PyPBS Design and Methodologies

Greg Hoover and Forrest Brewer
Department of Electrical and Computer Engineering
University of California, Santa Barbara

Abstract

This paper presents results on processor specification from a specialized high-level finite state machine (FSM) language. The language is an extension and enhancement of earlier production based specification (regular automata) work using modern software techniques of modularity, abstraction, and object orientation. A brief overview of the language, its synthesis technique, description methods and advantages are presented for a variety of common synchronous pipeline structures in the context of the design of a synthesizable instruction-stream parallel microcontroller.

1 Introduction

Growing complexity in digital circuit design has led to new abstraction techniques for logic and sequential design. Most of these techniques have addressed functional design and correctness, however, a few target high sequential complexity and control-dominated designs. Typical design abstractions enable hierarchy and construction reuse in the specification as well as compositional semantics which allow specification fragments to be composed in a meaningful way. Equally important in a high performance design flow is a mechanism to encourage efficient design specification in which simplicity of the specification implies simplicity of the designed hardware. This often overlooked issue is crucial if designer understanding is to be captured in the specification. For example, in design of a microprocessor, it is often necessary to create several signals which are nearly logically redundant as well as redundant control machine states to meet spatial and temporal constraints. In such designs, classical state and logic reductions mix functionally disjoint parts of the design and often lead to impractical results. This does not imply that no optimizations are useful, but that control design needs to model timing complexity as well as spatial locality in such applications.

The specification notion in PyPBS is very simple— all machine actions are conducted under the assumption of unit (not zero) cycle time and are enabled by recognition of the appropriate sequence of conditions. Each submachine is a named production and begins activity on receipt of an activation context. A production returns a valid context and performs actions only if the intervening sequence of inputs is recognized. A specification is a composition of productions based on the rules of extended regular automata. Thus, there is no limit on the number of parallel contexts which can be present in a machine or submachine. Further, there is no notion of implied data dependence among actions. The language focus is expressiveness of the specification limited by simplicity of hardware implementation. The rationale for these choices stem from the practical issues of constrained hardware design. In general, a designer must work within a rather restrictive set of sequential constraints— protocols of memories, previously constructed designs, interface specifications and the like. PyPBS allows the designer to develop designs within such constraints in an incremental and applicative way.

Experience in numerous design constructions shows that there are many behaviorally equivalent but not necessarily sequentially equivalent design alternatives that may have quite distinct timing and power constraints when mapped into target hardware. PyPBS promotes an incremental design refinement technique in which designer controlled changes are rapidly reflected in the resulting synthesis. This is aided by a powerful set of sequential primitives and an applicative synthesis technique that limits the scope of design circuit changes, even if the changes are very high up in the control hierarchy. This is possible because the synthesis process creates register-heavy, high performance designs. Targeted optimizations usually only decrease the implementation size, not the target performance.

This paper presents enhancements to the PBS language and compiler, including: 1) enhanced, orthogonal language constructs, 2) generalized compilation algorithm, 3) modular, object-oriented design framework, and 4) design specification choices illustrated via the design of a fine-grain, instruction-stream parallel, multi-threaded processor.

This paper is organized as follows: Section 2 summarizes previous work. Section 3 describes the structure and constructs of the new PyPBS language. Section 4 describes the compilation (translation) process. Specification design
strategies are presented in Section 5. Results for the multi-threaded microcontroller design are presented in Section 6, with future areas of work and conclusions presented in Section 7.

2 Relation to Prior Work

PyPBS belongs to a set of languages which impose synchronous timing behavior on their programs. There are several such languages, each with differing capabilities: State Charts [6] provides an efficient mechanism for state machine hierarchy specification, Estere [2] allows for finite state machine composition in imperative language form while the data-flow oriented languages Signal [1] and Lustre [4] allow declarative specification of sequencing. The general notion of a clock enabled variable in Signal and Lustre is similar to the notion of context in PyPBS, with the exception that there is no notion of variable assignment or of variable dependence in PyPBS. Thus, the substantive complexity of dependence following and related correctness procedures are missing. PyPBS attempts to mirror the minimal complexity construction of a set of timing signals which are used to control data-flow activity. It is thus similar to pure Estere and State-Charts in that functional activity is not modeled. In practice, correctness modeling can be prohibitively expensive for complex machines. Instead of making such behavior the default, PyPBS supports construction of static constraints or dynamic monitors [12] as required for validation and design debugging.

