A SystemC Register Model for Multiple Levels of Abstraction Using Advanced Object-Oriented Design Patterns

Lillian Tadros

***Abstract*—Available SystemC IP blocks are commonly modeled at either a *cycle-accurate* or a *functional* abstraction level. The large interval between these two choices, combined with non-trivial model replacement and integration effort, often leads to a prolonged use of the functional model, resulting in a verification gap once the model is eventually replaced with its cycle-accurate counterpart. A finer control of the granularity of both the adopted abstraction level and, even more important, the scope of the chosen abstraction, can thus enable localized, detailed analysis or verification with minimal impact on the simulation speed. This paper presents a model for processor registers in SystemC that makes use of several object-oriented design patterns in order to transparently change its behavior, starting from a functional model down to a cycle-accurate, pipelined version, while preserving the same interface. The register model has been integrated into a fork of the open-source processor generation tool TRAP-Gen, which generates SystemC processor models from high-level, Python-based descriptions.**

***Index Terms*—Abstraction level, processor generation, registers, systemC.**

I. Introduction

The rising complexity of Systems-on-Chips, combined with a persevering time-to-market pressure, has rendered virtual prototyping almost indispensable for early design space exploration, verification and parallel HW/SW, co-design [1]. Ideally, a virtual prototype should persist throughout most of the design flow, evolving from a functional model down to an implementation with as little superfluous effort as possible. In practice, however, system design is seldom a neat, straight-forward process. Instead, it often involves different groups working at a different pace, re-use of legacy IP and/or adoption of external IP [2]. This results in top-level models containing a mixture of tools, modeling styles, interfaces and abstractions with correspondingly significant integration demands. Thus, the longer a given model can perdure within the design cycle with minimal changes, the greater the savings in integration and verification efforts [3]-[5].

Nonetheless, state-of-the-art EDA tools, as well as off-the-shelf IP, are mostly geared towards an approach involving model refinement via *replacement* as opposed to *enhancement* [6]: Both handwritten as well as tool-generated models tend to offer either a full, cycle-accurate implementation and/or a high-level, functional view. This has several repercussions: First, there is the aforementioned model replacement effort whenever the abstraction level is lowered as the design materializes. Second, there is no possibility of changing the abstraction scope for selective, localized analysis or verification of a given, sub-modular hardware block. Third, the slow-down in simulation speed that accompanies accurate models [7], [8] might very well induce a prolonged application of the functional model, necessarily accompanied by the discovery of hitherto unknown errors once it is eventually replaced by a detailed implementation.

These shortcomings were perceptible during our work integrating new processors into the SystemC/TLM virtual platform SoCRocket [9], [10]. Our processor models were generated using the open-source processor generation tool TRAP-Gen (TRansactional Automatic Processor GENerator) [11] that generates SystemC models from a common, high-level Python description. Separate models are generated of either cycle-accurate (i.e. pipelined) or functional *behavior* combined with approximately-timed or loosely-timed *communication*. During the verification phase, we found we would have benefited from closer introspection, specifically of the processor registers, but were held back from utilizing the pipelined model due to code-size, compilation- as well as run-time considerations.

This led us to undertake a complete rewrite of the register generation methodology used in TRAP-Gen. Our approach envisioned a model for processor registers that can be parametrized in runtime to provide behavior varying in detail from cycle-accurate/pipelined up to un-timed/functional. In a sense, this does for intra-module behavior what is done by TLM sockets [12] for inter-module communication.

This paper is organized as follows: Section II evaluates some available processor models or processor-generating tools. We describe the premises and general concepts of our model in Section III. The multi-abstraction feature is introduced in Section IV. Section V presents some experimental evaluation. We conclude with some open aspects for future consideration in Section VI.

II. Related Work

To the best of our knowledge, there is no published work dedicated to dynamically varying the internals of a model to operate at a user-defined abstraction level. Nevertheless, it is interesting to compare the underlying methodology behind

Manuscript received January 30, 2017; revised June 25, 2017. This work was supported by the ARTEMIS-JU EMC2 project, grant no. 621429.

L. Tadros is with the Technische Universit ät Dortmund, 44227 Dortmund, Germany (e-mail: lillian.tadros@tu-dortmund.de).

processor models both in commercial and open-source IP.

Since we have no access to commercial products, we can only offer a cursory depiction thereof. ARM processor models are available from the former Carbon Design Systems, now part of ARM Ltd. [13]. They offer both a cycle-accurate Register Transfer Level (RTL) model and an instruction-accurate model that integrate with their SoC Designer Plus virtual prototyping environment. A similar two-level approach is taken by Synopsys' DesignWare, in this case the instruction-accurate nSIM and the cycle-accurate xCAM [14]. Cadence's Tensilica processors come in four flavors: Xtensa ISS / TuboXim for hardware developers and XTensa SystemC / XTensa Modeling Protocol for software developers [15]. The generation tool from Cadence [16] follows a similar division between abstractions. Mentor Graphics offers its microcontrollers in Verilog or VHDL RTL [17]. Processor models from Imperas [18] or the popular open-source QEMU project [19] are geared towards software development and thus offer a purely functional virtual machine interface.

