Compilation for Adaptive Computing Systems using Complex Parameterized Hardware Objects

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

Abstract. FLAME, the Flexible API for Module-based Environments, is a proposed standard interface for the integration of parameterized hardware generators into high-level design tools. This work introduces two new developments: The FLAME Primitives Catalog describes the behavior and interface of a set of hardware functions suitable as primitives for automatic compilation. The FLAME Shared Access Conventions define physical connectivity and logical protocols that allow arbitrary hardware modules to access shared resources (such as memory or I/O ports) in a coordinated manner.

Keywords: parameterized hardware, shared access

1. Introduction

In the quest for ever increasing computing power, many approaches have been examined beyond the traditional micro-processor. Some keywords include reduced-instruction set computers (RISC), super-scalar and speculative execution, very-long instruction-word (VLIW) architectures exposing the underlying parallel function units to software, and concepts such as simultaneous multi-threading (SMT) and chip-level multi-processing (CMP).

In all of these cases, dynamically loadable software executes on fixed hardware architectures. By allowing an additional degree of freedom, namely that of dynamically *adaptable* hardware, another approach to computing becomes feasible: Here, the hardware architecture itself can be molded to the requirements of the specific algorithm (in some cases even the specific data-set being processed).

Adaptive computing systems relying on configurable computation units can thus be customized not only by loading appropriate software, but also by adapting certain hardware aspects of the system to better handle the current computational problem. However, even the best hardware architecture needs to be exploited (programmed) in a user-friendly yet efficient manner. Otherwise, it will be shunned in favor of other, less powerful, but more accessible approaches.

To this end, a design flow for an adaptive computer will have to start with an algorithm described in a language familiar to the potential user base (e.g., C, MATLAB, FORTRAN) and then proceed to simultaneously optimize both soft- and hardware components. We participated in the development of one such flow, compiling from C to a adaptive hardware, that has been described

in [6] [7]. In this paper, we focus on the device-specific module generators that create the actual hardware implementations and their external interfaces.

2. Module Generators

Parameterized module generators [1] [2] [3] are used in many design flows targeting configurable computing platforms to quickly obtain high-performance hardware objects. The mode of use ranges from fully manual [4] to an integration into automatic flows [5] [6] [7] [8].

However, automatic integration is hindered by two major problems: First, no standardized interface exists currently that allows the main flow tools (compiler/synthesis, floorplanning, place and route) to automatically access a diverse set of generator libraries. Today, each vendor uses its own (often file-based) control protocols.

Second, even when automatic access is available, it is generally only used to initiate the creation of a (possibly placed) netlist and (maybe) a simulation model of a parameter-specific module instance. There is no feedback path from the generators to the main flow tools that allow these to retrieve information *about* instance-specific characteristics (e.g., area, timing, control interfaces, layout topology) in order to actually make meaningful trade-off decisions at the architectural level.

A static enumeration of this data (similar to the classic “library files” describing cells in semi-custom design) is no longer feasible: Modern generators are able to, e.g., completely restructure a circuit exploiting constant inputs [3]. This leads to a parameter value-dependence of many cell characteristics that cannot be expressed statically. Instead, an active interface for their dynamic calculation is required.

3. FLAME

The Flexible API for Module-based Environments (FLAME) solves these problems with a two-pronged approach. First, it provides a standardized design data model expressing generator capabilities and module characteristics to client tools. Second, it replaces the common file-based data exchange by an active interface (API), allowing an interactive dialog between client tools and module generators. In this manner, a module is instantiated by successive refinement: The client tools incrementally tighten constraints, while the generators reply with increasingly accurate area/time/power/... estimates, culminating at the highest refinement level in the generation of layout.

Note that FLAME *wraps* existing module libraries, it has no generation capabilities of its own. Furthermore, since it aims at the integration of automatic design flows, it does not contain a GUI. Instead, it defines multiple

data representations covering a spectrum of efficiency vs. portability for the exchange of information between EDA tools.

