Module Compaction in FPGA-based Regular Datapaths

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

When relying on module generators to implement regular datapaths on FPGAs, the coarse granularity of FPGA cells can lead to area and delay inefficiencies. We present a method to alleviate these problems by compacting adjacent modules using structure extraction, local logic synthesis, and cell replacement. The regular datapath structure is exploited and preserved, achieving faster layouts after shorter tool run-times.

1 Introduction

Regular datapaths are the core of many CPU and DSP architectures. The application of generator programs to create their constituent modules has a long history in VLSI design ([2], [6], [10], [13], [14], and many others). With growing FPGA die sizes, such datapath architectures are also implementable on FPGAs. However, current module generation techniques for FPGAs ([5], [19], [1]) do not address the area and delay inefficiencies caused by the coarse-grain architecture of FPGAs as compared to semi-custom or gate-array chips. Furthermore, misfeatures of current module generators include limited layout topology options [1] and the inability to regularly place simple non-FPGA-specific logic [19].

The following paper presents a method that mitigates these inadequacies: A linear placement of generated modules with regular layouts is compacted without disrupting the efficient structure, regardless of whether the modules are FPGA-specific or simple. The datapath regularity of horizontal data and vertical control flow is actively exploited and has been implemented in the framework of SDI [12]. SDI consists of a complete suite of tools (a comprehensive library of parametric modules, module generators, a floorplanner, and the compactor) and a strategy for their application to implement an efficient datapath combined with an irregular controller. The tools are currently targeting Xilinx XC4000 FPGAs. However, the general procedure can be applied to all FPGAs with matrix architecture. This paper describes only the compaction step, which processes just the regular part of the circuit.

2 Problem Description

A strictly module-based layout consists of a regular (often linear) placement of regularly generated modules. Since a module is always at least one logic block wide, partially utilized blocks 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 1 is an example for such a scenario: The 3-bit datapath contains three regular modules AND2, OR2, and AND2B1, implementing the functionality of a 3-bit wide 2-1 multiplexer. However, even assuming relatively 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, the sample datapath wastes 2/3 of its area and only runs at 1/2 the speed of the single block solution. This situation becomes worse with coarser-grained blocks such as the N-LUTs

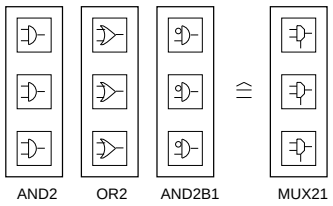


Fig. 1: Wasted space in a strictly module-based layout

found, e.g., in Xilinx XC3000/XC4000, AT&T ORCA, and Altera FLEX FPGAs. The compaction process breaks module boundaries in a strictly module-based layout and merges adjacent modules to better utilize the logic blocks.

3 Overview

In our approach, a circuit is composed of regular modules (vertical stacks of bit-slices ordered from bottom LSB to top MSB), that are placed in a regular linear arrangement by the floorplanner. Only after considering this initial floorplan, a flattening, minimization, and mapping process is performed to compact adjacent modules, reducing area and delay. Each of these operations preserves regular structures. The following placement of blocks within the compacted module also aims to create bit-slices suitable for vertical abutment and horizontal fit in the context of the initial floorplan. Observe that this approach aims at the compaction of entire sub-datapaths. This contrasts, for example, with the method in [18] for compacting single modules during their generation.

Conventionally, a circuit is composed of library cells, flattened, and reduced to basic gates. These are minimized, and the resulting netlist is mapped onto the basic FPGA logic blocks (e.g., [15], [3], [17], and many others). If placement did not occur during mapping (as in [7]), the resulting netlist must then be placed. Often, this is handled by simulated annealing ([16]). Structure or regularity information is lost during this process.

Our compaction is performed after a floorplanning tool has determined the linear placement of all modules in the datapath. The datapath may contain non-compactable modules. These are either highly irregular or very complex hard-macros, such as multipliers, laid out carefully to take advantage of the FPGA block and routing topologies and thus would deteriorate during the compaction, or macros exploiting special FPGA-specific features that are not covered by standard optimization tools. For the XC4000, this includes RAM/ROM blocks or the hard-carry logic for fast ripple-carry adders. These modules are not compacted and pass through the compactor unmodified.