The common factor in all preceding models is the utilization of different models of the same processor core, depending on use-case and target audience. Open-source models are no different: OpenCores [20] is the conglomeration of many processor models from both industry and academia, with a corresponding multitude of modeling languages, detail levels and styles. SoCLib [21] attempts to reduce the number of core models to less than the product of cores and abstractions by providing one abstraction-agnostic model per instruction set plus retargetable wrappers for the communication timing, in their case cycle-accurate, approximately-timed and untimed [22]. While this handles communication timing, the behavior timing is only approximated using annotation. Registers are modeled as integers for all abstractions, pipelining and similar implementation-dependent effects are not considered. Along the same lines, gem5 [23] offers four generic processor models of different behavioral granularities, to be combined with a given instruction set model in a mix-and-match fashion [24]. While this successfully reduces the total number of core models, the problem of integrating, maintaining and verifying multiple, in this case four, models of the same processor remains.

III. Requirements and Proposed Register Class Hierarchy

A. Requirements

Depending on the abstraction level, the behavioral model of a processor register can range from a simple value to a complex set of latches that propagate on clock cycle edges. After analyzing several instruction set architectures, we identified the following requirements for a register implementation:

1) *Modeled behavior*

a) Ability to access and manipulate named register fields (named bit sequences within a register).

b) Ability to define and manipulate register banks.

c) Ability to define, possibly variable, offsets to be added to register values (e.g. the +4/+8 PC (program counter) offset in ARMv7 [25]).

d) Ability to define a time delay between register writes and subsequent reads, introducing a notion of time for un-pipelined registers. This can be considered a loosely-timed behavioral modeling style, in a similar vein to loosely-timed TLM communication.

e) Ability to define register aliases, with or without offsets, to enable virtualization of the available physical registers (e.g. SPARC9 [26]).

2) *User experience and flexibility*

f) Choice of abstraction level as a run-time parameter. The set of available abstraction levels should be easily expandable.

g) Powerful and semi-uniform interface to the register hierarchy (registers, register fields, register banks).

h) Ability to attach callback functions to read and write events.

3) *Efficiency*

i) Run-time overhead reduction (number of function calls).

j) Compile-time overhead reduction (code size).

Many well-established design patterns found their way into our model [27]. In the rest of this section, we present our proposed model along with some design considerations.

B. Composite Interface: Register Field, Register and Register Bank

The hierarchy of registers, register banks and register fields (requirements 1a, 1b and 2b) naturally leads to a set of composite classes. We start with the setup depicted in Fig. 1.

There are several interesting points to note in this design:

1) *RegInterface*

This class defines both the value access and manipulation as well as the child management interface. It includes common operations that can be performed by all register classes (read, write, arithmetic, bitwise, relational, logical and debug operations).

The CHILD template parameter is used by the classes deriving from RegInterface for specifying the type of child they are composed of. Since our hierarchy only allows Bit ϵ RegField ϵ Register ϵ RegBank, a generic solution is consequently neither needed nor even sound. Hence, the derived classes store and manipulate the derived type directly, instead of a RegInterface, avoiding costly and unnecessary dynamic casting.

The child-parent access and storage mechanism is discussed separately for each of the three derived classes.

2) *Register*

This is a heavy-weight class that contains the full implementation of the access and manipulation interface (read/write, operators, etc.). All operators call either read() or write() instead of directly manipulating values. Since read and write are inlined, no runtime cost is incurred. The advantage of this approach will be obvious when discussing different abstractions in Section IV.

On initialization, the register sets up a - possibly empty - container of RegFields. Fields can be accessed using array notation (operator []). Named field access is achieved using the C++ enum class construct.

