Enforcing Architectural Contracts in High-level Synthesis

Nikhil A. Patil
University of Texas at Austin
npatil@mail.utexas.edu

Ankit Bansal
University of Texas at Austin
ankit@mail.utexas.edu

Derek Chiou
University of Texas at Austin
derek@ece.utexas.edu

ABSTRACT

We present a high-level synthesis technique that takes as input two *orthogonal* descriptions: (a) a behavioral architectural contract between the implementation and the user, and (b) a microarchitecture on which the architectural contract can be implemented. We describe a prototype compiler that generates control required to enforce the contract, and thus, synthesizes the pair of descriptions to hardware.

Categories and Subject Descriptors

B.5.2 [**Register Transfer Level Implementation**]: Design Aids—*Automatic Synthesis*; B.1.4 [**Control Structures and Microprogramming**]: Microprogram Design Aids—*Machine-independent microcode generation*

General Terms

Algorithms, Design, Languages

Keywords

Architecture, microarchitecture, E-unification, synthesis

1. INTRODUCTION

Processor design is a challenging task involving several man-years of labor. Describing hardware at the RTL-level is a significant component of this task. One way to reduce the complexity of describing hardware is to use a high-level synthesis language.

Computer architects generally make a strong distinction between architecture and microarchitecture. Architecture refers to the contract between the designer of a module and its user, whereas microarchitecture refers to the way an architecture is implemented. The instruction-set architecture (ISA) is conveniently expressed as a behavioral description of the instruction set, whereas the microarchitecture could be anything from purely combinational logic to a superscalar, out-of-order pipeline. It would therefore be convenient to have a synthesis language that allows the architecture and the microarchitecture to be specified separately. In fact, such a split description is commonly used during architectural simulation[18, 6]. This paper describes a compiler that synthesizes such a split description to hardware.

Most hardware can be (conceptually) partitioned into a data-path and a control-path. The data-path belongs entirely to the microarchitecture but the control-path is determined by both, the architecture and the microarchitecture. We define *microarchitectural control* as the control required to implement core microarchitectural ideas like pipelining, instruction scheduling, arbitration policies etc., and *architectural control* as the control required to implement each instruction on that microarchitecture. Specifying architectural control explicitly is tedious and error-prone, requiring serious verification; moreover, it may need revision if the architecture or the microarchitecture changes.

We require the user to specify the behavior of each instruction as a part of the architectural specification, but do not require her to specify *how* that instruction is implemented on the microarchitecture. This makes the description substantially simpler, and shifts the burden of generating the architectural control onto the compiler. Among other things, this includes generation of microcode. By automating microcode generation, we can avoid a large class of human errors. Since the architectural description can be reused over several iterations of the microarchitecture, this also reduces the amount of work required to roll out an implementation. In fact, an ISA description is often already present as a part of the instruction-set simulator. It is very easy to make incremental additions to the ISA—the compiler does the hard work of implementing them on each microarchitecture.

More significantly, we believe that our technique simplifies processor description to the point that it may eventually replace software simulation as the staple for design-space exploration. For example, one can use this technique to implement different ISAs on a common microarchitecture, to generate Pareto-optimal curves of microarchitectures for a given ISA, study the performance of benchmarks on different vector-instruction extensions, etc. Performing design-space exploration in a synthesizable language is already very attractive[5, 12], since it allows the architect to obtain much more accurate estimates of area, power and delay from the downstream synthesis and physical design tools. High-level synthesis languages are already being marketed for this purpose, but none that we know of attempt to automatically generate the architectural control.

High-level synthesis research has divided itself into two fundamental paradigms[3]: (a) behavioral synthesis, which provides the user with a familiar language (like C) but gives little or no control over the generated microarchitecture, and (b) non-behavioral synthesis, which gives full control over the microarchitecture, but does not support a high-level behavioral specification (e.g., Kiwi[9], Bluespec[15]). As architects, we want both: a top-level behavioral specification and full control over the microarchitecture. This suggests two approaches: (a) start with a C-synthesis tool and add support for constraining the compiler to a user-specified microarchitecture, or (b) start with a non-behavioral synthesis tool and add support for behavioral specification. We take the latter approach in this paper.