Prior to the compaction process, the floorplanner selects sub-datapaths (sets of compactable modules) that are to be merged into a single module (Section 5). The module boundaries are broken up, and the separate functions of each module bit-slice are combined. Note that only the inter-module boundaries are broken, not the regular bit-slice structures within the modules.

By taking advantage of the regularity of the datapaths, the problem size for further operations can be reduced: The structure of a merged module is searched for repeatedly occurring sub-circuits (*zones*, Section 6). Each of these zones of duplicated logic is processed only once, and replicated as required.

The compaction itself applies standard logic optimization and

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

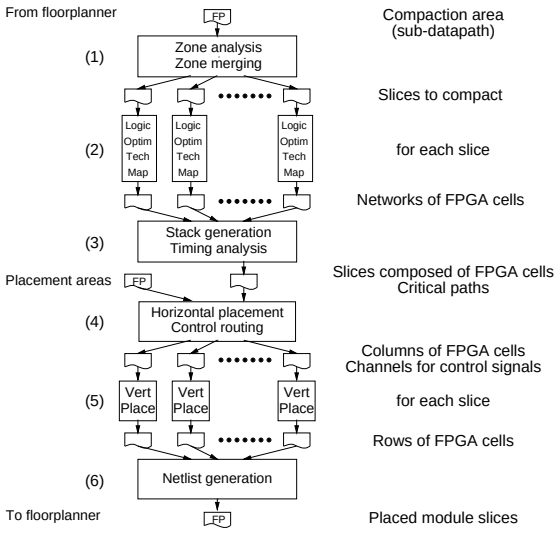


Fig. 2: Steps of the compaction process

technology mapping to the zone functions (Section 7). Since the placement information provided by the module generators is lost afterwards, the mapped FPGA blocks of the merged module have to be placed again in the context of the original floorplan.

The specialized two-phase placement algorithm is timing-driven (Section 8) and takes the regular datapath structure and FPGA-specific routing topologies into account. During the first phase, blocks are placed horizontally, observing the alignment of adjacent zones, and vertical control signals are globally routed (Section 9.1). The second phase assigns row locations to the blocks (Section 9.2). Since vertical placement occurs separately for each zone, it has a smaller problem size and can thus be more detailed, allowing the use of a finer representation of the routing structure of the target FPGA.

Finally, the placed netlist of the sub-datapath is assembled by duplicating and vertically stacking the zones according to the original width requirements.

The result is a new regular module fitting within the initial floorplan, but with reduced area and number of logic levels. Pin assignment and routing still have to be performed using conventional tools. Currently, the PPR program of the Xilinx XACT suite is employed to handle these tasks (Section 10).

The compaction process in Figure 2 will be explained in detail in the next sections.

4 Definitions

A circuit consists of *cells* (*nodes* or *ports*) that can be placed at (x,y) inside or adjacent to a *placement area* with height H and width W. Ports have just locations (no extent) and are either *data* or *control* ports. The location of a port may be *locked* to one or more sides of the placement area, fixing one or both of the coordinates during placement. A two-terminal net (*TTN*), represented by (a,b), has the output of cell a as source and an input of cell b as sink. A *path* $((a,b), (b,c), \ldots)$ consists of an unbroken sequence of TTNs. A *slice* s is a sub-circuit (e.g., bit-slice). It can be instantiated i times to form a *zone* (s, i) of replicated identical logic. When used in this manner, the slice s is called the *master slice* of the zone. A *stack* $\mathcal{S}$ is a sequence of zones describing a vertical stacking of zones from top to bottom. A *module* $\mathcal{M}$ consists of a stack $\mathcal{S}_{\mathcal{M}}$ and a netlist of TTNs. A *datapath* $\mathfrak{D}$ is a sequence of modules describing a linear placement from left to right. An FPGA matrix is composed of a grid of *blocks* (e.g., XC4000 CLBs).

