Generator-based Design Flows for Reconfigurable Computing: A Tutorial on Tool Integration using FLAME

Andreas Koch
UC Berkeley, ICSI
USA
akoch@icsi.berkeley.edu

Ulrich Golze
TU Braunschweig, Abt. E.I.S.
Germany
golze@eis.cs.tu-bs.de

High-performance design flows for FPGAs often rely on module generators to implement fast sub-circuits. However, the very flexibility of current generator systems makes their automatic use by synthesis and floorplanning steps difficult. We present a common model to express generator capabilities and design characteristics to client tools. Examples show how an active query/reply scheme supports a stepwise refinement by incrementally tightening constraints to narrow down the range of possible solutions.

1 Introduction

While well known for decades, the use of module generators in VLSI design flows has recently been exploited with renewed interest. In the ASIC field, commercial products such as ModuleCompiler [1] are achieving good results in terms of design time and quality. 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 [2] [3] [4] [5]. Thus, compilation systems for reconfigurable computers often include module generators as a major processing step.

2 Hindrances

Unfortunately, the very flexibility of parametrized generators makes their integration with the main design flow (synthesis, floorplanning, place and route) difficult. For example, modern generators can completely restructure a general circuit description to optimally match constant inputs. When one of the main flow tools requires information about such a module, 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 alternatives. Thus the use of passive (static) data for library information becomes impractical. However, existing active interfaces such as DPCS [6] or ITC/MDS [7] often require non-portable services such as dynamic loading, specialized compilers or proprietary libraries.

In addition, the common formats for characterizing library elements, such as EDIF [8], ".lib" [9], and ALF [10], are tailored for ASICs as target technology. While they allow the detailed specification of electrical, timing, and geometrical parameters, this abstraction level is mismatched for describing designs targeting FPGAs: Vendors generally do not make low-level physical information available, and the higher-level concepts required for efficient module embedding (e.g., function, control interface, and sequential timing etc.) are not covered by existing formats.

Here we present a brief tutorial on how FLAME [13], the Flexible API for Module-based Environments, can be used to resolve these difficulties by smoothly integrating module generators with the main design flow. It is currently used in academic as well as in industrial research on next generation compilation systems for reconfigurable computers, where it will allow the portable use of generator-based IP in a variety of tool chains.

3 FLAME

FLAME consists of a common model to express generator (server) capabilities and module design characteristics to client tools (e.g, synthesis and floorplanning), an active interface allowing a dialog between client and server to incrementally make design decisions, and portable representations for FLAME expressions.

FLAME can be extended by introducing new attributes while still preserving backwards compatibility. At most, old (non-extended) specifications lead to solutions with deteriorated quality, but the system always remains functional.

To facilitate its adoption, a functional FLAME interface has only simple minimal requirements. In general, several dozen lines of code should suffice to integrate a new generator into the main design flow. Afterwards, the interface may be gradually refined to improve solution quality.

FLAME does not rely on features of a specific implementation language or execution platform. The programming interface uses only a small number of procedural entry points and a single frame-based data structure.

The efficiency of generating and manipulating FLAME data allows the use of the interface in monolithically integrated tools as well as in loosely coupled systems distributed over a network. By choosing between three representations (binary, tokenized, and text) for FLAME data, the EDA developer can match efficiency and portability to the specific scenario. In the following examples, we will use the human-readable text representation, which is based on the well-known frame concept.

4 Active interface

Fig. 1 shows a sample dialog between clients floorplanning) in the main design flow and servers in the library. The queries by the clients become increasingly more specific, and only the requested specifications are being returned. This approach reduces computation and data transfer times, since only a minimum of information can be supplied. A memoization mechanism (not shown) serves to further increase efficiency by answering previously encountered requests from a cache without forwarding them again to the generators.

