CHAPTER 15

DATAPATH COMPOSITION

Andreas Koch
Department of Computer Science
Embedded Systems and Applications Group
Technische Universität of Darmstadt, Germany

As shown in Chapter 14, a wide variety of algorithms can be employed for placing arbitrary netlists on various reconfigurable fabrics. To achieve this generality, the input netlists are treated as random collections of primitive elements (gates, lookup tables [LUTs], flip-flops) and interconnections. These approaches do not attempt to exploit any kind of structure that might be present in their input circuits. Many practically relevant circuits, however, do exhibit regularities in their composition (e.g., by following a classical bit-sliced design). Since the days of manual full-custom ASIC design ("polygon pushing"), regularity in circuit *structure* has been exploited with great success to derive a corresponding regular circuit *layout*—for example, by abutment of replicated bit-slice layouts.

This chapter describes the application of this idea to efficient layout of regular bit-sliced datapaths on reconfigurable fabrics. It will begin by considering how to characterize, extract, and preserve regularities at different abstraction levels. The next steps describe the datapath composition tool flow and address issues such as mapping dataflow operators to hardware units and arranging these in an abutting regular layout. We will also cover how quality can be improved even further by judiciously dissolving regularity boundaries in parts of the datapath performing cross-boundary optimization, and finally reregularizing the optimized circuit.

15.1 FUNDAMENTALS

With the increasing use of reconfigurable devices as core processing units in adaptive computer systems, the architecture and implementation of high-performance compute units on reconfigurable fabrics becomes ever more important. A *datapath* is one architectural style of realizing a given computation (Figure 15.1(a)) in hardware. It is often described as the number of interconnected *operators* in the form of a dataflow graph (DFG) or control dataflow graph (CDFG), shown in Figure 15.1(b). The execution of the operators is orchestrated by a supervising *controller* (Figure 15.1(c)). The controller is generally not considered part of the datapath, but together the datapath and controller form a *compute unit*. For purposes of this discussion, we will assume that we are processing a CDFG but will concentrate on its dataflow part.

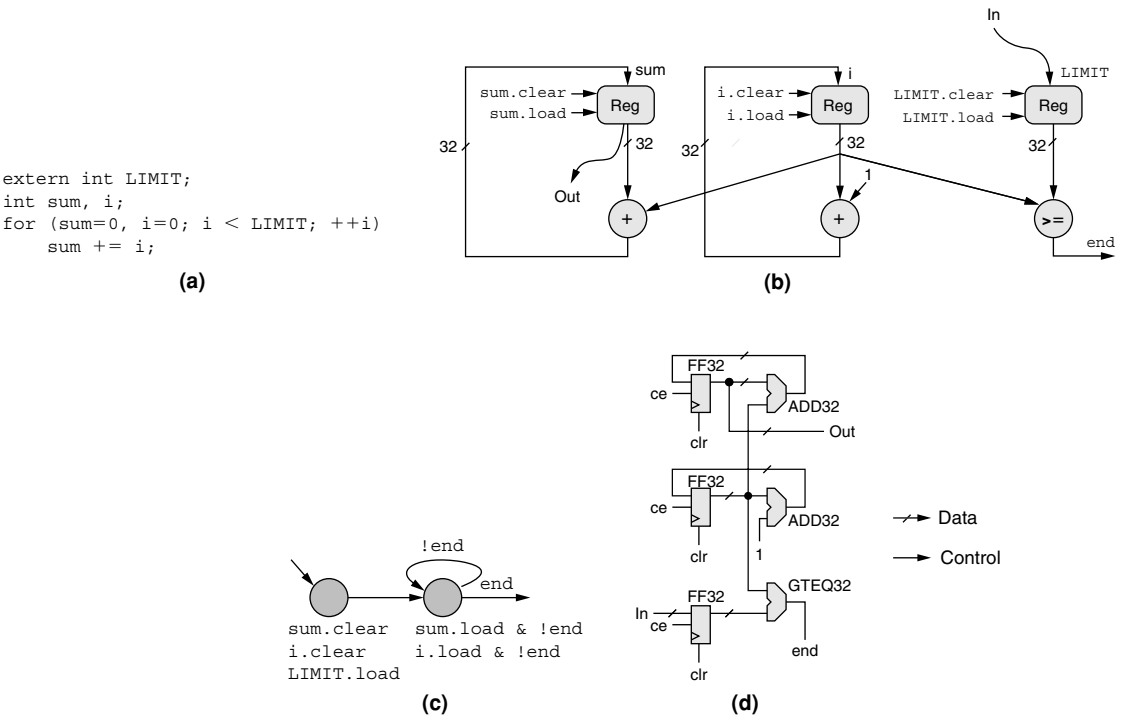


FIGURE 15.1 ■ From computation to realization.

The datapath is created in hardware by mapping the CDFG operators to *hardware operators*, or HWOPs (see Figure 15.1(d)). Generally, HWOPs have multibit *data* inputs and outputs for the operand(s) and result(s) (e.g., ADD32 HWOPs). Some may also have *control* inputs (e.g., the load and clear signals of the FF32 HWOPs) or outputs (e.g., for indicating certain conditions such as the GTEQ32 output). These control signals are generally much narrower than bused data signals, often only a single bit wide. In some cases, an HWOP is available in several different *implementations*, all having the same function but differing, for example, in their area/speed characteristics or layout shape.

15.1.1 Regularity

The multibit-wide HWOPs are often assembled by repeatedly instantiating and interconnecting narrower template circuits in an adjacent fashion until the specific HWOP's desired bit width is reached (Figure 15.2). These template circuits will be called *master slices* here, while their instances are generally referred to as *bit slices*. We will further extend this terminology to call areas where the same master slice has been instantiated a number of times a *zone*, and a sequence of zones is termed a *stack*. Together, these concepts describe an HWOP as a *regular* circuit.

Such a structure has a natural direction of *dataflow* (horizontally in the case of Figure 15.2). When processing word-wide data, the individual bits of the

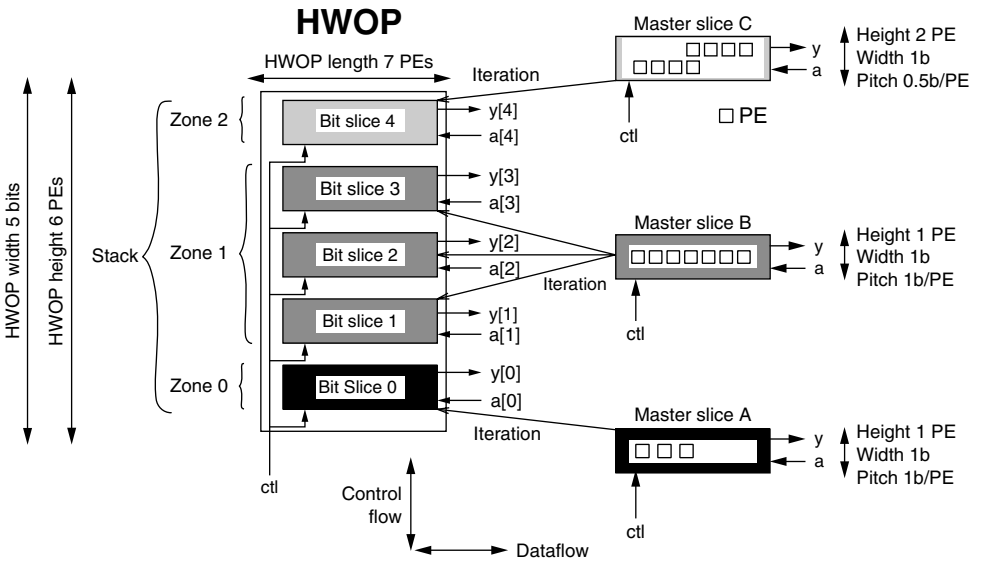


FIGURE 15.2 ■ Regular HWOP structure.

words are arranged orthogonally to the direction of dataflow (in the figure, vertically). With few exceptions (e.g., bus-wide logic gates), the position of individual bits is not arbitrary but follows an ordering from least significant (LSB) to most significant (MSB). For example, stacking ripple-carry full-adder bit slices generally has the first slice process the LSB and the last slice process the MSB. Ports on the master slice (e.g., a, y) do not have a bit significance of their own. Only after instantiating the masters as bit slices can the significance be derived from their iteration number (e.g., port a on the bottommost slice will have a significance of 0; the one above that, 1, etc.).

For describing the characteristics of elements such as HWOPs, bit slices, and master slices, four quantities are useful. Any of these elements may process multiple bits from a single word, with the logical *width* being the largest number of such bits. *Height* and *length* refer to the bounding box of the element layout on the target device. They are specified in device-dependent units, such as processing elements (PEs), cells, configurable logic blocks (CLBs), and the like. The *pitch* of a master slice is the width divided by the height—essentially, the number of output bits per unit height. To reduce interconnect lengths, all HWOPs in the datapath should have the same pitch and the LSBs of all data nets should be vertically aligned.

