MULTI-VIEW OPERATION-LEVEL DESIGN – SUPPORTING THE DESIGN OF IRREGULAR ASIPS

by

Scott J. Weber, Matthew W. Moskewicz,
Manuel Löw and Kurt Keutzer

Memorandum No. UCB/ERL M03/12

4 April 2003

MULTI-VIEW OPERATION-LEVEL DESIGN – SUPPORTING THE DESIGN OF IRREGULAR ASIPS

by

Scott J. Weber, Matthew W. Moskewicz,
Manuel Löw and Kurt Keutzer

Memorandum No. UCB/ERL M03/12

4 April 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Multi-view Operation-level Design – Supporting the Design of Irregular ASIPs

Scott J. Weber, Matthew W. Moskewicz, Manuel Löw, Kurt Keutzer

ABSTRACT

Architectural description languages are increasingly being used to define architectures and to generate key elements of their programming environments. A new generation of application-specific instruction processors is emerging that does not fit well in an instruction-set architecture paradigm. In this paper, we propose a methodology that supports the design of a wider range of programmable architectures than existing architecture description languages. We achieve this through a flexible framework that is centered on a constraint-based functional hardware description language that abstracts computation using the notion of primitive *operations*. Using multiple *views* acting on a common model, our methodology ensures consistency between architectural design, tool generation, and hardware implementation. We term our methodology *multi-view operation-level design*.

1. INTRODUCTION

Traditionally, the number of programmable architecture design starts has been few, and as a result, little commercial tool support exists for this market. But, with the increasing replacement of application-specific integrated circuits (ASICs) with application-specific instruction processors (ASIPs) it seems natural that electronic design automation should support such systems. These designs require a suite of development tools including a simulator, assembler, and compiler. To actually realize the hardware design, a netlist must be generated from a description of the architecture in a traditional hardware description language (HDL).

A variety of architectural description languages (ADLs) [1][2] have been proposed to assist in the design of programmable architectures. A common aspect of existing ADLs is to restrict the scope of architectures supported in order to manage the complexity of the associated tool generation. We feel that these restrictions will force the designers of emerging irregular architectures, such as the IXP1200 [3] and Calisto [4], and Virtex II Pro [5] to continue to use traditional design practices that involve the manual construction of simulators, assemblers, compilers, and HDL models. Manual construction is time consuming and error-prone. If you agree that ASIP architectures will be increasingly diverse and irregular, then it is imperative to provide a better methodology to enable their design.

To support this new generation of ASIP architectures, we propose a component abstraction above the register-transfer level (RTL) abstraction that allows for the design of any programmable architecture that can be described as an arbitrary collection of datapaths which can be configured. RISC datapaths, systolic arrays, special purpose co-processors, and FPGA look-up tables all fall within this category. From each datapath description, we automatically extract all operations that the datapath can execute. From these extracted operations, we are able to generate the tools required for programmable architecture design. These tools include high-speed cycle-accurate simulators and an assembler. It is important to note that we can produce these tools without committing to a particular control strategy (e.g. horizontal microcode, traditional RISC controller, etc.). The design can be simulated with control abstracted as a simple trace of the operations to be executed. Ultimately, a controller that implements the control strategy must be chosen so that a synthesizable RTL hardware model can be produced for the complete design.

At the core of our methodology is a common model with well-defined semantics. The model contains information about the structure of the architecture and the supported operations. Multiple *views* (synonymous with abstractions) are then developed on top of the model to encapsulate and present the information required by each facet of the design process. Practically, a view consists of a set of tools that operate on a model to extract some particular information or perform some transformation. Views have the invariant that they must respect all of the semantics of the model, and never infer anything that is not present in the model. This implies that once a view is (correctly) written, the outputs of that view will be correct by construction with respect to any model. Thus, the multiple views enable designers such as the simulator writer, assembly writer, and compiler writer to work with the design from differing but always consistent perspectives. The key to achieving semantic consistency over such a broad range of architectures in our methodology is an operation-level description that captures both structural and behavioral aspects of the design while orthogonalizing control aspects.