Previous work by Seawright [13] described a high-level synthesis system based on the hierarchical Production-Based Specification (PBS). The underlying abstraction model was extended regular automata directly translated to circuit form. It avoided representation explosion through direct construction from an abstract syntax tree (AST), avoiding the RE to NFA conversion. This approach was shown to produce favorable machine encoding when compared to existing graph-based techniques in terms of logic complexity and depth. Crews [5] showed that PBS could be used to perform sequential optimization and construct machines that exhibit synthesis complexity scaling in terms of the number of state bits rather than the number of states. His optimizations included simple heuristics on the synthesized circuit and relatively powerful RE-based simplification on the AST. Finally, Seawright [14] described a system for partitioning the AST to provide a bridge to conventional sequential synthesis techniques that was particularly useful for counter and timer applications. These techniques resulted in hardware machine implementations that were comparable or superior to those synthesized using conventional methods and which could be scaled to very large (thousands of bits) FSMs. This work was the progenitor of the Synopsys Protocol Compiler [11], and also of this work.

Berry and Gonthier [2] proposed Estere, a synchronous language for reactive programming. This language provides a means for non-deterministic machine description based on “reactive” recognition of inputs. From these independent reactions, a non-deterministic finite automata (NFA) model can be recognized and translated into a deterministic state graph. Modern Estere compilers avoid the potential explosion of the translation process and can produce high quality sequential hardware designs. In general, however, the bulk of Estere application has been in reactive software systems.

Both PBS and Estere provide a means for “safe” machine composition, which is required to create large designs. The notion of “safe” describes the property that a target automaton’s protocol is not changed by inclusion in a larger construction. In Estere, this is managed by enforcing a single point of control model within a control structure. This is familiar to software designers and makes for efficient translation of Estere code to machine instructions. In contrast, PBS use of regular automata is “safe” in the opposite sense: each submachine is assumed to be as general as any possible invocation sequence can make it. In other words, any PBS production is designed under the assumption that a new invocation may occur each cycle. No attempt is made to preserve the potentially lost points of control. This second view is a close match to the actual behavior of hardware designs. Thus, while Estere can be compiled into efficient hardware, many varieties of pipeline behavior are difficult to specify and synthesize efficiently. Estere, on the other hand excels in error handling and modal changes, which need relatively complex primitives in PyPBS.

The Estere/C (ECL) environment presented by Lavagno and Sentovich [10] provides a novel approach to integration of both control and data component specification. ECL allows the designer to quickly determine trade offs between varying hardware/software partitionings by splitting control-dominated, reactive parts from data-dominated parts. Control-oriented components are mapped to Estere while data-oriented components are mapped to C, requiring additionally generated “glue logic” for integration of the two. While ECL is highly flexible, allowing Estere specification, C specification, and the mixed ECL language, hardware synthesizable output is limited to Estere components. PyPBS is a similar extension language in that its actions are currently unit-time Verilog code fragments. It offers similar flexibility in the specification of synthesizable hardware systems.

