Structured Design Implementation — A Strategy for Implementing Regular Datapaths on FPGAs

Andreas Koch
Department for Design of Integrated Circuits, Tech. Univ. of Braunschweig, Germany
koch@eis.cs.tu-bs.de

SDI is a strategy for the efficient implementation of regular datapaths with fixed topology on FPGAs. It employs parametric module generators, a floorplanner based on a genetic algorithm, and a circuit compaction phase through local technology mapping and placement by ILP models. Initial results promise faster layouts than with general algorithms.

1 Introduction

With increasing FPGA sizes, applications exceeding simple glue logic have become possible. Current FPGAs have sufficient capacity to accommodate circuits providing CPU and DSP functions. However, most CAD tools have not yet been modified to efficiently cope with the different requirements these designs pose.

We examine various ways to speed-up the performance of 16–32 bit datapaths common to such CPU and DSP circuits. This paper presents an overview of the multi-component "strategy" Structured Design Implementation (SDI) (Figure 1), which aims to better utilize regular structures in the circuits during technology-mapping and placement.

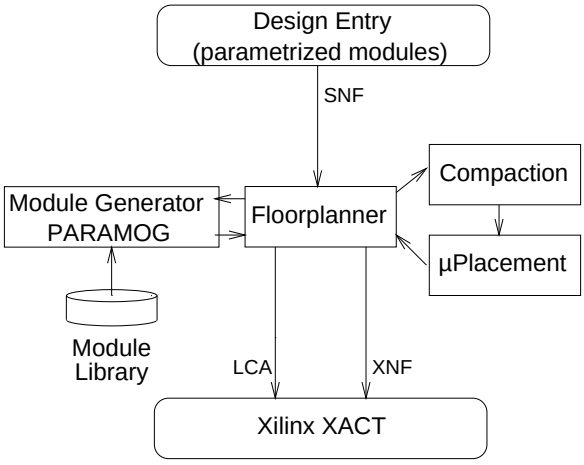

Fig. 1: SDI overview

While the strategy can easily be applied to all FPGAs with a matrix structure (e.g., AT & T ORCA), SDI was developed to accelerate the reconfigurable FPGA-based co-processor SPARXIL [Koc94], and therefore currently targets XC4000 CLBs.

2 Previous Work

In order to compare and contrast SDI with previous work, a look at non-FPGA specific methods for handling regular structures is helpful. They can roughly be classified by two categories: on the one hand, there are algorithms which extract regularity information from a flat or hierarchical netlist after its creation. The *macros* recognized in this manner are then treated specially during placement. Such extraction-based algorithms are described in [Oda87], [Hir88], and [Yuw93]. The placement of the extracted macros is then performed in a force-directed ([Oda87], [Chi93]) or by simulated annealing [Yuw93].

On the other hand, an older approach is based on employing regular *modules* already during design entry. The user-defined modules are then placed as whole units and not merged with the remainder of the circuit. The methods described in [Cai90] and [Ben93] are representative for many similar approaches. In these algorithms, primitive (AND, OR, INV, MUX) as well as more complex modules (ADD, REG, SHIFT) are generated according to specified operand and result widths and pin arrangement. The automatically created modules are then placed linearly, e.g., in [Cai90] by an A^*-algorithm based on a modified branch-and-bound method.

SDI falls into the last category: design entry and floorplanning occur on the module level. In contrast to the tools mentioned above, however, SDI modules can be broken down into their constituents in order to efficiently map adjacent modules into the target logic blocks of the target FPGA. The unified function of the merged modules is mapped locally and *micro-placed*, preserving the regular structure of the datapath. This procedure differs from the previous approaches to placement-oriented technology-mapping ([Mur91], [Cha93]), which disregard circuit regularities.