The communication interface has been designed to allow asynchronous multi-threaded execution (running multiple generators in parallel). It its independent of the communication protocols, which can range from simple function invocations in monolithically integrated systems, spawning external programs (such as Perl scripts) communicating via pipes, to socket-based exchanges in a system distributed over the Internet. Note that existing HTTP server technology can be leveraged off easily by encapsulating FLAME expressions in HTML.

This could lead to a scenario where IP providers offer access to their wares from their web sites. Characteristics about each product could be retrieved free of charge, only when netlist or layout data would be requested would the customer account be charged.

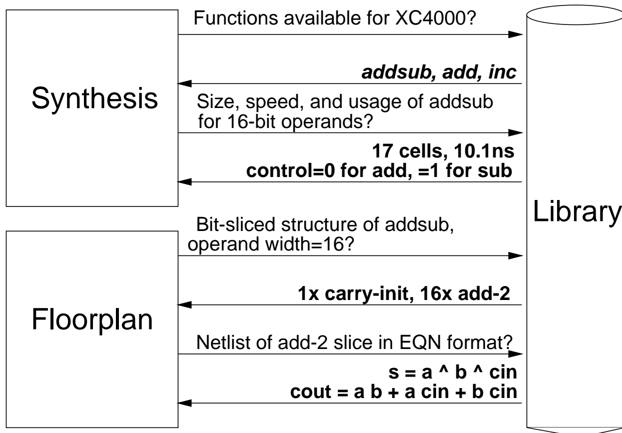

Fig. 1: Sample query/reply scenario

5 System architecture

A simple procedural interface consisting of five functions allows the sending and receiving of queries and replies. The *FLAME Manager* (Fig. 2) distributes the queries it receives from its clients to the servers, collects individual replies, and packages them into a composite reply which is returned to the client.

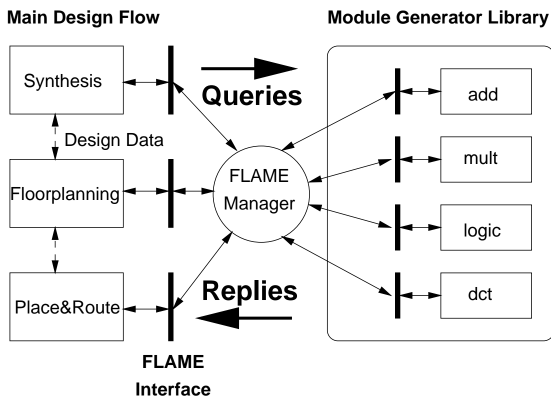

Fig. 2: FLAME system architecture

As with other EDA data formats, FLAME employs LISP-style associative lists of symbols (extensible keywords), integers, strings, and lists. In addition, it allows the seamless embedding of foreign data (e.g., EDIF netlists or Verilog simulation models).

The human readable text representation is easiest to process even for simple tools (such as Perl scripts), while a pre-tokenized representation can be processed and transferred more efficiently even between tools implemented in different languages and environments. For even higher performance, the implementation-language specific binary representation (generally built using pointers or references) can be passed between integrated tools using the same language binding.

FLAME requires a language binding specific for each implementation language [11]. The binding provides an efficient internal representation for FLAME data and functions to manipulate it. Furthermore, it can translate to and from the standard text and pre-tokenized representations and communicate with the FLAME Manager.

On top of this infrastructure lies a well defined set of attributes to concisely represent module characteristics. These include, but are not limited to: behavior, control interface, timing, area, power, bit-sliced structure, topology, and embedded foreign data (netlists, layout, simulation models). To reduce the number of separate queries, sets of related attributes are bundled into *views*.

6 Design hierarchy

Fig. 3 shows the FLAME design entities and their relations. 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 (e.g., clock speed, but not pipelining!) and power consumption. In a software system, cells would be considered module interfaces, and implementations module bodies. The cell is composed from *sub-modules* (cell instances) and *stacks*. Stacks contain one or more *zones* of replicated logic. The smallest design unit is the *slice* (sometimes called *master-slice* to emphasize the replication aspect).

