On Tool Integration in High-Performance FPGA Design Flows

Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Gaußstr. 11, D-38106 Braunschweig, Germany
koch@eis.cs.tu-bs.de

Abstract. High-performance design flows for FPGAs often rely on module generators to counter coarse logic-block granularity and limited routing resources, However, the very flexibility of current generator systems complicates their integration and automatic use in the entire tool flow. As a solution to these problems, we have introduced FLAME, a common model to express generator capabilities and module characteristics to clients such as synthesis and floorplanning tools. By offering a unified view of heterogeneous generator libraries, FLAME allows the seamless and efficient use of flexible generation techniques in automatic compilation flows targeting configurable hardware.

1 Introduction

While well known for decades, the use of algorithmic module generation in VLSI design flows has recently been exploited with renewed interest. Especially for FPGAs with their limited interconnect resources and coarse-grain logic blocks, module generators have traditionally been the tool of choice to quickly provide fast and dense circuits [1–4]. The need for structured circuit generation becomes even more pronounced when FPGAs are not just used to implement glue logic, but as compute elements in configurable computing machines (FCCM). Many research efforts on automatic compilation to FCCM targets include module generation as a crucial step [5–7].

However, the very flexibility of parametrized generators makes their integration with the main design flow (synthesis, floorplanning, place and route) difficult. The solution quality of any optimization performed by these tools will be limited by the information offered *about* the available module alternatives. For example, modern generators can completely restructure a general circuit description to optimally process constant inputs [3]. In an FCCM, this restructuring could even be postponed to take place at run-time (once the actual data values are available) instead of at compilation time. When one of the main flow tools requires information about a module this flexible, the sheer volume of the design space covered by each generator (such as behavior, time, area, power, and layout) precludes a simple enumeration of all variations.

Furthermore, common formats for characterizing library elements [8–10] are tailored for ASICs and do not cover the higher-level concepts required for efficient module embedding (e.g., function, control interface, sequential timing, etc.).

In contrast to the practical tutorial presented previously [14], this work concentrates on a high-level overview of the concepts underlying our proposed solution.

2 FLAME

FLAME, the Flexible API for Module-based Environments aims to resolve these difficulties. It is not a module generation system itself, but a wrapper around heterogeneous module libraries.

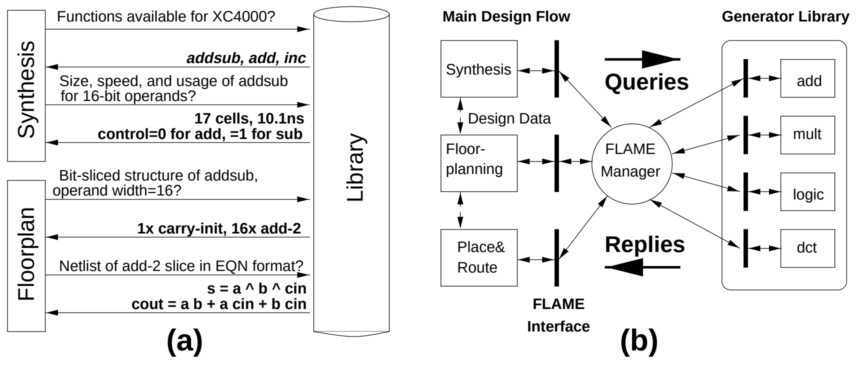
FLAME offers client tools (synthesis, floorplanning) a single unified means to access generators (termed servers) and determine module characteristics without regard for their original vendor, target technology, or implementation.

To enable this functionality, FLAME specifies a common model independent of an implementation language or platform to express generator capabilities and module design characteristics to client tools. Instances of this model can be represented in a portable manner and are exchanged between clients and servers via a simple, yet flexible active interface (API). Note that since FLAME aims at inter-tool communication, a GUI is not required.