We show that for a common class of microarchitectures, in which instructions are decoded into microinstructions (μops) that "flow" with data, it is possible to associate all architectural control with the μop. By abstracting the μop into a special entity that the compiler automatically generates, we let the user specify the architecture (§2) and microarchitecture (§3) with almost no information overlap. We describe a compiler that generates correct architectural control that automatically *enforces* the architectural contract.

The crucial insight here is that, modulo certain restrictions, the functionality of an instruction can be represented as a mathematical term, and the functionality of a microarchitecture can be represented as a mathematical function. The inputs to this function are the control-bits in the μop. Our major contribution is that we reduce the problem of generating architectural control, to the problem of *equational unification* on the terms representing the instruction and the functionality of the microarchitecture (§4).

We have implemented a prototype compiler that generates hardware from a split description (§5). Rather that emit hardware as Verilog, we emit Bluespec SystemVerilog[15], and then use the Bluespec compiler to synthesize to Verilog. This gives us the ability to emit high-level constructs, generate human-readable output, and allows us to leverage optimizations implemented in the Bluespec compiler. However, our technique is not specific to Bluespec.

2. ARCHITECTURAL CONTRACT

Unlike interfaces in software, which are procedure calls, raw hardware interfaces are simply wire-to-wire connections. High-level synthesis languages sometimes embed a handshake protocol into the interface. For example, Bluespec associates a ready and an enable signal with each method. The caller must verify that the ready signal is asserted and then assert the enable signal (in the same cycle) to make a call. Thus, an interface specifies a *contract* between the "caller" and the "callee"[4]. Rather than imposing the burden of obeying such interface contracts on the user, the Bluespec scheduler actually *guarantees* that they will never be violated. This philosophy of embedding correctness guarantees in the compiler, rather than merely verifying that generated hardware is correct (in the restricted sense that contracts are not violated), is informally called *correct-by-construction.*

Such an interface contract is inherently *timed* because, the interface protocol is timed. In addition to the timed interface contract, most interfaces have a *semantic* contract associated with them. However, this contract is often only present in comment blocks and module names. For example, a module named `FIFO` is expected to have first-in-first-out behavior and a module named `RippleAdder` is expected to add its inputs. Such contracts are not enforced by a compiler, but verified a posteriori.

We define the *architecture* of a module as the *untimed semantic contract* between the user and the implementation of that module: the external user expects a certain functionality, and the internal implementation provides that functionality. On the other hand, the timing behavior of the module is entirely contained within the *microarchitecture.* Unlike an interface contract, which must be respected by the external user, an architectural contract must be respected by the internal implementation.

```
**architecture** Simple
  **memory** $rf = \texttt{Bit\#(32)} \times 8$
  **memory** $mem = \texttt{Bit\#(8)} \times 2^{32}$
  **instruction** inc $(r)$
    $rf[r] \Leftarrow rf[r] + 1$
  **instruction** loadbyte $(r, a)$
    $rf[r] \Leftarrow \text{signExtend}(mem[a])$
  **instruction** jmp $(pc, imm)$
    **return** $pc + imm$
  ...
```

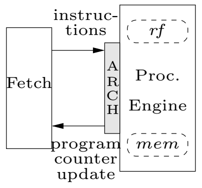


Figure 1: A simple architecture (pseudocode)

The user specifies this contract as a set of *instructions* operating on the *architecturally-visible state* (Fig. 1), such as the register file and the virtual memory in a typical processor. These are merely abstract notions of what the hardware is operating on, and don't necessarily have an associated counterpart in the implementation. For example, the virtual address space may be vastly greater than the total physical memory, and the architectural register file may only exist as a table of pointers, pointing into a larger physical register file.

An instruction is specified as a set of non-conflicting actions, where each action is a combinational update on the architectural state. Actions are said to conflict if they specify updates that cannot be implemented in parallel, e.g., updates to the same register. Such an instruction definition can be mechanically derived from a behavioral specification, like that used in instruction-set definitions. The behavioral specification could contain a loop, as long as it can be statically unrolled. An explicit architectural contract could also be used as a "run-time assertion" during simulation.