The next few subsections will give a brief general overview of FLAME. For a more detailed description, see [12]. Recent FLAME developments such as the Primitives Catalog on and the Shared Access Conventions are presented in the final two sections.

3.1. Active Interface

A sample for a dialog between client tools and generators is shown in Fig. 3.1. Computation times can be reduced since results need only be computed to the abstraction level of the current query. E.g., when requesting area and delay estimates for synthesis, it is not necessary to place and route the circuit down to the layout level.

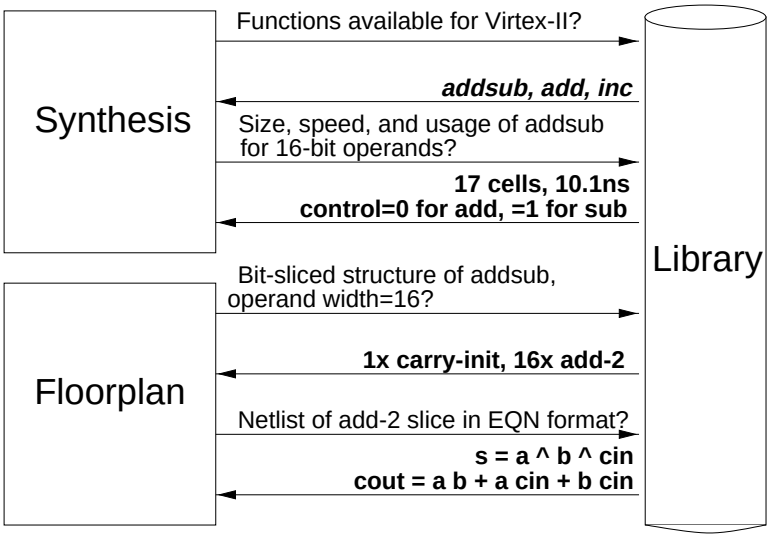

Figure 1. Sample dialog

This dialog can be carried out in a number of data representations ranging from a highly efficient binary format to a human-readable textual representation. Furthermore, it is independent of the transport mechanism and allows for monolithic (all components linked into a single executable) as well as distributed (e.g., IP accessed over the Internet) systems.

3.2. Views

The concept of a “view” is used in FLAME to group related data. For example, a client only has to query for a “synthesis” view to receive a collection

of characteristics such as timing, area, control interface, and power estimates. It is the view mechanism that is used to restrict the scope of generator computation to the information that is needed at a single step in the design flow. This avoids computing *all* data, and only have most of it discarded when the module is not selected early on in the compilation process.

3.3. Design Hierarchy and Regularity

The amount of data exchanged between clients and servers is also controlled by strictly following a hierarchy of design entities (Figure 3.3), where lower levels (more detail) are only accessed when required. The explicit representation of regularity (e.g., the iteration of bit-slices) also serves to limit the amount of data exchanged and can reduce the computation times when exploited by the tools.