The module generator component PARAMOG ([Sad95], [Dit95], Figure 1) of SDI is implemented similarly to classical examples ([Shu89], [Ben93]). Requests are parametrized by data types, bit widths, and area limits, and PARAMOG offers a selection of possible layouts realizing the specified function. Unlike the more complex LORTGEN system [Bra94], PARAMOG does not have an extensive internal knowledge base and does not evaluate the quality of the proposed implementations. These tasks are handled by the floorplanning component FLOORPLANNER [Put95] of SDI. In contrast to LORTGEN, PARAMOG generates pre-placed and pre-routed modules (*hard macros*) that fully use FPGA-specific features such as the hard-carry logic of the XC4000 chips. Such specialized tools for FPGAs have only recently begun to appear, examples include Xilinx X-BLOX and the Atmel module generators.

It should be pointed out again that SDI does not consist of a single tool, but a suite of tools and a strategy for their use. 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 standalone tools that cover only part of the design implementation process (e.g., the ASYL synthesis and mapping tool [Bab92]). However, these tools can often be integrated into SDI with minimal effort (see Section 7).

Class	Functions
Logic	NOT, AND, NAND, OR, NOR, XOR, XNOR, MUX
Shift and Rotate	LSHIFTA, RSHIFTA, LSHIFTL, RSHIFTL, LROT, RROT
Storage	REG, RAM, ROM, CONST
Arithmetic	ABS, COMPL1, COMPL2, INCDEC, ADDSUB, MULT
Comparison	COMPARE
Counter	COUNT

Table 1: SDI module library overview

3 Structured Design Entry

Datapaths are entered into SDI in the form of parametrized modules. Thus, information on the logical structure of the circuit can be passed down from design entry (e.g., a schematic) to the placement and routing tools and does not have to be reconstructed from an unstructured, possibly flattened netlist. The preserved regular structure can be used to optimize the circuit as well as the internal operation of the CAE algorithms.

Currently, designs are expressed in the SDI netlist format *SNF*, which is a textual netlist of module declarations, module instantiations, and interconnections. Furthermore, it associates values with module parameters such as bus widths, data types, and optimization requests (speed vs. area). Interfaces and netlist converters from higher-level front ends such as schematic editors and structural HDL are presently being investigated.

Table 1 lists the modules available to the designer or synthesis tools. Note that these are generic modules: for example, the ADDSUB module can generate layouts for adders, subtractors, and adder-subtractors depending on the presence of an AddSub control input.

Furthermore, the modules use FPGA architecture-specific features. For example, many arithmetic modules employ the hard-carry logic of the XC4000 chips and memories are implemented by directly configuring CLBs as RAM or ROM.

4 Target Topology

The chip topology targeted by SDI is characterized by a fixed three-partite layout (Figure 2). The large middle section holds the regular part of the datapath. This part consists of a horizontal arrangement of modules, each composed of vertically stacked bit-slices. The stacking order from bottom to top usually reflects an LSB to MSB orientation (this may change within folded modules). 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 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 adder's LSB 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 each area are application-specific and must be designated by the user. In the case of FPGAs with a fixed pin-out on a PCB (such as the SPARXIL processor), the area heights are primarily determined by the interconnection pattern on the PCB.

The SDI chip topology can accommodate multiple data flows on a single FPGA as long as their total width remains smaller than the FPGA width. All data flows can gain access to the external data

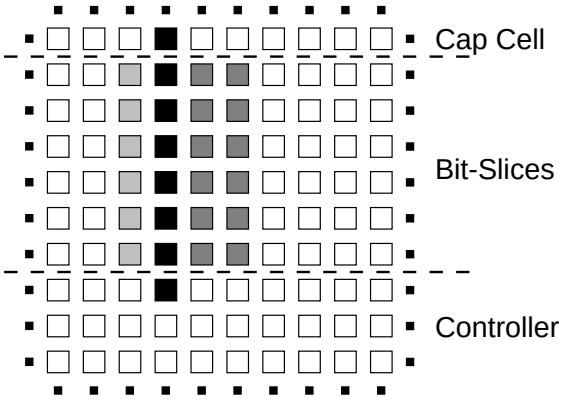

Fig. 2: On-chip topology

I/O pads using the horizontal long lines (HLLs) available on Xilinx XC4000 FPGAs.

The routing structure also follows classical lines by organizing the data flow horizontally and the control flow vertically. Thus, high-fanout control signals can be routed efficiently on vertical long lines (VLLs) with minimum skew, while the HLLs remain available for tri-state data busses and long-distance connections.

