Modeling Application-Specific Processors for Embedded Systems

Florian Brandner•, Viktor Pavlu°, and Andreas Krall°

•COMPSYS, LIP, ENS de Lyon
UMR 5668 CNRS – INRIA – UCB Lyon
florian.brandner@ens-lyon.fr

° Institute of Computer Languages
Vienna University of Technology
vpavlu,andi@complang.tuwien.ac.at

Abstract: Embedded systems often have to operate under rigid power and performance constraints. Off-the-shelf processors often cannot meet those requirements, instead *Application-Specific Instruction Processors* (ASIP) are used that are tuned for the particular system at hand.

A popular and powerful way of modeling ASIPs is the use of a *Processor Description Language* (PDL). These languages capture the internal hardware organization as well as the processor's instruction set using a formal specification. Given a processor description, generator tools can (semi-)automatically derive software development tools, instruction set simulators, and even hardware reference models.

An integral part of the software, running on the ASIP, is the interaction with devices outside of the computing platform. However, these external devices are neglected by many PDLs. This is, in part, due to their diverse nature and complex behavior. Explicitly including such devices in processor models, is thus unlikely to give a practical solution.

We propose a basic set of communication patterns for the xADL processor exploration system that allow to interact with external devices, while otherwise treating them as black boxes. The xADL system allows to model three kinds of communication: (1) data exchange using dedicated instructions or memory mapped I/O, (2) asynchronous delivery of data directly into processor registers or memory, and (3) asynchronous signaling using interrupts. A major advantage of our approach is that all side-effects of these interactions are visible to the xADL tool suite. For example, our compiler generator accounts for side-effects during code generation, while the generated simulators reduce simulation time by refactoring the expensive emulation of interrupts.

1 Introduction

Modern embedded and cyber-physical systems have to perform complex and demanding computations, while, at the same time, consuming a minimal amount of power, restricting heat dissipation, and minimizing the physical dimension of the device. Traditional off-the-shelf processors often cannot meet these strict requirements. A very powerful, but inflexible, alternative is the use of specialized hardware components that are tuned for the particular application. These devices often consume a minimal amount of power, while

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

meeting performance constraints. The downside of these hardwired devices is that changes late during the design processes, or worse, after deployment, are exceedingly expensive or even impossible. The use of *Application-Specific Instruction Processors* (ASIP) is an attractive and increasingly popular alternative that combines the benefits of specialized hardware with the programmability of general purpose processors.

A simple form of ASIP can be derived by adding specialized instructions to an existing processor core [Gon00]. The specialized instructions are then used by the programmer to improve the efficiency of crucial algorithms. More sophisticated systems discover a set of beneficial *instruction set extensions* [PPIM03] via application profiling. These systems can also be used to rediscover program fragments that match these instruction extensions in a compiler [MWK+09]. Instruction set extensions are, however, limited by the interface provided by the underlying core processor. In addition, it is rather complicated to provide processor families optionally sharing certain extensions that are applicable to a wider field of applications or system configurations.

Processor Description Languages (PDL) overcome these limitations by providing a formal model of the processor's hardware components and instruction set. These models can be processed and extended by humans and tools alike and thus provide great flexibility during the development of customized ASIPs. PDLs can usually be classified in either of the following categories. *Behavioral* languages allow to model the instruction set of a processor using an abstract specification of instruction semantics. *Structural* languages, on the other hand, focus on the underlying hardware components and their interconnections. While behavioral languages are best suited to *high-level* tasks, such as the automatic customization of a compiler for a particular ASIP, structural languages provide the necessary information for *low-level* tasks, such as the generation of a hardware reference design. A third class of PDLs, so called *mixed* languages, thus combine a behavioral and a structural view in order to cover high-level and low-level tasks equally well.

Most existing PDLs focus exclusively on the processor itself, neglecting external devices. ASIPs intended for use in an embedded system naturally have to deal with external devices, e.g., reading the measurements of a sensor. The inability of existing PDLs to interface to these devices often represents an obstacle in the development and test phase of a new ASIP design. Integrating full specifications of external devices into processor models is unlikely to result in practical solutions, due to the diverse nature of these devices. Alternative solutions that do not overload the PDL, but provide the necessary flexibility, are thus needed.

