of tall modules. Should this assumption fail, a more accurate assessment would be necessary.

Given the aims of the horizontal phase, the placer minimizes the objective function $w_d d_{max} + w_c(c_{VLL} - c)$. w_d and w_c are user-definable weights that can be employed to trade-off a faster layout against a larger number of control lines. By default, their values are set to 1. The term $(c_{VLL} - c)$ makes the placer purely timing-driven once the minimum number of routing channels has been reached $(c_{VLL} = c)$.

Phase 2: Vertical Placement

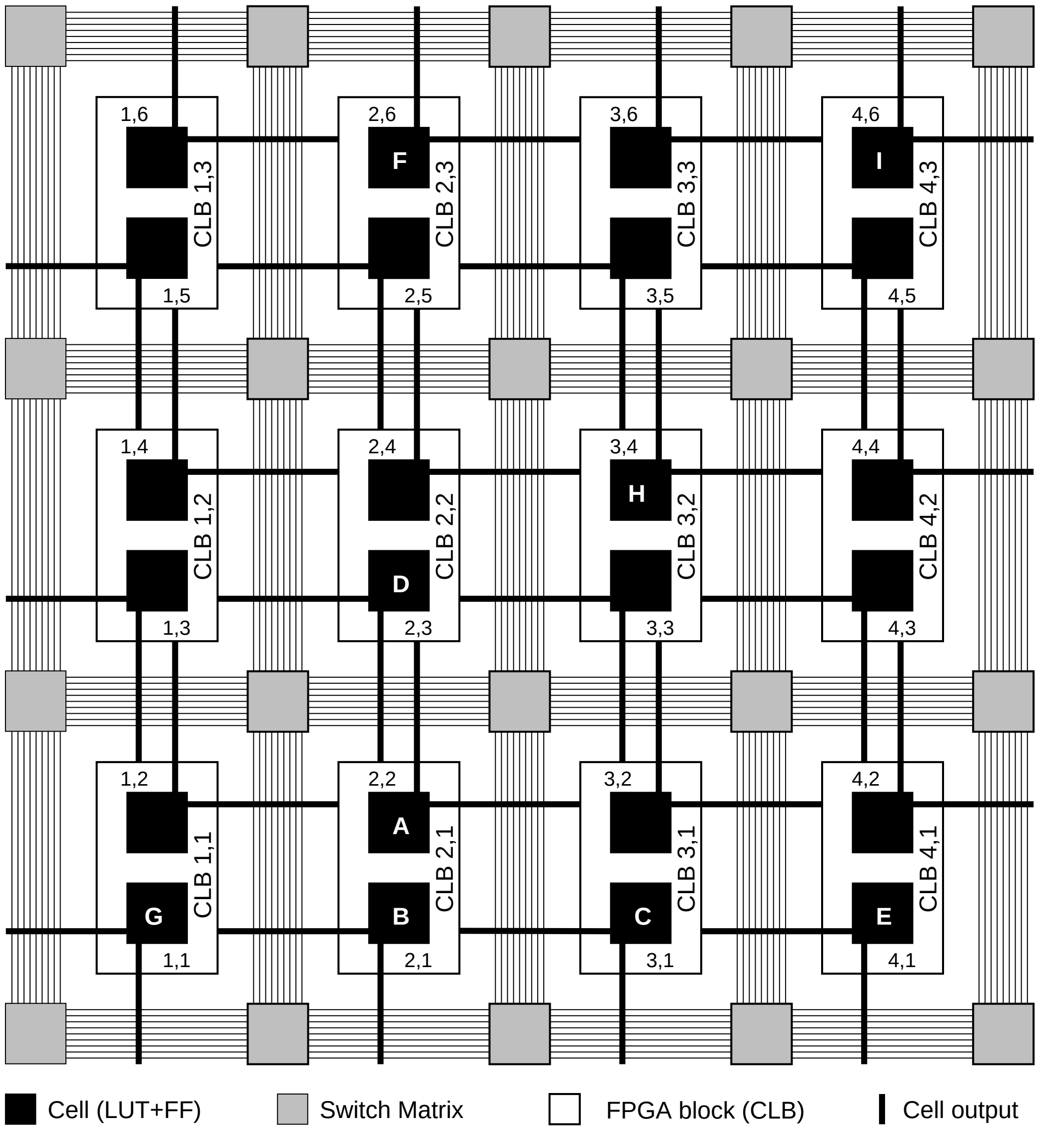
The second phase assigns row locations to the cells in the columns assembled earlier. In contrast to the horizontal phase, the vertical placement phase (Figure 2.20.5) concentrates solely on wiring delay minimization on the critical paths. Since it is not concerned with inter-slice dependencies, its scope can be limited to a single optimized master-slice. As before, locations of external I/O ports are determined in context of the original floorplan.

With the problem size reduced further, it becomes possible to use a more precise model of the FPGA routing architecture that better reflects the non-continuous distance relations. This more detailed model generates measurably better layouts over those obtained using simple manhattan distances, especially for more complex slices. Figure 2.32 shows the model, which is a simplified view of the XC4000 routing network (Section 2.1). Cells A to I have been labeled to serve as example TTN nodes in further explanations.

Since cell-to-CLB assignment takes place at this level, CLB boundaries have to be considered here. This is done by overlaying the cell grid with a CLB matrix of the same length, but of half the height of the cell matrix.

The mapping of real routing resources to the model is shown in Figure 2.33. The model encompasses direct connections (no switch matrices passed) and general single-length connections (one switch matrix per segment). Vertical long lines were handled in the horizontal placement phase. Horizontal long lines were allocated during floorplanning to create chip-wide busses or to route long-range inter-module signals. To limit the complexity of the model, double-length lines (Section 2.1.2) are presently not included.

The upper cell of a CLB will be placed in the G-LUT, and thus use the Y and YQ outputs, the lower cell will be located in the F-LUT with its output being routed through the X and XQ pins (Figure 2.28). Y/YQ and X/XQ output pins are assumed equivalent for routing purposes: Both Y/YQ pins reach above and to the right of their CLB, both X/XQ pins below and to the left (Figure 2.5, Section 2.1.2). The location of input pins is not included in the model because they are located at all four sides of the CLB. A signal is assumed to be available at the inputs of all cells within a CLB when it reaches the CLB boundary. The capacity of the routing channels is not considered (but see Section 2.7.1).



TTN	d_{SM}
A, F	2
A, B	0
A, D	0
D, A	0
B, D	1
A, C	0
C, B	0
B, C	1
A, H	1
B, H	2
I, H	2
A, I	3
B, I	4
A, E	2
B, E	2

Figure 2.32: Vertical placement model with example TTN routing lengths d_{SM}

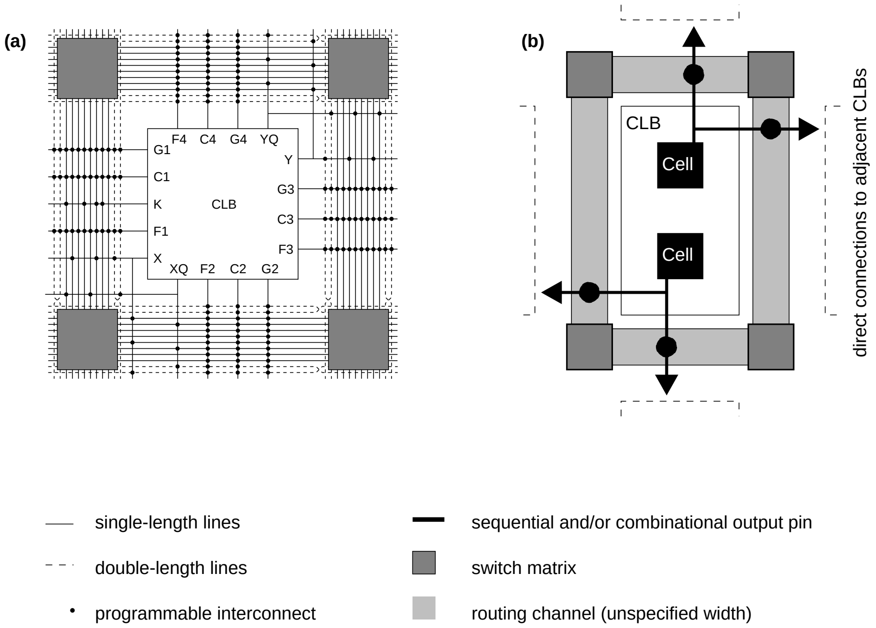

(a)
G1
F4
C4
G4
YQ
Y
C1
G3
K
CLB
C3
F1
F3
X
XQ
F2
C2
G2
(b)
CLB
Cell
Cell
direct connections to adjacent CLBs
single-length lines
double-length lines
programmable interconnect
sequential and/or combinational output pin
switch matrix
routing channel (unspecified width)

Figure 2.33: Real CLB routing structure (a) and abstract model (b)

The delay metric employed in this phase is not based on simple manhattan distances, but on an actual count of switch matrices (SM) in a signal path. For its calculation, three major cases based on the horizontal distance of cells a, b on a TTN (a, b) have to be considered. For each case and sub-case, the corresponding TTNs in Figure 2.32 will be pointed out.

If the horizontal distance is 0, the SM-distance d_{SM} is the simple CLB manhattan distance $|y_a - y_b|$ if the cells are placed in different non-adjacent CLBs ((A,F), $d_{SM} = 2$) . If they are placed within the same CLB, the SM-distance becomes 0 ((A,B), $d_{SM} = 0$). In the case of adjacent CLBs in the same column, the possibility of a direct $d_{SM} = 0$ connection depends on the LUT assignment of source cell a in a CLB: If a is below b, a should be assigned to the G-LUT ((A,D), $d_{SM} = 0$). If a is above b, a is better placed in the F-LUT ((D,A), $d_{SM} = 0$). If these assignments are not possible, the signal will have to pass through one SM ((B,D), $d_{SM} = 1$).

If the horizontal distance is 1, a direct connection is possible if the two cells are placed in the same row and a is assigned a suitable LUT. Specifically, if b is to the right of a, a should be assigned to the G-LUT ((A,C), $d_{SM} = 0$). If b is to the left of a, the F-LUT should be chosen ((C,B), $d_{SM} = 0$). Otherwise d_{SM} is the manhattan distance of one SM ((B,C), $d_{SM} = 1$). When a and b are placed in different rows, d_{SM} becomes the $|y_a - y_b|$ ((A,H), $d_{SM} = 1$), adjusted for an inopportune LUT assignment: The distance is increased by one if b is to the right and above a and a was assigned to the F-LUT ((B,H), $d_{SM} = 2$). Similarly, an assignment that places a in the G-LUT but has b located to the left and below a, will incur this SM-penalty ((I,H), $d_{SM} = 2$).

If the horizontal distance is greater than 1, another effect becomes evident: When the vertical distance also becomes greater than 1, the SM-distance is reduced by 1 over the pure manhattan $|x_a - x_b| + |y_a - y_b|$, since the corner SM can be shared to advance in horizontal and vertical directions with a single step ((A,I), $d_{SM} = 3$). This occurs in addition to the correction for inopportune LUT assignments as outlined above ((B,I), $d_{SM} = 4$). However, when a and b are placed in the same row, both effects vanish and d_{SM} reverts to a pure manhattan distance ((A,E), $d_{SM} = 2$, (B,E), $d_{SM} = 2$).

2.8 Design Integration

Microplacement is the last step of SDI relying on novel tools. The lowest levels of design implementation (pin-assignment, routing, bit-stream generation) rely on vendor tools. For the Xilinx FPGAs currently supported, this is primarily the PPR (partition-place-route) program of the XACT tool suite [Xili94c].

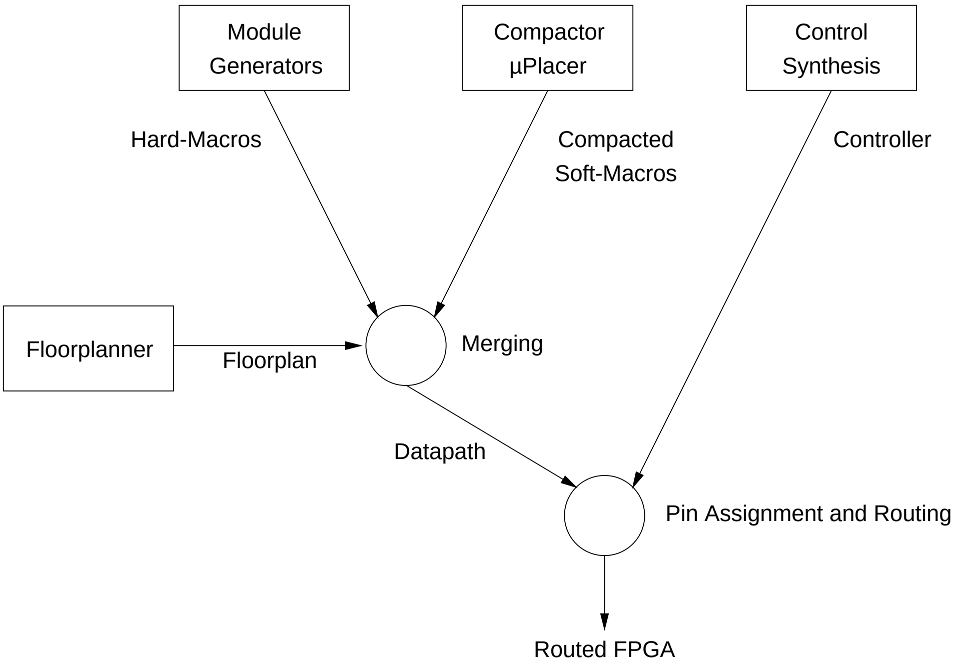


Figure 2.34: Design integration

With the soft-macros now compacted, the different parts of the whole chip have to be integrated (Figure 2.34). This occurs in two steps: First, the layouts of hard-macros and compacted modules are composed according to the original linear floorplan. Note that the resulting datapath core layout is only partially routed (only within the original hard-macros) and no pin-assignment has been performed in the compacted modules. These operations occur in the next step: The regular datapath layout and the irregular controller circuit are then merged by PPR for partitioning (controller only), placement, and routing. The datapath layout is placed according to the SDI-generated location data, while the controller circuit is processed by a simulated annealing based algorithm. All open nets (those not already routed in the hard-macros) are then handled by the maze-router of PPR with a rip-up and retry extension. Routing thus connects hard-macros, compacted modules, and the irregular controller. Remember, that during microplacement, the logic blocks were already aligned to allow efficient routing of control signals on VLLs (Section 2.7.4). The final result of this procedure is a bit-stream ready for downloading, combining regular and irregular elements.

The next chapters will describe key elements of SDI in greater detail. Chapter 8 gives some experimental results quantifying the performance increases achievable using our approach, and Chapter 9 suggests directions for further research.

3 Module Generators and Library

The application of software tools to create customized circuit components from a reusable parametrizable “template” has a long history in VLSI design. Beginning with early uses to create simple regular arrays of tiled sub-layouts [McWi78] to the generation of complete processor cores [BDRA93], such automatic generation of modules (also known as macros, mega-cells, building-blocks, etc.) continues to play an important part in many current design styles. Recently, interest in the use of parametrized module libraries for technology-independent specification of designs that can still be efficiently mapped to different target technologies lead to the standardization of the Library of Parametrized Macros LPM[EIA93]. Apart from administrative specifications, such as data formats for module and parameter formulation in various input languages (EDIF, Verilog, and VHDL), LPM defines a library of 25 generic modules with their parameters (Table 3.1). Due to limited resources, the current SDI library covers only a major subset of the LPM functionality, but could be extended to the full specification.

CONST	INV	AND	OR	XOR
LATCH	DFF	TFF	RAM_DQ	RAM_IO
ROM	DECODE	MUX	CLSHIFT	COMPARE
ADD_SUB	MULTIPLIER	COUNTER	ABS	BUSTRI
FSM	TTABLE	INPAD	OUTPAD	BIPAD

Table 3.1: Current list of LPM modules

3.1 Previous Work

LPM specifies just the interfaces of its modules, it makes no statements regarding the generation process or format. With the long history of automated module generation, the capabilities and specialties of actual module generators span a broad spectrum. In the context of this work, we will only examine generators transforming a parametrized function instance (adder, multiplier, etc.) into a layout or a structural netlist.

Structured generators creating netlists are technology-independent to a large degree. Classical solutions [BaSe88b] and recent approaches [BrMR94] are very similar, differing mainly in the variety of design alternatives available for the generation of a specific function. Even layout generators, being far more technology dependent, bear considerable resemblance for macro-cell and FPGA target technologies.

3.1.1 Generating Modules for Macro-Cells

Depending on their routing behavior, traditional layout generators targetting macro-cell technologies can often be divided into two classes. One kind does not supply explicit routing and connects signals solely by abutment [Law85], while the other kind includes routing facilities. Routing is usually performed by the integration of conventional routers [BaMS88c], the use of patterned routing tiles [SYYH92], or a combination thereof [IsYo87]. A common limitation of generators for macro-cells is to support only a single height (pitch, Section 2.3.1)) for bit-slices [BeGr93]. Pitch-matching [King84] can be used to circumvent this particular problem, and to reduce area and delay in general, a post-generation geometric compaction phase is often employed [MaWa90].

Module generators also differ in their ability to respect pre-specified bounding-box aspect ratios during layout generation. Some, such as [GuSm86], accept an aspect parameter prior to generation. Others rely again on a post-processing step to transform the layout to the desired shape [DeNe87].

3.1.2 Generating Modules for FPGAs

FPGA module generators can also be classified according to their routing facilities. However, far more interesting are their pitch-matching, compaction, and shape-handling capabilities, since these points are complicated by the inflexible coarse-grained FPGA base architecture.

Xilinx X-BLOX

X-BLOX[Xili94a] emphasizes the automatic propagation of datatypes between modules. However, it cannot generate placed modules (neither geometrically nor topologically) for “simple” logic like bus gates or multiplexers. When X-BLOX does assemble placed modules (for hard-carry arithmetic or certain types of registers), it provides a topological LB placement, but no routing information. X-BLOX macros are generated only with a single pitch and shape, with neither pitch-matching nor shape-transformation being available. For compaction (esp. in the case of the “simple” logic), X-BLOX relies on the logic synthesis and technology mapping facilities in PPR [Xili94c]. However, this step neither preserves nor exploits any module-inherent regularity.

Oxford PRG XMACROS

In contrast, XMACROS[Law96] can generate pre-placed and pre-routed modules. The library is rather specialized (clocks, counters, buffers/latches, decoders, channels, RAM interfaces, Transputer links). However, in addition to module generation, XMACROS can also automatically supply OCCAM-like code describing an interface to the generated hardware [Page95]. As in X-BLOX, neither pitch-matching, shape-handling, nor compaction routines are supplied. The last two restrictions, however, are intentional: Due to the requirement to reliably process asynchronous communication protocols, the modules generated by XMACROS have to guarantee very precise timing specifications. Any kind of post-generation compaction would disturb these timing constraints.

TU Chemnitz LORTGEN

LORTGEN [BrMR94] concentrates on choosing architectural design alternatives for complex modules (e.g., a parallel multiplier) for a given set of parameters (including timing and area constraints). It contains extensive knowledge bases (far exceeding the simple geometrical composition rules of most other generators) and performs a fuzzy evaluation of different implementation options. E.g., for the multiplier, three alternatives for the generation of partial products are considered, four alternatives for their addition and 40 implementations for the final vector merging adder. For a 22x9 multiplier, this leads to the evaluation of 480 design alternatives for the architecture of a single module. However, LORTGEN generates only structural netlists, no placement or routing is provided.

3.1.3 Module Templates

Module generators also vary in the form used to describe the “template”. Generators like [BaSe88b] for macro-cell technologies and [MaJO96] for FPGAs use dedicated languages to specify the tiles and iteration patterns. Layout generators usually employ more detailed languages than netlist generators, with the FPGA-based languages being most technology-dependent. FPGA-layout languages such as FRADL [MaJO96] allow the description of circuits even at the most architecture-specific levels (different routing resources, detailed LB configuration).

Another approach is to embed the template-describing operations into a conventional programming language. For example, GENVIEW [CGPP91] embeds the constructs for generating macro-cell layouts in C functions. MACLOG [BoSW90] and SDI’s Paramog [Ditt95] [Sade95] build on a C++ framework in an object-oriented manner. X-BLOX[Xili94a] uses a fact-and-rule database specified in PROLOG, and XMACROS[Law96] depends on a notation formulated in ML.

3.2 Anatomy of an SDI Module

A module is defined by the parametrized algorithm for its generation. This algorithm assembles a module instance according to an actual set of parameter values. Often, this is done by composing the module instance from smaller components (slices, tiles, leaf cells, etc.). Module generation can be performed at any level, ranging from a high-level behavioral description down to the layout level.

In the context of SDI, module generation is supported at the layout and at the structural level. Module layouts consist of a relative placement of LBs with an already fixed module-internal routing, and can be relocated to any absolute position on the FPGA die. A structural module description comprises a netlist containing both combinational and sequential elements with only topological placement, but no routing information (see Section 2.3.1).

3.2.1 Classes of Regularity

In both cases, explicit regularity information is generated and made available to other SDI components. All modules currently in the SDI library fall into the class of circuits known as regular arrays. A *regular array* consists of multiple instances of only few different processing elements (PE) with identical (or at least similar) interconnections between PEs throughout the array [MaJO96], resulting in a linear array or rectangular mesh. Connectivity is generally densest between adjacent PEs on a two-dimensional surface. Regular arrays can be further classified according to their dimensionality and complexity. For example, they include linear iterative arrays (such as a ripple-carry adder), as well as more complex two-dimensional logic arrays (e.g., a carry-save multiplier [Omon94]). The functionality of the arrays ranges from bit-level arrays with a PE complexity of a full adder or less, to word-level arrays with PEs containing, e.g., a multi-bit ALU or a parallel multiplier.

The current SDI library contains only bit-level arrays composed of slices containing one or two bits of, e.g., a bussed AND gate, a bus multiplexer, a counter, or a Booth multiplier. However, word-level arrays can be processed as well. E.g., Section 8.3 describes a 4-bit 74181 ALU used as PE.

3.2.2 Geometric Hierarchy

The quantities describing a module "template" for generation are similar to those used in Section 2.6 to describe the compaction process. However, the 2-D geometric composition view (array of tiles) of the layout is emphasized over the 1-D circuit view (stack of slices) during generation.

The construction of a regular-array module in Paramog [1] according to a geometrical hierarchy is shown in Figure 3.1. At the highest level, it is partitioned into *v-zones* (VZ), zones of vertically replicated logic. Each instance of the replicated logic is called a *v-segment*. The example has the three v-zones A, B, and C stacked from bottom to top. For each VZ, the *v-segment height* in LBs and the vertical replication count (the number of segments) is specified. In the example, VZ C contains two instances of the same sub-circuit. Each of these v-segments is two LBs high. Thus, VZ C has a height of four LBs. VZ B has four v-segments at this level, and VZ A has only a single one.

---

[1] Note that the compaction and microplacement phases use a different hierarchy order, Section 5.2.9.

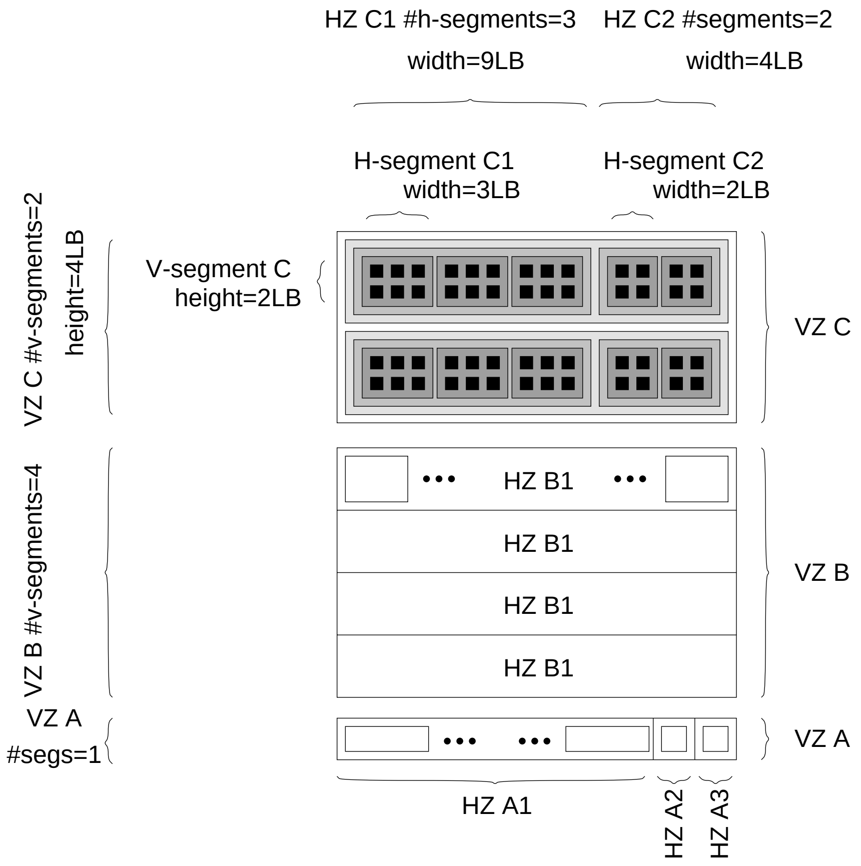
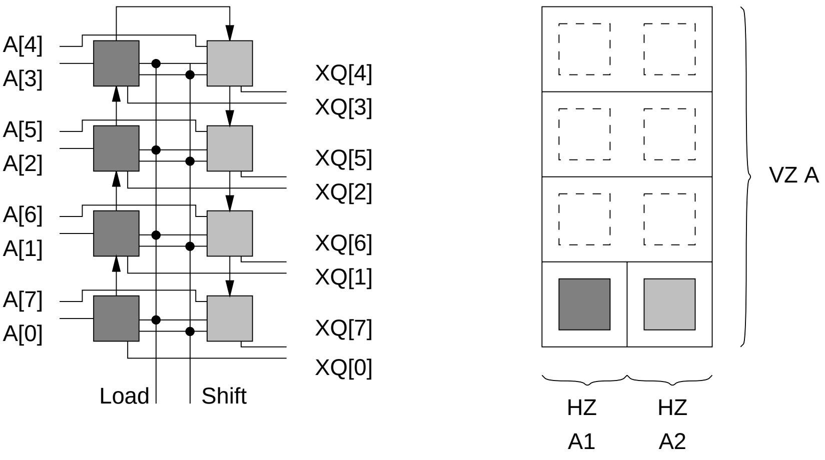


Figure 3.1: Geometric regularities in a module

At the next level down in the geometric hierarchy, each v-segment is decomposed into *h-zones* (HZ), zones of horizontally replicated logic. Each instance of the replicated logic is called an *h-segment*. For each h-zone, the *h-segment width* and the horizontal replication count is specified. In the example, VZ C has the two h-zones $C1$ and $C2$, adjoined from left to right. HZ $C1$ has an h-segment width of three LBs and contains three replicated h-segments, the h-segment width HZ $C2$ is two LBs and the replication count is two. Analogously, VZ B contains only a single h-zone $B1$, while VZ A contains the three h-zones $A1$, $A2$, and $A3$.

At the lowest level, similar to leaf-cells in macro-cell module generators [GrPe97], [Raba85], the h-segments themselves are described. An h-segment consists of LB functions, a placement of the LBs in the h-segment relative to its lower-left corner, and optionally fixed routing between LBs. As can be seen, the bounding-box of an h-segment is determined by the v-segment height and the h-segment width.



(a) Generated Circuit

(b) Module Topology

Figure 3.2: Circuit and underlying module topology

Figure 3.2 shows these relations using the example of a folded 8-bit shift register. It consists only of a single v-zone A, that contains four v-segments. Each of the v-segments has two h-zones, each h-zone with a single h-segment each. Each h-segment contains a single LB: $A1$ provides an upward shift, while $A2$ supplies the downward shift direction.

3.2.3 Structural Modules

When structural modules are generated, the relative LB placement and routing derived from the information in the h-segments is discarded, and the LB functions of all h-segments within a VZ are merged. The result is an unplaced netlist representing the replicated circuit in each VZ. The vertical replication counts and the stacking order from bottom to top describe a topological arrangement of logic with no concern for geometric shape constraints (each node of the netlist is viewed as a topological point). Retaining at least a topological placement differentiates the SDI approach from earlier netlist generators, such as [BaSe88b], which do not provide any placement info, or [BeGr93], which relies on user-provided placement hints (special naming conventions).

3.3 Module Generation in SDI

In SDI, the sub-system Paramog is responsible for all aspects of module generation. This section will outline key points of the work already presented in [Sade95], [Ditt95][2]. Paramog has the following features:

- Outputs placed and routed macros, or structural netlists
- Topology information available even for netlist output
- Module shape constraints and logic structure considered during generation
- Offers alternative implementations for speed vs. area trade-off
- Flexible pin-assignments for layout generation
- Support of FPGA-specific features

3.3.1 Paramog Architecture

All interaction with Paramog (Figure 3.3) occurs through the Paramog manager. This program accepts requests from SDI FloorPlanner in the form of simple text files, and coordinates the use of internal Paramog operations. The module generators themselves are stand-alone programs built upon a common C++ framework. Each of them has access to a dedicated database of tiles, called *models* in this context, that can be assembled algorithmically to form the desired module. Depending on the selected output format, the composed module can be written ether as complete layout, or as a structural netlist with only topological placement.

3.3.2 Design Cycle

A Paramog design cycle begins with a query for design alternatives fitting a given set of parameters. Paramog responds with a set of different design variants, usually offering different choices regarding layout shape, logic organization (BPLB, Explanation 3), and speed. The requestor, in SDI always FloorPlanner, then evaluates the alternatives against each other, possibly in a wider context. Once a concrete choice has been made, the desired design variant and output format can be selected for generation.

---

[2] For improved consistency, some definitions (esp. h-zone and v-zone) have been changed in this work.

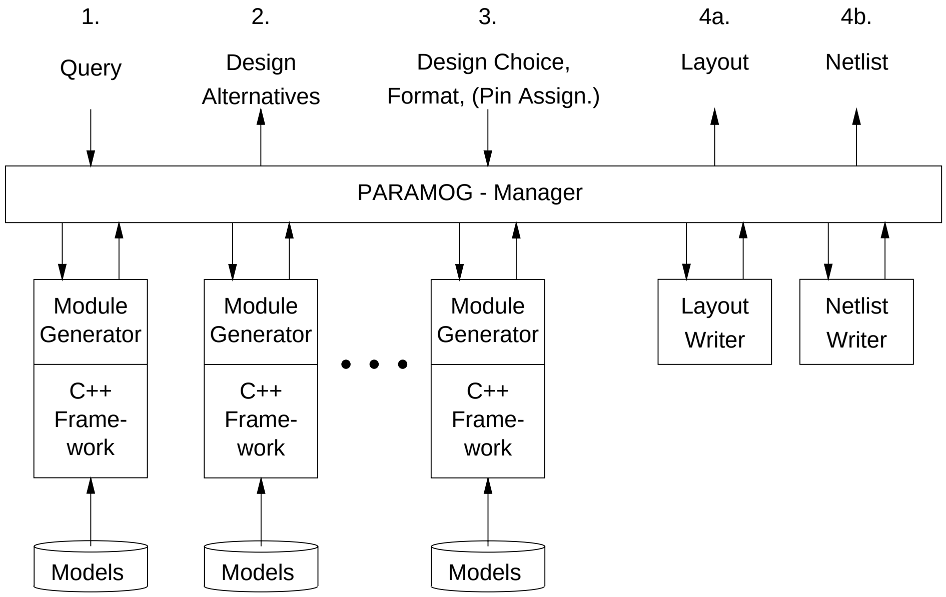


Figure 3.3: Paramog architecture

3.3.3 Module Parameters

In query mode, the caller selects a function name from the list of supported functions (Table 2.1) and parametrizes data operands, data results, and control signals with their bit-widths and data types. While bit-widths can be specified to accommodate fixed-point PEs, none of the current modules makes use of this feature. Data types currently distinguish between signed and unsigned numbers.

An optimization preference is expressed as a percentage value each for time and area. A value of 100 specifies the fastest or smallest designs, respectively, available in the library. Paramog will only propose alternatives meeting or exceeding both of the given optimization requests.

Module topology can be constrained by limiting the maximum height of the module in LBs, and defining the logic structure as a BPLB value. If necessary, Paramog can generate folded layouts (see Section 3.3.4).

Only the signal widths and their types must be specified, all other parameters are optional. If omitted, the solution space is left unconstrained.

3.3.4 Design Alternatives

The Paramog response to a query consists of a text file listing zero or more alternative implementations of FloorPlanner's request. An alternative is initially characterized by its actual area vs. time tradeoff as two percentages. The actual timing behavior for the alternative is given as a delay in ns for purely combinational modules, or a maximum operating frequency and beat clock count for sequential operations (e.g., a pipelined shift-add multiplier).

Physical extents and logical sizes for the alternative implementations are detailed as height and width of the LB matrix, and the actual BPLB value. Additional fields describe the alignment of module operand and result signals relative to the common base line separating controller and datapath areas in the chip topology (Section 2.3.2).

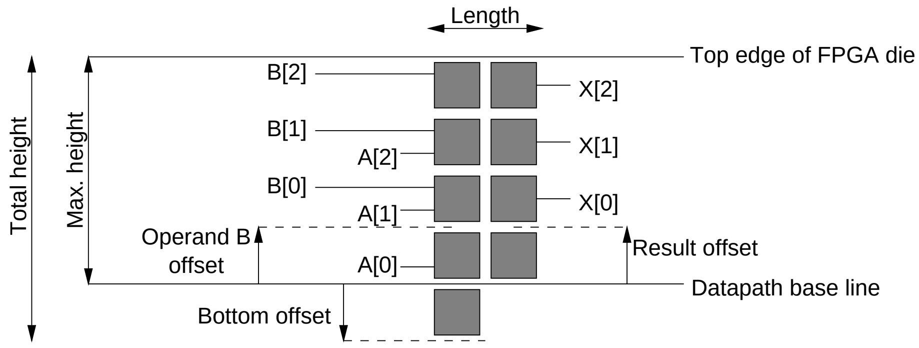


Figure 3.4: Physical extents of a module and signal alignments

Figure 3.4 shows a module two LBs long that fits a maximum height of four LBs, extends one LB beneath the datapath area base line, and thus has a total height of five LBs. The LSB of the X result bus and the B operand are offset by a single LB from the datapath base line, while the operand A is exactly aligned.

Each variant also specifies which of the horizontal and vertical long lines are used in the layout for internal routing. Combined with the alignment information, this permits congestion management already during module selection and floorplanning.

Next, the tiled array structure of each design alternative is formulated according to the geometric hierarchy (see Section 3.2.2). At the bottom, nested in v-segments and h-segments, are the descriptions of single LBs at the equation level. Supplying the equations enables the floorplanner to determine the compactibility of adjacent modules (Section 2.6). To further aid in avoiding congestion, the LB models also list routing channels used for internal signals, and possible pin assignments of module signals to LB pins.

As an example of the alternatives offered by Paramog, Figure 3.5 shows different layouts proposed for various left shift registers with different widths and BPLB parameters. Note that a module at the layout level consists of placement and routing information, with the routing also specific to the targeted FPGA. As can be seen, the generator is aware of the three different routing resource types of the XC4000 and uses them accordingly. Furthermore, observe that Paramog can generate folded layouts when the width of the datapath exceeds the height of the bit-slice area (Figure 2.15). The bit ordering in the datapath is also variable: e.g., layout (e) in Figure 3.5 has an ascending order in both columns, while layout (b) alternates the bit order from ascending in the left column to descending in the right column. The actual contents and pin-assignments of the LBs can be overlooked in Figure 3.5.

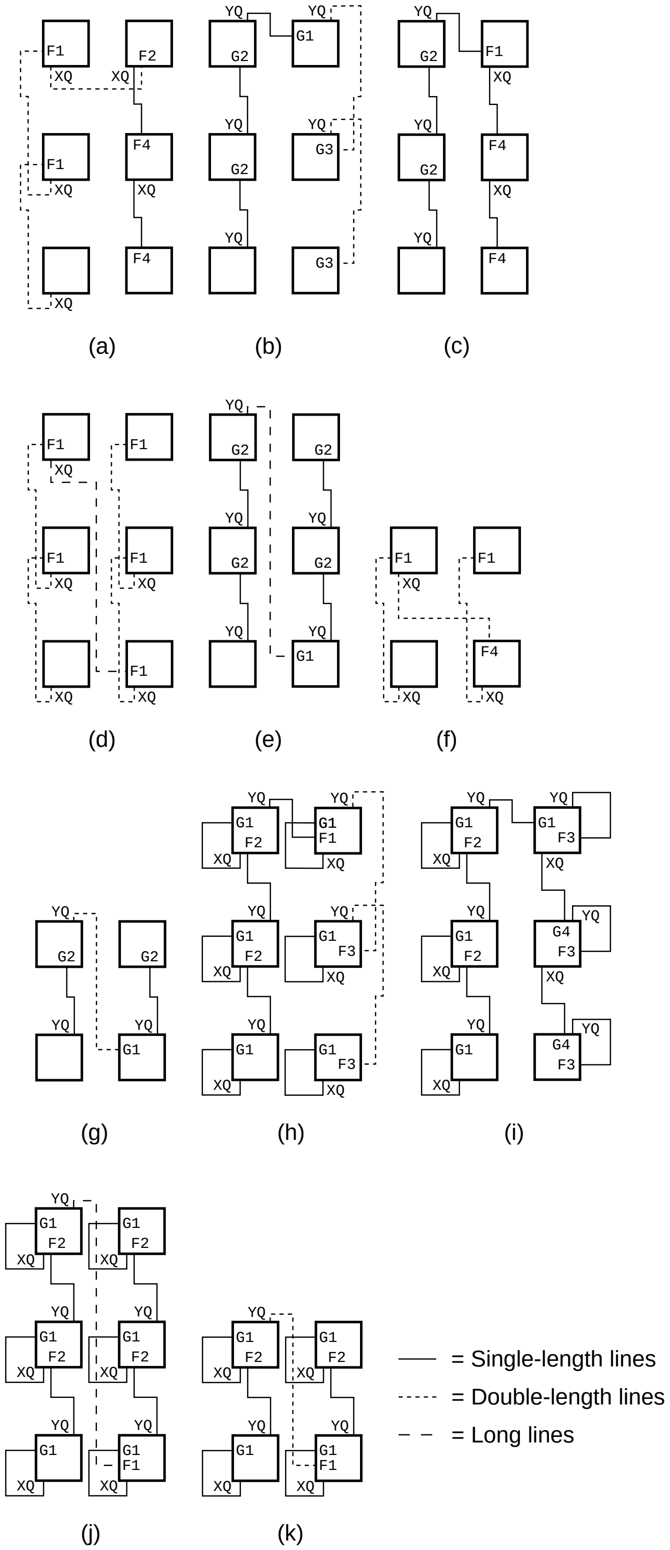


Figure 3.5: Layout styles for arithmetic left shift registers. (a) - (e) 6 bits, 1BPLB; (f) - (g) 4 bits, 1BPLB; (h) - (j) 12 bits, 2BPLB; (k) 8 bits, 2BPLB

3.3.5 Generator Output

When FloorPlanner has selected a concrete implementation for each module instance from the alternatives offered by the generators, it calls Paramog again with instructions which variant to actually generate.

Writing structural netlists

For structural netlist generation, no additional information is needed. The resulting output consists of two files: One file contains the equations and storage configuration for each LB, the second one lists all intra-module nets. While this process does not generate placement or routing data at the FPGA-level, the topological arrangement of logic according to the geometric hierarchy (Section 3.2.2) is exported. This allows FloorPlanner to take maximum advantage of the sliced structure during the compaction phase.

Adapting layouts to actual pin assignments

When running this phase of Paramog to actually generate a placed and routed module, further specifications need to be provided. In contrast to generators like XMACROS [Law96], or the arrays describable in FRADL [MaJO96], Paramog keeps only a minority of pins locked on internal nets in the module templates. The majority of module pins (both input and output) can be flexibly assigned to LB pins to better match the floorplan context. The LB configuration equations will be adjusted automatically to fit the pin-assignment selected. In the simplest case, this process is limited to the adjustment of a single LUT (e.g., as in Figure 2.26). However, due to the wide degree of freedom FloorPlanner has in assigning pins, more sweeping structural changes affecting multiple LUTs also have to be managed.

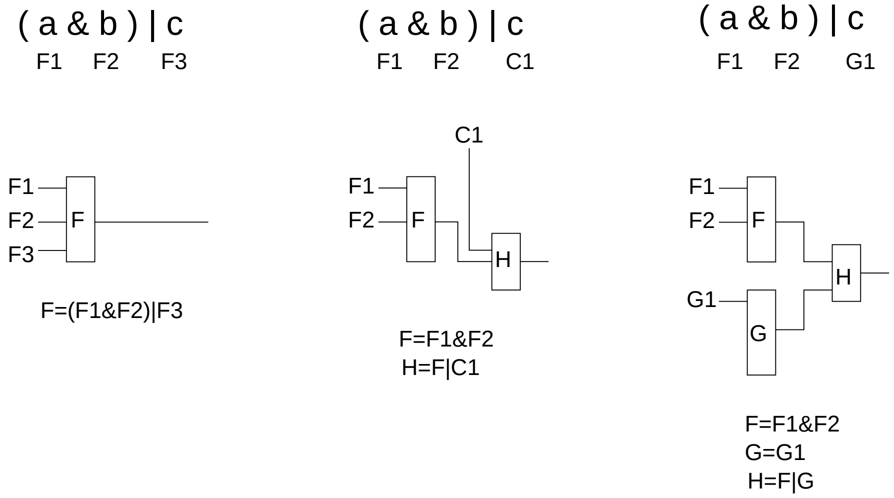


Figure 3.6: Effects of different pin assignments for logic function $ab + c$

Figure 3.6 gives an example for some of the different CLB configurations depending on the pin-assignment selected. Note that the assignment and configuration of implicitly required LUTs is performed automatically by Paramog. FF access is also covered by this procedure, adding an auxiliary LUT as required (Figure 3.7.a), but preferring direct connections when possible (b)[3].

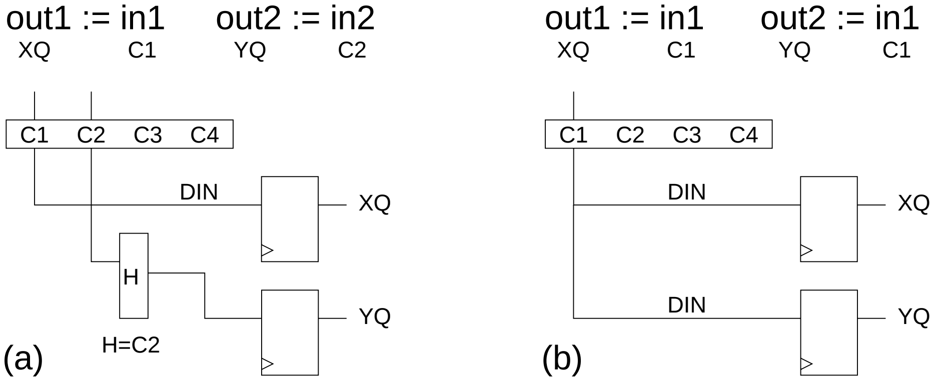


Figure 3.7: Effects of different pin assignments for FFs

Layouts are written in *relocatable LCA* (RLCA) format. LCA format itself is a technology-dependent, textual description of an FPGA layout for Xilinx FPGAs. It contains configuration information for each of the CLBs, IOBs, TBUFs, and routing at the *programmable interconnection point* (PIP) level (programming pass-transistors and multiplexers). However, conventional LCA files use an absolute coordinate system overlaid on the die. LCA is thus unsuitable for the description of partial dies, such as module instances. RLCA format is similar to LCA, but all coordinates (CLB, IOB, PIP) are relative to a module-local coordinate system with an origin at the lower-left corner. The `reloc` tool (part of SDI) can relocate RLCA files to any location on an FPGA die. It takes irregularities in the routing structure into account and modifies the layout accordingly. E.g., for modules containing VLLs crossing the middle of the die, the programmable splitters located there are set to connect both parts of the VLL.

3.3.6 XC4000-specific Features

Paramog supports the XC4000 architecture at a very fundamental level. This permits the use of features not directly expressible in a general netlist. Among these are the on-chip memory capability, allowing the use of CLBs as pre-initialized ROMs, or asynchronous RAMs, with either 16x2-bit or 32x1-bit organizations. The RAM/ROM modules in the Paramog library are mapped directly to this structure.

The fast-carry logic, which provides a dedicated, low-delay routing network for ripple carries, and hard-codes logic operations commonly used for arithmetic functions, is used for the efficient implementation of fast adder/subtractor

---

[3] In Figure 3.7, “:=” denotes a sequential assignment via clocked flip-flops (similar to the PALASM [AMDI90] notation)

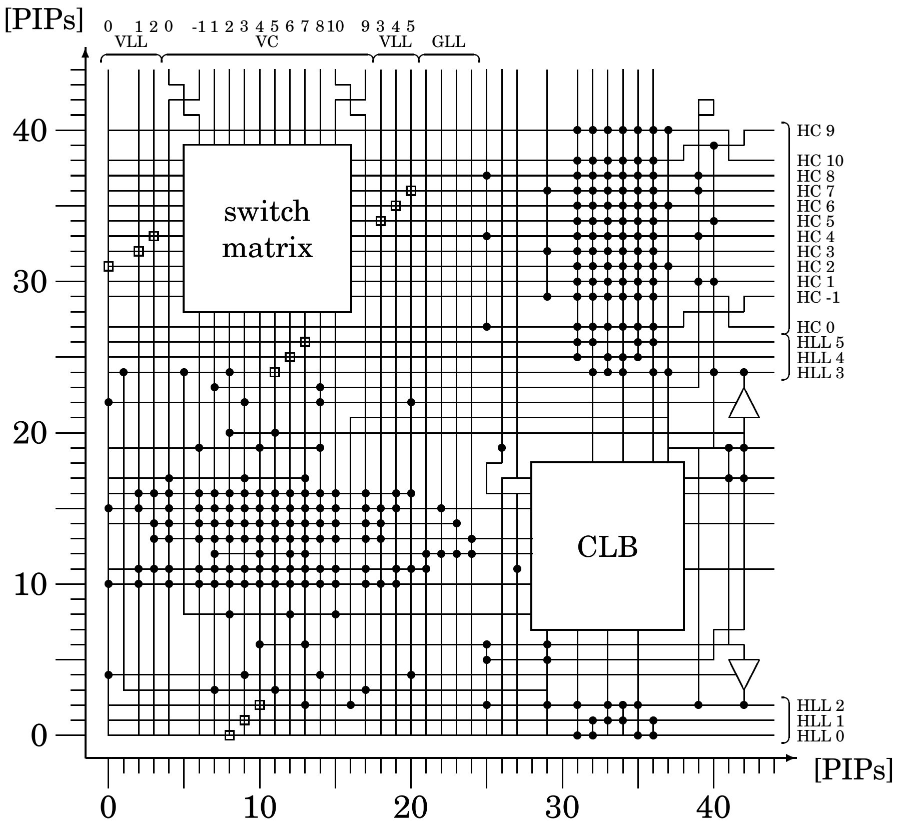


Figure 3.8: Basic tile for layout generation. "VC"=vertical channel, "HC"=horizontal channel, "GLL"=global long line

and counter modules.

The four different routing resources of the XC4000 (general purpose, double length lines, h/v long lines, global long lines) are fully supported. The basic tile structure used for layout composition (Figure 3.8) permits routing at the PIP level, allowing a careful optimization of signal delays in the module templates.

The on-chip tri-state buffers are not handled in the current implementation of SDI. Were they to be added, it would most likely be advantageous to consider them as part of a global connectivity allocation strategy [PBLS91] at the floorplan level, instead of using a module-local view during generation.

3.3.7 Implementation Details

Paramog is completely implemented in C++. The module generators themselves are separate programs, building on a common framework, also formulated in C++. By inheriting from the abstract base class `ModuleGenerator`, concrete generators gain access to file operations dedicated to the different formats, basic text scanning and parsing facilities, data validation routines, and text generators for various constructs in the output files. Auxiliary classes provide container data types and consistent error handling. Interestingly, this is a similar structure to MACLOG [BoSW90]. While both systems were developed independently, both frameworks favor a task-oriented approach (file operations, I/O abstraction) over an object-oriented view (abstraction of the module properties themselves). [BoSW90] sketches initial object-oriented attempts that were found unsuitable, and in turn discarded for the task-based view. In light of more recent research in object-oriented design [Booc94] and implementation techniques [GHJV95], further work aiming at a true object-oriented solution for the module generation domain seems worthwhile.

Paramog employs SIS [Sent92] during the logic-adjustment step in pin-assignment (Section 3.3.5). The optimization procedure `script.algebraic` is used to simplify partial functions during the re-structuring. Since SIS does not process the inhomogeneous mix of hard-wired 4- and 3-LUTs in an XC4000 CLB, LUTs are assigned by Paramog. Its simple heuristics are limited, however, in that they cannot map all realizable functions to the CLB structure. The tile functions in the current library are all handled correctly, though. A general solution for mapping to an irregular LB containing multiple LUTs with some hard-wired connections is the subject of on-going research [ChRo92], [Murg93].