Regularity in datapaths does not appear just in the replicated logic elements but also in commonly occurring interconnect patterns (Figure 15.3):

Data nets are generally multibit buses that carry operands and results between HWOPs, where they are connected to data ports (e.g., `op1`, `op2`). Each signal in the bus has an associated bit significance and generally connects to the HWOP at a data port with the same significance. Shifts and permutations occur only rarely [23].

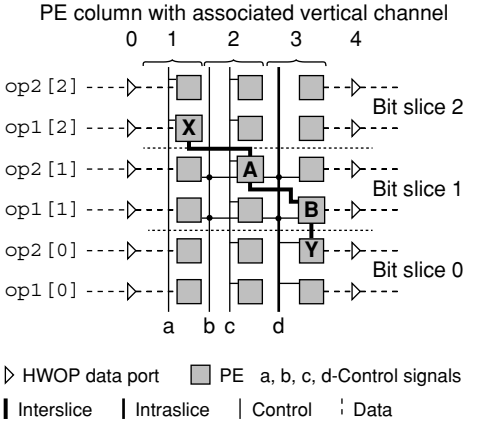


FIGURE 15.3 ■ Regular interconnection patterns.

Control nets are generally narrower, often only a single bit wide. In general, they connect an HWOP to a controller but not to another HWOP in the datapath. Control signals attach to the HWOP at control ports. In many cases, a control signal connects to the same control port in all bit slices of a zone. With our assumption of horizontal dataflow, in the following discussion control signals are assumed to run vertically.

Interslice nets run between separate bit slices in the same HWOP, thus vertically crossing slice boundaries (e.g., B–Y, A–X). Most commonly, they connect neighboring bit slices, but these may have different master slices, particularly near the top and bottom of a stack. An example of an interslice net is the carry net running between full-adder bit slices.

Intraslice nets connect individual logic elements within a bit slice (e.g., A–B). Since the internals of a bit slice are considered random logic, these nets do not follow specific interconnection patterns.

An example of a unified representation for both block and interconnect regularity, the Abstract Physical Model (APM), is proposed by Ye and De Micheli [22].

15.1.2 Datapath Layout

With these concepts in place, we can now consider the anatomy of our compute unit in greater detail (Figure 15.4(a)). The datapath will have a *regular* area, where pitch-matched HWOPs with a common direction of increasing bit significance process horizontal, LSB-aligned dataflows. Outside this area, HWOPs may contain *irregular* parts (e.g., carry initialization, overflow detection, or, for complex sequential HWOPs, even local controllers). The global controller for the compute unit is also placed outside the regular area. Generally, control nets are routed vertically across the regular area. This chapter does not address the handling of the controller, but concentrates on the datapath

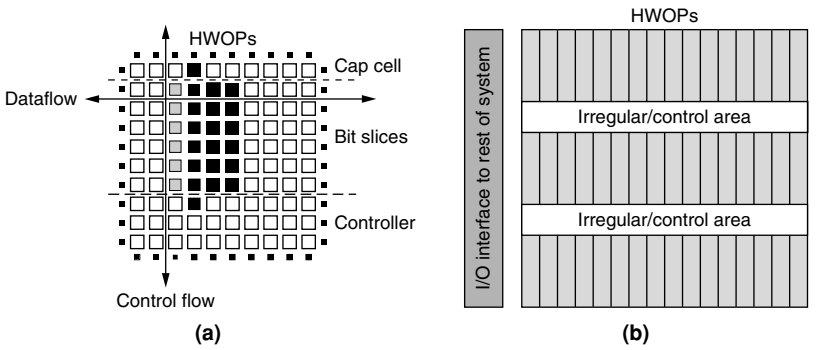


FIGURE 15.4 ■ Common datapath layouts: (a) classical linear and (b) multistripe.

instead. The controller can be placed via techniques such as those presented in Chapter 14.

Given these constraints, the best arrangement for minimizing interconnect lengths and delays for a small number of HWOPs will generally be linear. This approach has been exploited by devices like Garp [4], which realize such topologies directly in their chip architecture. However, once the number of HWOPs grows, the datapath generally needs to be wrapped into multiple stripes of HWOPs (Figure 15.4(b)).

15.2 TOOL FLOW OVERVIEW

Multiple steps are required to actually compose the datapath from individual HWOPs. These steps can be broadly grouped into the following categories:

Module generation: The HWOPs are often realized by procedural descriptions in the form of module generators (see Section 15.4). Thus, at some point in the flow other tools will interact with the library of module generators either to retrieve data *about* appropriately parametrized module instances or (later in the process) to generate the actual netlists. Often these netlists are already annotated with module-local relative placement information.

Mapping: The operators in the computation are mapped from the CDFG to the HWOPs realizing them in hardware. Beyond a straight 1:1 mapping, this can be performed in 1:*M* (if an operator requires multiple HWOPs) or *N*:1 fashion (if multiple operators can be combined into the same HWOP). The mapping calculated here need not be final, but can be altered in later flow steps. In some cases, the mapping step can also choose among multiple different HWOP implementations for an operator. This is sometimes called the *module selection* step.

Placement: HWOPs are assigned to actual PEs on the target device fabric. Similarly to the mapping step, *N*:1 and 1:*M* assignments are possible here. In the first case, a PE is so complex that it can implement multiple HWOPs at the same time. In the second case, each HWOP needs to be realized using multiple PEs. This is usually the case when targeting fine-grained devices such as field-programmable gate arrays (FPGAs).

Compaction: This is the altering of the HWOPs' structure after mapping (before or after placement). It generally indicates optimizing across HWOP boundaries. For example, it might merge connected adjacent HWOPs into a more compact/faster, but functionally identical, hardware block. This optimized block is then treated as any other HWOP in the datapath.

Not all of the flows discussed next perform all of these steps, and their execution order can vary. Additionally, some steps may be repeated.

Certain combinations are also possible. For example, in some flows placement and the mapping of operators to HWOPs occur simultaneously. For coarse-grained targets, operators can be mapped to HWOPs that are placeable in the same PE. For fine-grained devices, HWOP implementations can be selected whose layouts fit together with minimal area.

15.3 THE IMPACT OF DEVICE ARCHITECTURE

The tool flow required for creating a datapath on a reconfigurable fabric of PEs is highly dependent on the target device architecture. For coarse-grained target devices, the operators of the computation can often be mapped to PEs in a one-to-one fashion. On a fine-grained device, the operators have to be assembled from individual PEs.

Bit-sliced is not the only way to realize HWOPs. They may as well be completely irregular internally, or they may be monolithic (Figure 15.5). In both cases, many of the optimizations described in Section 15.7 that affect the internal structure of HWOPs are not be applicable. However, the techniques for processing multiple HWOPs at the datapath level (Section 15.6) remain relevant.

If the reconfigurable fabric has a linear or a two-dimensional matrix structure (Figure 15.6(a–c)), this can be exploited to efficiently map the regular

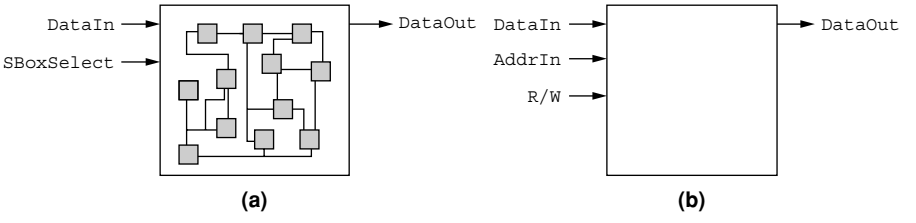


FIGURE 15.5 ■ Non-bit-sliced HWOPs: (a) irregular and (b) monolithic.

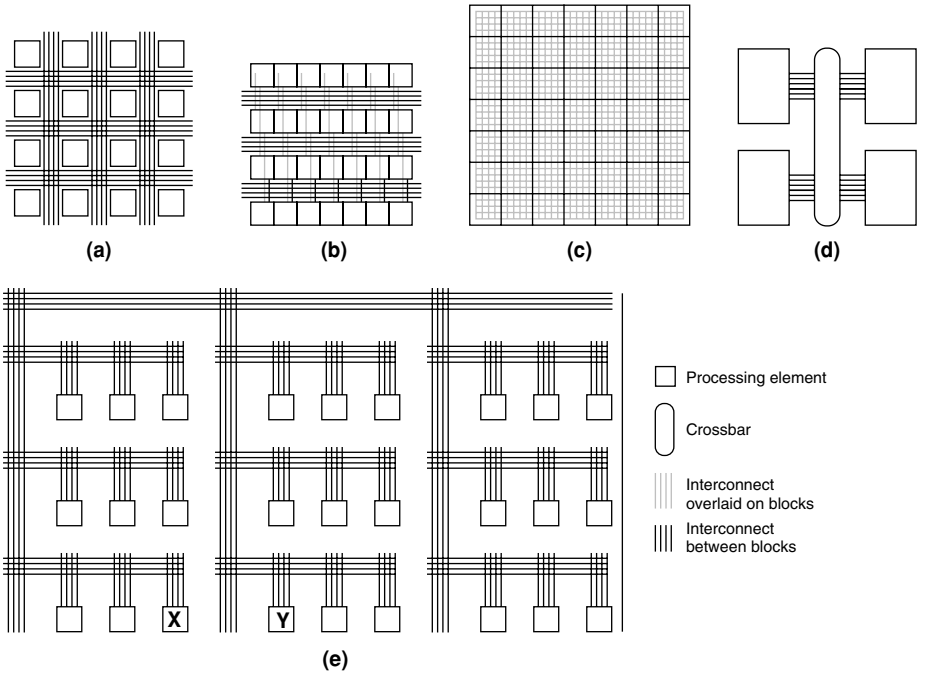


FIGURE 15.6 ■ Reconfigurable fabric architectures: (a) symmetrical array, (b) row-based, (c) sea-of-gates, (d) hierarchical PLD, and (e) hierarchical FPGA.

datapath structure to a corresponding regular geometric layout. For other kinds of target devices—for example, those having fully hierarchical structures (d–e in Figure 15.6)—algorithms optimizing for geometric arrangement are unsuitable, because geometrically adjacent blocks on the device might not actually be neighbors in the interconnect network (Figure 15.6(e), PEs `X` and `Y`). While other techniques such as hierarchical partitioning and clustering [19] could be used instead, they no longer attempt to take advantage of the datapath regularity.