The paper is organized as follows. In Section 2, we review prior work on ADLs. In Section 3, we discuss multi-view design. In Section 4, we use a small example to illustrate how one designs a micro-architecture in the architecture view. Section 5 discusses how the operation view is generated. Section 6 discusses how the high-speed cycle-accurate simulator is generated. Section 7 discusses the assembly view. Section 8 discusses the hardware view. A case study is presented in Section 9. Future work is discussed in Section 10. We conclude in Section 11.

2. PRIOR WORK

Within the last decade, a variety of ADLs (references to the ADLs named here can be found in the surveys [1][2]) have been introduced to assist in the development of new architectures. All ADLs must make trade-offs among ease-of-expression, breadth of architectures supported, and the quality of the generated tools. Behavioral ADLs (nML, ISDL, and CSDL) allow designers to describe the instruction semantics to create a programmer's view of the architecture. In fact, Target Compiler Technologies[1] has commercialized the use of nML in its

[1] http://www.retarget.com

Chess/Checkers [6] environment. However, the lack of sufficient structural details in behavioral ADLs makes it difficult to extract an efficient micro-architecture.

To attack this limitation, mixed ADLs (FlexWare, Maril, HMDES, TDL, EXPRESSION, and LISA) were developed to enable designers to describe the instruction semantics, the micro-architectural structure, and a mapping between the two. In fact, companies such as AXYS[2] and LISATEK[3] are providing high performance cycle-accurate simulators and system integration tools based on custom processors described with LISA [7]. However, these ADLs, as well as the behavioral ADLs, lack support for designs outside a limited family of architectures and system integration is performed in an informal manner.

One way to model architectures outside the scope of these ADLs is to use a HDL such as Verilog, VHDL, or SystemC. These languages support any architecture, but lack the notion of operations. Stylized HDLs such as the structural ADL MIMOLA [8] do capture the concept of instructions albeit with the requirements that they are single-cycle microcoded operations. The MIMOLA effort demonstrates that it is possible to extract instructions [9], generate a compiler and a simulator, and synthesize a design from a single low-level description. Tools supporting MIMOLA, however, assume certain uses of architecture features such as a program counter, a register file, instruction memory, and single-cycle operations.

The existing ADLs have certainly enabled the use of a broader scope of programmable architectures in designs. However, ADL developers are faced with the dilemma that they want to support a wider a range of architectures without sacrificing generated tool quality. We address this dilemma by developing an operation-level abstraction capable of modeling any programmable architecture. We then formalize the activity of generating tools and HDL models from this abstraction as a process of restrictions and invariant preserving transformations. The range of architectures that we support is then no longer a function of the ADL but of the breadth of the available views.

3. MULTI-VIEW

Traditionally, a HDL is used to describe a micro-architecture. However, HDLs lack semantics to make a formal distinction between the control logic and the datapath. Architects instead make this distinction by specifying the instructions and the corresponding control logic manually. Validating the correctness of these instructions requires extensive simulation and is error prone. To alleviate the pain of verification in our approach, we have developed an operation-level abstraction that separates control from data, thus enabling the exportation of the architecture as a set of operations.

Abstracting an architecture as a collection of operations is a key enabler for our multi-view methodology. The abstraction gives us the model from which the simulator, assembler, and compiler *views* can be developed. Using these *views*, the simulator, assembler, and compiler can then be automatically generated. Furthermore, the underlying model allows us to develop these multiple views without requiring a single cumbersome syntax for all views. We feel that no single syntax is natural for architects, compiler designers, and simulator writers because each is concerned with different facets of the design.

Our methodology dictates that views must work only with semantics explicit in the model. We do not guarantee that a view will succeed in manipulating the model. It is up to the designer of a view to ensure that any transformation that a view applies to the model has the intended effect, and that any outputs of the view represent correct extractions of information in the model. The issue of designing and verifying the views themselves is outside the scope of this paper, but there are many alternatives, including standard random simulation as well as formal verification techniques similar to those used to verify that multiple specifications of a single processor conform to some property [10].

To illustrate the benefits and limitations of the multi-view methodology, consider the creation of a compiler view. The intent of a compiler view is to provide a means to automatically generate a compiler given an architecture. Because our underlying model is quite descriptive, it is possible to design an architecture outside the scope of a compiler view. For example, a compiler view that only understands non-clustered architectures may fail if the architecture has a split register file. To solve this problem, either the architecture view is restricted or the compiler view is enhanced. It is exactly this appropriate defining, partitioning, and restricting of views that is the cornerstone to a successful framework.