Hoe and Arvind [7] proposed an operation-centric hardware abstraction model useful for describing systems that exhibit a high degree of concurrency. In this model, operations are specified as atomic actions, permitting each operation to be formulated as if the rest of the system were frozen. A recent extension of BlueSpec [8] proposes a forcing operator enabling sequential constraint of the output de-
sign. Despite similarity of application, PyPBS and Blue-
spec serve substantially different needs. PyPBS provides a
practical specification and synthesis strategy for extremely
complex sequential protocols and related systems. Much
of the power of BlueSpec is lost in such a scenario. On
the other hand, Bluespec allows practical reasoning about
complex operation dependency which is not supported in
PyPBS. In the middle ground, both languages have been
used to describe microcontrollers, with Bluespec providing
succinct emulation of Itanium scale designs. In contrast,
PyPBS is used in this paper to concisely design a multi-
stream processor with precisely defined interfaces, both to
external devices and to internal sequential peripherals.

The IBM/Accellera Formal Specification language
PSL/SUGAR [3] has become the basis for formal property
specification in System Verilog. This language contains ex-
tended regular expression assertions as well as CTL and
LTL model checking capabilities. Although PSL is not in-
tended for design implementation, the similarities show that
formal property checking can be done in this format. There
was a property checking extension of PBS in its Protocol
Compiler instantiation as well.

3 Specification Language

The PBS language is a union of a production-based
grammar and tagged, extended regular expressions, where
a production is effectively a named reference to a particular
regular expression. The tags provide a means for describ-
ing action clauses and thus FSM output. Productions, as a
form of HDL, provide efficient reuse of descriptive elements
and a hierarchical description topology. Extended regu-
lar expressions provide exponential compaction of the de-
scription and combinational expressiveness. Conventional
regular expressions are infix descriptions, strongly limiting
the structures which can be efficiently described. PyPBS
expands the basic regular expression operator set, providing
operators for sequential description and access to non-
local machine state, as well as operators that target ease of
use. The notion of sequential regular expressions has been
adopted by other specification systems [3] and provides a
compact representation of complex behavior that can be ef-
efficiently modeled and verified.

The remainder of this section describes the new PyPBS
language operators (Section 3.1) and the modular design
framework employed (Section 3.2).

3.1 Language Operators

PyPBS unifies the sequential and combinational opera-
tors of PBS, providing a single operator set for description
of all machine behavior. Figure 1 lists the PyPBS operators.

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>∼</td>
<td>complement</td>
</tr>
<tr>
<td>,</td>
<td>concatenation of events (left then right)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>∨</td>
<td>exclusive or / sequential repetition</td>
</tr>
<tr>
<td>∗</td>
<td>Kleene closure (zero or more)</td>
</tr>
<tr>
<td>+</td>
<td>one or more</td>
</tr>
<tr>
<td>#</td>
<td>fork</td>
</tr>
<tr>
<td>?:</td>
<td>if-then-else</td>
</tr>
<tr>
<td>[]</td>
<td>group</td>
</tr>
<tr>
<td>&lt;foo&gt;</td>
<td>throw</td>
</tr>
<tr>
<td>foo&gt;</td>
<td>catch</td>
</tr>
<tr>
<td>foo &lt;bar&gt;</td>
<td>named reference</td>
</tr>
</tbody>
</table>

Figure 1. PyPBS Language Operators

controller -> monitor# , load , run:
controller -> ( monitor | load) , run:
monitor -> ... , False;

Figure 2. Fork Operator

Extended regular expression operators, including sequen-
tial concatenation, provide the foundation for all machine
description. In addition to this set, PyPBS provides opera-
tors for forking control, accessing machine state via named
references, multi-way throw/catch, if-then-else, and logic
grouping. The newly added language operators provide a
means of describing behavior that would be cumbersome or
inefficient to describe otherwise.

Fork
The fork operator splits a concurrent context to two child
nodes. However, unlike binary operators, the return con-
text of one child is ignored. This allows a behavior to be
initiated without infix side effects. Figure 2 presents two
equivalent specifications for a generic controller. The ma-
chine behavior starts by activating the monitor specification,
loading the machine, and then running. The use of the fork
operator obviates the necessity for the monitor specifi-
cation to explicitly return ‘False’. The benefit is code that is
simpler to understand, simpler to optimize and simpler to
synthesize since we know the return is ‘don’t care’ by con-
struction.