3. MICROARCHITECTURE

The specific way in which one implements an architecture is called the *microarchitecture.* Our language allows the user to specify the architecture and microarchitecture separately, and the compiler is responsible for generating hardware such that the architectural contract is not violated.

Although arbitrary microarchitectures cannot be specified in an architecture-independent fashion, for a large subset of microarchitectures, it is possible to do so. We focus on clocked synchronous hardware with *data-stationary control.*

Data-stationary control was first introduced to simplify microprogramming of inorder pipelines[11]. In a data-stationary pipeline, the instruction is first "decoded" into one or more microinstructions (μops). Each μop flows through the microarchitecture, meeting resources like read and write ports of register files, arithmetic units, memory ports, etc. Control information that tells each resource what to do is encoded in the μop. A resource may generate data, which then flows with the μop to be consumed by a downstream resource. Data-stationary pipelines are insensitive to the latency of individual resources, hence they greatly simplify microprogram generation over other control strategies. This concept is easily generalized to other kinds of microarchitectures, as long as (a) the decoding happens in the beginning, (b) each μop flows through the microarchitecture along with the data, and (c) a μop meets each resource *at most* once. We impose no other restrictions on the microarchitecture.

An architect visualizes such a microarchitecture as a *directed acyclic graph* with the resources as vertices and μops flowing along the edges (Fig. 3(a)). Implicit in this resource graph is the architectural control, which consists of:

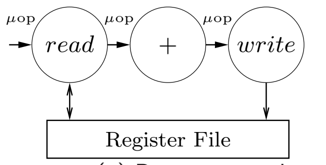
```
architecture Example
  memory $rf$ = Bit#(32) × 8
  instruction inc ($r_1$)
    $rf[r_1] \Leftarrow rf[r_1] + 1$
  instruction mov ($r_1, r_2$)
    $rf[r_1] \Leftarrow rf[r_2]$
```

(a) An example architecture (pseudocode)

```
module mkExample implements Example
  RegFile#(Bit#(3), Bit#(32)) $rf \leftarrow$ mkRegFile(8)
  RWire#(Bit#(3))             $w_1, w_2 \leftarrow$ mkRWire
  FIFO#(UOP)                  $p_1, p_2, p_3 \leftarrow$ mkPipelineFIFO
  method Action issue (UOP $uop$)
    $p_1$.enq($uop$)
  let $r$ = Valid($p_1$.first().read_in())
  rule stage1 when $r \neq w_1$ and $r \neq w_2$
    let $uop = p_1$.first()
    $uop = uop$.read()
    $p_2$.enq($uop$)
    $p_1$.deq()
  rule stage2
    $w_1$ := $p_2$.first().write_in_addr()
    $p_3$.enq($p_2$.first().add())
    $p_2$.deq()
  rule stage3
    $w_2$ := $p_3$.first().write_in_addr()
    $p_3$.first().write()
    $p_3$.deq()
```

(b) An example microarchitecture (Bluespec pseudocode)

Figure 2: Architecture & Microarchitecture



```
function RC($r_1, r_2$)
  let $x = rf[\square]$
  let $y = \square + \square$
  $rf[\square] \Leftarrow \square$
end function
```

(a) Resource graph expressed as a program

```
function RC($r_1, r_2, \alpha, \beta, \gamma, \delta, \varepsilon$)
  let $x = rf[\alpha(r_1, r_2)]$
  let $y = \beta(r_1, r_2, x) + \gamma(r_1, r_2, x)$
  $rf[\delta(r_1, r_2, x, y)] \Leftarrow \varepsilon(r_1, r_2, x, y)$
end function
```

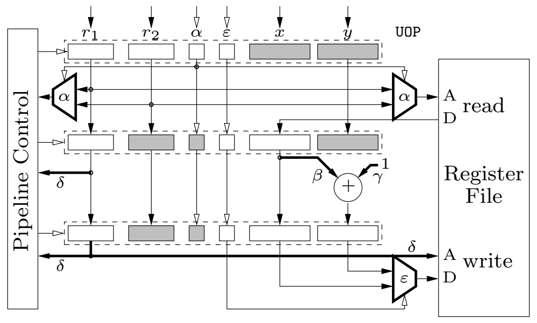
(b) Resource Configuration: unknowns replaced by labels (α–ε). r_1, r_2 are fields in the instruction.