Figure 3 shows a datapath of two modules, an 8-bit ALU and an 8-bit logical left shifter. ALU[7:0] is composed of a single zone created by instantiating the 4-bit ALU master slice named ALU4

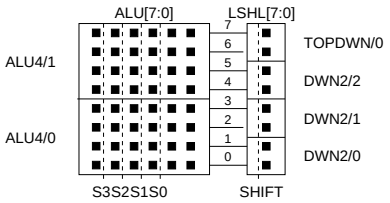


Fig. 3: Sample datapath

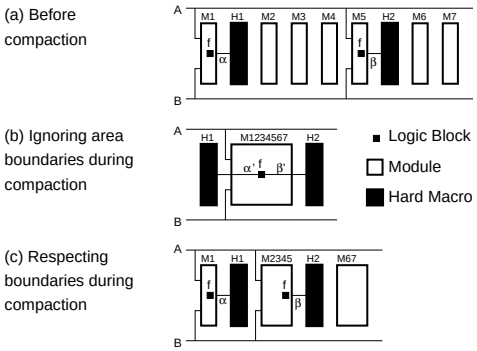


Fig. 4: Boundaries of compaction areas

twice. Thus, the stack associated with ALU[7:0] is $\{(\text{ALU4},2)\}$. The ALU4 slice consists of 24 cells in a (4,6) placement area. The ALU[7:0] module has operation select signals $S_3, \ldots, S_0$ as control inputs. If we assume that ALU[7:0] is used in a ripple-carry configuration, it will have the carry signal as a vertical inter-slice net between ALU4/0 and ALU4/1. The shifter module LSHL[7:0] contains the stack $\{(\text{DWN2},3), (\text{TOPDWN},1)\}$ and has the shift enable signal SHIFT as control input. Each of the slices will have a vertical inter-slice net to propagate a bit n to the next lower slice as bit $n-1$. Note that one slice of ALU[7:0] can process 4 bits, while the shifter master slices TOPDWN and DWN2 process only 2 bits per slice.

5 Selecting Sub-Datapaths for Compaction

Prior to compaction, the floorplanner determines parts of the original datapath to be compacted (top of Figure 2).

Although this selection is not part of the compaction operation itself, it significantly influences the quality of the resulting layout. Because the complete datapath might contain modules not amenable to compaction, the floorplanner has to determine the largest sets of suitable modules. Each of these sets is considered a sub-datapath of the whole datapath. The sub-datapaths are then handled independently, allowing the parallel compaction of each set of modules.

Figure 4 shows an example: The floorplanner has calculated a linear placement of modules (case *a*). H_1 and H_2 are hard-macros, and thus mark the boundaries of the three compactable sub-datapaths $\{M_1\}$, $\{M_2, \ldots, M_5\}$, and $\{M_6, M_7\}$. An even tighter packing might be obtained if the boundaries were ignored (in Figure 4.b, the duplicated function f is removed), but this would risk a degradation of wire lengths α' and β' over their pre-compaction levels. When the boundaries are respected, area is traded for speed: The compacted modules M_1, M_{2345}, and M_{67} are larger than $M_{1234567}$, but the wire lengths remain unaffected (Figure 4.c). Since the compactor is primarily performance-oriented, it follows the approach in Figure 4.c.

6 Zone Analysis and Merging

After determining their extent for compaction, each sub-datapath $\mathfrak{D}$ is processed separately. Since they are independent of each other, all steps of the complete compaction process in Figure 2 can be performed in parallel for all such $\mathfrak{D}$. The regular structure of $\mathfrak{D}$ is exploited to reduce run-times of the following compaction steps.

$\mathfrak{D}$ is searched for zones of recurring logic as the first compaction

```
S := {};          /* initially, we don't know any zones */
row := 1;         /* start at the bottom of the datapath to compact ... */

/* ... and work your way upwards */
while ( row <= max. height of datapath ) do {
   s := instances hit by horiz. scanline at row;
   h := height of tallest instance in s;

   temp := {};  /* prepare to assemble instances into temp */
   r := row;
   /* now collect instances across the stacks */
   while (r <= row + h) do {
      collect instances hit by horiz. scanline at r into temp;
      advance upwards to r + 1;
   }
   if (temp is a new slice) then
      add temp to S with iteration count of 1;
   else
      /* we have seen an instance of this zone before */
      just incr. the iter. count of the last occurence;

   /* now advance to the next row of instances */
   row := row + h;
}
```

Table 1: Algorithm for zone analysis

step (Figure 2.1). The functions of the separate modules $\mathcal{M}$ in $\mathfrak{D}$ are merged into a single module $\mathcal{M}'$ having the complete functionality of $\mathfrak{D}$. The master slices of the zones of $\mathcal{M}'$ are created by extending the bit-slices of each $\mathcal{M}$ across the module boundaries.

The master slices of the zones are now manipulated further and replicated as required. Note that $\mathcal{M}'$ is not yet optimized in this step (Section 7), and that the zones still contain placed blocks, since the original layouts have not yet been invalidated.

