Efficient Integration of Pipelined IP Blocks into Automatically Compiled Datapaths

Andreas Koch
Technical University of Darmstadt
FB20, Embedded Systems and Applications Group
Hochschulstr. 11
D-64289 Darmstadt, Germany
koch@esa.informatik.tu-darmstadt.de

Abstract

Compilers for reconfigurable computers aim to generate problem-specific optimized datapaths for kernels extracted from an input language. In many cases, however, judicious use of pre-existing manually optimized IP blocks within *these datapaths could improve the compute performance even further. The integration of IP blocks into the compiled data paths poses a different set of problems than stitching together IPs to form a system-on-chip, though: Instead of the loose coupling using standard busses employed by SoCs, the one between datapath and IP block must be much tighter. To this end, we propose a concise language that can be efficiently synthesized using a template-based approach for automatically generating lightweight data and control interfaces at the data path level.*

Keywords*: high-level synthesis, interface synthesis, hardware compiler, IP-based design, datapath, controller*

1 Introduction

Automatic high-level language compilers [1] [2] are one of the prime means to make the compute power of reconfigurable computers available to developers. However, despite the progress in such compile flows, the generated hardware often does not reach the quality of designs carefully optimized by an expert designer. Thus, it becomes desirable to tightly integrate optimized custom IP blocks with the compiler-generated datapath.

While this mixed method is still new in the world of hardware design, it has been established for decades in the software area. There, it is quite common to call highly optimized assembly code libraries (e.g., for math or graphics) from high-level programming languages. Thanks to well defined binary interface and calling conventions, cross-abstraction level calls are easily performed.

For hardware design, the situation is much more complex. One of the reasons appears to be the increased flexibility of custom hardware compared to a fixed-function processor: The same functionality can be realized in dedicated hardware in many different ways and thus be perfectly matched to the rest of the system environment.

However, automatically building a complete system-on-chip from these disparate components is difficult. While some attempts have been made to standardize on-chip communications [3] [4] [5], they have not achieved total success. Many IP blocks still do not use one of these proposed standard interfaces, but instead rely on their own custom interfaces, which have to be "wrapped" before connecting to a standard bus.

Furthermore, when compiling an accelerator unit for a reconfigurable computer, the generated hardware should fully exploit the adaptive nature of the target architecture: Reconfigurability allows the use of highly efficient problem-specific hardware structures, instead of the more general approaches (e.g., networks-on-chip) that are often used in the ASIC world.

Thus, instead of using a general-purpose communications structure to assemble a system-on-chip, we are aiming for the tight integration of a larger number of smaller IP blocks directly *into* the compiled datapaths. For this applications, the standard busses mentioned above are generally too heavyweight, with specialized high-bandwidth low-latency point-to-point connections being far preferable.

One of the tasks that has to be performed to achieve this goal is the creation of interface controllers that translate from the various IP-specific protocols for initialization, data exchange, etc., to a common protocol compatible with the central data path controller. Ideally, the creation of the wrappers should be performed "on-the-fly" during hardware compilation, without requiring time-consuming HDL-based synthesis steps. However, the wrappers must

be capable of handling even complex control schemes and pipelined operation. Prior work [6] [7] has already detailed UCODE, a simple language for concisely describing such interface controllers. We now contribute a novel way to quickly synthesize hardware from UCODE: A subcircuit "template" is associated with each kind of UCODE instruction; these templates are then composed following the UCODE description to build the entire interface controller circuit. As will be shown in Section 6, area/time trade-offs can easily be performed by changing the templates and mapping rules.

2 Related Work

Flexibly connecting mismatched interfaces has been the subject of many research efforts. The approaches range from constructing product FSMs to build protocol converters [8] using libraries of interface modules [9] [10] to extracting event graphs from timing diagrams [11]. A good overview and a formal model of the problem can be found in [12].

However, none of of these methods matches our scenario of tightly integrating pre-existing IP blocks into automatically compiled datapaths. For this tight degree of coupling, the FIFOs proposed in [13] are inappropriate. In our usage scenario, FIFOs for each IP block would inordinately increase the latency of the entire data path. Thus, our approach aims to avoid the introduction of additional delay elements.