The current approach of writing (R)LCA files is also questionable with regard to the recent Xilinx guidelines declaring the XNF file format, extended with relative location constraints (RLOC), as the format of choice for placed netlists. While XNF does not permit the formulation of routing information (neither at the resource, nor at the PIP level), in contrast to LCA, it is far easier to use in conjunction with other tools of the XACT design flow (XNFPREP, PPR, etc.). A new implementation of the module generators would most likely trade the performance gain due to the optimized module-internal routing for a smoother integration into the complete design cycle.

Figure 3.9 shows one of the possible implementations of a 6x6-bit Booth multiplier. Note that this module has a local controller, which descends below the baseline into the main controller area of the FPGA. Other implementations allow to place the local controller at the top of the module layout, or to interleave the two operands instead of stacking them. Regarding routing, the following aspects can be pointed out: High-fanout control signals have been routed on VLLs, the intra-module routing is regular, but connections to external pads are unconstrained and can flexibly adapt to pad locations.

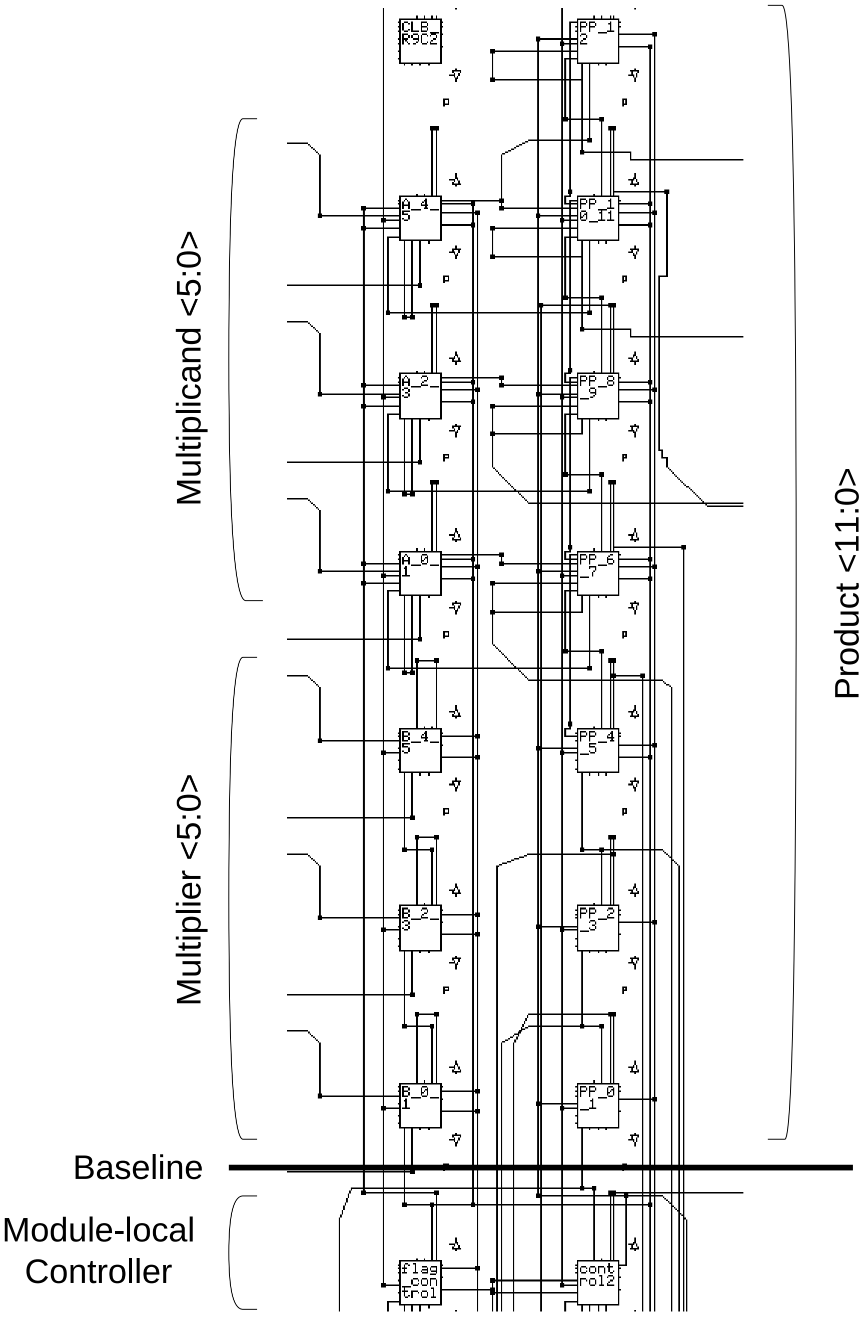


Figure 3.9: Sample layout for a 6x6-bit multiplier

4 Module Selection and Floorplanning

As shown in Figure 2.1, the SDI floorplanner is the nexus of the design flow. It reads the circuit to be processed as an SNF file, queries the module generators (Chapter 3) for possible implementation alternatives, and then proceeds to simultaneously obtain an optimal linear module placement (Section 2.3.1) and to assign a concrete implementation to each cell in the circuit (Section 2.5).

The solution strategy of the floorplanner uses a genetic algorithm [Gold89] [SrPa94] to explore the design space. This chapter will present some keypoints of the algorithms and data structures used. For an exhaustive treatment of the floorplanner in particular, and genetic algorithms in general, especially experimental results and a detailed discussion of optimization parameters, we refer the reader to [Putz95a] [Bode97].

4.1 Optimization by Genetic Algorithms

A genetic algorithm is a heuristic that considers multiple partial solutions (a *population* of *individuals*) in parallel. The solution space is explored by creating new solutions by merging parts of two existing solutions (Section 4.2), or by altering existing solutions (Section 4.5). Next, all partial solutions are evaluated according to certain criteria (Section 4.7). The result of the evaluation (the *fitness*) determines, whether a partial solution will still be considered (is *selected* for *survival*, Section 4.8) in the next iteration (*generation*) of the optimization. Unfit individuals are then discarded, and the process repeats. It continues until a maximum number of generations has been exceeded, or user-constrained optimization thresholds are met.

4.2 Problem Description

The floorplanner aims at an efficient linear arrangement of module instances (Figure 2.15 and 2.16) and the assignment of a concrete implementation to each instance (out of the possible alternatives offered by the module generators, Figure 2.18). To this end, we attempt to

- minimize maximal net length between two modules;

- maximize compactibility of the entire datapath;
- minimize total length of the design (should fit into FPGA);
- minimize routing density in all horizontal channels;
- maximize homogenization of bit-slice pitch across the datapath.

Each of these criteria is evaluated separately (multi-criteria optimization, Section 4.7). Thus, the order of the criteria above does not reflect a priority.

4.3 Solution Representation

Each solution for a datapath of n module instances is represented by a sequence of tuples (a *chromosome*)

$$((l_1, g_1, a_1), \dots, (l_n, g_n, a_n)),$$

such that for $k \in \{1, \dots, n\}$, l_k is the location (*locus*) of module instance k in the linear topological placement (left-to-right), g_k is the module instance (*gene*), and a_k is a concrete implementation (*allel*). Note that the order in the linear placement may be different from the order of the tuple in the solution string. The chromosome is just an *encoding* of the actual solution for optimization purposes, different chromosomes may represent the same placement and module assignments. E.g.,

$$\begin{gathered}((3, \mathsf{Adder1}, \textit{Add-16-folded-2bplb}),\\ (1, \mathsf{Mux1}, \textit{Mux2-16-linear-2bplb}),\\ (2, \mathsf{Value1}, \textit{Reg-16-linear-2bplb}))\end{gathered}$$

describes the same placement and implementation selection as

$$\begin{gathered}((2, \mathsf{Value1}, \textit{Reg-16-linear-2bplb}),\\ (3, \mathsf{Adder1}, \textit{Add-16-folded-2bplb}),\\ (1, \mathsf{Mux1}, \textit{Mux2-16-linear-2bplb})).\end{gathered}$$

However, each chromosome will contain the same genes (all describe the same circuit, consisting of the same module-instances). The allel of each gene, however, may vary (different implementation alternative for a module-instance).

4.4 Genetic Crossover Operators

The solution space is explored by altering the chromosomes according to certain rules (*genetic operators*). The *crossover* operators create a new chromosome (*offspring*) c_3 by combining two existing, randomly selected chromosomes (*parents*) c_1, c_2.

4.4.1 Uniform Crossover

Uniform crossover first determines in the set I of all module instances occurring in the two existing solutions c_1, c_2, and creates a new chromosome c_3 consisting only of the crossover locii in c_1 (gene and allel tuple-components are left empty). Then, it scans left-to-right over all tuples in c_3. For each tuple t in c_3, a random module-instance g is selected from I. We then select, also randomly, the current allel a of g from either c_1 or c_2. The gene and allel components of t are set to g and a, respectively, and g is deleted from I.

As a result, we obtain a new chromosome c_3 that possibly has a different placement (different g, a might be assigned to the l copied from c_1), as well as a possibly different implementation assignment (a different a, either from c_1 or c_2, might be assigned to g).

Example 8 Assume

$$\begin{aligned} c_1 =&((1, \mathsf{A}, A_1), (2, \mathsf{B}, B_1), (3, \mathsf{C}, C_1), (4, \mathsf{D}, D_1)),\\ c_2 =&((1, \mathsf{B}, B_2), (2, \mathsf{C}, C_2), (4, \mathsf{A}, A_2), (3, \mathsf{D}, D_2)).\end{aligned}$$

We thus determine $I = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$. Initialization leads to

$$c_3 = ((1, \bot, \bot), (2, \bot, \bot), (3, \bot, \bot), (4, \bot, \bot)),$$

Starting a $t = c_{3_1} = (1, \bot, \bot)$, we randomly select B from I. We randomly decide to use the allel from c_2 of B, which is B_2. Thus, $c_{3_1} = (1, \mathsf{B}, B_2)$. We then delete B from I, yielding $I = \{\mathsf{A}, \mathsf{C}, \mathsf{D}\}$. By proceeding in this manner for C, A, and D, we might end up with

$$c_3 =((1, \mathsf{B}, B_2), (2, \mathsf{C}, C_1), (3, \mathsf{A}, A_1), (4, \mathsf{D}, D_2)).$$

c_3 has both changed placement and implementation assignments. However, it is still consistent in that each of the initial instances appears exactly once.

Since uniform crossover modifies locii, genes, and allels on a single tuple-basis, it is the most destructive genetic operator (no context is preserved from the initial chromosomes c_1, c_2).

4.4.2 One-Point Crossover

One-point crossover preserves more of the initial context by keeping longer subsequences of tuples from the initial chromosomes. It operates by randomly determining a crossover point $1 \le k \le n - 1$. c_3 is initialized by copying c_1, but then setting all genes and allels in tuples $c_{3_i}, k + 1 \le i \le n$ to $\bot$. Then, we scan over c_2 for all genes g not already found in $c_{3_j}, 1 \le j \le k$ (the preserved subsequence of c_1). For each such gene g and its allel a (in c_2), we set the first unused tuple of c_3 (gene is $\bot$) to g and a.

As before, we modify both placement and implementation assignment. However, in contrast to uniform crossover, the k-element subsequence of c_1 is preserved in c_3.

Example 9 Consider the c_1, c_2 of the last example. Assume $k = 2$. Thus, c_3 initially is

$$((1, \mathsf{A}, A_1), (2, \mathsf{B}, B_1), (3, \perp, \perp), (4, \perp, \perp)).$$

Scanning over c_2, we find $g = \mathsf{C}$ as a gene not found as $c_{3_j}, 1 \leq j \leq 2$. Thus, the next unused tuple of c_3, $(3, \perp, \perp)$, becomes $(3, \mathsf{C}, C_2)$. Continuing our scan over c_2, we find D as another new gene (A was skipped). The next free location in c_3 is $(4, \perp, \perp)$, now becoming $(4, \mathsf{D}, D_2)$.

The result is the offspring chromosome

$$(1, \mathsf{A}, A_1), (2, \mathsf{B}, B_1), (3, \mathsf{C}, C_2), (4, \mathsf{D}, D_2).$$

In general, placement and implementation assignments are changed, but the subsequence of the first $k = 2$ elements of c_1 has been preserved in c_3.

4.4.3 Two-Point Crossover

Two-point crossover is a generalization of one-point crossover. It preserves subsequences both at the beginning and end of c_1 in c_3. Initialization consists of randomly determining two crossover points $1 \leq k_1 < k_2 \leq n-1$, and copying c_1 to c_3, but setting all genes and allels in $c_{3_j}, k_1 \leq j < k_2$ to $\perp$. Next, we scan the entire chromosome c_2 for the genes g occurring as c_{1_j}, with $k_1 \leq j < k_2$ (between the crossover points in c_1). When we find such a gene, we copy g with its allel in c_2 to the next unoccupied ($\perp$) position in c_3 (keeping the locus initially copied from c_1). The positions of $c_{3_j}, k_2 \leq j \leq n$ are left untouched (resulting in the preservation of a c_1 subsequence at the end of c_3).

Example 10 We will continue to use the c_1, c_2 of the last example. Assume $k_1 = 2$, $k_2 = 3$. Thus, c_3 is initialized to

$$((1, \mathsf{A}, A_1), (2, \perp, \perp), (3, \mathsf{C}, C_1), (4, \mathsf{D}, D_1)).$$

We then commence our scan of c_2 for genes occurring in c_1 between the crossover points. We find such a gene $g = \mathsf{B}$ at c_{2_1} (B occurs in c_{1_2}, between the crossover points). We copy g with its allel B_2 to the next unoccupied position $(2, \perp, \perp)$ in c_3, resulting in

$$c_3 = ((1, \mathsf{A}, ((1, \mathsf{A}, A_1), (2, \mathsf{B}, B_2), (3, \mathsf{C}, C_1), (4, \mathsf{D}, D_1)).$$

Since the crossover sequence in this example only has a length of 1, the crossover operation finishes at this point.

The subsequences of c_1 lying outside the crossover region have been preserved, while all those genes originally occurring inside may have been assigned new locii and allels.

4.5 Genetic Mutation Operators

While crossover operates on two different chromosomes, *mutation* alters a single chromosome.

4.5.1 Allel Mutation

The allel mutation operator randomly changes the currently selected implementation to another one of the alternatives proposed by the module generator for a randomly selected instance (gene). All placement information is left intact.

Example 11 As before, we will reuse the initial chromosome c_1 from the previous example. Assume that we randomly select $k = 3$, and now modify the allel of c_{1_k} = (3, C, C_1). Furthermore, assuming the existence of an alternate implementation C_7, the post-mutation chromosome could become

$$c'_1 = ((1, \mathsf{A}, A_1), (2, \mathsf{B}, B_1), (3, \mathsf{C}, C_7), (4, \mathsf{D}, D_1)).$$

4.5.2 Position Mutation

The position mutation operator randomly exchanges two gene-allel tuples in the target chromosome. In this manner, only placement is modified, implementation selections are left untouched.

Example 12 Re-using the chromosome c_1, we randomly select $1 \leq j < k \leq n$, and exchange the genes and allels of c_{1_j} and c_{1_k}. Assuming that $j = 2, k = 4$, the post-mutations chromosome becomes

$$c'_1 = ((1, \mathsf{A}, A_1), (4, \mathsf{D}, D_1), (3, \mathsf{C}, C_1), (2, \mathsf{B}, B_1)).$$

4.5.3 Translocation Mutation

Translocation mutation operates by moving an entire, randomly determined subsequence of genes and allels (note: not locii) to a random position on the chromosome, changing the placement. In this manner, the partial solution represented by the subsequence is preserved. The same effect could be achieved by the position mutation operator. However, it would need to be applied numerous times, and would most likely disrupt the partial solution in the process. Translocation is thus similar to one- and two-point crossover in that it operates on a larger context than single genes and allels.

Example 13 We randomly determine $1 \leq j \leq k < l \leq n$. j, k will delimit the subsequence to move, l is the target position of the tail. For c_1, assume we have $j = 2, k = 3, l = 4$. We thus move the genes and allels of the subsequence (((2, B, B_1), (3, C, C_1)), such that this partial solution ends at position $k = 4$ in the mutated chromosome. All genes and allels formerly occupying these positions will be moved to the left. After mutation, we thus have

$$c'_1 = ((1, \mathsf{A}, A_1), (2, \mathsf{D}, D_1), (3, \mathsf{B}, B_1), (4, \mathsf{C}, C_1)).$$

4.5.4 Reversal Mutation

Reversal is the last mutation operator of the floorplanner. It reverses the order of genes of allels within a subsequence (changing their placement, since locii are preserved). As before, the same effect could be achieved by the position operator. However, the next-to relations in the encoded subsequence (which are kept intact during reversal), would probably be disrupted.

Example 14 To apply a reversal mutation, we randomly determine $1 \leq j < k \leq n$, which mark the boundaries of the subsequence to reverse. Assume $j = 2, k = 4$, and our well-known c_1. Using these parameters, we obtain

$$c'_1 = ((1, \mathsf{A}, A_1), (2, \mathsf{D}, D_1), (3, \mathsf{C}, C_1), (4, \mathsf{B}, B_1)).$$

4.6 Genetic Inversion Operator

All of the preceding operators (both crossover and mutation) modify the solution. However, the solution representation itself (the string of tuples, Section 4.3) is not changed. Due to the nature of the genetic operators (operating on single tuples or substrings of tuples), it might be advantageous to change the *encoding* of the solution (not the solution itself). This is achieved by the inversion operator, which randomly exchanges entire tuples (locus, gene, allel) in the chromosome. Note that this changes only the representation of the same solution.

Example 15 Assume for c_1, and for some randomly selected j, k with $1 \leq j < k \leq n$, current values $j = 2, k = 4$. We now swap the entire tuples c_{1_j} and c_{1_k}, resulting in a post-inversion chromosome

$$c^i_1 = ((1, \mathsf{A}, A_1),(4, \mathsf{D}, D_1)\ , (3, \mathsf{C}, C_1), (2, \mathsf{B}, B_1)).$$

c^i_1 represents the same solution (topological left-to-right placement and implementation selection) as c_1, but the changed encoding will alter the effects of the genetic operators.

4.7 Multi-Criteria Evaluation

After creating new chromosomes by crossover, or altering existing chromosomes by mutation, we determine the quality of a given solution (chromosome) by evaluating it according to certain criteria and use these results to decide whether the solution should be considered further, or discarded (Section 4.8).

To compute the quality of a given solution (chromosome), we evaluate it in terms of

- net delays,
- compactibility,
- fitting inside FPGA boundaries, and
- routability.

4.7.1 Net Delays

We will attempt to minimize the length of the longest net between two modules. To this end, the delay in ns t_d of each two-terminal net is estimated by

$$t_d = \begin{cases} 1.30 + 0.24 \cdot d & 1 \leq d \leq 6 \\ 2.20 & 7 \leq d \leq 10 \\ 3.75 & d > 10, \end{cases}$$

where d is the horizontal distance in logic blocks between two ports. The first estimate reflects a connection using the general interconnection network (switch matrix-based, single-length and double-length lines), the second estimate considers routing on a horizontal long line (without tri-state buffers, within one chip-half) and the last one a connection via horizontal long line (without tri-state buffers, crossing the chip-half boundary). Note that in contrast to the more precise delay models used e.g., in Section 7.5, we do not consider direct connections (routing by abutment) at the floorplan level. The delays were experimentally determined [Ditt95] [Sade95] for Xilinx XC4010-5 FPGAs [Xili94e].

4.7.2 Compactibility

The computation of a precise measure for compactibility would require an entire iteration of the compaction process (Chapter 6), including local logic synthesis, and then calculate the ratio of pre- to post-compaction area and delay. However, since this procedure is far too complex to be performed for each iteration of the genetic algorithm, we estimate compactibility as follows: For each contiguous section of soft-macros (Explanation 5) in the current topological placement (Section 2.6.3), we examine the constituent modules.

Each soft-macro implementation alternative is classified in the module generator templates as *primitive* or *complex*: A primitive soft-macro consists only of very simple logic (fundamental logic functions, e.g., AND, OR, NOT, etc.). A complex soft-macro implements a more complicated function (MUX, SHIFT, etc.), but without employing FPGA-specific features (carry-logic, memory blocks).

When a contiguous soft-macro section contains only primitive modules, we estimate that compaction will reduce its length at most to 1/3 of its pre-compaction length, but not below the length of the longest module in the pre-compaction section.

If the contiguous soft-macro section under analysis contains at least one complex soft-macro, we assume a best-case compaction to 1/2 of its original length, also with the pre-compaction length of the longest module as lower bound.

The rationales behind these estimates are:

- The simple logic functions of primitive macros waste a lot of logic block capacity, and thus compact very well.
- Since complex soft-macros implement more complicated functions, they already have a better logic block utilization, and thus do not compact as much.
- We assume that each module is composed as efficiently as possible. Thus, if a module needed more than a single logic block in length before compaction, the logic within is most likely too complex to be reduced further during compaction.

4.7.3 Fit into Target FPGA

Solutions with a length exceeding the length of the target FPGA are penalized proportionally to the excess length

$$l_e = l_{solution} - l_{FPGA}$$

4.7.4 Routability

As described in the introduction and Section 2.1, FPGAs have fixed routing channel width. A solution requiring more routing resources will most likely be unroutable or significantly slower. We compute an estimate of the required channel width by calculating the total number of horizontal inter-module connections passing by each column. When this number exceeds a certain fraction of all available routing resources, we penalize the solution proportionally to the excess channel width

$$c_e = c_{solution} - c_{limit}$$

Given the available horizontal routing resources per logic block of 10 single- and double-length lines, and 5 horizontal long lines (Section 2.1), we estimate that 2/3s will be available for inter-module connections, while the remaining 1/3 is assumed to be used within each module. A more precise metric could use the actual module-specific routing density specifications (`CHANNEL` and `LONGLINE`, [Ditt95] [Sade95]), but this data is not evaluated in the current floorplanner. Thus, we currently use $c_{limit} = h_{dp} \cdot 10$, where h_{dp} is the height of the datapath (Section 2.3.1).

4.8 Selection

The usual way to deal with the evaluation of multiple criteria is the computation of a weighted sum, which then determines the overall quality of a solution. However, the optimization is then heavily influenced by the weight of each criterion. As shown in [EsKu96], this can lead to sub-optimal solutions, especially when the user is unfamiliar with the optimization behavior of the concrete problem.

Since a genetic algorithm considers a population of multiple solutions in parallel, it can employ a different strategy for true multi-criteria optimization.

After the genetic algorithm has modified the existing population using crossover (Section 4.2), mutation (Section 4.5), and possibly inversion (Section 4.6) operators, each chromosome is then evaluated according to Section 4.7.

The following selection step determines the solutions that survive into the next generation of the optimization process, and discards all others found unfit.

For multi-criteria optimization, we perform selection separately for each evaluation criterion combined with selection based on a weighted sum. Thus, in addition to the solutions that seem to have an overall good quality (at least according to the current weights), we also consider solutions that objectively (independent of weight choices) have a high quality in each specific criterion. The intent is to allow these specialized solutions to eventually evolve into even better general solutions (with an overall good quality).

The floorplanner applies a combination of the following strategies to select the next solution generation.

4.8.1 Elite Selection

For each criterion (including the overall quality), the elite approach selects the n_{elite} best solutions for survival. While this would seem to be the only reasonable procedure (we are, after all, interested in obtaining the best possible solution), its use as the sole selector could lead to premature convergence of the genetic algorithm (due to insufficient diversity in the “gene pool” of the population). The result could be a sub-optimal solution.

4.8.2 Expected Value Selection

To increase the genetic diversity in the population, we select n_{expect} chromosomes, such that the number of occurrences of a single chromosome is proportional to its relative fitness. This approach first computes the fitness sum F_q over all chromosomes for each quality criterion q. Each chromosome k, with a fitness $f_q(k)$ is then transferred into the next generation in $\lfloor n_{expect} \cdot f_q(k)/F_q \rfloor$ copies. Thus, the number of occurrences of k in the next generation matches its relative fitness in the current generation. When the number of individuals selected in this manner is smaller than n_{expect}, we fill the remainder with randomly selected individuals. In contrast to elite selection, individuals with lower quality have a chance of surviving (albeit only in numbers proportional to their low fitness).

Example 16 Assume that the current population contains the individuals $\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ with $f_q(\mathsf{A}) = 4,\ f_q(\mathsf{B}) = 3,\ f_q(\mathsf{C}) = 2,\ f_q(\mathsf{D}) = 1$. Using selection by expected value, we compute $F_q = 10$. Assuming we want to select $n_{expect} = 8$ individuals, A survives in $\lfloor n_{expect} \cdot f_q(\mathsf{A})/F_q \rfloor = \lfloor 8 \cdot 4/10 \rfloor = 3$ copies, B in $\lfloor 8 \cdot 3/10 \rfloor = 2$ copies, C in $\lfloor 8 \cdot 2/10 \rfloor = 1$, and D in $\lfloor 8 \cdot 1/10 \rfloor = 0$ copies (D is discarded). Since we have only obtained $3 + 2 + 1$ individuals in this manner, we randomly pick another two chromosomes to create a selection of n_{expect} individuals. E.g., we might pick B and D, resulting in the next generation {A, A, A, B, B, C, B, D} (note that populations are multi-sets).

4.8.3 Fitness Selection

For even greater diversity, we introduce a random factor when defining the selection by fitness. Here, we randomly pick a chromosome k from the current population. It has a chance $f_q(k)/F_q$ (defined in previous section) to survive into the next generation. We repeat this pick-and-gamble process until we have obtained n_{fit} survivors. In this manner, even solutions with very low quality just might survive, and thus provide diversity for the optimization. On the other hand, even very good solutions could be discarded purely by chance. While selection by fitness helps to avoid premature convergence, relying on it as the sole selection method would lead to very slow convergence.

4.8.4 Random Selection

Random selection disregards fitness altogether. It just selects n_{random} individuals for survival in the next generation. While this approach guarantees the greatest diversity, it does not converge at all. Its primary use lies in completing a generation left under-populated by one of the other selection schemes (e.g., Section 4.8.2).

4.9 Parameters and Dynamic Fuzzy-Control

The operation of the genetic algorithm is controlled by many parameters. Among these are, e.g.,

- the number of individuals in the population,
- the application probabilities for each crossover operator,
- the application probabilities for each mutation operator,
- the probability of performing inversion,
- the number of individuals to select using each selection method,
- the maximum number of generations to process,
- the minimal requirements on each of the quality criteria,
- the weights for the weighted sum of quality criteria.

```
if (gaCycNum is small) then (popSize is veryLarge);
if (gaCycNum is medium) then (popSize is large);
if (gaCycNum is large) then (popSize is normal);
if (gaCycNum is immense) then (popSize is normal);
```

Figure 4.1: Decaying population size during optimization

```
if (lastIncrease is verySmall) then (mutRate is veryHigh);
if (lastIncrease is small) then (mutRate is high);
if (lastIncrease is medium) then (mutRate is medium);
if (lastIncrease is large) then (mutRate is low);
if (lastIncrease is veryLarge) then (mutRate is low);
if (lastIncrease is immense) then (mutRate is low);
```

Figure 4.2: Increasing mutation rate after reaching local optimum

While the genetic algorithm can operate with static parameters (all values fixed during the entire run-time), the dynamic adaptation of some parameters at optimization time can lead to further improvements for some circuits [Bode97]. The actual modification of parameters is performed by a fuzzy controller.

E.g., it is advantageous to begin optimization with a large population size, and then reduce it over time (when the optimization converges). Figure 4.1 shows the entries in the fuzzy rule-base expressing this behavior. `gaCycNum` is the fuzzy variable for the current generation number.

Another rule might express the following strategy: When we have just improved the best solution (according to one or more criteria), we raise the mutation rates to explore the "surroundings" of this new local optimum. In order to allow convergence, we then reduce the mutation rates again. Figure 4.2 show the appropriate fuzzy rules. The fuzzy variable `lastIncrease` records the time (in generations) since the last quality improvement.

In this manner, even more complex adaptations may be specified. [Bode97] describes a fuzzy rule base that reacts to a static population (no recent improvements) by *briefly* raising crossover and mutation rates, thus sending a momentary "shockwave" through the population (increasing diversity), but allowing the population to settle down immediately afterwards (emphasizing convergence).

4.10 Capabilities and Limitations

While the current implementation of the floorplanner performs most of its tasks according to specification, the actual performance falls somewhat short of the expectations. The main weakness is the need for multiple floorplanning runs[1] to obtain a layout of acceptable quality. The "best-so-far optimum" [RuPS91] differs wildly between runs. E.g., for the ALU (consisting of 26

---

[1] Each run consists of thousands of GA generations.

modules) of the SRISC processor [Bruc94] , floorplanning runs of 30000 generations have “best” overall quality values varying between 28.56 and 63.09 [Bode97]. This result is even more disappointing when considering that for each generation in a single run, the floorplanner already processed $5 \ldots 200$ possible solutions (depending on fuzzy-controlled population size) in parallel. While this misbehavior can be ameliorated by choosing the best layout from a number of runs (10-20 seem to provide acceptable results), the run-time requirements are no longer reasonable.

The actual implementation of FloorPlanner was performed independently of the mainstream SDI research. While FloorPlanner provides the floorplanning and selection functions required by SDI, the group responsible emphasized the general study of complex heuristics. In light of the practical results, the current optimization heuristic thus seems to be more an interesting experiment in the theory of genetic algorithms and fuzzy-control, than a successful application of these techniques to a concrete VLSI optimization problem. Due to the disbanding of the floorplanning group, further research into the cause of the quality deviation, or the implementation of an alternate heuristic, was unfortunately not possible.

Nevertheless, the fundamentals underlying SDI (such as 1-D module placement, automatic alternative selection) are not invalidated by the deficiencies of the current floorplanner. Given sufficient time and human resources, it would easily be possible to realize an alternative optimization using more efficient heuristics (e.g., [SaCh94] or simulated annealing). An optimizer relying on simulated annealing (as used in Section 7.4) might use moves consisting either of a two-exchange of modules, or the selection of an alternative implementation for a single module.

Another approach could use a simple preprocessing phase to perform alternative selection independently from floorplanning: Since we are aiming at a homogeneous bit-slice pitch, and the module generators usually offer implementations in any of the common BPLB values 1/2, 1, 2 (Explanation 3), we might just assemble a layout containing only module implementations with the minimum BPLB value over all possible implementations. In most practical cases, this procedure would make optimal implementation selections. The floorplanning would then reduce to the well-known 1-D linear arrangement problem.

5 Fundamentals for Compaction and Microplacement

This chapter will introduce a more formal notation to allow a concise formulation of algorithms later on. The informal explanations given in Chapter 2 will be refined and expanded upon here.

To this end, we will employ *clarifications* to clarify colloquially used terms (e.g., circuit and gate), and *definitions* to formally define their underlying mathematical representation (often graph-based). These definitions list the *components* of a structure and their respective *properties* (e.g., constraints and interdependencies). Many concepts will be demonstrated using selected examples.

While this chapter follows a bottom-up approach to make the material accessible even to the uninitiated reader, a basic understanding of modern VLSI design in general, and logic synthesis/physical design automation in particular, will prove helpful. Since most of the terminology and concepts defined here are widely used in design automation and synthesis, almost any introductory text may be consulted for a broader background. Suggested sources include [Sher95] [Leng90] for physical design automation, and [CoDi96] [MuBS95] for logic synthesis. For a refresher in the basic graph theory underlying many of the formal models, we refer the interested reader to standard textbooks such as [Deo74] [Maye72].

5.1 Basics

This section defines fundamental mathematical entities and introduces basic notational conventions.

Clarification 17 $\mathbb{Z}$ is the set of integers, $\mathbb{N}$ the set of natural numbers (excluding 0), $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$, $\mathbb{R}$ the set of real numbers, and $\mathbb{B}$ is the set of boolean truth values $\{\mathit{TRUE}, \mathit{FALSE}\}$. *BOOL* is the set of boolean functions of type $\mathbb{B}^n \to \mathbb{B}$. The symbol $\perp$ denotes an undefined value[1]. A *sequence* is an ordered set. A *partition* of a set is a cover by disjoint subsets, called *parts* of the partition. Given a binary relation R, with domain *dom* R and range *rg* R, $\widetilde{R}(C) = \{b \in \mathit{rg}\, R \mid \exists\, a \in C : \ (a, b) \in R\}$ is the set of *direct successors* of C.

---

[1] Note that this will differ from the classical usage $f(x) = \perp \Leftrightarrow x \notin \mathit{dom}\, f$.

Conversely, $\underset{\sim}{R}\,(C) = \{a \in dom\, R \mid \exists b \in C : (a, b) \in R\}$ is the set of *direct predecessors* of C.

Clarification 18 Scalar values will be notated using small letters $a, b, c, ...$, sets as capital letters $A, B, C, ...$ and families (sets-of-sets) using gothic script $\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, ...$ Functions returning sets will be notated with an initial capital letter. Lexical symbols are assumed to be words Σ^+ over a common alphabet Σ (we use the set of letters and digits) and notated in bold face, e.g, **a**, **b**, **carry**, ... Names of entities are notated in a sans-serif font, e.g. mux, sel, add2, ...

When referring to components of systems defined as tuples, e.g., $G = (V, E)$, we will add the name of the system in angle brackets if ambiguities exist. Using this convention, V could be clarified as $V\langle G\rangle$.

Example 19 $\mathbf{a} = a \in A \in \mathfrak{A}$, where the value of the variable a is the lexical symbol **a**.

5.2 Structure and Behavior of Digital Circuits

In this section, we introduce graph-based representations for hierarchical, regular digital circuits.

Clarification 20 In SDI, *hierarchy* signifies a *containment* hierarchy of circuits composed from subcircuits. *Regularity* is the multiple occurence of subcircuits with a common structure and behavior (Section 2.2, [Leng86]). Each occurence is termed an *instance* of a common *master*.

All representations will be based on graphs extended with application-specific components.

Definition 21 An *extended graph* $G = (V, E, c_1, \ldots, c_n)$ is a graph (V, E) extended with additional components $c_1, \ldots, c_n$. ■

We assume that different extended graphs may be substituted for one another as long as they have the required components (similar to inheritance in modern programming languages). E.g., if a structure is defined as a "class of extensions of (V, E, A, B)", the set may as well contain extended graphs of the forms, e.g., (V, E, A, B, Q), or (V, E, A, B, Q, Z), but not, e.g., (V, E, A, X).

5.2.1 Hierarchy

The next two definitions allow the description of hierarchical structures. This is done by imposing constraints on the components of the extended graph of Definition 21.

Definition 22 A *graph hierarchy* $(T, hier)$ consists of a *hierarchy function* $hier$ defined on a set $T = \{G_1, \ldots, G_n\}$ of extended graphs such that $hier(G_i) : V\langle G_i\rangle \rightarrow T \cup \{\perp\}$, $1 \leq i \leq n$. $hier(G_i)(v)$ descends into the *node graph* underlying a node $v \in V\langle G_i\rangle$. If v is a leaf node, $hier(G_i)(v) = \perp$. The tree T_G induced by the repeated application of $hier$ is the *hierarchy tree* $T_G = (V, E)$, with $V = T$ and

$$E = \{(G, hier(G)(v)) \mid G \in T \wedge v \in V\langle G\rangle \wedge hier(G)(v) \neq \perp\}$$

G_T is the class of all hierarchy trees. ■

Definition 23 Given a hierarchy tree T_G, $r(T_G) \in V\langle T_G\rangle$ is the *root* of T_G.

In order to refer to any node within the node graphs of the hierarchy tree, we need to introduce a hierarchical naming scheme.

Definition 24 A *hierarchical name* has the form "(*graph* '/')* (*node* '/')* *node*" in BNF. The semantics for $G_i/v_1/v_2/\ldots/v_k$ are $v_1 \in V\langle G_i\rangle$, $v_2 \in V\langle hier(v_1)\rangle$, $\ldots$, and $v_k \in V\langle hier(v_{k-1})\rangle$. ■

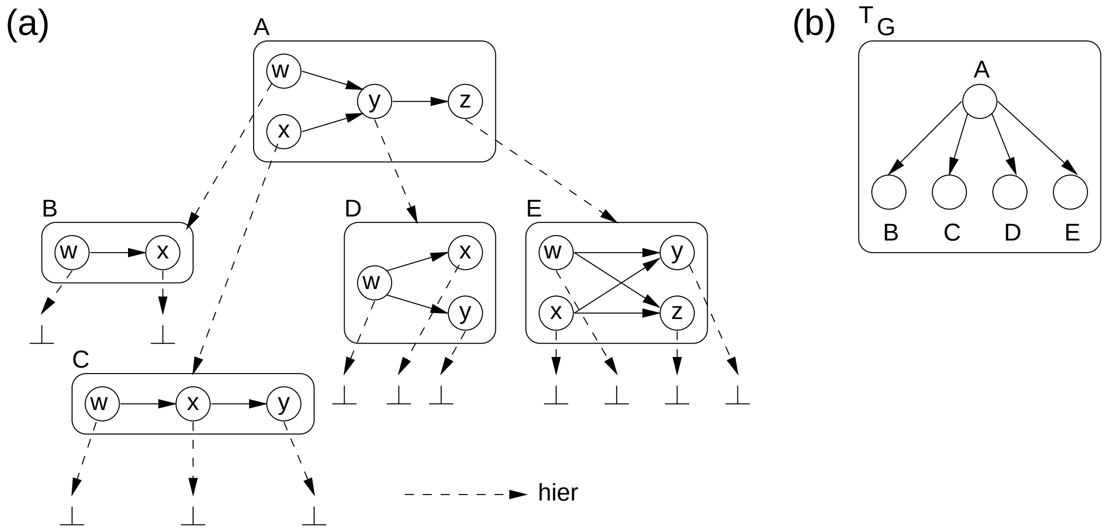


Figure 5.1: (a) Graph hierarchy T and (b) hierarchy tree T_G

Example 25 The graph hierarchy T shown in Figure 5.1.a has $T = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$. The *hier*-relations of the node graphs induce the hierarchy tree T_G in Figure 5.1.b. The hierarchical name of the top-left node in A is A/w. Its underlying node graph is $hier(\mathsf{A/w}) = \mathsf{B}$, with $hier(\mathsf{A/w/w}) = \perp$. Thus, the node w in the node graph B is a leaf node. Analogously, the rightmost node in A is A/z, with an underlying node graph $hier(\mathsf{A/z}) = \mathsf{E}$, and $V\langle\mathsf{E}\rangle = \{\mathsf{w}, \mathsf{x}, \mathsf{y}, \mathsf{z}\}$ (relative to E), or $V\langle\mathsf{E}\rangle = \{\mathsf{z/w}, \mathsf{z/x}, \mathsf{z/y}, \mathsf{z/z}\}$ (relative to A), or $V\langle\mathsf{E}\rangle = \{\mathsf{E/w}, \mathsf{E/x}, \mathsf{E/y}, \mathsf{E/z}\}$ (relative to $\mathsf{T_G}$), or $V\langle\mathsf{E}\rangle = \{\mathsf{T_G/E/w}, \mathsf{T_G/E/x}, \mathsf{T_G/E/y}, \mathsf{T_G/E/z}\}$ (absolute).

5.2.2 Regularity

We now extend our hierarchical structures with additional components to allow the description of regularity relations.

Definition 26 We define the class G_R of regular hierarchy trees with roots R_R recursively as follows:

A *regular hierarchy tree* $R_G = (T_G, Tmaster, master)$ is a hierarchy tree T_G in which for all node graphs $G' \in V\langle T_G\rangle$

$$Tmaster(G') = (U, itno) \in (G_R \cup \{\bot\}) \times (\mathbb{N}_0 \cup \{\bot\})$$
$$master(G') = (mr, itno) \in (R_R \cup \{\bot\}) \times (\mathbb{N}_0 \cup \{\bot\})$$

defines the *master tree* U, the *master root* mr and the *iteration number* $itno$ such that

- The hierarchy subtree U' rooted at G' is isomorphic to the hierarchy subtree U rooted at $mr = corr_G(G') = r(U)$ by some *correspondence* $corr_G : V\langle U'\rangle \to V\langle U\rangle$.
- For all node graphs G'' in U' and the corresponding $G = corr_G(G'')$ in U, G'' and G are isomorphic by some correspondence $corr_V : V\langle G''\rangle \to V\langle G\rangle$.

No subtree U' may contain a node graph that occurs as a master root when transitively applying *master* to the G'' in U'.

U' is termed an *instance* of U. $master(G) = \bot$ means that no such relationship between hierarchy subtrees is defined. ■

While Definition 26 allows to express the master-instance relation, regular circuits are composed by the multiple occurence of subcircuits. These can be grouped by their common master, and identified by different iteration numbers.

Definition 27 Building on Definition 26, for a subtree U' rooted at G', $master(G') = (mr, itno)$, and U as the subtree rooted at mr: If $itno \in \mathbb{N}_0$, U' is part of a related set of instances. In this context, U' is called an *iteration* of U. If $itno = \bot$, no such relationship is defined.

Note that Definition 26 imposes constraints only on the nodes and edges (isomorphism), and on the component $master(G)$. Constraints on other extension components $c_1, \ldots, c_n$ will have to be expressed on a case-by-case base.

Example 28 Figure 5.2.a shows a regularity tree R_G. With the exception of R_G/D and R_G/E, all node graphs $Q \in V\langle R_\mathsf{G}\rangle \cup V\langle R_\mathsf{H}\rangle \cup V\langle R_\mathsf{I}\rangle$ are assumed to have $master(Q) = (\bot, \bot)$.

For $R_\mathsf{G}/\mathsf{D} \in V\langle R_\mathsf{G}\rangle$, $master(R_\mathsf{G}/\mathsf{D}) = (R_\mathsf{H}/\mathsf{X}, \bot)$. The regular hierarchy subtree U' rooted at R_G/D, with $V\langle U'\rangle = \{R_\mathsf{G}/\mathsf{D}, R_\mathsf{G}/\mathsf{E}\}$, is isomorphic to the regular hierarchy tree U rooted at $mr\langle master(R_\mathsf{G}/\mathsf{D})\rangle = R_\mathsf{H}/\mathsf{X}$, with $V\langle U\rangle = \{R_\mathsf{H}/\mathsf{X}, R_\mathsf{H}/\mathsf{Y}\}$.

Furthermore, each of the node graphs in $V\langle U'\rangle$ is isomorphic to the corresponding node graphs in $V\langle U\rangle$. E.g., the node graph corresponding to R_G/D in the instance is $corr_G(R_\mathsf{G}/\mathsf{D}) = R_\mathsf{H}/\mathsf{X}$ in the master. The node in the master corresponding to the upper-right node $R_\mathsf{G}/\mathsf{D}/\mathsf{x}$ in D of the instance is $corr_V(R_\mathsf{G}/\mathsf{D}/\mathsf{x}) = R_\mathsf{H}/\mathsf{X}/\mathsf{x}$.

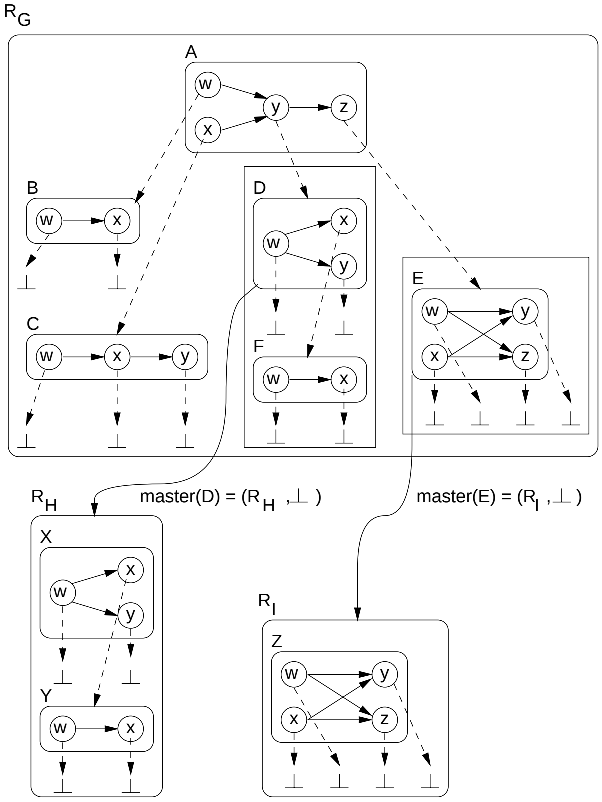


Figure 5.2: Regularity tree R_G.

5.2.3 Grouping Bits

Datapaths usually don't operate on single-bit quantities, but on multi-bit words representing scalar values like numbers, characters, pixels in an image etc. To generate fast layouts for the processing units, this parallel evaluation of multiple associated bits should be modelled.

Clarification 29 A *word* w with a *width* $n \in \mathbb{N}$ consists of n separate boolean values w_i, $0 \leq i \leq n-1$, which determine the *value* $x(w)$ of w. This is done by assigning each w_i a *significance* $s_i \in \mathbb{Z}$. $x(w)$ is then computed as $x(w) = \sum_{i=0}^{n-1} 2^{s_i} \cdot w_i$. Note that this definition allows to express dual fractions (using $s_i < 0$).

Example 30 The value 4.75 can be expressed as the 5-bit word $w = 10011$ when viewing the rightmost bit as w_0, and using $s_0 = -2$, $s_1 = -1$, $s_2 = 0$, $s_3 = 1$, $s_4 = 2$. w is then evaluated as $x(w) = 1 \cdot 2^2 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 4 + 0.5 + 0.25 = 4.75$.

Definition 31 A *variable* $t = (sym, sig) \in \Sigma^+ \times (\mathbb{Z} \cup \{\perp\})$ consists of the *symbol* $sym\langle t\rangle$, representing a boolean value, and its *significance* $sig\langle t\rangle$, which may be undefined. It may be abbreviated as $sym\langle t\rangle_{sig\langle t\rangle}$, with the subscript being ommitted if $sig\langle t\rangle = \perp$. ■

5.2.4 Structure of Circuits: Network Skeleton

With the concepts defined thus far, we can now model the structure (interconnection pattern) of hierarchical, regular circuits processing multi-bit words.

Clarification 32 In context of this text, a digital circuit of interconnected hardware *units* is called a *network*. A unit u applies $n(u) \in \mathbb{N}_0$ local *boolean functions* of type $\mathbb{B}^{m(u)} \rightarrow \mathbb{B}$ to $m(u) \in \mathbb{N}_0$ *inputs* to compute $n(u) \in \mathbb{N}_0$ *outputs*. Inputs and outputs of units are collectively referred to as *terminals*. They form the endpoints of *connections* between units within the network, such that at most one output terminal connects to zero or more input terminals[2].

Primary ports are inputs or outputs that are externally visible and could be used to connect a given circuit to other circuits.

Networks that do not contain storage elements (flip-flops, latches, etc.), but only boolean functions, are called *combinational*, otherwise they are called *sequential*. When formulating functions, the "=" operator is used to denote a combinational evaluation, while ":=" indicates the presence of a D-flipflop (the only storage element treated in this text). This is still only a structural notation, and should be identified with the assignment operator of imperative programming languages.

The hierarchy and interconnection pattern between units in a network determine the *connectivity structure* of a circuit. Together with the local functions of its units, it determines the *behavior* of circuit. This behavior could be expressed as a global function of *primary input ports* (PI) to *primary output ports* (PO).

Example 33 $y = ab$ is a combinational function computing the state of the output y as a logic "and" of the states of the inputs a, b. $Q := D$ models a basic D-flipflop (the clock signal is not considered explicitly). A D-flipflop with clock enable e could be expressed as $Q := e'Q + eD$[3].

Definition 34 A *network skeleton* $N_S = (V, E, Ttn, In, Pi, Po, Ui, Uo, ptype, loc, h, l)$ is an extended graph.

1. A node in V is a *unit* or a *primary port*, with V being partitioned into the set of *primary input ports* Pi, the set of *primary output ports* Po, and the set of units In.
2. A node $v \in V$ has a set of variables partitioned into the *input variables* $Ui(v)$ and the *output variables* $Uo(v)$. Primary inputs have only a single output variable and no input variables, primary outputs have only a single input variable and no output variables.
3. For a node $v \in V$, $To(v) = \{v\} \times Uo(v)$ are its *output terminals*, $Ti(v) = \{v\} \times Ui(v)$ are its *input terminals*.

---

[2] Note that dangling (unconnected) terminals may exist. [3] The prime (') indicates inversion (logical "not").

4. $Ttn \subseteq \{(t_o, t_i) \mid \exists\, u, v \in V : \ t_o \in To(u) \wedge t_i \in Ti(v)\}$ are the *two-terminal nets*. For any input terminal t_i, we must have $|\underset{\sim}{Ttn}\,(t_i)| \leq 1$.
5. The edges are $E = \{(u, v) \mid \exists\, p, q : \ ((u, p), (v, q)) \in Ttn\}$.
6. $ptype : Pi \cup Po \rightarrow \{control, data, hybrid\}$ determines the *port type* of each primary port.
7. $loc : V \rightarrow (\mathbb{Z} \cup \{\bot\})^2$ assigns each node of N_S a *location* (x, y) (Section 5.3.1).
8. $h : V \rightarrow (\mathbb{N}_0 \cup \{\bot\})$ is the *height* of each node (Section 5.3.1).
9. $l : V \rightarrow (\mathbb{N}_0 \cup \{\bot\})$ is its *length* (Section 5.3.1).

■

The constraint on the TTNs prevents short circuits by allowing at most a single output terminal to connect to a given input terminal. An input terminal may remain unconnected, however.

Data ports are usually connected horizontally within the datapath (Section 2.3.1). However, exceptions like carry-chains or the data propagation in a shift-register have a vertical data flow. A control port always has a vertical signal flow and can be used to connect the datapath to the external controller. Note the hybrid ports that connect to both the datapath and the controller.

