Automatic Generation of Memory Interfaces

David Kammler, Bastian Bauwens, Ernst Martin Witte,
Gerd Ascheid, Rainer Leupers, and Heinrich Meyr
Institute for Integrated Signal Processing Systems, RWTH Aachen University
52056 Aachen, Germany
Email: kammler@iss.rwth-aachen.de

Anupam Chattopadhyay
CoWare India Private Ltd.
Logix Techno Park, Sector 127
Noida - 201 301, India
Email: anupamc@coware.com

***Abstract*— With the growing market for multi-processor system-on-chip (MPSoC) solutions, application-specific instruction-set processors (ASIPs) gain importance as they allow for a wide tradeoff between flexibility and efficiency in such a system. Their development is aided by architecture description languages (ADLs) supporting the automatic generation of architecture specific tool sets as well as synthesizable register transfer level (RTL) implementations from a single architecture model. However, these generated implementations have to be manually adapted to the interfaces of dedicated memories or memory controllers, slowing down the design space exploration regarding the memory architecture. In order to overcome this drawback, this work extends RTL code generation from ADL models with the automatic generation of memory interfaces. This is accomplished by introducing a new abstract and versatile description format for memory interfaces and their timing protocols.**

***Index Terms*—Architecture description language (ADL), application-specific instruction-set processor (ASIP), memory interface**

I. Introduction

Nowadays, the market for multi-processor system-on-chip (MPSoC) solutions is expanding dramatically. Often, the development of such an MPSoC includes the design of new processor architectures which are tailored to a particular application. A common technique to develop these application-specific instruction-set processors (ASIPs) is the use of an architecture description language (ADL) [1]–[7]. Present ADL tool suites enable the automatic generation of a fully synthesizable hardware description language (HDL) model of the architecture on register transfer level (RTL) [8]–[11]. Optimizations and standard processor features like debug mechanisms are supported by the automatic generation process making the generated model suitable for final implementation. This decreases development time drastically as designers can concentrate on the actual architectural features on the high abstraction level of ADLs rather than spending time on detailed modeling on RTL. Accesses to memories and busses are usually modeled as abstract function calls in order to e.g. transfer the address or data. Neither the definition of the interface pins nor a highly accurate description of the timing protocol are required on this level. However, both are mandatory for an accurate implementation on RTL. Adding this low level information to the ADL model by specifying the pins and their usage directly is no option. This would cause an overhead and lower the abstraction level of the ADL model inadequately, making it complex, hard to maintain and improper for fast design space exploration. Especially, attaching different memory types to the processor would result in many changes to the model, thus slowing down dramatically the exploration of memory architectures which is of special importance for the development of tailored ASIPs. Moreover, the speed of the generated processor simulator would decrease with the increasing detail of the model.

In this paper we present a solution for modeling memory interfaces in ADLs avoiding the previous drawbacks. The contribution of this paper is the development of a new abstract and versatile description format for memory interfaces, which is external to the actual ADL model of the processor core. Such a memory interface description (MID) covers the definition of memory ports including their pins as well as the timing protocol including the usage of address/data busses with fixed timing or handshake mechanisms. Keeping the MID separate from the ADL model has several advantages:

Orthogonalization of processor model and memory interface description: Separating the ADL model of the processor from the MID allows for fast design space exploration of the ASIP on ADL level and pin accurate implementation on RTL at the same time with the same ADL model.

Reuse of MIDs: Once specified, MIDs for a dedicated memory or memory system can be reused for other architectures or if several identical memories are attached to a single processor.

Rapid exploration of different memory architectures: Different memory architectures, including their physical parameters (e.g. area, timing, etc.), can be explored easily by selecting different MIDs for the HDL code generation.

Maintaining simulator performance: Since the model is not extended with pin level details, the simulator performance is not affected. Nevertheless, the MID can still be used by other tools of the ADL tool suite if required.

Independence of memory vendors: Designers can more easily switch to other vendors with comparable memories by simply modifying or replacing MIDs. Changes to the ADL model in order to adapt the interface are not required.

The MID is defined so that the description of a wide range of real world memory interfaces is possible, from simple synchronous SRAMs to DDR SDRAM controllers. Even memory systems with cache hierarchy are supported as long as an HDL description is available.