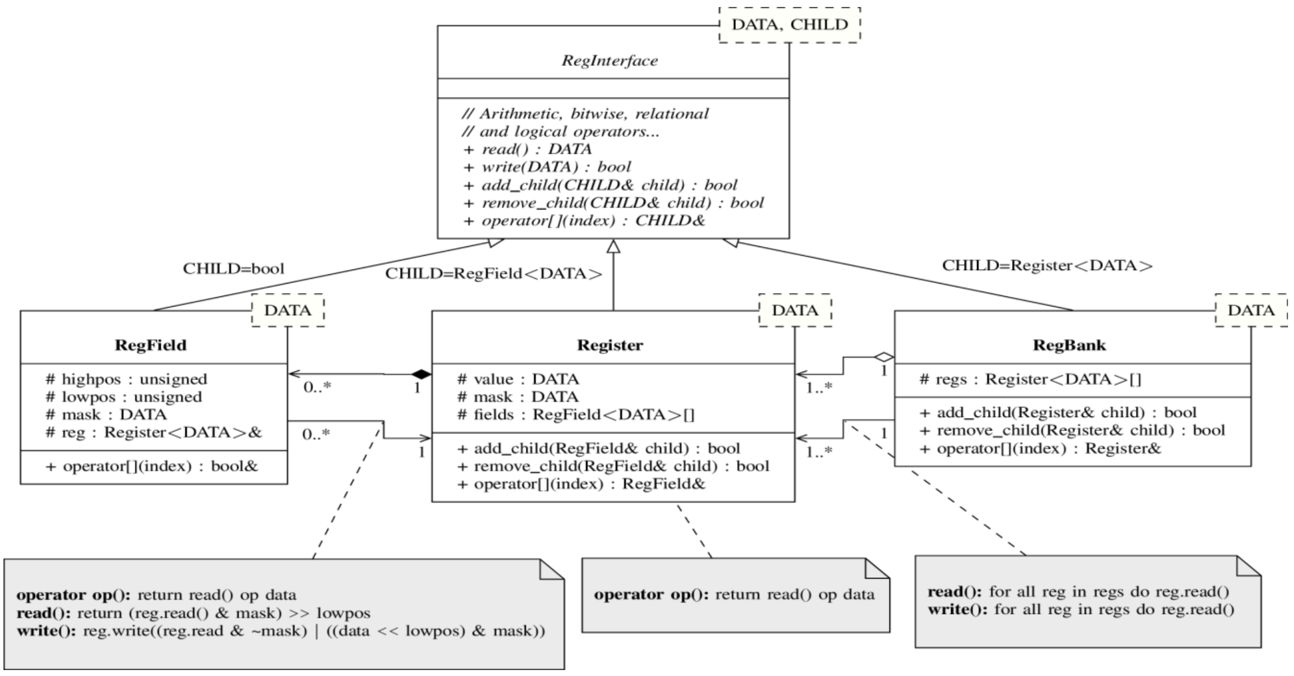
DATA, CHILD
RegInterface
// Arithmetic, bitwise, relational
// and logical operators...
+ read() : DATA
+ write(DATA) : bool
+ add_child(CHILD& child) : bool
+ remove_child(CHILD& child) : bool
+ operator[](index) : CHILD&
CHILD=bool
CHILD=RegField<DATA>
CHILD=Register<DATA>
DATA
RegField
# highpos : unsigned
# lowpos : unsigned
# mask : DATA
# reg : Register<DATA>&
+ operator[](index) : bool&
DATA
Register
# value : DATA
# mask : DATA
# fields : RegField<DATA>[]
+ add_child(RegField& child) : bool
+ remove_child(RegField& child) : bool
+ operator[](index) : RegField&
DATA
RegBank
# regs : Register<DATA>[]
+ add_child(Register& child) : bool
+ remove_child(Register& child) : bool
+ operator[](index) : Register&
0..*
1
0..*
1
1..*
1
1..*
1
operator op(): return read() op data
read(): return (reg.read() & mask) >> lowpos
write(): reg.write((reg.read & ~mask) | ((data << lowpos) & mask))
operator op(): return read() op data
read(): for all reg in regs do reg.read()
write(): for all reg in regs do reg.read()

Fig. 1. Register field, register and register bank hierarchy.

3) *RegField*

A field stores a reference to a register object. It does not directly manipulate the register value, but delegates all accesses to the read() and write() operations of the parent register, after some bit manipulations and access checks. This makes this class very light-weight.

The child management of fields is not immediately obvious. We chose to define a bit as the "child" of a field. Adding or removing bits, as declared in RegInterface, obviously makes little sense, but accessing bits in the form field [bit] is a consistent interpretation of the access operator. As with all other access functions, the bit value is not saved in the field but read on demand from the parent register.

4) *RegBank*

Encapsulating registers in one place provides two advantages: First, it greatly simplifies passing registers around the processor modules (individual instructions, ABI, tests, etc.). Second, it provides a unified interface, so that the processor or some test function can, for example, conveniently call RegBank::reset() or RegBank::write() with no further knowledge of the names, number of properties of the stored registers. The advantages of this encapsulation will be more apparent upon discussing register aliases in section III-E.

The relationship between a register bank and the container register is a weak aggregation, as opposed to the stronger composition existing between registers and fields. Accordingly, registers can exist independently of banks and manage their own creation and destruction.

5) *Iterator interface*

Further tweaking the previous design, we add a robust child iteration interface in addition to the child access operations provided by RegInterface. The RegIterator class in Fig. 2 provides basic sequential access with bounds-checking. It saves a pointer to the iterable container, i.e. RegField, Register or RegBank, and the index of the current iterator position. As with RegInterface, the CHILD template parameter enables specifying the underlying node type for more efficient access. Most of the well-known iterator manipulations from the standard C++ library are provided. The iterator class relies on RegInterface::operator[] for accessing children.