Definition 35 A *network skeleton tree* $T_S = (R_G, port)$ consists of a regular hierarchy tree R_G and the functional *port*.

1. All node graphs in R_G are network skeletons.
2. In any node graph G of R_G, a unit $v \in In\langle G\rangle$ with $H = hier(v) \neq \bot$ is a non-leaf unit. It must have $Ui(v) = \widetilde{Uo}(Pi\langle H\rangle)$ and $Uo(v) = \widetilde{Ui}(Po\langle H\rangle)$.

For the regularity relations in R_G, we have to constrain the network skeletons:

3. Given any instance network skeleton I and its master network skeleton $M = corr_G(I)$, the two-terminal nets correspond by

$$((u, p), (v, q)) \in Ttn\langle I\rangle \Leftrightarrow ((corr_V(u), p), (corr_V(v), q)) \in Ttn\langle M\rangle.$$

4. The instance primary inputs have $\widetilde{corr}_V(Pi\langle I\rangle) = Pi\langle M\rangle$ (analogously for outputs).
5. The unit input variables (output variables) ports are constrained by $Ui\langle I\rangle = Ui\langle M\rangle \circ corr_V$ ($Uo\langle I\rangle = Uo\langle M\rangle \circ corr_V$).
6. Port types also have $ptype\langle I\rangle = ptype\langle M\rangle \circ corr_V$.

For each node graph G_i, the functional $port\langle G_i\rangle$ assigns each non-leaf unit $v \in In\langle G_i\rangle$ a function

$$p(v) : Ui\langle G_i\rangle(v) \cup Uo\langle G_i\rangle(v) \rightarrow Pi\langle hier(v)\rangle \cup Po\langle hier(v)\rangle,$$

which maps the variables of a terminal to the corresponding primary ports at the next lower hierarchy level[4]. For leaf units v, $p(v) = \perp$. ■

Definition 36 As a shorthand for working with terminals $t = (u, (x, b))$, we define $sig(t) = b$, $sym(t) = x$, and $nod(t) = u$. ■

Definition 37 The set of *terminal fanouts* for an output terminal t_o is defined as $Pfo(t_o) = \widetilde{Ttn}(t_o)$. Conversely, the set of *terminal fanins* for an input terminal t_i is defined as $Pfi(t_i) = \underset{\sim}{Ttn}(t_i)$.

The set of *fanouts* of a unit u is defined as $Fo(u) = \widetilde{E}(u)$, the set of *fanins* is $Fi(v) = \underset{\sim}{E}(v)$. ■

Definition 38 For an output terminal t_o in a network skeleton, $n = \{t_o\} \times Pfo(t_o)$ is a *net* n with *pins* $Pins(n) = \{t_o\} \cup Pfo(t_o)$, *source* $source(n) = t_o$, and *sinks* $Sinks(n) = Pfo(t_o)$. A *multi-terminal net* (MTN) is a net that may have $|Sinks(n)| \geq 1$. A two-terminal net always has $|Sinks(n)| = 1$. ■

Observe, that in this definition, N_S is not required to be acyclic. This will be used later on to allow feedback on sequential elements.

Example 39 In Figure 5.3, two network skeletons A, B are used as node graphs in a larger regular hierarchy tree (not shown). E.g.,

$$V\langle \mathsf{A}\rangle = \{\mathsf{A/a}, \mathsf{A/b}, \mathsf{A/c}, \mathsf{A/d}, \mathsf{A/e}, \mathsf{A/f}, \mathsf{A/g}\}$$

, with $Pi\langle \mathsf{A}\rangle = \{\mathsf{A/a}, \mathsf{A/b}\}$, and $Po\langle \mathsf{A}\rangle = \{\mathsf{A/f}, \mathsf{A/g}\}$.

Consider the unit $\mathsf{A/c} \in In\langle \mathsf{A}\rangle$: it has $Ui\langle \mathsf{A/c}\rangle = \{(\mathbf{a}, 0), (\mathbf{a}, 1)\}$, and $Uo\langle \mathsf{A/c}\rangle = \{(\mathbf{x}, \perp), (\mathbf{y}, 0)\}$. This leads, e.g., to the output terminals $To\langle \mathsf{A/c}\rangle = \{(\mathsf{A/c}, (\mathbf{x}, \perp)), (\mathsf{A/c}, (\mathbf{y}, 0))\}$.

Using this notation, the two-terminal net $((\mathsf{A/b}, (\mathbf{a}, 0)), (\mathsf{A/c}, (\mathbf{a}, 0))) \in Ttn\langle \mathsf{A}\rangle$ makes the output variable $(\mathbf{a}, 0)$ of the primary input $\mathsf{A/b}$ available as the input variable $(\mathbf{a}, 0)$ of the unit $\mathsf{A/c}$. Note that the solid arrows represent TTNs, not edges. E.g., $(\mathsf{A/c}, \mathsf{A/e}) \in E\langle \mathsf{A}\rangle$. The primary ports have associated port types, e.g., $ptype\langle \mathsf{A/f}\rangle = control$.

The unit $\mathsf{A/c}$ is a non-leaf unit, since $hier(\mathsf{A/c}) = \mathsf{B}$. Note that it has $\{(\mathbf{a}, 0), (\mathbf{a}, 1)\} = Ui\langle \mathsf{A/c}\rangle = \widetilde{Uo}(Pi\langle hier(\mathsf{A/c})\rangle) = \widetilde{Uo}(Pi\langle \mathsf{B}\rangle) = \widetilde{Uo}(\{\mathsf{B/a}, \mathsf{B/b}\}) = \{(\mathbf{a}, 0), (\mathbf{a}, 1)\}$. Analogously, $Uo\langle \mathsf{A/c}\rangle = \widetilde{Ui}(Po\langle hier(\mathsf{A/c})\rangle)$.

Furthermore, examples for terminal fanouts and fanins include $Pfo((\mathsf{A/c}, (\mathbf{x}, \perp))) = \{(\mathsf{A/d}, (\mathbf{a}, \perp)), (\mathsf{A/e}, (\mathbf{a}, 0))\}$, and $Pfi((\mathsf{A/c}, (\mathbf{a}, 0))) = \{(\mathsf{A/b}, (\mathbf{a}, 0))\}$. The corresponding fanouts and fanins are $Fo(\mathsf{A/c}) = \{\mathsf{A/d}, \mathsf{A/e}\}$ and $Fi(\mathsf{A/c}) = \{\mathsf{A/a}, \mathsf{A/b}\}$.

---

[4] Note that the function nature of $p(v)$ restricts the valid choices of variables for primary ports.

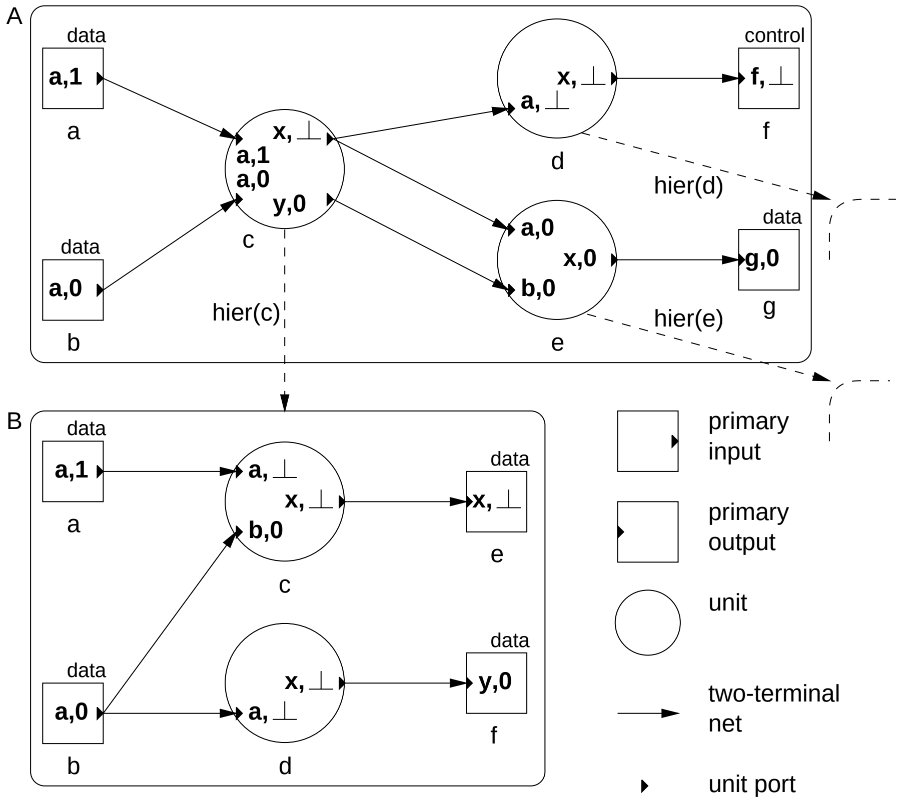


Figure 5.3: Network skeletons

An example for a multi-terminal net is $n = \{((\mathsf{A/c}, (\mathbf{x}, \bot)), (\mathsf{A/d}, (\mathbf{a}, \bot))), ((\mathsf{A/c}, ((\mathbf{x}, \bot))), (\mathsf{A/e}, (\mathbf{a}, 0)))\}$, with $Pins(n) = \{(\mathsf{A/c}, (\mathbf{x}, \bot)), (\mathsf{A/d}, (\mathbf{a}, \bot)), (\mathsf{A/e}, (\mathbf{a}, 0))\}$, $source(n) = (\mathsf{A/c}, (\mathbf{x}, \bot))$, and $Sinks(n) = \{(\mathsf{A/d}, (\mathbf{a}, \bot)),(\mathsf{A/e}, (\mathbf{a}, 0))\}$.

To given an example for the use of *port*, consider $port\langle \mathsf{A}\rangle(\mathsf{A/c})(\mathbf{a}_0) = \mathsf{B/b}$, or $port\langle \mathsf{A}\rangle(\mathsf{A/c})(\mathbf{x}) = \mathsf{B/e}$.

5.2.5 Behavior of Circuits: Network

By annotating the nodes of a network skeleton with the local functions mentioned in Clarification 32, we add a behavioral description.

Definition 40 A *network* $N = (N_S, F, seq)$ is a network skeleton N_S extended with the components $F : \widetilde{To}(In) \to BOOL$ and $seq : \widetilde{To}(In) \to \mathbb{B}$.

The functional F associates a boolean function $F(t_o)$ with each output terminal $t_o = (v, q)$, such that $F(t_o) : \mathbb{B}^{|\,Ui(v)|} \to \mathbb{B}$. The *support variables* of $F(t_o)$ are $Ui(v)$.

$seq(t_o)$ is *TRUE* iff t_o is delayed by a storage element[5]. ■

Definition 41 A *network tree* is a network skeleton tree in which exactly the leaves are networks, all other node graphs are network skeletons.

---

[5] For simplicity, we consider only D-flipflops with a common implicit clock.

For the regularity constraints, the isomorphism is extended to F and seq by

1. For the local functions of $v \in In\langle I\rangle$, $q \in Uo(v)$, we require
$$F\langle I\rangle((v,q)) = F\langle M\rangle((corr_V(v),q)).$$

2. The sequential flag is also $seq\langle I\rangle((v,q)) = seq\langle M\rangle((corr_V(v),q))$.

■

This is an SDI specific restriction: behavior is explicitly defined only in the leaves of the hierarchy, there are no hierarchical networks. Thus, in a network N and $v \in In\langle N\rangle$, $hier(v) = \perp$.

Note that the local functions themselves are not changed: They are defined in terms of the variables $Ui(v)$ and $Uo(v)$, which remain unaltered during instantiation (Definition 35).

For brevity, we will refer to the components of the network skeleton underlying a network without explicitly qualifying them with "$N_S\langle N\rangle$". Thus, in context of the network N, V refers to $V\langle N_S\langle N\rangle\rangle$.

Example 42 Consider again the network skeleton B in Figure 5.3. It can be extended into a network by defining $F\langle\mathsf{B}\rangle$ and $seq\langle\mathsf{B}\rangle$. E.g., $F\langle\mathsf{B}\rangle : (\mathsf{B/c}, (\mathbf{x}, \perp)) \mapsto g$, and $F\langle\mathsf{B}\rangle : (\mathsf{B/d}, (\mathbf{x}, \perp)) \mapsto h$, with $g = \mathbf{a}\mathbf{b}_0$, $h = \mathbf{a}'$[6]. Note that the support variables for each function are the input variables: $Ui(\mathsf{B/c}) = \{\mathbf{a}, \mathbf{b}_0\}$, $Ui(\mathsf{B/d}) = \{\mathbf{a}\}$.

By also defining $seq\langle\mathsf{B}\rangle : \{((\mathsf{B/c}, (\mathbf{x}, \perp))) \mapsto \mathit{TRUE}(\mathsf{B/d}, (\mathbf{x}, \perp)) \mapsto \mathit{FALSE}\}$, we might express that the function g is delayed by a storage element, while the function h is purely combinatorial.

Restrictions for Logic Synthesis

While the networks defined in Definition 40 are very general, common logic synthesis and technology mapping methods only work on a more restricted structure.

Definition 43 A *gate network* is a network in which

1. all units only have a single output terminal each, and

2. primary ports in the gate network are uniquely identified by their variable, and

3. for each unit $v \in In$, $\widetilde{sym}(To(v)) \cap \widetilde{sym}(Ti(v)) = \emptyset$, and

4. all variables (x,b) with a given variable symbol x either have significances $b \in \mathbb{Z}$, or all have $b = \perp$.

---

[6] We are now using the abbreviated notation for variables, see Definition 31.

∎

Items 1 and 2 allow the identification of units and primary ports by their variables, item 3 distinguishes of input and output terminals by their symbols. The use of item 4 will become evident in Definition 46.

Definition 44 A *gate network tree* is a network tree in which all leaves are gate networks.

5.2.6 Master-Slices and Slices

Bit-sliced circuits are characterized by the multiple occurence of subcircuits with the same connectivity structure and behavior, but varying bit-significances.

Definition 45 A *master-slice* $N_{ms} = (N, \mathit{voffs}, h)$ is a gate network N extended with the *vertical significance offset voffs*, and the *height h*.

1. $$\mathit{voffs} : \bigcup_{v \in V} (\mathit{To}(v) \cup \mathit{Ti}(v)) \to (\mathbb{Z}\backslash\{0\}) \cup \{\perp\}$$
2. $h \in \mathbb{N}_0$ (see Section 5.3.3).
3. The variable and offset of a terminal $t = (v, (x, b))$, with $\mathit{voffs}(t) = s$, may be referred to as $x_{b,s}$, and is called a *symbolic significance*.

∎

Definition 46 A *master-slice tree* is a gate network tree in which exactly the leaves are master-slices.

Regularity for master-slices is defined by the rules

1. Master-slices are not only instantiated, but iterated. In the instance z with iteration number i of a master-slice s in the master-slice tree, $\mathit{master}(z) = (s, i)$.
2. For an iteration s_i of a master-slice s, $h\langle s_i\rangle = h\langle s\rangle$.
3. *voffs* is not inherited by the instances, it is ignored when determining isomorphism.
4. The vertical significance offset alters the significance of variables in the networks of the instances depending on the iteration number. The significance b' of a terminal

$$(v, (x, b')) \in \bigcup_{v \in V\langle s_i\rangle} (\mathit{Ti}(v) \cup \mathit{To}(v))$$

in the instance s_i relates to its corresponding terminal

$$(corr_V(v), (x, b)) \in \bigcup_{v \in V\langle m \rangle} (Ti(v) \cup To(v))$$

in the master m as

$$b' = b + itno\langle s_i \rangle \cdot voffs\langle m \rangle((corr_V(v), (x, b))).$$

■

Note that item 1 also alters the local function definitions, since they rely on a unit's input terminals as support variables.

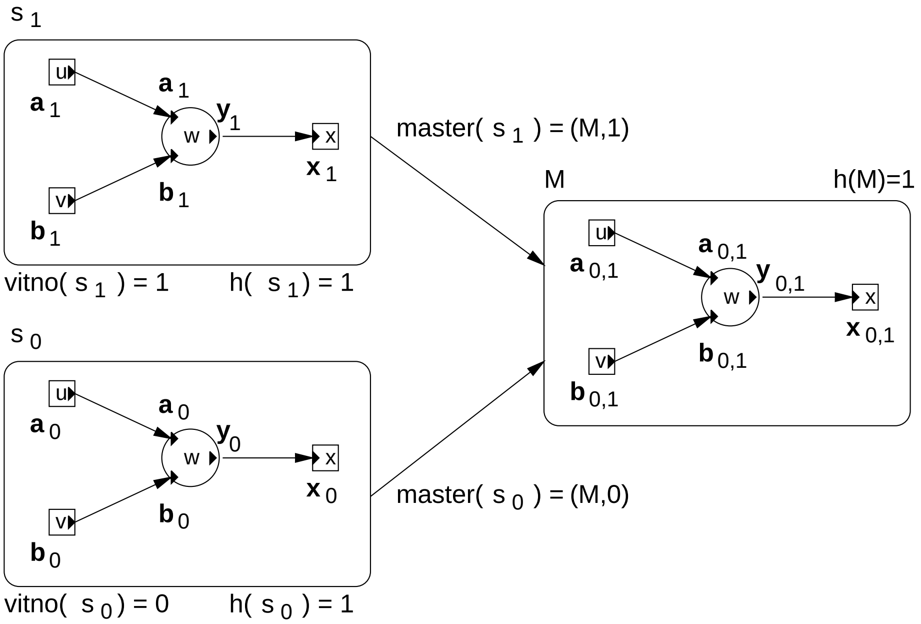


Figure 5.4: Iterating a master-slice to obtain slices

voffs assigns each terminal a value, by which its significance will be offset when the master-slice is iterated to produce slices. In Figure 5.4, M is a master-slice. Note the variables annotated with *voffs*. E.g., $voffs(\mathbf{b}_{0,1}) = 1$.

Definition 47 A *slice* N_s is a gate network that is an iteration of the master-slice $mr\langle master(N_s)\rangle$.

In slices, the iteration number component $itno\langle master(N_s)\rangle$ is referred to as *vertical iteration number*, abbreviated $vitno\langle N_s \rangle \in \mathbb{N}_0$. ■

Figure 5.4 shows two slices $\mathsf{s}_0, \mathsf{s}_1$ of the master-slice M. Note that $voffs\langle \mathsf{M} \rangle$ has not been propagated into the instances.

Definition 48 A *slice tree* is a gate network tree in which exactly the leaves are slices. ■

Example 49 In Figure 5.4, we will follow the output terminal of unit $\mathsf{M/w}$ in the master-slice M through an iteration to obtain the slice s_1, with $vitno\langle \mathsf{s}_1 \rangle = 1$. In the master-slice, the terminal is $t = (\mathsf{M/w}, (\mathbf{y}, 0))$, with $voffs\langle \mathsf{M} \rangle(t) = 1$. The corresponding terminal in s_1 is $t' = (\mathsf{s}_1/\mathsf{w}, (\mathbf{y}, b'))$. We compute its bit-significance

$$b' = sig(t) + vitno\langle \mathsf{s}_1 \rangle \cdot voffs\langle \mathsf{M} \rangle(t) = 0 + 1 \cdot 1 = 1.$$

Thus, the output terminal of s_1/w is $t' = (\mathsf{s}_1/\mathsf{w}, (\mathbf{y}, 1))$.

We proceed analogously for the input terminals to obtain $(\mathsf{s}_1/\mathsf{w}, (\mathbf{a}, 1))$, $(\mathsf{s}_1/\mathsf{w}, (\mathbf{b}, 1))$. Now assume that $F(\mathsf{M/w}, \mathbf{y}_{0,1})) = \mathbf{a}_{0,1}\mathbf{b}_{0,1}$. Since this local function uses the input terminals as support variables, and these have been changed during iteration, the corresponding function in the slice becomes $F((\mathsf{s}_1/\mathsf{w}, (\mathbf{y}, 1))) = \mathbf{a}_1\mathbf{b}_1$.

Constraints on Master-Slices

We have to impose further constraints on master-slices to allow only the linear or folded bit-sliced units as discussed in Section 2.3.1.

Definition 50 A terminal $t \in \bigcup_{v \in V\langle N_{ms} \rangle}(Ti(v) \cup To(v))$ with $voffs\langle N_{ms} \rangle(t) = \perp$ has *static* significance, otherwise it has *dynamic* symbolic significance. ■

All constraints described in this subsection apply only to dynamic significances.

Clarification 51 The set of significances $S = \{lsb, \ldots, msb\} \subset \mathbb{Z}$ is *logically complete*. It is delimited by its *least-significant bit)* $lsb(S)$ and *most-significant bit)* $msb(S)$. Two sets of significances S_1, S_2 *logically abut* if either $lsb(S_1)+1 = msb(S_2)$ or $lsb(S_2) + 1 = msb(S_1)$.

The datapaths treated in this text are required to consist only of logically complete bit-sliced units. E.g., we will not consider a unit which processes bits 1, 4, 5, 7 in an 8-bit datapath. However, a unit processing bits 0, 1, 2, 3, 4, 5, 6, 7 in a 32-bit datapath is logically complete (all significances between its LSB 0 and MSB 7 are handled). Even though it processes only a subset of the 32 bits in a word, the subset processed is contiguous.

One way to achieve this logical completeness in a bit-sliced datapath unit would be to require the significances of all bit-slices in the unit to logically abut, and each bit-slice itself to be logically complete. However, these constraints prove too restrictive to handle the valid case of a unit folded as in Figure 5.5. While the entire unit is logically complete (all significances between 0 and 7 are being processed), an individual bit-slice is not logically complete. E.g., the bottom-most bit-slice (iteration number 0) processes bits 0 and 7, but none of the bits in between. To handle the case of such alternately folded units, a more complicated model is required.

Definition 52 1. Given the master-slice m, the set of *ascending terminals* $T^+(m)$ is defined as

$$T^+(m) = \{t \in \bigcup_{v \in V\langle m \rangle} (Ti(v) \cup To(v)) \mid voffs(t) \geq 0\}.$$

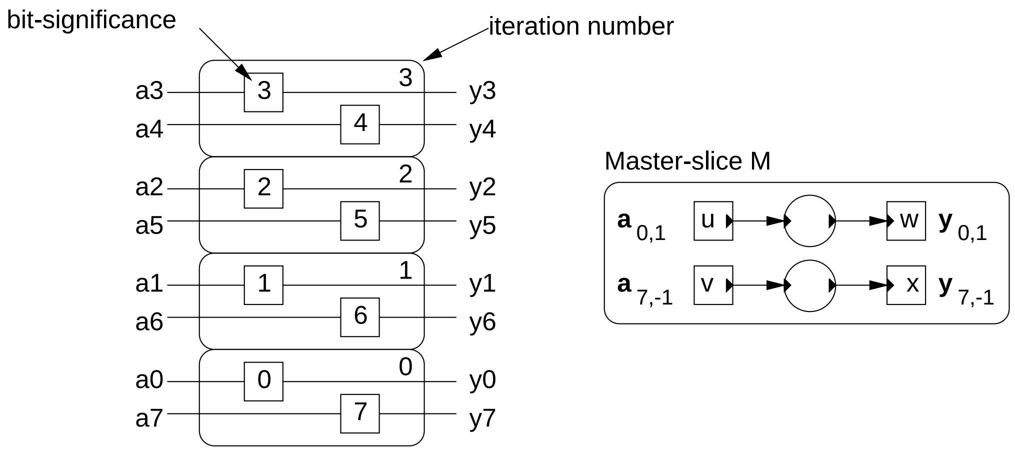


Figure 5.5: Logical completeness and abutment in an alternately folded unit

2. The set of *descending terminals* $T^{-}(m)$ is defined analogously.
3. The *least-significant ascending significance* for a symbol x is defined as

$$lsb^{+}_{sym}(x, m) = \min\{sig(t) \mid t \in T^{+}(m) \wedge sym(t) = x\}.$$

4. Conversely, the *most-significant descending significance* is defined as

$$msb^{-}_{sym}(x, m) = \max\{sig(t) \mid t \in T^{-}(m) \wedge sym(t) = x\}.$$

■

With these concepts, we can now precisely formulate the constraints on master-slices to build logically complete bit-sliced units.

Definition 53 A *valid master-slice* m is a master-slice that fulfills the following constraints:

1. All terminals $t \in T^{+}(m)$ ($t \in T^{-}(m)$) with the same symbol $x = sym(t)$ must have the same $voffs(t)$, which is abbreviated to $voffs^{+}(x)$ ($voffs^{-}(x)$).
2. For each symbol x occurring on any ascending terminal $t \in T^{+}(m)$, and all significances $b \in \{0, \ldots, voffs^{+}(x) - 1\}$, there exists a terminal

$$(v, (x, lsb^{+}_{sym}(x, m) + b)) \in T^{+}(m).$$

3. For each symbol x occurring on any ascending terminal $t \in T^{+}(m)$,

$$voffs^{+}(x) = |\{(v, (x, b)) \mid \exists\, v, b : \ (v, (x, b)) \in T^{+}(m)\}|.$$

4. For each symbol x occurring on any descending terminal $t \in T^{-}(m)$, and all significances $b \in \{voffs^{-}(x) + 1, \ldots, 0\}$, there exists a terminal

$$(v, (x, msb^{-}_{sym}(x, m) + b)) \in T^{-}(m).$$

5. For each symbol x occurring on any descending terminal $t \in T^-(m)$,

$$voffs^-(x) = -|\{(v, (x, b)) \mid \exists\, v, b : (v, (x, b)) \in T^-(m)\}|.$$

G_{ms} is the class of valid master-slices. ∎

In Definition 53, Item 1 is needed for logical completeness. Items 2 and 4 also enforce logical abutment. Items 3 and 5 are required to prevent significance collisions between different iterations.

Example 54 Assume an invalid master-slice m containing the primary port variables and vertical significance offsets $\mathbf{a}_{0,2}, \mathbf{a}_{1,2}, \mathbf{a}_{2,2}$. m fulfills Definition 53.1 and .2. However, when iterating m twice, we would find a variable $\mathbf{a}_2$ on ports both in iteration 1 (due to the altered significance for $\mathbf{a}_{0,2}$: $0 + 1 \cdot 2 = 2$) and in iteration 0 (the significance of $\mathbf{a}_{2,1}$ becomes $2 + 0 \cdot 2 = 2$ during iteration).

Enforcing Item 3 on the example, we determine that a corrected set of symbolic significances would be $(\mathbf{a}, 0, 3)$, $(\mathbf{a}, 1, 3)$, $(\mathbf{a}, 2, 3)$. Now, the significance collision doesn't occur.

From now on, we assume that all master-slices are valid.

Example 55 We now apply Definition 53 to the master-slice M in Figure 5.5. For brevity, we will consider only the primary ports. The ascending terminals are $T^+(\mathsf{M}) = \{(\mathsf{u}, \mathbf{a}_{0,1}), (\mathsf{w}, \mathbf{y}_{0,1})\}$, the descending terminals are $T^-(\mathsf{M}) = \{(\mathsf{v}, \mathbf{a}_{7,-1}), (\mathsf{x}, \mathbf{y}_{7,-1})\}$.

Consider, for example, the symbol $\mathbf{a}$: Its least-significant ascending symbolic significance is $lsb^+_{sym}(\mathbf{a}, \mathsf{M}) = 0$, while its most-significant descending symbolic significance is $msb^-_{sym}(\mathbf{a}, \mathsf{M}) = 7$.

Since each of the symbols $\{\mathbf{a}, \mathbf{y}\}$ in M occurs exactly on one ascending (descending) terminal, Definition 53.1 holds by default.

Next, we demonstrate the application of Definition 53.2 on the symbol $\mathbf{a}$. It occurs on the terminals $\{(\mathsf{u}, \mathbf{a}_{0,1}), (\mathsf{v}, \mathbf{a}_{7,-1})\}$, of which only terminal $(\mathsf{u}, \mathbf{a}_{0,1}) \in T^+(\mathsf{M})$. The interval for b is thus $\{0, \ldots, voffs^+(\mathbf{a}) - 1\} = \{0\}$.

In order to fulfill Item 2, a terminal with the symbol $\mathbf{a}$, and with the significance 0 (the sole value for b in the interval) must exist in $T^+(\mathsf{M})$. Since such a terminal does indeed exist in the form of $(\mathsf{u}, \mathbf{a}_{0,1}) \in T^+(\mathsf{M})$, we have fulfilled the requirements of Definition 53.2 for $\mathbf{a}$.

Definition 53.3 also holds, since the number of ascending terminals with the symbol $\mathbf{a}$ is 1, and the single terminal has a $voffs^+$ of 1.

By applying Definition 53.2 through 53.5 to the other symbols, we find that they hold in all cases. Thus, M is a valid master-slice.

5.2.7 V-Zones: Multi-Iteration Circuits

We will now introduce circuits composed of multiple iterations of a single master-slice. This intermediate step in the construction of circuits consisting of instances of arbitrary master-slices reduces computation times by the exploitation of regularity.

Definition 56 A v-zone $N_v = (N_\mathcal{S}, \mathit{vmaster}, \mathit{repl})$, is a network skeleton $N_\mathcal{S}$ extended with the components $\mathit{vmaster} \in G_{ms}$, and $\mathit{repl} \in \mathbb{N}$.[7] It is composed of *repl* iterations of the single master-slice *vmaster*. In context of a v-zone, a master-slice iteration is referred to as a *v-segment*.

1. $|\mathit{In}| = \mathit{repl}$.
2. Internal nodes are the v-segments with vertical iteration numbers $0, \ldots, \mathit{repl} - 1$ of the master-slice *vmaster*.

■

Consider the difference between $\mathit{vmaster}\langle N_v\rangle$ and $\mathit{master}(N_v)$. The first specifies the network that is iterated *inside* the v-zone, while the second would indicate that the entire v-zone is an instance of a "master-v-zone", a concept not required in SDI.

Definition 57 A *v-zone tree* is a slice tree that has a v-zone N_v at the root, with each internal node of N_v having an underlying (via *hier*) slice. ■

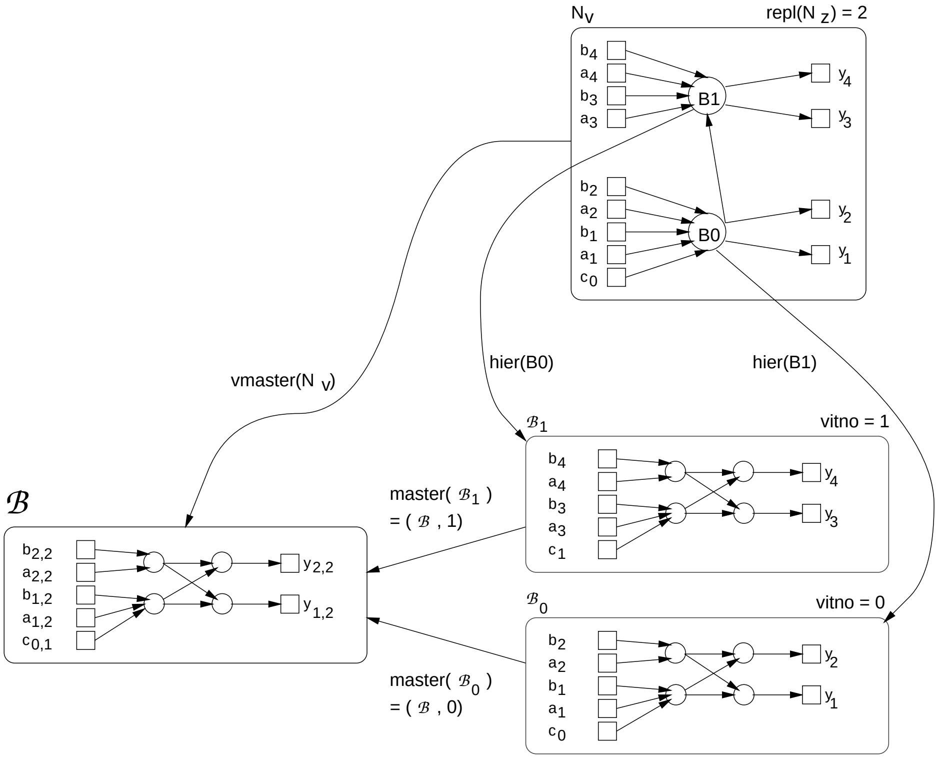


Figure 5.6: v-zone tree with root N_v, showing hierarchy and regularity

---

[7] For brevity, we ommit the "$N_\mathcal{S}\langle N_v\rangle$" qualifier (similar to the notation for components of a network).

Example 58 Figure 5.6 shows a v-zone tree rooted at $\mathsf{N_V}$, composed of two v-segments $\mathsf{B0}$, $\mathsf{B1}$ (with the same connectivity structure and behavior), and an internal connection between them (for clarity, we have ommitted the terminals in the figure). $\mathsf{N_V}$ has the $repl\langle\mathsf{N_V}\rangle = 2$ internal nodes $\mathsf{B0}, \mathsf{B1}$ representing the two v-segments and a TTN $((\mathsf{B0}, (\mathbf{y}, 2)), (\mathsf{B1}, (\mathbf{c}, 1)))$ for the inter-v-segment connection. Both nodes have the same internal connectivity structure and behavior as the master-slice $vmaster\langle\mathsf{N_V}\rangle = \mathcal{B}$. The networks underlying the v-segments of $\mathsf{N_V}$ are $hier(\mathsf{B0}) = \mathcal{B}_0$ and $hier(\mathsf{B1}) = \mathcal{B}_1$, with $vitno\langle\mathcal{B}_0\rangle = 0$ and $vitno\langle\mathcal{B}_1\rangle = 1$. Since both slices have the same master-slice, they share a common connectivity structure and behavior. This relationship is represented by $mr\langle master(\mathcal{B}_0)\rangle = mr\langle master(\mathcal{B}_1)\rangle = \mathcal{B}$.

Note again the application of Definition 53 to ensure that $\mathcal{B}$ is a valid master-slice, and the use of Definition 46 to calculate the bit-significances in the v-segments.

5.2.8 Stacks: Multi-v-zone Structures

With v-zones describing the iteration of instances of the same master-slice, and the v-zone tree constraining their hierarchical composition, we now define circuits composed of multiple v-zones, thus containing instances of different master-slices. Following our aim of exploiting regularity, we also allow the iteration of these structures. Due to the conceptual similarity to the relation between master-slices and slices, we will defer all examples and figures until Figure 5.7, which demonstrates some of these higher-level concepts in concert.

We will define a master-stack, which is a network skeleton that contains one or more v-zones, and a stack, which is a network skeleton with the same connectivity structure and behavior as its master-stack.

Definition 59 A *master-stack* $N_{mS} = (N_{\mathcal{S}}, hoffs, l, Vplace)$ is a network skeleton $N_{\mathcal{S}}$ extended with the *horizontal significance offset hoffs*, the *length l*, and the *vertical topological placement Vplace*.

1. $hoffs \in \mathbb{N}$.

2. $l \in \mathbb{N}_0$ (Section 5.3.3).

3. Each $v \in In$ is a v-zone.

4. The master-slice of each v-zone must be unique in the entire master-stack.

5. $Vplace$ is a sequence of the $v \in In$ describing their bottom-top topological placement (Definition 84).

■

Definition 60 A *master-stack tree* is a network skeleton tree with a master-stack N_{mS} at the root, and only v-zone trees as subtrees via $\widetilde{hier}(In\langle N_{mS}\rangle)$.

All v-segments s_i within the entire master-stack tree have $l\langle s_i\rangle = l\langle N_{mS}\rangle$. ■

Iterating a stack proceeds analogously to Definition 46 for master-slices. For sake of brevity, we will abstain from explicitly defining the horizontal iteration functions for stacks, with one important exception: The iteration of a master-stack alters the bit-significance in the v-segments occuring as leaves. This changes (by Definition 35) the variables of all primary ports.

Definition 61 A *stack* N_S is a network skeleton that is an iteration of the master-stack $mr\langle master(N_S)\rangle$.

In stacks, the iteration number component $itno\langle master\rangle$ is referred to as *horizontal iteration number*, abbreviated $hitno\langle N_S\rangle \in \mathbb{N}_0$. A stack has the same length as its master-stack.

Stacks are referred to using their master-stack and iteration number, notated as $master_{hitno}$. ■

Definition 62 A *stack tree* is a network skeleton tree with a stack N_S as root, and only v-zone trees as subtrees via $\widetilde{hier}(In\langle N_{ms}\rangle)$.

All bit-significances in the entire network skeleton tree U' rooted at N_S are offset by $hoffs\langle mr\langle master\rangle\rangle \cdot hitno$ over their corresponding significances in the regular hierarchy tree rooted at $mr\langle master\rangle$.

Note that, in contrast to master-slices, we don't support descending bit-significances for master-stack iteration, $hoffs\langle N_{mS}\rangle$ is always positive.

Example 63 Consider a terminal $t = (\mathsf{M}/\mathsf{u}, \mathbf{a}_{0,1})$ in a master-slice M whose instances (=v-segments) occur in a v-zone z in a master-stack mS, with $hoffs\langle \mathsf{mS}\rangle = 4$.

In the third vertical iteration s_3 of M in z in mS, the corresponding terminal becomes $(\mathsf{s}_3/\mathsf{w}, \mathbf{a}_3)$ (significance computation during master-slice iteration, Definition 46).

In horizontal iteration S_2 of mS (note: this is a master-stack iteration, not a master-slice iteration!), the v-segment s'_3 in v-zone z' in S_2 is assumed to be corresponding to s_3 in z in mS.

The variable significance for the underlying v-segments of S_2 have been altered as per Definition 62: After the stack iteration, the significance of our sample terminal has been offset by $hitno\langle \mathsf{S}_2\rangle \cdot hoffs\langle \mathsf{mS}\rangle = 2 \cdot 4 = 8$, leading to $(\mathsf{s}'_3/\mathsf{u}, \mathbf{a}_{11})$.

In another similarity between a v-zone (containing iterations of a master-slice) and master-stack (containing v-zones), certain constraints have to be imposed on the v-zones in a master-stack to guarantee logical completeness and abutment between individual v-zones and stacks. Clashes of duplicate variables (same symbol and same significance) also have to be avoided. For brevity, we will define these constraints in prose. However, they could as well be formulated more formally similar to Definition 52.

Definition 64 A *valid master-stack* N_{mS} fulfills the criteria for logical abutment and completeness. To this end, each symbol occuring at the top level of N_{mS} must occur with all significances between and including its least- and most-significant occurrences. Furthermore, all variables on primary ports must be unique before and after iteration.

G_{mS} is the class of all valid master-stacks. ■

5.2.9 H-Zones: Multi-Stack Structures

Analogously to the step from v-segments to v-zones, we now ascend to the next higher level of hierarchy by introducing circuits composed of multiple iterations of the same master-stack.

We will define the *h-zone*, a structure composed of one or more iterations of a single master-stack[8]. In context of a h-zone, a master-stack iteration will be referred to as a *h-segment*.

Definition 65 A h-zone $N_h = (N_S, \mathit{hmaster}, \mathit{repl})$, is a network skeleton N_S extended with the components $\mathit{hmaster} \in G_{mS}$ and $\mathit{repl} \in \mathbb{N}$.

1. $|In| = \mathit{repl}$.
2. Internal nodes are the h-segments with horizontal iteration numbers $0, \ldots, \mathit{repl} - 1$ of the master-stack *hmaster*.

■

Definition 66 A *h-zone tree* is a network skeleton tree that has a h-zone N_h at the root, with each internal node of N_h being the root of an underlying (via *hier*) stack tree. ■

As before, we enforce our constraints for logical abutment, completeness, and unique variables.

Definition 67 To guarantee logical abutment and completeness within a h-zone N_h, each symbol x occurring at the top-level of N_h must occur with all significances between and including its least- and most-significant occurrences. Furthermore, all variables on primary ports must be unique. ■

Example 68 Figure 5.7 shows the hierarchy (*hier*) and regularity (*master*) relations from master-slice up to h-zone using the example of a left-shifter. As before, we ommitted terminals and node names for clarity.

The instance hierarchy at the left side has the h-zone H' at the top-level. It consists of the $\mathit{repl}\langle \mathsf{H}' \rangle = 2$ iterations $\mathsf{H}'/\mathsf{S}_0, \mathsf{H}'/\mathsf{S}_1$ of the master-stack $\mathit{hmaster}\langle \mathsf{H}' \rangle = \mathsf{S}$. The structure underlying H'/S_1 is the stack $\mathit{hier}(\mathsf{H}'/\mathsf{S}_1) = \mathit{hmaster}\langle \mathsf{H}' \rangle_1 = \mathsf{S}_1$.

This stack inherits the connectivity structure and behavior from its master-stack $\mathit{master}(\mathsf{S}_1) = \mathsf{S}$. Since S consists only of the single v-zone S/Z, so will the stack S_1: it consists only of S_1/Z.

While connectivity structure and behavior of the of the v-zone in the master and instance hierarchy are also identical, they are different but isomorphic graphs: $\mathit{hier}(\mathsf{S}_1/\mathsf{Z}) = \mathsf{Z}'$, and $\mathit{hier}(\mathsf{S}/\mathsf{Z}) = \mathsf{Z}$. This represents the constraint that *hier*-relations may not cross between master and instance regular hierarchy trees.

With $\mathit{vmaster}\langle \mathsf{Z}' \rangle = \mathit{vmaster}\langle \mathsf{Z} \rangle = \mathsf{s}$, and $\mathit{repl}\langle \mathsf{Z}' \rangle = \mathit{repl}\langle \mathsf{Z} \rangle = 4$, we describe the internal structure of the v-zones as containing v-segments $\mathsf{Z}'/\mathsf{s}_0, \mathsf{Z}'/\mathsf{s}_1, \mathsf{Z}'/\mathsf{s}_2, \mathsf{Z}'/\mathsf{s}_3$ in the instance, and $\mathsf{Z}/\mathsf{s}_0, \mathsf{Z}/\mathsf{s}_1, \mathsf{Z}/\mathsf{s}_2, \mathsf{Z}/\mathsf{s}_3$ in the master hierarchy.

---

[8] Note that this hierarchy ordering is different from the one used in the module-generators

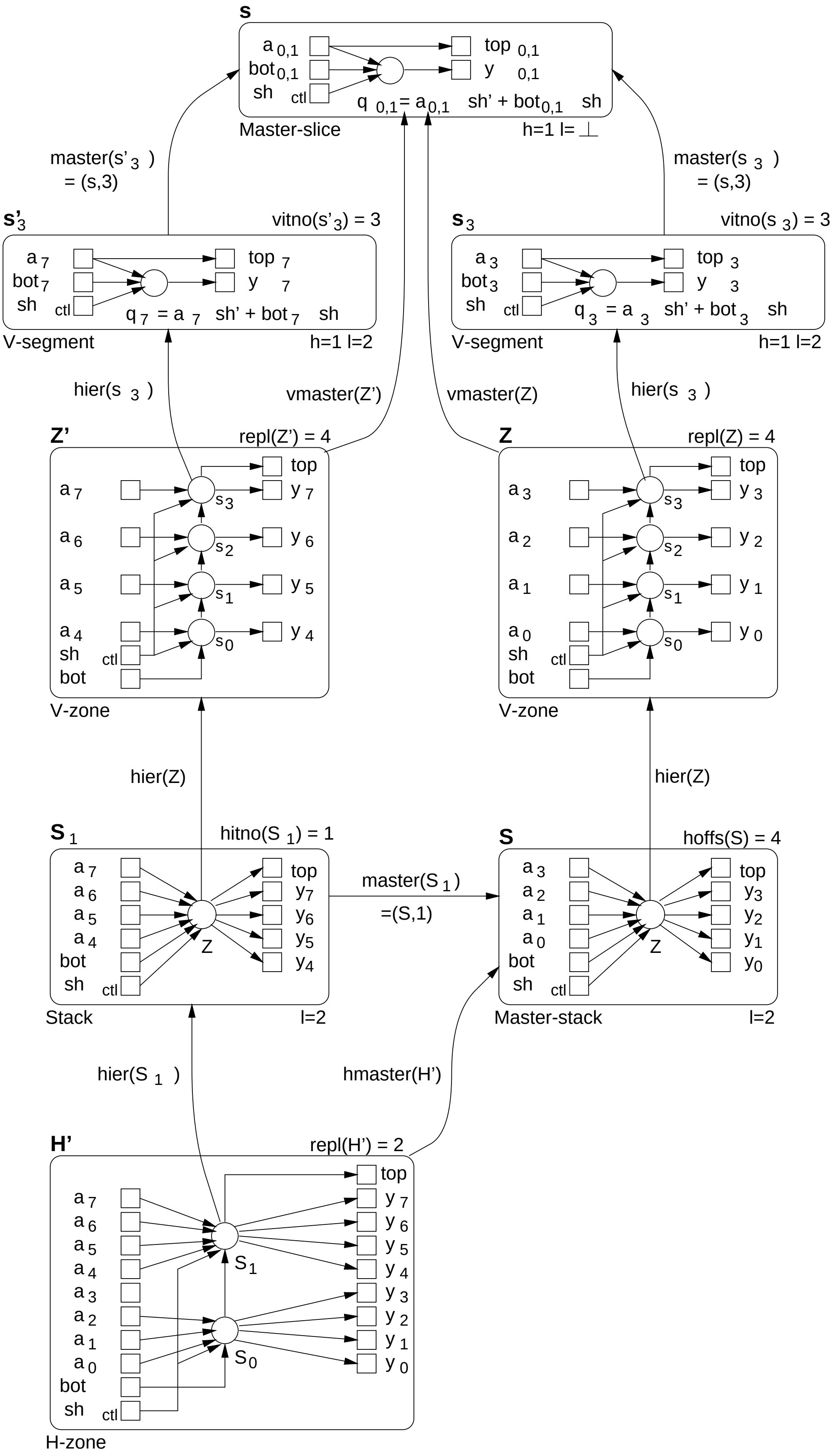


Figure 5.7: From v-segment to h-zone: hierarchy and regularity relations

Each of these v-segments has the same master-slice, e.g.

$$master(\mathsf{Z}'/\mathsf{s}_3) = master(\mathsf{Z}/\mathsf{s}_3) = vmaster\langle\mathsf{Z}\rangle = vmaster\langle\mathsf{Z}'\rangle = \mathsf{s}.$$

Since s is a gate network, it can actually define behavior in terms of its local functions. All other abstraction levels can express behavior only by the upwardly-propagated v-segment functions and their own interconnection pattern.

After describing the hierarchical and regular structure composition, we will now examine the computation of bit-significances, using the primary input port with the symbolic significance $\mathbf{a}_{0,1}$ as an example. When considering it inside of s'_3, it should become the variable $\mathbf{a}_3$ (Definition 46). However, since s'_3 (as shown in Figure 5.7) is used in the larger context of S_1, the bit-significances have been altered during stack-iteration as described in Definition 61: Since $hoffs\langle mr\langle master(\mathsf{S}_1)\rangle\rangle = hoffs\langle\mathsf{S}\rangle = 4$, each stack-iteration will offset its underlying significances by 4. Thus, with s'_3 occuring in the first iteration, the symbolic significance $\mathbf{a}_{0,1}$ will actually become the variable $\mathbf{a}_{3+4} = \mathbf{a}_7$. The hierarchical nature of this significance-alteration is also apparent in h-zone Z', which also has its significances offset by 4 over its corresponding h-zone Z in the master hierarchy.

After this demonstration, we might question the use of separate v-zones and h-zones. After all, we could just have iterated the master-slice s eight times for the same result (after adding the required nets for sh, top, and bot between the instances). However, the origins of the names v-zone ("vertical zone") and h-zone ("horizontal zone"), v-segment ("vertical segment") and h-segment ("horizontal segment"), as well as "vertical" and "horizontal significance offsets" already hint at a dependency between these entities and geometric layout, which will be revealed in Section 5.3.

5.2.10 Modules: Multi-h-zone Structures

Similar to the relation between v-zones and a stack, we now describe circuits composed of iterations of *different* master-stacks. As before, we will allow the instantiation of these structures to form even more complex circuits.

To this end we will define *master-modules* which contain one or more h-zones. An instance of a master-module is called a *module* and has the same connectivity structure and behavior as its master-module.

Definition 69 A *master-module* $N_{mM} = (N_{\mathcal{S}}, compactable, Hplace)$ is a network skeleton $N_{\mathcal{S}}$ extended with the components $compactable \in \mathbb{B}$ and the horizontal topological placement $Hplace$.

1. Each internal node v is a h-zone.
2. For all $v \in In$, each $hmaster\langle v\rangle$ may occur only once.
3. *compactable* is *TRUE* iff N_{mM} represents a soft-macro (Explanation 5).
4. *Hplace* is a sequence of all $v \in In$ describing their left-right topological placement (Definition 84).

■

Definition 70 A *master-module tree* is a network skeleton tree with a master-module N_{mM} at the root, and only h-zone trees as subtrees via $\widetilde{hier}(In\langle N_{mM}\rangle)$. ■

Note that master-modules are only instantiated, not iterated: All bit-significances have already been computed and do not change between instances.

Definition 71 A *module* N_M is a network skeleton. It has the same connectivity structure and behavior as its master-module *master*. ■

Definition 72 A *module tree* is a network skeleton tree with a module N_M at the root, and only h-zone trees as subtrees via $\widetilde{hier}(In\langle N_{mM}\rangle)$.

For purposes of regularity, the components $compactable\langle mr\langle master(N_M)\rangle\rangle$ and *Hplace* are not propagated from master-module into modules, and disregarded when establishing the isomorphism between master and instance. ■

Observe the difference between the master-module/module relation and, e.g., the master-slice/slice relation: While the slices of the same master-slice are related by their iteration numbers, no such link exists between modules of the same master-module. Modules are only related by their common master-module (Clarification 20 and Definition 26), no other kind of dependency exists.