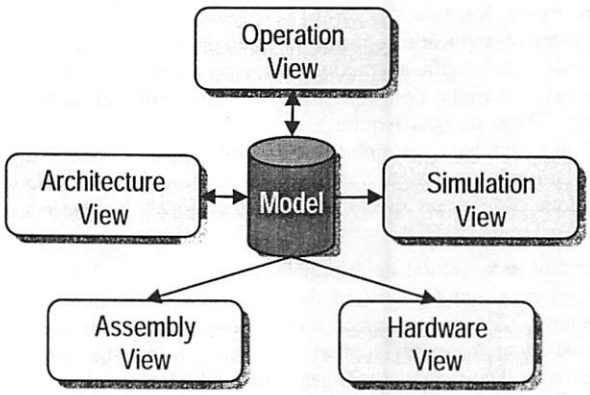

Figure 1. Multiple Views

To demonstrate the methodology, we propose an initial framework composed of the five views shown in Figure 1. Designers can compose hardware blocks in the architecture view. The computational functions that can be performed by the architecture are extracted and presented by the operation view. The simulation view uses information from the architecture and operation views to generate a high-speed cycle-accurate simulator. In order to program the architecture, a simple traditional assembler is generated in the assembly view. Finally, a synthesizable RTL hardware model is generated in the hardware view, thus enabling a realization of the architecture. These views represent an introduction to a framework which will also include a compiler view.

4. ARCHITECTURE VIEW

The architecture view provides the designer the ability to describe the datapath. In our architecture view, datapaths are

[2] http://www.axysdesign.com
[3] http://www.lisatek.com

described by connecting *actors* with *signals* between *ports*. The term *color* refers to an abstract property used to distinguish signals. Rather than encoding signals as a collection of bits (as in a traditional HDL), we instead abstract the interesting properties of signals by choosing appropriate colors. The coloring of a signal subsumes the traditional notions of a signal of having a "type", a constant value, or a symbolic value. Generally, a signal has a set of possible colors determined by its context. There is a color, "not present", which corresponds to a logical "X" in a three-valued logic. A signal has this color when it contains no information. Almost all signals have the "not present" color in their set of possible colors. Signals such as the control inputs to multiplexers are generally abstracted as enumerations of constants because all possible values of these signals must be distinguishable during analysis.

Colors can be grouped together into sets (hereafter referred to as color-sets). Besides equality, which is defined over all colors, color-sets may have additional axiomatic relations defined. For example, the bit vector color-set has the standard logical and arithmetic functions that one would expect in an HDL. The colors in a color-set not only include constants (expressed as "color-set.color"), but also symbolic variables (expressed as "color-set.varname"). For example, each source in a circuit will introduce a fresh symbolic color. The signal bound to each source will have only this color in its set of possible colors (technically, because sources can be unused, "not present" is also a possible color for source signals). Additional symbolic colors are introduced to represent all possible computation by actors. All such computation by actors occurs within *firing rules* which define the axiomatic relations between inputs and outputs. Firing rules can only be active when certain color constraints on the signals bound to the ports of the actor are met. Note that this is not the same as a simple color-set constraint, because it can also test against specific colors. Additionally, actors specify a validity constraint using first-order logic over the "activeness" of their firing rules. Typically, this constraint specifies that at least one firing rule is active, but more sophisticated constraints are possible and sometimes necessary. In our current system, we have defined the three color-sets as shown in Table 1.

Table 1. Color-set Definitions

Color-set	Description
X	Set containing only "not present" color.
enum	Set of colors for abstracting enumerations.
bit	Set of colors for abstracting bit vectors.

Examples in Figure 2 illustrate how actors are defined in our language. For each firing rule, the color constraints are in parentheses after the name of the rule, and the axiomatic relations are contained in the curly braces. The first actor is a demux which has two rules for passing the input to the appropriate output port, and one rule that does nothing. The second and third actors represent an incrementer and decrementer, respectively. The final actor is a mux. Note that the color constraints for the mux and demux match against a specific color on the control signal, but only check color-set membership on the other signals. This distinction is typical of the difference between what are traditionally considered control and data signals. When analyzing different configurations of the hardware, control signals tend to have constant values whereas data signals tend to remain symbolic variables. However, the framework makes no formal distinction between control and data signals.