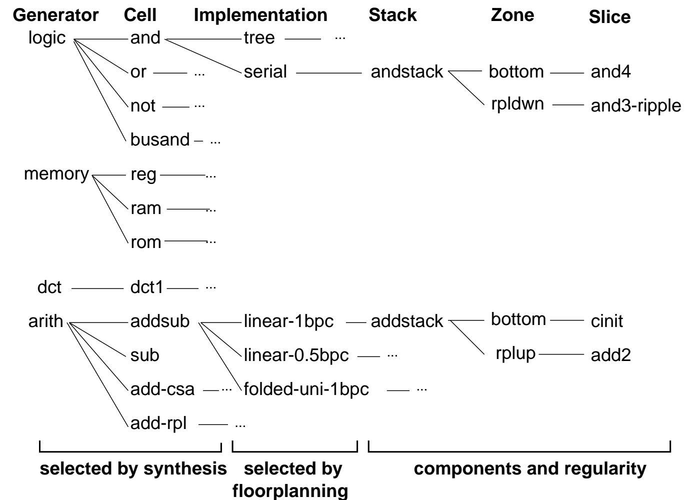

Fig. 3: FLAME design entities

When the synthesis system is covering its data flow graph with modules in the library, it selects suitable cells. Since all implementations in 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 [12], while keeping the synthesized global controller intact.

The next examples show a typical dialog between clients and servers, beginning with the start of the synthesis run and ending with floorplanning. Note that the examples have been chosen to demonstrate basic FLAME concepts, the complete functionality is described in [13].

7 Library functions

As the first compilation step, the synthesis tool will request an index of all cells available in the target technology (possibly including technology independent cells). This step would be considered “loading the library” in a static-file based approach. To this end, the client will issue the following query, which is then distributed to all servers:

```
(QUERY 1 1
 (VIEW "behavior"
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")))
```

In this example, we assume the existence of two generators offering applicable cells. Cell behavior is currently specified using either

arithmetic or logic expressions in infix notation, or as a procedure prototype. The reply to the "behaviore" query lists the generators and their cells together with behavioral descriptions and logical interfaces.

```
(REPLY 1 1
 (VIEW "behavior"
  (STATUS QUERYOK)
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")

  (GENERATOR "arith" 2

   (CELL "muldiv" 3
    (INTERFACE (LOGICAL (INPUT  (("A" "B")))
                        (OUTPUT (("P" "Q")))))
    (BEHAVIOR  ("muldiv"
                 (FUNCTION (INFIX  "P=A*B")
                           (INFIX  "Q=A/B"))))

   (CELL "addsub" 1
    (INTERFACE (LOGICAL (INPUT  (("A" "B")))
                        (OUTPUT (("SD")))))
    (BEHAVIOR  ("addmode"
                 (FUNCTION (INFIX "SD=A+B")))
               ("submode"
                 (FUNCTION (INFIX "SD=A-B")))))

  (GENERATOR "logic" 1

   (CELL "and" 1
    (INTERFACE (LOGICAL (INPUT  (("A" "B")))
                        (OUTPUT (("Y")))))
    (BEHAVIOR  ("and"
                 (FUNCTION (INFIX "Y=A&B")))))

   (CELL "parity" 1
    (INTERFACE (LOGICAL  (INPUT    (("A")))
                         (OUTPUT   (("Y")))))
    (BEHAVIOR  ("parity"
                 (FUNCTION (PROC "parity(A,Y)")))))
)))))
```

Observe that the *addsub* cell can compute either the sum or the difference of its inputs, while *muldiv* simultaneously computes the product and quotient of its inputs on separate outputs. With this functional information, synthesis can now proceed to cover the data flow graph.