As usual, we extend the logical abutment and completeness constraints to encompass this level of the hierarchy.

Definition 73 A *valid master-module* N_{mM} fulfills the criteria for logical abutment and completeness. To this end, each symbol x occurring at the top-level of N_{mM} must occur with all significances between and including its least- and most-significant occurrences. Furthermore, variables on primary ports must be unique for all h-zones in N_{mM}.

G_{mM} is the class of all valid master-modules. ■

5.2.11 Datapaths: Multi-Module Structures

We have now reached the summit of the structural hierarchy, the datapath itself.

Definition 74 A *datapath* $N_D = (N_S, Hplace)$ is a network skeleton N_S extended with the sequence *Hplace*.

1. Each internal node is a module.

2. Each master-module may occur more than once in the modules of N_D.

3. *Hplace* is the sequence of all $v \in In$ and describes their left-right topological placement (Definition 84).

■

Definition 75 A *datapath tree* is a network skeleton tree with a datapath N_D at the root, and only module trees as subtrees via $\widetilde{hier}(In\langle N_D\rangle)$. ■

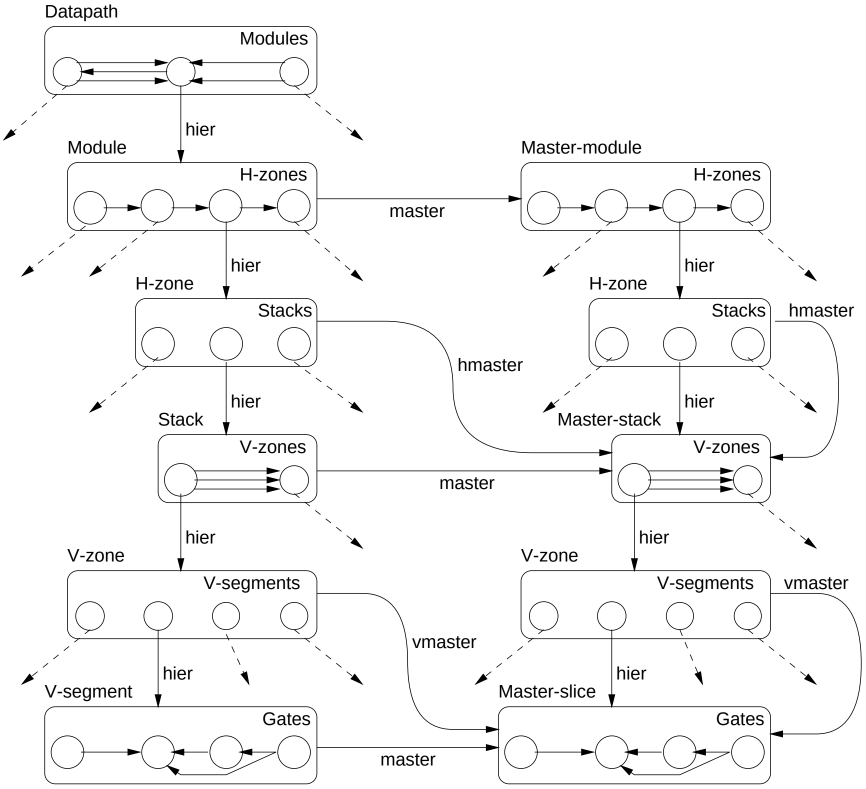


Figure 5.8: Regularity and hierarchy in a datapath

Example 76 Figure 5.8 shows the hierarchy and regularity relations for an entire datapath. For clarity, only internal nodes are shown, and all names have been omitted. Note the parallel hierarchies of masters (right) and instances (left). Furthermore, observe the alternating pattern of replicating (v-zone, h-zone) and instantiable structures (master-slice, master-stack, master-module) in the master hierarchy. Regularity in SDI is thus expressed by the combination of master/instance relations and replication (iteration) counts.

5.2.12 Flattening the Hierarchy

For reasons of brevity, we will abstain from formally defining operators for flattening, relying on an informal description of the process instead. The interested reader will easily be able to deduce the details from the example given below.

Clarification 77 *Flattening* is the process of transforming a structure at level n of the hierarchy into the externally equivalent structure at the next lower level $n - 1$. *Externally equivalent* means that both structures have identical primary ports, and show identical behavior as far as can be observed at the primary ports.

E.g., a v-zone (nodes are v-segments) can be flattened into a gate network (nodes are gates) by replacing each internal node v of the v-zone by its underlying gate network $hier(v)$. Note that the v-zone primary ports remain untouched (as per Clarification 77). Nodes in the gate network will have to be relabeled (e.g., by adding the iteration number of their original v-segment) to avoid clashes, and their associated functions changed to refer to the relabeled nodes. TTNs at the v-zone level are turned into equivalent TTNs at the v-segment level, and the v-segments' primary ports are replaced by TTNs directly between their connected nodes.

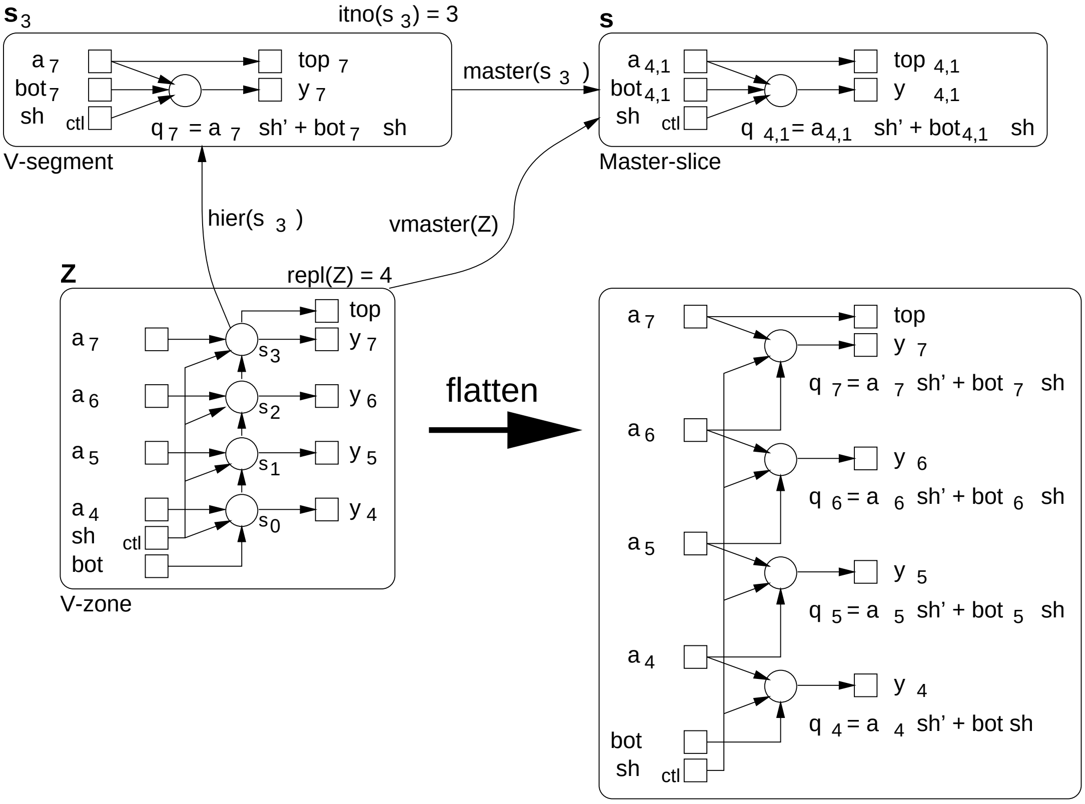


Figure 5.9: Flattening a v-zone into v-segments

Example 78 Figure 5.9 shows an example of such a flattening operation. Note how multiple occurrences of the sh primary control input in the separate v-segments have been replaced by a single port, and appropriate nets added in their stead. Conversely, the bot and top primary ports (at the v-segment level) have been completely replaced by TTNs. An example for a changed local function definition is f_{q_4}, where the variable $\mathbf{bot}_4$ has been replaced by **bot**. All of the v-zone's primary ports have remained intact. Thus, the resulting network shows the same externally visible behavior as the original v-zone.

The primary flattening operation in SDI flattens datapaths into slices. It can be implemented in a manner that allows us to keep track of the position of each slice in the original hierarchy.

Definition 79 A datapath tree rooted at N_D flattened down to the slice level is the *flattened network skeleton* $N_f = (N_{\mathcal{S}}, module, hzone, hitno, vzone, vitno)$, which is a network skeleton $N_{\mathcal{S}}$ extended with the components *module*, *hzone*, *hitno*, *vzone*, and *vitno*.

1. In is the union of all v-segments of all v-zones of all stacks of all h-zones of all modules in the module trees with roots $In\langle N_D\rangle$.
2. The top-level primary ports remain untouched during flattening, thus $Pi\langle N_f\rangle = Pi\langle N_D\rangle$ and $Po\langle N_f\rangle = Po\langle N_D\rangle$.
3. $Ui\langle N_f\rangle$ ($Uo\langle N_f\rangle$) is the union of the $Ui\langle N_v\rangle$ ($Uo\langle N_v\rangle$) of the v-zones N_v in N_D.
4. $E, Ttn, ptype, hier, sym, sig$ are computed during flattening (as sketched in Figure 5.9).
5. Each internal node is a slice (former v-segment) with an underlying (via *hier*) gate network describing its structure and behavior.
6. $module : v \in In \mapsto N_m$ associates a slice with its original module N_m.
7. $hzone : v \in In \mapsto N_h$ associates a slice with its original h-zone N_h within its module N_m.
8. $hitno : v \in In \mapsto j \in \mathbb{N}_0$ associates a slice with its original horizontal (stack) iteration number j within its h-zoneN_h.
9. $vzone : v \in In \mapsto N_v$ associates a slice with its original v-zone N_v within its stack N_{h_j}.
10. $vitno : v \in In \mapsto k \in \mathbb{N}_0$ associates a slice with its original vertical iteration number k within its v-zone N_{v_k}.

■

Numerous examples for the use of flattened network skeletons will be presented in Chapter 6.

5.3 Topological and Geometrical Layout

In the previous sections, we have dealt only with structural and behavioral information. However, SDI is a system which also includes physical design automation functions. It thus needs a way to represent placement data.

Clarification 80 A *layout* is an arrangement of units on the plane $\mathbb{Z}^2$. ■

The $\mathbb{Z}^2$ placement grid is not completely arbitrary, it models the array of underlying FPGA cells.

Clarification 81 *Height* is a vertical distance, *length* is a horizontal distance[9]. The fundamental unit of distance in layouts is the cell (Explanation 7), it is defined to have height 1 and length 1.

5.3.1 Representing Geometrical Layout

Geometrical layouts can be described by the loc, h, l components of network skeletons (Definition 34)

The height $h(v)$ of a node v is assumed to extend upwards from its location, while its length $l(v)$ extends to the left. Together, $h(v)$ and $l(v)$ delimit the bounding box of v. Some details are notable in this definition: Units may be unplaced ($\perp$) in one or both coordinates. Also, units with height and/or length of zero are legal. This feature will be used to handle primary ports, which do not consume chip area[10], but only routing resources.

Definition 82 A *geometrical layout tree* is a network skeleton tree in which all nodes v of all node graphs have $loc(v) \in \mathbb{Z}^2$, and $l(v), h(v) \in \mathbb{N}_0$. ■

All locations at hierarchy level n are specified relative to the location l_c of the containing unit at level $n-1$. Thus, l_c is projected at the origin $(0,0)$ in level n. Hierarchy level 0 is assumed to have a location of $(0,0)$.

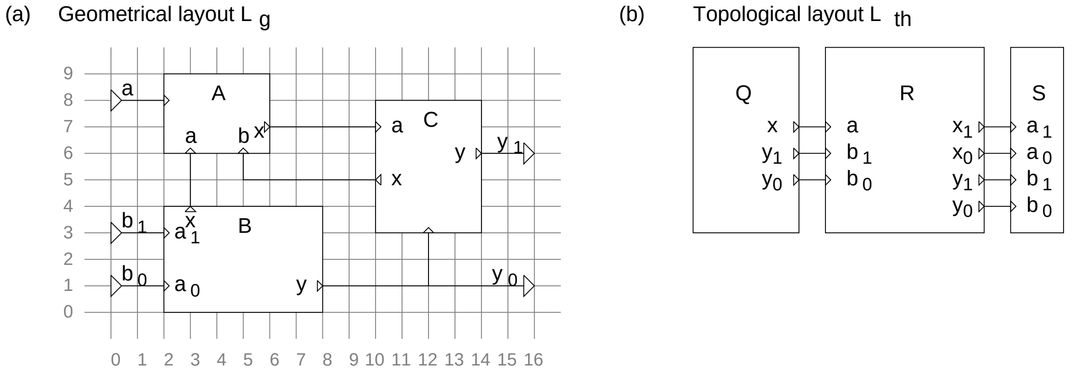


Figure 5.10: Geometrical (a) and topological layout (b) with representations

---

[9] Remember that *width* is only a logical quantity, Explanation 2.

[10] The exception are chip-level pads, which will not be considered here.

Example 83 Figure 5.10.a shows a geometrical layout L_g of a network [11]. It represents the placement and has, e.g., $loc\langle \mathsf{a} \rangle = (0, 8)$, $h\langle \mathsf{a} \rangle = l\langle \mathsf{a} \rangle = 0$, $loc\langle \mathsf{C} \rangle = (10, 3)$, $h\langle \mathsf{C} \rangle = 5$, $l\langle \mathsf{C} \rangle = 4$. Note how we modelled the point nature of primary ports. The location of unit terminals is determined by the location of the corresponding primary port at the next lower hierarchy level. E.g., for output variable x of C, the corresponding primary port C/x would have $loc\langle \mathsf{C/x} \rangle = (0, 2)$. Since the placement origin in C is $loc\langle \mathsf{C} \rangle$ in L, the coordinates of C/x in L become $loc\langle \mathsf{C} \rangle + loc\langle \mathsf{C/x} \rangle = (10, 3) + (0, 2) = (10, 5)$.

5.3.2 Representing Topological Layout

Another way to describe a layout is to specify the relative positions of units to each other. The complexity of representable layouts depends on the relations available. SDI requires only 1-D (linear) topological layouts, we can thus use the order of elements in a sequence to describe the linear arrangement of units.

Definition 84 A *horizontal topological layout* $L_{th} = (N_{\mathcal{S}}, Hplace)$ is a network skeleton $N_{\mathcal{S}}$ extended with the sequence $Hplace$. The elements of $Hplace$ are the units of $N_{\mathcal{S}}$. The order in $Hplace$ describes their horizontally abutting left-to-right placement.

A *vertical topological layout* $L_{tv} = (N_{\mathcal{S}}, Vplace)$ analogously defines a vertically abutting bottom-to-top placement. ■

Definition 85 A *topological layout tree* is a regular hierarchy tree in which all node graphs are topological layouts. ■

Example 86 The horizontal topological layout of units in Figure 5.10.b is described as a left-to-right arrangement by the sequence $Hplace\langle L_{th} \rangle = (Q, R, S)$.

5.3.3 Describing Regular Bit-Sliced Layouts

While Section 5.3.1 allows the representation of arbitrary geometric layouts, the regular bit-sliced layout sketched in Section 2.3.1 can be composed in a bottom-up manner with only a few general parameters using a combination of topological and geometrical representations.

Clarification 87 A *bit-sliced module layout* is described by the heights of its master-slices and the lengths of its master-stacks.

Note that the master-instance relation propagates the height of the master to all of its instances. Furthermore, all v-segments in a stack have the same length (propagated from master-stack to stack to v-zone to v-segment), allowing directly abutting placement of stacks. We do not constrain the stack

---

[11] Inter-unit connections are just shown for completeness, their locations are not actually defined.

height, it is allowed to vary to accommodate folded modules (like a 12-bit module implemented as an 8-bit and a 4-bit stack) or different datapaths widths. However, to minimize wasted chip area, we will aim at a homogeneous datapath height after processing (Chapter 7).

The previously defined quantities height and length will now be used: A master-slice contributes solely a geometrical quantity (height) to the entire placement, while a master-stack contains geometrical (length) and topological (v-zone sequence $\mathit{Vplace}\langle N_{mS}\rangle$) information. Both master-modules and datapaths specify only topological constraints (h-zone sequence $\mathit{Hplace}\langle N_{mM}\rangle$ and module $\mathit{Hplace}\langle N_D\rangle$ sequence, respectively).

The extended skeletons themselves do not describe placement as defined in Section 5.3.1. However, a geometric layout tree can be derived by interpreting their data appropriately in a bottom-up manner. As before, we will refrain from formally specifying the transformation, relying on a verbal description of the process and a comprehensive example instead.

V-zone level A v-segment s is located at horizontal coordinate 0 and vertical coordinate $\mathit{vitno}\langle s\rangle \cdot h\langle \mathit{vmaster}\langle N_v\rangle\rangle$ in its enclosing v-zone N_v.

Stack level All v-zones in a stack N_S and all of their v-segments have a length of $l\langle \mathit{master}(N_S)\rangle$. The order in $\mathit{Vplace}\langle N_S\rangle$ describes an upward sequence (increasing vertical coordinates) of abutting v-zones.

H-zone level A h-segment S is located at vertical coordinate 0 and horizontal coordinate $\mathit{hitno}\langle S\rangle \cdot l\langle \mathit{hmaster}\langle N_h\rangle\rangle)$ in its enclosing h-zone N_h.

Module level The order in $\mathit{Hplace}\langle N_{mM}\rangle$ describes a left-to-right sequence (increasing horizontal coordinates) of abutting h-zones.

Datapath level The order in $\mathit{Hplace}\langle N_D\rangle$ describes a left-to-right sequence (increasing horizontal coordinates) of abutting modules.

Algorithm 1: Deriving geometrical layout

Algorithm 1 finally spells out the origin of the names v-zone ("vertical zone") and h-zone ("horizontal zone") etc., as already pointed out at the end of Section 5.2.9: v-segments and v-zones describe a vertical, upwardly growing layout, while stacks and h-zones are responsible for horizontal layouts, growing from left-to-right.

Definition 88 The geometrical layout tree computed by applying Algorithm 1 is called a *bit-sliced geometrical layout tree*. The resulting layout is *valid* in that no v-segments overlap. ■

The layout derived in this manner only specifies placement down to the v-segment level, the placement of individual gates inside a v-segment remains unspecified. V-segments form the atomic components of our bit-sliced structures, they are not bit-sliced themselves. Thus, their placement is irregular and cannot be computed by simple rules like Algorithm 1. It will be determined by the methods shown later in Chapter 7.

Example 89 Figure 5.11 shows the hierarchical composition of a datapath layout. The height of each v-segment is determined at the master-slice level. V-segment locations inside a v-zone ascend with the iteration number. E.g., v-segment 0 of v-zone vA has the location $(0, 0)$, v-segment 1 is placed at $(0, 2)$. The length of v-zones and their v-segments is determined at the master-stack level. E.g., master-stack SA fixes a length of 4 for v-zones vA, vB, and their v-segments $\mathsf{msA}_0, \mathsf{msA}_1, \mathsf{msB}_0$. The upwards topological arrangement of v-zones in SA is defined by the sequence $\mathit{Vplace}\langle\mathsf{SA}\rangle = (\mathsf{vA}, \mathsf{vB})$, leading to $\mathit{loc}\langle\mathsf{vA}\rangle = (0, 0)$, and $\mathit{loc}\langle\mathsf{vB}\rangle = (0, 4)$. The layout at the h-zone level yet again depends on the iteration number, with each stack displaced by its length to the right. E.g., $\mathit{loc}\langle\mathsf{SA}_1\rangle = (4, 0)$. In master-modules, the layout is defined by the left-to-right sequence of h-zones $\mathit{Hplace}\langle\mathsf{M}\rangle = (\mathsf{hA}, \mathsf{hB})$, yielding, e.g., $\mathit{loc}\langle\mathsf{hB}\rangle = (8, 0)$. At the datapath level, another left-to-right sequence determines the layout $\mathit{Hplace}\langle\mathsf{D}\rangle = (\mathsf{mA}, \mathsf{mB})$.

Using this hierarchical method of layout composition, and the fact that the location of each element is the origin of the next lower hierarchy level, we can now determine the geometrical position of each element by applying Algorithm 1. E.g., to determine the location of v-segment 1 of v-zone vA in h-segment 1 of h-zone hA within module mB at the datapath level, we proceed in a top-down manner: Using $\mathit{Hplace}\langle\mathsf{D}\rangle$, we find that $\mathit{loc}\langle\mathsf{mB}\rangle = (14, 0)$. Descending to the master-module level, we compute $\mathit{loc}\langle\mathsf{hA}\rangle = (0, 0)$. In the h-zone hA, h-segment 1 has $\mathit{loc}\langle\mathsf{hA}_1\rangle = (4, 0)$. The corresponding master-stack SA contains two v-zones, with $\mathit{loc}\langle\mathsf{vA}\rangle = (0, 0)$. Inside this v-zone v-segment 1 has $\mathit{loc}\langle\mathsf{msC}_4\rangle = (0, 2)$. The absolute location is now determined by ascending through the hierarchy, and adding the relative locations $(0, 2) + (0, 0) + (4, 0) + (0, 0) + (14, 0) = (18, 2)$. As demonstrated by Figure 5.11, that is indeed the location of the v-segment. By performing these computations for each element, a complete geometrical layout tree can be assembled.

At this point, we have built a conceptual framework for the representation of structural, behavioral, geometrical, and topological information. The description of the SDI algorithms in the next chapters will be based on these fundamentals.

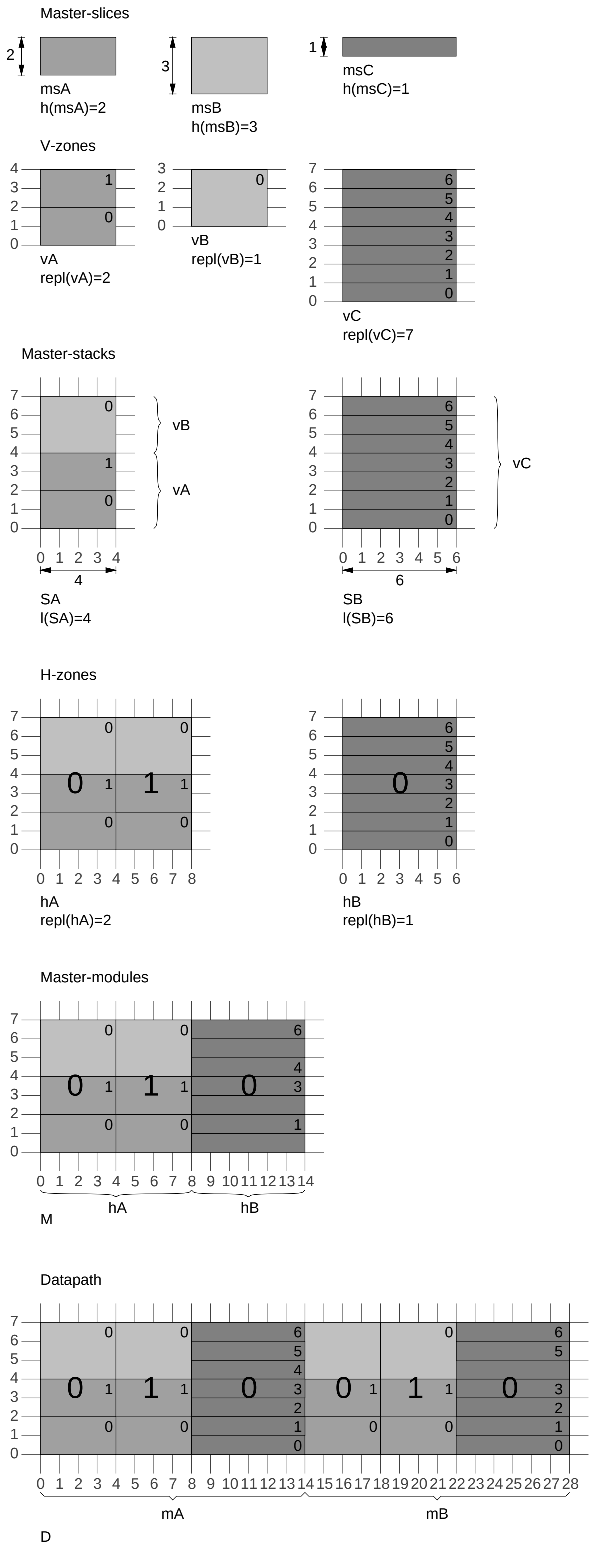


Figure 5.11: Hierarchical bottom-up layout of a datapath

6 Regular Compaction

After the floorplanner has created a linear placement of module instances, and selected an actual implementation for each instance, the coarse-grain architecture of the XC4000 FPGAs requires an additional optimization phase to improve layout density and circuit performance (Section 2.6).

The regular compaction phase searches the datapath floorplan for sets of optimizable module instances, builds a map of the topological structure across all instances in each set, and then tries to find regularity in the map. The result is a new stack (Explanation 6) for each module set. Any regularity thus determined is then exploited to reduce computation time, and preserved in the optimized layout. Area and delay reductions are achieved by locally applying logic synthesis and technology mapping algorithms to the newly created master-slices.

Note that the approach described here is dramatically different from that introduced in the section “Zone analysis and merging” of [Koch96b]. That algorithm worked by running a horizontal scanline across the geometrical layout of the subdatapath under compaction. While it completely preserved the vertical topological placement of slices, it did not consider connectivity issues. The current procedure uses an improved graph-based approach that can recognize more complicated forms of regularity (e.g., regular bit-permutations or shift registers) [1] . However, since it loses the vertical topological placement, we had to extend the microplacement step (Section 7.1) with a preprocessing phase that quickly redetermines vertical arrangements from the still-intact interconnection patterns.

Since our target structure is a stack, we are only interested in extracting vertically arranged structures from the datapath. Thus, h-segments will play only a minor role, and are flattened into v-segments early in the process.

6.1 Finding Optimizable Areas

As outlined in Section 2.6.3, the datapath is not processed indiscriminately, only soft-macros are affected. The optimization occurs in the scope of the soft subdatapath. Since the order of subdatapaths (soft and hard) reflects the module ordering in the original floorplan, the planned topology is preserved.

Algorithm 2 is used to collect the largest contiguous soft-macro sequences

---

[1] Should you get lost in the complexity of this multi-step process, Section 6.6 gives a top-level summary explaining the relations between different circuit structures and views.

Algorithm 2: Find largest contiguous soft subdatapaths of datapath D
Require: D is a datapath
Ensure: C_D is the sequence of longest contiguous soft-macro sequences in D

```
$C_D \leftarrow \emptyset$
$d \leftarrow \emptyset$ {work subdatapath}
for all $m \in Hplace(D)$ do {run left-to-right over all modules}
  if $compactable(m)$ then {is m a soft macro?}
    append element $m$ to $d$
  else if $d \neq \emptyset$ then {m is a hard-macro, any soft-macros found yet?}
    append sequence $d$ to $C_D$ {yes, close the working subdatapath}
    $d \leftarrow \emptyset$
  end if
end for
if $d \neq \emptyset$ then {close a trailing subdatapath}
  append sequence $d$ to $C_D$
end if
```

in a datapath D. It scans across all modules in D from left-to-right and remembers soft-macros in the temporary sequence d. When it hits a hard-macro (or the right edge of the datapath), it closes the temporary sequence d and enters the newly found soft subdatapath into the output sequence C_D.

Example 90 Consider the example using Figure 2.22.a as D, giving

$$In(D) = (M_1, H_1, M_2, M_3, M_4, M_5, H_2, M_6, M_7),$$

with $compactable(m) = TRUE$ for $m \in \{H_1, H_2\}$, $FALSE$ otherwise. Applying the algorithm to D yields $((M_1), (M_2, M_3, M_4, M_5), (M_6, M_7))$ as the sequence of largest contiguous soft-macro sequences of D. The ordering of the soft-macro sequences, and of the modules within, still reflects the original floorplan.

Each of these module sequences in C_D can be turned into a datapath graph by applying Algorithm 3.

The optimization process shown in Figure 2.20 is then performed separately for each soft subdatapath. Due to their data independence, this optimization may also be executed in parallel for each subdatapath (not taken advantage of in the current SDI implementation).

Example 91 Figure 6.1 shows an example for the application of Algorithm 3 to build a subdatapath from the largest contiguous soft-macro sequence (in the example, (B, C)). Note how the variables of the newly created primary ports are qualified (using ".", to avoid confusion with the "/" separator in hierarchical names) with their original node to prevent conflicts. E.g., without the qualification, **B**.**b** and **C**.**b** would become indistinguishable (just **b**). This would violate the function requirement on *port* at the next higher hierarchy level.

Algorithm 3: Determine datapath D_Q for the the module sequence $Q \in C_D$
Require: D is a datapath , $Q \in C_D$
Ensure: D_Q is a datapath with $In(D_Q) = Q$

```
{the internal nodes are just the modules in the input sequence}
$In\langle D_Q\rangle \leftarrow Q$
{turn all TTNs with a sink outside of $D_Q$ into POs}
for all $((u, p), (v, q)) \in Ttn\langle D\rangle \wedge v \notin In\langle D_Q\rangle$ do
  create new primary output $w$
  $Po\langle D_Q\rangle \leftarrow Po\langle D_Q\rangle \cup \{w\}$
  {propagate variable and ptype from fanin terminal into new primary port}
  $Ui\langle D_Q\rangle(w) \leftarrow \{(u.sym(p), sig(p))\}$
  $ptype\langle D_Q\rangle(w) \leftarrow ptype(port\langle D\rangle(v)(p))$
  {add TTN connecting the new primary and old unit ports}
  $Ttn\langle D_Q\rangle \leftarrow Ttn\langle D_Q\rangle \cup \{((u, p), (w, (u.sym(p), sig(p))))\}$
end for
{turn all TTNs with a source outside of $D_Q$ into PIs}
for all $((u, p), (v, q)) \in Ttn\langle D\rangle \wedge u \notin In\langle D_Q\rangle$ do
  create new primary input $w$
  $Pi\langle D_Q\rangle \leftarrow Pi\langle D_Q\rangle \cup \{w\}$
  {propagate variable and ptype from fanout terminal into new primary port}
  $Uo\langle w\rangle \leftarrow \{(v.sym(q), sig(q))\}$
  $ptype\langle D_Q\rangle(w) \leftarrow ptype(port\langle D\rangle(v)(q))$
  {add TTN connecting the new primary and old unit ports}
  $Ttn\langle D_Q\rangle \leftarrow Ttn\langle D_Q\rangle \cup \{((w, (v.sym(q), sig(q))), (v, q))\}$
end for
{build union to get all nodes}
$V\langle D_Q\rangle \leftarrow In\langle D_Q\rangle \cup Po\langle D_Q\rangle \cup Pi\langle D_Q\rangle$
{add TTNs internal to the subdatapath}
$Ttn\langle D_Q\rangle \leftarrow Ttn\langle D_Q\rangle \cup \{((u, p), (v, q)) \in Ttn\langle D\rangle \mid u, v \in In\langle D_Q\rangle\}$
{determine edges from TTNs as per Definition 34}
$E\langle D_Q\rangle \leftarrow \{(u, v) \mid \exists\, p, q : ((u, p), (v, q)) \in Ttn\langle D_Q\rangle\}$
{Ui, Uo, hier and port are restricted to the subdatapath}
$Ui\langle D_Q\rangle \leftarrow (Ui\langle D\rangle$ restricted to $V\langle D_Q\rangle)$
$Uo\langle D_Q\rangle \leftarrow (Uo\langle D\rangle$ restricted to $V\langle D_Q\rangle)$
$hier\langle D_Q\rangle \leftarrow (hier\langle D\rangle$ restricted to $V\langle D_Q\rangle)$
$port\langle D_Q\rangle \leftarrow (port\langle D\rangle$ restricted to $V\langle D_Q\rangle)$
{we don't support instances of entire datapaths}
$master\langle D_Q\rangle = (\perp, \perp)$
```

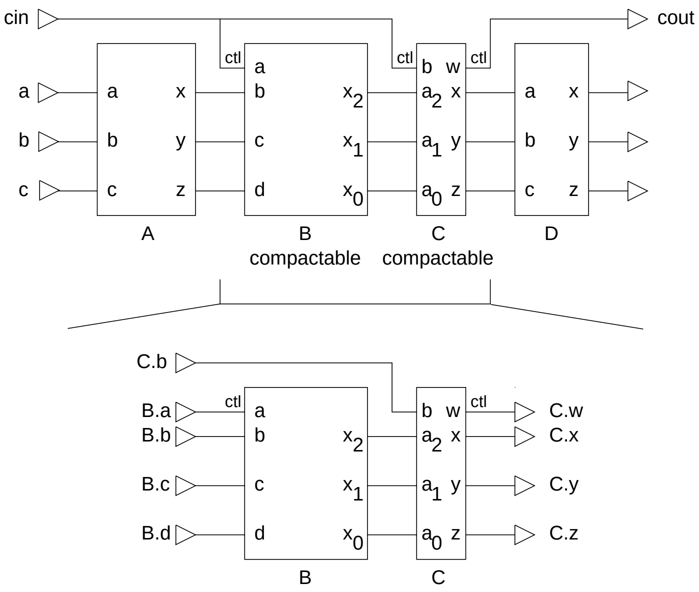


Figure 6.1: Applying Algorithm 3

6.2 Flattening the Subdatapath

After extracting the contiguous soft-macro sequences, and turning them into subdatapaths, we now flatten the subdatapaths down to the v-segment level for structure extraction and regularity analysis.

As outlined in Section 5.2.12, flattening is implemented in a manner that preserves information on the original location of each v-segment in the former hierarchy.

Example 92 Figure 6.2.a shows the soft subdatapath $\mathcal{D}$ consisting of the three modules A, B and C. Flattening turns it into the flattened network skeleton $\mathcal{D}_f$ shown in Figure 6.2.b. Note how the ports of intermediate hierarchy levels have vanished: Only the primary ports of the entire subdatapath, and the terminals of the v-segments remain.

The former hierarchy information has been preserved during flattening. E.g., assume the top v-segment in the second column of Figure 6.2.b is called g. In the example, $module(\mathsf{g}) = \mathsf{A}$, $hzone(\mathsf{g}) = \mathsf{H1}$, $hitno(\mathsf{g}) = 0$, $vzone(\mathsf{g}) = \mathsf{V0}$, and $vitno(\mathsf{g}) = 3$.

For clarity, terminal labels etc. have been omitted in Figure 6.2. The legend in the bottom-right corner shows the terminal names and symbolic significances for each master-slice.

6.3 Structure Extraction

As the first step in the optimization, the connectivity structure of the subdatapath under compaction is extracted. This is done by discovering circuits that are possible master-slices, usable for regularly composing the entire compacted subdatapath (Section 2.6.4).

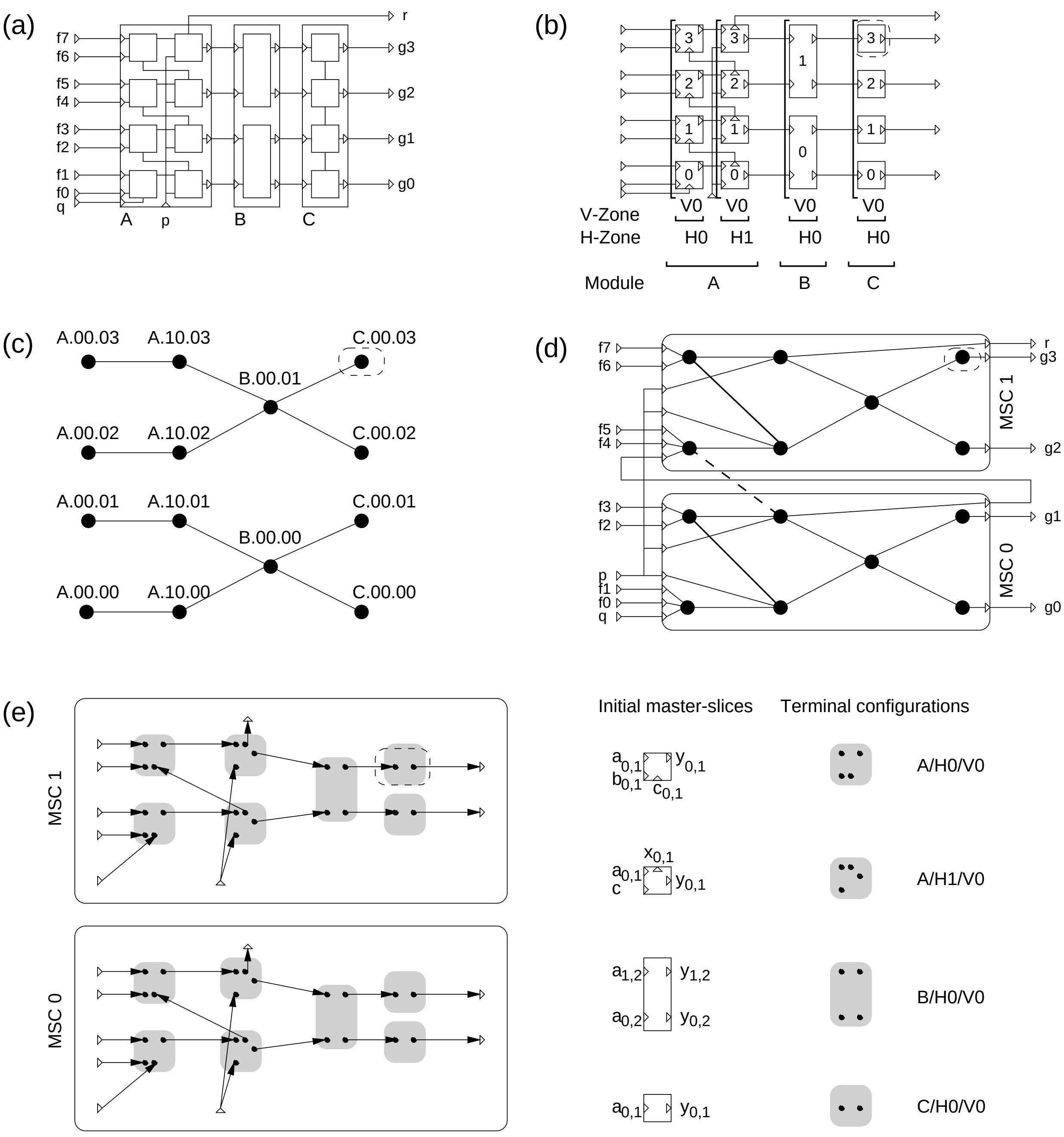


Figure 6.2: Flattening, structure extraction and regularity analysis

While each of the module instances in the subdatapath already has a known v-segment-based bit-slice structure[2], this stage collects data *across* module boundaries. In its current implementation, extraction relies on, and is limited by, the known regular structure of each module instance. Using SDI, this information is provided by the module generators (Section 3.2.2).

6.3.1 Requirements on Master-Slice Candidates

The extraction of potential master-slices, called *master-slice candidates* (MSC), aims at discovering bit-sliced structures across module-instance boundaries. It is performed under two conflicting requirements:

1. MSCs should contain as much *interconnected* logic as possible. The potential gains during logic optimization increase with the size of the circuits being processed. The processing of larger MSCs will lead to faster, smaller compacted modules.
2. MSCs should be small in terms of contained logic, and offer high potential for regular replication. The smaller problem size will reduce computation times (e.g., for logic optimization and microplacement), while the emphasis on replication will lead to very regular compacted structures.

We use the basic datapath topology (Section 2.3.1) to devise a compromise between these conflicting objectives. As an intermediate step, we determine raw master-slice candidates.

A *raw master-slice candidate* (rMSC) in the flattened network skeleton N_f is a flattened network skeleton whose internal nodes, edges and TTNs are restricted to those occurring in a connected subgraph (disregarding the direction of edges) of the graph

$$(In\langle N_f\rangle, \{(v_1, v_2) \in E\langle N_f\rangle \mid module(v_1) \neq module(v_2) \wedge v_1, v_2 \in In\langle N_f\rangle\}).$$

$\mathfrak{M}_r(N_f)$ is the set of rMSCs of N_f.

Algorithm 4: Finding raw master-slice candidates

We achieve the first requirement of larger MSCs by building the rMSCs across module boundaries (inter-module connections are being followed). Aiming at the second requirement, MSC size is limited by disregarding intramodule edges, which are often inter-bit-slice connections.

Example 93 Continuing the last example, Figure 6.2.c shows the undirected graph corresponding to the directed graph

$$(In\langle\mathcal{D}_f\rangle, \{(v_1, v_2) \in E\langle\mathcal{D}_f\rangle \mid module(v_1) \neq module(v_2) \wedge v_1, v_2 \in In\langle\mathcal{D}_f\rangle\}).$$

---

[2] Remember that more complex, non-bit-sliced modules are treated as hard-macros (Explanation 5).

Note how the intra-module edges have disappeared. In the figure, the label of a node (=v-segment) v has the form

$$module(v) \text{ "." } hzone(v)\ vzone(v) \text{ "." } hitno(v)\ vitno(v).$$

After finding the rMSCs, we turn them into MSCs by recovering those intra-module connections that are also intra-rMSC connections. Furthermore, since an rMSC doesn't have any primary ports (it is defined only on the units in $In\langle N_f\rangle$), they have to be recovered, too.

Algorithm 5: Refining an rMSC into an MSC

Require: N_f is a flattened network skeleton, $R \in \mathfrak{M}_r(N_f)$

Ensure: R is an MSC

```
{inherit MSC-internal TTNs from $N_f$ and add MSC primary ports}
$Ttn\langle R\rangle \leftarrow \emptyset$
for all $n \in \{((u,p),(v,q)) \in Ttn\langle N_f\rangle\}$ do
  if $(u,v) \in E\langle R\rangle$ then {MSC-internal net?}
    $Ttn\langle R\rangle \leftarrow Ttn\langle R'\rangle \cup \{n\}$
  else {connection to primary port}
    if $u \notin V\langle R\rangle$ then {source outside}
      create new primary input port $u'$ with terminal $(u', (u.sym(p), sig(p)))$ in $R$
      $Pi\langle R\rangle \leftarrow Pi\langle R\rangle \cup \{u'\}$
      $Ttn\langle R\rangle \leftarrow Ttn\langle R'\rangle \cup \{((u', (u.sym(p), sig(p))), (v,q))\}$
    else {sink outside}
      create new primary output port $v'$ with terminal $(v', (v.sym(q), sig(q)))$ in $R$
      $Po\langle R\rangle \leftarrow Po\langle R\rangle \cup \{v'\}$
      $Ttn\langle R\rangle \leftarrow Ttn\langle R\rangle \cup \{((u,p), (v', (v.sym(q), sig(q))))\}$
    end if
  end if
end for
{collect all nodes}
$V\langle R\rangle \leftarrow In\langle R\rangle \cup Pi\langle R\rangle \cup Po\langle R\rangle$
{update edges from TTNs}
$E\langle R\rangle = \{(u,v) \mid \exists\, p,q : ((u,p),(v,q)) \in Ttn\langle R\rangle\}$
```

The result of applying Algorithm 5 to each rMSC in $\mathfrak{M}_r(N_f)$ yields $\mathfrak{M}(N_f)$, the set of master-slice candidates in the flattened network skeleton N_f. Note that a new primary port, whose variable is qualified with the original node, is created for each TTN that connects to a primary port outside of the rMSC. Each of the newly created primary ports has a fanin (fanout) of exactly 1. Figure 6.3 shows an example for this connectivity.

Example 94 Figure 6.2.d shows the result of applying Algorithm 5 to the rMSCs of Figure 6.2.c. Note that intra-MSC intra-module edges have been recovered (shown

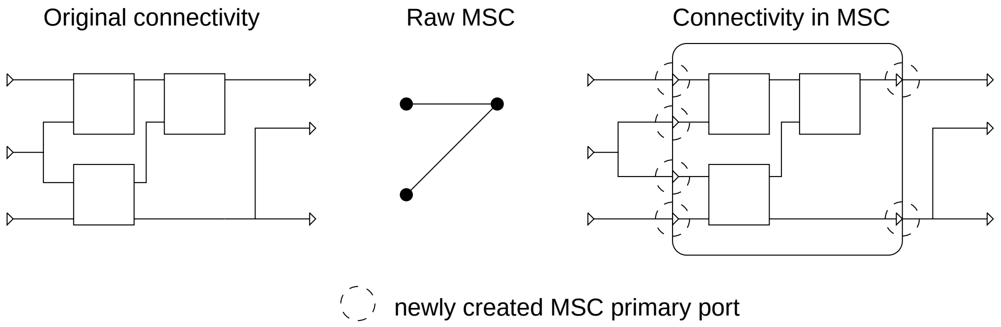


Figure 6.3: Creating new primary ports in MSCs

with wider lines), and that inter-MSC intra-module edges are still disregarded (shown with a dashed line).

Furthermore, all nets crossing MSC boundaries (this includes those connected to primary ports of the subdatapath) now end at newly created primary ports within the MSC (compare with Figure 6.2.b and Figure 6.3).

By now, we have extracted the connectivity structure from the original subdatapath in a manner that still allows to backwards-trace each component to the original flattened network skeleton. No information has been lost in the process.

6.4 Regularity Analysis

Next, we will search for regular structures by looking for similarities between master-slice candidates. This analysis is based on the detection of special forms of isomorphism between MSCs.

However, it is not sufficient to perform the isomorphism tests on the MSCs: Since the edges in an MSC may represent multiple TTNs, they are too coarse (Figure 6.4), we need the finer granularity obtained by expanding each node v in the MSCs into separate nodes for the terminals of v. This representation is the *terminal graph* $G_{Ttn} = (V_{Ttn}, E_{Ttn})$ of the MSC.

Example 95 Figure 6.4 shows a smaller example for the structure extraction and regularity analysis process. Figure 6.4.b shows the rMSCs extracted from the soft-subdatapath in Figure 6.4.a. Note that the two rMSCs are isomorphic, even though the corresponding portions of the datapath clearly aren't. After expanding the rMSCs via an intermediate MSC step (not shown) into the more detailed terminal graphs, this false isomorphism can be recognized using the techniques of the next section.

Definition 96 The tuple $(\mathit{module}, \mathit{hzone}, \mathit{vzone}, \mathit{sym}, \mathit{sig}, \mathit{hoffs}, \mathit{voffs}, \mathit{hitno}, \mathit{vitno})$ is a *terminal label*.

1. *module* is a module.
2. *hzone* is a h-zone.

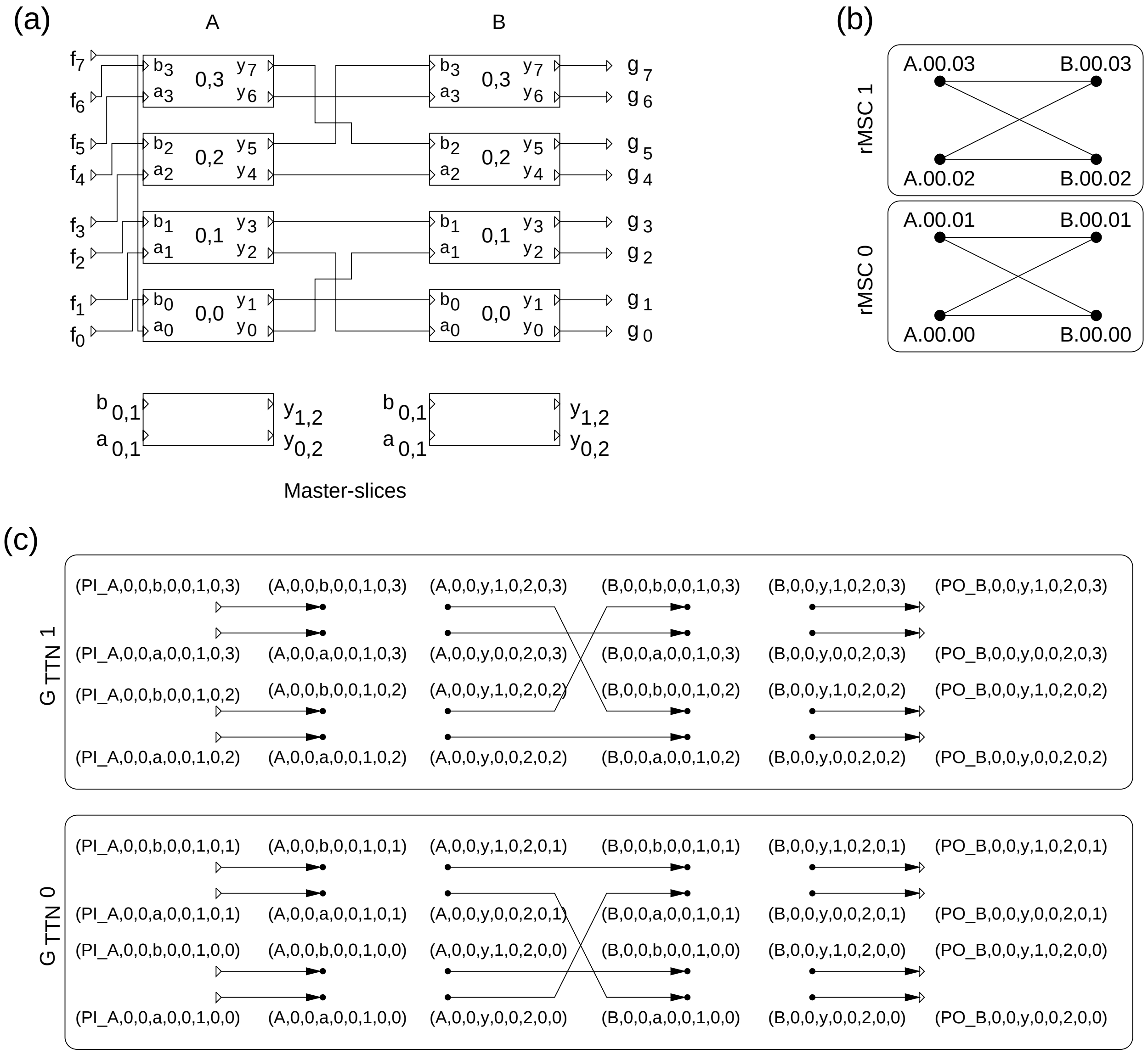


Figure 6.4: Increased precision of terminal graphs

3. *vzone* is a v-zone.