The algorithm in Table 1 analyzes a given $\mathfrak{D}$. Applied to the example of Figure 3, it proceeds as follows: The initial bottom scanline collects ALU4/0 and DWN2/0 into s. Thus, h becomes 4. The inner loop hits ALU4/0 and DWN2/0 twice, then it advances upwards to hit ALU4/0 and DWN2/1 twice. Each instance hit, however, is added only once per module to *temp* (e.g., we don't add ALU4/0 twice). Since we have now reached the upper border of ALU4/0, the inner loop terminates, and we add the new zone to $\mathcal{S}$ with an iteration count of 1. We now repeat the process for the next row up and acquire a second zone of one slice containing ALU4/1, DWN2/2 and TOPDWN/0. This results in a stack $\mathcal{S}_{\mathcal{M}'}$ {({ALU4,DWN2,DWN2)},1), ({ALU4, DWN2, TOPDWN},1)}. The networks in the master slices of the zones are now merged by following their intra-zone (but inter-module!) connections. Thus, $\mathfrak{D}$ is being merged into a single $\mathcal{M}'$ with the two slices ALU4-DWN2-DWN2 and ALU4-DWN2-TOPDWN.

In the current example, we have not saved any work, because each slice occurs just once in $\mathfrak{D}$. Nevertheless, if we assume a 12-bit datapath similar to the one in Figure 3, the slice {ALU4, DWN2, DWN2} would occur twice in $\mathfrak{D}$ as zone ({ALU4, DWN2, DWN2}, 2). It would only be processed once during compaction, the results being duplicated to build the 8-bit bottom zone of the 12-bit $\mathcal{M}'$. The gains are even more pronounced with wider datapaths, such as 32 bits. In addition, the zone analysis and merging operation can be performed in parallel on each of the compaction areas specified by the floorplanner.

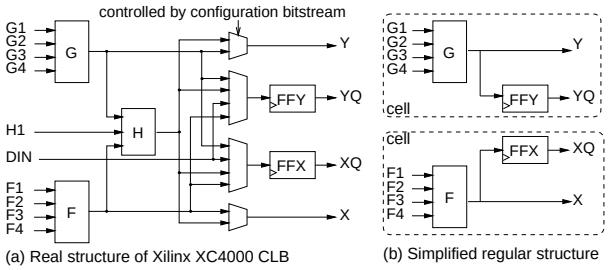
The algorithm in Table 1 is a simplified version. The full implementation also considers special cases like vertically overlapping slices and changing port locations between slices.

7 Logic Optimization

After obtaining the master slices of $\mathcal{M}'$, we now reduce area and delay. The regularity extraction performed on $\mathfrak{D}$ in Figure 2.2 allows parallel processing of just the master slices instead of a monolithic operation on all nodes in $\mathcal{M}'$.

Optimization applies classical logic synthesis and technology mapping algorithms to each master slice of $\mathcal{M}'$. This proceeds *across* the boundaries of the modules $\mathcal{M}$ originally making up $\mathfrak{D}$, but *preserves* the regular architecture of the datapath itself (vertical stacks of bit-slices). The potential for optimization grows with the size of the master slice (see Section 5 for limitations).

Since the compaction process is generally independent of the sub-algorithms employed, it can easily take advantage of any new advances in the fields of optimization and mapping. For example, the initial version of the compactor supported only the "xl" (MIS-PGA) commands in SIS 1.3 [17] to perform technology mapping to N-LUTs. The current compactor can also employ the more recent FlowMap package [8] that emphasizes delay over area minimization, allowing the user to make a trade-off by choosing the algorithm. Minimization and mapping transform the networks of FPGA blocks (CLBs for XC4000, Figure 5.a) separately for each master slice of $\mathcal{M}'$ into optimized networks of cells. All placement information is lost and has to be recreated by the following steps.



(a) Real structure of Xilinx XC4000 CLB (b) Simplified regular structure

Fig. 5: Real and simplified structures for XC4000 CLBs

For the XC4000, a cell consists of a 4-LUT, optionally combined with a flip-flop (Figure 5.b). The current compactor implementation does not attempt to handle irregularities in the FPGA logic blocks (e.g., the H-block in XC4000 CLBs). Thus, two cells fit inside the regular part of a CLB. With recent FPGA architectures striving to avoid irregular structures (e.g., Altera FLEX and Xilinx XC5000/XC6200 chips), this restriction seems less severe and could even be removed by the integration of the appropriate CLB packing algorithms.