Our automatic memory interface generation is integrated into the LISA ADL development framework [7], which has been developed at the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University and is now commercialized by CoWare Inc. [12].

The rest of the paper is organized as follows. Section II discusses related work. Section III gives an overview of features of existing memory interfaces and timing models for memory accesses. The general approach for the automatic generation of memory interfaces is presented in section IV. The format for our MID is discussed in in section V. Section VI shows how the generation process is integrated into our framework and discusses the structure of the generated hardware (HW). The paper is concluded in section VII.

II. Related Work

In the following we give a brief overview on those ADLs and tool suites reported to be used for HW implementation and focus on their capabilities regarding memory interfaces. A detailed discussion of prominent ADLs can be found in [1]. ADLs can be classified into three categories regarding their nature of information: *structural*, *behavioral*, and *mixed* [1].

Structural ADLs describe architectural structures in terms of components and their connectivity. MIMOLA [2] is an example for this class of ADLs. Since its abstraction level is close to RTL, it provides a close link to gate-level synthesis tools but as a result also requires detailed modeling of the memory interface and transfers via this interface.

Behavioral ADLs concentrate on the instruction set of an architecture and neglect detailed HW structures. ISDL [3] as one of these languages has been used for HDL-code generation [8], however no information about the implementation of memory accesses in the resulting Verilog description is available.

Mixed ADLs contain structural information as well as details on the instruction set, like e.g. nML [4]. nML models are processed by the commercial HDL generator GO [13]. Through application programming interfaces (APIs), users can plug in their own HDL implementations of the memory architecture [14]. However, no detailed information on this process is publicly available. Sim-nML [5], a derivative of nML, has also been used for generating sythesizable Verilog models [9]. Here, the processor model is expected to work with a simple single port external memory. EXPRESSION [6] is another mixed ADL used for HDL-code generation [10] utilizing predefined functional blocks provided as VHDL code. For the implementation of memory accesses, appropriate functions for several types of memories, such as SRAM or DRAM, are used. No information about the resulting VHDL code is available, as the library of predefined functional blocks cannot be obtained publicly. As described in [15], EXPRESSION can also be used for the co-exploration of processor architectures and memory subsystems. The main goal is the optimization of resulting compilers. However, no HDL code is generated based on this co-exploration. LISA [7] enables a similar exploration of the memory system [16] and has been used for generation of optimized RTL implementations of processors [11]. However, in [11] the memory system exploration is not combined with the actual architecture implementation on RTL and the support for memory interfaces and protocols is limited to simple predefined interfaces with a fixed protocol.

This work aims at overcoming the described limitations of available solutions by extending HDL-code generation with support of arbitrary memory interfaces and protocols on a cycle- and pin-accurate level without sacrificing the high abstraction level of the ADL model.

III. Memory Systems

In order to enable the automatic generation of memory interfaces, present memory systems first have to be categorized by their individual features and characteristics. Thus, over 25 different types of memories and memory controllers from 14 different vendors have been investigated and abstracted. This section first describes the most important aspects of available memory systems regarding their interface. Features that need to be supported by the HDL code generation are identified. Moreover, a general timing model for memory accesses is introduced.

A. Memory Types and Features

Asynchronous/synchronous memories: The timing of *asynchronous* memories is specified by time intervals and relative signal changes. A clock is not required. For *synchronous* memories, all actions are aligned to a clock signal. The values of interface signals are latched internally at the (typically rising) edge of the clock. While this permits easy synchronization between a clocked processor architecture and a memory system, it also means that the reaction to a control signal cannot happen before the next clock cycle. In order to obtain a higher data throughput, some memory systems also utilize the falling edge of a clock signal to align data transfers (double data rate, DDR) [17]. The automatic generation of memory interfaces focuses on the support of synchronous memory systems. A direct support of asynchronous memories or DDR is currently not available, since these memories are rarely used for ASIP design. However, these types of memories can be accessed via an appropriate memory controller.

Static/dynamic memories: Static memories retain their data contents as long as the memory is supplied with power. On the contrary, *dynamic* memories require a periodic refresh to be issued. There are a lot of memory controllers available, which handle these dynamic aspects, e.g. [18]–[20]. Therefore, refresh mechanisms are not targeted directly by the memory interface generation, and dynamic memories can be connected via appropriate controllers.