Instead of including those device specifications in our processor modeling system, we provide flexible primitives in our structural xADL language that allow to interface with external devices. These primitives capture various communication patterns typically found in embedded systems. We distinguish between three forms of communication: (1) explicit data exchange using dedicated instructions or memory mapped I/O, (2) asynchronous communication through direct access to the processor registers and/or memory from external devices, and (3) the signaling of external events using interrupts. We demonstrate that these primitives are flexible and expressive to model the communication with external devices, which are otherwise treated as black boxes by our system. We also show that the information encoded in our processor models can be effectively exploited in our generator

ISBN 978-3-88579-286-4

tools. Our simulator generator, for example, is able to refactor the expensive checks for pending interrupts in order to improve the simulation performance using dynamic binary translation [Bra09b]. Another example is the compiler generator [BEK07, Bra10], which is capable of accounting for potential side-effects of interactions with external devices. An overview of the xADL tool suite and a detailed evaluation is presented in [Bra09a].

The remainder of this paper is organized as follows. Related work on formal processor modeling using processor description languages is given in Section 2. Section 3 introduces the general principles of the structural xADL language that captures the hardware organization of a processor and relies on *instruction set extraction* in order to derive an abstract behavioral model. The following Section 4 presents three basic communication patterns provided by the xADL language in order to interface with external devices. We finally conclude in Section 5.

2 Related Work

One of the most influential processor description languages is *nML* [FPF95], where the processor's instruction set (ISA) is modeled by an attributed grammar. AND-rules in the grammar combine attributes of other rules, while OR-rules allow to compactly enumerate instruction variants. In its latest form, nML includes a basic skeleton defining the internal organization of the processor. nML does not allow to model a wide range of communication and I/O patterns, due to its instruction-centric view.

The *MIMOLA* language [Mar79, Mar84] and its software system (MSS) is one of the few well-known structural processor description languages. It originated from architecture synthesis and micro-programming of the synthesized hardware blocks and is thus well suited to model arbitrary hardware structures – including blocks to communicate and interface with external devices. A complex extraction algorithm provides an instruction set view of the processor and allows to derive an instruction selector for a compiler based on tree pattern matching [LM97, LM98]. The capabilities of this approach are limited, since the input to the extraction algorithms is a general synthesizable hardware specification. Our language, in contrast, is designed to ease the specification of the processor's ISA based on a comprehensible extraction approach and suitable abstractions. Consequently, the derived instruction model is more generic and provides additional information to the generator tools. On the downside, our approach is less flexible in capturing arbitrary hardware structures. We present, in this work, three generic communication patterns that allow our processor models to interact with arbitrary additional hardware structures or external devices, which are otherwise treated as black boxes by our system. This approach gives a good compromise between flexibility and expressiveness, while, at the same time, allows to off-load the specification of external entities outside of our PDL.

Most modern PDLs follow a mixed approach, unifying the benefits of both behavioral and structural models. The *EXPRESSION* language [HGG+99] is a typical mixed processor description language, where a processor model consists of several independent views that separately capture the instruction set, the hardware structure, and the abstract instruction

semantics for retargetable compilation. A major problem of EXPRESSION is that the various views are highly redundant and hard to extend. The software tools, furthermore, cannot automatically verify consistency among the various views of the processor.

A very mature framework has been developed around the *LISA* language [PHvM99]. Instructions are composed of so-called *operations* that provide information on the behavior, the assembly syntax, and the binary encoding. The instruction behavior is described using C, C++, or SystemC, which prohibits high-level applications such as compiler generation but gives great flexibility to model the communication with external devices. Ceng proposed an additional section to model the abstract behavior of instructions for the automatic generation of a compiler [CHL+05].

A core ability of MADL [QM03, QRM04] is the possibility to formally define the processor behavior via *Operation State Machines* (OSM). Interactions among these state machines are controlled by token managers that are not covered by the MADL language and have to be supplied separately, e. g., using C++ code. The interface to the token managers can be used to realize similar communication patterns as proposed here. However, the behavior of the token managers is completely opaque to the MADL generator tools and thus cannot be exploited. Our approach similarly off-loads the complexity of modeling the external devices. Still, the relevant side-effects are visible to the xADL tools, which are able to exploit this additional information, e.g., in order to speed-up simulation.

A recent book by Prabath Mishra and Nikil Dutt provides an excellent introduction to processor description languages and their applications [MD08]. The book covers most of the languages and systems presented here, and many more, in great detail.