15.3.1 Architecture Irregularities

Even in seemingly regular fabrics, irregularities often occur at the detail level. Consider, for example, the logic block structure of the Xilinx XC4000 FPGA (Figure 15.7). The base architecture of this device is a symmetrical array of CLBs, each of which contains two 4-LUTs and registers. However, each CLB also provides an additional 3-LUT. While very useful (e.g., for the efficient implementation of 4-input multiplexers or 5-input functions within a single CLB), the 3-LUT impedes the regularity in that it is no longer possible to realize two instances of a master slice that uses the 3-LUT within a single CLB. Also, when using the 3-LUT it is no longer possible to employ the registers in the CLB independently from the 4-LUTs: Only one of the registers can be directly connected to a CLB external port (`DIN`); the other one is not reachable from the outside.

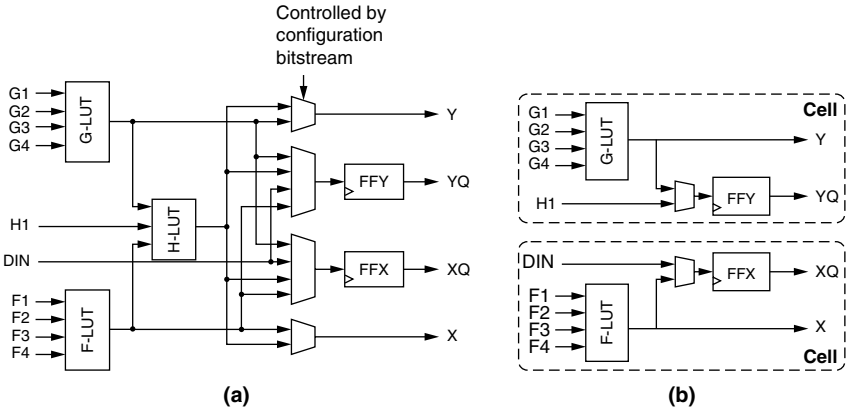


FIGURE 15.7 ■ Regularizing an existing device architecture: (a) the real structure of the Xilinx XC4000 CLB and (b) the simplified regular structure.

These irregularities can be alleviated by disregarding the 3-LUT for regular logic, using it solely to make the other register accessible via the `H1` port. As a result, each CLB can now be used to implement two fully regular bit slices, with the registers accessible both from inside and outside the bit slice.

Interconnect features also have an effect on datapath placement style. The physical direction of bit significances on the fabric is sometimes dictated by the running order of fast carry wires, which, on most devices is fixed. Also, high fanout control signals (e.g., the select signal of wide multiplexers) can be distributed across an entire HWOP by special long-distance interconnects. For example, on the Xilinx Virtex series of chips, so-called vertical long lines connect to all PEs on both sides of a vertical routing channel and are thus ideally suited for control routing. As will be shown in the following section, tool flows for datapaths can take advantage of all these features for efficient layout.

15.4 THE INTERFACE TO MODULE GENERATORS

As in many hardware design flows, individual hardware cells (in our case, the circuits used as HWOPs), are retrieved from a library. Instead of static cells, however, a more flexible approach uses procedural module generators to tailor these circuits to fit current requirements. For example, a multiplier might have eight pipeline stages in one context and only four in another, matching it to the latency/clock speed of the rest of the datapath. No longer a passive collection of cell descriptions, the library now becomes *active*: It accepts a set of constraints from another part of the flow and delivers a matching circuit.

The very flexibility of these parametrized generators complicates their integration with the rest of the tool flow: Other tools need not only the circuit description in the form of a (possibly preplaced) netlist but also data *about* this specific instance. Different tools are interested in different aspects of the

circuit. This plethora of cell *views*, combined with the sheer volume of the design space covered by each parametrized generator, precludes a simple enumeration of all alternatives. Thus, the traditional static library data files, holding tables of delays, bounding boxes, and the like, for a set of fixed parameter values, become impractical.

The Flexible API for Module-based Environments (FLAME) [11] is one approach to overcoming these difficulties. It consists of three major components: (1) the communications interface between the generator library and the other flow tools, (2) the design data model, and (3) the library specification.

A reference realization of a FLAME-based generator library exists in the form of the Generic Library for Adaptive Computing Environments (GLACE) [14]. This package has successfully been used in the COMRADE compiler [7], which compiles C into hybrid hardware/software applications for adaptive computer systems. GLACE uses a Java-based FLAME implementation, but could be called from other languages using the Java Native Interface (JNI).

15.4.1 The Flow Interface

The communications infrastructure and API provided by the FLAME Manager (Figure 15.8) replace static library files with an active function call–based interface. Clients in the main design flow can thus enter into a dialog with the module libraries and retrieve data specific to the actual parameter values of the current instance. In GLACE, the client queries accepted by the FLAME Manager are forwarded to the circuit generation code [6], resulting in the retrieval of circuit characteristics, or the creation of actual netlists.

15.4.2 The Data Model

The information exchanged in this manner just described is represented using the FLAME design data model. This model is partitioned into a number of task-specific views: A frontend compiler might request a "behavior" view to determine which functions are available for a given target technology. Later on, it could

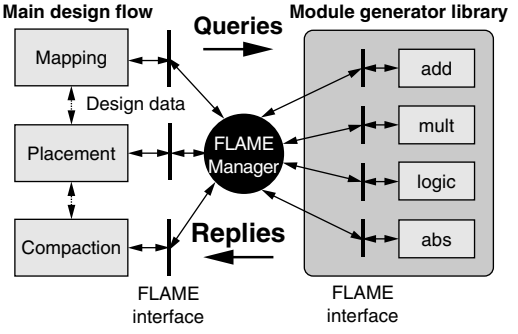


FIGURE 15.8 ■ FLAME system overview.

query for a "synthesis" view to retrieve area and timing characteristics for a specific module instance. Additional views include "topology" for layout shapes and port pitch, and "netlist," "placed," and "mapped" views describing the circuit itself. For the latter, standard formats such as EDIF are encapsulated inside the FLAME messages.

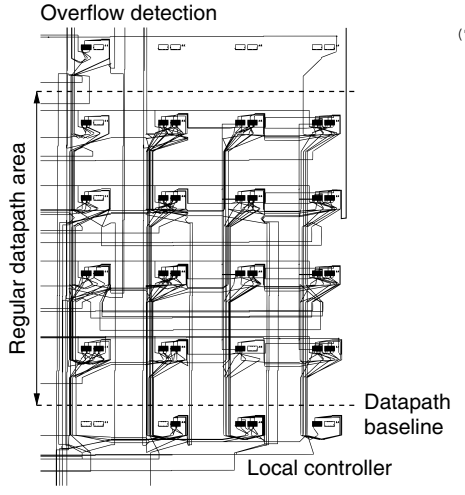
15.4.3 The Library Specification

The FLAME library specification describes a set of behaviors and interfaces. One or more of these can be attached to a hardware cell to precisely define its function for automatic use by another tool. For example, the cell of a runtime controllable adder/subtractor might have both the addition and subtraction behaviors attached. The interface carefully distinguishes between the logical (e.g., the operands of the adder) and the physical perspective (e.g., clock ports and clock enable signals). Furthermore, a FLAME interface extends beyond port specifications such as width and data type to the control characteristics of the cell. This can cover "start" and "done" signals as well as mode switches (e.g., alternating between addition and subtraction). By considering all of these aspects, another tool can choose the cell most applicable to a given task and automatically drive it correctly from the central datapath controller.

15.4.4 The Intra-module Layout

For efficiency, most module generators create circuits whose internal PEs have already been preplaced. In this case, the module generators and the datapath placement tools must agree on a set of common layout conventions. Otherwise, the regular target layout described in Section 15.1.1 will not be achievable.