Pipelined memories: In case of a *pipelined* memory, a new access can be initiated before the previous access is finished. This behavior is e.g. shown by standard synchronous SRAMs and supported by the HDL code generation.

Burst access: A common technique to increase data transfer rates of memories are *burst accesses*. During a burst access, multiple data words are automatically transferred in subsequent cycles without additional address transfers. The *burst length* of a memory access specifies the number of data words that are transferred during a burst access. The automatic memory interface generation differentiates between two supported types of burst lengths. Memories can support a fixed set of *definite burst lengths* to select from. Other memory systems, especially when using memory controllers, can support the usage of an arbitrary burst length. Then, an interface control signal indicates the end of the burst access. This is referred to as *indefinite burst length*.

Data masking: When transferring a new data word to a memory system during a write access, it can be required to replace only specific bits/bytes at the specified word address. This feature is called *data masking* and is fully supported by the HDL code generation.

Flushing of memory accesses: Some memories allow to cancel pending memory accesses. In the following we will

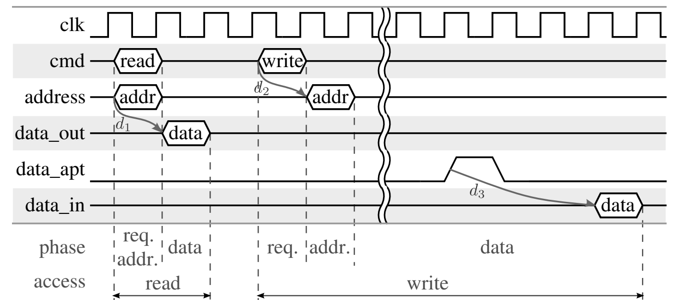

Fig. 1: Example for the memory access timing model

refer to this feature as *flushing* of memory accesses, which is supported by the HDL code generation.

B. Terms and Definitions

The memory system communicates with connected HW components via an interface, which is made up of *pins*. In the scope of this paper, the term pin does not necessarily correspond to a package pin of an external memory. Since memories in SoCs are often placed on the same die as the processor core, a pin also corresponds to internal signals that are used for the communication between processor and memory system. Note, that in this context a single pin can refer to signals with more than one bit.

A subset of the pins of the interface, which offers all required functionality to perform a memory access to the memory system, is referred to as *port*. The interface of a memory system may consist of one or more ports, typically acting independently of each other. A *read port* only supports read accesses to a memory system, whereas a *write port* allows for write accesses only. If both types of accesses are supported on the same port, it is referred to as *read-write port*.

C. Memory Access Timing Model

Independent of the actual specifications of a memory interface, each access to a memory system can be split into three different phases (Fig. 1). During the *request phase*, the access is started and communicated to the memory system, typically by using control signals of the memory interface. The address to be accessed is forwarded to the memory system during the *address phase*. The data transfer is performed during the *data phase*. Individual phases may overlap, e.g., request and address may be provided at the same time (read access, Fig. 1), or require additional delays between the phases (write access, Fig. 1). In order to abstract from the timing varieties of different memory systems, each phase is divided into individual atomic elements as described in the following.

Assignments: An assignment represents the action of a signal or pin being set to a specified value. A typical assignment is the assertion of a control signal for a single cycle in order to request an access to a memory system. The transfer of address and write data are special types of assignments. Reading data from a memory system is also considered as an assignment, even if in this case data is read from the memory system, and thus the direction of the assignment is reversed.

Delay cycles: During a read or write access, memory systems can require a *fixed* number of delay cycles between individual assignments (arrows d_1 to d_3 in Fig. 1). A particular delay always consists of the same number of cycles. Otherwise it would not be possible to determine the end of a delay without any handshake mechanisms.

Handshakes: Memory accesses may require *variable* number of delay cycles. Handshake mechanisms are used to model such access protocols. A Handshake can be described as a specific combination of values of interface signals used to inform about the status of a memory system. In Fig. 1, for instance, a handshake (*data_apt*) is used to indicate that the memory accepts write data after a predefined delay d_3.