8 Cell control, area, and timing

When the covering requires trade-off information for a particular cell, a "synthesis" view is requested for it. The query includes the required constraints for parameters such as target technology, operand widths, data types and constant values (where applicable).

```
(QUERY 1 2 (VIEW "synthesis"
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")
  (GENERATOR "arith" 0
   (CELL "addsub" 0
    (INTERFACE (LOGICAL
     (INPUT  (("A" "B") (WIDTH 16) (SIGNED)))
     (OUTPUT (("SD")    (WIDTH 16) (SIGNED))))))
)))
```

This query will retrieve detailed information on a 16-bit version of the adder/subtracter. The resulting reply contains the control specification for the cell as well as the area and time points for different physical implementations.

```
(REPLY 1 2 (VIEW "synthesis"
 (STATUS QUERYOK)
 (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")

  (GENERATOR "arith" 2
   (STATUS QUERYOK)
   (UNIT (TIMESCALE -10))

   (CELL "addsub" 1
    (STATUS OK)

    (INTERFACE
     (LOGICAL
      (INPUT  (("A" "B") (WIDTH 16) (SIGNED) ))
      (OUTPUT (("SD")    (WIDTH 16) (SIGNED) )))
     (PHYSICAL
      (INPUT  (("A" "B")
               (WIDTH 16) (SIGNED)   (DATA)))
      (OUTPUT (("SD")
               (WIDTH 16) (SIGNED)   (DATA))))
      (INPUT  (("nAdSb"))
               (WIDTH 1)  (UNSIGNED) (CONTROL)))

    (BEHAVIOR
     ("addmode"  (FUNCTION (INFIX "SD=A+B"))
                  (UCODE (LEVEL (("nAdSb") 0))))
     ("submode"  (FUNCTION (INFIX "SD=A-B"))
                  (UCODE (LEVEL (("nAdSb") 1)))))

    (IMPLEMENTATION "softcarry-1bpc" 1
     (TIMING ( ("addmode" "submode")
               (FIXED
                (REQUIRED () 0 0 0)
                (ARRIVAL  () 0 405))))
     (AREA   ("4LUTS" 32 32 800)))

    (IMPLEMENTATION "softcarry-0.5bpc" 1
     (TIMING ( ("addmode" "submode")
               (FIXED
                (REQUIRED () 0 0 0)
                (ARRIVAL  () 0 395))))
     (AREA   ("4LUTS" 32 32 800))))))
```

The "synthesis" view reveals more information about the *addsub* cell. Namely, it has an additional control input *nAdSb*, which is used to switch between addition and subtraction. The manner in which *nAdSb* is actually used is described by the *UCODE* attribute: Since the cell is purely combinational, the operation is controlled by simply changing the level to "0" for addition, and "1" for subtraction. *addsub* is available in two implementations which vary in timing and area. *softcarry-0.5bpc* is laid out using a looser logical pitch. Both addition and subtraction have the same data-independent timing, which is specified in terms of required times (on inputs) and arrival times (on outputs). Since the *TIMESCALE* was set to 0.1ns, the value 405 actually stands for a worst-case combinational delay of 40.5ns from all inputs to all outputs[1]. Area is measured in one or more technology-specific units defined in an external technology file. In the example, both implementations use 32 of the 800 available "4LUTS" of the target chip.

Above and beyond this simplified example, FLAME allows the specification of sequential delays (latencies) for pipelined modules. In addition, a more precise delay model using path-based timing is available. The FLAME control specification also extends to cover complex multi-cycle control flows (such as loading operands and opcodes into a programmable unit), variable execution times, and interleaved execution cycles (e.g., simultaneously loading new operands while unloading a result).

[1]For brevity, required and arrival times were not specified on a per-port basis, which would more accurately reflect the precise timing of a ripple-carry adder.

For the queried cell, the synthesis tool now has sufficient information (area-time spectrum for the various implementations) to compare different flow graph coverings. Furthermore, the description of the cell control interface allows the synthesis of suitable control logic driving it.