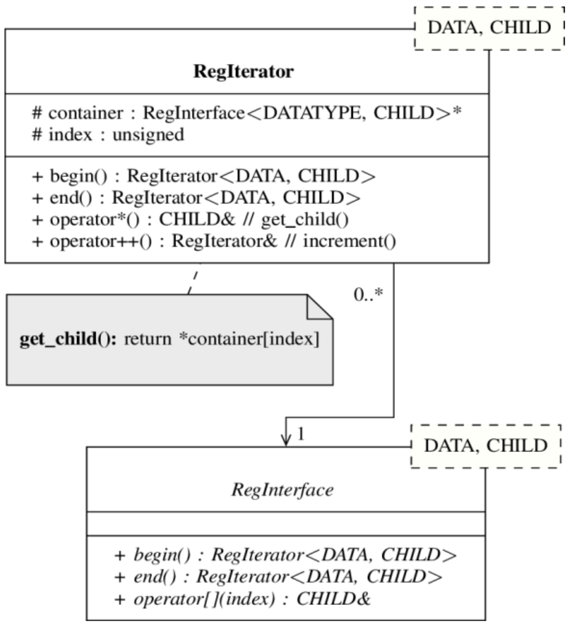
DATA, CHILD
RegIterator
# container : RegInterface<DATATYPE, CHILD>*
# index : unsigned
+ begin() : RegIterator<DATA, CHILD>
+ end() : RegIterator<DATA, CHILD>
+ operator*() : CHILD& // get_child()
+ operator++() : RegIterator& // increment()
get_child(): return *container[index]
0..*
1
DATA, CHILD
RegInterface
+ begin() : RegIterator<DATA, CHILD>
+ end() : RegIterator<DATA, CHILD>
+ operator[](index) : CHILD&

Fig. 2. Register iterator.

C. *Observer Interface: RegCallback*

Requirement 2c foresees hooking user-defined functions, i.e. *callbacks*, to read and write operations on a register interface class. We adopt the scireg interface as provided by Cadence [28], which already defines a notion for callbacks. The interface foresees an abstract callback type that supports a do_callback() method, as in a classic observer pattern. Each callback stores a type, specifying whether to be called on a read, write or state change. The offset and size attributes allow attaching callbacks to specific register fields. As with all preceding constructs, both fields and banks delegate callback execution to registers. The register class stores a callback container and provides methods for adding, removing and executing callbacks. This is illustrated in Fig. 3.

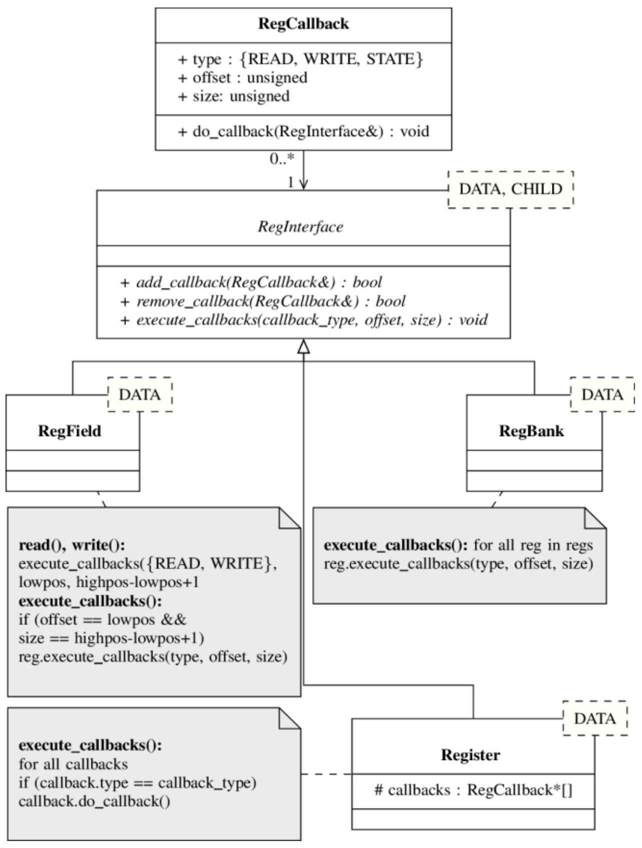
RegCallback
+ type : {READ, WRITE, STATE}
+ offset : unsigned
+ size: unsigned
+ do_callback(RegInterface&) : void
0..*
1
DATA, CHILD
RegInterface
+ add_callback(RegCallback&) : bool
+ remove_callback(RegCallback&) : bool
+ execute_callbacks(callback_type, offset, size) : void
DATA
RegField
DATA
RegBank
read(), write():
execute_callbacks({READ, WRITE}, lowpos, highpos-lowpos+1
execute_callbacks():
if (offset == lowpos &&
size == highpos-lowpos+1)
reg.execute_callbacks(type, offset, size)
execute_callbacks(): for all reg in regs
reg.execute_callbacks(type, offset, size)
execute_callbacks():
for all callbacks
if (callback.type == callback_type)
callback.do_callback()
DATA
Register
# callbacks : RegCallback*[]

Fig. 3. Register callbacks[1].