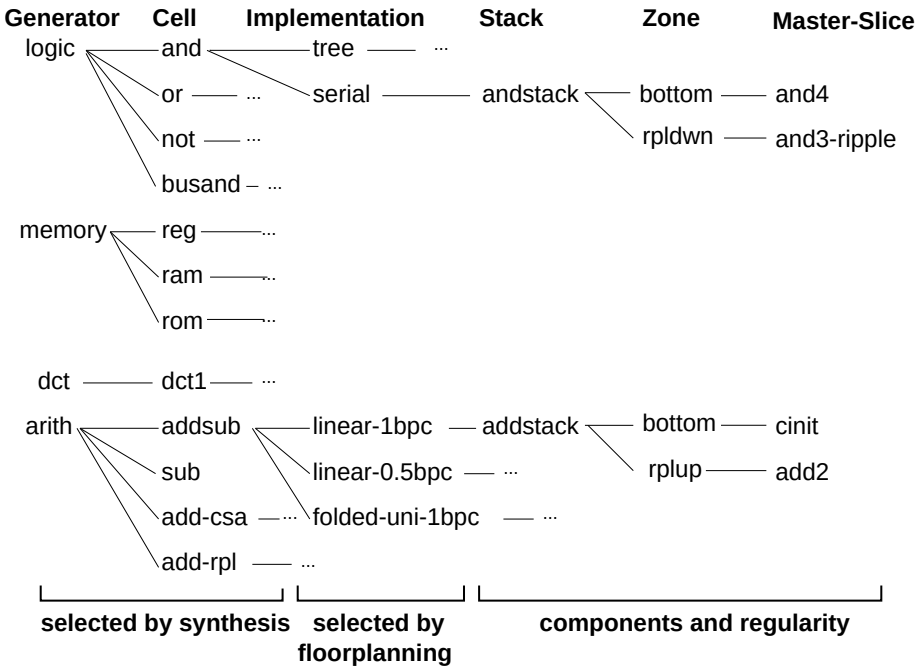

Figure 2. FLAME design entities

To illustrate the hierarchy, consider the following example: A generator *arith* might provide the cells *addsub* (switchable adder-subtracter), *sub* (subtracter), *add-csa* (adder), and *add-rpl* (adder). The adder-subtracter 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).

3.4. Target Technology

The capabilities of storage elements and tri-state buffers as well as available routing and logic resources are abstracted by FLAME in a portable manner. Design tools are thus presented with a uniform view of the different underlying FPGA architectures, allowing both the easy re-targeting of designs between architectures as well as the development of portable CAD tools supporting multiple technologies.

Despite the abstraction, all commonly used features are modeled: This includes the polarities and presences of control inputs such as clock-enable, the storage element type (edge vs. level triggered), and reset behavior (sync vs. async, set vs. reset).

3.5. Parameters

The generator clients in the main design flow create a module instance by imposing constraints on a wide spectrum of parameters. Standard parameters for cells include the bit widths and data types of operands as well as the presence of constant operands (which can be folded directly into the generated circuit). For efficient synthesis, outputs can also be optionally registered or made tri-stateable. In addition to this standard set, an arbitrary number of user-defined parameters can be passed. E.g., a FIR filter might also accept a list of coefficients as parameter.

3.6. Cell Characteristics

Given a set of parameter constraints, the generator can then proceed to calculate a large number of characteristics specific for that set of constraints ("dynamic data book approach").

3.6.1. *Function*

The function(s) of a cell in FLAME are described using either an expression in infix notation (such as $Y = A\&B$ for a bitwise AND), or using a procedure prototype (e.g., FIR(Y,A,COEFFS) for a FIR filter). It is assumed that primitive modules (AND, ADD, MUX, ...) will be instantiated automatically by the compiler, while complex modules (e.g., FIR/IIR, FFT, DCT, SKIPJACK, ...) must be explicitly instantiated by the user as a function call. Section 4 describes module functions in greater detail.

3.6.2. *Interface*
In addition to the cell function, FLAME describes its logical and physical interfaces. E.g., while the logical interface of a serial adder might just list the operand inputs and the sum output, the physical interface could also reveal the clock and Start (=clear stored carry) inputs. Special module requirements (such as access to external memory or peripheral devices) are also specified here. These capabilities are discussed in Section 5.

Specifying the control interface completes the information required to automatically use a cell in a synthesized circuit. Control specifications might range from a simple addition/subtraction switch by changing the value of a control input from 0 to 1, to complex multi-cycle sequences of simultaneously loading and unloading data into and from a computation unit that signals its completion after a variable number of cycles. FLAME relies on six control instructions to provide the information required by synthesis to create the appropriate FSM.

3.6.3. *Timing*
Timing characteristics can be described in FLAME using both path- and slack-based models. They cover not only combinational delays, but also latency values for pipelined execution. For units with variable (data-dependent) execution times, best-case, average-case, and worst-case timing can be indicated to guide the module selection by the compiler.