IV. A Versatile Memory Interface Approach

On the ADL level, memory accesses are initiated via a dedicated API. This memory API covers all the features and aspects described in the previous section. It is based on the cycle-accurate API presented in [16] and allows for invoking the individual phases (request, address, data) of a memory transfer from different places of the architecture—and therefore also different pipeline stages if desired.

TABLE I: Internal interface signal types

Signal Type	Dir.	Signal Type	Dir.	Signal Type	Dir.
enable read	in	write data	in	busy	out
enable write	in	write data enable	in	address request	out
burst length	in	data mask	in	write data request	out
burst length enable	in	read data	out	read data valid	out
address	in	confirm read	in	address timing	out
address enable	in	flush	in	data timing	out
				last data	out

In order to map high-level accesses via the memory API to low level HW structures, first a general set of signal types is identified, that enables to cover all memory API functionalities including request initialization, address transfer, data transfer (burst and non-burst), control and status propagation. Then, during the HDL code generation process, the memory API function calls are used to derive the signals actually required in the specific case, which finally build up a *unified internal interface*. The 19 predefined signal types required in general are listed in Table I including their directions (from memory perspective). The bit widths of signals carrying address, data, burst length and data mask depend on the characteristics of the memory system. All other signals are single-bit signals. The specific signals which are finally selected for the interface depend on the type of the corresponding memory accesses in the ADL model. A detailed description of the actual protocol is omitted here as the mapping from the API to the internal interface signals is straightforward.

The unified internal interface, flexible and adapted to its individual usage, is not memory specific and does not allow to directly connect the target memory resource. Therefore, it needs to be mapped to a *memory-resource specific external interface*, which is defined by the pins and available ports of the targeted memory resource and independent of the memory accesses throughout the processor model. As depicted in Fig. 2, this mapping is performed by a *memory-interface controller*, which encapsulates all the knowledge about the external interface. For each attached memory resource a dedicated memory-interface controller is instantiated.

The interface pins, which connect the memory to the memory interface controller, are defined by their name, bit width and direction (input or output). Apart from these *interface pins*, the external interface may contain *clock*, *reset* and *unused pins*. The latter are required in order to define the complete interface of the memory system even if it is only used partially.

For HDL code generation, the exact setup of pins and ports

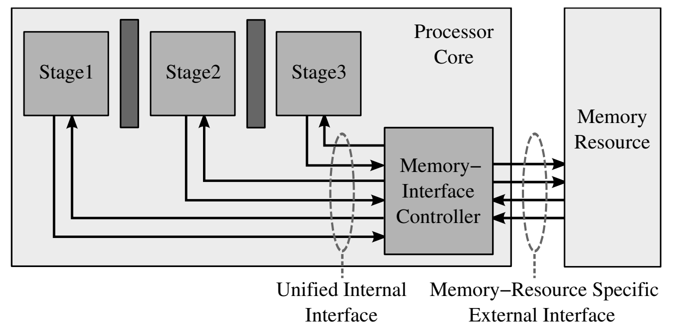

Fig. 2: Mapping the ADL-derived internal interface to a memory-resource specific external interface

composing the external memory interface needs to be covered by a memory interface description (MID), which is discussed in the following.

V. Memory Interface Description

In order to connect the generated HDL model seamlessly to an existing memory resource, the interface and protocol need to be known explicitly during the HDL code generation process. However, detailed information of pins, ports, timing protocol, etc. are not available on ADL level. Actually, the interface information is not architecture but memory dependent. Therefore, we propose to capture MIDs in dedicated external files not directly belonging to the ADL model and thereby orthogonalizing processor model and MID.

An MID needs to capture two different kinds of information: *structural* and *behavioral*.

A. Structural MID Elements

Describing *structural* elements is straightforward since it requires only the definition of pins and ports of a memory interface. As detailed in the previous section, for each pin a name, bit width and direction need to be specified. A *port* groups a set of pins that provides all the functionality required for an access to the memory system. However, additional semantic information also needs to be covered in the port description in order to identify the purpose of each pin. Therefore, previously defined interface pins or bit ranges of a pin can be declared as *data bus*, *address bus* or *data mask bus*. From the direction of the interface pin declared as data bus, the port type (read, write, read/write) can be derived directly. Apart from these specifications, the type (definite or indefinite) and the supported length of *bursts* need to be defined as well. A structural element used by many interface descriptions on RTL are VHDL generics or Verilog parameters as they offer a way to parameterize a component. Thus, the MID offers to set specific values for generics. Additionally, certain resource parameters defined in the ADL model can be propagated via special *generics* to set e.g. the bit widths of the address and data buses.