D. Decorator Interface: Register Alias

The concept of a *register alias* is warranted by instruction sets that distinguish between the set of available *physical* registers and the set of logical registers that are addressable in a given configuration, the latter being a subset of the former. An intuitive solution implementing aliases as pointers or references to registers is not viable for several reasons:

1) Defining an alias as a simple register pointer (Register*) requires that the user remembers to use a different syntax for registers and aliases, which is hardly feasible.
2) References (Register&) are transparent to the user, but have the disadvantage that they are not meant to be re-assignable, which retracts the gain in having aliases in the first place. Furthermore, many constructs, such as arrays, are not possible with references as subjects.
3) As identified in requirement 1e, an alias can possibly be prescribed to return a fixed offset to the register value. This is obviously not realizable with plain-old-data.

Thus, we augment our class hierarchy by a RegAlias class that retains the same interface as the underlying register. Along the same lines as the field and bank classes, the alias class also delegates everything to the underlying register, as shown in Fig. 4. An optional offset is added dynamically whenever the register value is read, whether due to an external request or for further processing[2].

In addition to the base register, an alias stores optional pointers to predecessor and successor aliases that enable building an arbitrarily long alias chain. The offset is defined relative to the predecessor alias for convenience. In practice, however, all value manipulations are performed directly on the register. The list of aliases is required for correct offset calculations whenever an alias is relocated to point to a different register or to another alias using set_alias(). An example for offset calculations is illustrated in Fig. 5.

The addition of aliases requires adding an alias container to RegBank, in addition to the available register container. This further accentuates the transparency attained by the register bank construct. The possible dynamic overhead of searching two containers at every register access is mitigated in the case of generated models, which is our original intent, since it becomes possible to hard-code the individual register and alias names, whereby lookup is reduced to a simple member access.

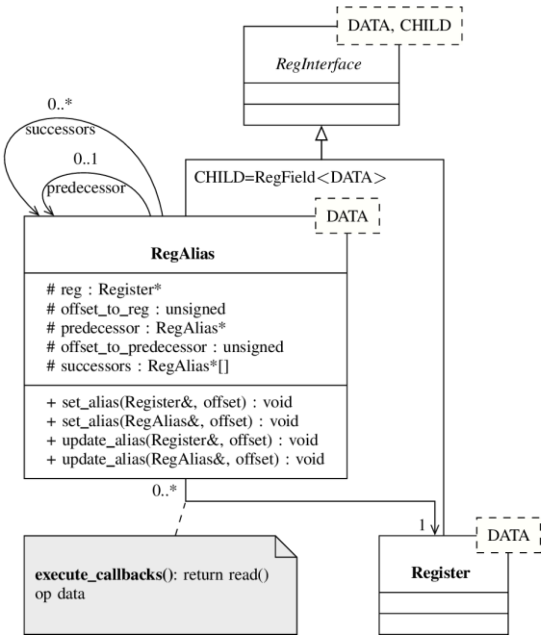
DATA, CHILD
RegInterface
0..*
successors
0..1
predecessor
CHILD=RegField<DATA>
DATA
RegAlias
# reg : Register*
# offset_to_reg : unsigned
# predecessor : RegAlias*
# offset_to_predecessor : unsigned
# successors : RegAlias*[]
+ set_alias(Register&, offset) : void
+ set_alias(RegAlias&, offset) : void
+ update_alias(Register&, offset) : void
+ update_alias(RegAlias&, offset) : void
0..*
1
DATA
execute_callbacks(): return read() op data
Register

Fig. 4. Register alias class diagram.

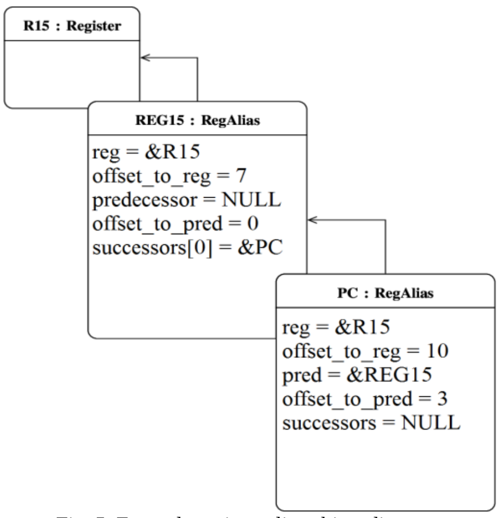

Fig. 5. Example register alias object diagram.

[1]The scireg interface depicted here is abbreviated and modified for clarity.