3 Processor Modeling using xADL

The xADL language is a structural processor description language based on XML. Its main building blocks are component types and interconnected instances thereof that model the internal organization of the processor. Conceptually, though, the hardware structure is only a means to express the processor's instruction set, which is automatically extracted.

xADL processor models consist of four major parts: a configuration section, meta-information on instructions and programming conventions, a set of component type declarations, and finally, a specification of the processor's data path using component instances. The configuration section provides parameters, such as the bit-width of the data path, the number of functional units or the number of registers, while the meta-information covers the syntax and binary encoding of instructions as well as programming conventions of the application binary interface (ABI), e. g., the stack layout and register usage conventions on function calls. We will not discuss those sections in more detail here. The type declarations provide reusable and extensible definitions of the processor's hardware components. These components are first instantiated from types and then interconnected to compose the data path of the processor. The processor's instruction set is not explicitly specified. We use a comprehensible and flexible extraction algorithm that enumerates all possible instructions that can be implemented using the available hardware resources.

ISBN 978-3-88579-286-4

3.1 Component Types

A fundamental concept of the xADL language are *component types*, which provide blueprints for register files, caches, memories, and functional units. Types specify the basic properties of a given hardware component, such as the number of input and output ports or the component's data width. xADL supports inheritance and type arguments similar to C++ templates to provide additional flexibility for the development of generic, reusable, and extensible specifications.

Immediate Fields are bit-fields embedded in the current instruction word. While typically not represented by dedicated hardware structures, xADL models immediate fields as separate components in a processor description that serve as a data source in the data path.

```
<ImmediateType name="ImmW_t" width="16" />
```

Figure 1: An immediate type of a MIPS processor model.

Register Files are built out of a number of base-registers that all have the same bit-width. The contents of the base-registers are accessible through input and output ports. For each such port, the bit-width and an offset relative to the least significant bit of the base-register can be specified to model sub-registers and register-pairs. The following example depicts a register file definition for a MIPS processor. The size and number of the base-registers is given using two configuration parameters `width_p` and `count_p`. Three separate ports are defined, `Rs`, `Rt`, and `Rd`, which all, by default, inherit the bit-width of the base-registers. The base-register with index zero is immutable and always provides a constant value of zero.

```
<RegisterType name="R_t" width="width_p" repeatcount="count_p" >
 <Constant index="0" value="0" />
 <Port name="Rs" writeable="false" />
 <Port name="Rt" writeable="false" />
 <Port name="Rd" readable="false" />
</RegisterType>
```

Figure 2: Type of the general purpose register file of a MIPS core.

Storage Elements represent data caches and memories accessible to the processor. xADL models memories and caches as black boxes regarding the underlying implementation technology. Both have input and output ports similar to register files, but their internals are not part of the processor description. Instead, annotations provide a timing model and a partitioning of the memorys address space. Caches have additional connections to other caches or memories in order to retrieve data on a cache-miss. Typical cache organizations found in modern processors are captured in predefined cache templates that can also be selected through annotations. An example of a memory type is shown in Section 4.1.

Functional Units are the main building blocks of every xADL processor model. Units represent the computational resources of a processor and thus carry most of the behavioral information used for instruction extraction. The language distinguishes two kinds of functional units: regular functional units that perform computations and containers. Containers provide a means of structuring a processor model by encapsulating parts of the processor's data path to be reused and customized through type arguments and inheritance.

Regular functional units perform computations on behalf of a specific instruction. Their capabilities are captured by a set of operations that the functional unit can perform in a given cycle. The operations are in turn specified using a sequence of micro-operations that represent indivisible calculation steps. The xADL language defines a rich set of predefined micro-operations, including arithmetic and logic operations, comparisons, control operations, and utility functions. Figure 3 depicts the definition of a simplified arithmetic unit from a MIPS processor model. The functional unit has several input and output ports that can be used within the unit's operations in order to read input values or supply values to other functional units. Two operations are defined. A `nor`, which performs a simple `or` of the value supplied by the two input ports `Rs_i` and `Rt_i`. The result of this computation is negated by a `not` that writes the result to the output port `Rd_o`. The `ori` operation similarly performs a logical `or`.