If-Then-Else
The ternary if-then-else operator is a common construct
in many languages and its functionality is self-evident.
Though its functionality can be duplicated using the basic
regular expression operators, the shorthand notation allows
code to be more easily written and understood. In contrast
to other PyPBS operators, ITE clauses apply their functionality to the input context of their child expressions rather than their output context. This prevents undesired output that stems from evaluation of child productions.

**Grouping**
The grouping meta operator provides a means of timing/area tradeoff as well as providing unambiguous combinational/sequential operator discrimination. By grouping a production, the designer is explicitly specifying that PyPBS should construct the logic context as a single bit. By default, each production terminal is encoded as a one-hot NFA state bit. By grouping a set of productions, the logic is composed and internal sequentialization is lost. The cost of such compression is larger fan-in scope for the control signals (potentially lowering controller performance). In practice, grouping brackets can be thought of as timing fences, placing the point at which the productions will be clocked. This provides an effective mechanism for local retiming adjustments.

**Throw/Catch**
Multi-way throw is similar in concept to “goto”, providing explicit transfer of control to another part of the machine description. Unlike “goto” however, PyPBS throws return True immediately allowing subsequent specification to continue uninterrupted, and allow multiple throw points to correspond to a single target. Multi-way catch expands this functionality, allowing a throw to activate multiple points of control in the machine description. These operators enable non-local description of machine behavior, convergence, and limited recursion. Used judiciously, throw/catch provides an efficient means of avoiding infix constraints in the regular automata.

The generic controller specification in Figure 3 illustrates several uses of the throw/catch operators. In the specification, a behavioral monitor runs in parallel to the desired machine functionality. The main event loop is specified using a tail recursive strategy that is initiated and maintained via a multi-way throw. Behavioral and input errors are routed from the monitor and read productions to a dedicated error handler and the generic cleanup routine via multi-way throw/catch. Though machines utilizing the throw/catch capabilities of PyPBS can often be described without these faculties, throw/catch tends to provide more compact and easily written specifications, as well as a reduction in the cost of controller implementation.

**References**
PyPBS provides a powerful mechanism for inspecting concurrent machine state through the use of production references. Unlike most control synthesizers in which access to state information is assumed, PyPBS only assumes access to local context by default. This assumption leads to substantial orthogonality in machine construction, but could lead to very redundant design in some cases. The reference mechanism provides designer access to control information from any context in the machine namespace, at the cost of additional Mealy dependence. In this way, the tradeoff of whether to use ‘far’ information in local state is made apparent to the designer. This mechanism is the sole means by which PBS designs can specify looping combinational control logic. (Similar issues occur in Esterel and in any Synchronous language). Debugging and modular design (Section 3.2) are enhanced by references and the ability to expose these signals as machine outputs during the debugging process.

### 3.2 Modular Framework

PyPBS borrows its modular framework from the Python programming language. Modules are represented by PyPBS specification files, with module names being derived from their respective file name. As shown in Figure 3, machines are named and form an encapsulated design unit. The PyPBS internal organization of a design is depicted in Figure 4, where a top-level design object maintains the set of modules used in the design, and each respective module retains the set of module declarations and machines that it specifies.

Modules and machines retain unique namespaces for declared objects, enforcing scoping rules on the construction. External declared objects can be referenced by their corresponding module.machinemodule object path, providing specification compaction through reuse of common object declarations. This reuse is particularly useful as it applies to action declarations which can be shared among multiple machine specifications. Resolution of objects proceeds in a bottom-up fashion by searching the namespace of the local machine followed by the enclosing module. Fully qualified objects are identified directly by traversing their declared path from the top of the design hierarchy.