[2] This interpretation, though sound, causes interesting mathematical properties, such as (alias = x) != x, which may confuse the unwary.

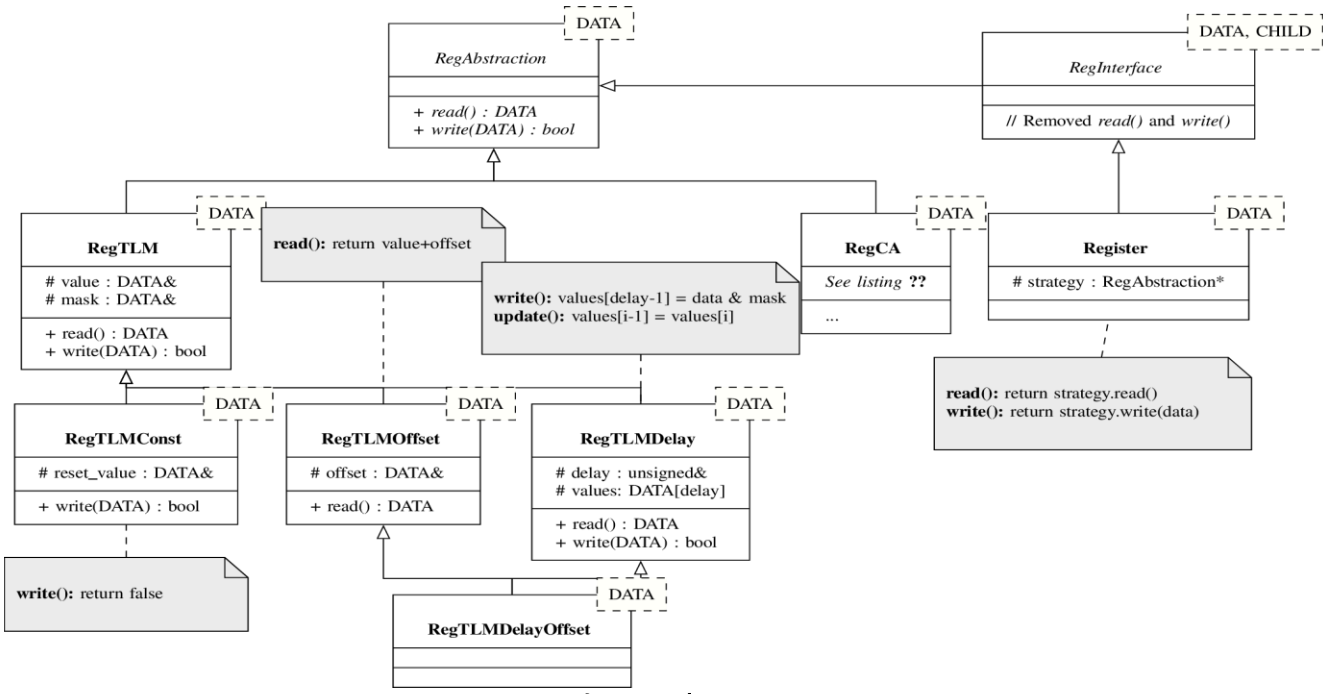

Fig. 6. Register abstractions.

IV. Implementing Behavioral Abstractions

A. Strategy Interface: Register Abstraction

We now elaborate on the concepts already described to add support for multiple behavioral abstraction levels. The attention given so far to redirecting the interface to the Register::read() and write() methods now pays off: It suffices to relocate both methods to a class that supplies a strategy appropriate for the chosen abstraction level. On a read or write, the register simply delegates the behavior to the chosen strategy class. The heavy-weight register class can thus be used for all abstractions by just varying the strategy (requirement 2a).

Fig. 6 depicts an abstract RegAbstraction class, now responsible for the read/write interface.

RegTLM [3] provides an implementation for un-timed registers. Reads and writes are simply applied to the register value, to which the strategy class has access.

Although RegTLMDelay, RegTLMOffset and RegTLMDelayOffset technically derive from RegTLM, they are actually loosely-timed models. They fulfill requirements 1c and 1d by introducing a notion of time via a time-delay or a value-offset, respectively. The former buffers as many values as the given delay in clock cycles. Every clock cycle, an update function advances the array one step. The latter is mostly needed by PC registers to mimic the value increment in different pipeline stages.

The fully-pipelined RegCA model requires some elaboration and is deferred to Section IV-B.

An interesting consequence of strategies is that we can expand on the idea to handle the behavior of special registers.

[3]The name "TLM" is perhaps somewhat misleading, as it is usually applied to inter-module communication. "Functional" would perhaps be more appropriate. We still adopt it, since TLM communication is usually combined with un-timed/loosely-timed behavior.

Constant-value registers, for instance, have different write-logic from the regular case. This makes for a second level of delegation, where an abstraction delegates to a more specialized implementation. This is illustrated in the RegTLMConst class in Fig. 6.