Figure 15.9 shows such a regular layout, along with the FLAME description of its topology, using an unsigned 8-bit multiplier from GLACE as an example.




```
(TECHNOLOGY "Xilinx" "Virtex" "XCV50PQ240I" "-4"
 (STATUS QUERYOK "technology ok. area unit is 'CLB's...")

  (MATRIX                  ] Target device has matrix architecture

   (SHAPE                  ]
    ("CLB" (RECT 4 6 1))   ] Layout is a single 4x6 CLB rectangle
   )                       ] extending 1 unit below baseline

   (PORTLOC
    (PORTS
     ( ("a" 7 0) ("b" 7 0) ("start" 0 0) ("out" 7 0) ("done" 0 0) )
     (PITCH 2 1)           ] Port pitch for busses is 2 bits per CLB
     (COORD 0 0)

     (FOLDING LINEAR)      ] Layout is not folded
    )
   )
  )
```

FIGURE 15.9 ■ Module topology and FLAME reply.

The layout has the LSBs of the operand and result data busses aligned at a common baseline. This sequential HWOP has two irregular components, which are placed below and above the regular datapath region. For that reason, in order to preserve regularity within the stack, we had to leave extra space on the top and/or bottom to accommodate any irregularities (such as overflow detection, sign handling, etc.). All buses are spaced with a pitch of 2 bits per CLB of layout height.

15.5 THE MAPPING

Mapping techniques can be distinguished by whether they map in *N*:1 fashion (i.e., *multiple* CDFG operators into a single HWOP) or map (at least initially) in 1:1 fashion.

15.5.1 1:1 Mapping

Here each CDFG operator is considered individually. However, trade-off decisions can still occur with regard to the different HWOP alternatives for it:

Area/delay trade-offs can be performed to allow the selection of smaller but slower HWOPs for operations that are not on the critical path of the computation.

Topology matching can be performed to match the heights of the HWOPs across the datapath (Figure 15.10(a)). This can be necessary when a few HWOPs in the datapath are significantly wider than the rest (e.g., 64-bit modules in

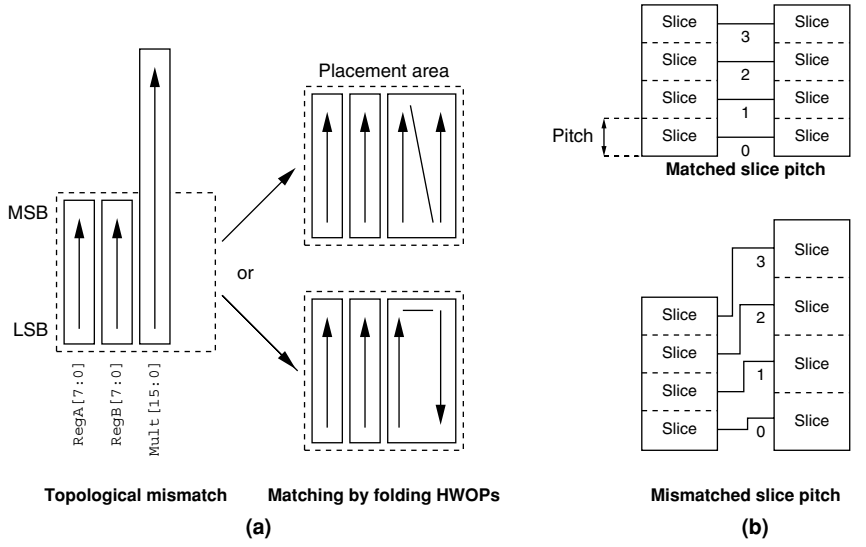


FIGURE 15.10 ■ Topology and pitch matching.

a mostly 32-bit datapath). Here regularity can be traded for area efficiency by selecting implementations for these modules that have been folded, doubling the length but halving the height.

Pitch matching occurs if modules in the library are available only with a limited number of pitch values. The goal here is to compose the datapath with the least number of pitch mismatches (Figure 15.10(b)).

Various techniques can be employed to solve these optimization problems. Since in general no single *best* solution exists for complex cases, it is practical to use an algorithm that can generate sets of good (Pareto-optimal) solutions. The SDI system [10] used a genetic algorithm in the floorplanning step to perform these calculations.

However, this approach is only applicable if a very flexible module library exists that actually gives the optimization heuristics some leeway to operate. This was the case with the PARAMOG library used in SDI, but the effort to implement this degree of flexibility is significant. More current module libraries, such as GLACE, often provide a smaller variety of implementations (generally just one) for each operator, allowing the replacement of complex heuristics with just a few simple rules for pitch and topology matching.

15.5.2 *N*:1 Mapping

In this approach, multiple operators can be mapped to a single HWOP, often using a tree-covering approach. The initial CDFG is split into a forest of trees (Figure 15.11) using techniques that splitt at multi-fanout nodes (between `B` and `D,F`) and possibly partially duplicate the operator cones rooted at the multi-fanout node (duplicating `A` into `A,A'`). While this limited approach no longer optimally solves the *graph*-covering problem, it is necessary in order to avoid the NP-completeness of computing the latter.

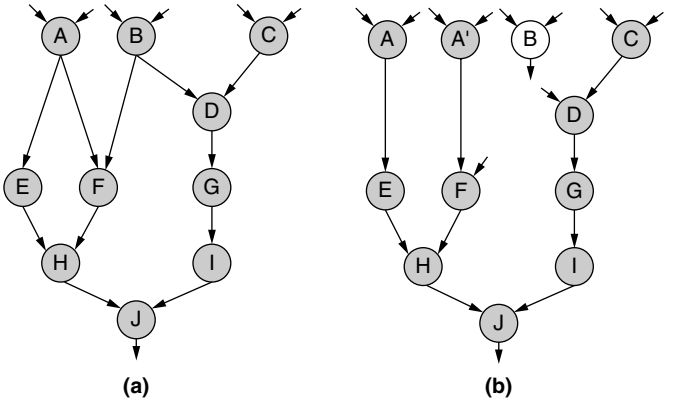


FIGURE 15.11 ■ Conversion of CDFG to a forest of trees: (a) input dataflow graph and (b) forest of dataflow trees.

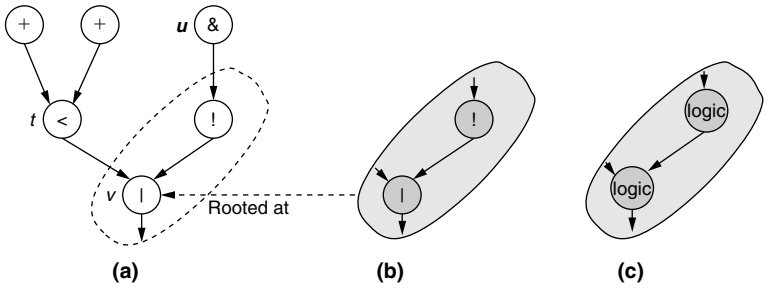


FIGURE 15.12 ■ Covering operator trees using patterns: (a) the dataflow tree, (b) the HWOP pattern P, and (c) HWOP equivalence class pattern C.

GAMA [3] employs a linear time algorithm using dynamic programming to cover the operator trees with HWOPs (Figure 15.12). This algorithm, which has its origin in the code generation steps of compilers, treats the operator(s) realizable by each HWOP as a *pattern*. Patterns are described as productions in a grammar, from which a code generator-generator creates the actual tree-covering code.

For each operator tree, the covering proceeds from the leaf nodes toward the root, applying all matching patterns that can be locally rooted at the currently examined node (`v` in the example, roots pattern `P`). A cost function computing delay and area characteristics determines the desirability of using the current pattern at this point. It is based on the cost of the currently tried pattern plus the previously computed costs (dynamic programming) of the fanin nodes to the pattern (`u,v` in the example). The "best" pattern covering each node/subtree is then selected using heuristics that either do a straight area minimization or attempt to additionally minimize delays. This best solution is then stored in the local root node, and the covering proceeds to the next node. Once the tree's root node has been matched with a best pattern, the final covering can be retrieved by starting with the root pattern and then processing the current pattern's fanin nodes. At each of these fanin nodes, the best pattern selection stored there is retrieved. This phase of the algorithm thus works recursively toward the leaves.

The algorithm has some limitations that *must* be worked around:

- First, tree covering in this fashion relies on the principle of optimality, where the combination of optimal solutions to subproblems leads to an optimal solution of the entire problem. This is indeed achievable when optimizing for minimal area. However, when attempting to minimize delays the timing criticality of operators can vary depending on later covering decisions. Thus, at the time of decision the criticality of the current node is not known.