3.6.4. *Area*
The resource requirements of a module instance are modeled as a vector reflecting the heterogeneous nature of units on an FPGA (e.g., logic blocks, memory, DLLs, . . .). Since the performance of FPGA-based circuits is highly dependent on a good routing solution, the routing requirements and characteristics of the generated circuit can also be described at multiple levels of detail. Tools can use this data, e.g., for managing congestion by placing densely routed modules at the edges of the datapath.

3.6.5. *Layout Topology*
For regular logic optimization and floorplanning [10], the FLAME design data model supplies constructs to describe a regular composition (e.g., bit-sliced) as well as topological information such as the port location and pitch, and shape of the final layout.

4. Primitive Functions

While the FLAME specification itself covers the inter-tool communication protocols and the model (meta data) for circuit parameters and characteristics, it does not define the actual behavior and interface of concrete modules.

Table I. Basic library functions

Description
simple arithmetic (addition, subtraction, negation)
comparison (magnitude and equality)
boolean logic
multiplexing
negation
RAM
registers
ROM
shifting (arithmetic and logical)

Since these are often application-domain specific, it is expected that they are detailed in separate documents.

The first of these documents, the FLAME Primitives Catalog (FPC), concentrates on describing a set of basic circuits usable as translation targets ("hardware op-codes") for a general-purpose high-level hardware compiler. It contains behavioral and interface descriptions for 19 functions from the areas shown in Table 4.

Each of these functions can be implemented in one or more cells. Conversely, each cell may implement one or more of these functions. Note an important difference between this effort and previous ones such as LPM [11]: The FPC describes *behaviors*, not actual hardware realizations. E.g., while LPM contains an `LPM_ADD_SUB` circuit that defaults to `ADD` when the `Add_Sub` control input is unused, the FPC simply defines `add` and `sub` behaviors that may be provided in any combination by various cells. For example, in FLAME, the function of `LPM_ADD_SUB` would be expressible by attaching both `add` and `sub` to the same cell (as well as a control input for selecting the actual operation at run-time). However, in FPC these behaviors could also be used in conjunction with the behavior `logic` to describe an ALU module, a circuit which is not contained in LPM at all.

The FPC defines a variety of rules and guidelines for the interface to these functions. Specifications cover the minimum operand widths, the minimum set of data types to be supported, port naming conventions, default values for optional inputs, rules for matching the widths of operands, suggested area units for the Xilinx XC4000 and Virtex FPGA series, and the idioms for accessing technology-specific features on these targets.

Each of the individual functions is then described in a manner similar to the following example treating "addition".

`add(sum, [cout,] [ovfl,] a, b [, cin])`

"Add the two input operands `a` *and* `b` *producing a result* `sum`*. The addition may optionally take a carry input* `cin` *and produce a carry output* `cout` *and overflow output* `ovfl`*."*

Name	Description	Kind	Width	Type	Usage
sum	result	out	$1 \ldots 32+$	uint+	data
cout	carry result	out	$0, 1$	uint	control
ovfl	overflow	out	$0, 1$	uint	control
a	first operand	in	$1 \ldots 32+$	uint+	data
b	second operand	in	$1 \ldots 32+$	uint+	data
cin	carry operand	in	$0, 1(0)$	uint	control

In the 'Width' column, '$1 \ldots 32+$' indicates that a generator must support operand widths at least in the range of 1 to 32 bits (but possibly wider). '$0, 1$' signifies an optional port (having a width of 0 or 1 bits). In the case of an input (e.g., '$0, 1(0)$'), this can be extended with a default value when the port is unused. 'uint' declares a data type as unsigned integer, 'uint+' states that at least the unsigned integer data type must be supported.

All functions in the FPC are consistently described in this fashion, thus defining an unambiguous 'contract' between module generators and module users (front-end compiler). Further application domains (e.g., signal processing and cryptography) are expected to be addressed in additional function catalogs later.

5. Accessing Shared Resources

Despite the efforts working towards fully automatic translation of a high-level description into efficient hardware, there will always be cases that are handled better using a carefully tuned manual design. This reflects the current situation in the software arena, where (e.g. for fast 3D graphics) critical kernels are still being implemented in highly optimized assembly language. These manually instantiated modules often implement a complex algorithm (e.g., SKIPJACK cryptography, DCT filtering, . . .) and run much faster than if they were assembled from primitives by the compiler.