Although not shown in Figure 2, two primitive state actors, the *register* and *flip-flop*, are also defined. A *register* holds a value written to it until the value is overwritten. A *flip-flop* holds the value written to it for one cycle. Both of these state actors are parameterized by size. With these primitive state elements, we can define RAMs, ROMs, register files, and other useful components. However, it is important to understand that operations as defined here always occur within a single cycle, and thus for analysis, all state elements are constrained to appear as sources or sinks.

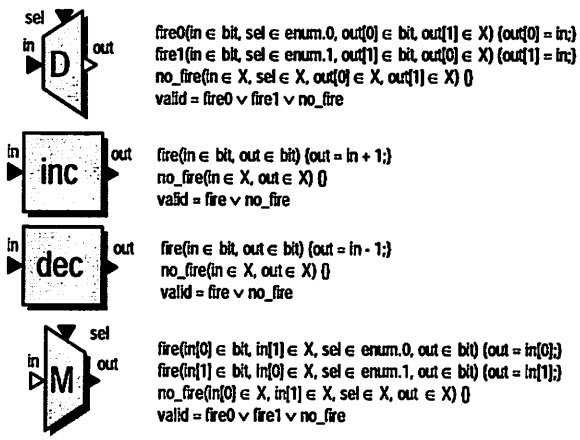

Figure 2. Actor Semantics

After defining the actors, an architect composes them by connecting their ports with signals in a hierarchical schematic editor. Most architects will use predefined library actors which are parameterized by port width. An example composition is shown in Figure 3. We purposely leave ports unconnected if we want these ports to be programmable. The non-determinism of these ports will be leveraged in the operation view to generate the supported operations of the micro-architecture.

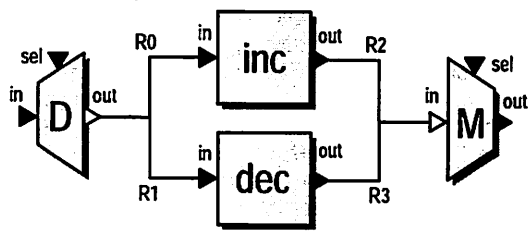

Figure 3. Example Architecture

We have developed a particular architecture view based on a constraint-based functional language, but other architecture views that are capable of describing our wide range of architectures could also be used or developed. For example, the MIMOLA ADL [8] could be used. In fact, our architecture view is similar to MIMOLA except that we abstract control values. The abstraction enables delaying the encoding of control until it is actually needed in the hardware view. Furthermore, we automatically generate the encoding.

5. OPERATION VIEW

The operation view provides the designer the ability to extract all interesting configurations of the datapath described in the architecture view. To extract the operations, we first translate the design to first order logic. Figure 4, for example, demonstrates the formulation of the architecture shown in Figure 3. The possible colors for each signal are defined as a set. Each actor has a set (named actor.fire) that encodes which of its firing rules are active. Because the validity constraint for each of these actors is a simple disjunction of its rules, the set is constrained to be non-empty.

$D.in \in \{X, \chi\}$, $D.sel \in \{X, 0, 1\}$, $R0 \in \{X, \delta\}$, $R1 \in \{X, \varepsilon\}$, $R2 \in \{X, \phi\}$, $R3 \in \{X, \gamma\}$, $M.sel \in \{X, 0, 1\}$, $M.out \in \{X, \alpha, \beta\}$, $D.fire \subseteq \{fire0, fire1, no\_fire\}$, $inc.fire \subseteq \{fire, no\_fire\}$, $dec.fire \subseteq \{fire, no\_fire\}$, $M.fire \subseteq \{fire0, fire1, no\_fire\}$

$((D.in = \chi \wedge D.sel = 0 \wedge R0 = \delta \wedge R1 = X) \Leftrightarrow (fire0 \in D.fire)) \wedge$
$((D.in = \chi \wedge D.sel = 1 \wedge R0 = X \wedge R1 = \varepsilon) \Leftrightarrow (fire1 \in D.fire)) \wedge$
$((D.in = X \wedge D.sel = X \wedge R0 = X \wedge R1 = X) \Leftrightarrow (no\_fire \in D.fire)) \wedge$
$(D.fire \neq \varnothing) \wedge$