The set of built-in micro-operations is sufficient for the description of most instructions and processors. However, user-defined micro-operations are often useful, e. g., to model the communication with external devices as will be discussed later.

```
<UnitType name="EX_t">
 <Input name="Rs_i" width="32" />
 <Input name="Rt_i" width="32" />
 <Input name="ImmW_i" width="16" />
 <Output name="Rd_o" width="32" default="inactive" />

 <Temporary name="tmp" width="32" />

 <Operation name="nor" >
   <Body>
     <or  d="tmp"  a="Rs_i" b="Rt_i" />
     <not d="Rd_o" a="tmp" />
   </Body>
 </Operation>

 <Operation name="ori" >
   <Body>
     <or  d="Rd_o"  a="Rs_i" b="immW_i" />
   </Body>
 </Operation>
</UnitType>
```

Figure 3: Simplified type of the arithmetic unit of a MIPS processor model.

3.2 Datapath and Pipeline

A processor's data path consists of component instantiations from the previously defined types that are connected through their respective input and output ports using data links. Data links correspond to wires between the ports of two components. Regular data links simply connect two instances within a single pipeline stage, no buffering of the data is performed. In order to model a pipelined processor, data links can be extended using the `stageboundary` keyword which causes a logical register to be placed on the data link. Both forms of data links are grouped into collections, so-called connects. Figure 4 depicts the instantiations of an immediate `ImmW`, a register `R` and a functional unit `EX` from the respective types. The functional unit is connected to the two other instances forming a rather simple data path. Two connects specify data links to the input ports of the unit, either from the register file to the pair `Rs_i` and `Rt_i` or from the register file and an immediate to the pair `Rs_i` and `ImmW_i`. The output port `Rd_o` is always connected to the register file. As this example shows, connects provide a means of connecting components, modeling pipeline stages, and, in addition, specifying groups of legal port assignments.

A complete processor specification also has to deal with potential hazards, i. e., conflicts arising from accesses to common resources or data updates. Structural hazards are resolved automatically using a resource model that is derived from the component instances. To resolve data hazards xADL offers another type of connections called hazard links. Three different data hazard links are provided by the description language. `Forward` links model the bypassing of register values from one pipeline stage to another in order to hide execution latencies. `Stall` links do not carry data, but instead cause the instruction that reads from or writes to the port at the head of the link to wait until the other instruction has completed its operation. The third kind of hazard link, `ignore` links, does not imply any action, but may be used for documentation and verification purposes.

```
<Immediate name="ImmW" type="ImmW_t" />
<Register name="R" type="R_t" />
<Unit name="EX" type="EX_t">
 <Connect>
  <Input input="Rs_i" select="R.Rs" stageboundary="true" />
  <Input input="Rt_i" select="R.Rt" stageboundary="true" />
 </Connect>
 <Connect>
  <Input input="Rs_i" select="R.Rs" stageboundary="true" />
  <Input input="ImmW" select="ImmW" stageboundary="true" />
 </Connect>
 <Connect>
  <Output input="Rd_o" select="R.Rd" stageboundary="true" />
 </Connect>
 <Hazard output="Rd_o" type="forward" select="EX.Rs_i␣EXE.Rt_i" />
</Unit>
```

Figure 4: Instantiation of components and their corresponding interconnections.

Ignoring hazard links, the ports and connects of a processor specification induce an acyclic hyper-graph [BM07] that is further complemented by additional hyper-edges connecting the input and output ports of functional units and storage components. We refer to this hyper-graph as the *processor graph*. It forms the basis of the instruction set extraction that will be presented shortly. Figure 5 shows a graphical representation of a simple processor graph with three functional units, a cache, an immediate, and two register files.

3.3 Signals and Aborts

In contrast to structural and data hazards, control hazards often originate from a small class of instructions that need to directly control the data path and its operation – a typical example of such instructions are branches and jumps. Signals provide a way to communicate asynchronously between the currently active instructions in the pipeline. A signal corresponds to a global single-bit control line in hardware that causes individual instructions in the pipeline to abort.