Another common approach [13] [14] relies on extracting the interface description from the HDL code of the IP blocks. With the increasing use of encrypted soft-cores or netlist-only firm cores, this approach becomes rather impractical. To avoid these difficulties, we rely on UCODE as an IP-external description of interface characteristics.

Pipelining, a feature crucial for high throughput datapaths, is also often lacking from the approaches listed here. There have been some efforts to apply a data-flow based approach to the problem, but they sometimes lack flexibility. For example, the technique in [15] can only handle static data-flow and requires a fixed send-receive protocol. Other work, such as [16], is more flexible, but does not cover the direct hardware mapping of the described primitives. In this text, we extend UCODE as a flexible description for interface protocols with an efficient mapping onto actual hardware.

3 Target Architecture

Our application setting is shown in Figure 1. IP blocks are to be inserted into compiler-generated datapath by automatically synthesizing a thin wrapper both on the data and the control sides, connected using dedicated point-to-point links to the datapath and the global controller. This global controller is responsible for higher-level control decisions

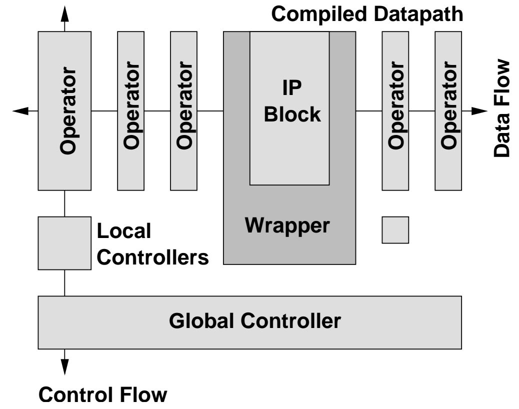

Figure 1. Application scenario

(e.g., switching an IP block into another operating mode, starting/canceling speculative execution). The wrapper controller in turn acts on a lower level and orchestrates the control sequencing and data exchange *within* a function selected by the global controller. On the data side, the formats used in the datapath and on the IP block are assumed to be mostly compatible. However, minor transformations, such as serial-to-parallel conversions, bus (de)composition, and physical-logical port renaming are supported in the wrapper.

The following sections will discuss how to concisely describe the wrapper function, the manner of integration with the global controller, the actual template-based synthesis and optimized mapping of the abstract circuit to real hardware.

4 Interface Description

Similar to the approach in [14] and [16], we compose the descriptions of the controller functions from a small number of primitives. However, we also allow the description of pipelining, port renaming, and embedded wired logic. All of our primitives (called UCODEs) have been defined in terms of underlying abstract hardware functions. These *templates* can be composed and then efficiently mapped to the target architecture (but not necessarily exactly as depicted, see Section 6).

When a new IP block is prepared for automatic integration, it is the task of a human expert to author the corresponding UCODE descriptions for the various capabilities of the block. These descriptions will generally be manually extracted from the data sheets and manuals delivered by the IP vendor.

In this work, we concentrate on the low-level description and template-based synthesis of the wrapper. The complete specification [7] also covers higher-level constructs such as initialization, parallel/serial execution modes etc.

```
//UCODE for cache_write operation

Seq ucwrite = new FSeq(); // create empty sequence of UCODE objects

ucwrite.cat( // combinationally apply data and control signals
  new Level(
    new FSeq(
      new PortValue(CACHE_OE, 0),
      new PortValue(CACHE_WE, 1),
      new PortPort(CACHE_ADDR, addr),
      new PortPort(new BusPort(CACHE_WIDTH_16BIT), new BusPort(width, 0)),
      new PortPort(new BusPort(CACHE_WIDTH_8BIT), new BusPort(width, 1)),
      new PortPort(CACHE_WRITE, datain))));

ucwrite.cat( // wait for cache port ready
  new Continue(new PortValue(CACHE_STALL, 0)));

ucwrite.cat( // signals must be kept stable to next edge for sampling by cache port
  new PosEdge(new FSeq(
    new PortValue(CACHE_OE, 0),
    new PortValue(CACHE_WE, 1),
    new PortPort(CACHE_ADDR, addr),
    new PortPort(new BusPort(CACHE_WIDTH_16BIT), new BusPort(width, 0)),
    new PortPort(new BusPort(CACHE_WIDTH_8BIT), new BusPort(width, 1)),
    new PortPort(CACHE_WRITE, datain))));
```