3 Active Interface

To avoid the problems associated with static module description files, FLAME relies on a dynamic exchange of queries and replies between clients and servers instead. Figure 1.a shows such a sample dialog, using synthesis and floorplanning as clients in the main design flow. The retrieval of information (which could also include complete netlists or layouts) proceeds as a stepwise refinement by incrementally tightening constraints to narrow down the range of possible solutions. In addition to parameters such as operand widths and data types, queries are also hierarchically scoped (see Section 6) to further constrain solutions.

Figure 1. (a) Sample query/reply scenario, (b) System architecture



This approach reduces computation times by allowing results to be calculated only to the detail actually required by the current query abstraction level. For example, during the initial synthesis phases, only timing and area estimations are required. The generation of a complete layout at this step would be wasteful and is not necessary. On the other hand, once synthesis has decided on a certain implementation, the floorplanning phase requires precise shape information for optimal operation. This more detailed data, which might also be more time consuming to compute (especially for data-dependent units generated at FCCM run-time), can then be provided only for the single selected implementation. In a similar manner, local logic optimization [11] can retrieve netlists from a per-bit-slice scope instead of trying to find regularity in the flat, unstructured netlists of the entire module.

The internal FLAME architecture is shown in Figure 1.b. The communication between clients and servers is mediated by the FLAME Manager, which accepts queries from clients and forwards them to the appropriate servers. Next, the individual replies from the servers are assembled into a composite reply and passed back to the clients. Further capabilities can include the automatic

Table 1. FLAME views

Name	Contains	Required?
behavior	functionality, logical interface	yes
synthesis	timing, area, power, control, and physical interface	yes
structure	regularity (bit-slices), hierarchy	no
topology	layout shape, interconnect densities	no
netlist	device-independent netlist	yes
mapped	device-dependent netlist	no
placed	device-dependent placement	no
routed	device-dependent layout	no
simulation	simulation model	yes

translation between different FLAME representations and gatewaying between various communication mechanisms (direct function call, program invocation, network). In addition, a memoization mechanism in the FLAME Manager can serve to further increase efficiency by answering previously encountered requests out of a cache without forwarding them again to the generators.

4 Data Representations

FLAME data can be represented in multiple formats, which generally trade-off ease-of-use and portability vs. efficiency. A human readable text representation is easiest to process even for simple tools (such as Perl scripts), while a tokenized representation (keywords replaced by integer tokens) can be processed and transferred more efficiently between tools implemented in different languages and environments. For maximum performance, an implementation-language specific binary representation (generally built using pointers or references, e.g., [12]) can be passed between integrated tools using the same language binding. This approach results in much lower communications overhead than the traditional file-based approach and is one of the unique characteristics of FLAME.

5 Views

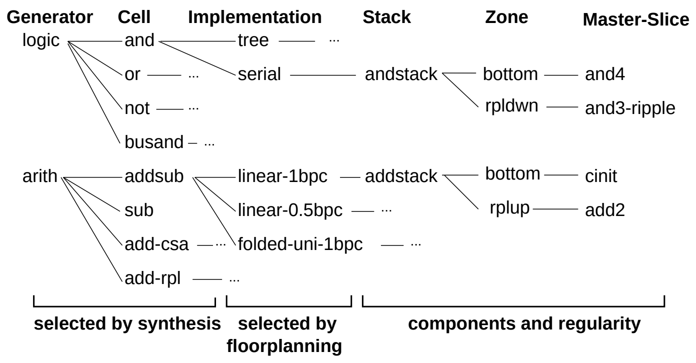
Views serve to group related items of data. Thus, a client only has to ask for a single view, instead of querying for each individual datum. For the server, views reduce the workload by allowing it to compute only the information in the specified view, instead of all information fulfilling the query constraints. As mentioned earlier, a synthesis tool is generally not interested in a placed and routed layout during data flow-graph (DFG) covering. It uses the “synthesis” view to only consider appropriate aspects such as module function, timing, and area. Table 1 shows some of the supported views.

FLAME defines required and optional views. The latter are used for advanced optimizations [15] [11], or to allow a client to pre-compute information for fast processing. This is most applicable to the “placed” view, in which the generator can offer a pre-placed layout which takes maximum advantage of a regular circuit structure.