  To mitigate this issue, GAMA attempts to estimate the criticality using an initial purely delay-oriented covering pass. Then the final covering proceeds in an area-minimizing fashion until the currently accumulated

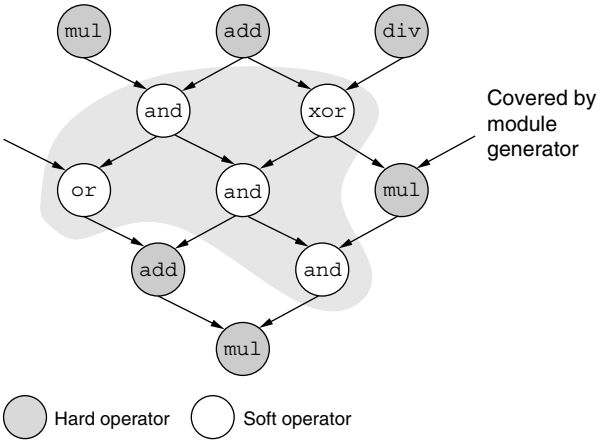


FIGURE 15.13 ■ Subgraph covering with flexible generator.

delay at a node exceeds its estimate. At this stage, the cost function is switched from area to delay minimization.

- Second, the runtime of the algorithm depends linearly on the number of patterns in the grammar (which equal different modules in the library). When the PEs of the target device are very flexible (e.g., LUT based), they can implement a wide spectrum of CDFG primitive operators (e.g., AND, OR, INV, ADD, SUB, combinations . . .). Without further refinement to the approach, a straight description of this flexibility in the grammar will lead to an explosion in the number of rules. However, in practice, many operators are *equivalent* for mapping purposes. For example, all 2-input logic operators map in exactly the same way in all patterns in which they occur. This fact can be exploited by defining equivalence classes for all operators (e.g., logic, additive) and then defining the grammar rules in terms of these classes (`C` in Figure 15.13). Combined with the factoring out of common subpatterns, this significantly reduces the complexity of the grammar.

15.5.3 The Combined Approach

A completely different approach maps some operators in a 1:1 fashion and others in an *N*:1 fashion. This combination employs powerful module generators that can generate regular modules covering entire subgraphs of the CDFG. As an example, the LogicGen tool [20] can handle arbitrary multibit logical expressions, including shifts and permutations, with optional registering of the outputs. It extracts a regular structure from the input operators and synthesizes logic-optimized bit slices using SIS [16], which are then preplaced in a regular layout. To apply LogicGen, the CDFG is searched for the largest subgraphs of plain logic modules. Each of these clusters is then handed to the tool in its entirety, allowing it to exploit reconvergent fanouts, factorization, and the

like. All operators in the cluster are thus covered by a single, LogicGen-created HWOP. Operators that are not amenable to traditional logic optimizations, such as arithmetic and memories that are usually implemented on device-specific blocks, are then mapped into corresponding HWOPs in a 1:1 manner by dedicated module generators.

15.6 PLACEMENT

The HWOPs resulting from mapping have to be placed on the device fabric. This can happen either during mapping or in a separate step afterward. Placement approaches can be classified into three groups according to the nature of the generated placement (see Figure 15.14). Purely linear techniques create a one-dimensional arrangement of HWOPs in a single stripe. Others compute a placement consisting of multiple stripes, which is sometimes referred to as 1.5 dimensional or constrained two dimensional. A last group of algorithms generates arbitrary two dimensional arrangements, an approach closely related to the classical floorplanning or macro-module scenarios in ASIC tool flows.

15.6.1 Linear Placement

An example of linear placement, GAMA [3], performs a one-dimensional placement simultaneously with the mapping step (see Figure 15.15(a)). It assumes that the external I/Os to the datapath are located on only one side of the stripe (at the right in the figure). The roots of all subtrees are placed toward this I/O side, with the root of the entire HWOP tree directly adjacent to the I/Os (`op3` in the figure). Furthermore, the HWOPs within a subtree are all placed contiguously, which means that (at least initially) HWOPs from different subtrees (here `op1` and `op2`) will not be intermingled in the placement. The placement algorithm thus consists of recursively deciding in which linear order to place the fanin HWOPs of a node.

Note that the placement order *does* affect the routing delay between different HWOPs (Figures 15.15(b) and (c)). The timing estimates calculated in this fashion are used in the cost function guiding the mapping (covering the trees

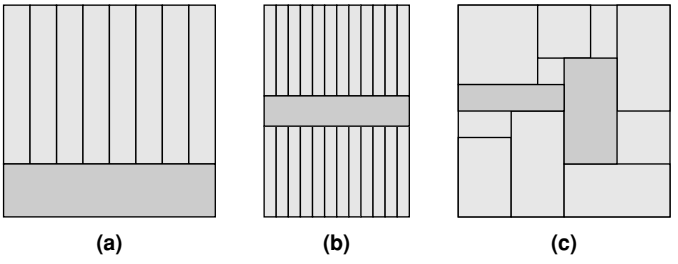


FIGURE 15.14 ■ Placements styles: (a) linear, (b) constrained two dimensional, and (c) full two dimensional.

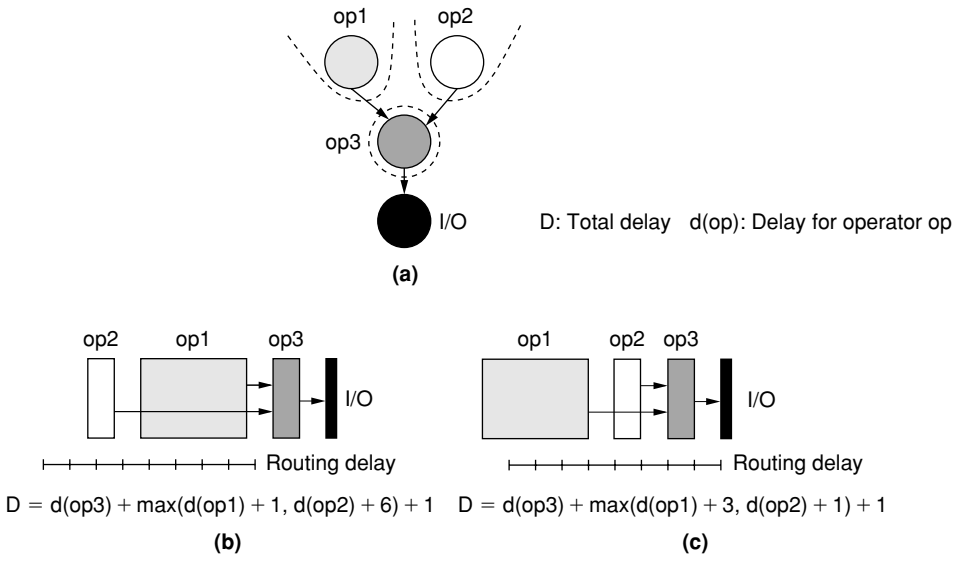


FIGURE 15.15 ■ Simultaneous tree covering and placement.

with HWOP patterns). The different trees of the forest (into which the CDFG has already been split) are placed in the stripe using a greedy algorithm that aims to place critical path trees close to each other. After this purely constructive initial placement, a greedy clustering algorithm can move HWOPs globally, across subtree and tree boundaries, in a further attempt to reduce routing delays. In practice, however, the quality gains achievable using this simple cleanup pass are negligible.

The techniques proposed by Ababei and Bazargan [1] are an example of a separate postmapping linear placement step, which employs two core algorithms to quickly determine linear placements in polynomial time. The first, shown in Figure 15.16(a), tries to heuristically compute a minimum bandwidth/minimum wirelength placement by transforming a matrix representation of the input circuit into band form and reflecting the transformation steps in HWOP swaps. This algorithm is applicable to general CDFGs.

The second, faster algorithm (Figure 15.16(b)) gives even better results, but is limited to operating on trees (similarly to GAMA). It proceeds topdown, recursively placing the nodes in a linear arrangement. The root is placed in the middle; the left subtree of the root, to the left; and the right subtree, to the right. The order in which the nodes are visited depends on the summed lengths of all HWOPs in the subtrees rooted at each node (this is called the *volume* of a node): Nodes rooting smaller volume subtrees are visited first, placing them closer to the root. In Figure 15.16, the length of all HWOPs is assumed to be 1.

In a refinement, Ababei and Bazargan [1] then extend the techniques for partial reconfiguration: A sequence of CDFGs is arranged so that previously placed

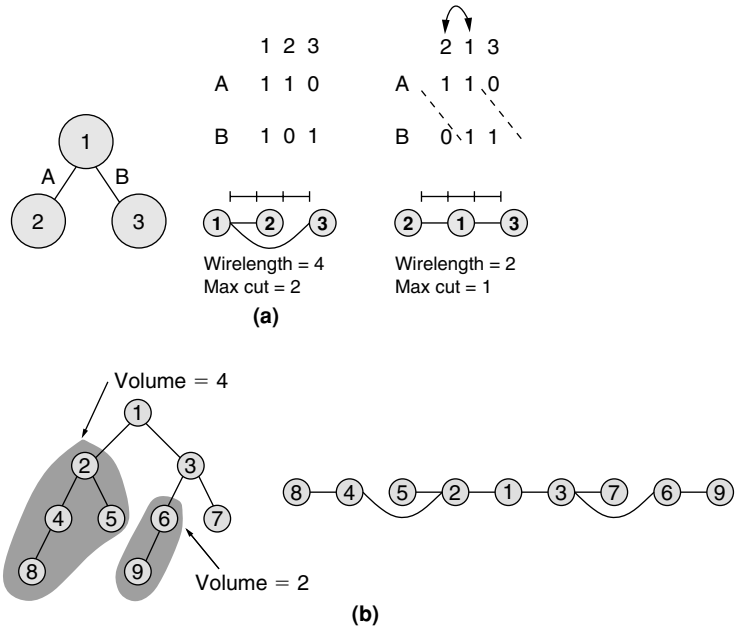


FIGURE 15.16 ■ Postmapping linear placement.