Figure 2. Example for UCODE embedded in Java

4.1 Compute Model

Despite the hardware-centric formulation of our controller behavior, the underlying model of computation has formal roots in Petri nets: The presence of a token (logic '1') indicates an active state, multiple states may be active at the same time, and tokens may be created, deleted, and rerouted during the controller execution. All of our primitives accept a token, many also propagate it (possibly after modification). The global controller activates a wrapper controller by injecting an initial token into the first state. In a similar fashion, a token leaving the final state can indicate completion of the wrapper operation and transfer control back to the global controller. Pipelining, however, requires additional infrastructure (described in Section 5).

4.2 Input/Output

Compared to [14], I/O has been been unified here (no distinction is made between control and data) and extended (we explicitly model *time*, currently defined by edges of a *single* clock domain).

The I/O operations shown in Figure 3 are initially distinguished by whether they operate combinationally or sequentially. In the first case, the UCODE statement `LEVEL` is used, in the second one, the `POSEDGE` and `NEGEDGE` statements will be employed. The latter differentiate between synchronizing to the rising or falling edge of the central clock.

Note that the textual syntax shown here is purely a human-readable convenience. After it has been written to describe a specific IP block, UCODE is only handled within design tools, and can thus be represented more efficiently in binary form. For example, our current implementation of a UCODE-based tool flow actually uses Java object graphs for efficient storage and manipulation of the UCODE de-

```
io := iomode [{ portmap }];
iomode := io_comb | io_seq ';' ;
io_comb := 'LEVEL';
io_seq := ('POSEDGE' | 'NEGEDGE' ) [repeat];
repeat := '*' count;
count := cardinal;
portmap := '(' physport logport  ')';
physport := port | literal ;
logport := port | literal ;
literal := cardinal;
port := name ['[' [msb ':'] lsb ']'];
msb := cardinal;
lsb := cardinal;
```

Figure 3. Input/Output primitives

scriptions: The programs are stored as sequences of statement objects, and textual references, e.g., to I/O ports, have been replaced by direct references to the corresponding design database objects. Figure 2 shows an example for such a UCODE fragment embedded in Java. The fragment shown describes the memory write operation of a value `datain` to address `addr` via a cache interface [17].

As primary arguments, each of the primitives takes a set of *portmap* pairs, each pair associating a physical port with a logical port on a bus or sub-bus basis. Such a pair represents a permanent (wire) or temporary (muxed/demuxed) connection between the two ports. Alternatively, one of the ports may be replaced by a constant literal. This indicates the application of the literal value to the remaining port of the pair.

Figure 4 shows the underlying hardware templates of the sequential operators. When the state is activated by an arriving '1' token, the associated action occurs: In the input case (a), the selected logical input port is applied to the specified physical port of the IP block in time to be sampled for the *next* clock edge. In the control case (b), the presence of the

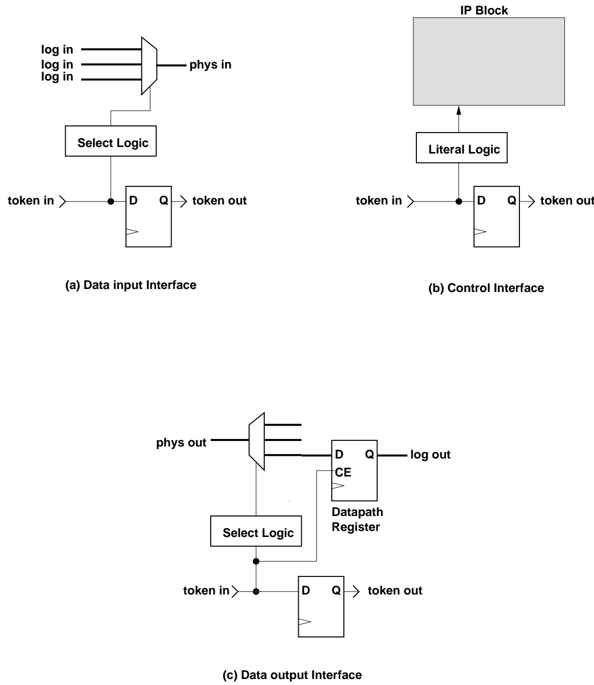

Figure 4. Sequential I/O templates

token indicates the application of a literal value (generated by the Literal Logic) to one or more physical ports of the IP Block. Finally, in the output case (c), the given physical output port is applied to the selected logical output to be sampled into a datapath register at the next clock edge. After the clock edge indicated by the UCODE, the token is then propagated.