B. Pipelined Register Model

Given that as good as every modern processor is pipelined, a cycle-accurate register model has to consider the manifold effects of pipelining on register values. A typical processor pipeline is illustrated in Fig. 7. Each pipeline stage manipulates a latched version of a given register which propagates one stage forward on clock cycle edges. The value written in the *write-back* stage is fed back to the first stage.

A modern pipeline that supports *forwarding* also feeds certain latches back to preceding pipeline stages to prevent the manipulation of stale values. This is for example the case for the PC register in the decode and write-back stages of the LEON3 processor [29].

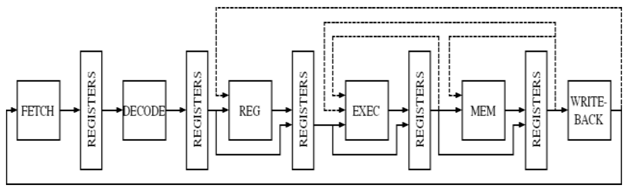

Fig. 7. Example of pipelined registers, inspired by [30]. The dashed lines are possible forwarding paths.

Our TRAP-Gen-generated instruction set implementation integrates behavioral blocks provided by the user per instruction and pipeline stage. For non-pipelined models, the behavior is summed up in one procedure. This dictates that the notion of register latches should be kept concealed from the user, i.e. a block of instruction behavior must use the same register name irrespective of the pipeline stage. After evaluating several options, we finally opted for the pipeline register model to contain a fast and simple array of values that model the latches. To enable transparent access to the correct latch, an auto-generated Register::set_stage() function is transparently called at the beginning of each instruction stage-behavior function. The register stores the current stage, and all subsequent reads or write manipulate the correct latch. An excerpt of the implementation is provided in listing 1. To add support for forwarding, we can store special write-back sequences as hash tables that map a pipeline stage to one or more stages. At each clock cycle edge, an update() function (line 17) propagates values one step forward as well as to all stages specified in the hash table. Since the correspondence is known a priori, we let TRAP-Gen generate a hard-coded mapping in clock_cycle_func() (line 22) which is called by update() every cycle. This keeps the implementation general without compromising simulation speed.

```
 1 template <typename DATA>
 2 class RegCA
 3 : public RegAbstraction<DATA> {
 4
 5   public:
 6   typedef bool (*clock_cycle_func_t)();
 7
 8   const DATA read() const {
 9     return values[current_stage] & read_mask;
10   }
11
12   bool write(const DATA & data) {
13     values[current_stage] = data & write_mask;
14     return true;
15   }
16
17   void update() {
18     // Propagate from each stage to the next.
19     // Last stage feeds back to first.
20     // Honor special feedback paths.
21     if (clock_cycle_func != NULL)
22       clock_cycle_func();
23   }
24
25   void set_stage(unsigned stage) {
26     current_stage = stage;
27   }
28 };
```

Listing 1. RegisterCA implementation.

V. Evaluation of the Register Model

To assess the performance of our model, we generated four versions of six different processors using TRAP-Gen, as listed in Table I: The first two versions were generated using the fixed-abstraction register model initially deployed in TRAP-Gen (*initial/untimed* and *initial/cycle-accurate*). The third and fourth models (*combined*) are an abstraction-agnostic version generated using our methodology, which we parametrized twice to run at the above-mentioned abstraction levels. The difference in code size between the combined/un-timed and combined/cycle-accurate models is due to the fact that, even though the registers are abstraction-parameterizable, the rest of the processor is still modeled at a fixed abstraction (see future work in Section VI).

The combined model shows a consistent decrease in the code size of the generated processor compared with the initial implementation. The effect is most perceptible for cycle-accurate models, which show a reduction of up to 85% in the case of the modern ARM Cortex A9. This dramatic decrease is due to the greater savings incurred on large register sets and/or deep pipelines.

TABLE I: Comparison of Generated Models

	Source Code Size [kiB]			
	Untimed		Cycle-accurate	
	Initial	Combined	Initial	Combined
ARM7TDMI	537	526	1453	618
ARM9TDMI	634	585	1777	704
ARM Cortex A9	1976	1188	12133	1768
LEON2	867	828	3170	1555
LEON3	851	812	4012	1793
MICROBLAZE	662	550	1191	592

VI. Conclusion

We have presented a comprehensive SystemC model for processor registers that covers several levels of abstraction. Our model greatly eases mixing and altering abstraction levels with little loss of simulation speed. Using several object-oriented design patterns, it enables changing the abstraction on a sub-module granularity, in our case, that of registers instead of processors. Our results have been integrated in the open-source processor generation tool TRAP-Gen. We have generated six processors using the cycle-accurate and un-timed modeling styles. Compared to the initial implementation in TRAP-Gen, we achieved a code-size reduction ranging between 50%-85%.