B. Behavioral MID Elements

Behavioral elements of an MID are needed in order to describe the timing of actions and reactions of the memory system. This means for the memory interface controller, that certain actions need to be triggered under certain conditions in order to map the internal interface protocol to that of the external interface and vice versa. In order to abstract the description of this mapping process, *events* and *commands* are used in the MID. Events describe certain conditions of the internal and external interface, that lead to the execution of actions encapsulated as a list of commands. It is worth mentioning that commands are not necessarily executed immediately when a certain event occurs. Depending on the timing protocol of the memory, delayed execution of commands may be required. Both, events and commands, can be related to either the *internal* or the *external* interface. Internal events and commands need to cover all the functionality of the internal interface (and therefore also of the memory API on ADL level) in a rather abstract manner in order to allow for a convenient description without details of the internal interface protocol. In contrast to that, external events and commands have to act at a pin accurate level. Fig. 3 shows all event handlers and commands that have been identified according to these criteria.

Details on the *internal events* are given in the following. *On initialization* events are triggered immediately after releasing the reset, providing a way to implement an initialization phase, if required by the target memory system. *On request read/write* are events occurring during the request phase of a memory access, while *on read/write data* events correspond to the data phase. The *on flush request* event can be used to cancel pending memory requests. In order to react on a change of the length of burst transfers the *on set burst length* event is provided. With the *on burst end* event the end of a burst transfer can be detected. This is of special importance for memories offering an indefinite burst length. In order to enable individual and independent treatment of memory ports (if there are more than one), event handlers can be bound to a previously defined port.

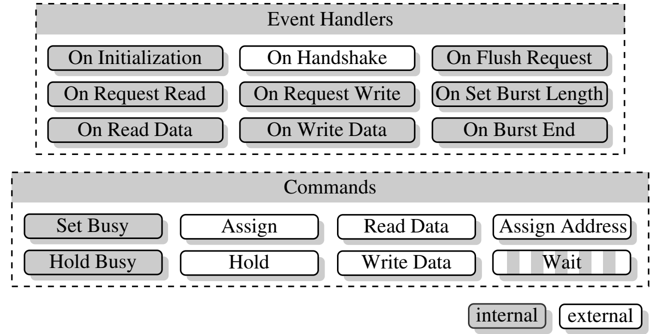

Fig. 3: List of events handlers and commands

External events, in contrast to internal events, need to capture conditions on a lower level, namely reacting on changes of pin values of the external interface. For this purpose only a single event handler is specified, the *on handshake* event. In order to allow for complex conditions, handshakes contain several *handshake criteria* that can be combined by logical conjunction or disjunction. The condition of a *handshake criterium* can be a certain value of a pin or bit range of a pin. Additionally, *handshake criteria* can further contain a list of criteria or even refer to other handshakes enabling the description of complex nested conditions.

Internal commands describe actions to feed back information of the current memory status to the processor core. In most cases the status information can be generated automatically within the memory-interface controller by knowing the exact protocol and therefore timing of the external interface. However, one status information often depends directly on external information from the memory system and cannot be generated automatically: the *busy status*. Especially for dynamic memories encapsulated by a memory controller or other complex memory systems, the status of the memory determines whether new requests can be accepted or not. In order to deal with this issue, two commands are available: *set busy* and *hold busy*. Both types differ in their temporal impact on the value of the internal control signal. The *set busy* command assigns a value for "busy" or "not busy" for a single cycle, while the *hold busy* command modifies its default value which is held until changed by another *hold busy* command.