The combinational I/O operations depicted in Figure 5 operate similarly. The crucial difference is the now purely combinational nature of the operation (no time steps as defined by clock edges pass).

It is obvious, that the final Logic blocks controlling the multiplexers and the datapath control inputs must be composed by merging the Logic blocks of *all* UCODEs that apply to the same port.

Consider the following example: Assume that an IP block implements the logical behavior `mul(prod,a,b)`. The physical interface, however, has a single input port `D` through which to load the multiplicator and multiplicand sequentially on successive clock cycles. The loading process must be started by raising the control input `s`. After accepting the multiplicand, the result becomes valid on the physical output port `Y` four clocks later and can then be sampled back into the datapath on the following clock edge.

Figure 6 shows the UCODE description of both the control and data interfaces in the wrapper. The *abstract* (technology independent) circuit for this description can be generated simply by composing the templates and merging the Logic blocks (Figure 7). Due to the simplicity of the example, the Logic blocks are trivial or have even been optimized away entirely (e.g., since there is a 1-1 mapping of the physical port `Y` to the logical port `prod`, no demultiplexer and associated control logic are required). The hardware was composed by chaining the circuits underlying the UCODE primitives via their token inputs and outputs. For each primitive, the form appropriate for data (ports `D`, `Y`) or control (port `s`) manipulation is employed.

The shift and wired logic operations mentioned in Section 4 are realized by offsetting the `msb` and `lsb` indices of physical and logical ports against each other. The UCODE in Figure 8.a sign-extends the 4b physical port `D` to map to the 8b logical port `x`. In a similar fashion, split ports may be handled. The code in Figure 8.b assembles two physical ports to map to a wider logical port. The expression in Figure 8.c converts a 22b word address on `PA` to a byte-oriented address `addr`.

4.3 Control Flow

While the I/O primitives can already handle simple IP blocks on their own, many blocks have more complex interfacing requirements. Two of the most common ones are handshaking and (closely related) variable execution times (latencies). For these cases, the straightline execution of the I/O UCODEs no longer suffices. The `CONTINUE` UCODE shown in Figure 9 is similar to the *wait for event* primitive in [14], but extends the concept by allowing logical expressions in a sum-of-products form.

Each *portequals* states that the indicated physical port

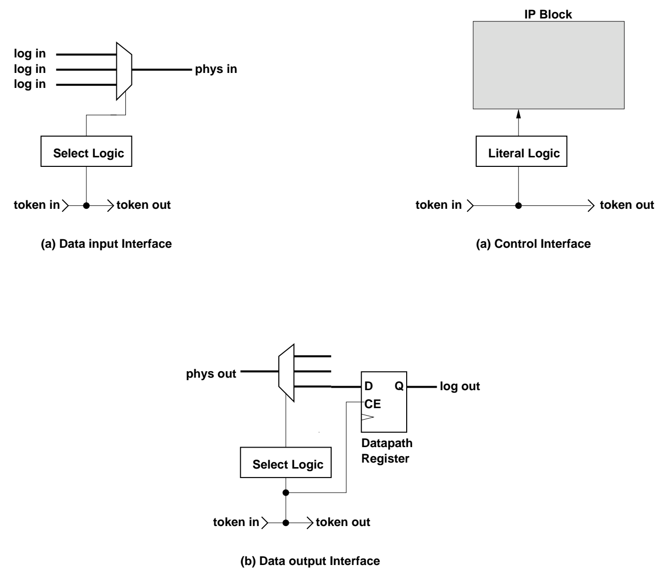

Figure 5. Combinational I/O templates

(or bit sub-range thereof) must be equal to the given literal value. The UCODE waits in the current I/O state until *all* conditions within a CONTINUE become true (logical product), or that *any* of a group of successive CONTINUE primitives match (logical sum).

The hardware templates underlying this UCODE are shown in Figure 10. The Condition Logic is derived by ANDing the conditions within each CONTINUE and ORing these separate outputs for successive CONTINUE statements.