In our future work, we aim to benchmark our model with respect to runtime performance as well as other criteria. Our long-term goal is extending the concepts developed here to other processor elements. We envision unifying TRAP-Gen-generated models into a single, abstraction-parametrized model.

References

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Hingham, MA: Kluwer Academic Publishers, 2002.

[2] S. Sarkar, S. C. G., and S. Shinde, "Effective IP reuse for high quality SoC design," in *Proc. the 2005 IEEE International SOC Conference*, Sept. 2005, pp. 217–224.

[3] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-chip Designs*, Norwell, MA: Kluwer Academic Publishers, 1998.

[4] B. Bailey and K. Werner, *Intellectual Property for Electronic Systems: An Essential Introduction*, Waltham, MA: International Engineering Consortium, 2007.

[5] T. D. Schutter, *Better Software. Faster! Best Practices in Virtual Prototyping*, Mountain View, CA: Synopsys Press, 2014.

[6] I. Petkov, P. Amblard, M. Hristov, and A. Jerraya, "Systematic design flow for fast hardware/software prototype generation from bus functional model for MPSoC," in *Proc. the 16th International Workshop on Rapid System Prototyping (RSP)*, Montreal, Canada: IEEE, Jun. 2005, pp. 218–224.

[7] B. Bunton, "A comparison of TLM modeling styles' performance and accuracy," in *Proc. the 18th North American SystemC Users' Group Meeting (NASCUG)*, Jun. 2012.

[8] A. Alali, I. Assayad, and M. Sadik, "Modeling and simulation of multiprocessor systems MPSoC by SystemC/TLM2," *International Journal of Computer Science Issues (IJCSI)*, vol. 11, no. 2, May 2014.

[9] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "SoCRocket - a virtual platform for the european space agency's SoC development," in *Proc. the 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, May 2014, pp. 1–7.

[10] SoCRocket-transaction-level modeling framework for space applications. [Online]. Available: https://github.com/socrocket
[11] L. Fossati. Trap-gen. [Online]. Available: https://code.google.com/archive/p/trap-gen/
[12] M. Montoreano, "Transaction level modeling using OSCI TLM 2.0," *Open SystemC Initiative (OSCI), Tech. Rep.*, May 2007.
[13] IP Exchange. Carbon design systems. [Online]. Available: http://www.carbondesignsystems.com/
[14] Designware processor IP portfolio, synopsys, Inc. [Online]. Available: https://www.synopsys.com/dw/doc.php/ds/cc/dw-processor-solutions.pdf
[15] Tensilica Processors, Cadence Design Systems, Inc. [Online]. Available: http://ip.cadence.com/knowledgecenter/know-ten/.
[16] Xtensa Processor Generator, Cadence Design Systems, Inc. [Online]. Available: http://ip.cadence.com/hwdes/
[17] MicroController IP, Mentor Graphics. [Online]. Available: https://www.mentor.com/products/ip/peripheral/microcontroller/.
[18] ISS - The Imperas Instruction Set Simulator, Imperas Software Ltd. [Online] Available: http://www.imperas.com/iss-the-imperas-instruction-set-simulator/
[19] F. Bellard. QEMU. Open Source Processor Emulator. wiki.qemu.org/Main Page.
[20] OpenCores. [Online]. Available: http://opencores.org/
[21] SoCLib. [Online]. Available: http://www.soclib.fr/
[22] N. Pouillon, A. Becoulet, A. V. de Mello, F. Pecheux, and A. Greiner, "A generic instruction set simulator API for timed and untimed simulation and debug of MP2-SoCs," in *Proc. 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, Jun. 2009, pp. 116–122.
[23] The gem5 simulator. [Online]. Available: http://www.gem5.org/
[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[25] *ARM Architecture Reference Manual*, ARM Ltd. Std. ARMv7-A and ARMv7-R edition (C.b), July 2012.
[26] D. L. Weaver and T. Germond, *The SPARC Architecture Manual: Version 9*, SPARC International Inc. Std., 1994.
[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Elements of reusable object-oriented software," in *Professional Computing Series*, B. W. Kernighan, Ed. Addison-Wesley, Mar. 2007.
[28] S. Swan and J. Cornet. Beyond TLM 2.0: New virtual platform standards proposals from ST and Cadence. [Online]. Available: wiki.qemu.org/Main Page
[29] *GRLIB IP Core User's Manual*, Version 1.4.1 - b4156 May 2015, Cobham Gaisler, Göteborg, Sweden, 2015.
[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture. A Quantitative Approach*, Waltham, MA: Morgan Kaufmann, 2012.

Lillian Tadros obtained her Dipl.-Ing. in electrical engineering and information technology from the Ruhr-Universität Bochum, Germany, in 2007. She joined the Institute for Robotic Studies of the Technische Universität Dortmund, Germany, in 2014, where she is currently a research assistant. Her research focuses on multi- and many-core platforms for embedded and cyber-physical systems.