One important parameter characterizing the whole datapath is the number of bits processed in each logic block (*BPC*, see Figure 3). It should be homogeneous over all modules, otherwise the regular structure of the datapath will be severely disturbed since excessive routing has to take place to compensate. For Xilinx CLBs, the BPC is usually one or two (since an XC4000 CLB has two independent LUT outputs). The BPC number might be different inside a module (e.g., when LUTs and FFs are used independently), but the external interface will normally have just one of the two possible BPC values.

Figure 4 shows two examples for modules with a 2 BPC topology. Note that n-bit gates like the NOR16 in (b) can be realized with 2 BPC in $\log_4 n$ levels of LUTs, occupying only a single column of LUTs for $n \leq 32$.

The topology was already considered for the hardware design and PCB layout of the SPARXIL processor. The 20x20 matrix of the XC4010 FPGAs is vertically organized with one CLB as cap cell, 16 CLBs for the datapath (yielding a maximum datapath width of 32 bits with unfolded 2 BPC modules) 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.

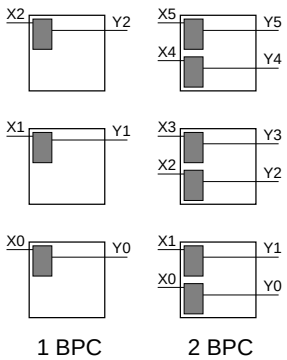

Fig. 3: Bits processed per logic block

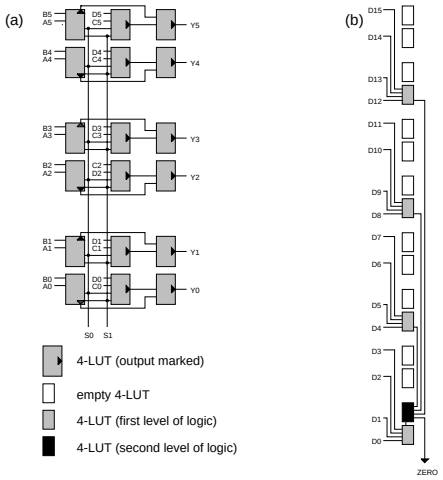

Fig. 4: 2 BPC Modules: (a) 6-bit 4-1 Mux, (b) 16-bit NOR

5 Module Generation

FLOORPLANNER requests layout alternatives from PARAMOG according to the information in the SNF file. PARAMOG processes the user-specified parameters for each module instance and prepares a list of possible layouts with different topologies, which it returns to FLOORPLANNER.

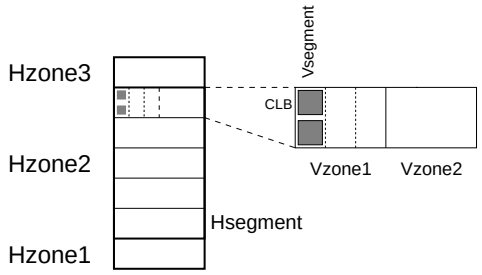

Fig. 5: Regular structure of modules

Module generation is a process also based on regular structures (Figure 5). A module consists of one or more different zones with the same X origin (*hzones*), which in turn are composed of one or more different zones with the same Y origin (*vzones*). Each hzone or vzone may be replicated producing one or more *h- or v-segments* of the module. Vzones contain the most basic building blocks (CLBs for the Xilinx XC4000 FPGAs). Hzones are stacked vertically bottom to top, while vzones are arranged horizontally left to right inside an hzone. Thus, the zones describe the kinds of different logic inside the module, while the segment configuration forms its physical structure (location and extents of an area of similar logic). The example module in Figure 5 is horizontally composed of three areas with different logic (hzones 1 ... 3). The logic described by hzone 2 is replicated in five hsegments. Hzone 2 itself consists of two different areas of logic (vzones 1 and 2). Vzone 1 of hzone 2 describes the configuration and routing of two CLBs. The contents of this vzone are replicated thrice as three vsegments and make up the first half of this slice of the module. This organization allows the efficient description even of complex layouts.