8 Pre-Placement Activities

Since the minimization and mapping steps change the circuits in the master slices, the initially generated module layouts are no longer valid and the cells of the slices have to be re-placed.

In order to execute a timing-driven cell placement, a critical path analysis of the complete $\mathcal{M}'$ has to be performed (Figure 2.3). To do so, $\mathcal{M}'$ is assembled by interpreting the topology in $\mathcal{S}_{\mathcal{M}'}$ and instantiating the slices accordingly. Next, the cells are interconnected with vertical inter-slice nets and control nets.

The delay trace can then be executed using either the unit delay or unit-fanout delay models of SIS. Afterwards, the arrival and required times of inter-slice nets are back-annotated to 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.

While these timing constraints are not accurate enough to estimate a real inter-slice path through its master slices, they can be used to determine paths that are critical at all (having slacks ≤ 0). Multi-terminal nets are decomposed into one or more paths of TTNs.

The result is a list of critical paths for each master slice, sorted by ascending length. The timing-driven placement uses these lists to minimize wire lengths on critical paths.

The floorplanner is responsible for determining the placement

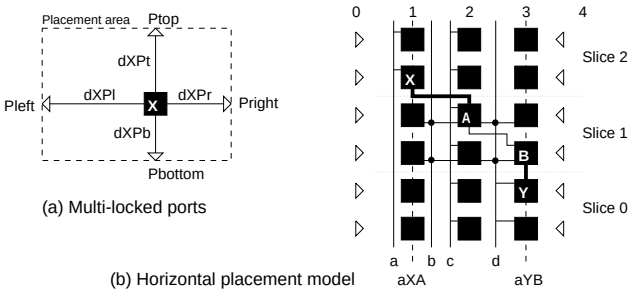


Fig. 6: Multi-locked ports and horizontal placement model

area for each slice, in particular its height H. The whole floorplan will profit from a homogeneous bit-slice height (pitch) across all modules (compacted and hard-macros). Thus, these calculations cannot be performed by the compactor with its local view of the subdatapath $\mathfrak{D}$.

9 Cell Placement

In order to create a regular placement of the cells in the optimized master slices, the placer has to consider the context of $\mathcal{M}'$ in the original datapath as laid out by the floorplanner. In particular, the location of data I/O ports and the general topology of the original datapath have to be observed (how high should a bit-slice be for maximum regularity?).

The timing-driven placer is currently based on 0-1 integer linear programs (ILP). It executes in two separate phases for column (Figure 2.4) and row locations (Figure 2.5). Both phases have different aims, which would be too complex for a single ILP model. Using heuristics different from the ILPs, placement might be attempted in a single phase. The compaction process is open to such alterations in sub-algorithms. An alternative placer using simulated annealing has already been implemented for experimentation. Due to space limitations, only the models underlying the ILPs will be described, see Section 10 for general comments on their actual formulation.

The two phases of the current ILP placement minimize the maximum wire length d_{max} on the critical paths in their objective functions. The length d_p of a path p is obtained by adding up the lengths $|a-b|$ of the TTN segments (a, b) in p.

Since the two phases have different scopes (module in the horizontal vs. slice in the vertical phase), the placer uses different length metrics in each phase. Due to its more limited scope, the metric used in the vertical phase (Section 9.2) can be more precise than in the horizontal phase (Section 9.1).

Except for the allocation of vertical long lines (*VLL*) in the horizontal phase, no effort is made to balance congestion in routing channels. This seems feasible, because the pins on a CLB are interchangeable to a large degree. Thus, the pin assignment and routing steps can relieve congestion by swapping pins to less dense channels.

Both phases handle multi-locked ports identically (Figure 6).a. Assuming that a port P, sourced by node X, is multi-locked to all sides of the placement area, the distance $d_{XP} = \max(d_{XP_l}, d_{XP_t}, d_{XP_r}, d_{XP_d})$ used as the length of TTN (X, P) for critical path calculations will be modeled by taking the maximum distance of all TTNs connecting its source node X with the corresponding port location of P.

9.1 Horizontal Placement

During horizontal placement (Figure 2.4), 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) To allow vertical inter-slice nets in adjacent slices of $\mathcal{S}_{\mathcal{M}'}$ to be routed by abutment, if possible. (3) To minimize the maximum routing length on critical paths. For horizontal placement, all master slices of $\mathcal{M}'$ have to be considered simultaneously, since control signals and vertical inter-slice signals cross slice boundaries.