6 Design Hierarchy

As another measure to limit the amount of data exchanged between clients and servers, FLAME operates on a hierarchy of design entities (Figure 2), where lower levels (more detail) are only accessed when required.

Figure 2. FLAME design entities



A *generator* is a concrete piece of code that creates different views of a circuit according to parametrized descriptions. A *cell* is a functional unit that can be generated by the specific generator. Different cells may supply identical or different functions and interfaces. An *implementation* is an actual circuit conforming to the behavior and interface of the enclosing cell. All implementations of a cell must have the same function and interface. In general, they differ in more physical aspects such as layout footprint, logical pitch, topology, low-level timing, and power consumption. The cell is composed from *sub-modules* (instances of other cells) and *stacks*. Stacks contain one or more *zones* of replicated logic (Figure 3.a). The smallest design unit is the *master-slice*, which will be iterated to form the zone.

To illustrate the hierarchy, consider the following example: A generator *arith* might provide the cells *addsub* (switchable adder-subtractor), *sub* (subtractor), *add-csa* (adder), and *add-rpl* (adder). The adder-subtractor is available in three implementations (*linear-1bpc*, *linear-0.5bpc*, and *folded-uni-1bpc*) that realize it in different physical layout styles. In the implementation *linear-1bpc*, the circuit consists of a single stack *addstack* defining two zones, *bottom* and *rplup*. The zone *bottom* holds a single iteration of the master-slice *cinit* (carry initialization), while the zone *rplup* contains multiple (up to the desired operand width) iterations of the master-slice *add2* (full-adder bit-slice).

When the synthesis system is covering the data-flow graph of the current design with modules in the library, it selects suitable cells. Since all implementations of a cell are guaranteed to have the same external interface (which includes, e.g., control specifications and pipelining), the floorplanner can then perform lower-level optimizations, such as matching the physical layouts across all modules by selecting appropriate implementations within the cells [15], while keeping the synthesized global controller intact.

Stacks, zones, and slices describe the regularity aspects of a design (e.g., the composition of a 16-bit ripple adder by replicating a 1-bit adder). This information can be exploited in later design flow steps to perform further optimizations (e.g, regular merging of modules) [11] and reduce computation times (by solving small problems and replicating the results) [15].

The use of hierarchical instead of flat composition also allows sophisticated run-time optimizations, such as reaching into a module to substitute faster/larger or slower/smaller submodules depending on whether they are located on or off the critical path at the system level.

7 Target Technology

FLAME abstracts key device features in a portable manner. This enables the development of technology-specific generators that offer a technology-independent interface for instantiation and composition. Design tools are thus presented with a uniform view of the different underlying FPGA architectures, allowing both the easy retargeting of designs between architectures as well as the development of portable CAD tools supporting multiple technologies.

Area is measured as a vector containing an entry for each kind of active resource (logic block, memories, arithmetic unit etc.), giving both the number of units used and the total number available. Thus, the area cost in terms of "scarcity" of resources is easily computed.

Routing density, which indicates the amount and/or type of routing resources used within a layout, is represented similar to area. This information is required for optimal datapath composition to avoid placing a very dense module in the middle of a linear arrangement of modules, thus making it very difficult to route signals through this obstacle.

The precise and complete description of the capabilities of on-chip storage or tri-state elements is crucial for efficient circuit synthesis. Modeled are the trigger type (edge or level) and the presence of set, reset, and enable signals including their polarities.

8 Function

In many compilation flows for FCCMs, the synthesis tool/compiler builds a data-flow graph of the operations required by the user program, which must then be mapped into hardware units. A common solution to this problem performs a covering of the operations in the data-flow graph with the hardware operations available in the module library, e.g. [16]. For this process (which in itself is not covered by FLAME), the function of each cell offered by the module generators must be described in a standardized way.

In FLAME, combinational logic and arithmetic functions are best formulated as an expression in infix notation using standard operators. For example, the expression `Y=A+B` could describe the semantics of an adder, while `Y=A&B&C` characterizes a 3-input AND.