$((R0 = \delta \wedge R2 = \phi) \Leftrightarrow (fire \in inc.fire)) \wedge$
$((R0 = X \wedge R2 = X) \Leftrightarrow (no\_fire \in inc.fire)) \wedge$
$(inc.fire \neq \varnothing) \wedge$

$((R1 = \varepsilon \wedge R3 = \gamma) \Leftrightarrow (fire \in dec.fire)) \wedge$
$((R1 = X \wedge R3 = X) \Leftrightarrow (no\_fire \in dec.fire)) \wedge$
$(dec.fire \neq \varnothing) \wedge$

$((R2 = \phi \wedge R3 = X \wedge M.sel = 0 \wedge M.out = \alpha) \Leftrightarrow (fire0 \in M.fire)) \wedge$
$((R2 = X \wedge R3 = \gamma \wedge M.sel = 1 \wedge M.out = \beta) \Leftrightarrow (fire1 \in M.fire)) \wedge$
$((R2 = X \wedge R3 = X \wedge M.sel = X \wedge M.out = X) \Leftrightarrow (no\_fire \in M.fire)) \wedge$
$(M.fire \neq \varnothing)$

$[[\delta]] = [[\chi]]$; $[[\varepsilon]] = [[\chi]]$; $[[\phi]] = [[\delta]] + 1$; $[[\gamma]] = [[\varepsilon]] - 1$; $[[\alpha]] = [[\phi]]$; $[[\beta]] = [[\gamma]]$.

Figure 4. First Order Logic Formulation of Example

After formulating the architecture as a first order logic expression, we then find the set of satisfying solutions. These solutions represent the operations supported by the architecture. The solutions are found using an iterative SAT procedure shown in Figure 5 called FindMinimalOperations (FMO). The procedure must be restricted to find "minimal" solutions, as there are generally an exponential number of solutions, scaling with the amount of independent parallelism in the design. A solution is "minimal" when no additional signals can be "X" (or not present) and still satisfy the model. The inner loop of FMO does this minimization. Following the creation of an operation, we restrict the model formula so that subsequent "minimal" operations are not simply combinations of previous operations. By limiting FMO in this manner, we can quickly find the supported operations. Either the assembly or compiler view then combines these operations in time and/or space to create what would be considered more traditional instructions.

Figure 6 shows the resulting solutions found by running FMO on the example architecture in Figure 3. Each solution indicates the colors of the signals. The first solution represents the operation to do nothing, the second solution performs the increment operation, and the third solution is the decrement operation. The associated axiomatic relations will allow us to construct logic that computes the values of any sink signal's symbolic colors in terms of the source signals' symbolic colors.

It is common that some of the operations generated are unwanted or have spatial and/or temporal constraints between each other. A spatial constraint indicates that a set of operations always occur together on a cycle, whereas, a temporal constraint specifies a particular order for a set of operations. Using these constraints, we can build traditional instructions by chaining operations. For example, a three-operand register-to-register "add" instruction for a 5-stage DLX would be implemented by adding a temporal constraint to the set of operation constraints. The constraint would indicate that the operations that constitute the "add" in the IF, ID, EX, MEM, and WB stages occur one cycle after another in that pipeline order. Removing operations and constraining relationships between operations allows a hardware view to synthesize more efficient control. For this reason, we view restricting and constraining operations as an integral part of the design process. The unrestricted operations and constraints between these operations represent the programmability of the architecture.

BASE is the CNF formulation of the model
NET is the set of net literals
CERT is a certificate, i.e. set of literals (satisfying the BASE)
CERTS is the set of all CERT
present: (NET x CERTS) → boolean (true if net is present in the cert)
isSatisfiable: ALL_CNFS → boolean (true if the CNF is satisfiable)
getCertificate: the last certificate that made *isSatisfiable* TRUE