HWOPs and their interconnect can be reused in succeeding configurations, thus reducing the amount of configuration data. In the (albeit limited) experiments, up to 74 percent of HWOPs and 36 percent of inter-HWOP connectivity could be reused between configurations. However, with increased reuse, the delays and wirelengths began to deteriorate over independent placements (without reuse).

Other techniques that have been applied to compose linear stripes of HWOPs are spectral partitioning [13], genetic algorithms [10], and quadratic placement [22]. In the last case, it was determined that the quadratic placement needed to be postprocessed for by computing the optimal arrangement of HWOPs in a small window (less than or equal to five HWOPs long) using exact methods (e.g., exhaustive search, branch/bound). The process is then repeated, sliding the window across the stripe, until no further improvement can be realized.

15.6.2 Constrained Two-dimensional Placement

With the focus on linear datapath structures, published work on constrained two-dimensional or 1.5-dimensional datapath placement is sparse. Some limited results are reported by Thorns [18]: The CLAP tool first performs a clustering procedure similar to that in VPack [2] to determine the HWOPs to fit into each stripe. Then the horizontal arrangement of HWOPs inside a stripe, as well as the vertical and horizontal arrangements of entire stripes, is optimized using different moves in an adaptive simulated annealing algorithm [2], resulting

in the constrained layout shown in Figure 15.14. Again, only a limited set of benchmarks was evaluated for CLAP. However, even for a small 28-module datapath, the constrained two-dimensional approach reduced the delay by more than 20 percent over a linear placement created using a GAMA-like technique.

15.6.3 Two-dimensional Placement

A full two-dimensional placement is generally not applicable to the datapath structures discussed previously. However, if the target device architecture does not impose a specific ordering of bit significances (for example, when no hardwired carry logic is present), two-dimensional placement can be performed by treating the HWOPs as conventional macro blocks. A family of such placement algorithms has been described for the tools TS-FP [5] and Frontier [17] (Figure 15.17). Both distinguish between *hard* macros, with fixed rectangular shape, and *soft* macros, with a malleable shape. In both cases, the algorithms partition the device fabric into a number of *bins*, whose size depends on the area of the largest hard macro present in the input circuit. Smaller macros are then clustered up to the bin size to avoid wasting intrabin area.

This clustering process takes into account a number of factors: the compatibility of the macro shapes inside a bin (shapes in bin must geometrically fit in the bin bounding box), the relative size of the cluster compared to the entire circuit, the relative size of the blocks in the cluster, and the connectivity of the macros in the cluster. If, after clustering, the number of clusters exceeds the number of available bins, the size of the bins is increased and the clustering process is repeated. The clusters are then assigned to individual bins using standard placement techniques.

Intrabin placement is now performed constructively. TS-FP places hard macros from right to left by abutment, leaving the left side of the bin free for

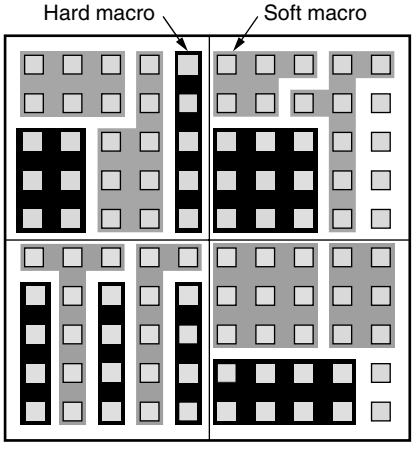


FIGURE 15.17 ■ Bin-based two-dimensional HWOP floorplanning.

soft macros. Frontier (shown in Figure 15.17) spreads the hard macros horizontally across the entire length of a bin, leaving the unused space *between* them for the soft macros. These are then placed in the free regions. TS-FP performs a geometrical minimax matching, reshaping the logic of each soft macro to fit into available space while attempting to keep the macros' initial internal placement intact. Frontier uses a simpler approach, laying a snakelike pattern across the free space, filled by sequentially selecting from the soft macro an unassigned PE that leads to the minimal overall wirelength. To improve routability, Frontier additionally employs a final low-temperature annealing pass for the PEs in the soft macros. These are allowed to move across macro and bin boundaries. The annealing start temperature is set sufficiently high to allow perturbation of the layout but low enough to ensure that the basic bin structure is kept intact.

15.7 COMPACTION

In a 1:1 mapping of simple CDFG operators (for example, trivial logic gates) to HWOPs, the PEs inside an HWOP are often not used to their full capacity. This inefficiency is worse when coarse-grained PEs are being targeted, and it accumulates across all HWOPs implementing simple operators. Figure 15.18 shows an example of this in which the functionality of a 2-input multiplexer described using simple logic HWOPs requires three PEs—even though it would completely fit in a single PE.

Compaction dissolves the boundaries of selected HWOPs and optimizes their contents as a whole, resulting in the creation of a new super-HWOP that realizes all of the original functions in a smaller/faster fashion. The procedure can generally be split into four phases:

1. Select the HWOPs to merge and compact.
2. Analyze regularity across the selected HWOPs to derive new master slices.
3. Optimize the newly discovered master slices.

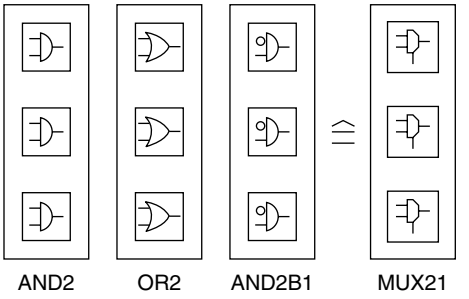


FIGURE 15.18 ■ Wasted space in the layout of very simple HWOPs.

4. Construct the super-HWOP by instantiating and placing the optimized master slices according to the regular inter-HWOP structure discovered previously.

15.7.1 Selecting HWOPs for Compaction

Two approaches have been proposed for selecting candidate HWOPs for compaction. Early work, such as the Structured Design Implementation (SDI) approach [8–10], aimed to keep a precomputed one-dimensional placement intact and so only considered connected *neighboring* HWOPs to compact. However, more recent research [21, 23] shows that better area efficiency is achievable by selecting candidates purely based on their connectivity, independent of any placement.

Additionally, depending on the actual optimization procedures to be performed on the selected candidates certain HWOPs, despite being connected and adjacently placed, might later be unsuitable for compaction. For commonly used optimization methods, this category generally includes HWOPs exploiting target device–specific features such as hardwired carry chains or fixed-function blocks (e.g., multipliers or memory blocks). Thus, their enclosing HWOPs are exempt from compaction.

15.7.2 Regularity Analysis

Since compaction is a regularity-preserving transformation, regularity aspects have to be considered both in its preparation and while it is taking place. Although methods exist to determine regular patterns in arbitrary circuits [12,15], it is much more efficient to keep track of this data from the moment of HWOP circuit generation. The method developed by Ye and colleagues [21,23] requires knowledge of the netlists at the bit slice level. SDI, supported by the powerful PARAMOG module generator library, goes beyond that by explicitly describing both regularity (in the model described in Section 15.1.1) and hierarchy (using master slice/bit slice relationships).

Based on the detailed data, SDI can consider more complicated structures for regular compaction. Figure 15.19 shows how it can isolate two new master slices and their instances from the HWOPs `ALU` and `LSHR` under compaction, even though the number of bit slices between these HWOPS differs. The inter-HWOP regularity consists of a 2-zone stack. The top zone holds a single instance of a newly discovered master slice, which consists of the original master slices `ALU4`, `TOPDWN`, and `DWN`. The second zone has two instances of a new master slice, which consists of `ALU4` and two instances `DWN`. Ye and colleagues' technique [23] would not attempt to merge these two HWOPs, as it can only compact HWOPs with the same number of bit slices.

15.7.3 Optimization Techniques

The core of compaction lies in the intermodule optimizations applied to the super-HWOP constructed by merging the original HWOPs. Here, Ye and

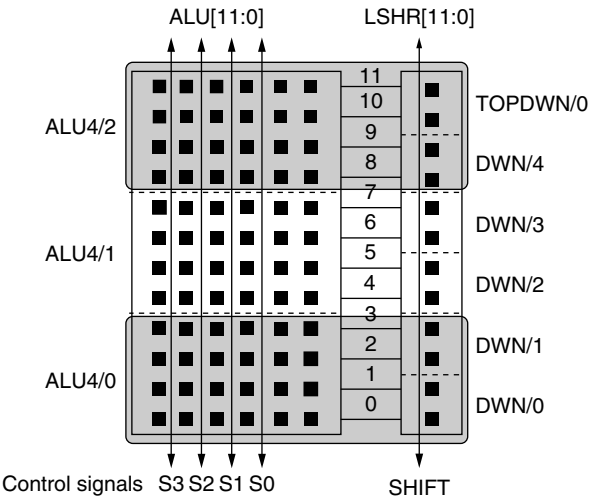


FIGURE 15.19 ■ Extracting inter-HWOP regularity.

colleagues' approach [23] performs two additional steps compared to SDI: word-wide transformations that affect entire HWOPs followed by exploiting the context (external signals) of the HWOPs under compaction. The main processing step of both SDI and the system of Ye et al. [23], however, consists of applying traditional logic synthesis and optimization algorithms at the bit slice level.