Primitive (e.g., muxes, register banks, etc.) or high-level (FIR, FFT, processors, etc.) cells, which cannot conveniently be described by the infix expression, are accessed using well-known names. E.g, `LPM_REG` could be used for a register bank when using LPM [17] as guideline for module names.

In addition to modules performing a single function, FLAME also allows the specification of units computing multiple functions in parallel and/or in sequence. E.g., a cell could compute the sum and difference of operands simultaneously, or be a controllable adder/subtractor which can perform either one of these operations in sequence.

9 Interfaces

Describing only the function(s) of a cell is of course insufficient to allow its actual use in a circuit. To this end, the cell interface, which consists of port (how to connect the cell) and control (how to use the cell, Section 10) information has to be specified.

FLAME distinguishes between logical and physical interfaces. E.g., a sample cell *MulDiv* might logically accept operands *A* and *B* to compute a product or quotient *PQ*. Physically, however, it could accept the operands on the rising edges of successive clock cycles at a single input port *D*, after an opcode choosing either multiplication or division has been loaded (also through *D*, but with an asserted control input *Op*). When a control output *Done* becomes asserted, the result can then be retrieved from the physical output *Y*. FLAME easily allows the description of such a logical-to-physical mapping. By hiding these details in the physical interface, the initial DFG covering pass of synthesis only deals with the logical interface. This should lead to simplified tools and quicker execution (since fewer details have to be considered).

Both kinds of interface allow the definition of input, output and bidirectional ports as well as the application of constant and late-bound (loaded at run-time into the FPGA) values. FLAME distinguishes between data and control ports (to guide a regular datapath layout [15]). All ports are also characterized in terms of data types and bit widths. Currently, declarations for signed and unsigned integers as well as for fixed precision numbers are defined. If available on the target technology, outputs can also be registered and/or tristated on request.

10 Control

While the behavioral view describes the functionality of a cell, it does not specify how the functions can actually be *used*. This might range from a simple addition/subtraction switch by changing the value on a control input from 0 to 1, to a multi-cycle sequence of loading operands and opcodes into a complex functional unit that signals the end of a variable-length execution by asserting a control output (e.g., the example in Section 9). In FLAME, such control sequences for combinational and sequential cells can be specified in terms of six primitive statements: *LEVEL* asserts a signal combinationally (no cycles pass). *POSEDGE* and *NEGEDGE* assert signals in time for the positive (negative) clock edge, they take one clock cycle of time. *CONTINUE* waits until the specified signals have the specified values, but takes no time in itself. *START* marks an entry point for a new thread of control, which is spawned using *RESTART*. With this information, synthesis can automatically generate an appropriate FSM to integrate the cell into the host circuit. As an example, the control interface of the sample cell *MulDiv* (Section 9) in multiplication mode could be formulated as:

```
(POSEDGE  (("Op")   1) (("D")  0x42))  ; load magic number for multiply mode
(LEVEL    (("Op")   0))                ; now switch to operand input mode
(START)                                ; label for starting another multiply operation
(POSEDGE  (("D")  ("A")))              ; load first operand A through port D
(POSEDGE  (("D")  ("B")))              ; load second operand B through port D
(CONTINUE (("Done") 1))                ; wait until port Done becomes asserted
(RESTART)                              ; load new operands (fork to START) ...
(POSEDGE  (("Y")  ("PQ")))             ; ...  and simultaneously unload result through PQ
```

11 Timing

Once it is clear what a cell does and how it can be used, the single most important characteristic for high-performance designs is the cell timing. While the path-based modeling approach is more precise, it often becomes unmanageable due to the huge number of timing paths in larger circuits. The complexity of the slack-based model (timing specified in terms of required and arrival times) grows only linearly in the number of ports, but becomes pessimistic for circuits with wildly differing internal path lengths. FLAME supports both specifications and leaves the choice to the generator implementor.