The underlying ILP is based on the model shown in Figure 6.b. The placement areas for Slice0, Slice1, and Slice2 in the example each consist of a (2,3) grid of cells. Data ports of $\mathcal{M}'$ can be placed adjacent to the areas in columns 0 (for left ports) and 4 (right ports). Each column also has an associated control routing channel with 10 vertical long lines (VLL) for control routing (a maximum of 2 VLLs per channel is used in the example). This channel is assumed to lie left of the cell column. A control signal in channel n is available to cells in columns n and $n-1$ (e.g., control b in channel 2 reaches cells in columns 1 and 2). Note that for control routing, the channel $W+1$ directly to the right of the placement area (H, W) is also considered available.

If necessary, control signals can be replicated and routed in multiple channels (not shown in the example). Thus, the number c of VLLs used for control routing can be greater than the number of control signals.

The alignment for inter-slice connections of adjacent cells (TTNs (X,A) and (Y,B) in the example) is modeled by determining the deviation from the ideal alignment lines (a_{XA} and a_{YB}) as $|x_X - x_A|$ and $|x_Y - x_B|$, respectively. The placer should minimize the maximum alignment error a_{max}. The example has $a_{XA} = 1$ and $a_{YB} = 0$, thus $a_{max} = 1$.

The wiring delay of intra-slice TTNs, such as (A,B), is also modeled as $|x_A - x_B|$. However, this metric becomes increasingly inaccurate with growing 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 of a maximum of one switch matrix for 4 cells (4-LUTs) in a (4,1) area. Thus, $|A-B|$ 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 lacking an estimation are measurably worse in terms of delay than those with the proposed estimation. The impreciseness of this 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 three quantities introduced in the preceding paragraphs, the objective function for the horizontal placement phase becomes $\min(\ w_d d_{max} + w_c c + w_a a_{max}\ )$. w_d, w_c, and w_a are user-definable weights. E.g., a user might increase w_d over w_c when a faster circuit at the cost of an increased number of control lines is desired.

9.2 Vertical Placement

In contrast to the horizontal phase, the vertical placement phase (Figure 2.5) 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 master slice.

With the reduced problem size, 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 7 shows the model, which is a simplified view of the XC4000 routing network. Cells A to I have been labeled to serve as example TTN nodes in further explanations. 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 are presently not included.

The horizontal phase was concerned only with placing cells. The vertical phase, however, has to take FPGA block boundaries into ac-

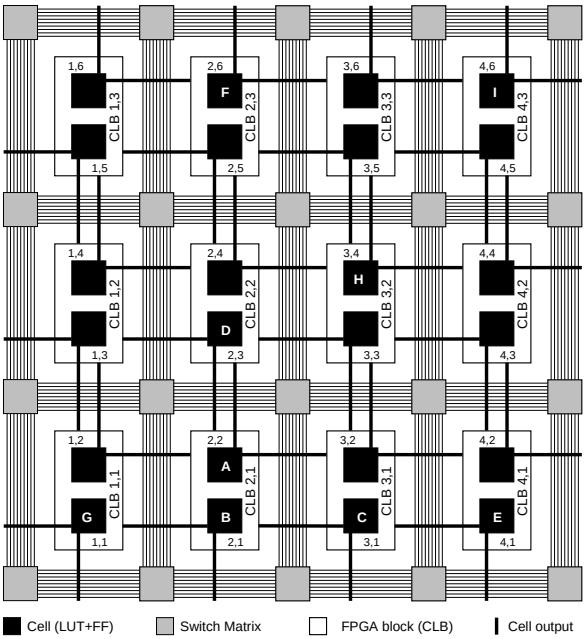


Fig. 7: Vertical placement model

count and thus operates on a CLB matrix with the same width, but half the height of its underlying cell matrix. 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 5). Y/YQ and X/XQ output pins are assumed equivalent for routing purposes: The Y/YQ pins reach above and to the right of their CLB, the X/XQ pins below and to the left. The location of input pins is not modeled 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 metric employed in this phase is not based on simple manhattan distances, but only on an actual count of switch matrices (SM) in a signal path. In order to do so, three major cases based on the horizontal distance of cells (a, b) of a TTN have to be considered. For each case and sub-case, the corresponding TTNs in Figure 7 will be pointed out.