While even the base FLAME specification allows the automatic linking (including control FSM creation) of such manually instantiated modules with instances requested by the compiler, it does not cover cases in which a module instance needs access to shared resources such as local or shared memory, I/O ports, or on-chip peripherals. Especially the last scenario will become more prevalent with the trend towards system-on-a-chip integration.

For complex algorithms, the need for storage exceeding a few registers is currently the most common one. Base FLAME already covers the simple case in which a module can request to be placed such that it includes an on-chip memory bank. However, this memory bank (a rare resource on today's FPGA architectures) is then lost to other modules. Furthermore, when the module requires even more storage space, it will need to access the external memory bus in a manner coordinated with the rest of the datapath.

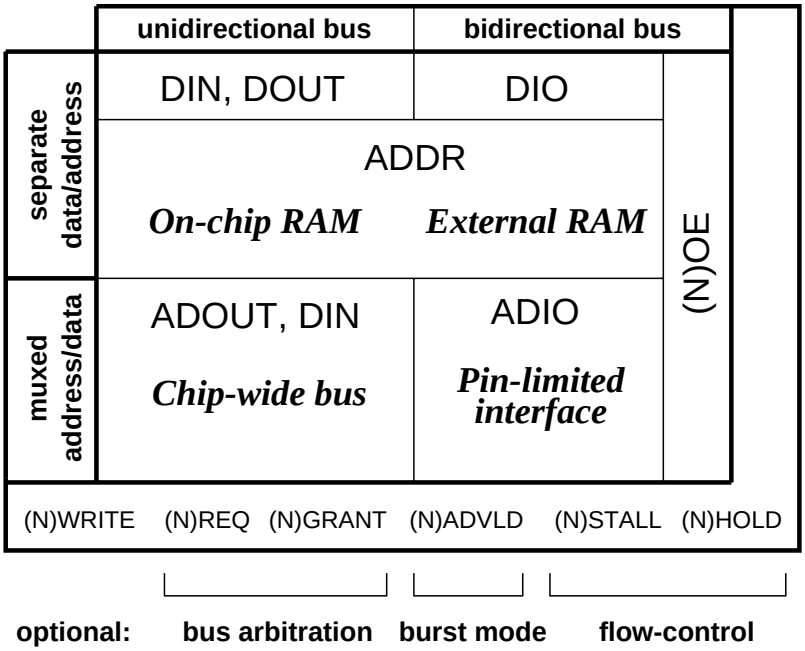

Figure 3. Bus architecture taxonomy

The FLAME Shared Access Conventions (SAC) specify a framework that abstracts the most common bus architectures and protocols in a portable manner. Figure 5 shows a taxonomy of different bus architectures. It covers the various degrees of directionality and muxing as well as mandatory and optional control signals. The italic labels in the figure indicate the most common application area for the indicated bus protocol. Table 5 lists the attributes that can be assigned to FLAME ports which enable these to access an external resource.

For example, on-chip memory, such as the Xilinx Virtex BlockSelectRAMs, is often connected using separate busses for data input DIN, data output DOUT, and address ADDR. In the simplest case, a programmable-write signal WRITE (or NWRITE for negative polarity) is all that is needed to access the memory. Contrast this with external memory, that generally uses a bidi-