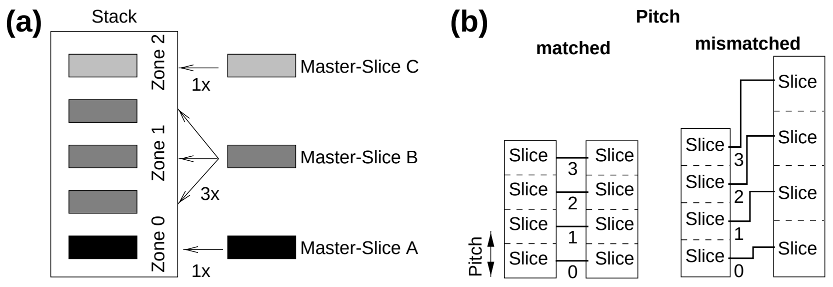
In contrast to the simple combinational or CLK$\rightarrow$Q delay values often found in ASIC-specific cell descriptions [9] [10], the often highly pipelined cells on FPGAs have additional requirements. All time specifications in FLAME also contain the number of the sequential cycle when an input must be valid or an output arrives (latency). This allows the description of pipelined or multi-cycle operations to synthesis, which can then also insert an appropriate number of deskewing registers when paths with different latencies converge. Furthermore, FLAME also allows the specification of the longest combinational delay before the first cell-internal storage element is reached. Together with the combinational delay beyond the last cell-internal storage element, the longest Q$\rightarrow$D delay can be calculated across cell boundaries, thus allowing the determination of the datapath-wide clock period. Additionally, cells also specify their throughput as the number of clock cycles per datum to describe the performance of partially pipelined units.

To allow the delay comparison between units with variable execution times (Section 10), their timing is specified separately as best case, average, and worst case timing. This expanded information enables clients to make timing-based trade-off decisions.

12 Structure

As shown in [15], the exploitation of regular structures during datapath synthesis can lead to considerable reductions both in compile time as well as in delay/clock period. While it is to some degree possible to extract regularity from an unstructured netlist, e.g., [19], it is far more efficient to let the generator actually provide this information to the client. Since a generator usually composes regular circuits in a regular manner (e.g., by iterating a sub-circuit in a for-loop), this structural knowledge is already available, and just needs to be made visible externally.

Figure 3. (a) Regular Structure, (b) Pitch Matching



Following this approach, FLAME allows the description of regular structures based on the iteration (replication) of master-slices to compose zones of replicated logic (Figure 3.a). Note that this replication also includes regular connectivity (typically, but not limited to, next-neighbor connectivity as used in ripple-carry adders and shift registers). One or more zones make up a stack. E.g., a ripple-carry adder might consist of a zone for initializing the carry chain, a zone of replicated full-adders, and a zone for processing an overflow bit. An entire cell is then assembled from one or more stacks and/or sub-modules.

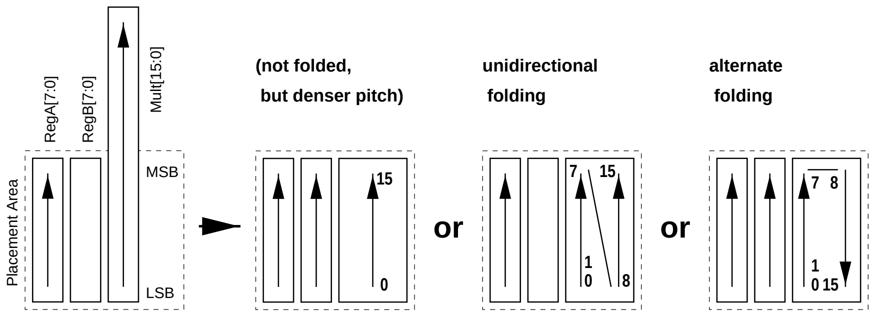
13 Topology

After synthesis has selected cells to cover the data-flow graph, (relying on the timing and area data retrieved from the generators), floorplanning [15] takes a more physical view to construct a high-performance datapath. To this end, FLAME models the topological characteristics of the layout. Included are the shape of the layout, the folding style for very wide modules, the pitch (spacing between bits of a regular bus-port), and the density (the availability of interconnect to route through the module). Furthermore, modules may be constrained to specific locations (e.g., to take advantage of specific function units such as memories).