If the horizontal distance is 0, the SM-distance 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), $d_{SM} = 2$).

	XACT PPR			SDI		
	UFC-A	T16	TALU32	UFC-A	T16	TALU32
Wire delays in ns						
best	36.3	30.4	42.1	33.5	28.0	35.9
worst	51.2	37.6	63.9	40.7	30.8	38.3
Total run-times for one iteration in s						
best	834	345	6460	444	143	925
worst	694	353	4493	430	131	971
runs	73	77	123	114	821	596

Table 2: Benchmark results and run-times

10 Experimental Results

The compactor has been implemented as part of the SDI strategy [12]. It consists of 6000 lines of C that extend SIS 1.3 [17]. The models are formulated as pure 0-1 problems to allow pre-processing by OPBDP [4], which performs "logic optimization" on the ILPs and quickly generates an upper bound using constructive enumeration techniques. CPLEX [9] solves the resulting models.

Due to the lack of an established benchmark suite for datapath structures, two non-standard circuits were selected as examples. In the context of the compactor, only regular datapaths are examined. Controller processing is left to other SDI components. To evaluate the quality of our regular approach and avoid inaccuracies due to different module generators libraries, all test circuits were entered manually. We compare the performance of our regularly compacted circuits against those obtained by the standard design implementation procedure using the Xilinx XACT PPR tool (irregular placement of flattened design).

UFC-A is part of an address generator for fast DES encryption. It was entered initially as 26 16-bit combinational modules. Regular compaction using MIS-PGA reduced the size from 368 to 96 LUTs. Irregular optimization and mapping of the flattened circuit by PPR yielded a reduction to 112 LUTs.

T16 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). It is composed by stacking a single slice of sixteen 4-LUTs 4 times per module. For this benchmark, logic optimization or technology mapping was performed neither by SDI nor by PPR in order to directly compare the regular (SDI) to the conventional irregular placement (PPR).

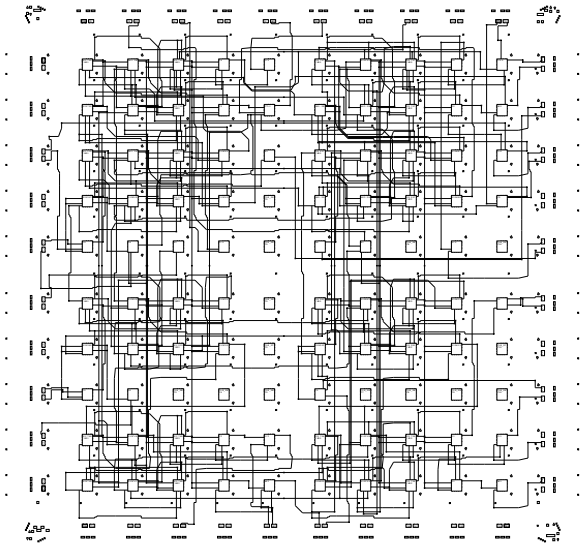
TALU32 is a 32-bit ALU with registered inputs built by stacking eight 74181 [11] 4-bit ALU slices. The 74181 slice has been minimized and mapped for area efficiency from 65 nodes to 24 4-LUTs by MIS-PGA commands.

PPR was always run with maximum optimization (placer_effort = 5) in performance-driven mode (dp2p, dc2p) with all pads floating. Both SDI and PPR placements were routed by PPR, also using maximum optimization (router_effort = 4). The run-times in Table 2 were measured on an unloaded Sparc 20/71 workstation with 64MB RAM. Since the simulated annealing in PPR is non-deterministic, measurements are listed for the best and worst cases over a number of runs.

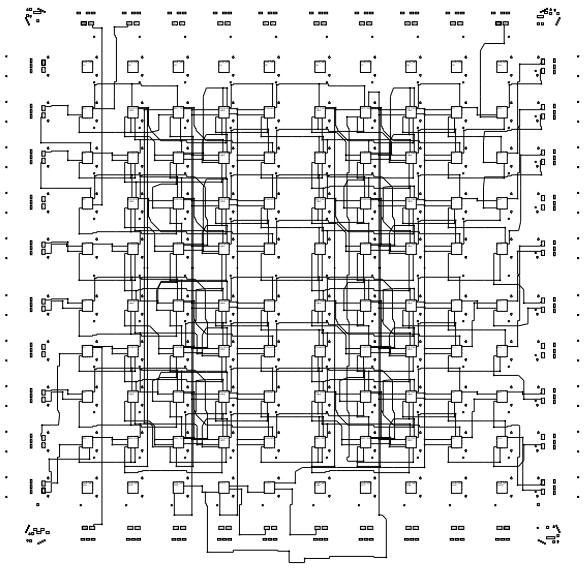
The two resulting layouts of T16 are shown in Figures 8(a) and 8(b). Even at first glance, the SDI-placed solution is obviously more regular, since the natural structure of the datapath has been