9 Implementation structure

To allow the floorplanning tools to exploit the regular bit-sliced nature common to many basic datapath modules [14], FLAME defines a structural view into a cell implementation. In queries, the *INTERFACE* has to be constrained as for the "synthesis" view. For brevity, we are showing the structure of an 8-bit AND gate here instead of the adder/subtracter.

```
(QUERY 1 3 (VIEW "structure"
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")
  (GENERATOR "logic" 0
   (CELL "and" 0
    (INTERFACE (LOGICAL
     (INPUT  (("A" "B") (WIDTH 8) (SIGNED)))
     (OUTPUT (("Y")     (WIDTH 8) (SIGNED)))))
    (IMPLEMENTATION "basic" 0)))))
```

Note that we are now operating at the implementation scope.

```
(REPLY 1 3 (VIEW "structure"
  (STATUS QUERYOK)
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")

  (GENERATOR "logic" 1
   (STATUS QUERYOK)

   (CELL "and" 1
    (STATUS QUERYOK)

    (INTERFACE
     (LOGICAL
      (INPUT  (("A" "B") (WIDTH 8) (SIGNED) ))
      (OUTPUT (("Y")     (WIDTH 8) (SIGNED) )))
     (PHYSICAL
      (INPUT  (("A" "B")
               (WIDTH 8) (SIGNED) (DATA) ))
      (OUTPUT (("Y")
               (WIDTH 8) (SIGNED) (DATA) ))))

    (IMPLEMENTATION "basic" 1
     (STATUS OK)
     (STRUCTURE
      (SLICEDLINEAR

       (SLICES
        ( SLICE "and2" 1
          (LEAF (INTERFACE (PHYSICAL
           (INPUT  (("a" "b")
                    (WIDTH 1) (UNSIGNED) (DATA)))
           (OUTPUT (("y")
                    (WIDTH 1) (UNSIGNED) (DATA)))
       )))))

       (STACKS
        ( STACK "andstack" 1
          (INTERFACE (PHYSICAL
           (INPUT  (("Op1" "Op2")
                    (WIDTH 8) (SIGNED) (DATA))
           (OUTPUT (("AndOut")
                    (WIDTH 8) (SIGNED) (DATA))))))
          (VERTICAL
           (ZONE "and2" 8
            (VCONNECT
             ( (("Op1" 0 1) ("a"))
               (("Op2" 0 1) ("b")) )
             ( (("y")       ("AndOut" 0 1)) )
             ()
       )))))

       (HORIZONTAL
        (LINEAR "andstack")
        (HCONNECT
         (0 1 ("A" 7 0)       ("Op1" 7 0))
         (0 1 ("B" 7 0)       ("Op2" 7 0))
         (1 0 ("AndOut" 7 0) ("Y" 7 0)))))
))))))
```

This reply reveals that the selected implementation consists of a single *slice* named "and2" that implements a 2-bit AND gate. (Fig. 4.a). The entire implementation is assembled by vertically iterating (tiling) this slice 8 times (iteration numbers 0 . . . 7, Fig. 4.b) to form the *stack* "andstack". This stack is instantiated once, and connected to the module primary ports as appropriate to assemble the entire module (Fig. 4.c).

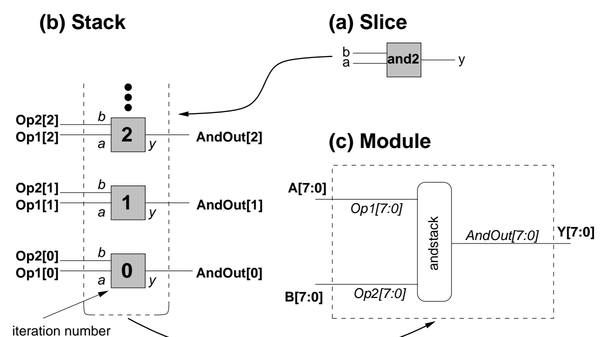

Fig. 4: Regularity and structure

While this example might seem like overkill for such a trivial circuit, the expressiveness allows advanced optimizations for more complex structures (possibly including sub-modules). E.g., it is possible to re-assemble a hierarchical module in context of the generated floorplan by substituting different sub-modules into its hierarchy. This could be used to select slower but smaller modules once it has been discovered during floorplanning that the top-level module is not on the critical path of the circuit.