The statement operates by routing an incoming token back to the last active I/O statement. Only if the joint condition of all successive CONTINUE statements becomes true, will the token continue past the UCODE to the next statement. The CONTINUE itself is purely combinational. A synchronous mode of execution can be achieved by following the CONTINUE with one of the sequential I/O statements POSEDGE or NEGEDGE.

As an example, reconsider the integration of the Mult16x16 IP block of the previous section. But here, instead of the fixed latency of four clock cycles, the IP block indicates the availability of a result in time for the next rising clock edge using a '1' on the physical port R. The corresponding UCODE fragment is shown in Figure 11, the corresponding hardware in Figure 12.

The back-edge of the CONTINUE statement routes the token to the input of previous I/O statement (the second POSEDGE of the fragment). Due to the trivial condition, the Condition Logic collapses to a single wire from R to the CONTINUE hardware. In a more complex application, the Logic would hold the sum-of-products realization of the intra- and inter-statement conditions.

4.4 Pipelining

For our application of tightly integrating an IP block into a heavily pipelined datapath, it is crucial to be able to describe pipelining characteristics. Specifically, we want to be able to model the prologue, the steady-state, and the epilogue of a pipelined IP block. START, shown in Figure 13, separates the prologue from the steady state. It also merges an incoming token from the back-edge into the forward direction (beginning the next pipeline iteration).

RESTART (Figure 14) indicates the beginning of the epilogue and duplicates an incoming token: One copy is passed forward into the epilogue of the pipeline iteration, the other copy is passed backward into the START circuitry, beginning the next pipeline iteration in the steady-state. RESTART effectively creates a new thread of execution which results in

```
POSEDGE (S 1) (D[15:0] a[15:0]);
POSEDGE (S 0) (D[15:0] b[15:0]);
POSEDGE; POSEDGE; POSEDGE; POSEDGE;
POSEDGE (Y[31:0] prod[31:0]);
```

Figure 6. UCODE for multiplier example

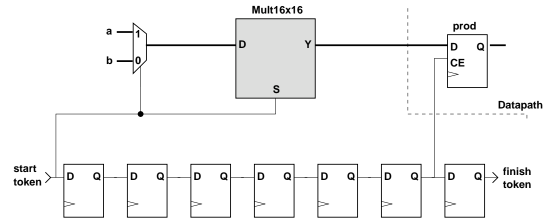

Figure 7. Wrapper for multiplier IP block

```
a) POSEDGE (D[3] x[7]) (D[3] x[6])
           (D[3] x[5]) (D[3] x[4])
           (D[3:0] x[3:0]);
b) POSEDGE (H[15:0] data[31:16])
           (L[15:0] data[15:0]);
c) POSEDGE (PA[21:0] addr[23:2])
           (0 addr[1:0]);
```

Figure 8. Wired logic and shifts

```
continue := 'CONTINUE' { portequals } ';' ;
portequals := '(' physport literal ')';
```

Figure 9. Flow control

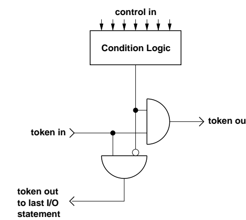

Figure 10. Control flow templates

```
POSEDGE  (S 1) (D[15:0] a[15:0]);
POSEDGE  (S 0) (D[15:0] b[15:0]);
CONTINUE (R 1);
POSEDGE  (Y[31:0] prod[31:0]);
```

Figure 11. UCODE for variable latency multiplier

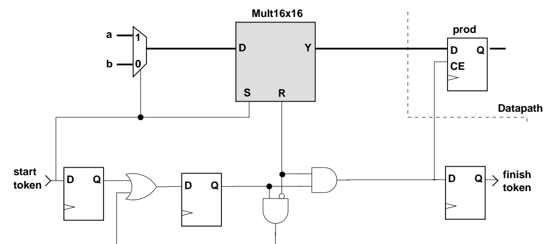

Figure 12. Wrapper for variable latency multiplier

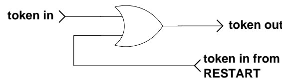

Figure 13. Pipeline steady-state join template

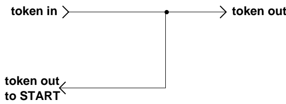

Figure 14. Pipeline steady-state fork template

multiple states becoming active in parallel (Petri net-like). Figure 15 shows the pipeline modeled by these UCODEs.