Including external modules via the import declaration allows local machine specifications to access externally declared objects or compose external machines into a local...
Design

Module foo
Declarations
Machine bar

Figure 4. Modular Framework

machine description. Section 4.3 describes the process of modular composition of machines. Alternatively, modular design facilitates the ability to design systems built by integrating smaller component machines. Modularity of both PyPBS input and output allow combination of machines using minimal glue logic, which is easily added given that PyPBS outputs synthesizable Verilog HDL. Inter-module communication is specifiable in multiple ways, but is aided by the ability to create named references to expressions and explicitly expose their values as outputs of the machine. Exposed signals retain their declared name, allowing identification and connection to serve a variety of purposes. The ability to identify internal machine signals also benefits debugging.

4 Compilation

PyPBS compilation is a multi-step process requiring several levels of translation from a high-level specification to a synthesizable circuit level finite state machine (FSM). The compilation process makes extensive use of an abstract syntax tree (AST) dubbed the production directed acyclic graph (DAG). Details of the production DAG are presented in Section 4.1. The synthesized machine behavior is generated directly from the production DAG in a single post-order traversal. The semantics of this build traversal are described in Section 4.2.

Binary decision diagrams (BDD) are used throughout the compilation process for generation of circuit level logic functions. BDDs provide a canonical logic representation, making equivalent function recognition trivial and providing exponential compression of logic truth tables. BDDs also allow construction complexity proportional to circuit size, rather than the number of implied logic function cubes or sequential state complexity as in conventional state listing techniques. Potential BDD logic explosion is avoided by choosing a family of composition rules and BDD ordering that inherently result in small BDDs for small specifications. In general, the BDDs are fully composed for combinatorial logic, but not for non-local sequential logic such as throw/catch. Seawright [13] showed that BDD size was polynomial (nearly linear) for regular automata.

The PyPBS modular framework (Section 3.2) provides the basis for modular designs to be integrated using either separate compilation or machine composition. In a system built from component specifications, it is possible to compile each component independently and integrate them using glue logic. The resulting ensemble of machines is representative of the desired system behavior. Alternatively, modular composition provides the ability to integrate component specifications into a single machine as described in Section 4.3.

4.1 Production DAG

PBS introduced the production DAG [5], an AST-like intermediary format, used for optimization and construction of the circuit level implementation. The use of regular expressions as the basic description language allows PBS productions to be translated into ASTs in a straightforward manner. The production DAG is the union of these production ASTs, as formed through a traversal starting at a production deemed to be the ‘top’ of the machine description, as illustrated in Figure 5. The ability to reuse productions in a specification means that the resulting syntax graph is often a DAG rather than a tree. Cyclic logic detection ensures that these productions are not used in a manner inconsistent with PBS’s abilities. Representing the high-level description as a DAG has several advantages: 1) it is possible to specify completely deterministic automata without use of traditional deterministic models, 2) there exist direct gate level implementations which can be logarithmically smaller than the number of states, and 3) the structure is amenable to optimization.

The notion of context was introduced as a unified model for construction and interpretation of machine behavior in PBS. This idea follows directly from automata based modeling where the state of an NFA can be seen as the combination of active nodes. In PBS, these active nodes are said to have an active context, which they conditionally pass to subsequent nodes in the automata based on acceptance of their own functionality. The notion of context is central to the PBS synthesis strategy and is representative of the activation signal used to trigger execution of a submachine in the current cycle. It is the analog of an active bit of state in a conventional FSM model. Since the context is treated as an NFA state in its own right, such bits naturally compose
Figure 5. Production DAG
to a deterministic machine state. The acceptance or output context of a node in the production DAG then becomes the conjunction of its input context and logic function.

Constructed machine behavior follows from a post-order traversal of the production DAG as described in Section 4.2. The notion of context allows any subgraph of the production DAG to be decomposed and synthesized independently provided there exists a way to compose the input and output context signals of the submachine. BDDs provide the mechanism for this composition, allowing the construction algorithm to be local and fast, yet preserve a number of key sequential machine optimizations.