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 best SDI solution is 8% to 26% shorter than in the maximally optimized PPR layout. The reproducibility of good placements with SDI is also improved over PPR: SDI has a best-worst interval of 2.8ns over 821 runs versus PPR with 7.2ns over 77 runs. For PPR, the interval is growing with the number of runs executed. The execution of an SDI placement followed by PPR routing takes roughly half as long as performing both placement and routing through PPR.

The gains are even more pronounced with the larger TALU32 circuit, with the routing delay of the best SDI-solution being 15% to 44% shorter than in the PPR-generated layout. The SDI best-worst interval is only 2.4 ns over 596 runs compared to 21.8ns over 123 runs for PPR. On average, one SDI-PPR cycle takes one-sixth of the time of a PPR-only cycle.



(a) PPR placement and routing



(b) SDI placement and PPR routing

Fig. 8: T16: Placement and routing

UFC-A does not improve as much as TALU32. This is most likely caused by its very simple bit-slices (few inter-slice connections or control lines). Performance improvements using SDI seem to grow not only with the regularity of the bit-slices, but also with the degree of datapath-like interconnections (module-wide control lines, inter-slice signals).

11 Conclusions

For strictly module-based datapaths, the compaction process has consistently outperformed the standard tools in both run-times and routing delay minimization. The general method is applicable for all FPGAs with a matrix structure.

Even further speed-ups of the algorithm are possible by fully exploiting the options for parallel execution and optimizing the ILPs (e.g., by adding explicit cutting planes). By refining the FPGA routing model (e.g., including double-length lines), an additional reduction in circuit delay times is also achievable. However, considering the limited number of circuits evaluated thus far, further benchmarking is necessary and still in progress.

12 References

[1] Atmel Corp., “IDS Reference - Component Generators”, *EDA software documentation*, San Jose (CA) 1994

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

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

[4] Barth, P., “A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization”, *MPI-I-95-2-003*, Max-Planck-Institut für Informatik, Saarbrücken 1995

[5] Brand, H.J., Müller, D., Rosenstiel, W., “Specification and Synthesis of Complex Arithmetic Operators for FPGAs”, in *Field Programmable Logic* , ed. by Hartenstein R.W., Servits, M.Z., Springer 1994, pp. 78

[6] Cai, H., Note, S., Six, P., DeMan, H., “A Data Path Layout Assembler for High-Performance DSP Circuits”, *Proc. 27th DAC 1990*, pp. 306

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

[8] 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

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

[10] Curry, D., “Schematic Specification of Datapath Layout”, *Proc. ICCD 1989*, pp. 28

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

[12] Koch, A., “Structured Design Implementation – A Strategy for Implementing Regular Datapaths on FPGAs”, *Proc. FPGA ’96*, pp. 151

[13] Marshburn, T., Lui, I., Brown, R., et al., “DATAPATH: A CMOS Data Path Silicon Assembler”, *Proc. 23rd DAC 1986*, pp. 722

[14] Matsumoto, N., Watanabe, Y., Kimiyoshi, U., “Datapath Generator Based on Gate-Level Symbolic Layout”, *Proc. 27th DAC 1990*, pp. 388

[15] 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

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

[17] Sentovich, E.M. et al., “SIS: A System for Sequential Circuit Synthesis”, *UCB/ERL M92/41*, Dept. of EE and CS, UC Berkeley 4 May 1992

[18] Vandeweerd, I., Croes, K., Rijnders, L. et al., “REDUSA: Module Generation by Automatic Elimination of Superfluous Blocks in Regular Structures”, *IEEE Trans. on CAD*, Vol. 8, No. 9, September 1989, pp. 989

[19] Xilinx Inc., “XACT X-BLOX User Guide”, *EDA software documentation*, San Jose (CA) 1994