OPERATIONS = {}, OPNETS = {}
while (*isSatisfiable*(BASE)) {
  do {
    C = *getCertificate*()
    "remove the constraints added in [1]"
    $BASE = BASE \wedge (\sum_i \{\neg net_i \mid net_i \in NET \wedge \neg present(net_i, C)\})$ [1]
    $BASE = BASE \wedge (\prod_i \{\neg net_i \mid net_i \in NET \wedge present(net_i, C)\})$ [2]
  } **while** (*isSatisfiable*(BASE))
  "remove the constraints added in [2]"
  "create new operation named 'op' based on the certificate"
  $OPNETS = OPNETS \cup \{op \rightarrow \{net \mid net \in NET \wedge \neg present(net, C)\}\}$
  $OPERATIONS = OPERATIONS \cup op$
  $BASE = BASE \wedge (\prod_i \{net_i \mid net_i \in NET \wedge present(net_i, C)\} \Leftrightarrow op)$ [3]
  $BASE = BASE \wedge (\sum_i \{net_i \mid net_i \in NET\}$
    $\wedge (\prod_j \{\neg op_j \mid op_j \in OPERATIONS \wedge net_i \in OPNETS(op_j)\}))$ [4]
}
"remove the constraints added in [3], and [4]"

Figure 5. FindMinimalOperations (FMO)

We have developed a particular operation view that utilizes SAT to extract the operations. The MIMOLA effort also provides a method to extract operations using BDDs [9]. However, our approach offers the designer the freedom to delay the control implementation as mentioned in Section 4. To support this abstraction, we perform a simple transformation of the design to first-order logic. We then apply SAT to extract the operations. In practice, we have found that this formulation is straightforward and easy to solve.

NOP: $D.sel = X \wedge D.in = X \wedge R0 = X \wedge R1 = X \wedge R2 = X \wedge R3 = X \wedge$
$M.sel = X \wedge M.out = X \wedge D.fire = no\_fire \wedge inc.fire = no\_fire \wedge$
$dec.fire = no\_fire \wedge M.fire = no\_fire$
inc: $D.sel = 0 \wedge D.in = \chi \wedge R0 = \delta \wedge R1 = X \wedge R2 = \phi \wedge R3 = X \wedge$
$M.sel = 0 \wedge M.out = \alpha \wedge D.fire = fire0 \wedge inc.fire = fire \wedge dec.fire = no\_fire \wedge$
$M.fire = fire0$
dec: $D.sel = 1 \wedge D.in = \chi \wedge R0 = X \wedge R1 = \varepsilon \wedge R2 = X \wedge R3 = \gamma \wedge$
$M.sel = 1 \wedge M.out = \beta \wedge D.fire = fire1 \wedge inc.fire = no\_fire \wedge dec.fire = fire \wedge$
$M.fire = fire1$

Figure 6. Satisfying Certificates for Example

6. SIMULATION VIEW

After describing the architecture and extracting the supported operations, we must provide a means to simulate the operations. Our simulator view provides the capability to generate a high-speed cycle-accurate simulator for the architecture. Given an operation, we translate any necessary axiomatic relations into operations of a language such as C/C++. For each operation found with the static analysis performed in the operation view, we are able to remove all of the control signals or any others which are constant leaving only the relevant state-to-state combinational logic for each instruction (an *instruction* is defined as a set of operations to be executed within a cycle). Furthermore, since the program is specified as a series of symbolic operations, there is no need to simulate or specify any particular encoding of the operations or decoder. The resulting simulator is a compiled-code simulator for any realization of this architecture.

If a control strategy (consisting of an encoding of the instructions and decoder logic) exists, then an interpretive simulator can be generated. We provide a mechanism to generate a simple default control strategy using horizontal microcode. An interpretive simulator can be useful if the program to be executed is not known. However, the interpretive simulator suffers in performance compared to the compiled-code simulator.

Our generated simulators are comparable to other top performing simulators such as LISA [7] and JACOB [11]. However, we feel that our simulators have several advantages: they support a wider range of architectures, generating the simulator from the extracted operations is simpler than the LISA approach [7], and the strong semantics of our underlying model allows for fairly aggressive optimizations and transformations.

7. ASSEMBLY VIEW

Our assembly view represents the first step towards the generation of a compiler. Given a set of operations, we generate an assembler for that set. The primary task of the assembler is to read an assembly code file and produce the appropriate pre-decoded machine code for our simulator. The assembly code file consists of a list of groups of concurrent operations that are, in turn, to be executed in sequential order. During the machine code generation, the assembler verifies that a group of operations can be "issued" together. To verify this property, we perform a satisifiability check on the model with the union of all firing rules in the group of operations forced to be active. If it is satisfiable, then the group of operations can be issued together. We also check that the spatial and temporal constraints are met. This is important because the simulator (which has the control logic simplified away) does not perform any such check, and clearly we do not wish to simulate operation combinations that are not possible on the actual hardware.