3.4 Instruction Set Extraction

The instruction set is implicitly contained in the structural xADL processor description and is automatically derived from the corresponding processor graph using an *instruction set extraction* algorithm. The algorithm proceeds in two phases. The first phase enumerates so-called *instruction paths*, which represent connected sub-graphs of the processor graph, where every hyper-edge has at most a single predecessor. An instruction path represents a possible path of execution through the processor's data path for a given instruction. The second phase enumerates all possible instructions that can be formed using the functional units along the instruction paths. At this stage, an instruction is represented by its instruction path and a set of operations, one for each functional unit on the path. In order to derive the final abstract instruction set model, we compute an ordering of the operations of an instruction and concatenate the respective micro-operations to form a linear sequence. The micro-operations are furthermore annotated with information on signals and hazards that could influence the execution of the respective instruction. The annotated linear sequence of micro-operations represents the final behavioral view that is used throughout the xADL

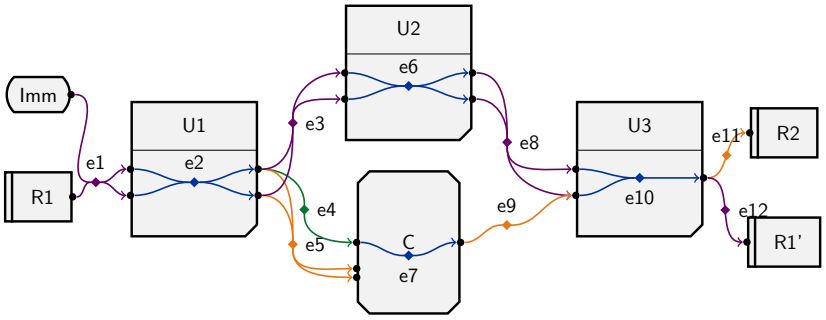

Figure 5: Example of a simple data path represented by a directed hypergraph.

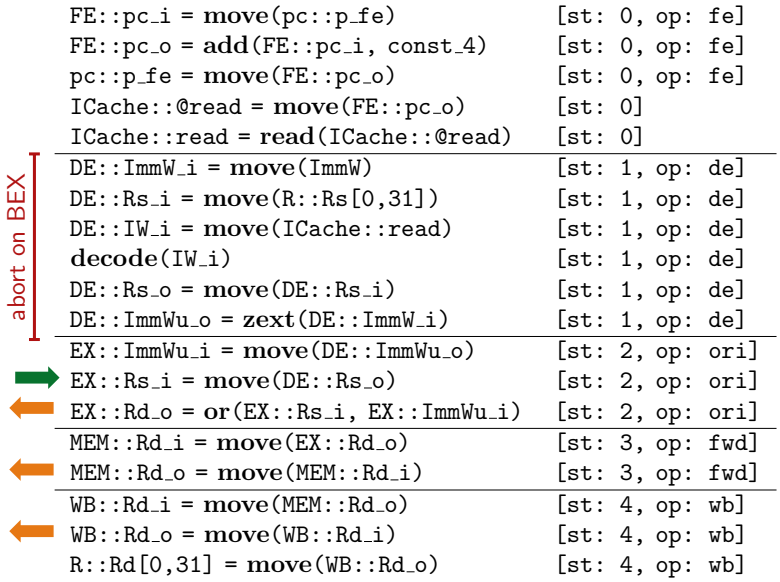

Figure 6: Behavioral model of the *ori* instruction, including annotations for the resolution of control hazards (via the *BEX* signal), register by-passes, hardware resources, and pipeline stages.

tool suite. Since every instruction is associated with an instruction path, we have a tight coupling between the behavioral and structural processor view.

An interesting feature of the xADL language is that the structure of the processor graph is allowed to be very generic. This can be used to define several parallel pipelines, modeling VLIW processors or out-of-order execution. We only impose one restriction on the graph: Instructions have to be fetched and decoded from one location, usually a memory or cache, i. e., the head of all instruction paths, representing the instruction fetch hardware, has to be identical.

Assuming each functional unit of the processor from Figure 5 is associated with exactly one operation $\mathtt{op_i}$. The instruction set extraction first computes five paths: $\{e1, e2, e3, e6, e8, e10, e11\}$, $\{e1, e2, e4, e7, e9, e10, e11\}$, $\{e1, e2, e3, e6, e8, e10, e12\}$, $\{e1, e2, e4, e7, e9, e10, e12\}$, and $\{e1, e2, e5\}$. Each path results in an instruction, e. g., the instruction corresponding to the first path can be modeled as a pair $(\{e1, e2, e3, e6, e8, e10\}, \{\mathtt{op_1}, \mathtt{op_2}, \mathtt{op_3}\})$. Figure 6 shows the final behavioral model of the MIPS *ori* instruction.