Word-level optimization

Word-level optimizations, which in Ye and colleagues' approach [23] were performed manually, alter the datapath from the structure described in the original CDFG. Two of the transformations are shown in Figure 15.20. The first, shown in Figure 15.20(a), tentatively collapses trees of multiplexers into a single wide multiplexer, modifying the select logic appropriately. If this replacement requires more area than the original version, the original version is retained. This transformation cannot be performed by optimizing at the slice level, because the multiplexer select logic is not part of the regular area holding the bit slices.

The second transformation, shown in Figure 15.20(b), is called operation reordering. It attempts to reduce area by restructuring individual multiplexers. A subcircuit, in which a multiplexer selects a single result from multiple identical operator instances, is turned into a form where multiple multiplexers select from a set of inputs feeding a single operator instance. Under the assumption that a multiplexer is smaller than the operator, this reduces area. Note, however, that this is not always the case: In many fine-grained architectures that combine LUTs and arithmetic carry logic within a logic block, both multiplexers and adders/subtractors may occupy the same number of logic blocks.

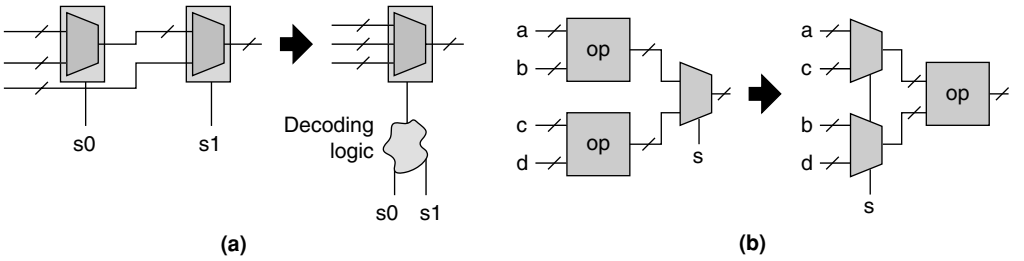


FIGURE 15.20 ■ Word-level optimizations performed by Ye et al. [23].

Furthermore, the second transformation is problematical in that it loses parallelism between the original multiple operator instances. Consider the following scenario: The operator instances 1 and 2 have data-dependent execution times t_1 and t_2, and the select input arrives at t_s after the operands of the operators. In the original case, both computations would be speculatively performed in parallel. The delay of the entire structure is then $\max(t_s, t_1)$ if the result of the first operator is selected, and $\max(t_s, t_2)$ otherwise. In effect, the delay of the select input hides part of the operator delay. In the reordered form, the operator can begin computation only *after* the select input has become valid, leading to total delays of $t_s + t_1$ and $t_s + t_2$, respectively.

The ramifications of such a transformation can be appraised to their full extent only when *building* the CDFG in the first place—for example, when considering instruction-level parallelism in a hardware compiler. At the same time, the multiplexer tree collapsing could also be performed, dispensing with a special optimization pass later in the design flow. Instead, the CDFG would contain generic multiplexer operator nodes with a varying number of inputs. During the mapping step, the module library would determine the best realization for each operator, also considering global issues such as the criticality of their signal paths.

Context-sensitive optimization

The tool flow designed by Ye and colleagues [23] then performs an additional suite of optimizations that also considers the super-HWOP in the *context* of the surrounding datapath (Figure 15.21). To this end, it partitions the super-HWOP into m-bit-wide *superslices*, each of which may thus consist of multiple bit slices. Next, the external ports of each superslice are examined for certain connectivity patterns and the presence of constant values. The actual optimizations are then performed in this superslice-specific context.

Constant inputs are absorbed for each of the superslices (Figure 15.21(a)). Similarly, nets that connect slice inputs directly to outputs are also pulled into the slice (Figure 15.21(b)). Multiple slice inputs all sourced by the same external signal are replaced by a single input that fans out to the original internal sinks (Figure 15.21(c)).

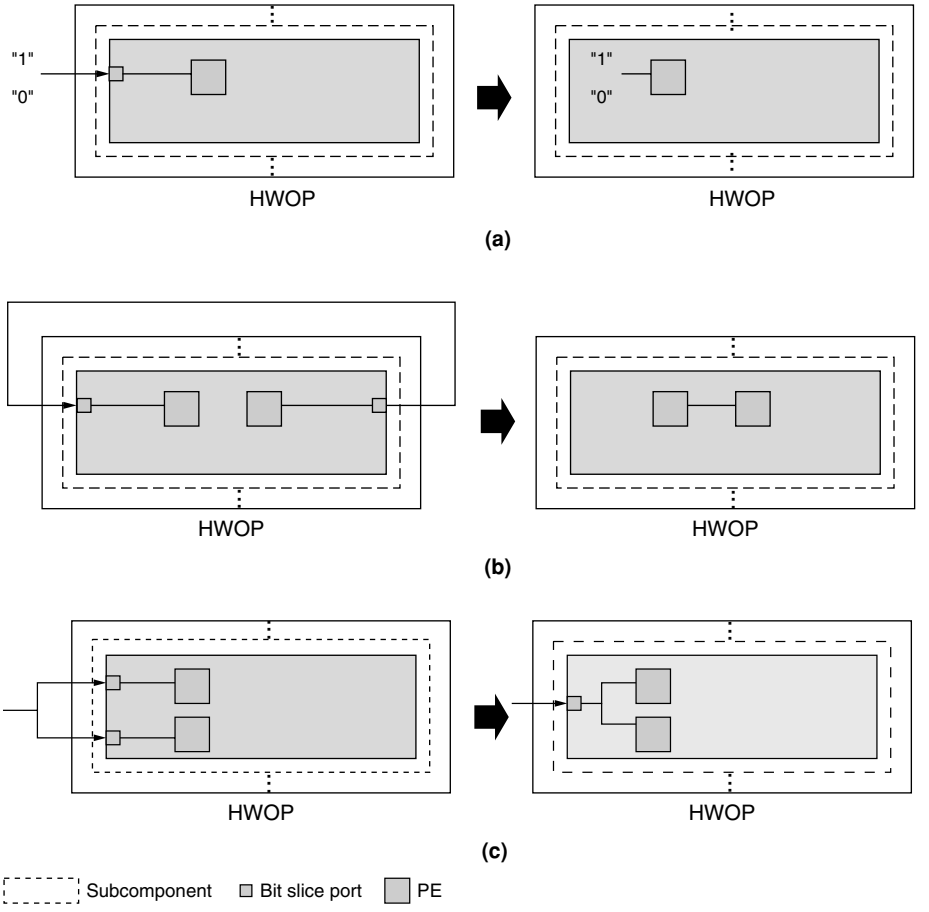


FIGURE 15.21 ■ Context-sensitive optimizations performed by Ye et al. [23].

These transformations occur *only* if all bit slices within a superslice have *identical* context (e.g., all bit slice input ports `a` within a superslice have the constant value 0 applied from the outside). Otherwise, the superslice is left unchanged.

The quantity m is thus a control for the internal regularity of the super-HWOP. With $m = 1$, the super-HWOP is partitioned into *width* superslices, each consisting only of a single 1-bit-wide bit slice. Each of these narrow superslices is thus affected by only very limited context: A superslice's single bit slice can be perfectly matched to its context (e.g., allowing the absorption of even irregular constant input patterns into each slice) in the super-HWOP. However, while allowing a large degree of optimization, this setting of $m = 1$ potentially introduces significant irregularity into the optimized super-HWOP (it may end up consisting of completely different bit slices). At the other extreme, with m = *width*, the super-HWOP is covered by a single superslice containing m 1-bit-wide

bit slices. Here optimization will occur only if the context affects *all* bit slices within the single superslice identically. Thus, even the optimized super-HWOP will be completely regular (composed only of identical bit slices). With the context required to be identical for more bit slices, however, fewer optimization opportunities arise. In Ye and colleagues' approach [23], a value of $m = 4$ is suggested as a good trade-off between widespread optimization and the preservation of a regular structure.

In effect, the idea of superslices is similar to the *zone* concept introduced in Section 15.5.1, although zones, with their variable granularity, remain more flexible than superslices, with their fixed granularity.

Logic optimization

In logic optimization, the netlists of the HWOPs under compaction are merged into HWOP-spanning bit slices (possibly newly discovered, as discussed in Section 15.7.2). The resulting larger merged netlists are then passed to conventional logic synthesis tools that can exploit the additional optimization opportunities resulting from them.