Depending on the available chip area for the datapath, modules that are too tall might have to be folded to fit (Figure 4). In that case, the floorplanner needs to ensure that all of these modules use a consistent folding style (long routing delays would occur otherwise). Matching port pitch across all modules in the datapath also aims at reducing routing delays (Figure 3.b). As indicated in Section 7, the routing density inside a module is described to allow congestion management at the datapath level before performing detailed routing.

Combining regular structure exploitation with floorplanning as described in Sections 12 and 13 has improved design performance by as much as 33% and reduced tool runtimes by up to 80% [15].

Figure 4. Layout folding



14 Embedded Foreign Data

FLAME concentrates on defining new abstractions for information *about* a module. However, once a certain implementation has been selected based on that data, the generated circuit itself has to be retrieved for further processing. Views such as “netlist”, “placed”, and “simulation” provide these lower-level design aspects. Since well established formats (possibly technology/vendor specific) exist for all of these representations, FLAME does not attempt to specify yet another “standard” in these areas. Instead, data in an existing format is wrapped in a FLAME shell for transfer. E.g., netlist data could be sent as EDIF, placement information on the XC4000 as XNF, and simulation models as VHDL or Verilog.

15 Results

The current FLAME technology demonstrator [12], containing the base library and a sample FLAME Manager, offers unified access to generators developed ad-hoc as well as to modules in the Xilinx CoreGen package [20]. In the near future, this prototype will be extended to also allow control of JHDL generators [21]. Since the system relies entirely on the high-performance FLAME binary representation (Sec. 4), and avoids cumbersome file operations, the overhead of the FLAME interface is negligible.

FLAME is documented in detail by a comprehensive manual [13]. Furthermore, a portable object-oriented model using Unified Modeling Language (UML) [22] for the binary representation has been developed. Since this model is exportable in multiple programming languages (e.g., C++, Java, Ada), it could be used as a starting point for individual implementation efforts. The prototype has been created in this manner by elaborating a model exported to Java.

The interface is currently used in academic as well as in industrial research projects on the next generation of EDA systems for reconfigurable computers,

16 Summary

We presented the motivations for and capabilities of FLAME, a new method for tool integration in generator-based EDA suites for FPGAs. FLAME encompasses all aspects of a design flow beginning with synthesis and ending at layout, modeling them by either introducing new abstractions, or encapsulating existing ones. By allowing the main flow tools easy and efficient access to a wide range of well-defined module parameters and representations, the full flexibility of a generator-based implementation method may be harnessed to create highly optimized circuits without human intervention.

Appendix: Examples

Loading the module library is generally the first step in any compilation flow run. In FLAME, this is formulated as a query for the “behavior” view (we are interested in all module functions) with the only constraint being the target technology. The corresponding FLAME expression in textual representation [13] is shown on the left, while the equivalent binary representation composed using the FLAME/Java interface [12] is shown in the right column.

```
(QUERY 1 1                        <--->   Query q = new Query(1, 1
 (TECHNOLOGY "Xilinx" "XC4000E"             new Technology("Xilinx", "XC4000E",
             "XC4003EPG191" "-3"                           "XC4003EPG191", "-3",
  (VIEWS
   (VIEW "behavior"))))                      new VBehavior()));
```

For brevity, we assume that our library contains only a single cell, a bus-wide gate switchable between AND/OR operations (example code printed in two columns).

```
(REPLY 1 1                                (INTERFACE
 (TECHNOLOGY "Xilinx" "XC4000E"            (LOGICAL
             "XC4003EPG191" "-3"             (INPUT  (("A") ) (("B") ))
  (VIEWS                                    (OUTPUT (("Y") ))))
   (VIEW "behavior"                       (BEHAVIOR
    (STATUS QUERYOK "view ok")             ("andmode" (FUNCTION (INFIX "Y=A&B")))
    (GENERATOR "andor" 1                   ("ormode"  (FUNCTION (INFIX "Y=A|B")))
     (CELL "andor" 1                      )))))))
```

The reply describes the existence of a generator “andor”, which can provide the single cell “andor”, which in turn has the logical inputs “A” and “B” and the logical output “Y”. It can perform two different operations, namely the logical AND of its inputs in “andmode”, or the logical OR in “ormode”. With this functional information, synthesis can now proceed to cover the data flow graph.