8. HARDWARE VIEW

We can simulate and program our architecture without explicitly specifying the control logic, but to actually realize an implementation of the architecture, we must commit to a control strategy. There are many control strategies. For very simple programs, a dedicated FSM that represents the program can be designed. Another simple control strategy that we have implemented as the default is horizontal microcode. More elaborate control strategies such as reconfigurable control and RISC or VLIW encoding schemes could also be designed. Each control strategy uses the information about the operations supported and the spatial and control constraints to optimize the generated controller. Once a control strategy has been chosen, the hardware view can then produce a synthesizable RTL hardware model by combining the information in the architecture view with the control strategy to realize an implementation. We then leverage a traditional HDL flow to generate hardware and provide combinational timing feedback for each operation.

9. CASE STUDY

To demonstrate and evaluate our tools and methodology, we supported an experienced industrial ASIC designer who implemented a channel encoding processor using our methodology (Figure 7). The processor is capable of performing CRC, UMTS Turbo/convolutional encoding, and 802.11a convolutional encoding. The design is composed of approximately 60 actors which are divided into a "control" plane composed of a PC and zero-overhead looping logic, and a "data" plane composed of a register file, accumulator, and bit manipulation unit. The division of planes is for the designer, not the tool. The designer chose to leverage our default control strategy generation (horizontal micro-code) by leaving the select ports of the multiplexers, the register file address ports, and immediate value ports unconnected. The non-determinism of these unknown values represents the programmability of the architecture; the generated controller must produce appropriate values for these ports on each cycle. Upon analysis, the design has 46 operations, of which half were unwanted and thus restricted.

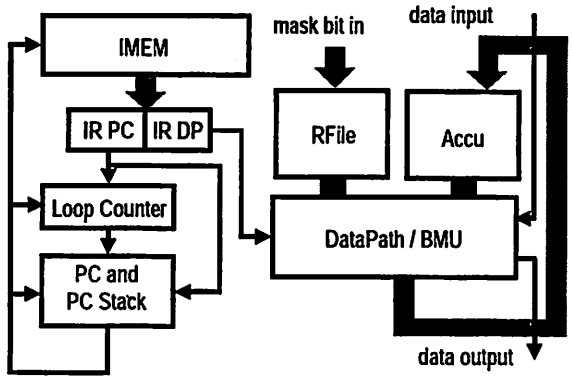

Figure 7. Channel Encoding Processor

To program the processor, each instruction generally consists of "control" and "data" operations. The "data" operations are specialized bit manipulation operations for CRC and convolution encoding. Due to resource sharing, most of these operations are mutually exclusive. The "control" operations are things such as increment PC, set loop counter, decrement loop counter, and test end of loop. These can be combined to form an instruction that, for example, decrements the loop counter, branches if the loop counter is 0, otherwise increments the PC. The instruction also includes additional data plane operations. The applications targeted for the processor contain many

streaming tasks that require multiple cycles per sample. To implement these applications, the designer created zero-overhead loops containing small sets of instructions to perform each task. Effectively a designer can use an ASIC methodology to design a programmable architecture by specifying the requisite "control" and "data" planes and leverage our methodology to extract the operations and generate the simulator and assembler.

The combined runtimes of each view are low enough to enable design space exploration. All experiments were performed on a 1.33 GHz Athlon with 768MB of memory. For the channel encoding processor, it takes 24 seconds to generate the operations, 5 seconds to generate the interpretive or compiled-code simulator, and half a second to compile the simulator. Our views are all written in Java, but we use relsat [12] in the FMO algorithm. In order to get the highest performance simulator, we generate C++ and use gcc3.2 with –O3 to create the simulator. The compiled-code simulator executes approximately 40 million *instructions* (i.e. cycles) a second. The interpretative simulator executes approximately 5 million *instructions* (i.e. cycles) a second. Each operation requires on the order of 10 host machine instructions for decoding control (which is only present in the interpretive simulator), and roughly one instruction for each integer computation. These performance numbers are on par with existing pipeline and cycle accurate simulators.