Table II. Interface signal attributes

Name	Kind	Parameters	Description
DIN	in	*rsrcnr*	Data input port. Data width determined by WIDTH attribute.
DOUT	out	*rsrcnr*	Data output port. Data width determined by WIDTH attribute.
ADOUT	out	*rsrcnr*	Multiplexed address/data output. Address width determined by WIDTH attribute.
ADDR	out	*rsrcnr*	Address output port. Address width determined by WIDTH attribute.
ADIO	i/o	*rsrcnr dwidth*	Multiplexed address output/data input/output. Address width determined by WIDTH attribute, data width determined by *dwidth*.
WRITE	out	*rsrcnr*	Assert to write data. Programmable polarity single-bit signal.
OE	out	*rsrcnr*	Assert to enable resource output drivers. Programmable polarity single-bit signal.
REQ	out	*rsrscnr*	Assert to request resource access. Programmable polarity single-bit signal.
GRANT	in	*rsrcnr*	Asserted when resource access is granted. Programmable polarity single-bit signal.
ADVLD	out	*rsrcnr*	Assert to load new address for burst-mode. Programmable polarity single-bit signal.
STALL	in	*rsrcnr*	Asserted when no data available. Programmable polarity single-bit signal.
HOLD	out	*rsrcnr*	Assert to pause burst transfer. Programmable polarity single-bit signal.

rectional (tri-stateable) data bus DIO, and thus requires an additional output-enable signal OE for the external drivers. Another architecture that attempts to reduce the number of busses while avoiding the need for tristate buffers (which might also be rare on the FPGA) uses a shared bus for all output signals, thus combining address and data output into ADOUT and a dedicated bus for input data DIN. This approach can be employed to good effect for the shared datapath-wide on-chip bus. In the extreme case, addresses and both input and output data are all multiplexed over the same bidirectional bus. This interface is less common for on-chip use, but might be applicable when communicating with I/O pin-limited external devices.

In general, the read/write control signal is always present (exceptions are ROMs and read-only devices such as temperature sensors). However, a num-

Table III. SAC resource attributes

Name	Parameters	Description
READLAT	*a-d-edges*	# clock edges from address to read data valid.
WRITELAT	*a-d-edges*	# clock edges between write address and write data.
BURSTLAT	*d-d-edges*	# clock edges per data item in a burst-transfer.
BUSTURN	*r-w w-r*	# clock edges to turn bidirectional bus around (switch from read-to-write and write-to-read).
MAXBURST	*size*	Maximum # words in a burst transfer.

ber of optional signals can be employed to satisfy more complicated interface needs. If the access request REQ and grant GRANT signals are not present in the interface, the central FSM can assume that the module wants control over the shared resource as long as it executes. Otherwise, control can be requested from the central FSM on an as-needed basis and granted dynamically. A burst-mode interface (multiple data items transferred per address) can be implemented using the advance-or-load signal ADVLD, that becomes asserted (=‘load’) when a new address has been put on the bus. When deasserted, transfers will proceed ascending from the last loaded addresses. Some devices (e.g., transfers over the PCI bus) have a variable latency. They can be accessed using an output STALL that allows the device to halt data transfers initiated by the user circuit. Analogously, the user circuit can use the HOLD input to pause data transfers initiated by the device (e.g., a burst transfer in progress).

The actual architecture requested by a module can be inferred from its physical interface. Ports are flagged with role attributes such as `(DIN 0)`, indicating, e.g., that this port should be the input bus from resource 0. Various parameters such as address ranges, data width, sub-word write-enables can be inferred from these and the standard FLAME port parameters (see Section 3.5).

A dedicated RESOURCE section in the FLAME “synthesis” view describes the nature of the specific resource requested (e.g., RAM, ROM, DAC, shared memory etc.) and its timing parameters on a per-resource number (*rsrcnr* in Table 5) basis. These attributes (shown in Table 5) include the latency in half-cycles (clock edges) for read address-to-data, write address-to-data, burst data-to-data, and bus turnaround (for bidirectional busses). For burst-capable resources, the maximum number of words in a burst is also indicated. Edges are used instead of clock cycles to allow the description of double-data rate (DDR) resources.

As an example, a module requesting access to 4K of 16-bit words using a protocol for synchronous zero-bus turn-around memory could establish the following physical interface in its “synthesis” view:

```
(INTERFACE (PHYSICAL
 (INOUT  (("D")  (WIDTH 16) (DIO 0)))
 (OUTPUT (("A")  (WIDTH 12) (ADDR 0))
         (("nE") (WIDTH 1)  (NOE 0) )
         (("W")  (WIDTH 1)  (WE 0)  ))
...
(RESOURCE ((0) "ram"
  (READLAT 2) (WRITELAT 2) (BUSTURN 0 0)))
```

Note that this resource always has fixed latencies (no flow-control), no burst-mode (each transfer will provide a valid address), and will be exclusively allocated to the module during the entire time it is executing (no bus arbitration signals).