inc r_1	mov (r_1, r_2)
$\alpha(r_1, r_2) \mapsto r_1$	$\alpha(r_1, r_2) \mapsto r_2$
$\beta(r_1, r_2, x) \mapsto x$	$\beta(r_1, r_2, x) \mapsto ?$
$\gamma(r_1, r_2, x) \mapsto 1$	$\gamma(r_1, r_2, x) \mapsto ?$
$\delta(r_1, r_2, x, y) \mapsto r_1$	$\delta(r_1, r_2, x, y) \mapsto r_1$
$\varepsilon(r_1, r_2, x, y) \mapsto y$	$\varepsilon(r_1, r_2, x, y) \mapsto x$

(c) Label bindings ('?' means unconstrained)

$\alpha(r_1, r_2) \mapsto r_1$ **or** r_2	$\triangleright$ 2 choices $\rightarrow$ 1 bit
$\beta(r_1, r_2, x) \mapsto x$	$\triangleright$ 1 choice $\rightarrow$ 0 bits
$\gamma(r_1, r_2, x) \mapsto 1$	$\triangleright$ 1 choice $\rightarrow$ 0 bits
$\delta(r_1, r_2, x, y) \mapsto r_1$	$\triangleright$ 1 choice $\rightarrow$ 0 bits
$\varepsilon(r_1, r_2, x, y) \mapsto y$ **or** x	$\triangleright$ 2 choices $\rightarrow$ 1 bit
	$\triangleright$ Total 2 bits

(d) Bits required to encode labels in the μop



(e) Synthesized hardware (microarchitectural control is not shown). Shaded hardware can be optimized away. Thick lines show the generated architectural control.

Figure 3: Resource configuration to hardware

(a) the logic for decoding an instruction into a μop sequence, (b) the encoding of the μop, (c) hardware needed to mux the right data to each resource, and (d) control needed to route each μop to the appropriate resources. On the other hand, microarchitectural control is visualized as if superimposed over the resource graph. This includes pipeline registers, issue policies, completion buffers—micro-architectural entities that affect timing. Our goal is that the compiler will generate all architectural control, and let the user specify arbitrary microarchitectural control.

3.1 Specifying microarchitectures

Our technique allows the user to specify the microarchitecture using arbitrary Bluespec constructs, but prevents access to the implementation of the μop.

The μop is encapsulated into an entity called the UOP. The user is expected to primarily operate on whole UOPs. This corresponds very strongly to the resource graph, where *not values but entire µops flow along the edges* (Fig. 3(a)). The user is required to specify the implementation of resources like register read/write ports, adders, dividers, floating point units, memory ports etc.

The internal structure of the UOP is determined by the compiler, and is hidden from the user. Both the control information and the data in the UOP are only exposed via public methods. (It will help here to think of UOP as a C++ class with private data and public methods.) The user may call these public methods from arbitrary Bluespec code.

Fig. 2(a) and Fig. 2(b) show the architecture and microarchitecture of a very simple processor. The architecture only specifies two instructions and a register file. The microarchitecture instantiates a register file and three pipeline registers (p_1, p_2, p_3). The method issue simply registers the input UOP into p_1. Rule stage1 dequeues the UOP in p_1, calls its read method and enqueues it into p_2. The call to the read method of the UOP causes the appropriate value to be read from the register file and logically appended to the UOP. Similarly, rules stage2 and stage3 call the add and write methods to perform the addition and write to the register file.

In addition, stage1 needs to stall whenever the its read-register conflicts with the write-registers of the following two stages. This computation is done in the **when** clause of stage1. These values are obtained by calling the read_in method of the UOP in p_1 and and the write_in_addr methods of the UOPs in p_2 and p_3.