10. FUTURE WORK

Our most ambitious effort is our compiler view. Like the assembly view, the compiler view is parameterized by the generated operations. Unlike the assembly view, the compiler view will accept programs in a higher level language and will be able to optimize such programs using heuristics we develop to map a program to the primitive operations. This view will complete the flow from a high-level language and architecture to machine code running on actual hardware. The most difficult step is to formulate traditional compiler optimizations in a way that is compatible with a highly variable and heterogeneous set of operations. Most likely, a combination of traditional hand-written heuristics (for control flow) and generic combinational matching (for straight line code) [13] will be required. We believe that such an approach will allow us to use a single compiler view to capture a larger family of programmable architectures than is currently possible with existing compilers.

11. CONCLUSION

We have shown how our operation-level abstraction enables a semantically consistent multi-view operation-level design methodology. This in turn enables automatic high-speed cycle-accurate simulator, assembler, and synthesizable RTL hardware generation. Our case study with an industrial ASIC designer has convinced us of the soundness and utility of the approach.

We believe that our methodology benefits from the fact that it supports a very broad range of programmable architectures. The key is to use a single common model to generate all needed views of the design. In this way, semantic consistency is guaranteed.

ASIC designers approaching ASIP design for the first time will find it natural to describe the architecture at the hardware description level in terms of operations, and our case study gives support for this assertion. The assembler and compiler can then be automatically generated in order to provide a programmer's view of the architecture. We believe that simulation must be fast and accurate in order to enable the exploration of a large design space and we automatically generate such a simulator in our approach. We also believe that a path to implementation must exist, and we provide such a path with our hardware view. In this way, we help to automate the design and abstraction of a new generation of irregular ASIP architectures.

12. REFERENCES

[1] W. Qin, S. Malik, "Architecture Description Languages for Retargetable Compilation," in The Compiler Design Handbook: Optimizations & Machine Code Generation, CRC Press, 2002, (Editors Y.N. Srikant and Priti Shankar).

[2] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau, "Architecture Description Languages for Systems-on-Chip Design," In *Proc. of 6th Asia Pacific Conference on cHip Design Languages (APCHDL'99)*, Invited paper, pp. 109-116, Oct. 1999.

[3] Intel Corporation. IXP1200 Network Processor Database.

[4] J. Nickolls, L.J. Madar III, S. Johnson, V. Rustagi, K. Unger, M. Choudhury, "Broadcom Calisto: A Multi-Channel Multi-Service Communications Platform," in Hot Chips 14, Aug. 2002.

[5] Xilinx Corporation. Virtex Pro II Data Sheet.

[6] D. Lanneer, J. Van Praet, A. Kifli, K. Shoofs, W. Geurts, F. Thoen, and G. Goossens. *CHESS: Re-targetable Code Generation for Embedded DSP Processors,* In *Code Generation for Embedded Processors* (p. Marwedel and G. Goossens, ed.), Kluwer Academic Publishers, 1995.

[7] S. Pees, A. Hoffman and H. Meyr, "Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, no. 4, Oct. 2000, pp. 815-834.

[8] R. Leupers and P. Marwedel, "Retargetable Code Generation Based on Structural Processor Descriptions," *Design Automation for Embedded Systems*, vol. 3, no. 1, Jan 1998, pp. 1-36.

[9] R. Leupers, *Instruction-Set Extraction,* In *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997, pp. 45-83.

[10] J. R. Burch and D. L. Dill, "Automatic Verification of Pipeline Microprocessor Control," *Proc. 6th Int'l Conf. Computer Aided Verification* (CAV 94), *Lecture Notes in Computer Science*, vol. 818, Springer Verlag, Berlin, 1994, pp. 68-80.

[11] R. Leupers, J. Elste, and B. Landwehr. "Generation of Interpretive and Compiled Instruction Set Simulators," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 1999.

[12] R. J. Bayardo Jr. and R. Schrag. "Using CSP look-back techniques to solve exceptionally hard SAT instances." In *Proc. of the Second Int'l Conf. on Principles and Practice*

of Constraint Programming (Lecture Notes in Computer Science 1118), 46-60, Springer, 1996.

[13] R. Joshi, G. Nelson, and K. Randall, "Denali: a goal-directed superoptimizer," Compaq SRC Report 171.