4 Interfacing with the Physical World

As for most other PDLs, the primary goal of our xADL language is to provide abstractions and primitives to describe a processor and its instruction set in a compact and concise way. However, ASIPs are usually embedded into a larger system and thus have to interact with the external world. We will present the basic primitives available in the xADL system that allow the modeling of three different communication patterns. The first form of commu-

nication is the explicit sending and receiving of data using dedicated instructions or via memory mapped I/O. Another approach allows to asynchronously read from or deliver data to the internal processor state, e. g., to access internal communication registers within the processor core. We use, so-called, *parallel instructions* to model this asynchronous form of communication by injecting *virtual* instructions into the processor pipeline that update or read the processor's internal state. A specialized form of this asynchronous communication is the signaling of external events that need immediate reaction using interrupts. Interrupts are, however, different since the execution of the current program is redirected to an interrupt handler. Parallel instructions in combination with signals allow to elegantly and concisely express this behavior.

4.1 Data Exchange using Dedicated Instructions

Memory Mapped I/O A very natural means to model communication in an ASIP is the use of dedicated instructions that initiate a data transfer. In the simplest form, we can use the infrastructure of the memory subsystem for this purpose, which is usually referred to as memory mapped I/O. The basic idea is to assign specific regions of the processor's address space to external devices. A read access to such a memory region triggers a bus transaction that is automatically re-routed to the corresponding device. Write accesses are treated in a similar fashion. The xADL language already allows to model the memory infrastructure, including caches and memories, we thus only have to provide additional annotations to designate the address ranges reserved for I/O. In addition, we have to specify how data caches treat data from memory mapped I/O – the usual approach is to bypass the data cache for these accesses. These annotations can later be used by the generator software, for example, in order to account for memory mapped I/O during simulation. The compiler generator similarly has to treat memory accesses to those regions conservatively. It is, for example, not safe to remove seemingly redundant memory accesses that perform I/O. Note that details on how bus transfers are performed are not included in the xADL model, we only capture the timing of transactions in order to avoid overloading the processor specifications. Figure 7 shows an example cache type definition, annotated with information on an `AddressRange`. The annotation defines an address space reserved for a device class `dev`, as well as the minimal and maximal access delays that have to be adhered when communicating with the device.

```
<MemoryType name="IOMemory_t" >
  <Input name="write" datawidth="32" alignment="32" />
  <Output name="read" datawidth="32" alignment="32" />

  <AddressRange offset="0x1000" length="16" class="dev"
                min-delay="100" max-delay="120" />
</MemoryType>
```

Figure 7: Excerpt of the data memory definition of a MIPS processor.

I/O Operations Bus transactions are sometimes too time consuming for certain communication tasks. It is thus desirable to provide dedicated I/O instructions that communicate directly with the device at hand. This can be specified through *user-defined micro-operations* in the xADL language. These micro-operations serve as black boxes within the processor model that represent the communication with the device. Details how this communication is carried out are, again, not included in the xADL language itself. The micro-operations are thus conservatively treated as black boxes by most generator tools. The compiler generator, for example, treats user-defined micro-operations conservatively and merely provides so-called *intrinsic* functions that permit the use of the involved instructions from high-level languages, such as C or C++. For the simulator generator it is of course not an option to treat those operations as black boxes. It thus requires semantic models in the form of C or C++ code for user-defined micro-operations. It is important to note that user-defined micro-operations are rather rare, therefore we believe that this approach represents an acceptable trade-off between the flexibility and simplicity.

A user-defined micro-operation can be defined using the `Function` keyword and a set of input and output arguments, as depicted by Figure 8. Once defined, a user-defined micro-operation can be invoked, just like any other micro-operation, using the `Call` keyword within the body of an operation in a unit type. The example defines a simple interface to send data, represented by the input argument `payload`, to an external device, which supplies an immediate response through the output argument `acknowledge`. It is important to note that the definition of the micro-operation is polymorphic with regard to the input and output arguments, i. e., the respective data representation depends on the context of the calling operation. Additional timing annotations can be specified in order to capture devices having longer communication delays.

```
<Function name="send_data">
  <Input name="payload" />
  <Output name="acknowledge" />
</Function>
```

Figure 8: Example of a user-defined micro-operation.

4.2 Asynchronous Data Exchange

Another communication pattern allows to model asynchronous delivery of data directly to a register or memory of the processor core using *parallel instructions*. Parallel instructions are special “instructions” that are, in contrast to regular instructions, not fetched from memory. They are implicitly executed on every cycle, independently from the program running on the ASIP, by dedicated functional units. During their execution, parallel instructions can read and write registers and even access memory and caches of the processor core.[1] They thus lend themselves to specify asynchronous data delivery and/or provide read accesses to the internal processor state.