In addition to this slice-internal optimization, the system of Ye and colleagues [23] can specialize the bit slices by considering the constant *external* inputs and connections that were discovered in the context-sensitive analysis pass.

15.7.4 Building the Super-HWOP

The optimization phase of compaction changes the circuit structure. Thus, any regular placement created by a generator is invalidated. Ye and colleagues' tool flow [23], which concentrates on measuring regularity and area overheads, does not perform the further processing steps itself. Instead, the resulting optimized bit-slice netlists are passed to standard place-and-route tools for further handling. In contrast, Structured Design Implementation (SDI), additionally aiming at delay minimization, attempts to restore a regular placement for the optimized super-HWOP. This *micro-placement* step, shown in Figure 15.22, exploits regularity by operating at the master slice level. The results are then automatically replicated across the entire super-HWOP according to its zone structure.

Microplacement operates on cells (LUT and FF blocks), and proceeds in two phases:

1. The placement of cells horizontally, grouped into columns (Figure 15.22(a)). This is performed across all master slices, ensuring that cells sharing a control net are located adjacently to a vertical routing channel. Such an arrangement allows the efficient routing of high-fanout control nets on vertical long lines. Analogously, cells on interslice nets are horizontally aligned to allow short-distance routing. The remaining cells are placed in a timing-driven fashion, using estimates for the as yet unknown vertical position. This placement phase optimizes the super-HWOP in the geometric context of the datapath by constraining the master slice I/O ports to the appropriate sides of the layout.

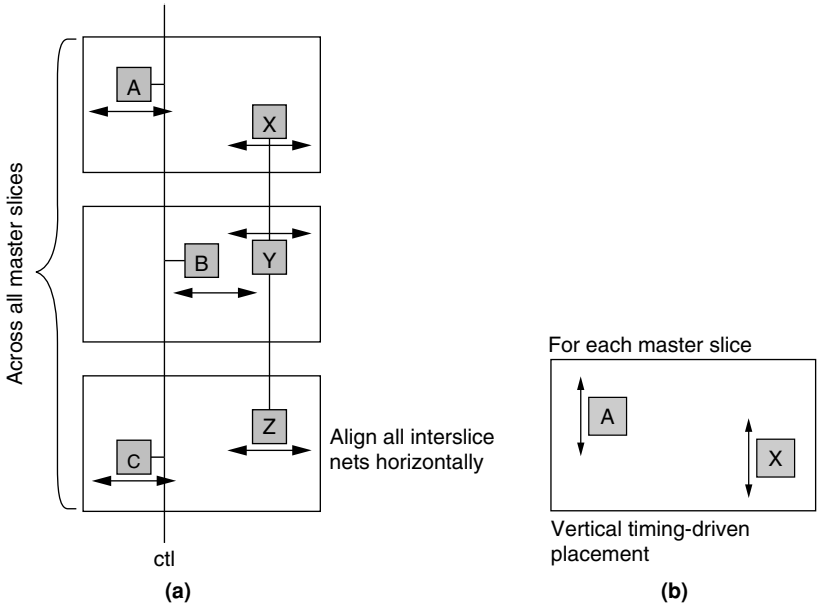


FIGURE 15.22 ■ Horizontal and vertical microplacement to restore regularity to compacted super-HWOP.

2. The placement of cells within the columns vertically (Figure 15.22(b)). This step looks across master slice boundaries only initially when performing a timing analysis on the entire super-HWOP. After annotating the timing criticalities calculated in this manner on the master slice ports, each master slice is placed independently in a purely timing-driven fashion. The timing model used here models the intricacies of the target device routing network and leads to measurably better results than simple Manhattan distances.

Since the microplacement results are replicated according to the regular structure previously determined for the super-HWOP, it is advantageous to employ high-quality algorithms. To this end, SDI uses a combination of well-converging heuristics and exact integer linear programming (ILP)-based methods. The latter are feasible because of the separation of the placement problem into horizontal and vertical phases, and the relatively small circuit size of the master slices (compared to the entire super-HWOP).

15.7.5 Discussion

Implementing a circuit in a regular bit-sliced fashion is generally associated with some area overhead compared to synthesizing/optimizing the circuit in an irregular flat manner. The reason is that the bit-slice boundaries prevent the exploitation of cross-slice optimization opportunities. The system devised by Ye and colleagues [23], with its additional interslice optimizations, observed area overheads of between 0 percent and 7.4 percent for superslice granularity

values of $m = 1$ (fully irregular) and $m = 32$ (fully regular with a width of 32 bits), respectively. For SDI, which lacks these optimizations, area increases of up to 17 percent were observed over the flat solution. However, by scrupulously maintaining a regular structure, SDI was able to reduce the total delay in the circuit by up to 33 percent over the flat implementation. A combination of the interslice optimizations of Ye and colleagues [23] with the microplacement of SDI appears to be promising to achieve further gains.

15.8 SUMMARY AND FUTURE WORK

This chapter presented an overview of some of the many issues to consider when realizing datapaths on reconfigurable logic devices. The aspect of *regularity* is a crucial one and must be considered both at the level of the target device architecture and during the operation of the EDA tools. Module generators are an efficient means to actually create the circuits making up the datapath. However, in addition they must offer sufficient metadata to the rest of the tool flow as a base for effective transformation and optimization steps.

With increasing requirements on datapath performance, tool flows and algorithms must keep up with improvements in device architectures. All of the techniques described here have the potential for further refinement. Refinement opportunities include module generators that better support specialization, floorplanning with constrained two-dimensional placement, and a compaction technique in which the best of these refinements is combined.

References

[1] C. Ababei, K. Bazargan. Non-contiguous linear placement for reconfigurable fabrics. *Proceedings of the of the Reconfigurable Architectures Workshop*, 2004.

[2] V. Betz, J. Rose, A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer, 1999.

[3] T. J. Callahan, P. Chong, A. DeHon, J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. *Proceedings of the of the International Symposium on Field-Programmable Gate Arrays*, 1998.

[4] T. Callahan, R. Hauser, J. Wawrzynek. The GARP architecture and C compiler. *IEEE Computer* 33(4), 2000.

[5] J. M. Emmert, D. Bhati. A methodology for fast FPGA floorplanning. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1999.

[6] B. Hutchings, P. Bellows. J. Hawkins, S. Hemmert. A CAD suite for high-performance FPGA design. *Proceedings of the of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[7] N. Kasprzyk, A. Koch. High-level-language compilation for reconfigurable computers. *Proceedings of the International Conference on Reconfigurable Communication-centric SoCs*, 2005.

[8] A. Koch. Module compaction in FPGA-based regular datapaths. *Proceedings of the Design Automation Conference*, 1996.

[9] A. Koch. Structured design implementation—A strategy for implementing regular datapaths on FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1996.
[10] A. Koch. *Regular Datapaths on Field-Programmable Gate Arrays*. CS doctoral thesis, technical, University of Braunschweig, 1997.
[11] A. Koch. On tool integration in high-performance FPGA design flows. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 1999.
[12] T. Kutzschebauch, L. Stok. Regularity-driven logic synthesis. *Proceedings of the International Conference on Computer-Aided Design*, 2000
[13] J. Li, J. Lillis, L. T. Liu, C. K. Cheng. New spectral linear placement and clustering approach. *Proceedings of the Design Automation Conference*, 1996.
[14] T. Neumann, A. Koch. A generic library for adaptive computing environments. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2001.
[15] R. Nijssen, J. Jess. Two-dimensional datapath regularity extraction. *Proceedings of the ACM SIGDA Physical Design Workshop*, 1996.
[16] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, et al. SIS: A system for sequential circuit synthesis. *EECS Memorandum No. UCB/ERL M92/41*, University of California, Berkeley, 1992.
[17] R. Tessier. Frontier: A fast placement system for FPGAs. *Proceedings of the International Conference on VLSI*, 1999.
[18] F. Thorns. *CLAP—Clustering and placement*. Diploma thesis, Technical University of Braunschweig, 2003.
[19] C. C. Vi, D. Lewis. Area-speed trade-offs for hierarchical field-programmable gate arrays. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1996.
[20] C. Wewetzer. *A Universal Generator for Logic Circuits on FPGAs*. Diploma thesis, Technical University of Braunschweig, 2005.
[21] A. G. Ye. *Field-Programmable Gate Array Architectures and Algorithms Optimized for Implementing Datapath Circuits*. Doctoral thesis, University of Toronto, 2004.
[22] T. T. Ye, G. De Micheli. Data path placement with regularity. *Proceedings of the International Conference on Computer-Aided Design*, 2000.
[23] A. G. Ye, J. Rose, D. Lewis. Synthesizing datapath circuits for FPGAs with emphasis on area minimization. *Proceedings of the International Conference on Field-Programmable Technology*, 2002.
[24] A. G. Ye, J. Rose. Measuring and utilizing the correlation between signal connectivity and signal positioning for FPGAs containing multibit building blocks. *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2005.