4. $sym \in \Sigma^+$.

5. $sig \in \mathbb{Z}$.

6. $hoffs \in \mathbb{N}_0$.

7. $voffs \in (\mathbb{Z}\backslash\{0\}) \cup \{\perp\}$.

8. $hitno \in \mathbb{N}_0$.

9. $vitno \in \mathbb{N}_0$.

Each node in the terminal graph is a terminal label. We expand the original TTNs to have terminal labels as source and sink, thus obtaining the edges in the terminal graph.

$(|\mathfrak{M}|^2 - |\mathfrak{M}|)/2$ isomorphism tests are required to check similarity between all MSC pairs in $\mathfrak{M}$ (Section 6.3). By exploiting sorted sequences of the terminal labels, each isomorphism test can be performed in $O(|V_{Ttn}| + |E_{Ttn}|)$ instead of the usual NP-complexity [Deo74].

6.4.1 Building Terminal Graphs

A fundamental operation during the creation of terminal graphs from MSCs is the creation of terminal labels from terminals. This is straightforward for units, since all information is available in the flattened network skeleton of the MSC. For the primary input (output) ports newly created by Algorithm 5, however, it has to be propagated from their fanout (fanin) terminal.

Algorithm 6 determines the label for the given terminal. Note that $sig(l)$ is the original significance of the terminal in the master-slice. It is recomputed by reversing the significance-altering operations of Definition 46 and Definition 62.

Example 97 The effect of Algorithm 6 can be seen in Figure 6.4. Consider the $\mathbf{a}_3$ input terminal of the top ($hitno = 0, vitno = 3$) v-segment of module A. Since it is located on a unit, the hierarchy information is already available in the flattened network skeleton. Only the original significance in the master-slice has to be recomputed. The example assumes that each module only has a single h-zone (with $hoffs = 0$) containing a single v-zone. In this context, combining the known $hoffs = 0, voffs = 1$ of the terminal with the known $hitno = 0, vitno = 3$, we backtrace $\mathbf{a}_3$ to the symbolic significance $\mathbf{a}_{0,1}$ in the master-slice. The result is the terminal label $(\mathsf{A}, 0, 0, \mathbf{a}, 0, 0, 1, 0, 3)$ in the terminal graph.

For the newly created primary ports, the required information is propagated from their connected unit terminals. E.g., the TTN in the network skeleton that connects the primary datapath input f_5 with the unit input terminal $(\mathsf{A}, \mathbf{a}_3)$, thus crossing the MSC boundaries, leads to the creation of a new input primary port during Algorithm 5. When determining the terminal label for the output terminal of this primary port in Algorithm 6, we inherit most of the data from the sink. In this case, the unit input

---

Algorithm 6: Create a terminal label

Require: t is a terminal of a node in MSC m

Ensure: return a valid terminal label

```
GenLabel(t,m):terminal label
$w \leftarrow nod(t)$
$l \leftarrow$ create new terminal label
if $w \in In\langle m\rangle$ then {terminal on a unit?}
    $M \leftarrow mr\langle master(w)\rangle$
    $module\langle l\rangle \leftarrow module(w)$
    $hzone\langle l\rangle \leftarrow hzone(w)$
    $vzone\langle l\rangle \leftarrow vzone(w)$
    $sym\langle l\rangle \leftarrow sym(t)$
    $hoffs\langle l\rangle \leftarrow hoffs\langle hzone(w)\rangle$
    $voffs\langle l\rangle \leftarrow voffs\langle M\rangle(t)$
    $hitno\langle l\rangle \leftarrow hitno(w)$
    $vitno\langle l\rangle \leftarrow vitno(w)$
    {compute initial significance (may be $\perp$)}
    $sig\langle l\rangle \leftarrow sig(t) - hoffs\langle l\rangle \cdot hitno(w) - voffs\langle l\rangle \cdot vitno(w)$
else {terminal is on a primary port}
    if $w \in Pi\langle m\rangle$ then {on a primary input?}
        {inherit hierarchy data from sink}
        $\exists(t, (v, q)) \in Ttn\langle m\rangle$ {$t$ always has single fanout, Algorithm 5}
        $module\langle l\rangle \leftarrow$ "PI_" $module(v)$ {prefix module name}
    else {it's a primary output port}
        {inherit hierarchy data from source}
        $\exists((v, q), t) \in Ttn\langle m\rangle$
        $module\langle l\rangle \leftarrow$ "PO_" $module(v)$ {prefix module name}
    end if
    {now propagate other hierarchy data}
    $M \leftarrow mr\langle master(v)\rangle$
    $hzone\langle l\rangle \leftarrow hzone(v)$
    $vzone\langle l\rangle \leftarrow vzone(v)$
    $sym\langle l\rangle \leftarrow sym(q)$
    $hoffs\langle l\rangle \leftarrow hoffs\langle hzone(v)\rangle$
    $voffs\langle l\rangle \leftarrow voffs\langle M\rangle(q)$
    $hitno\langle l\rangle \leftarrow hitno(v)$
    $vitno\langle l\rangle \leftarrow vitno(v)$
    {compute initial significance (may be $\perp$)}
    $sig\langle l\rangle \leftarrow sig(q) - hoffs\langle l\rangle \cdot hitno(v) - voffs\langle l\rangle \cdot vitno(v)$
end if
```

terminal $(\mathsf{A}, \mathbf{a}_3)$. The only change occurs on the *module* component of the terminal: The module containing the sink is prefixed with "PI_". In this manner, all primary input (output) ports will be assigned to a pseudo-module with a "PI_" ("PO_") prefix. This procedure yields $(\mathsf{PI\_A}, 0, 0, \mathbf{a}, 0, 0, 1, 0, 3)$ as the final terminal label.

Using Algorithm 6, we can now build a terminal graph (V_{Ttn}, E_{Ttn}) for each MSC in $\mathfrak{M}(N_f)$.

Algorithm 7: Building terminal graphs

```
Require: $N_f$ flattened network skeleton,
  $\mathfrak{M}(N_f) = \{m_1, \ldots, m_n\}$ set of master-slice candidates,
Ensure: $V_{Ttn}$ is an array of sequences of terminal labels
  $E_{Ttn}$ is an array of sequences of terminal labels pairs

  MAKETTNGRAPHS:void
  for $j = 1$ to $n$ do {init TTN graphs for all MSCs}
    {expand v-segments into their terminals and determine labels}
    $V_{Ttn}[m_j] \leftarrow$ sort $\bigcup_{v \in V\langle m_j \rangle} \bigcup_{t \in (To(v) \cup Ti(v))} \{\textsc{GenLabel}(t, m_j)\}$
    {expand TTNs into edges by determining labels for source and sink}
    $E_{Ttn}[m_j] \leftarrow$ sort $\bigcup_{(t_o, t_i) \in Ttn\langle m_j \rangle} \{(\textsc{GenLabel}(t_o, m_j), \textsc{GenLabel}(t_i, m_j))\}$
  end for
```

Algorithm 7 builds the terminal graph for each MSC in $\mathfrak{M}$. Note how v-segments and their terminals are expanded into terminal labels. V_{Ttn} and E_{Ttn} are arrays of sequences of terminals labels and terminal label pairs, respectively. The sort order may be arbitrary, as long as it is consistent. The key consists of all components of the terminal label, such that the order of components within a terminal label is also their priority during the sort operation.

Example 98 In this example, the relevant features in Figure 6.2 have been highlighted with a dashed rounded rectangle. For clarity, node labels have been omitted in the terminal graphs shown in Figure 6.2.e. However, the legend in the bottom left corner shows the master-slices with their corresponding terminal configurations. E.g., the left dot of the terminal configuration for the master-slice of C/H0/V0 corresponds to the terminal $\mathbf{a}_{0,1}$. The original v-segment boundaries have been highlighted with a light grey background.

Consider the top-left v-segment of Figure 6.2.b. In Figure 6.2.c, it is labeled C.00.03. When expanding this node into the terminal graph Figure 6.2.d, it results in the nodes $(\mathsf{C}, \mathsf{H0}, \mathsf{V0}, \mathbf{a}, 0, 0, 1, 0, 3)$, $(\mathsf{C}, \mathsf{H0}, \mathsf{V0}, \mathbf{y}, 0, 0, 1, 0, 3)$. Analogously, the TTN connecting its output with the MSC primary output port (created during Algorithm 5) becomes $((\mathsf{C}, \mathsf{H0}, \mathsf{V0}, \mathbf{y}, 0, 0, 1, 0, 3), (\mathsf{PO\_C}, \mathsf{H0}, \mathsf{V0}, \mathbf{y}, 0, 0, 1, 0, 3))$.

6.4.2 Constrained Isomorphism

We will be using a constrained form of isomorphism tests, relying on node labels, to avoid the NP-complexity of a general isomorphism test.

For two terminal graphs to be isomorphic in this context, nodes at the same position in the sort order of V_{Ttn} must have equal values for all components except *vitno* and *hitno* (Algorithm 8). The reasoning behind this constraint is, that these nodes will have the same underlying logic (gate network), which is independent of the specific iteration numbers.

Algorithm 8: Isomorphism constraints on terminal labels

Require: p, q are terminal labels

Ensure: return *TRUE* if all non-iteration number components are equal

```
LABELEQUAL(p, q):boolean

    LABELEQUAL ← $module\langle p\rangle = module\langle q\rangle$ and
                 $hzone\langle p\rangle = hzone\langle q\rangle$ and
                 $vzone\langle p\rangle = vzone\langle q\rangle$ and
                 $sym\langle p\rangle = sym\langle q\rangle$ and
                 $sig\langle p\rangle = sig\langle q\rangle$ and
                 $hoffs\langle p\rangle = hoffs\langle q\rangle$ and
                 $voffs\langle p\rangle = voffs\langle q\rangle$
```

When checking for isomorphism with regard to connectivity, iteration numbers do have to be considered, though. Multiple iterations of the same logic may be wired differently, thus resulting in different functions when merged. However, since a simple equality comparison of iteration numbers would fail (the terminals are usually located in different iterations, after all), we compare relative iteration numbers. These are determined by comparing the offset between the iteration numbers of the source and sink terminal of a TTN (Algorithm 9).

Example 99 Using these isomorphism requirements, the terminal label $(\mathsf{A}, 0, 0, \mathbf{y}, 0, 0, 2, 0, 0)$ would correspond to $(\mathsf{A}, 0, 0, \mathbf{y}, 0, 0, 2, 0, 2)$, but not to $(\mathsf{A}, 0, 0, \mathbf{y}, 1, 0, 2, 0, 0)$.

For edges, the TTN $e_1 = ((\mathsf{A}, 0, 0, \mathbf{y}, 1, 0, 2, 0, 0), (((\mathsf{B}, 0, 0, \mathbf{b}, 0, 0, 1, 0, 0)))$ would correspond to $e_2 = ((\mathsf{A}, 0, 0, \mathbf{y}, 1, 0, 2, 0, 1), (((\mathsf{B}, 0, 0, \mathbf{b}, 0, 0, 1, 0, 1)))$, but not to $e_3 = ((\mathsf{A}, 0, 0, \mathbf{y}, 1, 0, 2, 0, 2), (((\mathsf{B}, 0, 0, \mathbf{b}, 0, 0, 1, 0, 3)))$. In the first case, both TTNs e_1, e_2 have $\Delta\, hitno = 0$ and $\Delta\, vitno = 0$, while in the second case, e_3 has $\Delta\, hitno = 0$ and $\Delta\, vitno = 1$.

The entire isomorphism test algorithm is listed in Algorithm 10. It applies Algorithm 8 and Algorithm 9 to all nodes and edges in the terminal graph. Iff all requirements are fulfilled, it returns *TRUE*.

At this stage, we have introduced all parts of the isomorphism test, and can now proceed to define the entire regularity analysis algorithm.

After the execution of Algorithm 11, those MSCs with $master = (\perp, \perp)$ will be considered as the *merged master-slices* (mmS) of the combined modules (the

Algorithm 9: Isomorphism constraints on terminal labels in TTNs

Require: n_1, n_2 are edges in the TTN graph

Ensure: return *TRUE* if all non-iteration number components and the offsets between source and sink are equal

```
LabelEqualRel($n_1, n_2$):boolean

LabelEqualRel ←

    LabelEqual($t_o\langle n_1\rangle, t_o\langle n_2\rangle$) and
    LabelEqual($t_i\langle n_1\rangle, t_i\langle n_2\rangle$) and
    $(hitno\langle t_o\langle n_1\rangle\rangle - hitno\langle t_i\langle n_1\rangle\rangle) = (hitno\langle t_o\langle n_2\rangle\rangle - hitno\langle t_i\langle n_2\rangle\rangle)$ and
    $(vitno\langle t_o\langle n_1\rangle\rangle - vitno\langle t_i\langle n_1\rangle\rangle) = (vitno\langle t_o\langle n_2\rangle\rangle - vitno\langle t_i\langle n_2\rangle\rangle)$
```

Algorithm 10: Label-based test for constrained isomorphism

Require: m MSC (potential master), i MSC (potential instance of m)

Ensure: return *TRUE* is i could be an instance of m

```
IsIsomorphic(m, i):boolean
{run cheap tests first ...}
if $(|V_{Ttn}\langle m\rangle| \neq |V_{Ttn}\langle i\rangle|)$ or $(|E_{Ttn}\langle m\rangle| \neq |E_{Ttn}\langle i\rangle|)$ then
    IsIsomorphic ← FALSE
else {more expensive tests}
    {check terminals (nodes), but disregard iteration numbers}
    for $l = 1$ to $|V_{Ttn}[m]|$ do
        {elementwise comparison in sorted lists}
        $p \leftarrow V_{Ttn}[m][l]$
        $q \leftarrow V_{Ttn}[i][l]$
        if not LabelEqual(p,q) then {see Algorithm 8}
            IsIsomorphic ← FALSE
        end if
    end for
    {check TTNs (edges), compare relative iteration numbers}
    for $l = 1$ to $|E_{Ttn}[m]|$ do
        {elementwise comparison in sorted lists}
        $s \leftarrow E_{Ttn}[m][l]$
        $t \leftarrow E_{Ttn}[i][l]$
        if not LabelEqualRel(s,t) then {see Algorithm 9}
            IsIsomorphic ← FALSE
        end if
    end for
end if
IsIsomorphic ← TRUE
```

Algorithm 11: Regularity analysis

```
Require: $\mathfrak{M} = \{m_1, \ldots, m_n\}$ set of master-slice candidates
Ensure: $master(m_1), \ldots, master(m_n)$ set to an isomorphic master-slice candidate, or $(\perp, \perp)$ if no such candidate exists in $\mathfrak{M}$.

  {setup data structures for TTN graphs, Algorithm 7}
  MakeTTNGraphs
  {now check for similarities between MSCs}
  for $j = 1$ to $n$ do {potential master}
    for $k = j + 1$ to $n$ do {potential instance}
      $master(m_k) \leftarrow (\perp, \perp)$ {pessimism: no isomorphic master}
      if IsIsomorphic($m_j, m_k$) then {compare MSCs using Algorithm 10}
        $master(m_k) \leftarrow (m_j, \perp)$ {found one, remember the master}
      end if
    end for
  end for
```

entire subdatapath under compaction), while the other MSCs are treated as their instances.

$\mathfrak{M}_m(N_f)$ is the set of merged master-slices for the network skeleton N_f.

Example 100 The application of Algorithm 11 to Figure 6.2.c detects that MSC 1 is isomorphic to MSC 0 even at the terminal graph level. Thus, the entire subdatapath is composed only of instances of a single merged master-slice (mmS 0). It suffices to apply all further optimization operations to the logic functions of mmS 0, and propagate them into the isomorphically corresponding elements of the terminal graph for MSC 1 (the only instance of mmS 0).

In Figure 6.4.c, however, the algorithm doesn't discover such regularity: While both MSCs consist of the same v-segments (the MSCs are already isomorphic, and the terminal labels in the terminal graphs also correspond), the interconnection pattern (as seen in the terminal graphs) is different. Both of the MSCs are thus mMSs implementing different logic functions, and will have to be optimized individually.

6.5 Logic Optimization and Mapping

Once the merged master-slices (mmS) spanning the subdatapath under compaction have been determined from the MSCs, each mmS is processed using conventional logic synthesis, optimization, and mapping methods (generalized as *logic processing*) [CoDi96] [MuBS95]. This step actually merges and compacts the logic functions originating in different modules, but only within mmS boundaries. The result are *optimized master-slices* (omS), with $\mathfrak{M}_o(N_f)$ being the set of optimized master-slices for the flattened network skeleton N_f. Each instance (determined by Algorithm 11) is called an *optimized slice* (oS).

6.5.1 Tool Integration

By proceeding in this manner, the regular architecture of the datapath itself is preserved. Since mapping is FPGA-architecture dependent, logic processing is the first truly technology-specific step in the SDI design flow. However, it is only loosely coupled with SDI and relies on standard tools. SIS 1.3 [Sent92] is used for logic synthesis and optimization, and optionally FlowMap [CoDi94] for technology mapping to 4-LUTs. By choosing between the "xl" (MIS-PGA) commands in SIS, or FlowMap, the user can make a trade-off between delay over area minimization (Chapter 8). The integration of other tools in this phase is just a matter of netlist format compatibility. E.g., one could easily embed the Synopsys FPGA Compiler [Syno96a] for logic synthesis and mapping, or Tech. Univ. of Munich's TOS-TUM [LeWE96] dedicated technology mapper.

For logic processing, each of the mMSs is flattened down from v-segment level into individual gates, and separately optimized and mapped (possibly in parallel, Figure 2.20). If external tools were to be used, the gate networks could be exported using one of the netlist formats supported by SIS, and would be imported again after optimization. In the current implementation, this is not required since all algorithms (even FlowMap) are integrated in SIS, and thus operate on the internal circuit representation.

6.5.2 Pre- and Post-Compaction Isomorphism

While synthesis, optimization and mapping might entirely change the internal structure of the gate networks, primary ports are always left intact. They still match the pre-optimization ports in the mmS, which in turn have isomorphically corresponding primary ports in their instances in the flattened network skeleton.

We may thus define the network skeleton $N_o(N_f)$ as the *optimized network skeleton* of the flattened network skeleton N_f. N_o has the same global function as N_f, but is composed of instances (note: *not* iterations) of the optimized master-slices in $\mathfrak{M}_o(N_f)$. Each instance, called an optimized slice in this context, consists of logic processed and mapped cells (4-LUTs for the XC4000). The TTNs connect the terminals isomorphically corresponding between N_f and N_o (Figure 6.6). Furthermore, each optimized slice in $In\langle N_o\rangle$ has a master in $\mathfrak{M}_o(N_f)$. In this manner, the optimized slices, which now make up the subdatapath, can be embedded in the entire datapath.

The complex hierarchical and regular constructs used during structure extraction and regularity analysis are mostly discarded now: Each compacted subdatapath will consist simply of a vertical arrangement of instances of one or more mMSs. The entire compacted subdatapath will thus consist only of a single h-zone. Even the v-zone concept is not valid anymore, since the same mMSs may occur as master-slice multiple times (using the isomorphism obtained during regularity analysis, not iteration).

Example 101 Figure 6.5.a shows the gate network of the mmS from Figure 6.2.e.

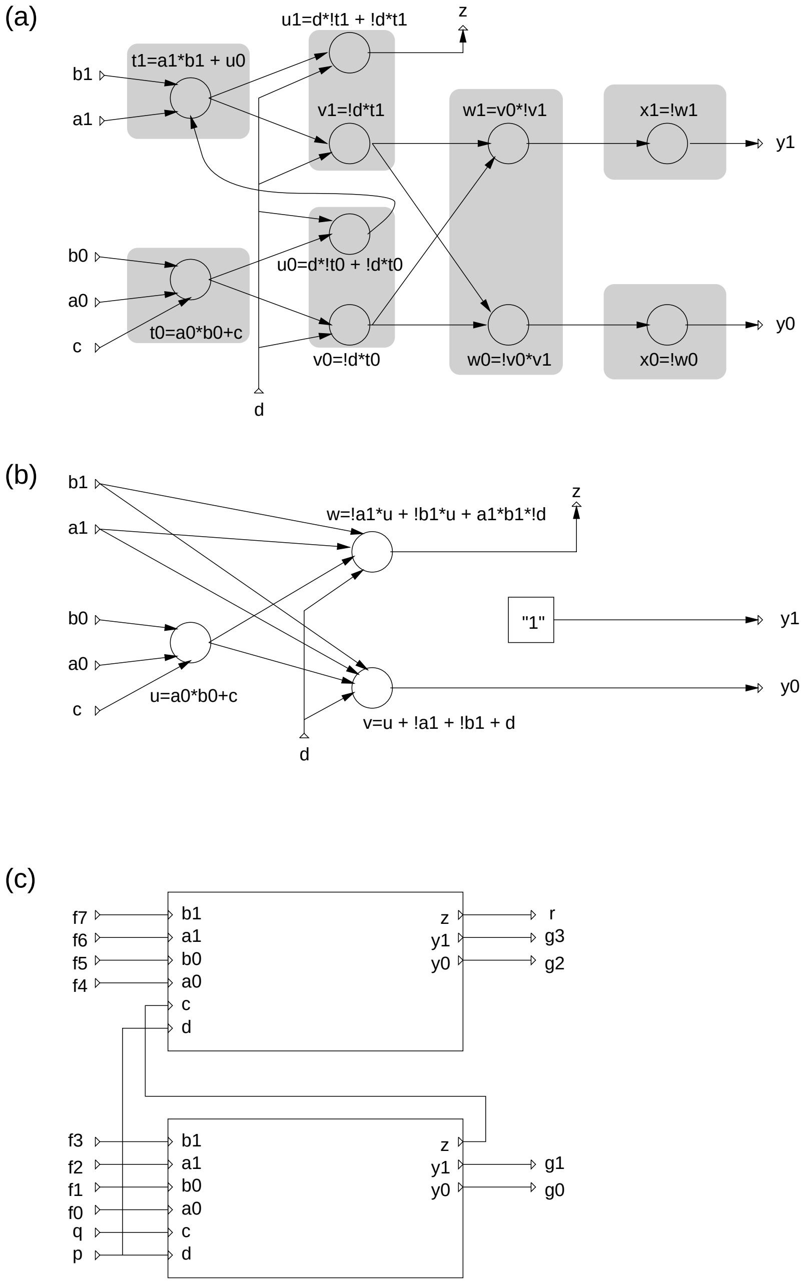


Figure 6.5: Effects of logic processing: (a) original, (b) processed circuit, (c) compacted subdatapath

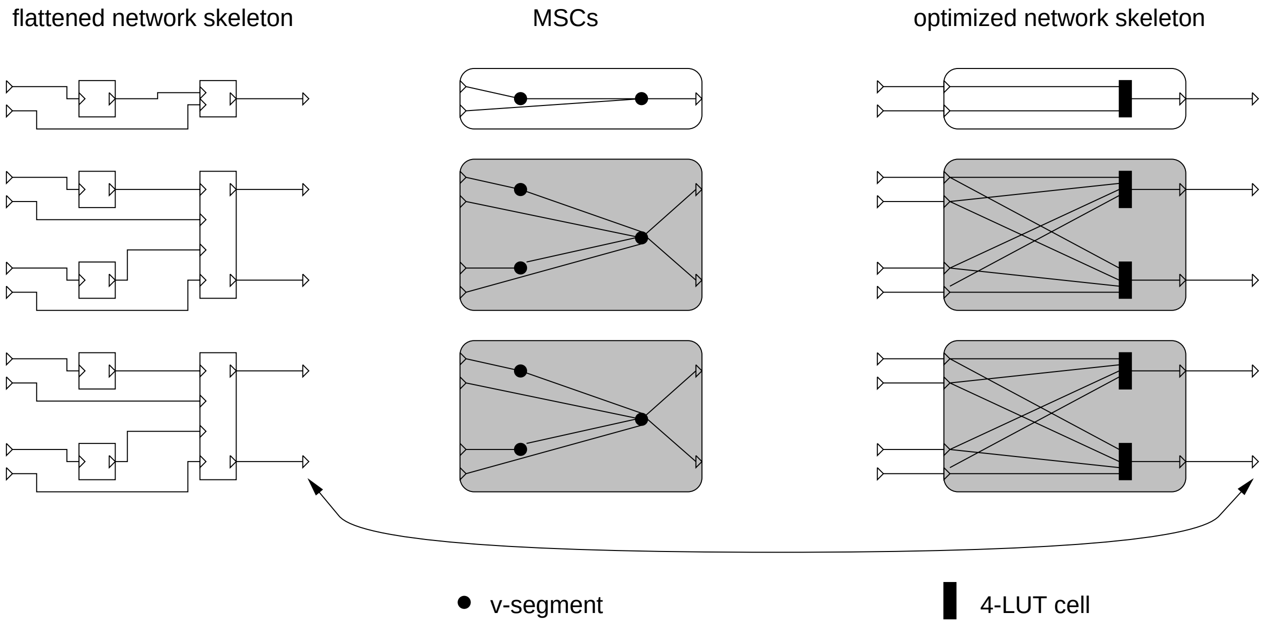


Figure 6.6: Isomorphic correspondence between terminals on N_f, $\mathfrak{M}$, and N_o

The original v-segments have been highlighted with a light grey background. Note how the primary ports still correspond between these hierarchy levels, and how a single v-segment may contain multiple gates. This gate network would use 10 4-LUTs (cells) on an XC4000, and has a total delay of 6 LUT levels.

Optimization using the SIS procedure `script` followed by a technology mapping using the "xl" commands produces the omS in Figure 6.5.b. Its area has been reduced to 4 4-LUTs (possibly down to 3 LUTs, if a logic "high" signal is available elsewhere, e.g. VCC), and the delay is shortened to 2 LUT levels.

Figure 6.5.c sketches the resulting topology after compacting the subdatapath Figure 6.2.a. Note how the optimized slices (instances of the omS) are connected with the original subdatapath primary ports, making up the optimized network skeleton N_o. All significance information has remained on the subdatapath primary ports, it is not considered during logic processing.

Figure 6.6 also demonstrates the isomorphic correspondence of terminals on primary ports and units (=oSs) between N_f and N_o.

6.6 Summary and Relations between Structures

After determining a soft-subdatapath to compact (Algorithm 2 and 3), we flatten it down to v-segment level, obtaining the flattened network skeleton N_f.

The structure of N_f is extracted by searching the units and their inter-module connections of N_f for connected subgraphs, which we have called raw master-slice candidates (rMSC, Algorithm 4).

Next, we add intra-rMSC connections from N_f to the rMSCs, and create new primary ports on the rMSCs for each TTN in N_f crossing rMSC boundaries to obtain master-slice candidates (MSCs, Algorithm 5).

We look for regularity (master-instance relations) between MSC pairs by testing them for a constrained isomorphism (Algorithm 11). For each isomorphically unique structure, we isolate a merged master-slice (mmS) that contains logic originating in different modules, but within MSC boundaries.

After applying logic processing operations to each of the mmS, we obtain optimized master-slices (omS) with reduced area and delay. With the isomorphism relations computed during regularity analysis, we can determine those parts of N_f corresponding to instances of the omS. Thus, we are able to assemble the optimized network skeleton N_o from the optimized slices (oS). The result is a regular circuit with the same global function as the initial soft-subdatapath, but reduced area and delay.

6.7 Effects on Placement

Logic processing yields a set of optimized gate networks that have to be instantiated and interconnected isomorphically corresponding to the original subdatapath in order reproduce its global function. However, all placement information has been lost during the process: The original placement of cells inside of a v-segment (provided by the module generators) has been invalidated by the structural changes during logic processing. The arrangement of the v-segments themselves was lost during structure extraction, since the connected subgraphs might span multiple geometrically non-adjacent v-segments. Since the v-segments inside of a connected subgraph will be merged, their individual placement also becomes invalid.

7 Microplacement

This chapter will examine the re-placement phase already introduced in Section 2.7 in greater detail[1]. As described in Section 6.7, logic processing invalidates both the vertical topological placement of v-segments (now instances of merged master-slices, Section 6.5) as well as the geometrical cell placement (Section 2.7.3) inside of each v-segment.

7.1 Vertical Topological Placement

The restoration of a vertical topological placement is based on the significance information for the primary ports of the subdatapath, and the fundamental topology of the datapath (Section 2.3.1). The procedure runs in

$$O((1 + \max|Fo|) \cdot (msb - lsb)),$$

where msb (lsb) are the maximal (minimal) significances of primary ports in the subdatapath, and $\max|Fo|$ is the maximal fanout of primary input ports. However, it cannot generate an optimal placement for circuits that have very irregular (non-bit-sliced) interconnection patterns.

In the first phase, Algorithm 12 attempts to build a vertical topological placement by ordering the optimized slices (oSs) in $In\langle N_o\rangle$ by their connections to primary outputs. This ordering proceeds from bottom to top with ascending significances. Within a given significance, the sequence of multiple oSs is arbitrary. After aiming at a homogeneous oS-to-PO arrangement, the remaining unplaced oSs are inserted into the existing topological placement by following their connectivity to primary inputs, also with ascending significance. When all PIs and POs are static (Definition 50), Algorithm 12 cannot obtain a regular topological placement. This may occur, e.g., when a module has a local controller. In that case, the control logic is separated from the regular part of the module and added to the main controller (Section 2.3.2). It is no longer subject to regular processing by SDI.

For bit-sliced circuits, the result is a very regular arrangement, where especially the output side is kept free of significance permutations. Applying Algorithm 12 to non-bit sliced circuits (with many non-contiguous or non-abutting significances on the MSCs), yields very inefficient layouts, however. Were SDI to be extended to also process these circuits, a full linear place-

---

[1] Though the actual SDI implementation is even more complex than described here.

Algorithm 12: Restoring vertical topological placement after compaction

Require: N_o is an optimized network skeleton.

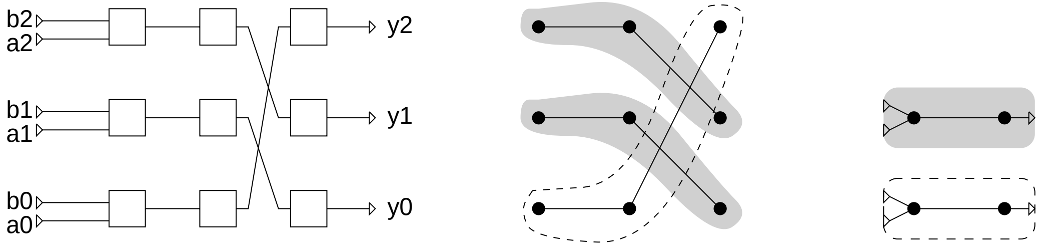
Ensure: $Vplace(N_o)$ is vertical topological placement of oSs in N_o

```
  {work sequence, pairs $(u, b)$ of oS $u$ and significance $b$}
  $tempplace(\mathfrak{M}) \leftarrow ()$
  $placed[*] \leftarrow FALSE${init all items}
  {determine significance bounds for $N_o$}
  $lsb \leftarrow \min_{v \in (Pi\langle N_o\rangle \cup Po\langle N_o\rangle), t \in (Ui\langle v\rangle \cup Uo\langle v\rangle)} sig(t)$
  $msb \leftarrow \max_{v \in (Pi\langle N_o\rangle \cup Po\langle N_o\rangle), t \in (Ui\langle v\rangle \cup Uo\langle v\rangle)} sig(t)$

  {scan for POs at significance $b$}
  for $b = lsb$ to $msb$ do
    $T_{po} \leftarrow$ all terminals of $Po\langle N_o\rangle$ with significance $b$
    for all $t \in T_{po}$ do
      if $((u, p), t) \in Ttn\langle N_o\rangle$ then {POs only have single fanin}
        if not $placed[u]$ then
          {add it to the topological placement}
          append $(u, b)$ to $tempplace$
          $placed[u] \leftarrow TRUE$
        end if
      end if
    end for
  end for

  {now scan PIs at significance $b$}
  for $b = lsb$ to $msb$ do
    $T_{pi} \leftarrow$ all terminals $Pi\langle N_o\rangle$ with significance $b$
    for all $t \in T_{pi}$ do
      for all $(t, (v, q)) \in Ttn\langle N_o\rangle$ do {PIs may have multiple fanouts}
        if not $placed[v]$ then
          insert $(v, b)$ into $tempplace = (\dots, (u_i, b_i), (v, b), (u_{i+1}, b_{i+1}), \dots)$
                      such that $b_i \le b \le b_{i+1}$
          $placed[v] \leftarrow TRUE$
        end if
      end for
    end for
  end for
  {get rid of significance, project only oS component}
  $Vplace(N_o) \leftarrow \widetilde{u}\langle (u, b) \in tempplace\rangle$
```

ment phase (similar to the approach sketched for horizontal floorplanning in Chapter 4) would be required.


(a) Subdatapath under compaction

(b) Raw MSCs

(c) Merged master-slices

(d) Post-compaction vertical topology

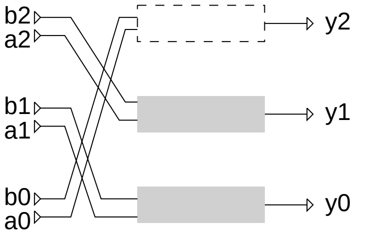


Figure 7.1: Post-compaction vertical topological re-placement

Example 102 Figure 7.1 shows an example for the application of Algorithm 12. The three rMSCs in (b) have been extracted from the subdatapath in (a). Further analysis discovers that two of the MSCs share a master-instance relation (highlighted with a grey background), while the third one (marked with a dashed border) seems to stand alone. Logic processing the masters results in the omSs in (c). Note how all placement (both topological and geometrical) has been lost. The algorithm determines the vertical topology shown in (d) for the arrangement of oSs. Observe the emphasis on keeping the output side significance permutation-free.

7.2 ILP for Horizontal Geometrical Node Placement

With a restored vertical topology of the optimized slices in the compacted subdatapath, we now proceed to place the units inside of each omS. As already described in Section 2.7.2, this placement is performed in a timing-driven manner, operates on a regular array of cells instead of the possibly irregular FPGA logic blocks (Section 2.7.3), and occurs in two phases (Section 2.7.4).

The first phase simultaneously processes all omSs to align units with control lines spanning the entire height of the subdatapath (Section 2.7.4, Figure 2.31). To compute optimal results, it has initially been implemented as a 0-1 integer linear program (ILP). For basics on the formulation of ILPs in general, and constraint-logic programming (CLP) problems in particular, see [Will93] [BeCo93] [Bart95] [Bart96].

However, even with optimizations [Bart95], the 0-1 ILP models become too unwieldy for larger circuits. Thus, an alternative heuristical implementation based on simulated annealing has also been provided. In our tests, it always reached the optimal placement (using the ILP solutions as reference) in very reasonable amounts of time (Section 7.4.3).

7.2.1 Determining the Placement Area

Since we are aiming at a geometrical placement, we require the extent of the placement area for each omS. It is calculated from the Floorplanner-specified BPLB-value (Explanation 3) using Algorithm 13.

Algorithm 13: Determining the placement area per omS

Require: $M \in \mathfrak{M}_o(N_f)$ is an optimized master-slice, bp is the target-BPLB value, $N_o(N_f)$ is the optimized network skeleton for the flattened network skeleton N_f.

Ensure: $h\langle M\rangle, l\langle M\rangle$ specify the extent of the placement area.

```
$h\langle M\rangle \leftarrow 0$
{get terminals connected to instances of M}
$I \leftarrow \{(u,p) \mid \exists\, v,q :$
        $((u,p),(v,q)) \in Ttn\langle N_o\rangle \wedge u \in Pi\langle N_o\rangle \wedge mr\langle master(v)\rangle = M\}$
$O \leftarrow \{(v,q) \mid \exists\, u,p :$
        $((u,p),(v,q)) \in Ttn\langle N_o\rangle \wedge v \in Po\langle N_o\rangle \wedge mr\langle master(u)\rangle = M\}$
$\mathfrak{S} \leftarrow$ logically complete sets of significances for the terminals $I \cup O$
{determine height to match target BPLB value in $bp$}
for all $S \in \mathfrak{S}$ do
  $h\langle M\rangle \leftarrow \max(h\langle M\rangle, \lceil (msb(S) - lsb(S))/bp \rceil)$
end for
{calculate length to fit all units into the now-fixed height}
$l\langle M\rangle \leftarrow \lceil |In\langle M\rangle| / h\langle M\rangle \rceil$
{determine length of entire $N_o$}
$l\langle N_o\rangle \leftarrow \max_{M \in \mathfrak{M}_o(N_f)} l\langle M\rangle$
```

Algorithm 13 calculates the height in cells by determining the largest complete significance intervals on primary ports in N_o, and dividing them by the fixed BPLB value. The length of the placement area is then calculated such that the placement area accommodates all units in the mMS (see Section 7.6 for an exception to this rule).

Example 103 After compaction, the subdatapath Figure 6.4.a will be composed of two optimized master-slices, each having a single oS (no isomorphic TTN graphs found, Section 6.4). The first omS 0 (shown in Figure 6.4.c as its pre-logic processing terminal graph $G_{Ttn}0$) connects to the primary output ports with variables $\mathbf{g}_0$, $\mathbf{g}_1$, $\mathbf{g}_2$, $\mathbf{g}_3$, and the primary input ports $\mathbf{f}_0, \mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_7$. $\mathfrak{S}$ thus is $\{\{0, 1, 2, 3\}, \{0, 1, 2\}, \{7\}\}$. Assuming a target BPLB value of $bp = 2$, the height will computed be $h\langle omS0\rangle = \lceil 3/bp\rceil = 2$ (due to $msb(\{0, 1, 2, 3\}) - lsb(\{0, 1, 2, 3\}) = 3 - 0 = 3$). Furthermore, assuming a post-compaction $|In\langle omS0\rangle| = 6$, the length of the placement area would become $l\langle omS0\rangle = \lceil 6/2\rceil = 3$.

Analogous computations would result in $h\langle omS1\rangle = 2$ and $l\langle omS1\rangle = 3$ (assuming the same BPLB value and number of post-logic processing units as for $omS0$).

7.2.2 Node Placement

Given $\mathfrak{M}_o$ as the set of optimized master-slices being microplaced, the solution variable $x_{v,j,m}$ is 1 iff the node $v \in V\langle m\rangle, m \in \mathfrak{M}_o$ is being placed in column $1 \leq j \leq l\langle m\rangle$ of the placement area.

First, we want to ensure that in each omS m, each node

$$v \in \{u \in V\langle m\rangle \mid u \in In\langle m\rangle \vee x\langle loc(u)\rangle = \perp\}$$

is placed exactly in one column j with $1 \leq j \leq l\langle m\rangle$. v is thus either a unit, or a primary port without a pre-specified horizontal location (a *floating* port):

$$\forall m \in \mathfrak{M}_o, v \in In\langle m\rangle : \sum_{j=1}^{l\langle m\rangle} x_{v,j,m} = 1 \tag{7.1}$$

The second set of constraints in the ILP enforces the height limit on the placement areas for each omS. It ensures for each mMS m that at most $h\langle m\rangle$ units in $In\langle m\rangle$ are placed in each column j with $1 \leq j \leq l\langle m\rangle$.

$$\forall m \in \mathfrak{M}_o, 1 \leq j \leq l\langle m\rangle : \sum_{v \in In\langle m\rangle} x_{v,j,m} \leq h\langle m\rangle \tag{7.2}$$

Note that multiple floating primary ports may be placed in the same column (they do not consume area in the layout, Section 5.3.1).

7.2.3 Control Signal Routing

The next constraint deals with the limited number (10 for the XC4000) of vertical long lines available for control routing in each vertical channel (Section 2.7.1, Section 2.7.4). Since control lines span multiple omS, they are handled at the level of the optimized network skeleton N_o.

The solution variable $l_{p,j}$ is 1 iff the control signal connected to primary port $p \in (Pi\langle N_o\rangle \cup Po\langle N_o\rangle) \wedge ptype(p) \neq data$ occurs on a VLL in column j with $1 \leq j \leq l\langle m\rangle + 1$[2].

$$\forall 1 \leq j \leq l\langle m\rangle + 1 : \sum_{\substack{p \in (Pi\langle N_o\rangle \cup Po\langle N_o\rangle) \\ \wedge ptype\langle N_o\rangle(p) \neq data}} l_{p,j} \leq 10 \tag{7.3}$$

---

[2] The term "+1" is explained in Section 2.7.4.

The following constraints will establish connectivity between control signals and their source or sink units. As sketched in Section 2.7.4, a control line routed in vertical channel column n can reach units on both sides of the channel (in columns n and $n-1$). This relation will be expressed using intermediate variables: $cR_{v,m,p,j}$ is 1 iff the unit $v \in In\langle m\rangle$, placed in column j with $1 \leq j \leq l\langle m\rangle$, could connect to the primary control port $p \in (Pi\langle N_o\rangle \cup Po\langle N_o\rangle) \wedge ptype(p) \neq data$ using a VLL in column $j+1$ (the VLL is located right of the unit).

$$\begin{aligned} x_{v,j,m} + l_{p,j+1} - 2 \cdot cR_{v,m,p,j} &\leq 1 \\ x_{v,j,m} + l_{p,j+1} - 2 \cdot cR_{v,m,p,j} &\geq 0 \end{aligned} \tag{7.4}$$

Analogously, the intermediate variable $cL_{v,m,p,j}$ is 1 iff a VLL lying left of v could provide the required connectivity.

$$\begin{aligned} x_{v,j,m} + l_{p,j} - 2 \cdot cL_{v,m,p,j} &\leq 1 \\ x_{v,j,m} + l_{p,j} - 2 \cdot cL_{v,m,p,j} &\geq 0 \end{aligned} \tag{7.5}$$

Combining Constraints 7.4 and 7.5, we are able to constrain a new intermediate variable $c_{v,m,p,j}$ such that it is 1 iff $v \in In\langle m\rangle$ placed in column j could be connected to $p \in (Pi\langle N_o\rangle \cup Po\langle N_o\rangle) \wedge ptype(p) \neq data$ from the left or right side (or both sides).

$$\begin{aligned} cL_{v,m,p,j} + cR_{v,m,p,j} - 2 \cdot c_{v,m,p,j} &\leq 0 \\ cL_{v,m,p,j} + cR_{v,m,p,j} - 2 \cdot c_{v,m,p,j} &\geq -1 \end{aligned} \tag{7.6}$$

For each required connection between a unit $v \in In\langle m\rangle$, and a control port $p \in (Pi\langle N_o\rangle \cup Po\langle N_o\rangle)$, we can now enforce that at whatever column j the unit v is placed, at least one way of connecting v to p must exist.

$$\sum_{j=1}^{l\langle m\rangle} c_{v,m,p,j} \geq 1 \tag{7.7}$$

We will apply these constraints once for

1. each primary control input port p of N_o,
2. each primary input port i in the master m corresponding to the input terminal (w, q) in the instances w of m through which a w is connected to p
3. each of the units (=4-LUTs) $v \in In\langle m\rangle$ connected to i (and thus p) by $v \in Fo(i)$,

4. all of the columns j inside the placement area.

Each relevant (=with instances connected to p) master is constrained only once per connected terminal, even though the master may occur with multiple instances in N_o.

When formulating the constraints, we have to cross a hierarchy and a regularity level: The primary control input ports are located at the N_o-level (nodes are optimized slices), while the units we are trying to place are located at the omS-level (nodes are LUTs). The somewhat obscure Item 2 makes the leap from a terminal (w, q) on an instance (=optimized slice) w in N_o to the primary input in its optimized master-slice m by first using *port* to determine the primary input in w underlying the terminal (Definition 35), and then computing the corresponding primary input in the master using $corr_V$ (Definition 26).

Algorithm 14: Generating constraints for input control signal VLL routing

```
for all $p \in Pi\langle N_o\rangle \wedge ptype(p) \neq data$ do
    $Q \leftarrow \{(mr\langle master(w)\rangle, corr_V(port\langle N_o\rangle(w)(q))) \mid$
                $\exists\, q, r, w :\ ((p, r), (w, q)) \in Ttn\langle N_o\rangle\}$
    for all $(m, i) \in Q$ do {omS, primary control input pairs}
        for all $v \in Fo(i)$ do
            for all $1 \leq j \leq l\langle m\rangle$ do
                enforce Constraint 7.4 for $v, m, p, j$
                enforce Constraint 7.5 for $v, m, p, j$
                enforce Constraint 7.6 for $v, m, p, j$
            end for
            enforce Constraint 7.7 for $v, m, p$
        end for
    end for
end for
```

Example 104 Algorithm 14 is sufficiently complex to deserve further examination. An optimized network skeleton N_o is shown at the left side of Figure 7.2. Some primary input nodes are labeled within the rounded rectangles, bold labels outside the rectangles are variables (symbol and significance).

N_o has $In\langle N_o\rangle = \{\mathsf{oS}_0, \mathsf{oS}_1, \mathsf{oS}_2\}$, and $\mathsf{g} \in Pi\langle N_o\rangle$, with $ptype(\mathsf{g}) = control$. Consider the TTNs

$$\{((\mathsf{g}, \mathbf{g}), (\mathsf{oS}_0, \mathbf{a}_0)), ((\mathsf{g}, \mathbf{g}), (\mathsf{oS}_1, \mathbf{a}_1)), ((\mathsf{g}, \mathbf{g}), (\mathsf{oS}_2, \mathbf{a}_2)),$$
$$((\mathsf{g}, \mathbf{g}), (\mathsf{oS}_0, \mathbf{b}_0)), ((\mathsf{g}, \mathbf{g}), (\mathsf{oS}_1, \mathbf{b}_1)), ((\mathsf{g}, \mathbf{g}), (\mathsf{oS}_2, \mathbf{b}_2))\}.$$

The outer loop of Constraint 14 will set p to g. The set Q iterated over by the next nested loop is constructed as follows: Consider the first TTN, it will have $p = \mathsf{g}$, $r = \mathbf{g}$, $w = \mathsf{oS}_0$, and $q = \mathbf{a}_0$. Thus, with $mr\langle master(\mathsf{oS}_0)\rangle = \mathsf{A}$, $port\langle N_o\rangle(\mathsf{oS}_0)(\mathbf{a}_0) = \mathsf{y}$, and $corr_V(\mathsf{y}) = \mathsf{e}$, (m, i) becomes (A, e). Note that this same value will result when

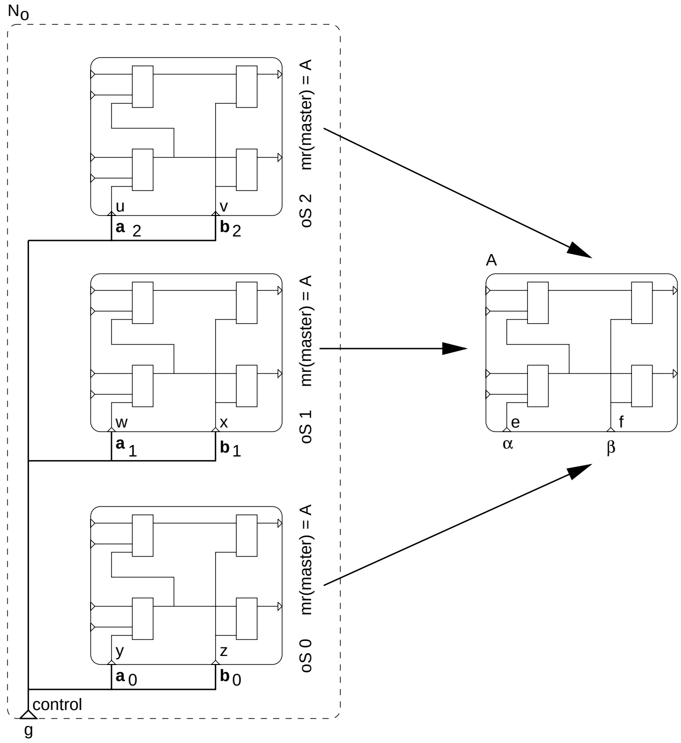


Figure 7.2: Control signal connectivity via VLL

processing the second and third TTNs (all end up at the primary input $\mathsf{e} \in Pi\langle \mathsf{A}\rangle$). The fourth through sixth TTNs, however, will evaluate (m, i) to (A, f).

In this manner, the six TTNs on the optimized slices will only cause the generation of two constraint sets, consisting of Constraints 7.4 through 7.7, for their optimized master-slice.

Analogously, for primary control outputs, we define Algorithm 15.