10 Implementation topology

The "topology" view provides the information required for floorplanning. The query is constrained in the same manner as for the "structure" view.

```
(QUERY 1 4 (VIEW "topology"
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")
  (GENERATOR "arith" 0
   (CELL "addsub" 0
    (INTERFACE (LOGICAL
     (INPUT  (("A" "B") (WIDTH 4) (SIGNED)))
     (OUTPUT (("SD")    (WIDTH 4) (SIGNED)))))
    (IMPLEMENTATION "softcarry-1bpc" 0)))))
```

The reply describes the floorplan footprint and logical pitch (measured in *bits-per-cell height* (bpc)) for the first implementation, this time in a 4-bit version (Fig. 5).

```
(REPLY 1 4 (VIEW "topology"
 (STATUS QUERYOK)
 (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")

  (GENERATOR "arith" 2
```

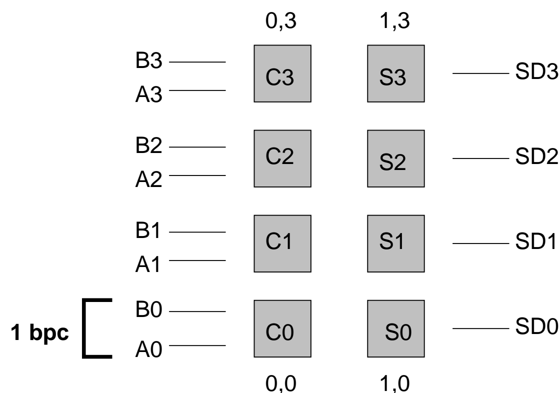

Fig. 5: Adder/subtracter topology

```
    (STATUS QUERYOK)
    (UNIT (TIMESCALE -10))

    (CELL "addsub" 1
     (STATUS QUERYOK)

     (INTERFACE
      (LOGICAL
       (INPUT  (("A" "B")
                (WIDTH 4) (SIGNED) ))
       (OUTPUT (("SD")
                (WIDTH 4) (SIGNED) )))
      (PHYSICAL
       (INPUT  (("A" "B")
                (WIDTH 4) (SIGNED)   (DATA)))
       (OUTPUT (("SD")
                (WIDTH 4) (SIGNED)   (DATA))))
       (INPUT  (("nAdSb"))
                (WIDTH 1)  (UNSIGNED) (CONTROL)))

     (IMPLEMENTATION "softcarry-1bpc" 1
      (STATUS QUERYOK)
      (TOPOLOGY (MATRIX
        (SHAPE (RECT 2 4))
        (PORT ((("A" 15 0) ("B" 15 0))
          (PITCH 1 1) (FOLDING LINEAR) (COORD 0 0)))
        (PORT ((("Y" 15 0))
          (PITCH 1 1) (FOLDING LINEAR) (COORD 1 0))))))
))))
```

All data ports have a logical pitch of 1 bpc. However, the inputs are assumed to be placed at the left edge of the circuit (column 0), while the output is located at the right edge (column 1). Matching logical pitch is important especially in FPGAs, where mismatched pitch would cause long delays due to the increased wiring.

11 Simulation

Each implementation must be available in a “simulation” view. In general, the data returned will be a behavioral HDL model encapsulated in a FLAME expression. For example, the query

```
(QUERY 1 5 (VIEW "simulation"
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")
  (GENERATOR "logic" 0
    (CELL "and" 0
      (INTERFACE (LOGICAL
        (INPUT  ( () (WIDTH 4) ))
        (OUTPUT ( () (WIDTH 4) ))))
      (IMPLEMENTATION "basic" 0
        (INSTANCE "andgate4")
        (FORMAT "verilog"))))))
```