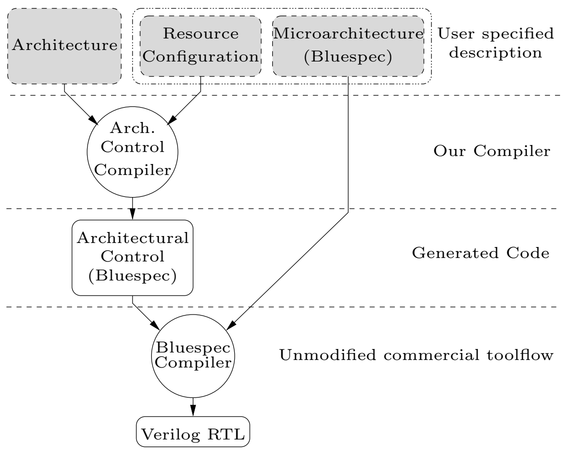

Architecture
Resource Configuration
Microarchitecture (Bluespec)
User specified description
Arch. Control Compiler
Our Compiler
Architectural Control (Bluespec)
Generated Code
Bluespec Compiler
Unmodified commercial toolflow
Verilog RTL

Figure 4: Our toolflow

3.2 Resource Configuration

In addition to the microarchitecture specified using Bluespec, our compiler requires the user to specify a resource configuration that summarizes the functionality of the microarchitecture. This program is sufficient (along with the architecture) to generate architectural control. Our toolflow is shown in Fig. 4. Currently, the user must ensure that the microarchitecture and resource configuration are consistent.

The resource graph captures what we call the "functionality" of the microarchitecture, and crucially, it can be expressed as a sequential program (as opposed to a concurrent program). For example, in Fig. 3(a), the first resource simply reads from the register file, the second does an addition, and the third writes to the register file. The program expresses the same information. The inputs to each resource are shown as unknowns ($\square$).

These unknowns must be determined from whatever information is available in the μop at that point. A value is only available at a resource-input, if there is a path from that value to the resource in the resource graph. In Fig. 3(a), the read index into the register file must be determined from the set $\{r_1, r_2\}$, operands to $+$ must be determined from $\{r_1, r_2, x\}$, and the write index/data to the register file must be determined from $\{r_1, r_2, x, y\}$.

This information is made explicit in Fig. 3(b). Here, α–ε are abstract functions that generate the appropriate value to feed the resource; we call these *labels*. Typically, a label just chooses from one of its arguments. However, it could also return a constant, sign-extend a value, bit-concatenate two values, etc. Hence, we model labels as abstract functions that compute the appropriate data for each resource. This program with explicit labels, along with the definition of each resource, is called the *resource configuration*.

The resource configuration could be inferred from the microarchitecture specified by the user. However, doing so would require our compiler to understand the semantics of Bluespec. Instead, we require the user to specify it manually. This has the added benefit of allowing the user to constrain the specified configuration, e.g., the user can fix a resource input to be a constant rather than a label, use the same label in multiple places, etc. This gives the user some additional control over the generated μops.

An instruction can be emulated by binding each label to some function; e.g., the user can easily verify that by substituting the bindings of Fig. 3(c) into the resource configuration RC, the two instructions of Fig. 2(a) can be emulated. In §4, we will describe how our compiler determines the bindings needed to emulate an instruction.

3.3 Synthesis

Each label is translated to hardware at the corresponding resource-input. Since a label may be bound to different functions to emulate different instructions, multiple functions may be synthesized at a resource-input. In this case, the μop would contain control bits to choose the appropriate binding, and a mux would be synthesized at the resource-input. Fig. 3(d) shows how the compiler calculates the number of bits required in the μop to implement the two instructions of Fig. 3(c).

Since the only input to a microarchitecture is the μop, the μop must be initialized to contain all the information contained in the arguments of the resource configuration (instruction fields and labels). As the μop flows through the system, it meets resources that logically append data to it. Thus, the size and structure of the μop depend on where it is in the microarchitecture.