Algorithm 15: Generating constraints for output control signal VLL routing

```
for all $p \in Po\langle N_o\rangle \wedge ptype(p) \neq data$ do
  $\exists\, q, r, w : ((w, q), (p, r)) \in Ttn\langle N_o\rangle$ {PO p only has single fanin}
  $m \leftarrow mr\langle master(w)\rangle$
  $o \leftarrow corr_V(port\langle N_o\rangle(w)(q))$
  $\exists\, v : v \in Fi(o)$ {PO $o$ has only single fanin}
  for all $1 \leq j \leq l\langle m\rangle$ do
    enforce Constraint 7.4 for $v, m, p, j$
    enforce Constraint 7.5 for $v, m, p, j$
    enforce Constraint 7.6 for $v, m, p, j$
  end for
  enforce Constraint 7.7 for $v, m, p$
end for
```

The differences between Algorithms 14 and 15 are caused by the fact that primary inputs may have multiple fanouts, while primary outputs always have a single fanin each. Thus, most of the loops reduce to a single iteration for primary outputs.

7.2.4 Critical Path Segment Delay Computation

Using the delay-tracing and backannotation process outlined in Section 2.7.2, we have obtained a set of critical paths $\mathfrak{P}(m)$ for each optimized master-slice $m \in \mathfrak{M}_o(N_f)$. Each critical path begins at a primary input of the optimized master-slice, then runs through a number of units, and ends at a primary output. It is described by the sequence $(a, v_1, \ldots, v_n, y)$, where $a \in Pi\langle m\rangle$, $v_k \in In\langle m\rangle$, $y \in Po\langle m\rangle$, for $1 \leq k \leq n$.

Fixing Horizontal Port Locations

Since the critical path delay in our model is primarily based on the distance between two nodes in the critical path, we need to consider the node locations. For units, they will be determined by the ILP model, e.g., by Constraint 7.1. For output ports on optimized slices that connect to primary ports of the entire optimized network skeleton, their horizontal locations can be fixed in advance by considering the compacted subdatapath in context of the entire datapath: When an instance of m has a connection to the left side of the subdatapath[3] through a port $p \in (Pi\langle m\rangle \cup Po\langle m\rangle)$, p is assumed to have $x\langle loc\langle m\rangle(p)\rangle = 0$ (it is located at the left edge of the placement area). Analogously, if connections from instances to the right edge of the datapath exist through p, it will be placed at the right edge of the placement area with $x\langle loc\langle m\rangle(p)\rangle = l\langle m\rangle + 1$. Primary ports that are only connected within the subdatapath (inter-oS connections) are left floating, with $x\langle loc\langle m\rangle(p)\rangle = \perp$. The special case of a port being connected to both sides is handled by duplicating the port with its connectivity, and locking one copy to the left and the other to the right side. For brevity, we assume this is handled transparently (as implemented), and will not complicate the constraint formulations.

To improve clarity, we will show the formulation of the following constraints as an ILP (not restricted to 0-1) whenever feasible. Remember that any ILP can be transformed into a 0-1 ILP by an appropriate binary encoding of variables [Will93].

Assume that we wish to calculate the length of the critical path segment (u, v), with $u, v \in V\langle m\rangle$, and (u, v) a subsequence of $P \in gP$. We have to handle the following cases[4]:

1. If one or both nodes are in $In\langle m\rangle$, their locations are determined by the 0-1 x variables, e.g. $x_{v,j,m}$ (Constraint 7.1).
2. If one of the nodes is a primary port $p \in (Pi\langle m\rangle \cup Po\langle m\rangle)$ that occurs on instances in $In\langle N_o\rangle$ which are connected to subdatapath primary ports

---

[3] The horizontal topological placement at the datapath level is still intact, Section 6.1

[4] The current SDI implementation can handle even more complicated cases crossing omS boundaries. They will be omitted here, since the vertical alignment constraints discussed in the next section provide a similar functionality.

$Pi\langle N_o\rangle \cup Po\langle N_o\rangle$, the horizontal location $x\langle loc(p)\rangle$ is constant, and has been fixed depending on subdatapath context.

3. If a node is a primary control port, it is disregarded here (it was already handled during control routing).

To determine the distance between two units $u, v \in In\langle m\rangle$, we model $|x\langle loc(u)\rangle - x\langle loc(v)\rangle|$ with two intermediate integer variables. If $d = x\langle loc(u)\rangle - x\langle loc(v)\rangle \geq 0$, then $dp_{u,v,m} = d$, otherwise $dp_{u,v,m} = 0$. If $d < 0$, $dn_{u,v,m} = -d$, otherwise $dn_{u,v,m} = 0$. In this manner, the absolute value is expressed as $dp_{u,v,m} + dp_{u,v,m}$.

If both nodes are units, this results in the constraint

$$(\sum_{j=1}^{l\langle m\rangle} j \cdot (x_{u,j,m} - x_{v,j,m})) - dp_{u,v,m} + dn_{u,v,m} = 0. \tag{7.8}$$

If $u \in Pi\langle m\rangle \wedge x\langle loc(u)\rangle \neq \perp$, we enforce

$$(\sum_{j=1}^{l\langle m\rangle} -j \cdot x_{v,j,m}) - dp_{u,v,m} + dn_{u,v,m} = -x\langle loc(u)\rangle. \tag{7.9}$$

If $v \in Po\langle m\rangle \wedge x\langle loc(v)\rangle \neq \perp$, the constraint becomes

$$(\sum_{j=1}^{l\langle m\rangle} j \cdot x_{u,j,m}) - dp_{u,v,m} + dn_{u,v,m} = x\langle loc(v)\rangle. \tag{7.10}$$

These are the valid node combinations, all others will be avoided in higher-level constraints (Section 7.2.5).

We accept partially inconsistent variable values in the model, caused by the separate representation of positive and negative components of distance (an absolute value). E.g., $5 - dp + dn = 0$ could erroneously result in $dp = 10, dn = 5$. However, since we will be minimizing the maximal distance computed as $dp + dn$, those terms contributing to the maximum will be constrained implicitly. The example would have $dp = 5, dn = 0$ in order to minimize $dp + dn$.

The next constraint is employed to compensate for inaccuracies due to the two-phase nature of microplacement (first horizontally, then vertically) . We introduce an 0-1 indicator variable $e_{u,v,m}$ that becomes 1 if both nodes u, v are placed in the same column ($dp_{u,v,m} + dn_{u,v,m} = 0$).

$$dp_{u,v,m} + dn_{u,v,m} + e_{u,v,m} \geq 1 \tag{7.11}$$

If $e_{u,v,m} = 1$, we will add an estimate of the vertical distance within the column (Section 2.7.4). Otherwise, the horizontal model would attempt to optimize delay by placing all nodes on critical paths in the same column (which would appear to give zero delay).

At this point, we can generate the required constraints for all critical path segments in Algorithm 16. Note that we create only one set of constraints for each segment, regardless how often it appears in different critical paths.

Algorithm 16: Generating constraints for all segments on critical paths

```
$done[*] \leftarrow FALSE$
for all $m \in \mathfrak{M}_o$ do
  for all $P \in \mathfrak{P}(m)$ do
    for all $(u, v) \in P$ do
      if not $done[(u, v)]$ then
        enforce appropriate one of Constraints 7.8, 7.9, 7.10 for $u, v, m$
        enforce Constraint 7.11 for $u, v, m$
        $done[(u, v)] \leftarrow TRUE$
      end if
    end for
  end for
end for
```

7.2.5 Maximal Critical Path Delay

After determining the individual segment delays by Algorithm 16, we proceed to compute the delays of entire critical paths (including the estimates for vertical distance, Constraint 7.11), and determine their maximal delay in the integer variable dm. While doing so, we skip all path segments (u, v) that either contain a control port, or an unplaced port. The first case was handled in the control routing constraints, the second case will be described during vertical inter-slice alignment[5].

$$\begin{aligned} \textsc{ValidSeg}(u, v) \leftarrow & \\ & (u \notin Pi\langle m\rangle \vee (ptype(u) = data \wedge x\langle loc(u)\rangle \neq \perp)) \\ & \wedge (v \notin Po\langle m\rangle \vee (ptype(v) = data \wedge x\langle loc(v)\rangle \neq \perp))) \end{aligned} \tag{7.12}$$

The helper function ValidSeg evaluates to *TRUE* if the segment (u, v) should be included in the path delay computation.

$$\forall m \in \mathfrak{M}_o \forall P \in \mathfrak{P}(m) : \; dm - \sum_{\substack{(u,v)\in P \\ \wedge \textsc{ValidSeg}(u,v)}} (dp_{u,v,m} + dn_{u,v,m} + \lfloor h\langle m\rangle/4\rfloor \cdot e_{u,v,m}) \geq 0 \tag{7.13}$$

Note how the estimated vertical distance adds another unit of distance per four cells of height in the placement area if the two nodes in the segment were placed in the same column ($e_{u,v,m} = 1$).

---

[5] The implementation contains special code to handle unplaced ports by tracing connectivity to the next placed element. An in-depth description, however, would only detract from the keypoints of model composition

7.2.6 Vertical Inter-omS Alignment

The only inter-omS constraints defined up to now concern the routing of control signals from primary ports of N_o to the optimized slices on vertical long lines. In addition, however, it is beneficial to also consider inter-oS signals. Examples for these signals include the bit propagation in shifter-like structures, or the carry signal in a ripple-carry adder. As shown in Figure 2.31, these signals should be routed by abutment (sink and source cells in same column) if possible, or at least with only minimal horizontal offset.

To this end, we will consider placement for

1. Floating primary ports p of the optimized network skeleton ($x\langle loc(p)\rangle = \perp$).
2. Units in the optimized master-slices.

Note the omission of primary ports at the omS-level: They become irrelevant when the entire compacted subdatapath is assembled from oSs. We determine placement only for those components that actually appear at that level. Similar to the approach used for control signal routing (Figure 7.2), we analyze connectivity in the optimized network skeleton N_o, and isomorphically map it to the optimized master-slices. In this manner, regularity is employed to constrain only the master instead of each separate instance.

Before formulating the constraints, we need to determine which circuit elements need alignment:

1. A floating primary input port of N_o connected to an optimized slice will be aligned with the fanout units of the corresponding master primary input port.
2. A connection between two optimized slices will align the fanin unit of the corresponding master primary output port with all fanout units of the corresponding master primary input port. Note that the omS of both oSs may be identical.
3. A floating primary output port of N_o connected to an optimized slice will be aligned with the fanin unit of the corresponding master primary output port.

Example 105 In Figure 7.3, the floating primary input port $\mathsf{o} \in Pi\langle N_o\rangle$ is connected via TTN $((\mathsf{o}, \mathbf{o}), (\mathsf{A}, \mathbf{a}))$ to the optimized slice A. The primary port inside this instance corresponding to terminal $(\mathsf{A}, \mathbf{a})$ is $port\langle N_o\rangle(\mathsf{A})(\mathbf{a}) = \alpha$.

To exploit regularity, we have to leave the instance view, and look into the master $mr\langle master(\mathsf{A})\rangle = \mathsf{X}$, which specifies the innards of A. We find that the primary port in the master corresponding to α in the instance is $corr_V(\alpha) = \mathsf{p} \in Pi\langle \mathsf{X}\rangle$. There, p connects to the two units $Fo(\mathsf{p}) = \{\mathsf{u}, \mathsf{v}\}$. Thus, we will want to make sure that the horizontal offset between o and u, as well as between o and v, is as small as possible.

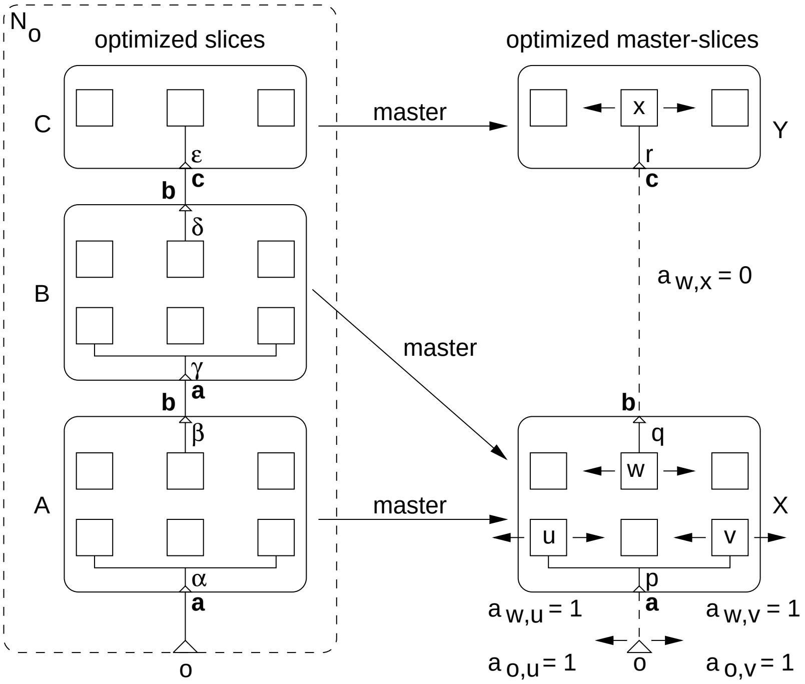


Figure 7.3: Vertical inter-optimized master-slice alignment

The next TTN considered in N_o is the inter-oS TTN $((\mathsf{A}, \mathbf{b}), (\mathsf{B}, \mathbf{a}))$. In the oSs, we determine the primary ports corresponding to the source and sink terminals as $port\langle N_o\rangle(\mathsf{A})(\mathbf{b}) = \beta$, and $port\langle N_o\rangle(\mathsf{B})(\mathbf{a}) = \gamma$. Changing to the master $mr\langle master(\mathsf{A})\rangle = \mathsf{X}$ and $mr\langle master(\mathsf{B})\rangle = \mathsf{X}$, we determine the corresponding primary ports as $corr_V(\beta) = \mathsf{q}$, and $corr_V(\gamma) = \mathsf{p}$. We need to align the fanin unit of the primary output q, $Fi(\mathsf{q}) = \mathsf{w}$, with the fanouts of the primary inputs p, $Fo(\mathsf{p}) = \{\mathsf{u}, \mathsf{v}\}$. Thus, the horizontal offset should be minimized between units w and u, and w and v.

Finally, the inter-oS TTN $((\mathsf{B}, \mathbf{b}), (\mathsf{C}, \mathbf{c}))$ is analyzed. Following our previous approach, we determine the primary ports in the instances as $port\langle N_o\rangle(\mathsf{B})(\mathbf{a}) = \delta$ and $port\langle N_o\rangle(\mathsf{C})(\mathbf{c}) = \epsilon$. The optimized master-slices are found as $mr\langle master(\mathsf{B})\rangle = \mathsf{X}$, $mr\langle master(\mathsf{C})\rangle = \mathsf{Y}$, with the primary ports in the masters being $corr_V(\delta) = \mathsf{q}$ and $corr_V(\epsilon) = \mathsf{r}$. Following the connectivity in the masters, we find $Fi(\mathsf{q}) = \mathsf{w}$ and $Fo(\mathsf{r}) = \{\mathsf{x}\}$. Thus, the resulting alignment should minimize the offset between the units w and x.

The set A, consisting of all node/master (floating primary port and unit) pairs to be aligned, is computed by the formula in Figure 7.4.

The first subset collects all primary inputs of N_o that have to be aligned with units in the omS, the second subset contains all units in omS that have to be aligned with each other, and the third subset holds all units in omS that have to be aligned with primary outputs of N_o.

For primary ports at the N_o-level, we use N_o as the master. This substitution is valid since the constraint formulation will only use the network skeleton-components. Note how the set-nature of A automatically removes duplicates, ensuring that each alignment will be expressed only once.

Following our previous approach of encoding the absolute value in separate

$$
\begin{aligned}
&A = \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (7.14)\\
&\quad \{((u, N_o), (v, mr\langle master(v'')\rangle)) \mid\\
&\quad \exists\, p, q :\ ((u, p), (v'', q)) \in \mathit{Ttn}\langle N_o\rangle \wedge u \in \mathit{Pi}\langle N_o\rangle \wedge x\langle loc\rangle(u) = \bot\\
&\quad \wedge \exists\, v' :\ v' = corr_V(port\langle N_o\rangle(v'')(q))\\
&\quad \wedge \exists\, v :\ v \in \mathit{Fo}(v')\}\\
&\cup\{((u, mr\langle master(u'')\rangle), (v, mr\langle master(v'')\rangle)) \mid\\
&\quad \exists\, p, q :\ ((u'', p), (v'', q)) \in \mathit{Ttn}\langle N_o\rangle \wedge u, v \in \mathit{In}\langle N_o\rangle\\
&\quad \wedge \exists\, u', v' :\ u' = corr_V(port\langle N_o\rangle(u'')(p)) \wedge v' = corr_V(port\langle N_o\rangle)(v'')(q)\\
&\quad \wedge \exists\, u, v :\ u = \mathit{Fi}(u') \wedge v \in \mathit{Fo}(v')\}\\
&\cup\{((u, mr\langle master(u'')\rangle), (v, N_o)) \mid\\
&\quad \exists\, p, q :\ ((u'', p), (v, q)) \in \mathit{Ttn}\langle N_o\rangle \wedge v \in \mathit{Po}\langle N_o\rangle \wedge x\langle loc\rangle(v) = \bot\\
&\quad \wedge \exists\, u' :\ u' = corr_V(port\langle N_o\rangle(u'')(p))\\
&\quad \wedge \exists\, u :\ u = \mathit{Fi}(u')\}.
\end{aligned}
$$

Figure 7.4: Computing the set of nodes to align

positive and negative components, the computation of the alignment offset reduces to

$$\forall((u, m), (v, n)) \in A :\ (\sum_{j=1}^{\max(l\langle m\rangle, l\langle n\rangle)} j\,(x_{u,j,m} - x_{v,j,m})) - ap_{u,m,v,n} + an_{u,m,v,n} = 0 \qquad (7.15)$$

Note that the solution variables $x_{v,j,m}$ now encompass the placement of floating ports (which occur in A). We compute the maximum alignment error am as

$$\forall((u, m), (v, n)) \in A :\ am - ap_{u,m,v,n} - an_{u,m,v,n} \geq 0 \qquad (7.16)$$

7.2.7 Objective Function

At this stage, we have described all constraints required to formulate the objective function expressing our placement aims (Section 2.7.4).

$$\textbf{minimize}\ w_c \cdot (\sum_{\substack{p \in (\mathit{Pi}\langle N_o\rangle \cup \mathit{Po}\langle N_o\rangle) \\ \wedge\, ptype(p) \neq data}} \sum_{j=1}^{l\langle N_o\rangle} l_{p,j}) + w_d \cdot dm + w_a \cdot am \qquad (7.17)$$

The first term is the total number of VLLs used for control routing (including duplicated signals, Constraint 7.3), the second term is the delay of the slowest optimized master-slice (Constraint 7.13, note: *not* the critical path delay of the entire subdatapath), and the third term is the maximal alignment error for vertical inter-slice connections (Constraint 7.16).

$w_c, w_d, w_a \in \mathbb{R}$ are user-defined weights. They could be used, e.g., to favor a faster circuit over one using fewer VLLs. For the benchmark circuits, the terms were of the same magnitude, and all weights were left at default values of 1.0.

7.3 Efficiently Solving 0-1 ILPs

7.3.1 Preprocessing and Constructive Enumeration

To efficiently solve larger 0-1 ILPs, we use a three-step procedure: In the first phase, the model is simplified using techniques specific to pseudo-boolean optimization problems (e.g., literal fixing, inter-literal equations, coefficient reduction, etc. [Bart96]). Next, a constructive enumeration (Davis-Putnam for constraint logic programs [Bart96] [Bart95]) quickly generates feasible solutions for the model. After obtaining the first feasible solution, the procedure artificially decreases the current value of the objective function to create an additional constraint for a next iteration. If yet another feasible solution (with the lowered objective value) is found, the process continues. The first and second phases are realized using the `opbdp` solver [Bart95].

In this manner, we obtain solutions with a steadily improving quality. In some cases, the constructive approach alone reaches a provably optimal solution in a reasonable amount of time, which is then used directly for the second (vertical) phase of microplacement (Section 7.5).

7.3.2 Pruned Branch-and-Bound

In other cases, however, the nature of the model (after simplification) is unsuitable for a fully constructive optimization, and the time and memory requirements of each improvement become impractical. After exceeding user defined limits for these quantities, the constructive approach is aborted.

As the last attempt at ILP solving, we then switch to the conventional branch-and-bound based solver `cplex` [Cple94] to process the simplified model (obtained in the first phase). In addition, we use the objective function value of the best feasible solution found using the constructive approach as an initial upper cutoff to prune the solution tree of the branch-and-bound algorithm: All branches that have a best possible solution (for the LP relaxation of the problem) worse than the predetermined upper cutoff will not be explored further (reducing time and memory requirements).

Further speedups are achieved by also generating specially-ordered-sets [Will93] [Cple94] for the $x_{v,j,m}$ solution variables.

7.3.3 Capabilities and Limitations

By employing this hybrid strategy consisting of both enumeration and branch-and-bound techniques, we exploit that some 0-1 ILPs which are very difficult to solve by one approach, may be quite easy for the other one [Bart95]. The current implementation solves 0-1 ILPs of, e.g., 1128 constraints and 981 0-1 variables (a 32-bit 74181-based ALU) in less than 15 minutes on a Sun SparcStation 20/71 (64MB RAM). Even further speedups could be realized by extracting the longest common subpaths from the critical paths, and then computing the delay along an entire subpath only once.

However, the ILPs resulting from some circuits are not optimally solvable with reasonable time and memory limits, even using the hybrid strategy. Furthermore, since the exact solution of 0-1 ILPs is a problem with NP-complexity, computation times and memory requirements will, despite all preprocessing and optimization efforts, eventually skyrocket with increasing problem size. To make SDI usable even for larger circuits, we have implemented a heuristic alternative to the exact ILP model.

7.4 Heuristic for Horizontal Geometrical Node Placement

7.4.1 Ensemble-Based Annealing

In order to increase the problem size processable by SDI, we have provided an alternative heuristic algorithm to the 0-1 ILP. It is based on ensemble-based simulated annealing [RuPS91], which calculates multiple solutions in parallel. Statistics collected across these *domain elements* are then used to dynamically adapt annealing parameters during the optimization process.

The model processed by the heuristic has the same optimization aims as the 0-1 ILP (Section 2.7.4). A domain element consists of the complete cell placement for an entire compacted subdatapath (across all optimized master-slices). Each domain element is initialized to a random cell placement. To ensure a smooth solution landscape, moves are currently limited to simple pairwise exchanges of two locations (units or empty cells) within the placement area of each optimized master-slice.

7.4.2 Optimization Cost Function

Each placement is then evaluated in terms of delay and number of control lines needed for routing. The dedicated alignment term of the 0-1 ILP model (Section 7.2.6) has been replaced by directly considering the horizontal distance for inter-omS connections in the critical path delay computations. The cost function thus becomes

$$\textbf{minimize} \ \ w_d \cdot dm + w_c \cdot el,$$

where $w_d, w_c \in \mathbb{R}$ are user-defined weights, dm is the slowest critical path delay through any optimized master-slice, and el is the excess number of VLLs for control routing. el is computed as the difference between the number of VLLs actually used, and the number of different control signals in the entire subdatapath under compaction. An optimal solution will have $el = 0$ (each control signal is only routed on a single VLL each), with the optimization then becoming purely timing-driven.

Apart from the annealing parameters, the speed of an annealing-based approach is heavily influenced by the efficiency of the cost function, which must be evaluated after every move. Since the computation of all critical path delays for a large number of paths with many segments, in addition to control routing, would take an inordinate amount of time, we have implemented an incremental algorithm: We compute only the changes in segment delay caused by each exchange, and then selectively update the path delays only for those paths that actually contain the changed segment. A complex data structure cross-referencing all nodes, segments and paths is built at initialization time, and reduces the determination of the circuit components affected by a move to simple lookups.

The core of the annealing algorithm itself based on the `EBSA 2.1` library [Frost93], which had to be heavily modified to allow for the fast incremental cost computation.

7.4.3 Capabilities and Limitations

As a result of the powerful dynamic parameter adaptation by ensemble statistics (which avoids the usual problems of hardcoded annealing schedules [RuPS91]), and the fast cost evaluation, the heuristic lifts the problem size restrictions imposed by ILPs, and avoids the case of ILPs with a structure not amenable to either solver. E.g., a sample 0-1 ILP problem with 1200 constraints and 930 variables could only be processed in 7300s by the combined solvers. Using annealing, the placement problem was solved in 600s. Furthermore, for all models that were exactly solvable at all, the heuristic also found a provably optimal solution. Our current experience suggests that the annealing-based placer is not subject to any limitations when used in context of SDI[6].

7.5 ILP for Vertical Geometrical Node Placement

Analogously to Section 7.2, we proceed to formulate constraints for the vertical geometrical placement model described in Section 2.7.4. With the problem size reduced to a single optimized master-slice, we can afford here to process a far more detailed model exactly, instead of resorting to a heuristical solution. The second placement phase finally assigns the units to CLBs. We thus have

---

[6] The restricted size of subdatapaths under compaction, and the exploitation of regularity leads to problem sizes of dozens of nodes, instead of millions of gates.

to distinguish between two different measurements for distance and coordinates: cells and CLBs. This was not required for horizontal placement, since horizontally, each CLB contains only a single cell. Vertically, however, a CLB holds two cells (Figure 2.32).

7.5.1 Node Placement

Given an optimized master-slice $m \in \mathfrak{M}_o$ to be microplaced, the 0-1 solution variable $y_{v,i}$ is 1 iff the node $v \in In\langle m\rangle$ is being placed in cell row $1 \leq i \leq h\langle m\rangle$ of the placement area.

First, we want to ensure that each node v is placed in exactly one row i.

$$\forall\, v \in In\langle m\rangle : \sum_{i=1}^{h\langle m\rangle} x_{v,i} = 1 \tag{7.18}$$

The second set of constraints of the ILP ensures that each row i is used at most once by all units previously placed in a column j.

$$\forall\, 1 \leq j \leq l\langle m\rangle \,\forall\, 1 \leq i \leq h\langle m\rangle : \sum_{\substack{v \in In\langle m\rangle \\ \wedge x\langle loc(v)\rangle = j}} y_{v,i} \leq 1 \tag{7.19}$$

7.5.2 Vertical Distance in CLBs

When determining the vertical distance in CLBs between two connected nodes u, v, we just halve the distance in cells. The absolute value of the CLB distance will be expressed as the intermediate variables $dcp_{u,v}$ for the positive part, and $dcn_{u,v}$ for the negative part.

Fixing Vertical Port Locations

At this stage, the locations of all omS primary ports are fixed, there are no floating ports left: The vertical location of ports connecting the subdatapath with the entire datapath is fixed during floorplanning. It depends, e.g., on the location of ports on adjacent hard-macros, or on the assignment of signals to chip-level pads.

Ports used as terminals for inter-oS connections within the subdatapath are locked to the top or bottom sides of the omS placement area. If an oS has a connection to an oS located above it (Section 7.1), we lock the port to the top of the omS (at CLB position $\lceil h\langle m\rangle/2\rceil$ for primary inputs, or $\lceil h\langle m\rangle/2\rceil + 1$ for primary outputs). If a connection to a lower oS exists, we lock the port to the bottom of the master of the originating oS (at CLB position 1 for primary inputs, and position 0 for primary outputs). The differentiation between primary inputs and outputs reflects the longer “reach” of output pins beyond the placement area itself (Figure 2.32). We thus model the delay of oS-external connections at their source terminals, and assume zero delay at the inputs. If an oS is connected to oSs both above and below it, we transparently duplicate the primary port with its connectivity, and lock one copy to the top, the other one to the bottom of the omS (as in Section 7.2.4).

For clarity of modeling, we assume the existence of two helper functions. For a port $p \in (Pi\langle m\rangle \cup Po\langle m\rangle)$, $\textsc{Top}(p)$ is 1 if p is top-locked, 0 otherwise. Analogously, $\textsc{Bot}(p)$ is 1 if p is bottom-locked, 0 otherwise.

The computations for the vertical CLB distance between two nodes $u, v \in V\langle m\rangle$ depend on the nature of u, v. For $u \in Pi\langle m\rangle$, we formulate

$$-(\sum_{i=1}^{\lceil h\langle m\rangle/2\rceil} i \cdot (y_{v,2\cdot i-1} + y_{v,2\cdot i})) - dcp_{u,v} + dcn_{u,v} = -\textsc{Top}(u) \cdot \lceil h\langle m\rangle/2\rceil - \textsc{Bot}(u) \cdot 1 \tag{7.20}$$

Note that the application of the helper functions occurs statically at constraint-generation time. The resulting constraint will consist only of purely linear terms and contains just the $y_{v,i}$, $dcp_{u,v}$, $dcn_{u,v}$ variables at solution time.

If both $u, v \in In\langle m\rangle$, the distance computation constraint becomes

$$(\sum_{i=1}^{\lceil h\langle m\rangle/2\rceil} i \cdot (y_{u,2\cdot i-1} - y_{v,2\cdot i-1} + y_{u,2\cdot i} - y_{v,2\cdot i})) - dcp_{u,v} + dcn_{u,v} = 0. \tag{7.21}$$

For $v \in Po\langle m\rangle$, we use

$$(\sum_{i=1}^{\lceil h\langle m\rangle/2\rceil} i \cdot (y_{u,2\cdot i-1} + y_{u,2\cdot i})) - dcp_{u,v} + dcn_{u,v} = \textsc{Top}(u) \cdot (1 + \lceil h\langle m\rangle/2\rceil) - \textsc{Bot}(u) \cdot 0 \tag{7.22}$$

Note the modeling of locked port locations depending on input/output characteristics and top/bottom locking.

The actual implementation generates two separate inequalities for positive and negative distance components, instead of the larger combined equality, to improve solving efficiency.

7.5.3 Recognizing Linear Horizontal Placement

As shown in Section 2.7.4 and Figure 2.32 (e.g., (A, E) and (B, E)), we need to recognize the special case of two connected nodes being placed in the same CLB row. To this end, we introduce the intermediate 0-1 variables $or_{u,v}$, which become 1 iff u, v have zero vertical CLB distance.

$$\begin{aligned} dcp_{u,v} + dcn_{u,v} + or_{u,v} &> 0 \\ dcp_{u,v} + dcn_{u,v} + h\langle m\rangle \cdot or_{u,v} &\leq h\langle m\rangle \end{aligned} \tag{7.23}$$

Together, these two constraints express an AND-clause: The first one forces $or_{u,v}$ to 1 if the distance is zero, the second one forces $or_{u,v}$ to 0 if the distance is non-zero.

7.5.4 Recognizing Horizontally Abutting Cells

Another special case are connected cells in horizontally adjacent CLBs in the same row ((A, C), (C, B) in Figure 2.32). By varying the vertical position of the source cell in the CLB (between F and G-LUT), we aim at exploiting asymmetries in the FPGA routing network to achieve zero switch matrix (SM) connectivity. We set intermediate 0-1 variables $nl_{u,v}$ ($nr_{u,v}$) to 1 iff u is directly left of (right of) v, and u is placed in the appropriate LUT for the direction: If u is left of (right of) v, u should be placed in the G-LUT (F-LUT).

First, we define the constraint if u lies left of v (remember that all horizontal locations are known by now).

$$\left(\sum_{n=1}^{\lfloor h\langle m\rangle/2\rfloor} y_{u,2\cdot n}\right) - dcp_{u,v} - dcn_{u,v} - (h\langle m\rangle + 1)\cdot nl_{u,v} \geq -h\langle m\rangle \tag{7.24}$$

The first sum term becomes 1 if u was placed in any G-LUT (vertical cell coordinates 2, 4, 6, ...). As long as only this term is 1, $nl_{u,v}$ may become 1 with the right-hand side of the inequality still holding. If u is not placed in a G-LUT (the sum term is 0), or either of the distance components becomes > 0, $nl_{u,v}$ cannot be 1 without violating the inequality. We define the constraint for the u right of v analogously.

$$\left(\sum_{n=1}^{\lceil h\langle m\rangle/2\rceil} y_{u,2\cdot n-1}\right) - dcp_{u,v} - dcn_{u,v} - (h\langle m\rangle + 1)\cdot nr_{u,v} \geq -h\langle m\rangle \tag{7.25}$$

Here, the first sum term only becomes 1 if u was placed in any F-LUT (vertical cell coordinates 1, 3, 5, ...).

7.5.5 Recognizing Vertically Abutting Cells

As with horizontal abutments (Section 7.5.4), we need to handle special cases with connected cells in vertically adjacent CLBs in the same column ((A, D), (B, D) in Figure 2.32). The intermediate 0-1 variables $na_{u,v}$ ($nb_{u,v}$) will become 1 iff u is in the CLB directly above (below) v, and placed in the F-LUT (G-LUT).

The vertical CLB distance is only computed at solution time (Section 7.5.2), not constant as in Section 7.5.4. Thus, we have to perform the test if the vertical distance is exactly 1 *within* the constraint. To this end, we will exploit the binary representation of distances (which was disregarded up to this point, Section 7.2.4). The distances are thus modeled as

$$dcp_{u,v} = \sum_{b=0}^{\lfloor \mathrm{ld}\, h\langle m\rangle\rfloor} 2^b \cdot dcp_{u,v,b},$$

analogously for $dcn_{u,v}$. For the distance to be exactly 1, we are thus imposing the subconstraint

$$(\sum_{b=1}^{\lfloor \mathrm{ld}\, h\langle m\rangle \rfloor} (1 - d_b)) + d_0 = \lfloor \mathrm{ld}\, h\langle m\rangle \rfloor + 1$$

on the individual "bits" of a distance d. We use $dcp_{u,v}$ as d for the relation u above b (the vertical CLB distance is positive), and $dcn_{u,v}$ as d for u below b (negative vertical CLB distance).

With this intention, me may formulate the constraint for u above v as

$$(\sum_{n=1}^{\lceil h\langle m\rangle/2\rceil} y_{u,2\cdot n-1}) + (\sum_{b=1}^{\lfloor \mathrm{ld}\, h\langle m\rangle \rfloor} (1 - dcp_{u,v,b})) + dcp_{u,v,0} - (\lfloor \mathrm{ld}\, h\langle m\rangle \rfloor + 2) \cdot na_{u,v} \geq 0 \tag{7.26}$$

Thus, $na_{u,v}$ may become 1 only if the source unit was placed in an F-LUT (the first sum is 1), and vertical CLB distance is +1 (in $dcp_{u,v}$, the bits $1 \dots \lfloor \mathrm{ld}\, h\langle m\rangle \rfloor$ are 0, and bit 0 is 1). We define the constraint for u below v analogously.

$$(\sum_{n=1}^{\lfloor h\langle m\rangle/2\rfloor} y_{u,2\cdot n}) + (\sum_{b=1}^{\lfloor \mathrm{ld}\, h\langle m\rangle \rfloor} (1 - dcn_{u,v,b})) + dcn_{u,v,0} - (\lfloor \mathrm{ld}\, h\langle m\rangle \rfloor + 2) \cdot nb_{u,v} \geq 0 \tag{7.27}$$

Here, u must be placed in a G-LUT, and the vertical CLB distance from source to sink must be -1 ($dcp_{u,v} = 0, dcn_{u,v} = 1$) for $nb_{u,v}$ to become 1.

We have to add two "guard" constraints to make sure that either $dcp_{u,v} > 0$ or $dcn_{u,v} > 0$. Since the $dcp_{u,v}$ and $dcn_{u,v}$ are not minimized directly (see Objective 7.39), they could float. E.g., $5 - dp + dn = 0$ could erroneously result in $dp = 10, dn = 5$. However, this would produce invalid results for the $na_{u,v}, nb_{u,v}$ calculations: In contrast to all other constraints, Constraints 7.26 and 7.27 don't just minimize a value, they compare it exactly (test if distance is equal to 1). Thus, floating variables (even if their sum is correct) would invalidate the exact comparison.

The 0-1 intermediate guard variables will be $gp_{u,v}$ (1 iff $dcp_{u,v} > 0$) and $gn_{u,v}$ (1 iff $dcn_{u,v} > 0$). Only one of the $gp_{u,v}, gn_{u,v}$ may be 1 for a valid absolute value computation.

$$\begin{aligned} dcp_{u,v} - h\langle m\rangle \cdot gp_{u,v} &\leq 0 \\ dcn_{u,v} - h\langle m\rangle \cdot gn_{u,v} &\leq 0 \\ gp_{u,v} + gn_{u,v} &\leq 1 \end{aligned} \tag{7.28}$$

Note how we compute a value larger than the maximal distance (the CLB height of the placement area) simply by using the height in cells. With these constraints, absolute values will always be computed correctly.

7.5.6 Vertical SM Distance in a Single Column

The following constraint computes the distance in SMs if a source u and sink v of a TTN were placed within the same column by the horizontal microplacement phase. Normally, the distance is the CLB distance, but may become 0 if the source is a unit, and both cells are abutting vertically (Section 7.5.5). The intermediate variable $ds_{u,v}$ will hold the distance in SMs.

$$ds_{u,v} - dcp_{u,v} - dcn_{u,v} + h\langle m\rangle \cdot (na_{u,v} + nb_{u,v}) \geq 0 \tag{7.29}$$

$ds_{u,v}$ thus assumes the value of the absolute CLB distance, or 0 if one of $na_{u,v}, nb_{u,v}$ is 1. If the source wasn't a unit, the optimizations by vertically abutting placement cannot be performed, and the constraint reduces to

$$ds_{u,v} - dcp_{u,v} - dcn_{u,v} \geq 0, \tag{7.30}$$

the simple CLB distance.

7.5.7 Arc-Based Unit-to-Cell Assignment

If the horizontal distance between connected units is ≥ 1, zero SM delay placement is impossible by vertical abutment (Section 7.5.5), possible by horizontal abutment for a distance of 1 (Section 7.5.4), and impossible again for all longer distances.

However, by proper assignment of the source unit to a cell inside a CLB, we can avoid an additional delay penalty (Section 2.7.4, cases (B,H) and (I,H)). The intermediate 0-1 variables $pa_{u,v}$ will be set to 1 iff the source unit is assigned to the cell whose reach covers the proper arc to the sink, or the assignment doesn't matter at all: If the source lies left of the sink, the source should be placed in the G-LUT, except if the sink lies below the source (cell assignment doesn't matter in this case). Analogously, if the source lies right of the sink, the source should be placed in the F-LUT, except if the if the sink lies above the source (no effect on delay).

For a horizontal distance of 1 and u lying left of v, we arrive at the constraint

$$\left(\sum_{n=1}^{\lfloor h\langle m\rangle/2\rfloor} y_{u,2\cdot n}\right) + dcp_{u,v} - pa_{u,v} \geq 0. \tag{7.31}$$

The first sum term becomes 1 if the source unit is placed in any G-LUT, and the second term becomes > 0 if u lies above v (the vertical CLB distance is > 0). $pa_{u,v}$ is thus set to 1 if one (or both) of the conditions are true. We define the constraint for the case of u lying right of v analogously.

$$\left(\sum_{n=1}^{\lceil h\langle m\rangle/2\rceil} y_{u,2\cdot n-1}\right) + dcn_{u,v} - pa_{u,v} \geq 0. \tag{7.32}$$

For horizontal distances > 1, we have to consider the special case of both units being placed in a single row (Section 7.5.3). In this situation, cell assignment also becomes irrelevant (Section 2.7.4, (A,E) and (B,E)), and we force $pa_{u,v}$ to 1. The resulting constraints are

$$\left(\sum_{n=1}^{\lfloor h\langle m\rangle/2\rfloor} y_{u,2\cdot n}\right) + dcp_{u,v} - pa_{u,v} + or_{u,v} \geq 0 \tag{7.33}$$

for the u-left-of-v arrangement, and

$$\left(\sum_{n=1}^{\lceil h\langle m\rangle/2\rceil} y_{u,2\cdot n-1}\right) + dcn_{u,v} - pa_{u,v} + or_{u,v} \geq 0. \tag{7.34}$$

in the u-right-of-v case.

7.5.8 SM Distance in Adjacent Columns

With these constraints, we formulate the calculation of the SM distance when two connected nodes are placed in adjacent columns. The computed distance will consist of both the vertical distance and the horizontal distance. As in Section 7.5.6, we will compute the SM distance in $ds_{u,v}$. If a unit u lies left of v, we use

$$ds_{u,v} - dcp_{u,v} - dcn_{u,v} + pa_{u,v} + h\langle m\rangle \cdot nl_{u,v} \geq 1. \tag{7.35}$$

This constraint has the following effect: Initially, $ds_{u,v}$ will be the absolute value of the CLB distance. However, if the source unit was not assigned to the cell appropriate for this placement arc ($pa_{u,v} = 0$), the SM distance $ds_{u,v}$ will be increased by 1 (due to the right-hand side of the inequality being 1). If, on the other hand, a direct (zero SM) connection by horizontal abutment was possible (indicated by $nl_{u,v} = 1$), $ds_{u,v}$ will become 0. As before, we use $h\langle m\rangle$ as a value guaranteed to be greater than $dcp_{u,v} + dcn_{u,v}$.

The analogous constraint for a unit u lying right of v thus becomes

$$ds_{u,v} - dcp_{u,v} - dcn_{u,v} + pa_{u,v} + h\langle m\rangle \cdot nr_{u,v} \geq 1. \tag{7.36}$$

If u is a primary port, the detailed cell assignment calculations are superfluous, and the separate constraints reduce to a simple

$$ds_{u,v} - dcp_{u,v} - dcn_{u,v} \geq 0. \tag{7.37}$$

7.5.9 Computing Net Delay in SMs

To compute the delay in SMs of an arbitrary TTN $((u, p), (v, q))$, we introduce a utility array HDIST$[(u, v)]$, used to store the horizontal distance of a node pair (u, v) for later use in Section 7.5.11. We then proceed as shown in Algorithm 17.

Algorithm 17: Generating constraints for vertical TTN delay in SMs

```
{compute vertical CLB distance}
enforce one of Constraint 7.20, 7.21, 7.22
          depending on whether $u$ is a PI, unit, or PO.
{do further optimizations if source is a unit}
if $u \in In\langle m\rangle$ then
  {direct vertical connection in same column possible?}
  if $x\langle loc(u)\rangle = x\langle loc(v)\rangle$ then
    enforce Constraints 7.26 and 7.27 {vertical adjacency}
    enforce Constraint 7.29 {vertical SM distance}
    HDIST$[(u, v)] \leftarrow 0$\{$u, v$ in same column\}
  else if $|x\langle loc(u)\rangle - x\langle loc(v)\rangle| = 1$ then {adjacent columns?}
    if $x\langle loc(u)\rangle - x\langle loc(v)\rangle < 0$ then {$u$ left-of $v$}
      {check adjacency and placement arc}
      enforce Constraints 7.24 and 7.31
      {compute hybrid vertical and horizontal SM distance}
      enforce Constraint 7.35
    else {$u$ right-of $v$}
      {check adjacency and placement arc}
      enforce Constraints 7.25 and 7.32
      {compute hybrid vertical and horizontal SM distance}
      enforce Constraint 7.36
    end if
    HDIST$[(u, v)] \leftarrow 0$\{horizontal distance in hybrid distance\}
  else {horizontal distance $> 1$}
    enforce Constraint 7.23 {test for single row-placement}
    {check for proper placement arc with test for single row}
    if $x\langle loc(u)\rangle - x\langle loc(v)\rangle < 0$ then {$u$ left-of $v$?}
      enforce Constraint 7.33
    else {$u$ right-of $v$}
      enforce Constraint 7.34
    end if
    {remember actual horizontal distance}
    HDIST$[(u, v)] \leftarrow |x\langle loc(u)\rangle - x\langle loc(v)\rangle|$
  end if
else {$u$ is a PI, no further optimizations}
  HDIST$[(u, v)] \leftarrow |x\langle loc(u)\rangle - x\langle loc(v)\rangle|$
end if
```

Note that we always calculate the vertical CLB distance. Further optimizations by appropriate cell assignments can only be performed if the source node is a unit. In the first case, the horizontal distance is always zero, and the vertical distance is either 0 or 1 (depending on whether direct interconnections by abutment were possible). In the second case, the horizontal distance between adjacent columns is 1. However, it also might be reduced to 0 by appropriate placement. Thus, the horizontal distance is also computed in the constraints (7.35, 7.36), and set to zero in the utility array. In the last case, we always have a non-zero horizontal distance, which we remember in HDIST. We just avoid even further delays by aiming at proper unit-to-cell assignment. If the source node was a PI, none of the optimizations is possible, we just remember the horizontal distance for later use.

7.5.10 Computing Path Delay in SMs

Using Algorithm 17, we can proceed to generate constraints to compute the SM delay along entire critical paths in $\mathfrak{P}(m)$ (cf. Section 7.2.4).

Algorithm 18: Generating constraints for path delay in SMs

```
$done[*] \leftarrow FALSE$
for all $P \in \mathfrak{P}(m)$ do
  for all $(u, v) \in P$ do
    if not $done[(u, v)]$ then
      enforce Constraints of Algorithm 17 for $u, v$ {segment delay for $u, v$}
      $done[(u, v)] \leftarrow TRUE$
    end if
  end for
end for
```

7.5.11 Computing Maximal Critical Path Delay

Now that we have assembled all constraints to compute the delay of each segment on a critical path, we have to determine the maximal SM delay dsm over all critical paths (Figure 7.5). Since we are using different methods to express horizontal distances, the computation of the maximal delay in this second phase of microplacement is more complicated than in Section 7.2.5.

The first sum term spans the delays of segments with a source unit and possible optimizations by abutting placement (vertically Section 7.5.5 and 7.5.6, and horizontally Section 7.5.4 and 7.5.8). The second sum term considers all segments with a horizontal distance > 0. Those of these segments with a source unit are further optimized by the third sum term, which evaluates proper placement arcs (Section 7.5.7) and the special case of both nodes being placed in the same row (Section 7.5.3). The last term sums all constant horizontal distances along the path.

$$\begin{aligned}\forall P \in \mathfrak{P}(m) :\ dsm &- (\sum_{\substack{(u,v)\in P \\ \wedge u\in In\langle m\rangle \\ \wedge \textsc{HDist}[(u,v)]=0}} ds_{u,v}) \\ &- (\sum_{\substack{(u,v)\in P \\ \wedge \textsc{HDist}[(u,v)]>0}} dcp_{u,v} + dcn_{u,v}) \\ &+ (\sum_{\substack{(u,v)\in P \\ \wedge \textsc{HDist}[(u,v)]>0 \\ \wedge u\in In\langle m\rangle}} pa_{u,v} - or_{u,v}) \\ &- \sum_{(u,v)\in P} \textsc{HDist}[(u,v)] \\ &\geq 0\end{aligned} \tag{7.38}$$

Figure 7.5: Computing maximal critical path delay dsm

7.5.12 Objective Function

Since the vertical placement phase is purely timing driven, the objective function reduces to a minimization of the maximal critical path delay.

$$\textbf{minimize}\ dsm \tag{7.39}$$

7.5.13 Solving the Vertical Microplacement 0-1 ILP

With the problem scope being restricted to a single optimized master-slice, the model complexity is small enough to be exactly solved. E.g., the vertical placement phase of the larger problem mentioned in Section 7.4.3 had only 416 variables and 253 constraints, and was solved in 471s on the SparcStation 20/71. Under these circumstances, we have refrained from implementing an alternative heuristic. For larger problem sizes, though, a simulated annealing-based heuristic implementing the same placement model could easily be implemented.

7.6 Handling Sequential Elements

While the integration of SDI with UCB SIS has many advantages in terms of code reuse and robustness, it does incur some disadvantages. The most important one concerns the handling of sequential elements (flip-flops and latches): In SIS, all combinational components and primary ports are represented as gate network in the `network_t` data structure, with each port or gate being a node.

Sequential components, however, which were only retrofitted into SIS (based, in turn, on the purely combinational MIS), do not appear in the gate network. Instead, they are stored in an external table (latch table $Ff\langle m\rangle$). The sequential elements are related to the combinational circuit by "dummy" primary ports in the gate network, which correspond to the ports on the sequential component (e.g., D, Q, CLOCK etc.). As a result, operations on all components of a circuit must be implemented both on the network and the latch table. Instead, it would have been preferable to build on a unified data structure containing all circuit components in a graph-based representation.

However, due to the current implementation specifics, the microplacement of sequential elements was not performed in the first two phases (horizontal and vertical microplacement), but is undertaken in two dedicated substeps. In contrast to microplacement in general, the separation is not based on horizontal and vertical geometries, but on flip-flop connectivity.

All operations are performed at the level of the optimized master-slice, inter-slice relations do not need to be considered. An additional complication, however, is caused by circuits that contain more flip-flops than LUTs. In this case, the length of the placement area has to be increased (not shown) in Algorithm 13 until $l\langle m\rangle \cdot h\langle m\rangle \geq \max(|In\langle m\rangle|, |Ff\langle m\rangle|)$.