Only one START/RESTART combo may exist within a UCODE program. This construct is the only way to actually iterate within the wrapper controller. All other loops must be realized in the global controller by repeatedly activating the wrapper controller. Furthermore, exploiting pipeline parallelism requires additional circuitry around the wrapper controller for cleanly terminating (draining) the pipeline. This will be discussed in Section 5.

To give an example on the use of pipelining, we will stay with our regular multiplier, but posit this time that it has a total latency of seven cycles (including loading the operands) and allows pipelined operation with an initiation interval of four cycles (then the next operands can be loaded). The UCODE description in Figure 16 models this behavior.

This UCODE fragment has an empty prologue, but the steady-state and epilogue follow the model of Figure 15. The corresponding hardware is shown in Figure 17.

5 Pipeline Administration

The abstract wrapper circuits created from the UCODE templates can be modified to optionally provide additional capabilities for the global controller. These extensions include cleanly stopping the pipeline and waiting for it to drain. For clarity of the following figures, we show only the abstract state flip-flops, but omit the combinational logic (e.g., for CONTINUE statements) in between.

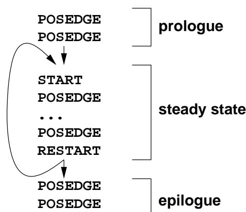

Figure 15. Model of pipeline structure

```
START;
POSEDGE (S 1) (D[15:0] a[15:0]);
POSEDGE (S 0) (D[15:0] b[15:0]);
POSEDGE; POSEDGE;
RESTART;
POSEDGE; POSEDGE;
POSEDGE (Y[31:0] prod[31:0]);
```

Figure 16. UCODE for pipelined multiplier

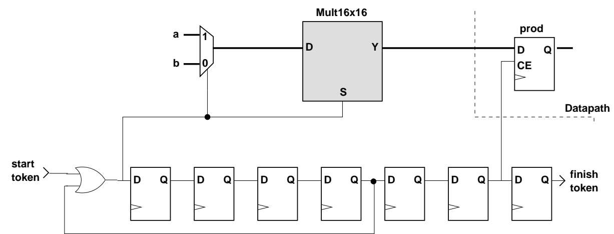

Figure 17. Wrapper for pipelined multiplier

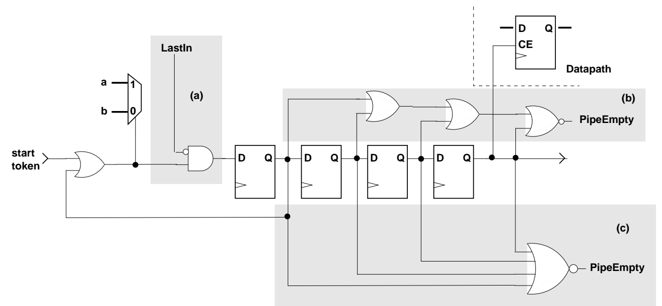

Figure 18. Stopping and combinationally draining the pipeline

5.1 Stopping the Pipeline

This functionality is provided by adding a global-controller manipulated input `LastIn` into the back-edge from `RESTART` to `START` via an AND with inverted input (Figure 18.a). It is crucial that this gate is inserted directly preceding the `D` input of the abstract flip-flop, otherwise the control signals generated by this `POSEDGE` or `NEGEDGE` statement (the mux control in the figure) would become invalid prematurely. By asserting `LastIn` simultaneously with the application of the last set of input data `a`, the final pipeline iteration will be started.

5.2 Draining the Pipeline

With variable-latency elements in the pipeline, it becomes difficult for the global controller to determine when the last data item has been completely processed. Two basic approaches present themselves: One method detects whether the pipeline is empty by checking that no abstract flip-flop holds a valid token and asserts the port `PipeEmpty` in that case. Depending on the speed/area requirements and the capabilities of the target technology, this can be realized either in a serial or in parallel fashion (Figure 18.b and .c). If any slow-down due to cascaded or very wide logic gates is unacceptable, the approach shown in Figure 19 can be used. While it completely avoids long combinational paths, it requires double the number of abstract flip-flops.