However, we implement the μop as a single struct `UOP`, which reserves space not only for all the instruction fields and labels, but also for the outputs of each resource. This makes `UOP` simple, but rather large in size. This overhead is easily mitigated by downstream synthesis tools which can iteratively remove registers and wires (a) which always have the same value (constant propagation), or (b) whose values are unused (dead-code elimination). In addition, retiming can help move muxes across registers.

Fig. 3(e) shows the result of synthesizing the specifications of Fig. 2 & 3(b). The array of horizontal rectangles represents a `UOP`. The shaded rectangles represent redundant hardware that can be easily removed by downstream synthesis tools. (For clarity, we haven't shown microarchitectural control, in particular the pipelining and stalling logic, which must be specified by the user). The thick lines represent architectural control that is synthesized by the compiler. Labels α and ε are synthesized to multiplexers and have corresponding control bits in `UOP`, whereas β, γ and δ are synthesized to wires. The downstream synthesis tool could further save registers by retiming the mux generated for ε across the pipeline register.

4. GENERATION OF ARCH. CONTROL

To map each instruction to one or more μops, we need to generate the set of labels that make the resource configuration program equivalent to the instruction. We shall formulate this problem of generating the set of labels in terms of the known problem of *equational unification*.

4.1 Equational Unification

A *term* is defined recursively, either as a *variable*, a *symbol*, or a *function* applied to one or more terms. Here, a symbol is simply an identifier with an associated meaning, whereas a variable only serves as a placeholder at which another term can be substituted.

The goal of *unification* is to solve a set of equations in terms ($\{s \overset{?}{=} t, \ldots\}$), more specifically, to find a variable-to-term mapping which makes a set of pairs of terms simultaneously equal. In *syntactic* unification, equality is interpreted structurally—the meaning of a symbol is not used to decide equality, e.g., $x+y$ and $y+x$ are not syntactically equal. On the other hand, in *equational unification* or E-unification, equality is determined using a set of equations E specifying axioms like commutativity, associativity, semantics of if-then-else etc.

The algorithm for syntactic unification[2] essentially walks the trees formed by the two terms, verifying if the symbols on each side are exactly the same until it reaches a variable on one of the terms. In this case, it simply creates a mapping from the variable in one term to the corresponding subterm in the other term, making sure that it does not contradict any of the existing mappings. The algorithm terminates without a solution if it encounters a symbol clash or an inconsistent variable-term mapping. A variant of this basic algorithm has linear-in-time asymptotic complexity.

The strategy for general E-unification is similar, however, one can no longer require the two symbols on either side to be structurally identical. Instead, the algorithm must use the axioms in set E to modify an equation $s \overset{?}{=} t$ such that the above approach does succeed. The procedure for "using" an axiom $l = r \in E$, is quite naïve: pick any non-variable subterm u present in either s or t, and replace it with r; then, add the additional constraint $l \overset{?}{=} u$ to the set of equations to unify. This step is called *lazy paramodulation*[7]. The algorithm nondeterministically applies lazy paramodulation at some subterm using some equation in E, repeatedly, and then proceeds to syntactically unify the result.

The algorithm implies a search tree in which the root node is the problem, edges are applications of the lazy paramodulation step and leaf nodes are solutions. This algorithm is *complete*—i.e., every unifier is reachable on the search tree. The search tree is finitely-branching but it has infinite paths. Thus, a breadth-first search will exhaustively enumerate each solution, however it may never terminate. Thus, this algorithm is NP-hard in general.

4.2 Problem Formulation

An instruction is just a set of combinational updates on architectural state, hence, it can be represented as a term. The arguments to the instruction are represented as constant symbols, and there are no variables in the instruction-term. Since the resource configuration is acyclic for a data-stationary microarchitecture, it can also be expressed as a term, with labels represented as variables.

To map an instruction to a *single* μop, we need to find a label-to-term mapping that makes the resource configuration and instruction terms equal. In other words, we need to E-unify the two terms. (Our problem is somewhat simpler, since there are no variables on one of the two terms.) The axioms E represent not only the semantics of the symbols in each term, but also equations that allow a label to be replaced by one of many values. For example, a label may be replaced by a field in the μop, a constant, etc.