Figure 6 shows different layouts PARAMOG proposes for various left shift registers with different widths and BPC parameters. Note that a module consists of placement and routing information, with the routing also specific to the targeted FPGA. As can be seen, the

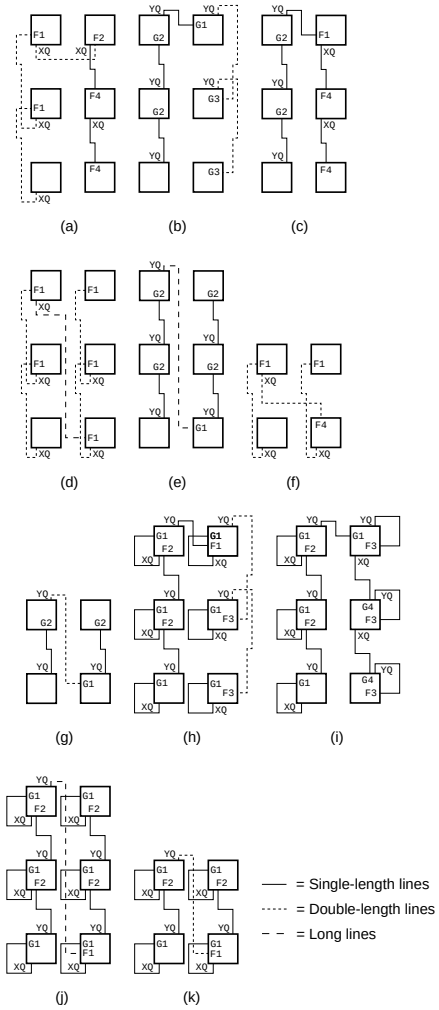

Fig. 6: Layouts for arithmetic left shift registers. (a) - (e) 6 bits, 1BPC; (f) - (g) 4 bits, 1BPC; (h) - (j) 12 bits, 2BPC; (k) 8 bits, 2BPC

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). The bit ordering of the datapath can be specified by the user: e.g., layout (e) in Figure 6 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.

6 Module Selection and Placement

Now that FLOORPLANNER has read the available layout topologies for all module instances of the datapath, it begins to linearly place instances in the regular region of the FPGA. During this process, different concrete layouts (as suggested by PARAMOG) are selected and evaluated in context.

FLOORPLANNER is based on a fuzzy-controlled genetic algorithm, and thus considers various different layout choices and placements simultaneously. The fuzzy-controller is responsible for 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. Since the solution

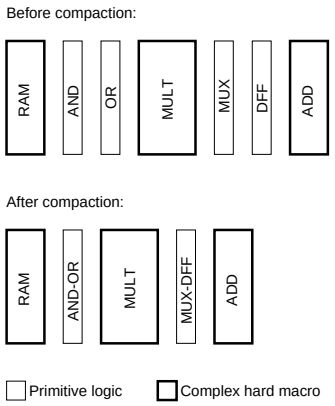

Fig. 7: Compaction of a linear placement

space consists only of placing complete module instances (usually far less than two dozens per FPGA) and selecting their configuration (around three to four after BPC normalization), it 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). Thus, exploitation of the regular structure allows the use of a computationally expensive, but powerful algorithm.

The fitness function of FLOORPLANNER mainly considers the following factors in its search for a high-quality chip layout: the uniformity of the instance BPC values, the wire length on the critical path (evaluating the FPGA's different routing resources with specific delay models) and the compactibility of adjacent modules.

7 Compaction

When FLOORPLANNER has finished its work and created a suitable linear placement of module instances in the datapath area, a compaction phase merging adjacent primitive modules is initiated.

Figure 7 justifies the need for compaction in module-oriented systems: since each component of the datapath is considered a module, and a module layout has a horizontal extent of at least one CLB, much space and time is wasted in multi-level logic networks when the logic of multiple levels could fit inside a single CLB. In order to prevent this, adjacent *primitive* logic modules should be merged together. In this context, primitive means modules either not employing architecture-specific features (such as hard-carry or RAM/ROM CLBs), which cannot be processed by conventional logic minimization and mapping tools, or large hand-optimized layouts (such as multipliers, dividers, and similar mega-cells). Such *complex* modules mark boundaries for the compaction process.