7.6.1 Placing Bound Flip-Flops

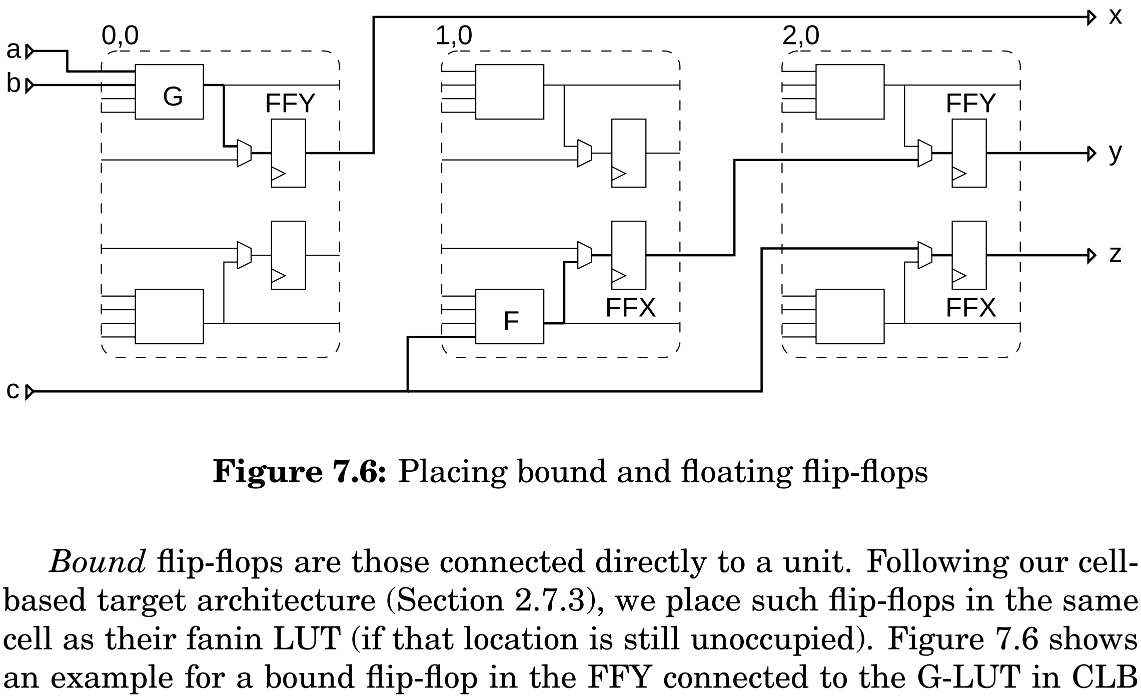


Figure 7.6: Placing bound and floating flip-flops

Bound flip-flops are those connected directly to a unit. Following our cell-based target architecture (Section 2.7.3), we place such flip-flops in the same cell as their fanin LUT (if that location is still unoccupied). Figure 7.6 shows an example for a bound flip-flop in the FFY connected to the G-LUT in CLB 0,0. The placement procedure for bound flip-flop just iterates over the latch table, and greedily assigns all unplaced bound flip-flops to the appropriate cell.

7.6.2 Placing Floating Flip-Flops

Bound flip-flops for which the greedy approach fails (optimal location already occupied), or those flip-flops either connected to another flip-flop, or a primary input port, are considered *floating* flip-flops. In Figure 7.6, both FFX and FFY in CLB 2,0 are floating flip-flops.

Floating flip-flops are placed using a simulated annealing-based heuristic (similar to Section 7.4). The evaluation function computes two kinds of SM distances for each flip-flop: The distance from the fanin node to the flip-flop input, and all SM distances from the flip-flop output to fanout nodes. The optimization then aims at minimizing the maximum of all these SM distances for all floating flip-flops in the circuit.

Algorithm 19: Computing switch matrix distances between arbitrary nodes

```
#define CLB_Y(r) (((r)+1)/2)  /* row (1-based): cells to CLBs*/
#define LEFTOF  (dx > 0)      /* dst left of src */
#define RIGHTOF (dx < 0)      /* dst right of src */
#define BELOW   (dy > 0)      /* dst below src */
#define ABOVE   (dy < 0)      /* dst above src */
#define HORIZ   ((dx == 0) && (ady > 1))  /* dst and src in same row */
#define VERT    ((adx > 1) && (dy == 0))  /* dst and src in same col */

/* distance metric (as used in vplace) */
int
sdi_ff_distance(sdi_loc *a, sdi_loc *b)
{
  short dx = a->col - b->col;  /* horizontal distance */
  short dy = CLB_Y(a->row) - CLB_Y(b->row); /* vert. CLB dist. */
  X = a->row & 1;              /* src is X output ? */
#define Y (!X)                 /* src is Y output ? */
  short adx = ABS(dx);         /* distance is computed as    */
  short ady = ABS(dy);         /*   absolute value of displacement */

  return (
   adx + ady -                      /* std. cartesian dist. */
    (   (((LEFTOF || BELOW ) && X)  /* reduce dist if proper arc */
      ||((RIGHTOF || ABOVE) && Y))
    && !(HORIZ || VERT) )           /* but not if in same row/col */
  );
}
```

Algorithm 19 lists the actual C code to quickly compute the SM distance between arbitrary nodes a, b in the cell-based architecture overlaid on the XC4000. Due to the high evaluation speed, we have not implemented incremental delay calculations, but recompute all flip-flop delays after each move. The moves of the simulated annealing are two-exchanges of flip-flop locations (either already occupied by a flip-flop, or currently empty).

In this manner, microplacement generates a complete placement of combinational and sequential elements in the optimized master-slices, such that their instances are suitable for stacking to assemble the compacted subdatapath.

7.7 Design Integration

Because of the limitations of the current floorplanner implementation Section 4.10, and the file format change from LCA to XNF in the Xilinx tools[7], design integration requires manual intervention. We currently import all generated netlists into SIS, export them again in EQN format, which is then converted to XNF format [Xili95a] using the tool EQN2XNF [KoCS90]. Since EQN format describes neither sequential elements, nor placement and partitioning, this data is written separately (directly in XNF) using a custom extension to SIS, and merged with the converted EQN data. The result is an XNF file containing combinational and sequential elements with partitioning and placement specifications.

Next, the datapath is imported into the Viewlogic Powerview design framework [View94a]. This is done by converting the XNF file into Viewlogic WIR format using the XACT tool XNF2WIR [Xili95b], and automatically generating symbols and schematics using VIEWGEN [View94b].

The SDI-processed datapath may now be merged with an irregular controller (either generated in, or imported into Viewlogic), and fed into the XACT designflow.

---

[7] Note that by now (June 1997), yet another change to EDIF is imminent.

8 Experimental Results

Since no standard corpus of benchmarks exists for regular datapaths, we evaluate our approach using four custom circuits. To this end, we concentrate on circuits whose speed can actually be influenced by proper floorplanning and compaction.

Two cases have to be considered: If the performance of a design depends only on the critical path through a specific (hard) macro, e.g., an adder's ripple-carry chain, it may be improved only by locally speeding up the module, and not by floorplanning and compaction[1]. On the other hand, performance gains by floorplanning and compaction are realized by minimizing the routing delays in an optimized regular placement. The number of logic levels on the critical path is generally not reduced as compared to that of flattened irregular circuits. Thus, when describing performance gains, we will quote two numbers, both in nanoseconds (ns): The *routing delay* consists only of the total interconnect delay between, but excluding logic blocks, the *total delay* also includes logic blocks. For each quantity, the improvement in quality will be expressed as a percentage (e.g., 25% means that the SDI design has only 75% of the delay of the XACT solution).

The partioning, placement and routing program PPR of the XACT toolsuite, which is also used for SDI routing, relies heavily on heuristics, and produces solutions widely varying in quality. Thus, we have run PPR multiple times to better explore the design space, and to observe the reproducibility of results[2]. Note, however, that only a single execution of the core SDI design cycle (floorplanning, compaction and microplacement) is performed for each experiment. Multiple SDI runs are not required, since the optimization problems are either solved exactly by ILPs, or with a heuristic tuned to reliably converge to an optimum (ensemble-based simulated annealing, Section 7.4).

All runtimes are listed in seconds, and apply to an otherwise unloaded SPARCstation 20/71 with 64MB RAM and local disk storage. The XACT version used was 5.2. Differences between the measurements given here and those in [Koch96a] [Koch96b] are due to changes in SDI and the earlier XACT version 5.1 used before.

---

[1] However, since the efficient realization of such modules is often dictated by the FPGA architecture (e.g., carry chains), only few alternatives exist for their implementation. [2] How many runs have to be performed to approach the upper performance limit?

8.1 Tools Used

Logic processing, placement, and routing in the Xilinx XACT-based designflow are performed by the program PPR [Xili94c]. For the placement and routing phases, PPR accepts parameters specifying the optimization effort. To obtain high-quality results, we always run PPR with maximal optimization settings (placer_effort = 5, router_effort = 4). These values put PPR in a timing-driven mode, where it tries to fulfill user-defined constraints on path delays [Xili94b]. The delays are specified separately, e.g., for paths beginning and ending at pads (dp2p), or beginning at a flipflop output and ending at a pad (dc2p). For each circuit, we determine the precise nature of the critical paths by using the XDELAY static timing analyzer (also part of XACT) [Xili94d]. Since PPR ignores delay constraints it deems infeasible, we experimentally determined the shortest delay limits actually respected by the program, and then let it optimize towards these shortest constraints.

In one case, we also use the Synopsys FPGA Compiler to resynthesize a logic unit after extending its word width from 8 to 32 bits [Xili94f]. As before, we run the tool with maximal optimization setting (map effort high) to obtain high-performance circuits [Syno96a]. Furthermore, this experiment also employs the XACT X-BLOX module generators [Xili94a] to generate regular modules in the XACT-based designflow.

In the SDI-based designflow, logic processing is performed by MIS-PGA [MSBS91a], FlowMap[CoDi94] or TOS-TUM [LeWE96]. Due to the multi-step nature of logic processing, *scripts* are used to describe each processing step and its parameters. For MIS-PGA and FlowMap, we use the scripts suggested in [Sent92] for optimization and mapping to LUT-based architectures. TOS-TUM is used with the high-effort performance-directed mapping for 4-LUTs described by its script mmap_h_p_4.scr. As suggested, we collapse the circuit prior to script execution.

The horizontal phase of the SDI two-phase placement step may either be performed using a 0-1 ILP (Section 7.2), or using a simulated annealing-based heuristic (Section 7.4). The vertical placement phase is always performed using a 0-1 ILP. The ILPs are solved using a hybrid strategy, combining the constructive OPBDP [Bart96] and branch-and-bound CPLEX [Cple94] solvers. The simulated-annealing is of the ensemble-based kind [RuPS91], and implemented building on a heavily modified version of the EBSA library [Frost93].

Design integration occurs as described in Section 7.7. It relies on the tools EQN2XNF [KoCS90], and XNF2WIR [Xili95b] for netlist conversion, VIEWGEN [View94b] for symbol generation, and Viewlogic Powerview [View94a] as front-end to the XACT tools.

Since SDI does not contain a dedicated router, it uses PPR for the routing phase. As usual, PPR is run with maximal timing driven optimization settings.

8.2 Generic 16-bit Datapath

8.2.1 Circuit

The circuit implemented is a 16-bit datapath consisting of two instances of a sample combinational module with a structure common to many bit-slices (shared control lines, vertical inter-slice signals). Each instance has four stacked segments of a single hzone of 16 4-LUTs (Figure 8.1).

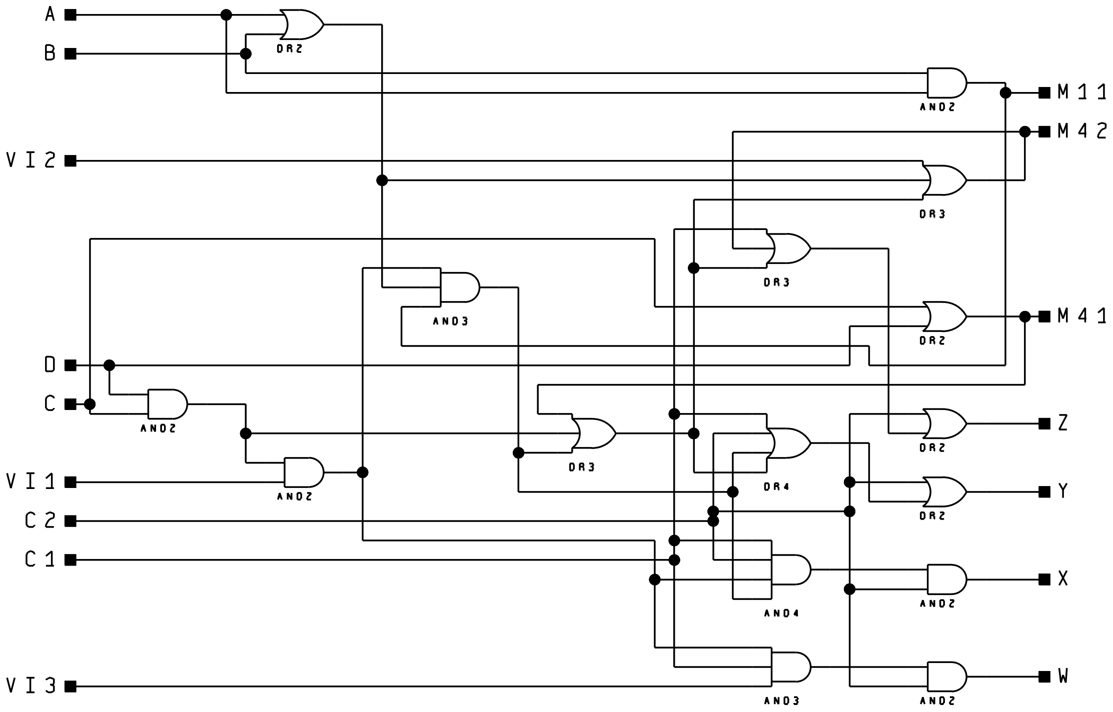


Figure 8.1: Single bit-slice of the example circuit

8.2.2 Processing

In order to directly compare placement results, technology mapping and minimization have been *disabled* both in SDI and PPR. PPR is run with maximum optimization in performance-driven mode (dp2p) with all pads floating. Both SDI and PPR placements were routed by PPR, also using maximum optimization.

Each placement and routing iteration of PPR takes an average of 307s. SDI takes 77s for horizontal placement, 8s for vertical placement and 55s for pad placement and routing via PPR for a total of only 140s on the same platform. The 0-1 ILP-based placement steps consisted of 392 variables and 436 constraints in the horizontal phase, and 240 variables and 155 constraints in the vertical phase.

8.2.3 Performance

Figures 8.2 and 8.3 show two layouts of the same circuit, one conventionally generated by the Xilinx tool PPR, the other one processed by our SDI. Figure 8.2 shows the best layout generated after 77 PPR iterations, Figure 8.3 shows the results of a single SDI run.

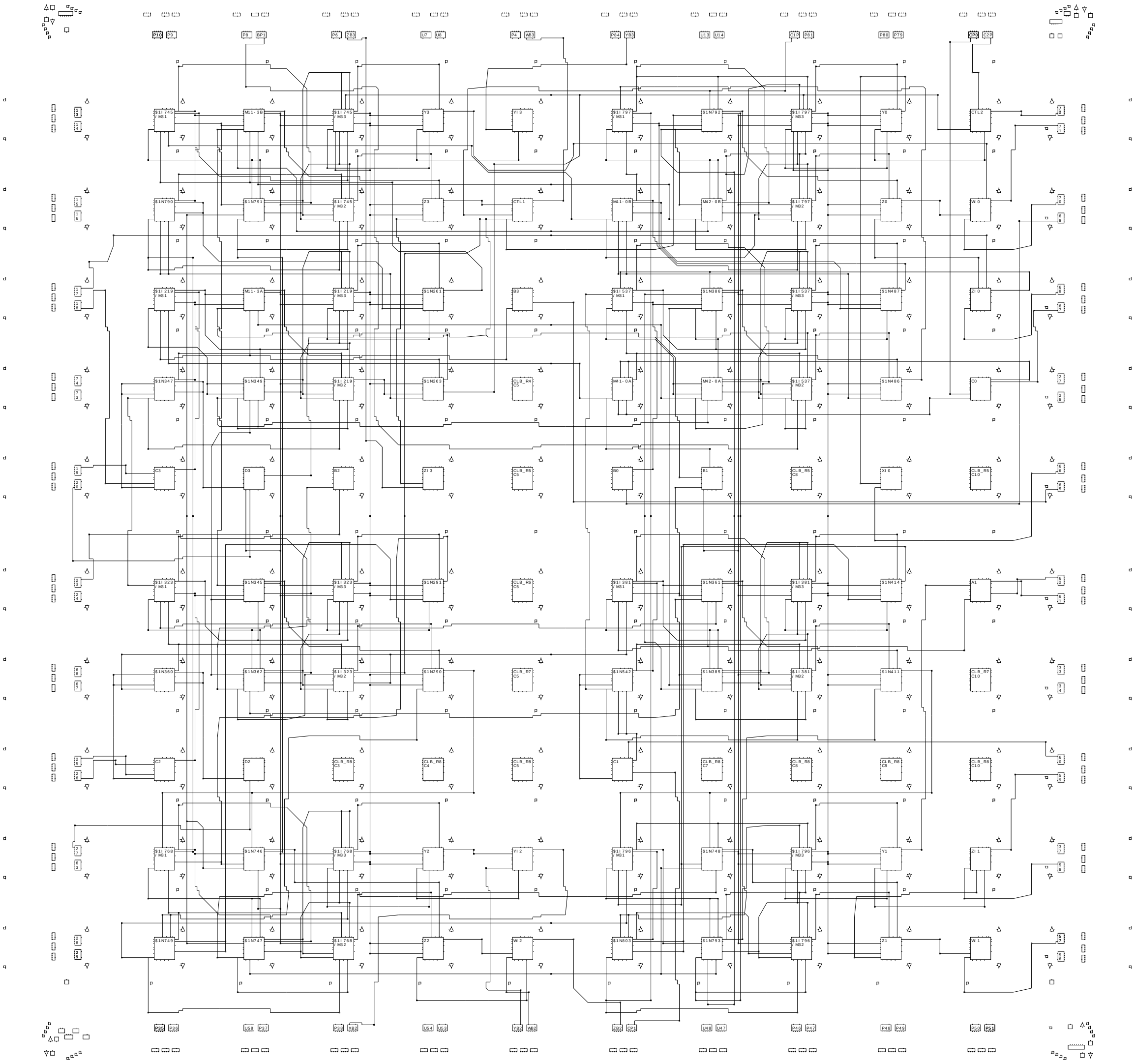


Figure 8.2: Placement and routing solely by PPR

Even at first glance, the SDI-generated solution is markedly more regular, since the natural structure of the datapath is exploited. The SDI layout is less congested than the PPR one, especially in the first quadrant. Table 8.1 shows the performance statistics for both layouts.

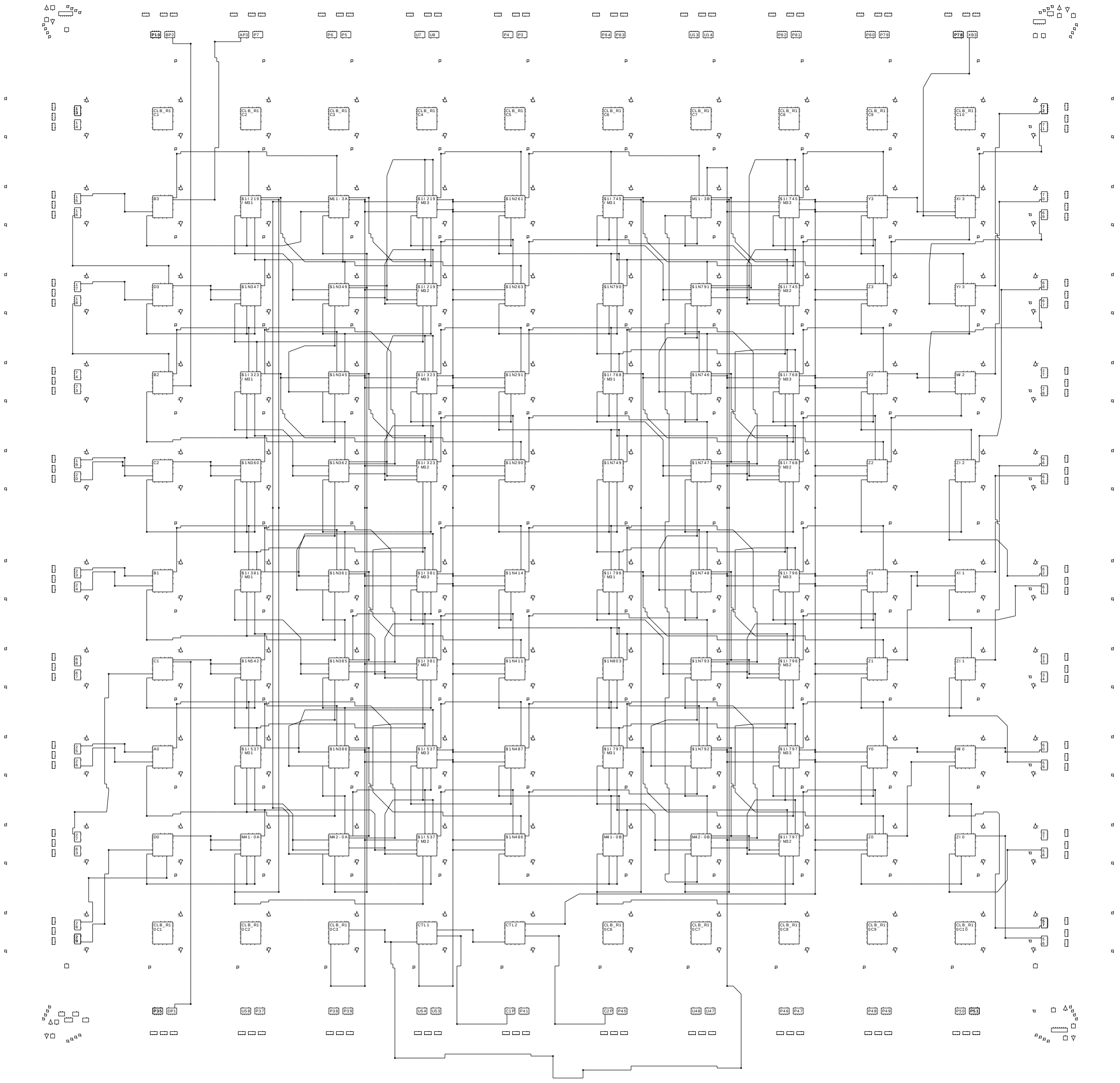


Figure 8.3: SDI placement with PPR routing

Design flow	Average runtime	#Runs	Routing delay best	Routing delay worst	Total delay best	Total delay worst	%Improvement routing	%Improvement total
XACT	307	77	30.4	37.6	118.4	123.7		
SDI	140	821	27.9	30.8	113.3	116.0	8 - 16	5 - 9

Table 8.1: Performance of generic 16-bit datapath

8.2.4 Comments

Note the improved reproducibility of results using SDI: Over trial 821 runs, the best-to-worst interval for SDI is just 2.8ns (10% from optimum). Thus, only very few SDI iterations (usually only a single one) are required to reliably determine the performance of a given circuit. For PPR, the best-to-worst interval after just 77 iterations is already 7.2ns wide (25% from optimum), and would grow steadily wider with an increasing number of runs.

8.3 74181-based 32-bit ALU

8.3.1 Circuit

The second example is a 32-bit ALU with registered inputs. It is composed of 8 4-bit 74181 slices in ripple-carry configuration. In this second example, both tools (SDI and XACT) perform their own technology mapping on a description of the ALU as shown in [Hwan79].

8.3.2 Processing

For SDI, mapping is performed by FlowMap or MIS-PGA (the “xl” commands in SIS). The mapped netlists are then placed by PPR or SDI, and all three circuits are routed by PPR. The design target is an XC4008PG191-5 chip.

As always, PPR is run with maximum optimization in performance-driven mode (dp2p, dc2p). For the XACT designflow, all pad placements were left floating. Both SDI and PPR placements were routed by PPR, also using maximum optimization.

In SDI horizontal placement is performed both by ILP and simulated annealing. The ILP model (Section 7.2) consisted of 913 variables and 1236 constraints, and unfortunately had a structure unsuitable even for the hybrid solver (Section 7.3). The required solution time of almost two hours was the main inspiration for implementing the annealing-based placement heuristic (Section 7.4) as an alternative. In contrast, the horizontal placement ILP for the ALU mapped with MIS-PGA (1128 constraints, 981 variables) is efficiently solved in only 570s. For both approaches, the vertical placement ILPs (372 variables/253 constraints for MIS-PGA/ILP, 426 variables/297 constraints for FlowMap/SA) were solved in 19s and 11s, respectively.

8.3.3 Performance

Table 8.2 shows the results for a number of PPR runs. Thus, the result of applying SDI with FlowMap/SA is an ALU (Figure 8.5) running 29% to 33% faster than the XACT-produced circuit (Figure 8.4), generated in one half to one third of the run-time of XACT, and with a reproducibility of 2% from optimum over hundreds of runs vs. 6% after two dozen runs for XACT.

Design flow	Average runtime	#Runs	Routing delay		Total delay		%Improvement	
			best	worst	best	worst	routing	total
XACT	1710	24	51.4	56.5	158.9	168.8		
SDI								
Flow/ILP	7311	135	37.0	40.1	115.4	118.6	29 - 35	28 - 32
Flow/SA	604	309	38.3	40.6	114.3	117.7	26 - 33	29 - 33
PGA/ILP	925	596	35.9	38.3	115.5	117.2	31 - 37	28 - 32

Table 8.2: Performance of 74181-based 32-bit ALU

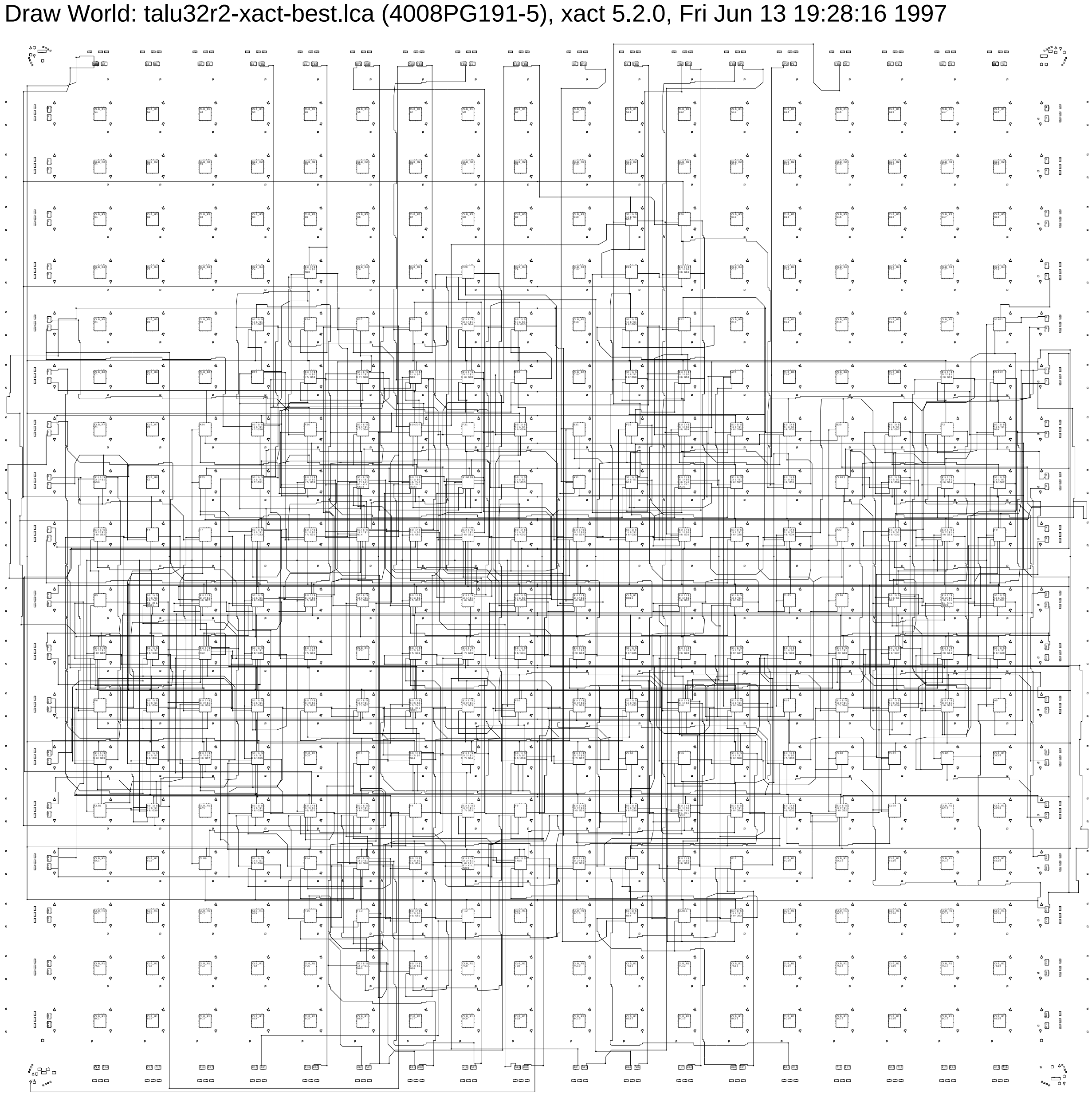

Draw World: talu32r2-xact-best.lca (4008PG191-5), xact 5.2.0, Fri Jun 13 19:28:16 1997

Figure 8.4: 32-bit 74181-based ALU implemented with XACT

Draw World: talu32r2-sdi-flow-best.lca (4008PG191-5), xact 5.2.0, Fri Jun 13 19:30:36 1997

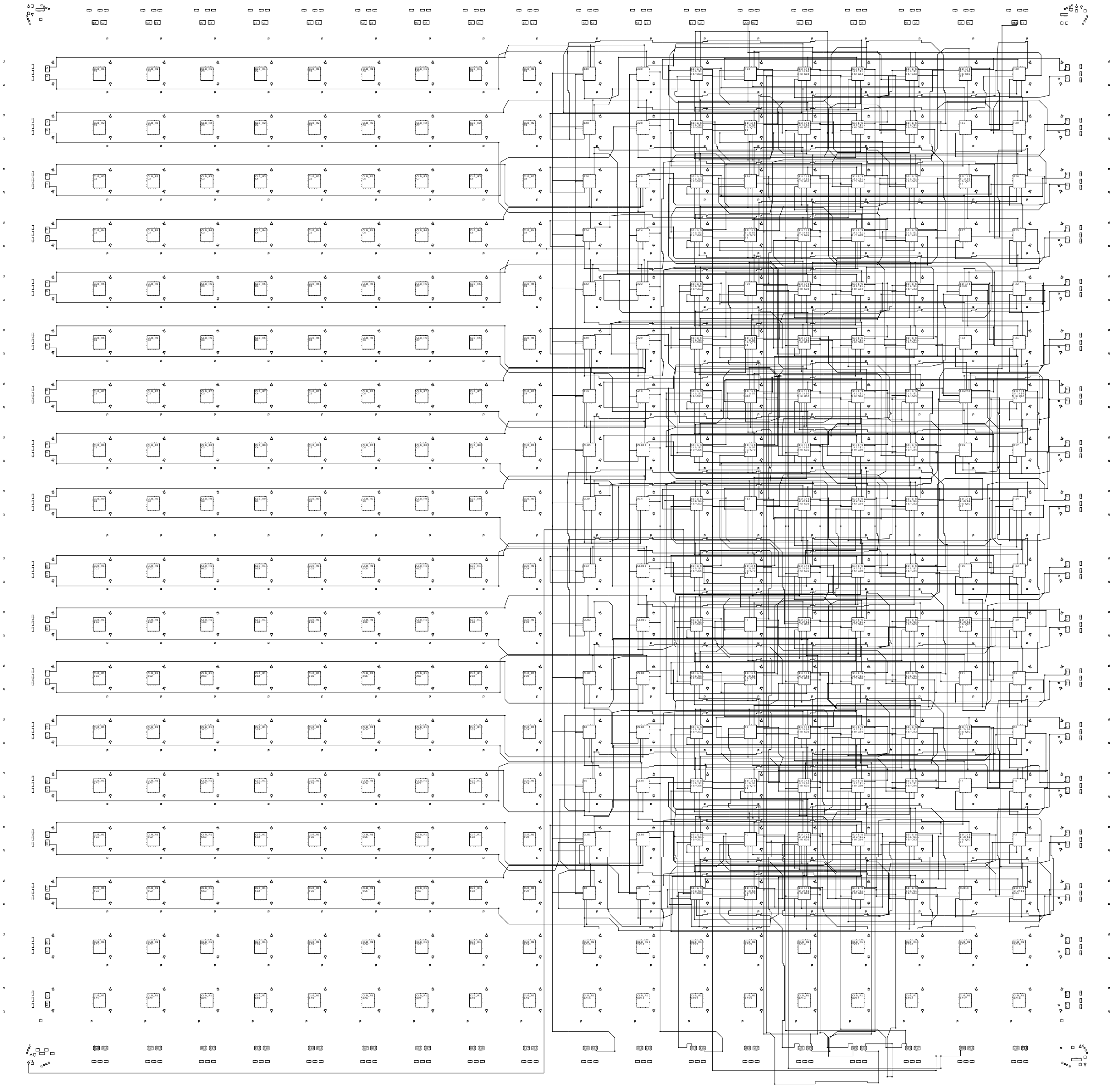


Figure 8.5: 32-bit 74181-based ALU implemented with SDI

Quantity	XACT	SDI		
		FlowMap/ILP	FlowMap/SA	MIS-PGA/ILP
4-LUTs in slice	(22)	26	26	24
CLBs in circuit	89	105	105	97
LUT levels in critical path				
mapped slice	?	4	4	5
placed circuit	20	15	15	15

Table 8.3: Logic processing statistics for 32-bit ALU

8.3.4 Comments

Since for this experiment, both designflows were permitted to perform their own logic processing, Table 8.3 shows some additional statistics.

The first column of Table 8.3 shows the results for an entirely XACT-based design flow. Since XACT processes the entire circuit without respecting slice boundaries, the number of 4-LUTs per slice can only be estimated, and the number of LUT levels per slice in the critical path is unknown.

The following points seem interesting: While FlowMap achieves a shorter critical path at the LUT level than MIS-PGA, the resulting circuit is more difficult to place and route. The routing delay in the critical path is marginally worse than that of the more conservative MIS-PGA solution.

When even more aggressive logic optimization methods employing computationally intensive methods such as kernel extraction (yielding a reduction to 20 4-LUTs per slice, 81 CLBs total) are applied, the clock speed degrades even further (slower than 190ns) as the circuit becomes too dense to be routed efficiently.

A characteristic of the regular layout generated by SDI is its height increasing proportionally to the bit-width (compaction turns folded structures into simple vertical ones). This can lead to wasted horizontal space (e.g., the left side of the XC4008 FPGA in Figure 8.5). It might be argued, that the irregular XACT placement might perform better when operating on a chip more closely matched to the size of the circuit to be processed. In the case of the 74181-based ALU, an XC4003 chip has sufficient capacity to hold an irregular placement (Figure 8.6). However, even then, the performance of the circuit falls short of the one for SDI generated designs: While the XACT runtime decreases to 1011s, the best achievable clock period (over 220 runs) still improves only to 155.1ns.

8.4 Address Generator for DES Encryption

8.4.1 Circuit

UFC-A is part of an address generator for a DES encryptor (based on the algorithm UFC [Glad97]). Figure 8.7 shows a single bit-slice in BLIF [Sent92]

Draw World: talu32r2-4003-best.lca (4003PG120-5), xact 5.2.0, Fri Jun 13 19:40:36 1997

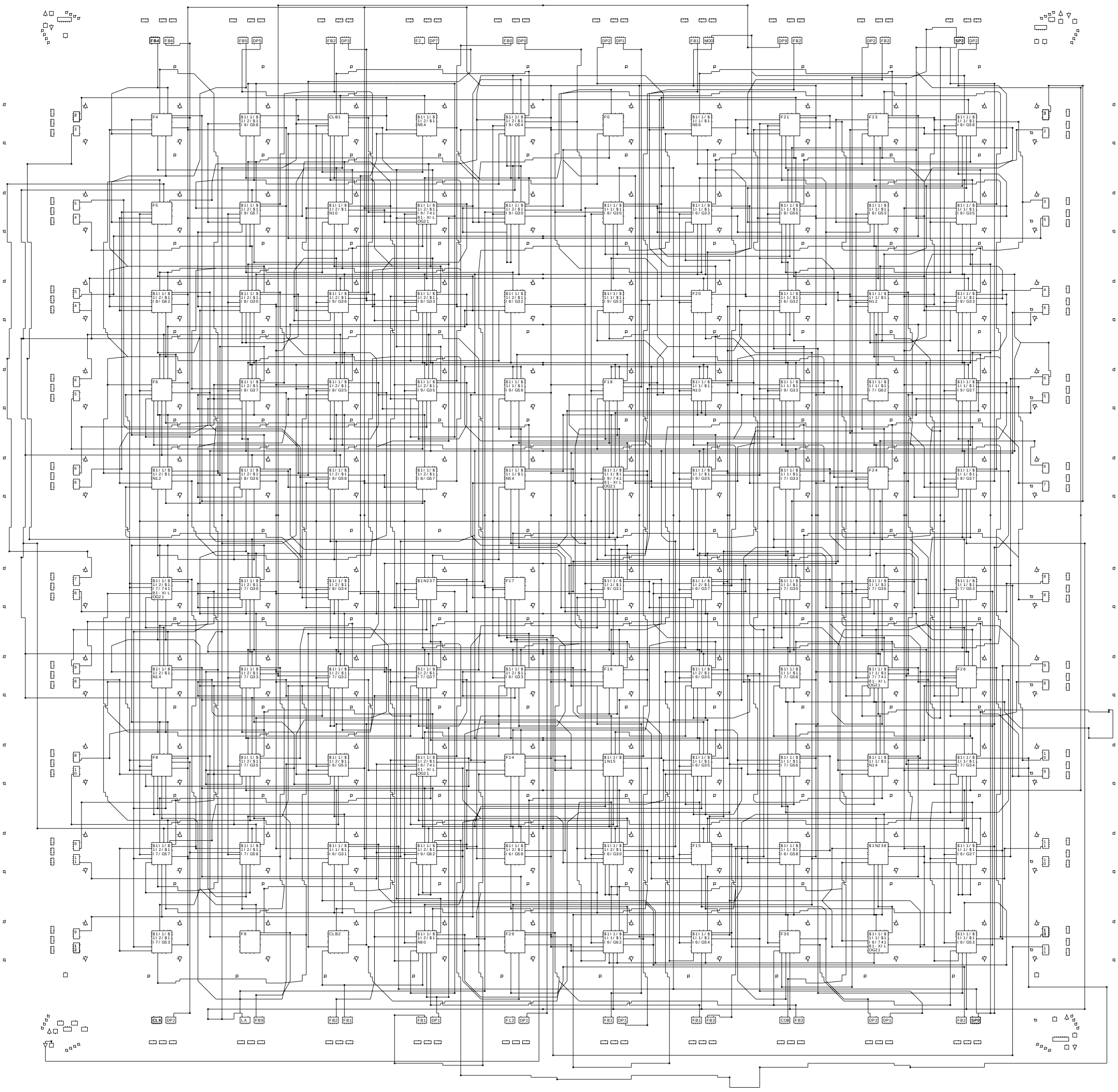


Figure 8.6: 32-bit 74181-based ALU implemented with XACT on XC4003

```
.model aolff
.inputs Data CIn InitKeytab SelOp10 SelOp11 SelOp2 SelAddr
        clockK clockKeyTab clockSB1 clockSB3 clockD clockA
.outputs Address COut

# constants
.names GND
0
.names VCC
1

# register file
# simulate clock enables with different clock signals
.latch Data RegK   re clockK      0
.latch Data KeyTab re clockKeytab 0
.latch Data RegSB1 re clockSB1    0
.latch Data RegSB3 re clockSB3    0
.latch Data RegD   re clockD      0
.latch Sum  RegA   re clockA      0

# extracted bit-slice (top-level cells only)
.subckt mux2 a=RegK b=KeyTab s=InitKeytab x=Ko
.subckt mux4 a=RegA b=Ko c=RegSB1 d=RegSB3 s0=SelOp10 s1=SelOp11 x=Op1

.subckt xor2 a=Data b=RegD x=XD
.subckt mux2 a=GND b=XD s=SelOp2 x=Op2
.subckt fulladd a=Op1 b=Op2 cin=CIn s=Sum cout=COut
.subckt mux2 a=Ko b=RegA s=SelAddr x=Address
.end
```

Figure 8.7: Bit-slice of address generator for DES encryption

format. To test the processing of more complex circuits without hard-macros, we have refrained from using a hard-macro adder, but composed a ripple-carry adder from basic gates. The resulting circuit consists of 26 16-bit soft-macros.

8.4.2 Processing

Logic processing the extracted bit-slice with TOS-TUM yields an optimized 2 BPLB slice of 16 4-LUTs, with a critical path of 4 4-LUTs. The reassembled circuit is thus composed of 128 LUTs and 96 flip-flops, and placed with the simulated annealing-based heuristic. XACT, relying on PPR for logic processing, yields the same number of elements. The circuit is laid out on an XC4003PG120-5 FPGA.

Design flow	Average runtime	#Runs	Routing delay		Total delay		%Improvement	
			best	worst	best	worst	routing	total
XACT	283	285	34.5	39.0	128.2	132.9		
SDI	216	41	30.8	33.2	124.4	127.4	11 - 22	3 - 7

Table 8.4: Performance of UFC-A address generator

8.4.3 Performance

Table 8.4 gives the performance data for the layouts created by XACT (Figure 8.8) and SDI (Figure 8.9).

8.4.4 Comments

Despite the noticeable improvement in routing delays, the total speedup realizable by SDI for UFC-A is limited by the number of logic blocks in the carry chain (18 LUT levels). When examining the two layouts, compare the mainly horizontal signal flow of the SDI-generated circuit with the mixed horizontal/vertical flow on the XACT solution. By preferring the horizontal, SDI has a higher routing density in horizontal channels. While the circuit remains routable, future work might actively consider routing congestion for non-control signals during microplacement (Section 2.7.1).

8.5 Logic Unit of RISC CPU

8.5.1 Circuit

The last example is part of the ALU for the SRISC CPU [Bruc94]. We concentrate on registers, logic functions (AND, OR, XOR, XNOR), and the shifter (left/right shift and rotate, logical and arithmetical modes). Following the intention of SDI to process wider structures, we extended the word size from the original 8 to 32 bits. Both XACT and SDI target an XC4010PG191-5 FPGA.

8.5.2 Processing

When extending the word size in the XACT design flow, we translated the ABEL-HDL description of the logic functions and shifter into Verilog, and synthesized them using the Synopsys FPGA Compiler (with high map efffort) to the XC4000 architecture [Xili94f]. To implement the registers with maximum efficiency, we employed the X-BLOX module generators [Xili94a]. After mapping, the circuit consists of 143 4-LUTs, 38 3-LUTs (XC4000 H-block), and 64 registers.

In SDI, structure extraction and regularity analysis discovered three different master-slices. All provide the same registers and logic functions. However, the bottom and top slices contain the special logic used for handling LSB and MSB in arithmetical and logical shift modes, while the middle slice just

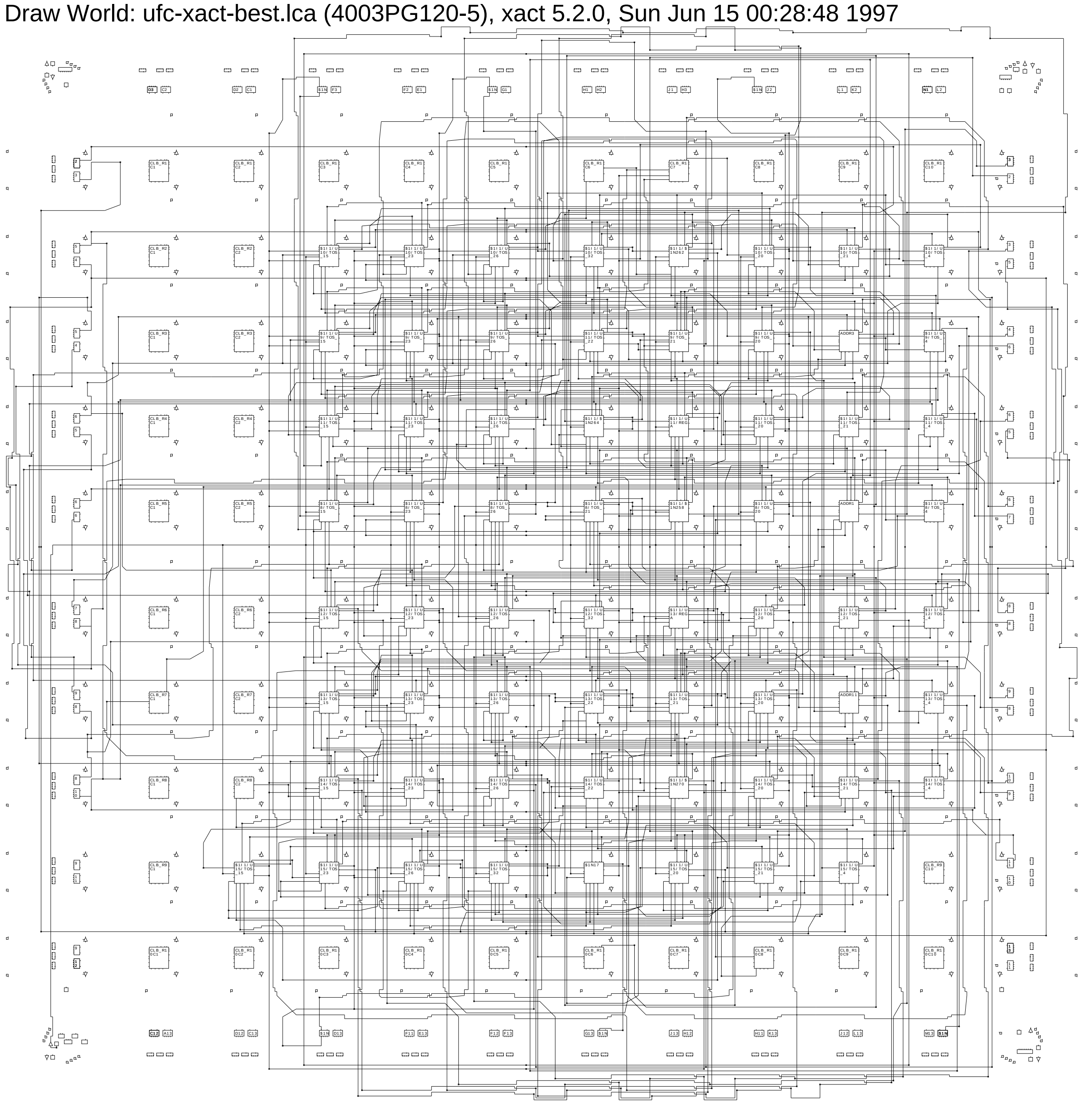

Draw World: ufc-xact-best.lca (4003PG120-5), xact 5.2.0, Sun Jun 15 00:28:48 1997

Figure 8.8: UFC-A address generator implemented with XACT

Draw World: ufc-sdi-best.lca (4003PG120-5), xact 5.2.0, Sun Jun 15 00:30:46 1997

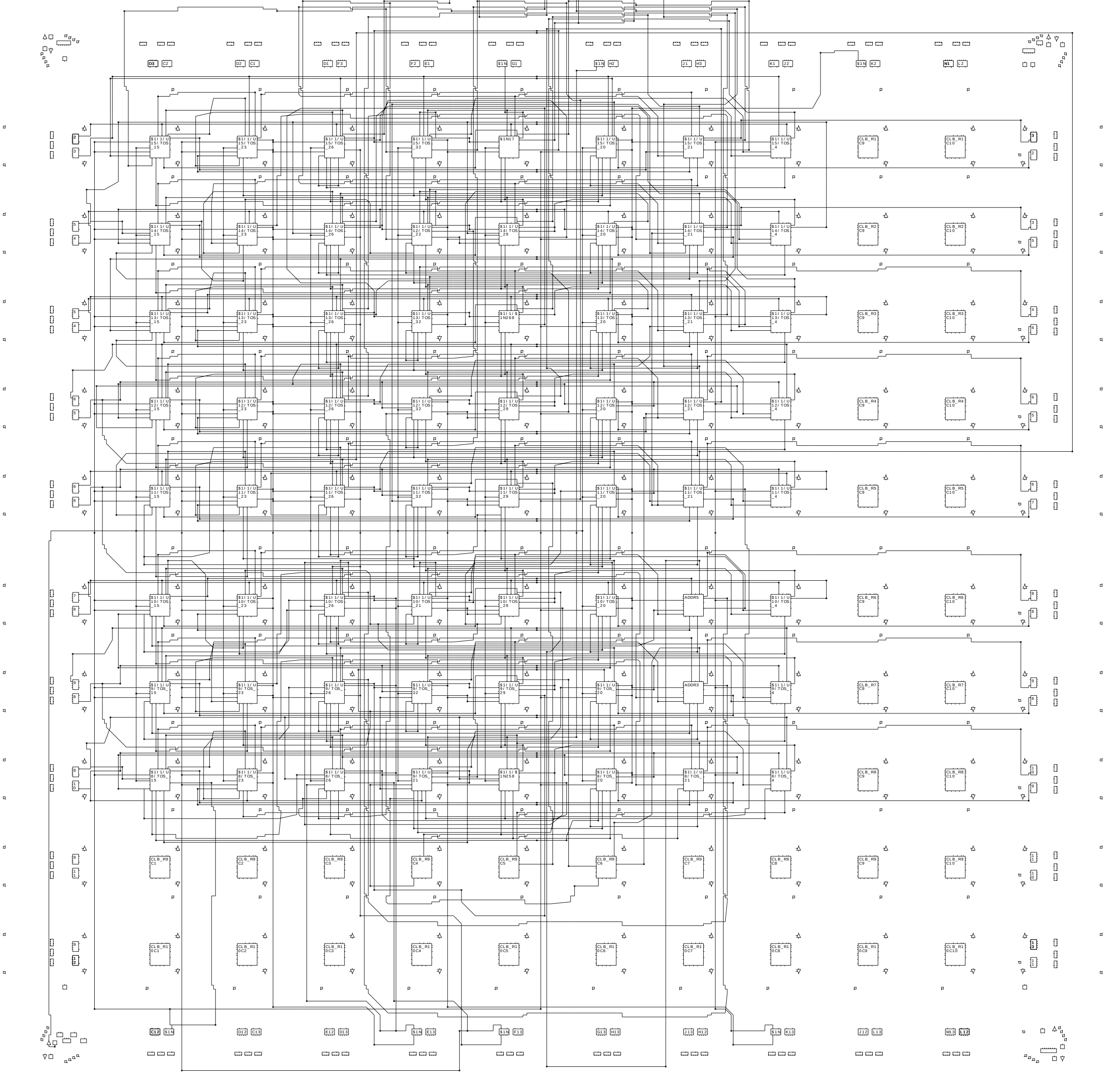


Figure 8.9: UFC-A address generator implemented with SDI

Design flow	Average runtime	#Runs	Routing delay		Total delay		%Improvement	
			best	worst	best	worst	routing	total
XACT	1131	37	15.6	20.2	44.6	49.1		
SDI	259	225	12.3	12.5	39.3	39.8	12 - 40	12 - 20

Table 8.5: Performance of 32-bit SRISC logic unit

provides an upward/downward shifting functionality. Thus, for the word size extension we simply replicate the middle slice 14 times instead of twice (using a 2 BPLB target topology to match the external adder/subtractor).