Additional features include the automatic conversion between different access protocols (e.g., burst access to a resource incapable of burst transfers) and non-unit stride bursts for streaming computations.

Together, these functions are sufficient to portably satisfy a wide spectrum of interface needs across different FPGA architectures and enable the seamless automatic integration of complex hardware objects (e.g., 3rd party IP blocks) into a synthesized datapath.

6. Status

FLAME currently consists of a comprehensive specification [12] and a technology demonstrator [9] containing the base library and a sample transport protocol. A generator library implemented using JHDL [13] providing all of the functions in the FLAME Primitives Catalog will be released shortly. Research also continues on adding a FLAME interface and the FLAME Shared Access Conventions to an experimental fully automatic compile flow targeting real hardware (based on the Xilinx Virtex FPGA series [14]).

7. Summary

FLAME is a general-purpose method that allows high-level design flows to evaluate and create hardware objects targeting configurable computing machines. The FLAME Primitives Catalog formulates a contract between module users and suppliers that covers the functions and interfaces (but not

the implementation!) of a basic set of hardware objects. Instances of complex modules can obtain access to shared resources such as memory by adhering to the FLAME Shared Access Conventions, which allow for flexible yet standardized connectivity and bus protocols. In concert, these components enable the interplay of both software tools and hardware objects to create powerful configurable computing solutions.

References

1. Xilinx Inc. X-BLOX Reference. EDA tool documentation, San Jose (CA), 1995
2. J. Dittmer and H. Sadewasser. Parametrisierbare Modulgeneratoren für die FPGA-Familie Xilinx XC4000. Diploma thesis, Tech. Univ. Braunschweig (Germany), 1995
3. M. Chu and N. Weaver et al. Object Oriented Circuit Generators in Java. *Proc. IEEE Symp. on FCCM*, Napa Valley (CA), 1998. pp. 158-166
4. Xilinx Inc. CORE Generator System User Guide. EDA tool documentation. San Jose (CA), 1998
5. M.B. Gokhale and J.M. Stone. NAPA-C: Compiling for a Hybrid RISC/FPGA Architecture. *Proc. IEEE Symp. on FCCM*, Napa Valley (CA), 1998. pp. 126-135
6. R. Harr. 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. A. Koch. Adaptive Rechensysteme und ihre Entwurfswerkzeuge. *Proc. E.I.S. Workshop*. Dresden, Germany, 2001. pp. 303-307
8. M. Hall. Design Environment for ACS (DEFACTO). *Proc. ACS PI Meeting*, `http://www.dyncorp-is.com/darpa/meeting/acs98apr/defacto.ppt`, Napa Valley (CA), 1998
9. A. Koch. FLAME/Java Release 0.1.1. `http://www.icsi.berkeley.edu/~akoch/research.html#FLAME`, Berkeley (CA), 1998
10. A. Koch. Regular Datapaths on Field-Programmable Gate Arrays. Doctoral thesis, Tech. Univ. Braunschweig (Germany), 1997
11. Electronics Industry Association (EIA), Library of Parametrized Modules. EIA/IS-103, 1993
12. A. Koch. FLAME: A Flexible API for Module-based Environments – User's Guide and Manual. `http://www.icsi.berkeley.edu/~akoch/research.html#FLAME`, Berkeley (CA), 2000
13. P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. *Proc. IEEE Symp. on FCCMs*, Napa Valley (CA), 1998. pp. 175-184
14. Xilinx Inc. Virtex 2.5V FPGAs. device data sheet. San Jose (CA), 1999

Address for Offprints:
Andreas Koch
Tech. Univ. Braunschweig (E.I.S.)
Gaußstr. 11
D-38106 Braunschweig Germany