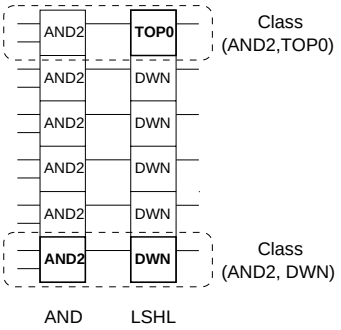

Fig. 8: Equivalence classes of module boundaries

While this partitioning of the circuit into areas of compaction creates smaller sub-problems, their size can be reduced even further by considering the regularity inherent in datapath modules. Figure 8 shows two adjacent primitive modules: a module implementing an AND operation on two busses with the result being fed into a logical shift left (LSHL) module. Since both of the modules are primitive, compaction is attempted. But due to the module regularity, the boundary between the AND and LSHL modules has only two areas with different logic: at one boundary area, an AND2 hzone meets a TOP0 hzone; all other hzones meeting at the boundary are of the AND2 and DWN variety. Thus, the minimization and mapping problem can be reduced to packing representatives for *equivalence classes* of logic at module boundaries. The results just have to be replicated across the width of the datapath to create a new module implementing the combined functions of the compacted modules.

Compaction itself is performed by merging all logic (across module boundaries) within an equivalence class and processing the resulting functions with classical logic synthesis and optimization tools. Thus, SDI can profit directly from advances in this field by integrating new algorithms as they become available. For example, the original version of SDI employed the minimization and partitioning algorithms present in SIS 1.3 [Sen92]. The current version, however, has integrated the more recent FLOWMAP [Con94] for partitioning purposes. Depending on the algorithm used, the compaction can emphasize delay or area reduction, but the classical techniques used do not consider regularity constraints. Thus, the existing regular placement within the class is lost during this operation, but the slice structure itself remains intact. A regular placement will be regenerated in the next step (see Section 8).

The compacted network has K-LUTs as its basic elements (for XC4000: K=4). At this time, the compaction makes no attempt to handle irregularities in the CLBs (e.g., the H-block). However, in light of a trend away from irregular structures in LUT-based FPGAs (e.g., the recent Altera FLEX and Xilinx XC5000 chips), this restriction seems less severe and could even be removed by integration of the appropriate CLB-packing algorithms.

The K-LUT can optionally be combined with a flip-flop, thus creating a registered output. For XC4000 FPGAs, the combination of 4-LUT and FF fills half of the regular part of a CLB and will be termed a *CLB-half*.

8 Micro-Placement

Since all placement information within an equivalence class is lost during minimization and partitioning, the CLB-half in the classes have to be re-placed. Note that the placement process does not place complete CLBs. It operates on a grid twice the CLB height of the placement area, thus performing CLB assignment of LUTs and FFs during the placement process. One of the primary criteria (apart from minimizing critical path length) for this placement is the restoration of a regular structure consistent with the one created by PARAMOG. Thus, the placement algorithm has to consider the following points:

1. critical path length
2. alignability between vertically connected adjacent slices
3. routing of control signals on VLLs
4. pin locations at the module boundaries

The micro-placement component of SDI is based on an integer linear programming (ILP) formulation of the problem. Since the algorithm also exploits the regularity, it has to deal only with a limited number of 16–20 CLB-halves and critical paths. The algorithm also considers signals which are locked to ports on multiple sides of the placement area. While the ILP's main aspects are generally

applicable to datapath placement and routing problems, it also has components that are specific to the target FPGA (mostly concerning special interconnection patterns). For example, on XC4000 FPGAs, it aims to route critical nets using direct connections between CLBs, avoiding switch matrices wherever possible. This has a direct influence on the assignment of the 4-LUTs in the network to one of the two 4-LUTs in a CLB.