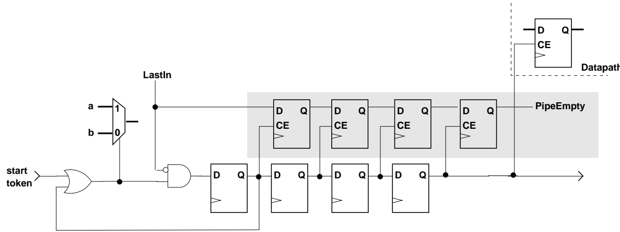

Figure 19. Sequentially draining the pipeline

6 Optimized Mapping

Even though we have expressed the precise semantics of the individual UCODE statements in terms of composed abstract hardware templates, this by no means indicates that the actually implemented hardware must have the same structure. On the contrary, in many cases it is beneficial to map only an optimized form of the wrapper to the target technology. Since our primary target are FPGAs, specifically the Xilinx Virtex FPGA architectures, we will discuss some procedures applicable to these devices.

While our abstract model of one flip-flop per state (one-hot encoded) has advantages both in theory (easy modeling of parallel states) and in practice (distributed controller, less routing congestion), in certain cases the flip-flop require-

```
; initialize
POSEDGE (CE 1) (SCALE_MODE 0)
        (FWD_INV 1) (START 1)
POSEDGE (START 0)
; start of steady-state
START
; wait for acceptance of first FFT block
CONTINUE (MODE_CE 1)
; write 16 time domain samples
POSEDGE *16 (DI_R[15:0] time_r[15:0])
            (DI_I[15:0] time_i[15:0])
; fork control flow for pipelining
RESTART
; wait for transformed data
CONTINUE (DONE 1)
; read 16 frequency domain samples
POSEDGE *16 (XK_R[15:0] freq_r[15:0])
            (XK_I[15:0] freq_i[15:0])
```

Figure 20. UCODE for wrapping 16-point FFT

ments exceed the capabilities even of flip-flop rich architectures. In these cases, target-specific blocks such as dedicated shift registers (SRL16) can be employed. Also, the presence of the `*` (repeat) operator indicates that a given delay in itself is not pipelined and can be densely mapped to a counter. Conventional logic synthesis and mapping algorithms [18] [19] are used in a tightly focused fashion to minimize and map the various Logic blocks associated with some UCODE operators.

This composing of templates in UCODE order and the selective application of limited-scope logic synthesis are require only short computation times. They can thus be performed "on-the-fly" during the high-level language compile flow, avoiding a full-scale HDL synthesis step involving complex external tools.

7 Experimental Results

The UCODE language described here has already been used for interfacing of simple [20] and larger IP blocks [21] to automatically generated datapaths.

To show the use of a medium-complexity IP block, Figure 20 depicts the UCODE for wrapping the Xilinx LogiCore 16-Point FFT [22]. After programming the operating mode, it accepts a 16-sample block of time-domain data. After the end of the computation is indicated, 16 frequency-domain samples can be unloaded from the IP block. In a pipelined fashion, the next set of time-domain can be provided to the core when it becomes available again.

Table 1 shows the area and time trade-offs when mapping the abstract hardware to the Virtex-II architecture directly one-hot encoded and using architecture-specific blocks (counters, shift-registers) on a speedgrade -4 device.

8 Future Work

The UCODEs introduced in this work form the core of the specification. However, for reliably interfacing with large IP blocks (e.g., media codecs) in context of [21],

Synthesis Style	Virtex-II Slices	Max. Clock [MHz]
One-Hot	25	467
Counter	13	248
SRL16	8	243

Table 1. Results of template-based synthesis

we have defined extensions such as timeouts and exception handling in the CONTINUE statement that integrate easily and with only minimal hardware overhead into the existing semantics and template-synthesis framework.

While our applications have not required it to date, irregular schedules could be handled elegantly by extending the CONTINUE statement with an implicit conflict controller [23] [24], thus avoiding the need for large Condition Logic blocks in the wrapper controller.