External commands refer to value assignments to signals of the external interface. The value of an external interface pin can be changed by using the command types *assign* and *hold*, where the *assign* command assigns a specific value to a pin for a single cycle and the *hold* command modifies its default value. The commands *assign address*, *read data* and *write data* trigger more direct interactions between the external interface and the internal interface. While *assign address* triggers the

```
<MemoryInterface ComponentName="SyncMem">
    <ClockPin Name="clk"/>
    <ResetPin Name="rst"/>
    <Pin Name="rw_addr"   Dir="In"   Width="12"/>
    <Pin Name="data_in"   Dir="In"   Width="32"/>
    <Pin Name="data_out"  Dir="Out"  Width="32"/>
    <Pin Name="ew"        Dir="In"   Width="1"   Default="0"/>
    <Port Name="rw_port">
        <AddressBusPin Pin="rw_addr" />
        <DataBusPin    Pin="data_in" />
        <DataBusPin    Pin="data_out"/>
    </Port>
    <OnRequestWrite Port="rw_port">
        <Assign        Pin="ew"   Value="1"/>
        <AssignAddress Port="rw_port"/>
        <WriteData     Port="rw_port"/>
    </OnRequestWrite>
    <OnRequestRead Port="rw_port">
        <AssignAddress Port="rw_port"/>
        <Wait          Cycles="1"    />
        <ReadData      Port="rw_port"/>
    </OnRequestRead>
</MemoryInterface>
```

Fig. 4: MID file example for a single port SSRAM

assignment of the address provided by the processor to the memory, the *read data* and *write data* commands specify the data transfers during the data phase of memory accesses. Due to its similarity to *assign address* and *write data*, the *read data* command is classified as external command although the actual direction of data is inverted.

Wait commands belong to both classes, internal and external commands, as they describe delays by means of number of clock cycles that can affect both sides. Note, that due to the usage of *wait* commands an order within a command list of an event handler is defined.

Due to the hierarchical nature of an MID (especially for the description of handshakes), an XML-based document is used. Approximately 30 lines of XML code are required to describe synchronous pipelined memories, while less than 100 lines are sufficient for more complex memory interfaces, such as those of DDR SDRAM controllers. A minimum basic yet fully functional example of an MID file for a single port SSRAM with one cycle latency is given in Fig. 4.

VI. Automatic Memory Interface Generation

The HDL code generation framework presented in [11] evolves around an intermediate representation, named unified description layer (UDL), which operates on an abstraction level between that of ADLs and RTL. According to common software design techniques the ADL code is parsed via a frontend and RTL code can be produced in different HDLs by dedicated backends. The automatic memory interface generation is integrated in the HDL code generation process via the UDL structures. First, all required information needs to be gathered to build up the structure of the memory-interface controller. On the one hand, the MID file is parsed to generate the *explicit* information about the memory structure. On the other hand, the UDL contains the memory access information from the processor. This information *implicitly* refers to the internal interface and scope of operation of the memory-interface controller. The implicit and explicit information are combined to generate the memory interface intermediate representation (MIIR). During this process, memory accesses are automatically mapped to available memory ports minimizing the number of potential access conflicts. The designer can also influence the mapping by manually forcing certain accesses to be mapped to the same port. The port mapping process includes sanity checks ensuring the integrity of individual accesses, i.e. all phases of an access are present. Finally, the MIIR is mapped

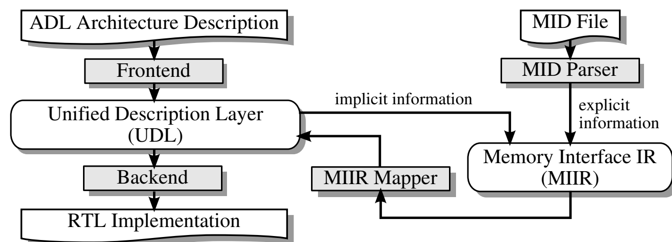

Fig. 5: HDL code generation flow

to components of the UDL and thereby integrated into the existing framework. The whole process is shown in Fig. 5.

A. Mapping MIIR structures to UDL structures

The main component to be instantiated in the UDL, is the memory-interface controller. The targeted structure of this controller is depicted in Fig. 6. The behavioral elements of an MID allow to describe the behavior of the memory interface in a functional way without details about HW structures. Therefore, a direct mapping of the MIIR is not possible. In the following we describe, how MIIR structures are mapped to the interface controller.