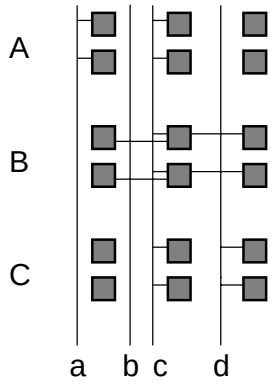

Fig. 9: Micro-placement of hzones A,B,C with control signals a,b,c,d

The algorithm runs in two phases. First, CLB-halves are assigned to columns. This considers all points just enumerated, but primarily optimizes the control signal routing. Apart from the CLB-half column locations, it generates global routing information for all control lines, replicating control signals in different columns if required. This first phase has to consider all equivalence classes of the merged module at once in order to perform the alignment of signals and VLLs crossing class boundaries.

The second phase performs row assignment within each column. Since it runs independently for each class (no inter-class dependencies exist at this point), it can employ a more detailed model of the target FPGA. It is this step that considers the effects of LUT selection on interconnect delays. Figure 9 shows the effect of micro-placing three classes of logic with four control signals. Note the alignment of LUTs accessing the same control VLL.

The placement step does not consider congestion in general. This seems feasible, since the pins on a CLB are interchangeable to a high degree. Thus, the pin assignment and routing phases can relieve congestion by swapping pins to less dense channels.

9 Pin Assignment and Routing

At present, these two steps are not covered by SDI, but have to be performed by the FPGA manufacturer's CAE tools (XACT by Xilinx). Design data is entered into XACT by two methods: compacted (minimized and micro-placed) modules with partitioning and placement information are transferred as relationally placed macros (*RPM*) in Xilinx netlist format *XNF*. This also allows using them as hard-macros in a schematic design, if a full datapath placement is not desired. PARAMOG-generated modules, on the other hand, contain routing data at the FPGA die level in addition to placement and partitioning information. Since XNF does not support that level of detail, the more arcane *LCA* format has to be used. Unfortunately, no easy method exists for linking a design that contains data in both formats. They have to be merged using the complicated route of the *guide design*. Currently, SDI does not automatically perform the required operations, but we expect to provide appropriate tools in the future.

With the regular datapath imported as XNF RPMs and LCA layout, the irregular controller consists of just a simple XNF netlist without placement or routing specifications. Both parts are then merged by the Xilinx tool *PPR* for partitioning, placement, and routing. The datapath is placed according to the SDI-generated location data, while the controller CLBs are processed by a simulated annealing based algorithm. All open nets (those not already routed in the LCA files) are then handled by the maze-router of PPR with a rip-up and retry extension.

The final result of this procedure is a bit-stream ready for downloading, combining regular and irregular elements in the manner desired in Section 4.

10 Parallel Processing

The exploitation of regularity in SDI has been designed with an eye towards the parallel execution of design steps on a workstation network or multi-processor machines. The current version of SDI is not parallel yet, but communication interfaces via FIFOs have already been built into the system. The following operations may be executed in parallel:

- requests to PARAMOG for each parametrized module
- finding equivalence classes in each compaction area
  - minimize and partition each equivalence class
- horizontal placement of each minimized compaction area
  - vertical placement of each eqv. class in comp. area

We do not expect the speed-up to scale linearly with the number of processors, however. The degree of parallelism achievable will mostly depend on the structure of the circuit being processed.

11 Example 1

Figures 11 and 12 show two layouts of the same circuit, one conventionally generated by the Xilinx tool PPR, the other one processed by our SDI. 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 10). In order to directly compare placement results, technology mapping and minimization have been disabled both in SDI and PPR. PPR was run with maximum optimization (placer_effort = 5) in performance-driven mode (dp2p) with all pads floating. Both SDI and PPR placements were routed by PPR, also using maximum optimization (router_effort = 4). Figure 11 shows