Further into the design flow, the synthesis tool needs the characteristics of a specific module instance to perform various trade-offs (area, time, . . .). To this end, a “synthesis” view is requested by an appropriately constrained query.

```
(QUERY 1 2                                (CELL "andor" 0
 (TECHNOLOGY "Xilinx" "XC4000E"            (INTERFACE
             "XC4003EPG191" "-3"             (LOGICAL
  (VIEWS                                      (INPUT  (("A") (WIDTH 8) )
   (VIEW "synthesis"                                  (("B") (WIDTH 8) ))
    (GENERATOR "andor" 0                      (OUTPUT (("Y") (WIDTH 8) ))))))))))
```

Here, we request information on an 8-bit wide instance with default data types (unsigned integer) and variable inputs. The resulting reply contains the control specification for the cell as well as the area and time point for an actual physical implementation.

```
(REPLY 1 2                                           (BEHAVIOR
 (TECHNOLOGY "Xilinx" "XC4000E"                       ("andmode"
             "XC4003EPG191" "-3"                        (FUNCTION (INFIX "Y=A&B"))
  (VIEWS                                                (UCODE (LEVEL (("mode" 0 0) 0))))
   (VIEW "synthesis"                                  ("ormode"
    (STATUS QUERYOK "view ok")                          (FUNCTION (INFIX "Y=A|B"))
    (GENERATOR "andor" 1                                (UCODE (LEVEL (("mode" 0 0) 1)))))
     (STATUS QUERYOK "generator ok")                 (IMPLEMENTATION "simple" 1
     (UNIT (TIMESCALE -10))                           (CATALOG
     (CELL "andor" 1                                    ("structure" )
      (STATUS QUERYOK "cell ok")                        ("topology" )
      (INTERFACE                                        ("netlist"    (FORMAT "verilog"))
       (LOGICAL                                         ("placed"     (FORMAT "xnf"))
        (INPUT  (("A") (WIDTH 8) (UNSIGNED))            ("simulation" (FORMAT "verilog")))
                (("B") (WIDTH 8) (UNSIGNED)))         (TIMING
        (OUTPUT (("Y") (WIDTH 8) (UNSIGNED))))          ( ("andmode" "ormode" )
       (PHYSICAL                                         (FIXED
        (INPUT                                            (REQUIRED
         (("A")    (WIDTH 8) (DATA) (UNSIGNED))            (("A" 7 0) ("B" 7 0) ("mode" 0 0) )
         (("B")    (WIDTH 8) (DATA) (UNSIGNED))              0 0 0)
         (("mode") (WIDTH 1)                              (ARRIVAL (("Y" 7 0) ) 0 22)
                   (CONTROL) (UNSIGNED)))                 (CYCLETIME 22)
        (OUTPUT                                           (THROUGHPUT 1))))
         (("Y") (WIDTH 8) (DATA) (UNSIGNED)))))       (AREA ("CLBS" 4 4 100))))))))
```

After declaring a time scale of 0.1ns per time unit, the generator confirms the constraints on the logical interface before revealing the physical interface. An additional input “mode” becomes visible, and the type and data/control nature of each port is declared. The behavior of the function is further refined by including control information: “andmode” is activated by applying a logical 0 on the “mode” input, “ormode” by a logical 1.

The cell is available in one concrete physical implementation named “simple”, which can be retrieved in a number of views listed together with the formats for the non-FLAME views. The implementation timing for both operating modes is expressed in the slack-based model, stating that as long as all inputs arrive at time 0, the output will be valid no later than 22 time units (2.2ns) afterwards. Since the circuit is purely combinational, it could be clocked with a period of 2.2ns, and accept one new datum per clock cycle. It requires 4 of the chip resource “CLBS”, of which 100 are available on the target architecture. See [12–14] for further examples.