An exhaustive solution set can be obtained by a breadth-first traversal of the search tree, but this is expensive, and the search may never terminate. Fortunately, like most synthesis problems, we only care about optimality up to a certain constraint: more solutions don't have to be inspected once a good-enough solution is found.

Instead of minimizing the number of control bits in the μop and the number of multiplexers at the resource-inputs, we incrementally build a heuristic for choosing a pseudo-best solution. To limit compiler run-time, we simply use the number of μops as a coarse-grained cost metric.

Exhaustive enumeration also complicates heuristics. For example, since $+$ is known to be commutative, if the register file has two read and two write ports, a two-register add instruction can be implemented in eight ways, but all solutions are only cosmetically different. For instructions spanning multiple μops, we have seen hundreds of practically equivalent solutions. Moreover, equivalent solutions tend to be generated together, thereby delaying a better solution from being evaluated. A probabilistic algorithm may alleviate this problem, but knowing when to stop remains difficult, particularly when no solution can be generated.

To investigate the cost of sacrificing optimality, algorithm in a severely restricted form that does not allow paramodulation to be applied at arbitrary subterms, but only at the top-level term (at each step of the algorithm). Moreover, we require that l and u above unify syntactically (thereby making the paramodulation eager rather than lazy). These restrictions have the combined effect that multiple axioms cannot be used at the same subterm. The resulting algorithm is not complete, and to make the algorithm work for common cases, we had to add additional axioms to our set of equations E; this was done by trivially composing pairs of existing axioms.

One way that is guaranteed to prune the search space is to enforce static type-checking. This allows the compiler to prove that the labels can never take certain values. For example, in Fig. 3(b), if the type of r_1 and r_2 is `RegID` and the type of y is `Bit#(32)`, then we can safely conclude that the function β cannot return the value r_1 or r_2, thereby making the problem smaller. However, our prototype compiler does not implement static types yet.

4.3 Multiple μops

When an instruction is translated to multiple μops, we may need to pass data between μops. To do so, we need some notion of a "temporary register". Our language allows the user to add a temporary register file to the microarchitecture just like a regular register file. The compiler knows that it can overwrite the values of a temporary register file, since it is not part of the architectural state. We provide two algorithms to map an instruction to multiple μops:

(a) If unification fails to find a single-μop solution, we compose two instances of the resource configuration together and attempt to unify again. This composition involves collapsing temporary register write-read pairs. If that fails, we try three instances, etc. until we succeed or time out.

(b) Another technique is to add an additional variable to the resource configuration to capture residual actions, whenever the algorithm only manages to partially unify the two terms. Similarly, residual values can be mapped to temporaries. These residual actions and values are then iteratively re-mapped onto the resource configuration to get the previous μops. (The μops are generated in reverse order.)

5. IMPLEMENTATION

The instructions in the architectural specification are specified as a set of Bluespec functions. For simplicity, we let Bluespec decide the bit-encoding of each instruction. For an extant architecture like x86, the user would have to manually translate from the actual instruction format to the bit-encoding expected by Bluespec to maintain bit-level compatibility. The resource configuration is specified using a custom Haskell eDSL (embedded domain-specific language).

Instruction	Num of μops	
	Actual	Best
`add eax, 1234`	1	1
`add eax, (eax+ecx)`	1	1
`add (eax*2+4), edx`	1	1
`add ecx, (eax)`	1	1
`push 0x1234`	1	1
`push (0x1234)`	2	1
`pop ebp`	2	2
`call 0x1234`	2	1
`xchg (eax), ebx`	1	1
`xchg eax, ebx`	–	–
`movs`	4	3

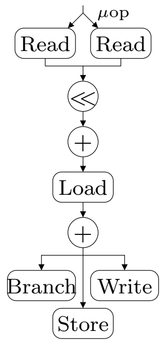


Figure 5: Comparison of generated μops to the best possible number; '–' means not implementable; value in ( ) represents a memory address

We believe the language is powerful enough to conveniently specify any data-stationary microarchitecture.