The *timing controller* and the *delay registers* build up the actual state machine of the memory-interface controller. The corresponding parts of the MID are described with event handlers and command lists. A command list can contain *wait* commands to implement delay cycles which can be realized in two following ways.

i) *Shift registers*: Shift registers implement a timer by shifting each cycle by a single bit from MSB to LSB. The timer is initialized by setting the MSB to 1. When the 1 reaches a certain position of the register, dedicated actions can be triggered. This implementation allows to pipeline the individual commands of an event handler as delayed commands remain scheduled even if the event is triggered again before their execution. This is of special importance for pipelined memories.

ii) *Counter Registers*: Counters require less register cells and a single incrementer for implementation making them especially attractive for long delays. However, they do not allow pipelined scheduling. For triggering delay-based events, comparators are needed.

In principle, all commands except *wait* can be directly translated to signal assignments on RTL, that are executed conditionally. The resulting if-blocks make up the actual *timing controller* and the *address* and *data forwarding*. In order to observe the actual execution condition of a command, the command list of an event handler is first divided by its *wait* commands into *command collections* with the same execution condition. The conditions refer either directly to the event or

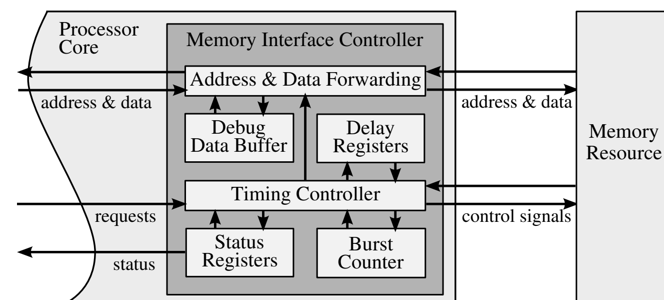

Fig. 6: Memory interface controller implementation

to a check of the counter status in case of delayed commands. Thereby, each command collection is mapped to a single if-block of the *timing controller* description.

Some features supported by the HDL code generation require implementation of additional components, like a *burst counter* for keeping control of burst accesses and *status registers* to propagate status information to the internal interface.

B. Interference with other Processor Features

A special component is the *debug data buffer*, which is instantiated when the processor core is generated with debug mechanism (see [11] for details). This mechanism can cause the whole core to pause execution. As memories typically do not support such a feature directly and continue their operation in this situation, incoming data may need to be buffered in order not to get lost. This example shows another general advantage of *automatic* memory interface generation, as the functionality of other automatic features of the HDL-code generator can be maintained without any manual code modifications.

C. Case Studies

In order to show the flexibility and efficiency of our approach, we extended a simple RISC core with various memory interfaces. Please note, that certain features of an interface (e.g. burst transfer) also need to be reflected by the instruction set of the architecture. Table II shows the required code sizes and area consumption of four different test cases. The first test case utilizes a typical synchronous SRAM with a single read-write port and one cycle latency. For the second test case we connected the DDR SDRAM controller from OpenCores [18] to our architecture. The *"full featured"* test cases describe virtual memory interfaces introduced to test all other supported features of our approach like write mask, initialization phase and burst access. Two versions of this interface are listed, one with five different definite burst lengths and one with indefinite burst length. As can be seen, the size of an MID file is factor 5-10 smaller than the corresponding VHDL implementation of the interface controller. The area consumption for any of the test cases is less than one kGate. Synthesis results have been obtained with Synopsys Design Compiler using a 130 nm standard cell library with a supply voltage of 1.2 V. The maximum frequency for all RISC variants is 450 MHz.

Besides these synthetic case studies a good real-life example for the effectiveness of our approach is the development of an ASIP for Retinex image processing presented in [21]. The target applications imposed strict constraints on the data processing. Therefore, the ASIP required a careful co-exploration of data path, control path and memory interface. The ASIP is designed with 1 program, 2 data memories and a ROM. It uses complex memory-to-memory instructions asserting up to 3 memory accesses in order to speed up data processing. To simplify matters, the development initially started using ideal memories without delay cycles. Afterwards, the switch to real SSRAMs with 1 delay cycle was done. This required the address and data phase for read accesses to be separated and distributed over the pipeline, entailing an extension of the pipeline from 6 to 7 stages. The proposed framework enabled to apply and verify these fundamental architectural modifications in a few hours only whereas a manual implementation on RTL would have required several days.