PBS makes use of operator-specific nodes in the production DAG, catering the build process uniquely for each composition operator. This creates substantial difficulties in DAG-level optimization, limiting the efficiency and generality of code restructuring at this level. PyPBS generalizes nodes in the production DAG, using four types of nodes to represent terminals, unary, binary, and ternary functions. Each node contains a prescription or template BDD representing the logic function that it implements. The generalized build algorithm (Section 4.2) constructs the machine behavior from these component BDDs. Optimization properties of these templated operators can easily be determined dynamically, greatly simplifying the complexity of expression level optimization.

4.2 Build

PyPBS machine constructions proceeds from a post-order traversal of the production DAG (Section 4.1). Nodes in the DAG contain BDD prescriptions that are representative of the logic function for that node. Logic prescriptions define the relationship between input and output contexts to and from parent and child nodes. This format allows a reduced set of BDD composition operations to describe complex relationships, such as Kleene closure. Composition of these BDD snippets is illustrated in Figure 6, with sequential constraints imposed by the notion of input and output context signals.

The unification of combinational and sequential operators as described in Section 3.1 has no bearing on the sequential semantics of the PBS construction algorithm. The notion of context coupled with the generality of the PyPBS build algorithm unifies the construction of all types of nodes, obviating the necessity for distinguished combinational and sequential logic specification.

Leaf or terminal nodes represent latched inputs to the machine and are the source of sequentiality in the construction. Composing the node’s input context with its corresponding machine input signal allows the latch to represent the sequential state of the machine in addition to the input. These latched inputs comprise the state bits of the generated machine and are used to separate events in time, controlling BDD and hence circuit size.

Grouping causes the construction algorithm to add a state bit for the functionality of the group rather than each terminal. This obviates additional combinational operators and reduces the number of state bits necessary for representing the desired machine behavior. In some cases this may lead to excessive fanin for state bit logic functions, resulting in performance degradation.

Ambiguous or circular logic constructions are flagged during construction. These infrequent constructions can be caused by non-local observation and feedback of context information. Usually such constructions are not desired and can always be removed by restructuring the code. On occasion, these constructions are better circuits than any non-looping implementation, but play havoc with conventional synthesis.

4.3 Modular Composition

As described in Section 4.1, every subgraph of the production DAG is a complete machine. The use of BDDs allows each subgraph to be synthesized independently and composed according to node functionality. By expanding
on this ability, externally specified machines can be composed in place, effectively integrating component specifications into a single machine. Modular composition allows components to be developed and tested prior to integration into the ensemble machine.

Independent submachine construction allows multiple inclusion without repeated translation of submachine specifications by compiling dependent submachines using a subset of template variables. When submachines are to be composed into a local specification, the template for the submachine is instantiated by substituting the template variable set and input context with variables from the local machine set and context. This process leaves the template construction unaltered for subsequent inclusions. Successful submachine composition requires that construction of all dependent specifications be completed prior to compilation of the including machine. Determining a suitable compilation order is accomplished via machine dependency analysis and longest prefix removal.

5 Design Specification Strategies

Enhancements to the PyPBS language have created stylistic choices that were previously unavailable. These choices manifest in the form of unique strategies for high-level description suited for particular design parameters. Five design strategies are presented in the context of a pipelined processor description. It should be noted that the key issue here is context or signal distribution, not state minimization as the former has a much stronger constraining effect on the system performance.

5.1 Standard Pipeline

The strengths of machine description using an NFA-based model are easily shown for protocol-oriented descriptions. In Figure 7 a pipeline strategy is presented in which machine context flows naturally from one pipeline stage to the next. Each stage is throttled by the acceptance of its antecedent stage whose specification may contain any set of static delays. Stages are identified by a single production, the behavior of which may contain both sequential and combinational descriptions, spanning multiple cycles. Under normal conditions a pipeline stage executes every cycle, specification of which is shown using a constant input context to the fetch stage.

PyPBS generates synthesizable Verilog HDL requiring minimally four state bits to correctly describe this behavior. It should be noted, however, that any of the stage references could be (and usually are) complex machines themselves, requiring a larger number of state bits to correctly specify their behavior.