will initiate the generation of a Verilog HDL module named *andgate4* for the 4-bit AND. The Verilog is embedded into FLAME as demonstrated by the reply:

```
(REPLY 1 5 (VIEW "simulation"
  (STATUS QUERYOK)
  (TECHNOLOGY "Xilinx" "XC4000" "XC4010" "-3")
  (GENERATOR "logic" 1
    (STATUS QUERYOK)
    (CELL "and" 1
      (STATUS OK)
      (INTERFACE
        (LOGICAL
          (INPUT  ( ("A" "B")
                    (DATA) (UNSIGNED) (WIDTH 4)))
          (OUTPUT ( ("Y")
                    (DATA) (UNSIGNED) (WIDTH 4))))
        (PHYSICAL
          (INPUT  ( ("A" "B")
                    (DATA) (UNSIGNED) (WIDTH 4)))
          (OUTPUT ( ("Y")
                    (DATA) (UNSIGNED) (WIDTH 4)))))
      (IMPLEMENTATION "basic" 1
        (STATUS QUERYOK)
        (INSTANCE "andgate4")
        (FORMAT "verilog")
        (SIMULATION !@#$ "verilog"
          `timescale 100 ps / 10 ps
          module andgate4(Y,A,B);
           output [3:0] Y;
           input  [3:0] A, B;
           assign Y[3:0] = #25 (A[3:0] & B[3:0]);
          endmodule
          !@#$))))))
```

From the union of all generated HDL models, the main flow tools can assemble a top-level model for the entire circuit.

12 Netlists

After floorplanning, the last steps consist of the actual generation of annotated netlists in various formats or refinements (e.g., netlist, placed, routed). For this task, standard formats such as EDIF or a structural HDL are used and embedded into the FLAME representation. They are retrieved via the same query and reply mechanism as the other views.

13 Summary and conclusion

In this paper, we presented a brief tutorial on using FLAME, a portable mechanism for inter-tool communication with an emphasis on module generators. We introduced the FLAME approach of representing and exchanging design data, and demonstrated some of its capabilities using selected examples.

FLAME provides a way to use generator-based IP in a variety of synthesis and floorplanning tools, thus allowing the free exchange of design flow components. While it allows the description of complex structures in great detail, it can also be used to quickly integrate a newly developed component tool into the main system, e.g., to a evaluate a new module generator in context.

In a larger scope, FLAME supports the seamless use of third-party IP either locally or by communicating with generators at a remote vendor site using HTTP-encapsulated messages. The currently employed ad-hoc generator interfaces support these capabilities only to a very limited degree.

References

[1] Synopsys Inc., “Module Compiler User Guide”, *EDA software documentation*, Mountain View (CA) 1997

[2] Xilinx Inc., “X-BLOX Reference”, *EDA software documentation*, San Jose (CA) 1995

[3] Dittmer, J., Sadewasser, H., “Parametrisierbare Modulgeneratoren fü r die FPGA-Familie Xilinx XC4000”, *Diploma thesis*, Tech. Univ. Braunschweig (Germany), 1995

[4] 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

[5] 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

[6] Silicon Integration Initiative, "Delay and Power Calculation using DCL", *http://www.si2.org/dcl*, Austin (TX) 1998

[7] Silicon Integration Initiative, "Message Dictionary Specification", *http://www.si2.org/dcl*, Austin (TX) 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 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., "FLAME/Java User's Guide", *http://www.icsi.berkeley.edu/~ akoch/research.html#FLAMEJ*, Berkeley, 1998

[12] Koch, A., "Regular Datapaths on Field-Programmable Gate Arrays", *Ph.D. thesis*, Tech. Univ. Braunschweig (Germany), 1997

[13] Koch, A., "FLAME: A Flexible API for Module-based Environments – User's Guide and Manual", *http://www.icsi.berkeley.edu/~ akoch/research.html#FLAME*, Berkeley, 1998

[14] Koch., A., "Module Compaction in FPGA-based Regular Datapaths", *Proc. 33rd Design Automation Conference (DAC)*, Las Vegas (NV) 1996