Our compiler emits Bluespec to (a) define the `UOP` struct, (b) decode the instruction into a list of `UOP`s, (c) wrap resource implementations to operate on `UOP`s, and (d) define functions that get synthesized to hardware at resource-inputs. We then use the standard Bluespec toolflow to compile the emitted Bluespec with the specified microarchitecture. As long as the user doesn't try to access the `UOP` private data, the user-written code and compiler-generated code is always compatible.

Fig. 5 shows a few representative x86-like instructions being mapped onto a resource graph. The number of μops generated are compared to the optimal number. Condition codes were not modeled. The compiler was constrained to run for less than 20 seconds per instruction, with the exception of `movs` for which it was given 100 seconds. (`movs` is a complex instruction that performs one memory-to-memory move and three register decrements.) The compiler correctly detected that the two-register `xchg` (register swap) cannot be implemented on this resource configuration (due to the lack of temporary registers).

6. RELATED WORK

Several synthesizable architecture/microarchitecture description languages exist[14], but none that we know of allow microarchitectures to be specified in a powerful non-behavioral high-level synthesis language.

MIMOLA[13] is an early '80s behavioral synthesis language that allows the instruction set to be separated from the target declaration, which specifies a set of one-way rules. The rules match patterns in the instruction-graph and replace them with corresponding patterns in the hardware. Since the technique operates over the entire microarchitectural state, it does not scale to modern microarchitectures.

User-guided HLS[1] attempts to generate hardware accelerators from software by automatically mapping the software onto a "draft data-path" (similar to our resource configuration), by using a coarse-grained and fine-grained scheduler. However, arbitrary microarchitectures cannot be supported.

Generalized instruction selection[17] uses one-way matching (a restricted form of unification) to compile software onto a specified instruction set. The Denali super-optimizer[10] uses E-graph matching and a SAT solver to compile small software kernels to provably optimal machine code. Similar techniques could be used to compile an instruction onto our resource configuration.

NISC[16] avoids the ISA abstraction altogether by mapping from software directly onto a specified data-path, and storing the μops in program memory. Tensilica[8] allows the user to add instructions and data-path elements to the Xtensa family of processors.

7. CONCLUSION

We have shown that it is possible not only to decouple architectural contracts from microarchitectures, but also to enforce them. We have prototyped a simple compiler that uses E-unification to generate architectural control required to enforce the contract, and synthesizes the split description to hardware. We are currently exploring more efficient E-unification algorithms.

8. ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under grants 0615352, 0747438, and 0917158. We are greatly indebted to the anonymous reviewers for their comments and corrections.

9. REFERENCES

[1] I. Augé and F. Pétrot. User-Guided High Level Synthesis. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*. Springer Netherlands, 2008.
[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
[3] P. Coussy and A. Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008.
[4] N. Dave, M. C. Ng, M. Pellauer, and Arvind. A design flow based on modular refinement. In *MEMOCODE*, 2010.
[5] K. Ekanadham, J. H. Tseng, P. Pattnaik, A. Khan, M. Vijayaraghavan, and Arvind. A PowerPC Design for Architectural Research Prototyping. In *WARP*, 2009.
[6] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2), 2002.
[7] J. H. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2-3), 1989.
[8] R. Gonzalez. Xtensa: a configurable and extensible processor. *IEEE Micro*, 2000.
[9] D. Greaves and S. Singh. Designing application specific circuits with concurrent C# programs. In *MEMOCODE*, 2010.
[10] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *PLDI*, 2002.
[11] P. M. Kogge. The microprogramming of pipelined processors. In *ISCA*, 1977.
[12] S.-L. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh. An FPGA-based Pentium in a complete desktop system. In *FPGA*, 2007.
[13] P. Marwedel. A retargetable microcode generation system for a high-level microprogramming language. In *MICRO'81*.
[14] P. Mishra and N. Dutt. *Processor description languages*. Morgan Kaufmann Series in Systems on Silicon. 2008.
[15] R. S. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *MEMOCODE*, 2004.
[16] M. Reshadi, B. Gorjiara, and D. Gajski. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *ICCD*, 2005.
[17] T. Richards. *Generalized Instruction Selector Generation*. PhD thesis, University of Massachusetts Amherst, 2010.
[18] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS*, 1998.