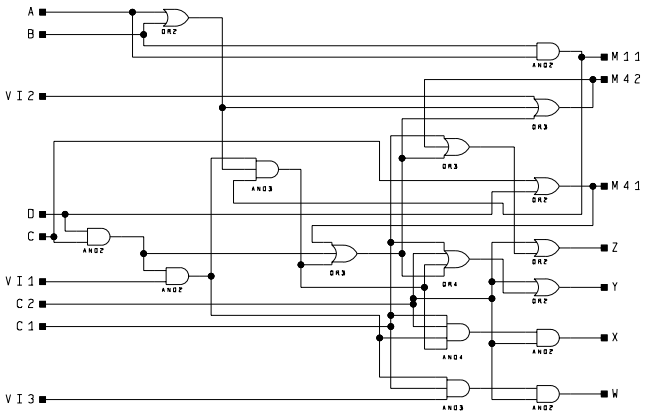

Fig. 10: Single bit-slice of the example circuit

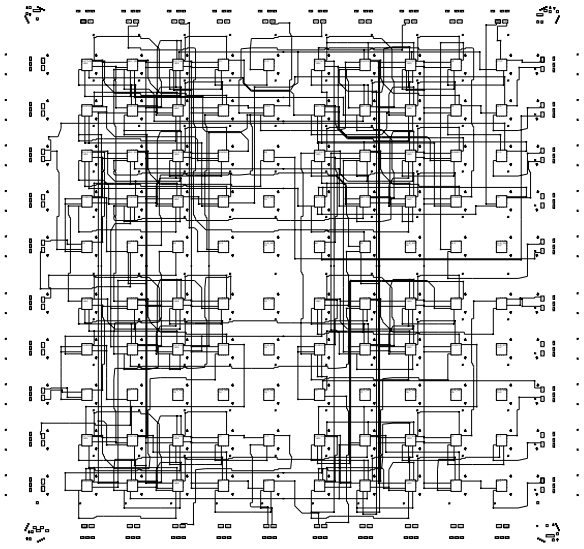

Fig. 11: Placement and routing solely by PPR

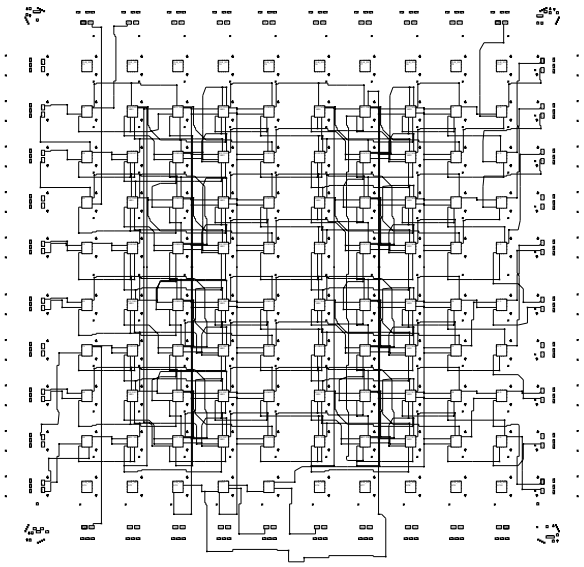

Fig. 12: SDI placement with PPR routing

the best layout generated after 15 PPR iterations, Figure 12 shows the results of a single SDI run.

Even at first glance, the SDI-generated solution is markedly more regular, since the natural structure of the datapath has been exploited. The SDI layout is less congested than the PPR one, especially in the first quadrant. Most of all, the routing delay in the critical path of the SDI solution is 13% shorter than in the maximally optimized PPR layout.

Each placement and routing iteration of PPR on an unloaded Sparc 20/71 workstation with 64MB RAM 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 140s on the same platform.

Quantity	XACT	SDI	
		FlowMap	SIS
4-LUTs in slice	?	26	24
CLBs in circuit	89	105	97
block levels in critical path			
mapped slice	?	4	5
placed circuit	20	15	15
routing delay			
best case	51.4ns	37.0ns	35.9ns
worst case	56.5ns	40.1ns	38.3ns
minimum clock period			
best case	158.9ns	115.4ns	115.5ns
worst case	168.8ns	118.6ns	117.2ns
execution time	1710s	7311s	925s
number of PPR runs	24	135	596

Table 2: Implementing a 32-bit ALU

12 Example 2

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. The starting point for both tools is a description of the ALU as shown in [Hwa79]. For SDI, mapping was performed by FlowMap and by SIS (using the script suggested for FPGAs in [Sen92]). The mapped netlists are placed by PPR or SDI, and all three circuits are routed by PPR. The designs target an XC4008PG191 chip.

Table 2 shows the results for a number of PPR runs. Since the placement ILP in SDI is exactly solved, only a single SDI placement needs to be executed per SDI mapping algorithm.