[1] Immediate and variable register operands are not possible, since they cannot be fetched from memory.

The major advantage of this approach is that all data exchanges are explicitly contained in the processor specification, i. e., values do not appear out of nowhere but instead originate from a parallel instruction. This provides valuable information to the generator software. The compiler generator, for instance, can account for side-effects that may arise from asynchronous data exchange. Also, the simulator generator can exploit the additional information in order to reduce simulation overhead. The interface to the device actually delivering the data is, as before, not explicitly specified in the xADL processor model itself – the previously presented user-defined micro-operations are, again, well suited for this purpose.

Parallel instructions do not require any extension to the syntax of the xADL language, in contrast to the communication patterns based on memory mapped I/O and dedicated I/O instructions. However, parallel instructions do not satisfy the definition of an instruction from Section 3.4. Instead of extending the xADL language, we thus extended the instruction set extraction algorithm such that parallel instructions are recognized and accepted. Figure 9 shows an extended version of the processor graph from Section 3.2, defining a simple parallel instruction using a new functional unit `PU`. The instruction reads from a new register port of `R1` and writes to the existing input port of the same register file.

Figure 9: Example of the simple data path from Figure 5 extended by a parallel instruction.

4.3 Signaling Asynchronous Events

The final communication pattern does not directly deal with data exchange, but rather provides a means to signal asynchronous events that need immediate attention, e. g., an interrupt signaling that data is available to be read from a sensor. As previously, for the modeling of asynchronous data exchange, interrupts are specified via parallel instructions. In contrast to data exchanges, interrupts cause the currently running program to be interrupted by redirecting the execution to an interrupt handler. This usually implies a disruption of the regular flow of instructions through the processor pipeline and the resetting of the processor's program counter. The former is represented by signals that are triggered by the parallel instruction modeling an interrupt dispatch. The signal then causes the in-

structions within the pipeline to abort before their execution is completed. At the same time, the program counter is stored, in order to resume the execution after the interrupt, and subsequently overwritten by the address of the interrupt handler. An interrupt dispatch is characterized by these very specific operations and can thus be easily detected by the xADL tools. It is furthermore possible to verify that interrupts are correctly modeled in the processor specification, e. g., by ensuring that no permanent side-effects are caused when an interrupt is triggered. As before, the explicit representation of interrupts and interrupt-like parallel instructions can be leveraged to improve the various generator tools. In particular, simulation performance can be drastically improved.

4.4 Time Synchronization

An important aspect that has to be accounted for when interfacing with external devices is timing. The xADL language specifies a static skeleton of the processor's data path, where timing is only expressed implicitly using pipeline stages and certain annotations describing the timing of memory and bus transactions. The interaction with external devices is thus implicitly synchronized using a global time model based on cycle counters. This is particularly important for the simulator generator that heavily utilizes global time tags to track when data is available. During a simulation run all external devices have to follow this timing scheme in order to guarantee correct synchronization between the simulation of the respective devices and the processor core.

4.5 Preliminary Results

The communication patterns described in the previous sections allow a wide range of devices to interface with an ASIP specified in the xADL language. We were able to leverage existing primitives, such as user-defined micro-operations and parallel instructions in order to implement those patterns, which are already supported by the existing tool suite. Even though interrupts are described using existing primitives they need special care. The problem stems from the redirection of the currently running program to the interrupt handler. All xADL tools have to conservatively assume that the behavior of every instruction could potentially be modified as a side-effect of an interrupt dispatch. This poses a severe problem, in particular, for compiler generation, which heavily relies on predictable semantics of the instruction set model.