TABLE II: Code size and area consumption of exemplary memory interface controllers

Interface Type	MID File size (lines)	HDL code size (lines)	Size of controller (gates)
single port SSRAM	31	290	232
DDR SDRAM controller	62	319	291
full featured, def. burst	87	700	824
full featured, indef. burst	67	705	722

VII. Conclusion

In this paper we presented a new abstract and versatile description for memory interfaces covering the definition of memory ports including their pins as well as the timing protocol. It has proven to support a wide range of real world memory interfaces starting form simple synchronous SRAMs up to complex interfaces for DDR SDRAM controllers. Our automatic memory interface generation for HDL code generation is integrated into the LISA ADL development framework [7].

Future work focuses on the identification of additional events and commands in order to describe not only memory but also simple bus interfaces.

References

[1] P. Mishra and N. Dutt, *Processor Description Languages*. Morgan Kaufmann Publishers, 2008.

[2] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor description," *Design Automation for Embedded Systems*, vol. 3, no. 1, pp. 75–108, 1998.

[3] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proc. 34th Design Automation Conf. (DAC 97)*, 1997, pp. 299–302.

[4] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. European Design and Test Conference (ED&TC 95)*, 1995, pp. 503–507.

[5] V. Rajesh and R. Moona, "Processor modeling for hardware software codesign," in *Proc. 12th Int. Conf. VLSI Design (VLSID 99)*, 1999, pp. 132–137.

[6] A. Halambi *et al.*, "EXPRESSION: a language for architecture exploration through compiler/simulator retargetability," in *Proc. Design Automation and Test in Europe (DATE 99)*, 1999, pp. 485–490.

[7] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[8] G. Hadjiyiannis and S. Devadas, "Techniques for accurate performance evaluation in architecture exploration," *IEEE Trans. VLSI Syst.*, vol. 11, no. 4, pp. 601–615, 2003.

[9] S. Basu and R. Moona, "High level synthesis from Sim-nML processor models," in *Proc. 16th Int. Conf. VLSI Design (VLSID 03)*, 2003, pp. 255–260.

[10] P. Mishra, A. Kejariwal, and N. Dutt, "Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models," in *Proc. 14th IEEE Int. Workshop Rapid Systems Prototyping (RSP 03)*, 2003, pp. 226–232.

[11] O. Schliebusch, H. Meyr, and R. Leupers, *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer, 2007.

[12] CoWare Inc., accessed April 2009. http://www.coware.com

[13] Target Compiler Technologies, accessed April 2009. http://www.retarget.com

[14] G. Goossens *et al.*, "Design of ASIPs in multi-processor SoCs using the Chess/Checkers retargetable tool suite," in *Proc. Int. Symp. System-on-Chip (SoC 06)*, 2006, pp. 1–4.

[15] P. Mishra, M. Mamidipaka, and N. Dutt, "Processor-memory coexploration using an architecture description language," *ACM Trans. Embedded Computing Systems*, vol. 3, no. 1, pp. 140–162, 2004.

[16] G. Braun *et al.*, "Processor/memory co-exploration on multiple abstraction levels," in *Proc. Design Automation and Test in Europe (DATE 03)*, 2003.

[17] *JEDEC standard JESD79-3C: DDR3 SDRAM*, JEDEC Solid State Technology Association, Nov. 2008.

[18] *OpenCores DDR SDRAM Controller Core Project*, accessed April 2009. http://www.opencores.org/?do=project&who=ddr_sdr

[19] *DDR SDRAM Controller Core, Product Specification*, Northwest Logic, accessed April 2009. http://www.nwlogic.com/docs/DDR_SDRAM_Controller_Core.pdf

[20] *DDR SDRAM Controller IP Core, Product Specification*, HiTech Global Design & Distribution, LLC, accessed April 2009. http://www.hitechglobal.com/ipcores/ddrsdram.htm

[21] S. Saponara *et al.*, "Algorithmic and architectural design for real-time and power-efficient Retinex image/video processing," *J. Real-Time Image Processing*, vol. 1, no. 4, pp. 267–283, 2007.