5.2 Standard Pipeline with Stall

While the standard pipeline presented in Section 5.1 is concise, its simplicity is often not exhibited in practical designs. It is often the case that processor instructions have non-static delay, requiring additional cycles to complete. Many high-performance designs handle these variations by extending pipeline length such that all instruction types are guaranteed to complete within the fixed pipeline. Extending pipeline length impacts performance and is infeasible in many systems.

An alternative method is to design the pipeline such that typical instructions complete in the fixed pipeline while remaining instructions cause the pipeline to stall. This strategy would allow the execution stage to complete instructions requiring additional cycles with prior stages stalling until completion. This behavior requires the ability to stall pipeline stages on a per-stage basis and is easily specified using PyPBS. Figure 8 presents one method for specifying pipeline operation given stall conditions. In the description, stall signals are possibly machine inputs or productions describing more complicated functionality.

Composition using the if-then-else operator guarantees that stall conditions are applied to the input context of the stage description. If this were specified using ‘and’ composition, stall conditions would not be recognized until the subsequent stage, generating possibly incorrect behavior. However, it is possible to use ‘and’ composition by relocating the stall behaviors a cycle sooner, causing stall conditions to be composed with the acceptance of the prior stage rather than the current stage. The ELLA [9] processor design utilizes this pipeline specification strategy distributing stall conditions in the description of each stage (Section 6).

Stalling pipeline stages can lead to precarious situations when trying to recover machine execution. Stall recovery describes the simultaneous activation of all pipeline stages and can be specified in several ways in PyPBS. As shown, each processor stage is proceeded by a stage that accepts only when the respective stall condition exists. This stage is transparent during normal operation and preserves local execution state under stall conditions. Alternative stall recovery specifications are possible using the PyPBS throw/catch operators but generally require additional design complexity and logic overhead. The specification can be synthesized with minimally seven bits of state.
5.3 Parallel Stage Pipeline Description

Figure 9 illustrates a specification strategy conceptually similar to Bluespec where events are specified independently without sequential constraints. While this specification style can lead to implementations requiring no state bits, adding sequential constraints to this construction would require complex logic descriptions in each stage resulting in significant overhead. Though shown using the throw/catch operators, an equivalent description could be created using the binary ‘or’ operator. This style demonstrates the flexibility of the PyPBS language for specification of designs of varying type.

5.4 Tail-Throttled Pipeline

Tail-throttling is a traditional design solution for controlling interlocking pipelines. In a tail-throttled pipeline the controller guards pipeline stage acceptance based on that of the next stage in the pipeline. This follows from the reasoning that if a pipeline stage accepts in the current cycle, it can process a new operation in the next cycle, and therefore its antecedent stage should also accept. This structure can be hard to describe using conventional specification techniques.

In PyPBS however, specification of tail-throttled structures is straightforward, and is presented in Figure 10. References provide each stage with immediate access to the state of its descendant in a clear and concise manner. The PyPBS compiler can recognize the relationship between stages as combinational and generate the implementation machine without unnecessary state bits.

5.5 Parallel Instruction Type Pipeline

Figure 11 shows the specification for a variable-length pipeline where the productions alu, load, store, and pc are specialized multi-cycle descriptions of the behavior for each of the corresponding functional units in the design. The three parallel pipelines can execute in either parallel or intermixed fashions. While this requires additional state bits to support the unique pipeline behaviors, it reduces the logic complexity required in expressing each stage, increasing performance. This strategy can be used to express processors with greater degree of parallelism such as superscalar and simultaneous multi-threaded (SMT) designs.