Our simulation framework relies on dynamic binary translation and thus faces a similar problem. Interrupts modify the behavior of every instruction and, in addition, can be triggered at any moment in time. This reduces the gains of dynamic binary translation, because code has to be generated that is able to simulate every possible outcome of the simulation. This increases code size and compilation time considerably. Another negative side-effect is that code optimizations that often improve simulation time considerably cannot be applied effectively, since intermediate states that usually could be eliminated

ISBN 978-3-88579-286-4

need to be retained for the, rare, event of an interrupt. We have extended our simulation framework to account for the special semantics of interrupts using a generic rollback mechanism [Bra09b]. Our system optimistically assumes that interrupts will not occur when the simulation is performed using binary translated code. In the rare event of an interrupt the simulation is aborted and reverted to a previously established safe point using a rollback. Simulation then resumes using a slower interpreter that faithfully captures interrupts. Similar optimization techniques are also applicable to the simulation of memory mapped I/O, e. g., to reduce the simulation overhead for memory accesses where the exact memory address is unknown. Our results indicate that compilation time is considerably reduced by about 30% and that simulation speed is reduced by up to a factor of 3 for a set of embedded benchmarks from the MiBench suite.

5 Conclusion

During the development of a new ASIP design for an embedded system, it is important to account for external devices, such as sensors and actuators, that will interact with the software running on the ASIP. Traditional PDLs neglect this problem to a large extent and are thus not suited to model ASIPs in this context.

We have shown how existing primitives of our xADL language can be used to express three communication patterns that are frequently found in embedded systems. While it is not practical to specify the exact behavior of these external devices within a PDL, our approach is able to capture the relevant side-effects of data exchange operations on the processor state in a concise and compact manner. A major advantage of our approach is that all generator tools, in particular the compiler and simulator generator, can leverage this information in order to improve the generated tools.

References

[BEK07] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *CASES '07: Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 13–22, 2007.

[BM07] John Adrian Bondy and U. S. R. Murty. *Graduate texts in mathematics - Graph theory*, volume 244. Springer, 2007.

[Bra09a] Florian Brandner. Automatic Tool Generation from Structural Processor Descriptions. In *KPS '09: Workshop on Programmiersprachen und Grundlagen der Programmierung*, 2009.

[Bra09b] Florian Brandner. Precise simulation of interrupts using a rollback mechanism. In *SCOPES '09: Workshop on Software and Compilers for Embedded Systems*, pages 71–80, 2009.

[Bra10] Florian Brandner. Completeness of Automatically Generated Instruction Selectors. In *ASAP '10: Conference on Application-specific Systems, Architectures and Processors*, pages 175–182. IEEE, 2010.

[CHL+05] Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. C Compiler Retargeting Based on Instruction Semantics Models. In *DATE '05: Conference on Design, Automation and Test in Europe*, pages 1150–1155, 2005.

[FPF95] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nML. In *EDTC '95: European Conference on Design and Test*, pages 503–507, 1995.

[Gon00] Ricardo E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, 2000.

[HGG+99] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Conference on Design, Automation and Test in Europe*, pages 485–490, 1999.

[LM97] Rainer Leupers and Peter Marwedel. Retargetable Generation of Code Selectors from HDL Processor Models. In *EDTC '97: European Conference on Design and Test*, pages 140–144, 1997.

[LM98] Rainer Leupers and Peter Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. In *Design Automation for Embedded Systems*, pages 1–36. Kluwer Academic Publishers, 1998.

[Mar79] Peter Marwedel. The MIMOLA design system: Detailed description of the software system. In *DAC '79: Design Automation Conference*, pages 59–63. IEEE, 1979.

[Mar84] Peter Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *DAC '84: Conference on Design automation*, pages 587–593, 1984.

[MD08] Prabhat Mishra and Nikil Dutt. *Processor Description Languages*, volume 1. Morgan Kaufmann, 2008.

[MWK+09] Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch, and Francois Charot. Constraint-Driven Instructions Selection and Application Scheduling in the DURASE system. In *Conference on Application-specific Systems, Architectures and Processors*, ASAP '09, pages 145–152. IEEE, 2009.

[PHvM99] Stefan Pees, Andreas Hoffmann, Vojin Živojnović, and Heinrich Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *DAC '99: Conference on Design Automation*, pages 933–938, 1999.

[PPIM03] Armita Peymandoust, Laura Pozzi, Paolo Ienne, and Giovanni De Micheli. Automatic Instruction Set Extension and Utilization for Embedded Processors. In *Conference on Application-specific Systems, Architectures and Processors*, ASAP '03, pages 108–118. IEEE, 2003.

[QM03] Wei Qin and Sharad Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. In *DATE '03: Conference on Design, Automation and Test in Europe*, pages 556–561. IEEE, 2003.

[QRM04] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A Formal Concurrency Model based Architecture Description Language for Synthesis of Software Development Tools. In *LCTES '04: Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 47–56. ACM, 2004.