References

1. Xilinx Inc., “X-BLOX Reference”, *EDA tool documentation*, San Jose (CA) 1995
2. Dittmer, J., Sadewasser, H., “Parametrisierbare Modulgeneratoren für die FPGA-Familie Xilinx XC4000”, *Diploma thesis*, Tech. Univ. Braunschweig (Germany), 1995
3. Chu, M., Weaver, N., Sulimma, K., DeHon, A., Wawrzynek, J., “Object Oriented Circuit Generators in Java”, *Proc. IEEE Symp. on FCCM*, Napa Valley (CA) 1998
4. Mencer, O., Morf, M., Flynn, M.J., “PAM-Blox: High Performance FPGA Design for Adaptive Computing”, *Proc. IEEE Symp. on FCCM*, Napa Valley (CA) 1998
5. Gokhale, M.B., Stone, J.M., “NAPA-C: Compiling for a Hybrid RISC/FPGA Architecture”, *Proc. IEEE Symp. on FCCM*, Napa Valley (CA) 1998
6. Harr, R., “The Nimble Compiler Environment for Agile Hardware”, *Proc. ACS PI Meeting, http://www.dyncorp-is.com/darpa/meeting/acs98apr/Synopsys%20for%20WWW.ppt*, Napa Valley (CA) 1998
7. Hall, M., “Design Environment for ACS (DEFACTO)”, *Proc. ACS PI Meeting, http://www.dyncorp-is.com/darpa/meeting/acs98apr/defacto.ppt*, Napa Valley (CA), 1998
8. Electronics Industry Association, “EDIF Version 4 0 0”, *ANSI/EIA 682-1996 Standard*, Washington (DC) 1996
9. Synopsys Inc., “Library Compiler User Guide Version 3.5”, *EDA tool documentation*, Mountain View (CA) 1997
10. Open Verilog International, “Advanced Library Format for ASIC Cells & Blocks”, *ALF Reference Manual Version 1.0*, Los Gatos (CA) 1997
11. Koch., A., “Module Compaction in FPGA-based Regular Datapaths”, *Proc. 33rd Design Automation Conference (DAC)*, Las Vegas (NV) 1996
12. Koch, A., “FLAME/Java Release 0.1.1”, *http://www.icsi.berkeley.edu/˜akoch/research.html*, Berkeley (CA), 1998
13. Koch, A., “FLAME: A Flexible API for Module-based Environments – User’s Guide and Manual”, *http://www.icsi.berkeley.edu/˜akoch/research.html*, Berkeley (CA), 1998
14. Koch, A., “Generator-based Design Flows for Reconfigurable Computing: A Tutorial on Tool Integration using FLAME”, *Proc. PACT’98 Workshop on Configurable Computing*, Paris (France), 1998
15. Koch, A., “Regular Datapaths on Field-Programmable Gate Arrays”, *Ph.D. thesis*, Tech. Univ. Braunschweig (Germany), 1997
16. Liao, S., Devadas, S., Keutzer, K., Tjiang, S., “Instruction Selection Using Binate Covering for Code Size Optimization”, *Proc. ICCAD ’95*, November 1995
17. EIA, “Library of Parametrized Modules”, *EIA/IS-103 Standard*, 1993
18. Callahan, T., Chong, P., DeHon, A., Wawrzynek, J., “Fast Module Mapping and Placement for FPGAs”, *Proc. ACM/SIGDA Symp. on FPGAs*, Monterey (CA), 1998
19. Nijssen, R.X.T., Jesse, J.A.G., “Datapath Regularity Extraction”, in *Logic and Architecture Synthesis*, eds. Saucier/Mignotte, 1995
20. Xilinx Inc., “CORE Generator System User Guide”, *EDA tool documentation*, San Jose (CA) 1998
21. Hutchings, B., et. al., “A CAD Suite for High-Performance FPGA Design”, *Proc. FCCM ’99*, April 1999
22. Fowler, M., Scott, K., “UML Distilled”, *Addison-Wesley*, 1997