9 Conclusion

Our lightweight approach (compared to full-scale protocol conversion) has proven suitable for practical use. Easily authored, concise UCODE descriptions allow the tight integration even of complex IP blocks into compiled datapaths with minimal computational effort. Instead of full HDL synthesis, simple mapping tools aware of some technology-specific features suffice to implement the actual circuits from the composed templates. The UCODE language and underlying compute model are also easily extended to accommodate future integration requirements.

By using UCODE descriptions to automatically generate efficient interface wrappers, the combination of optimized IP blocks and automatically created data paths can increase the performance of a flow targeting an adaptive computer in a manner similar to transparently calling assembly language routines from a high-level language. The complexity of the calling and parameter transfer mechanisms are hidden from the user by the abstraction of the UCODE description.

References

[1] Li Y.B., Harr R., et al. "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", *Proc. Design Automation Conference*, 2000

[2] Kasprzyk N., Koch A. "High-Level-Language Compilation for Reconfigurable Computers" *Intl. Conf. on Reconfigurable Communication-centric SoCs*, Montpellier (F), 2005

[3] VSI Alliance, "Virtual Component Interface Standard Version 2", www.vsia.org, 2001

[4] ARM Ltd., "AMBA Specification Rev 2.0", http://www.arm.com/products/solutions/AMBA_Spec.html, 2001

[5] IBM Corp., "Core Connect Bus Architecture", http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture,1999

[6] Koch A., "On Tool Integration in High-Performance FPGA Design Flows", *Intl. Workshop on Field-Programmable Logic and Applications*, Glasgow (Scotland), 1999

[7] Koch A., "FLAME: A Flexible API for Module-based Environments", E.I.S. Technical Report 2004-01, Tech. Univ. Braunschweig (Germany), 2004

[8] Passerone R., Rowson J.A. et al., "Automatic Synthesis of Interfaces between Incompatible Protocols", *Proc. Design Automation Conference*, 1998

[9] Sun J.S., Brodersen R.W., "Design of System Interface Modules", *Proc. Intl. Conf. on CAD*, 1992

[10] Lin B., Vercauteren S., "Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation", *Proc. Intl. Conf. on CAD*, 1994

[11] Chou P., Ortega R.B., Boriello G., "Interface Co-Synthesis Techniques for Embedded Systems", *Proc. Intl. Conf. on CAD*, 1995

[12] D'silva V., Sowmya A. et al., "A Formal Approach to Interface Synthesis for System-on-Chip Design", *UNSW-CSE-TR-304*, U New South Wales, 2003

[13] Smith J., DeMicheli G., "Automated Composition of Hardware Components", *Proc. Design Automation Conference*, 1998

[14] Narayan S., Gajski D.D., "Interfacing Incompatible Protocols using Interface Process Generation", *Proc. Design Automation Conference*, 1995

[15] Jung H., Lee K., Ha S., "Efficient Hardware Controller Synthesis for Synchronous Dataflow Graph in System Level Design", *Intl. Symp. on System Synthesis*, 2000

[16] Teifel J., Manohar R., "Static Tokens: Using Dataflow to Automate Concurrent Pipeline Synthesis", *Symp. on Adv. Rsrch. in Async. Circuits and Systems*, 2004

[17] Lange H., Koch A. "Memory Access Schemes for Configurable Processors", *Workshop on Field-Programmable Logic and Applications*, Villach (A), 2000

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

[19] Cong J., Ding Y., "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs", *IEEE Trans. on CAD*, Vol. 13, No. 1, January 1994, pp. 1

[20] Neumann T., Koch A. “A Generic Library for Adaptive Computing Environments”, *Workshop on Field-Programmable Logic and Applications*, Belfast (UK), 2001

[21] Lange H., Koch A. “Hardware/Software-Codesign by Automatic Embedding of Complex IP Cores” *Intl. Conf. On Field-Programmable Logic (FPL)*, Antwerp (BE), 2004

[22] Xilinx Inc., “High-Performance 16-Point Complex FFT/IFFT V1.0”, *product specification*, 2001

[23] Davidson E.S., Shar L.E., Thomas A.T., Patel J.H., “Effective Control for Pipelined Computers”, *Proc. COMPCON*, 1975

[24] Schaumont P., Vanthournout B., Bolsens I., DeMan H., “Synthesis of Pipelined DSP Accelerators with Dynamic Scheduling”, *Proc. Intl. Symp. on Systems Synthesis*, 1995