6 Results

ELLA, a fine-grained, multi-threaded microcontroller supporting Atmel’s AVR instruction set has been developed using the PyPBS specification tool [9]. The system is built around an extensible bus architecture allowing addition of custom peripheral components accessible via memory mapping. The current design supports eight instruction-interleaved threads, with no clear upper limit to the number of possible threads. PyPBS is used extensively throughout the design to specify all control elements including memory interfaces and datapath behavior. The datapath control specification amounts to 44 PyPBS productions(lines), with a total of nearly 6,500 lines of generated and written Verilog code for the design. ELLA targets a standard cell implementation in the TSMC 0.15μm process using Synopsys
execute -> [STALL_EXECUTE <STALL_EXECUTE> : exec_jump call branch | exec all | exec io skip | exec return | exec alu | exec load store | exec propagate pipeline | exec cleanup | FUSH_STORE { avr_actions.flush_store_action } ]
exec alu -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE & ALU.OP} | avr_actions.execute_alu_action ]
exec_load_store -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE & (STORE.OP | LOAD.OP)} | avr_actions.execute_load_store_action ]
exec_jump call branch -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE & PC.OP & (JUMP | CALL | RETURN | (BRANCH & BRANCH_ACCEPT))} ]
\hspace{1em} { avr_actions.execute_jump_call_branch_return_accept_action } [ \hspace{2em} ]
exec_call -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE & PC.OP & CALL} | avr_actions.execute_call_action ]
exec io skip -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE & PC.OP & SKIP & ALU.OP} | avr_actions.execute_io_skip_action ]
exec return -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE & PC.OP & RETURN} | avr_actions.execute_return_action ]
exec propagate pipeline -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE} | avr_actions.execute_propagate_pipeline_action ]
exec cleanup -> [\texttt{STALL_EXECUTE} & \texttt{FLUSH_STORE & LOAD.OP & STORE.OP} | avr_actions.execute_cleanup_action ]

Figure 12. ELLA Execute Stage Specification

Design Compiler for synthesis and Synopsys DesignWare implementations for applicable functional units. Synopsys generated timing estimates place machine performance above 290MHz using conservative timing models. Performance is currently limited by the data cache unit, showing potential for increased performance through cache redesign or removal.

Figure 12 highlights several features of the new PyPBS language through the specification of the execute stage of the ELLA pipeline. ELLA employs the standard pipeline with stall strategy as presented in Section 5.2. The main execute production is used to retain local activation during stall conditions and trigger specialized descriptions for each operation type during normal execution. Stall conditions are distributed among all operation-specific descriptions, but could easily be factored using if-then-else as previously shown. Each operation-specific description is independent of all others, typifying the incremental specification capabilities of PyPBS. Development of the ELLA processor proceeded by incrementally implementing each of the instruction types, a process that is generally infeasible using conventional techniques. The specification is made more readable by separate specification of actions as shown through reference to the avr_actions module.

7 Conclusion and Future Work

Improvements to the Production Based Specification model for high-level hardware design have been presented with design specification strategies for pipelined processor design using the PyPBS language. Enhancements to the language add flexibility and expressiveness to existing techniques allowing PyPBS to target a wider range of design types. Added features provide the designer with the ability to test variations on design specifications with short turn-around times and minimal effort. The translation process presents unique opportunities for optimization, partitioning, and retiming. PyPBS output integrates synthesizable Verilog control descriptions with corresponding Verilog data components, providing a unified output medium, while maintaining clear separation of control and data specification. PyPBS retains the strengths of PBS in terms of sequential expressiveness and adds language constructs, object-oriented design, and modular compilation, providing non-sequential description more efficiently than prior work.

System design enhancements are planned in the form of a graphical development environment, supporting hierarchical graphical views and point-and-click integration of PyPBS modules, as well as manual logic manipulation. PyPBS sequential operators for existence and constraint can be easily added allowing formal CTL or LTL assertion specification. These constraints can be used to create system monitors as was done by Oliveria [12], or can be used implicitly during synthesis to eliminate failing behavior sequences. Fault-tolerance methods are being explored for novel solutions to the single-event upset (SEU) model using limited redundancy and error detection and correction (EDC) techniques. Local image computation methods are being explored along with efficient sequential and combinatorial optimization techniques.

References