For logic processing in SDI, we again used TOS-TUM. In this manner, we obtained 9 4-LUTs for the bottom slice. The simpler middle slice consists only of 6 4-LUTs, and the top slice of 8 4-LUTs. The entire logic unit contains 101 4-LUTs and 64 registers. Horizontal microplacement in SDI is performed using simulated annealing.

8.5.3 Performance

Table 8.5 gives the performance data for the layouts created by XACT (Figure 8.10) and SDI (Figure 8.11).

8.5.4 Comments

Due to its regular, but more complicated structure, which in addition is free of long inter-slice critical paths (carry chains) limiting the speed-up potential, the SRISC logic unit is well suited to acceleration by SDI. Even when it is compared against an XACT solution also employing regular module generation techniques, the intra-module regularity provided by X-BLOX cannot compete with the inter-module view afforded by the SDI strategy.

8.6 Discussion

Since SDI mainly operates on a physical level, it cannot overcome speed limitations due to the logical or architectural nature of a design. However, within these limits, the gains achievable by primarily optimizing placement[3] are still appreciable. Even though the improvement may only be a few percent, this might make the difference between a design fulfilling the performance requirements on a given speedgrade of FPGA, or the need for a faster, but more expensive chip.

For those circuits better suited to regular optimization (e.g., the two ALUs), the performance gains can be quite substantial. As a secondary effect, the exploitation of regularity also reduces tool runtimes significantly. As shown for the SRISC logic unit, the stand-alone module generators currently available

---

[3] While the placement is also optimized for control routing, no actual routing is performed in SDI.

Draw World: ralu32-best.lca (4010PG191-5), xact 5.2.0, Sun Jun 15 02:34:11 1997

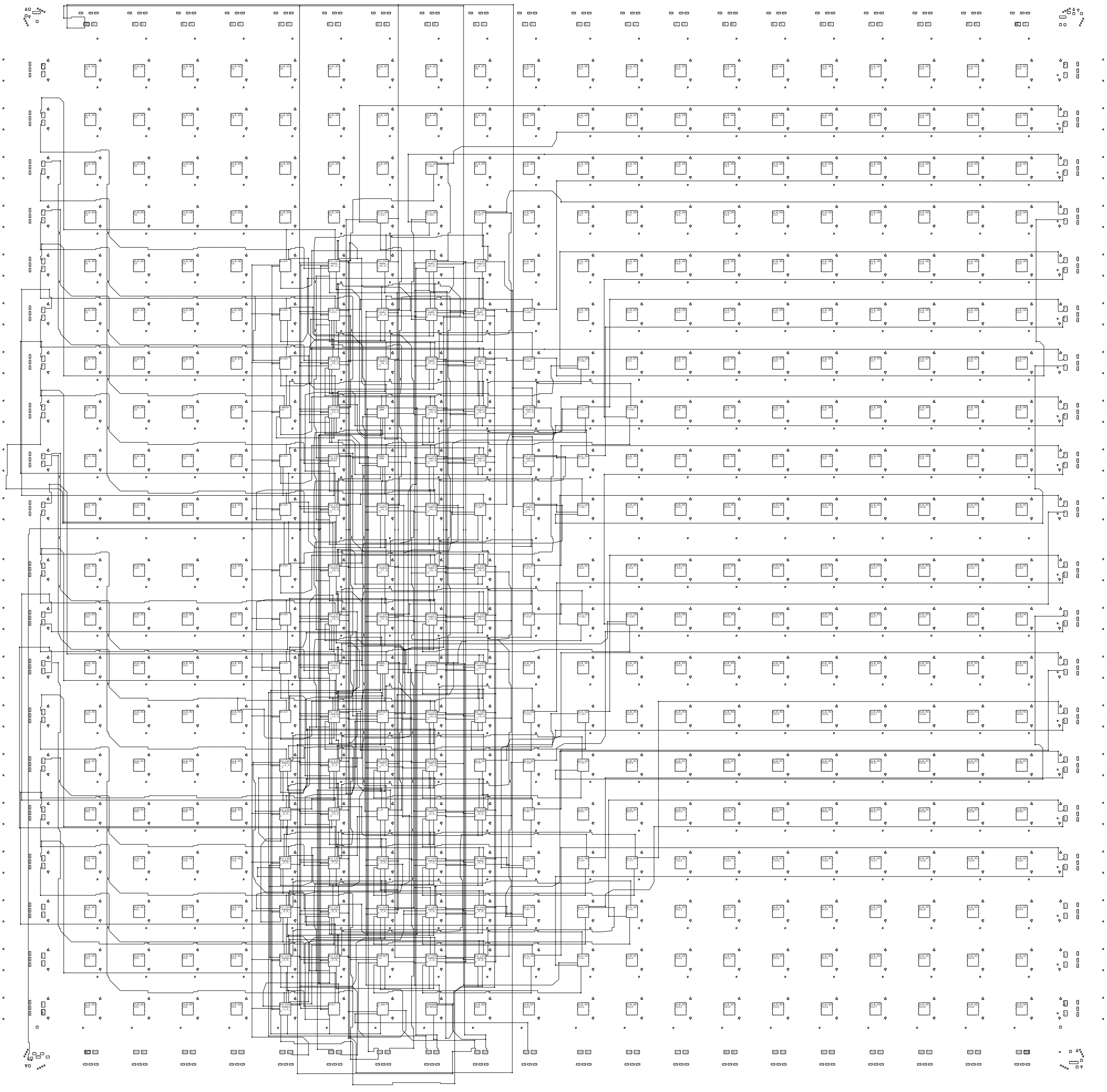


Figure 8.10: SRISC logic unit implemented with XACT

Draw World: rsditop32-best.lca (4010PG191-5), xact 5.2.0, Sun Jun 15 02:00:29 1997

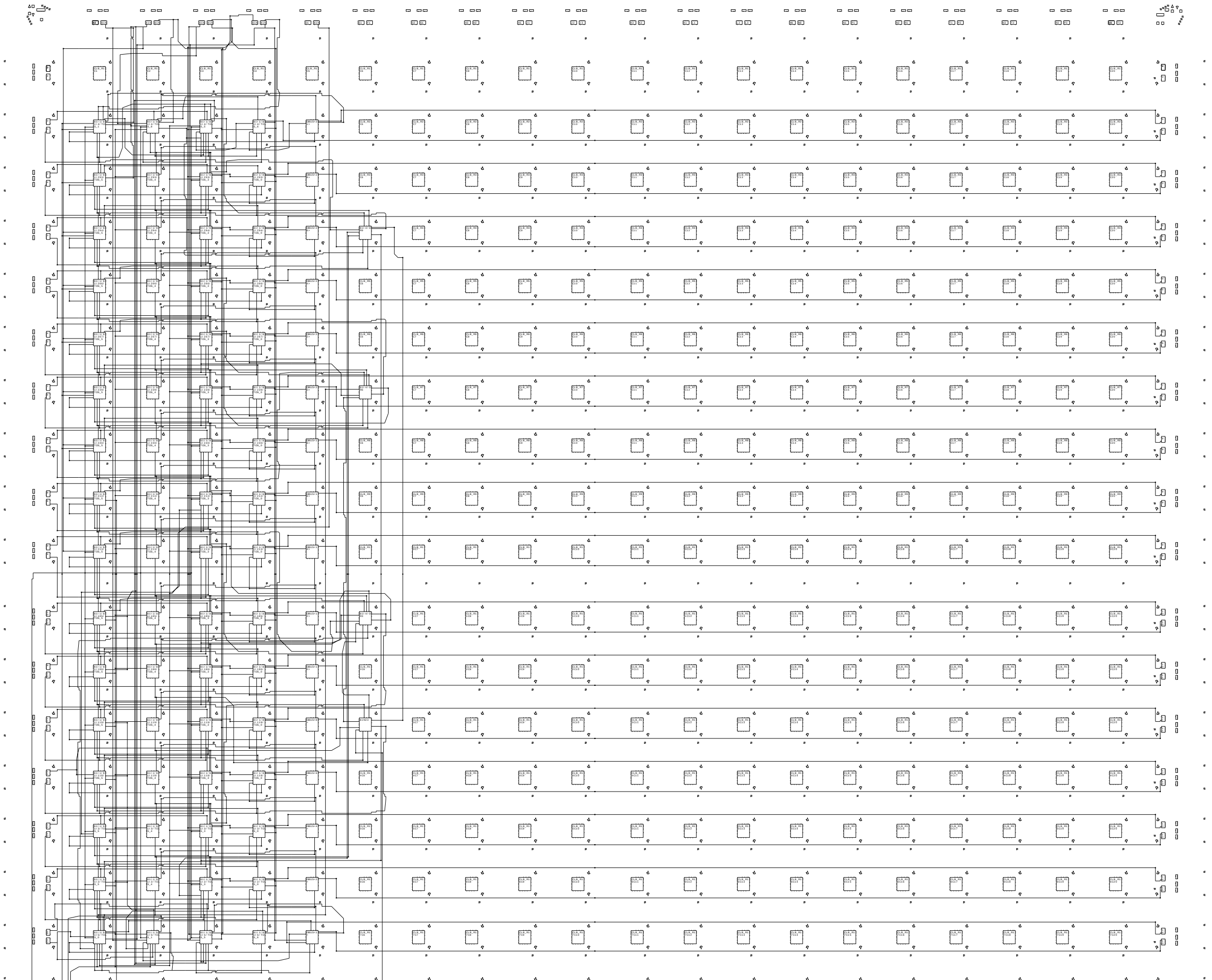


Figure 8.11: SRISC logic unit implemented with SDI

for many FPGA families (Xilinx X-BLOX, Actel ACTgen, Atmel generators) lose much optimization potential as compared to the wider scope enjoyed by a strategy such as SDI.

The evaluation process was complicated by three main aspects: First, the lack of an established suite of benchmark circuits for datapath structures. Second, the need to manually isolate datapaths in the sample circuits used. While this was still possible for simpler designs available as schematics, it proved to be unmanageable for designs specified using RTL-level HDL. And third, due to the immaturity of the floorplanner (Section 4.10), many design tasks (especially design integration and netlist generation) had to be performed manually.

While the last difficulty is "easily" remediable by creating a more robust floorplanner, the second problem offers much potential for future research (Chapter 9). Assuming the existence of an "automatic HDL datapath extractor", the first problem could then be solved by simply extracting regularity from various academic and industrial designs currently only available as HDL descriptions.

9 Summary and Future Work

After describing the limitations of current CAD tools for FPGAs, especially with regard to processing the datapath structures common to many CPU and DSP designs, this work introduced SDI, a novel approach to increase datapath circuit performance.

SDI comprises a specialized suite of easily extensible CAD tools coordinated by a comprehensive strategy. Our *leitmotiv* is the direct mapping of datapath *architecture* to *physical* layout. The front-end tools of SDI (module generators offering implementation alternatives, floorplanner, structure extraction, regularity analysis) are optimized for the processing of regular datapaths. They either preserve a known bit-sliced structure (module generators, floorplanner), or regenerate it in a larger context (structure extraction, regularity analysis). The back-end tools (module generators generating layouts or netlists, compaction, microplacement) then map the regular architecture to a regular layout optimized for the specific target FPGA. To this end, traditional logic synthesis and technology mapping techniques are fully embedded in the optimized design flow.

SDI is thus specialized on both the highest and lowest abstraction levels: Architectural features of datapaths, such as external control signals fanning out to multiple bit-slices, or inter-bit-slice signals, are directly considered as such during layout generation. In addition, FPGA-specific intricacies, like fast carry chains, on-chip memory blocks, and hybrid symmetrical/hierarchical interconnection networks are also taken advantage of in hard-macros to maximize low-level performance.

Despite the limitations of the current SDI implementation, with some components only being academic prototypes, and the lack of standard benchmark circuits, experimental results obtained for a number of representative designs are very promising. Structures that are either very simple (not much regularity to exploit), or that are limited by long inter-bit-slice (serial) critical paths can only be accelerated by a few percent. Circuits with a more complex regular structure (multiple different slices, larger slices), however, can gain over 30% in performance. For an optimization occuring at the very low abstraction level of physical layout, these improvements are quite respectable[1]. As a secondary benefit, the computation times for SDI are also far shorter than for conventional processing.

Even the smaller performance gains become appreciable when consider-

---

[1] E.g., the maximum gains for a novel performance-driven FPGA router [AlRo96] are between 2% - 18%, with an average of 10%.

ing the discrete speedgrades in which FPGA chips are sold. The few percent of SDI-optimized performance can make the difference between the performance requirements being met with a lower speedgrade, or the need to use a faster and more expensive chip. The performance improvement then translates directly into an economical gain, growing in proportion to the production volume.

While working on, and experimenting with, SDI, we discovered numerous areas for further research and development. To actually enable the use of SDI in a production setting, a fresh implementation of the floorplanner is absolutely required. In this reimplementation, the VLSI specifics should be emphasized over the optimization aspects of the problem. Furthermore, for ease of use, the input format should be changed from the proprietary SNF to a standardized format, such as a structural hardware description language (HDL, e.g., VHDL, Verilog), or EDIF. The performance of SDI-optimized designs could probably be further improved by a more active consideration of routing congestion, both during logic processing and microplacement. To cover the first area, a routability-oriented technology mapper such as [ScKC94] might be integrated. In the second area, basic congestion handling could be added to the microplacer, possibly remaining restricted to the more detailed vertical placement phase. Due to the risk of the resulting more complex ILP models becoming intractable, this step might also necessitate the switch to an annealing-based heuristic for vertical placement. If even shorter computation times for SDI were required, the currently untapped potential of parallel processing could be exploited. As described on numerous occasions in the main text, many of the operations following circuit regularities may be performed simultaneously and independently of each other.

Especially during the experimental measurement phase, the following issue became painfully apparent: The premise of letting the user manually separate the regular datapath from the irregular controller, while feasible during the era of schematic design entry, is no longer realistic with the continuing trend towards the higher abstraction levels. What is required is an automatic extraction of regular structures from an RTL-HDL description. Such a tool would then allow the clean integration of SDI into future design flows heavily relying on high-level synthesis [GDWL92] [Holt93] [Lin97] and hardware-software co-design [HEHB94] [EHBY96].

Furthermore, the compaction and microplacement components of SDI could most likely be embedded in such a synthesis system to allow the “on-the-fly” generation of regular modules from unstructured netlists. Current synthesizers, such as DesignCompiler [Syno96a], rely on hardcoded module generators (e.g., DesignWare [Syno96b]) to create regular structures. When encountering a subcircuit not covered by an existing generator, the logic is generated in an unstructured manner. But the SDI compaction and microplacement phases are specialized on optimizing subcircuits (compaction relies on local logic synthesis), and then restoring regularity by microplacement. In this manner, any kind of regular logic (discovered by the “datapath extractor” suggested above) could be synthesized into a regular module. The efficiency would most likely be degraded over that of a dedicated module generator, but still be improved over an unoptimized irregular structure.

From a software engineering point of view, the SDI project demonstrated the power of building on existing frameworks: The module generators and floorplanner were implemented as stand-alone tools using C++, but without emphasis on object-oriented design aspects. The remaining tools, such as compaction, microplacement, various utility functions (timing analysis, XNF netlist generation etc.) were embedded into the well-established SIS system. Even though “only” C was used as their implementation language, the clean and consistent design of SIS improved developer productivity and code reliability immensely. Unfortunately, SIS is already showing its age, e.g., by supporting only flat structures, and the awkward handling of sequential elements. Furthermore, with its roots firmly in the logic synthesis area, adding the operations and data structures required for physical design automation tasks, such as placement and routing, result in a kind of “impedance mismatch” due to the different circuit views. It would be beneficial for the entire EDA community if a more modern successor were developed. By actually employing current object-oriented techniques [Booc94] [GHJV95] [Koch97a], a robust, easily extensible framework offering multiple hierarchical design views (e.g., behavioral, structural, physical) and consistent basic services (e.g., HDL export/import, timing analysis etc.) could be developed. This groundwork could spare EDA researchers and developers everywhere the effort to start all new projects “from scratch”.

To conclude, while the fundamental approach of SDI to improve circuit performance by the exploitation of regularity has already proved successful, the project was also quite fertile in showing many new venues for further research. We especially recommend that the possibilities for integration with high-level synthesis, and of a unified EDA software framework, be pursued further.

Bibliography

[3Com95] 3Com Corp, “3Com Selects Actel FPGAs for LAN Logic Integration, Gate Array Migration Path”, in [Acte95a], pp. 11-27 1

[ACCK96] Amerson, R., Carter, R., Culbertson, W., Kuekes, P., Snider, G., “Plasma: An FPGA for Million Gate Systems”, *Proc. FPGA 1996*, pp. 10-16 2.3.1

[Acte95a] Actel Corp, “FPGA Data Book and Design Guide”, *Databook*, Sunnyvale 1995 2, 2.3.1, 95

[Acte96] Actel Corp., “Actel’s Reprogrammable SPGAs”, *Preliminary Advance Information*, Sunnyvale 1996 2

[AlRo96] Alexander, M.J., Robins, G., “New Performance-Driven FPGA Routing Algorithms”, *IEEE Trans. on CAD*, Vol. 15, No. 12, December 1996, pp. 1505-1516 1

[Algo92] Algotronix Ltd., “CAL1024 Product Brief”, *Datasheet*, Edinburgh 1992 2.3.1

[Alte96] Altera Corp., “MAX9000 Programmable Logic Device Family”, *Datasheet*, San Jose 1996 2

[Alte95] Altera Corp., “FLEX 10K Embedded Programmable Logic Family, ver. 1”, *Datasheet*, San Jose 1995 2, 2.3.1, 2.7.3

[AMDI90] AMD Inc., “PALASM-XL Reference Manual”, *EDA Software Documentation*, San Jose 1990 3

[AMDI96] AMD Inc, “The MACH 5 Family”, *Datasheet*, Sunnyvale 1996 2

[Atme94] Atmel Corp., “Atmel Configurable Logic Design & Applications Book”, *Databook*, San Jose 1994 2.3.1, 2.3.1

[ATTM94] AT&T Microelectronics, “ORCA FPGA Development System”, *EDA Software Documentation*, Allentown 1994 2.2

[ATTM95] AT&T Microelectronics, “AT&T Field-Programmable Gate Arrays”, *Databook*, Allentown 1995 2, 2.3.1

[BaCM92] Babba, B., Crastes, M., Saucier, G., “Input driven synthesis on PLDs and PGAs”, *Proc. EDAC 1992*, pp. 48-52 2.7.3

[BaSe88a] Barth, R., Serlet, B., "A Structural Representation for VLSI Design", *Proc. 25th DAC 1988*, pp. 237-242

[BaSe88b] Barth. R., Serlet, B., Sindhu, P., "Parametrized Schematics", *Proc. 25th DAC 1988*, pp. 243-249 3.1, 3.1.3, 3.2.3

[BaMS88c] Barth, R., Monier, L., Serlet, B., "PatchWork: Layout from Schematic Annotations", *Proc. 25th DAC 1988*, pp. 250-255 3.1.1

[Bart95] Barth, P., "A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization", *Memo MPI-I-95-2-003*, Max-Planck-Institut für Informatik, Saarbrücken 1995 7.2, 7.2, 7.3.1, 7.3.1, 7.3.3

[Bart96] Barth, P., "Logic-Based 0-1 Constraint Programming", Kluwer 1996 7.2, 7.3.1, 7.3.1, 8.1

[BeCo93] Benhamou, F., Colmerauer, A., "Constraint Logic Programming", MIT Press 1993 7.2

[BEKS95] Benner, T., Ernst, R., Könenkamp, I., Schüler, P., Schaub, H.-C., "A Prototyping System for Verification and Evaluation in Hardware-Software Cosynthesis", *Proc. 6th Int'l Workshop on Rapid System Prototyping 1995* 1

[BDRA93] Bolsens, I., DeWulf, R., Renaudin, M., Airiau, R., Betts, A., "Alternative Implementations of SMILE using Cathedral & VHDL", *Proc. 4th EUROCHIP 1993*, pp. 248-253 3

[BeGr93] Ben Ammar, L., Greiner, A., "A High Density Datapath Compiler Mixing Random Logic with Optimized Blocks", *Proc. EDAC 1993*, pp. 194-198 2.3.1, 3.1.1, 3.2.3

[BFRV92] Brown, S., Francis, R., Rose, J., Vranesic, Z., "Field-Programmable Gate Arrays", Kluwer 1992 2, 2, 2.3.1

[BhHi92] Bhat, N. B., Hill, D. D., "Routable Technology Mapping for LUT FPGAs", *Proc. ICCD 1992*, pp. 95-98

[Brow96] Brown, S., "FPGA Architectural Research: A Survey", *IEEE Design & Test*, Vol. 13, No. 4, 1996, pp. 9-15

[BrRo96] Brown, S., Rose, J., "FPGA and CPLD Architectures: A Tutorial", *IEEE Design & Test*, Vol.13, No. 2, 1996, pp. 42-57. 2

[Bode97] Bodenstein, M., "Anwendung und Optimierung eines Plazierungssystems f'ur FPGAs basierend auf einem fuzzygesteuerten Genetischen Algorithmus", *Diploma Thesis*, TU Braunschweig, Abt. E.I.S., 1997 2.5, 4, 4.9, 4.9, 10

[Booc94] Booch, G., "Object-Oriented Analysis and Design with Applications, 2nd ed.", Addison-Wesley 1994 3.3.7, 34

[BoSW90] Bower, W., Seaquist, C., Wolf, W., "A Framework for Industrial Layout Generators", *Proc. 27th DAC 1990*, pp. 419-424 3.1.3, 3.3.7, 3.3.7

[BrMR94] Brand, H.J., Müller, D., Rosenstiel, W., "Specification and Synthesis of Complex Arithmetic Operators for FPGAs", in *Field Programmable Logic – Architectures, Synthesis and Applications*, ed. by Hartenstein R.W., Servits, M.Z., Springer 1994, pp. 78-88 2.4, 3.1, 3.1.2

[Bruc94] Bruch, C., "Entwurf und Aufbau eines Specialized Reduced Instruction Set Computers (SRISC)", *Diploma Thesis*, University of Siegen, 1994 10, 8.5.1

[BuAK96] Buell, D.A., Arnold, J.M., Kleinfelder, W.J., "Splash 2 – FPGAs in a Custom Computing Machine", IEEE Computer Society Press, 1996 1, 2

[CGPP91] Compan, A., Greiner, A., Pecheux, F., Petrot, F., "GENVIEW: A Portable Source-Level Desbugger for MacroCell Generators", *Proc. ICCAD 1991*, pp. 408-412 3.1.3

[ChCh93] Chih-Liang, E.C., Chin-Yen, H., "SEFOP: A Novel Approach to Data Path Module Placement", *Proc. ICCAD 1993*, pp. 178-181 2.2

[Chip97] Chip Express Inc., "QYH500 Specifications", *http://www.chipexpress.com*, Santa Clara 1997 2

[ChRo92] Chung, K., Rose, J., "TEMPT: Technology Mapping for the Exploration of FPGA Architectures with Hard-Wired Connections", *Proc. 29th DAC 1992*, pp. 361-367 3.3.7

[ChYu93] Chau-Shen, C., Yu-Wen, T., "Combining Technology Mapping and Placement for Delay-Optimization in FPGA Designs", *Proc. ICCAD 1993*, pp. 123-127

[CNSD90] Cai, H., Note, S., Six, P., DeMan, H., "A Data Path Layout Assembler for High-Performance DSP Circuits", *Proc. 27th DAC 1990*, pp. 306-311 2.3.1

[CoDi94] Cong, J., Ding, Y., "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs", *IEEE Trans. on CAD*, Vol. 13, No. 1, January 1994, pp. 1-12 6.5.1, 8.1

[CoDi96] Cong, J., Ding, Y., "Combinational Logic Synthesis for LUT-Based Field Programmable Gate Arrays", *ACM Trans. on Design Automation of Electronic Systems*, Vol. 1, No. 2, April 1996, pp. 145-204 5, 6.5

[Cple94] CPLEX Optimization Inc., "Using the CPLEX Callable Library", *User Manual*, Incline Village (NV) 1994 7.3.2, 7.3.2, 8.1

[Cros94] Crosspoint Inc, "CP20K Field Programmable Gate Arrays", *Databook*, Santa Clara 1994 2

[CuoL96] Cuong-Chan, V., Lewis, D.M., "Area-Speed Tradeoffs for Hierarchical Field-Programmable Gate Arrays", *Proc. FPGA 1996*, pp. 51-57 2.3.1

[Demi94] DeMicheli, G., "Synthesis and Optimization of Digital Circuits", McGraw-Hill 1994

[DeNe87] Devadas, S., Newton, R.A., "Topological Optimization of Multiple-Level Array Logic", *IEEE Trans. on CAD*, Vol. 6, No. 11, November 1987, pp. 915-941 3.1.1

[Deo74] Deo, N., "Graph Theory with Applications to Engineering and Computer Science", Prentice-Hall 1974 5, 6.4

[Ditt95] Dittmer, J., "Parametrisierbare Modulgeneratoren für die FPGA-Familie Xilinx XC4000: Arithmetik mit der Hard-Carry-Logik und Speichermodule", *Diploma Thesis*, TU Braunschweig, Abt. E.I.S., July 1995 2.4, 4, 3.1.3, 3.3, 4.7.1, 4.7.4

[EbCF96] Ebeling, C., Cronquist, D.C., Franklin, P., "RaPiD – Reconfigurable Pipelined Datapath", *Proc. 6th FPL 1996*, pp. 126-135 2

[EIA93] Electronics Industry Association (EIA), "Library of Parametrized Modules (LPM)", *EIA/IS-103*, May 1993 3

[EHBY96] Ernst, R., Henkel, J., Benner, T., Ye, W. Holtmann, U., Hermann, D., Trawny, M., "The COSYMA Environment for Hardware/Software Cosynthesis of Small Embedded Systems", *Microprocessors and Microsystems*, Vol. 20, No. 3, May 1996, pp. 159-166 34

[EsKu96] Esbensen, H., Kuh, E.S., "EXPLORER: An Interactive Floorplanner for Design Space Exploration", *Proc. EDAC 1996*, pp. 356-361 2.5, 4.8

[Fiel95] Fields, C.A., "Proper Use of Hierarchy in HDL-Based High Density FPGA Design", in *Field-Programmable Logic and Applications, Proc. 5th FPL 1995*, ed. by Moore, W., Luk, W., Springer 1996, pp. 168-177

[Frost93] Frost, R., “EBSA C Library Documentation”, *User Manual*, San Diego Super Computing Center (SDSC) 1993 7.4.2, 8.1

[GDWL92] Gajski, D., Dutt, N., Wu, A., Lin, S., “High Level Synthesis”, Kluwer 1992 34

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., “Design Patterns”, Addison-Wesley 1995 3.3.7, 34

[Glad97] Glad, M, “GNU crypt 2.0.4”, *ftp://ftp.ifi.uio.no/pub/gnu/glibc-crypt-2.0.4.tar.gz*, 1997 8.4.1

[Gold89] Goldberg, D.E., “Genetic Algorithms in Search, Optimization and Machine Learning”, Addison-Wesley 1989 2.5, 4

[GrPe97] Greiner, A., Pêcheux, F., “ALLIANCE: A complete set of CAD tools for teaching VLSI design”, *ftp://ftp.ibp.fr/ibp/softs/masi/alliance/OVERVIEW.ps*, 1996 2.3.1, 7

[GrSt94] Groeneveld, P., Stravers, P., “Ocean: the sea-of-gates design system”, *http://cas.et.tudelft.nl/software/ocean/ocean.html*, 1994 2.3.1

[GuSm86] Gu, J., Smith, K.F., “KD2: An Intelligent Circuit Module Generator”, *Proc. ICCD 1986*, pp. 470-475 3.1.1

[HEHB94] Henkel, J., Ernst, R., Holtmann, U., Benner, T., “Adaptation of Partitioning and High-Level Synthesis in Hardware/Software Co-Synthesis”, *Proc. ICCAD 1994*, pp. 96-100 34

[Holt93] Holtmann, U., “High-Level Synthese mit BSS für den Einsatz im Hardware/Software Co-Design”, *Proc. 6th. E.I.S., Workshop 1993*, pp. 208-217 34

[HiSi88] Hirsch, M., Siewiorek, D., “Automatically Extracting Structure from a Logical Design”, *Proc. ICCAD 1988*, pp. 456-459

[Hwan79] Hwang, K., “Computer Arithmetic”, Wiley & Sons 1979, p. 121 8.3.1

[IsYo87] Ishikawa, M., Yoshimura, T., “A New Module Generator with Structural Routers and a Graphical Interface”, *Proc. ICCAD 1987*, pp. 436-439 3.1.1

[King84] Kingsley, C., “A Hierarchical, Error-Tolerant Compactor”, *Proc. 21st DAC 1984*, pp. 126-132 3.1.1

[KoCS90] Kong, J., Chan, P.K., Schlag, M., “EQN2XNF”, *Manual Page*, UC Santa Cruz, Computer Engineering, 1990 30, 8.1

[KoGo94] Koch, A., Golze, U., “A Universal Co-Processor for Workstations” in *More FPGAs*, ed. by Moore, W., Luk,W., Oxford 1994, pp. 317-328 2, 2.3.2

[Koch96a] Koch, A., “Structured Design Implementation - A Strategy for Implementing Regular Datapaths on FPGAs”, *Proc. 4th International Symposium on FPGAs 1996*, pp. 151-157 32

[Koch96b] Koch, A., “Module Compaction in FPGA-based Regular Datapaths”, *Proc. 33rd DAC 1996*, p. 471-476 6, 32

[Koch97a] Koch, A., “Objekt-orientierte Modellierung von hybriden Hardware-Software-Systemen am Beispiel des “European Home System” (EHS) Standards”, *Proc. 8. E.I.S. Workshop 1997*, pp. 204-212 34

[Koch97b] Koch, A., “Practical Experiences with the SPARXIL Co-Processor”, *31st Asilomar Conference on Signals, Systems, and Computers*, 1997 2

[Law85] Law, H.S., et al., “An Intelligent Composition Tools for Regular and Semi-Regular VLSI Structures”, *Proc. ICCAD 1985*, pp. 169-171 3.1.1

[Law96] Lawrence, A.E., “Macro Support for Xilinx Architecture”, *Draft, ftp://ftp.ox.ac.uk/pub/users/adrian/xmacros.ps.gz,* Oxford Hardware Compilation Group 1996 3.1.2, 3.1.3, 3.3.5

[LeWE96] Legl, C., Worth, B., Eckl, K., “A Boolean Approach to Performance-Directed Technology Mapping for LUT-Based FPGA Designs”, *Proc. 33rd DAC 1996*, p. 730-733 6.5.1, 8.1

[Leng86] Lengauer, T., “Exploiting Hierarchy in VLSI Design”, *Proc. Aegean Workshop on Computing*, LNCS Vol. 227, Springer 1986 2.2, 20

[Leng90] Lengauer, T., “Combinatorial Algorithms for Integrated Circuit Layout”, Wiley-Teubner 1990 5

[LeTa93] Lee, M.A., Takagi, H., “Dynamic Control of Genetic Algorithms using Fuzzy Logic Techniques”, *Proc. 5th ICGA 1993*, pp. 76-83

[Lin97] Lin, Y.-L., “Recent Developments in High-Level Synthesis”, *ACM Trans. on Design Automation of Electronic Systems*, Vol. 2, No. 1, January 1997, pp. 2-21 34

[LuDS95] Lu, A., Dagless, E., Saul, J., “DART: Delay and Routability Driven Technology Mapping for LUT Based FPGAs”, *Proc. ICCD 1995*, pp. 409-414

[Marp90] Marple, D., "A Hierarchy Preserving Hierarchical Compactor", *Proc. 27th DAC 1990*, pp. 375-381 2.6.5

[MaJO96] Marnane, W.P., Jordan, C.N., O'Reilly, F.J., "Compiling Regular Arrays onto FPGAs", in *Field-Programmable Logic and Applications, Proc. 5th FPL 1995*, ed. by Moore, W., Luk, W., Springer 1996, pp. 179-187 3.1.3, 3.1.3, 3.2.1, 3.3.5

[MaWa90] Matsumoto, N., Watanabe, Y., et al., "Datapath Generator Based on Gate-Level Symbolic Layout", *Proc. 27th DAC 1990*, pp. 388-393 3.1.1

[Maye72] Mayeda, W., "Graph Theory", Wiley 1972 5

[McWi78] McWilliams, T.M., Widdoes, L.C., "SCALD: Structured Computer-Aided Logic Design", *Proc. 15th DAC 1978*, pp. 271-277 3

[Meye97] Meyer, K., "Entwurf eines FPGA-basierten Co-Prozessors zur Objekt-Etikettierung in der Bilderkennung", *Diploma Thesis*, TU Braunschweig, Abt. E.I.S., 1997 2

[MSBS91a] Murgai, R., Shenoy, N., Brayton, R.K., Sangiovanni-Vincentelli, A., "Performance Directed Synthesis for Table Look Up Programmable Gate Arrays", *Proc. ICCAD 1991*, pp. 572-575 8.1

[MSBS91b] Murgai, R., Shenoy, N., Brayton, R.K., Sangiovanni-Vincentelli, A., "Improved Logic Synthesis Algorithms for Table Look Up Architectures", *Proc. ICCAD 1991*, pp. 564-567 2.7.3

[Murg93] Murgai, R., "Logic Synthesis for Field-Programmable Gate Arrays", *Ph.D. Thesis*, UC Berkeley 1993 3.3.7

[MuBS95] Murgai, R., Brayton, K., Sangiovanni-Vincentelli, A., "Logic Synthesis for Field-Programmable Gate Arrays", Kluwer 1995 5, 6.5

[Mukh86] Mukherjee, A., "Introduction to nMOS & CMOS VLSI Systems Design", Prentice Hall 1986

[NaBK95] Naseer, A.R., Balakrishnan, M., Kumar, A., "Delay Minimal Mapping of RTL Structures onto LUT Based FPGAs", in *Field-Programmable Logic and Applications, Proc. 5th FPL 1995*, ed. by Moore, W., Luk, W., Springer 1996, pp. 139-146 2.2

[NaKK94] Nauk, D., Klawonn, F., Kruse, R., "Neuronale Netze und Fuzzy-Systeme", Vieweg 1994 2.5

[NZKK94] Noffz, K-H., Zoz, R., Kugel, A., Klefenz, A., Männer, R., "Die Enable Machine – Ein Echtzeitmustererkennungssystem auf FPGA-Basis", *Proc. GI/ITG Workshop "Architekturen für hochintegrierte Schaltungen" 1994*, ed. Hartmut Schmeck, Universität Karlsruhe (AIFB), Report 303, pp. 67-69 1

[OdHN87] Odawara, G., Hiraide, T., Nishina, O., “Partitioning and Placement Technique for CMOS Gate Arrays”, *IEEE Trans. on CAD*, Vol. CAD-6, No. 3, May 1987, pp. 355-363 2.2

[Omon94] Omondi, A.R., “Computer Arithmetic Systems”, Prentice-Hall 1994, p. 154 3.2.1

[Page95] Page, I., “Constructing Hardware-Software Systems from a Single Description”, submitted to *VLSI Signal Processing*, ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Ian.Page/hwsw.ps.gz, July 1995 3.1.2

[Pmel96] Pilkington Micro-electronics Ltd., “Pilkington FPGA Preliminary Data”, *Data Sheet*, Northwich Cheshire 1995 2.3.1

[PBLS91] Pangrle, B.M., Brewer, F.D., Lobo, D.A., Seawright, A., “Relevant Issues in High-Level Connectitvity Synthesis”, *Proc. 28th DAC 1991*, pp. 607-610 3.3.6

[Putz95a] Putzer, H., “Ein fuzzy-gesteuerter Genetischer Algorithmus mit Anwendungsmöglichkeiten auf das Plazierungsproblem bei FPGA-Chips”, *Diploma Thesis*, TU Braunschweig, Abt. E.I.S., 1995 2.2, 2.5, 4

[Putz95b] Putzer, H., “Ein Fuzzy-Gesteuerter Genetischer Algorithmus mit Anwendungsmöglichkeiten auf das Plazierungsproblem bei FPGA-Chips”, *7. E.I.S. Workshop 1995*, pp. 265-269

[Qds96a] Quickturn Design Systems Inc, “Hughes Relies on Emulation To Verify New Telecommunications Architecture: billions of vectors emulated in minutes”, *http://www.quickturn.com/prod/success/hughes.htm*, Mountain View 1996 1, 1, 1

[Qds96b] Quickturn Design Systems Inc, “Quickturn Emulator Dramatically Cuts Debug Time for New Sun Microsystems’ Processor”, *http://www.quickturn.com/prod/success/Sun.htm*, Mountain View 1996 1, 1

[Qds96c] Quickturn Design Systems Inc, “System Realizer Family of Modular Emulation Systems”, *http://www.quickturn.com/prod/realizer/realizer.htm*, Mountain View 1996 1

[Quic95] QuickLogic Corp., “pASIC2 FPGA Family Technology White Paper”, *Tech Report*, Santa Clara 1995 2.3.1

[Raba85] Rabaey, J.M. et al., “An integrated automated layout generation system for DSP circuits”, *IEEE Trans. on CAD*, Vol. 4, No. 7, July 1985, pp. 285-296 2.3.1, 7

[RoSe96] Roy, K., Sechen, C., “A Timing-Driven Partitioning System for Multiple FPGAs”, *VLSI design*, Vol. 4, No. 4, 1996, pp. 309-328

[RuPS91] Ruppeiner, G., Pedersen, J.M., Salamon, P., “Ensemble approach to simulated annealing”, *Journal de Physique I*, **1** (1991), pp. 455-470 10, 7.4.1, 7.4.3, 8.1

[SaCh94] Saab, Y., Cheng-Hua Chen, “An Effective Solution to the Linear Placement Problem”, *VLSI Design*, Vol. 2, No. 2, pp. 117-129 2.5, 10

[Sade95] Sadewasser, H., “Parametrisierbare Modulgeneratoren für die FPGA-Familie Xilinx XC4000: Logikfunktionen Schieberegister und Multiplizierer”, *Diploma Thesis*, TU Braunschweig, Abt. E.I.S., July 1995 2.4, 4, 3.1.3, 3.3, 4.7.1, 4.7.4

[ScKC94] Schlag, M., Kong, J., Chan, P.K., “Routability-Driven Technology Mapping for Lookup Table-Based FPGA’s”, *IEEE Trans. on CAD*, Vol. 13, No. 1, January 1994, pp. 13-26 34

[SeSa85] Sechen, C., Sangiovanni-Vincentelli, A. “The TimberWolf placement and routing package”, *IEEE J. Solid-State Circuits*, SC-20(2), pp. 510-522, 1985

[Sent92] Sentovich, E.M. et al., “SIS: A System for Sequential Circuit Synthesis”, *Electr. Res. Lab. Memo No. UCB/ERL M92/41*, Dept. of EE and CS, UC Berkeley 4 May 1992 3.3.7, 6.5.1, 8.1, 8.4.1

[ShBh96] Shi, J., Bhatia, D., “Macro Block Based FPGA Floorplanning”, submitted for *Proc. ICCD 1996*

[Sher95] Sherwani, N., “Algorithms for VLSI Physical Design Automation, 2nd ed.”, Kluwer 1995 5

[Shro82] Shrobe, H.E., “The data path generator”, *Proc. Conf. on Adv. Research in VLSI 1982*, pp. 175-181 2.3.1

[Shun89] Shung, C.S. et al., “An Integrated CAD System for Algorithm-Specific IC Design”, *Proc. Intl. Conf. on System Design 1989*

[SrPa94] Srinivas, M., Patnaik, L.M., “Genetic Algorithms: A Survey”, *IEEE COMPUTER*, June 1994, pp. 17-26 4

[Sung83] Sungho Kang, “Linear Ordering and Application to Placement”, *Proc. 20th DAC 1983*, pp. 457-464 2.5

[Syno96a] Synopsys Inc., “Design Compiler Reference Manual”, *EDA Software Documentation*, Mountain View 1996 6.5.1, 8.1, 34

[Syno96b] Synopsys Inc., “DesignWare Developer Guide”, *EDA Software Documentation*, Mountain View 1996 34

[SYYH92] Suzuki, G., Yamamoto, T., Yuyama, K., Hirasawa, K., “MOSAIC: A Tile-Based Datapath Layout Generator”, *Proc. ICCAD 1992*, pp. 166-169 3.1.1

[Texa97] Texas Instruments Corp., “TGC6000 Gate Array Specification”, *http://www.ti.com/sc/docs/asic/gate/tgc6000.htm*, 1997 2

[View94a] Viewlogic Systems Inc., “Using Powerview”, *EDA Software Documentation*, Marlboro 1994 30, 8.1

[View94b] Viewlogic Systems Inc., “Viewgen Reference Manual”, *EDA Software Documentation*, Marlboro 1994 30, 8.1

[Will93] Williams, H.P., “Model Building in Mathematical Programming”, Wiley 1993 7.2, 26, 7.3.2

[Xili94a] Xilinx Inc., “X-BLOX User Guide”, *EDA Software Documentation*, San Jose 1994 2.4, 3.1.2, 3.1.3, 8.1, 8.5.2

[Xili94b] Xilinx Inc., “XACT Reference Guide, Vol. I: XACT Performance”, *EDA Software Documentation*, San Jose 1994 8.1

[Xili94c] Xilinx Inc., “XACT Reference Guide, Vol. II: PPR”, *EDA Software Documentation*, San Jose 1994 2.8, 3.1.2, 8.1

[Xili94d] Xilinx Inc., “XACT Reference Guide, Vol. III: XDELAY”, *EDA Software Documentation*, San Jose 1994 8.1

[Xili94e] Xilinx Inc., “The Programmable Logic Data Book, 2nd ed.”, *Databook*, San Jose 1994 4.7.1

[Xili94f] Xilinx Inc., “XACT Xilinx Synopsys Interface FPGA User Guide”, *EDA Software Documentation*, San Jose 1994 8.1, 8.5.2

[Xili95a] Xilinx Inc., “Xilinx Netlist Format (XNF) Specification”, *EDA Software Documentation*, San Jose 1995 30

[Xili95b] Xilinx Inc., “Viewlogic Interface Guide”, *EDA Software Documentation*, San Jose 1995 30, 8.1

[Xili96a] Xilinx Inc., “XC5200 Field Programmable Gate Arrays, Version 4.01”, *Datasheet*, San Jose 1996 2, 2.7.3

[Xili96b] Xilinx Inc., “XC6200 Field Programmable Gate Arrays, Version 1.0”, *Datasheet*, San Jose 1996 2, 2.3.1, 2.7.3

[Xili96c] Xilinx Inc., “XC4000 Field Programmable Gate Arrays”, *Datasheet*, San Jose 1996 2

[Xili96d] Xilinx Inc., “FPGAs Go Down Under in an ISDN Terminal Adapter”, *XCELL*, No. 21, pp. 6-7, San Jose 1996 1

[Xili97] Xilinx Inc., “The Future of FPGAs”, *White Paper*, San Jose 1997 2, 1

[YuWY93] Yu-Wen, T., Wu, A.C.H, Youn-Long, L., “A Cell Placement Procedure That Utilizes Circuit Structural Properties”, *Proc. EDAC 1993*, pp. 189-193 2.2

Abbreviations

ALU Arithmetic-Logic Unit

ASIC Application-Specific Integrated Circuit

BPC Bits-Per-CLB

BPLB Bits-Per-Logic Block

CAD Computer Aided Design

CLB Configurable Logic Block

CPU Central Processing Unit

DNA Deoxyribonucleic Acid

DSP Digital Signal Processor

EBSA Ensemble-Based Simulated Annealing

ECO Engineering Change Order

EDA Electronic Design Automation

EDIF Electronic Design Interchange Format

EEPROM Electrically Erasable Read-Only Memory

EPROM Erasable Programmable Read-Only Memory

FCCM Field-Programmable Custom Computing Machine

FF Flipflop

FPGA Field-Programmable Gate Array

GA Genetic Algorithm

GAL Generic Array Logic

HDL Hardware Description Language

HLL Horizontal Long Line

HZ H-Zone

ILP Integer Linear Program

IOB Input/Output Block

LB Logic Block

LCA Logic Cell Array

LPGA Laser-Programmable Gate Array

LPM Library of Parametrized Macros

LSB Least-Significant Bit

LUT Lookup Table

MIS Multilevel Interactive Synthesis

mmS Merged Master-Slice

MPGA Mask-Programmable Gate Array

MS Master Slice

MSB Most-Significant Bit

MSC Master-Slice Candidate

NRE Non-Recurring Engineering

omS Optimized Master-Slice

OPBDP Optimization by Pseudo-Boolean Davis-Putnam enumeration

PAL	Programmable Array Logic
PCB	Printed Circuit Board
PE	Processing Element
PGA	Programmable Gate Array
PI	Primary Input
PIP	Programmable Interconnection Point
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PO	Primary Output
PPR	Partition Place Route
PROM	Programmable Read-Only Memory
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
RLCA	Relocatable Logic Cell Array
RLOC	Relative Location Constraint
rMSC	Raw Master-Slice Candidate
ROM	Read-Only Memory
RPM	Relationally Placed Macro
RTL	Register-Transfer Logic
SA	Simulated Annealing
SDI	Structured Design Implementation
SIS	Sequential Interactive Synthesis
SM	Switch Matrix
SNF	Simple Netlist Format
SPARC	Scalable Processor Architecture
SPLD	Simple Programmable Logic Device
SRAM	Static Random-Access Memory
TBUF	Tristate Buffer
TOS-TUM	Technology Oriented Synthesis - Tech. Univ. Munich
TTN	Two-Terminal Net
UI	Unit Input
UO	Unit Output
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VLL	Vertical Long Line
VZ	V-Zone
XNF	Xilinx Netlist Format

Curriculum Vitae

Name: Andreas Hartmut Koch
Born: 28 February 1968 in Frankfurt/M., Germany
Marital Status: Single

Education

1974 - 1978	Grundschule in Vechta (Oldenburg)
1978 - 1980	Orientierungsstufe in Vechta
1980 - 1987	Gymnasium Antonianum in Vechta
1984 - 1985	Student exchange to Columbus, Ohio, USA
09 June 1985	Graduation from Northland High School
25 May 1987	Abitur at Gymnasium Antonianum
1987 - 1992	Study of Computer Science at Technical University Braunschweig, Germany
26 June 1992	Diploma in Computer Science

Work Experience: Academic

1989 - 1991	Student developer at “SIEMENS Reference Center for Scientific Application Software” at the Institute for Operating Systems and Computer Networks, Technical University Braunschweig
1990	Student developer at the Department for VLSI Design, Technical University Braunschweig
1992 - 1997	Research Assistant at the Department for VLSI Design, Technical University Braunschweig

Work Experience: Commercial

1987 - 1988	Independent software developer
1988 - 1992	Head of Development Staff, Procedia GmbH, Hannover, Germany

Lebenslauf

Name: Andreas Hartmut Koch
Geboren am: 28. Februar 1968 in Frankfurt am Main
Familienstand: Ledig

Ausbildung

1974 - 1978	Grundschule in Vechta (Oldenburg)
1978 - 1980	Orientierungsstufe in Vechta
1980 - 1987	Gymnasium Antonianum in Vechta
1984 - 1985	Schüleraustausch nach Columbus, Ohio, USA
09.06.1985	US High School-Abschluß an der Northland High School
25.05.1987	Abitur am Gymnasium Antonianum
1987 - 1992	Studium der Informatik an der TU Braunschweig
26.06.1992	Diplom in Informatik

Berufstätigkeit im akademischen Bereich

1989 - 1991	Studentische Hilfskraft am “SIEMENS Referenzzentrum für Technisch-Wissenschaftliche Anwendungssoftware” am Institut für Betriebssysteme und Rechnerverbund der TU Braunschweig
1990	Studentische Hilfskraft an der Abteilung Entwurf integrierter Schaltungen der TU Braunschweig
1992 - 1997	Wissenschaftlicher Mitarbeiter an der Abteilung Entwurf integrierter Schaltungen der TU Braunschweig

Berufstätigkeit im kommerziellen Bereich

1987 - 1988	Unabhängiger Software-Enwickler
1988 - 1992	Leiter der Entwicklungsabteilung der Procedia GmbH, Hannover,