The following points seem interesting: while FlowMap achieves a shorter critical path at the logic level than SIS, 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 SIS solution.

When even more aggressive logic optimization methods employing computationally intensive methods such as kernel extraction (yielding a reduction to 20 LUTs per slice, 81 CLBs total) are applied, the clock speed will degrade even further (slower than 190ns) as the circuit becomes too dense to be routed efficiently.

In the context of SDI, the investigation of mapping algorithms, such as Rmap [Sch94] and DART [Lu95], which consider the routability of the resulting circuit, seems worthwhile.

13 Implementation

The implementation of PARAMOG and the module generator library in C++ is completed. The compaction algorithms have been integrated into UCB SIS. The micro-placement ILPs are pre-processed using a modified Davis-Putnam enumeration [Bar95] and solved using the commercial tool CPLEX [Cpl94]. FLOORPLANNER has also been implemented in C++, but the fuzzy rule base and the fitness function still need to be tuned. For many of the conversion steps tools have been provided as Perl scripts.

14 Conclusion

We have presented a specialized procedure for the efficient implementation of wide datapaths on FPGAs. While it is too early to make a final assessment, initial results appear promising. The evaluation of the system is complicated by the lack of a standard corpus of benchmark circuits for datapath structures, similar to the established suites for logic synthesis and standard cell layout.

15 References

[Ben93] Ben Ammar, L., Greiner, A., “A High Density Datapath Compiler Mixing Random Logic with Optimized Blocks”, *Proc. EDAC 1993*, pp. 194-198

[Bab92] Babba, B., Crastes, M., Saucier, G., “Input driven synthesis on PLDs and PGAs”, *Proc. EDAC 1992*, pp. 48-52

[Bar95] 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

[Bra94] 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

[Cai90] 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

[Cpl94] CPLEX Optimization Inc., “Using the CPLEX Callable Library”, *User Manual*, Incline Village (NV) 1994

[Cha93] Chau-Shen, C., Yu-Wen, T., “Combining Technology Mapping and Placement for Delay-Optimization in FPGA Designs”, *Proc. ICCAD 1993*, pp. 123-127

[Chi93] Chih-Liang, E.C., Chin-Yen, H., “SEFOP: A Novel Approach to Data Path Module Placement”, *Proc. ICCAD 1993*, pp. 178-181

[Con94] 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

[Dit95] 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

[Hir88] Hirsch, M., Siewiorek, D., “Automatically Extracting Structure from a Logical Design”, *Proc. ICCAD 1988*, pp. 456-459

[Hwa79] Hwang, K., “Computer Arithmetic”, Wiley & Sons 1979, p. 121

[Koc94] Koch, A., Golze, U., “A Universal Co-Processor for Workstations” in *More FPGAs*, ed. by Moore, W., Luk,W., Oxford 1994, pp. 317-328

[Lu95] Lu, A., Dagless, E., Saul, J., “DART: Delay and Routability Driven Technology Mapping for LUT Based FPGAs”, *Proc. ICCD 1995*, pp. 409-414

[Mur91] 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

[Oda87] 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

[Put95] Putzer, H., “Ein fuzzy-gesteuerter Genetischer Algorithmus mit Anwendungsmöglichkeiten auf das Plazierungsproblem bei FPGA-Chips”, *7. E.I.S. Workshop 1995*, pp. 265-269

[Sad95] Sadewasser, H., “Parametrisierbare Modulgeneratoren für die FPGA-Familie Xilinx XC4000: Logikfunktionen Schieberegister und Multiplizierer”, *Diploma Thesis*, TU Braunschweig, Abt. E.I.S., July 1995

[Sch94] 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

[Sec85] Sechen, C., Sangiovanni-Vincentelli, A. “The TimberWolf placement and routing package”, *IEEE J. Solid-State Circuits*, SC-20(2), pp. 510-522, 1985

[Sen92] 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

[Shu89] Shung, C.S. et al., “An Integrated CAD System for Algorithm-Specific IC Design”, *Proc. Intl. Conf. on System Design 1989*, Hawaii

[Yuw93] 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