An Exercise in High-level Architectural Description using a Synthesizable Subset of Term Rewriting Systems

Computation Structures Group Memo 403
October 27, 1997

James C. Hoe, Martin Rinard and Arvind

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft. Huachuca contract DABT63-95-C-0150.
An Exercise in High-level Architectural Description using a Synthesizable Subset of Term Rewriting Systems

James C. Hoe, Martin Rinard and Arvind

February 2, 1998

1 Introduction

This paper presents a study on the viability of using term rewriting systems (TRS) as an architectural description framework in computer-aided hardware synthesis. The purpose of this study is to guide the development of a new description language called Bluespec. We want Bluespec to offer the semantics and the high-level constructs of TRS's to concisely and accurately describe complicated hardware mechanisms and behaviors. However, we also want to suitably constrain the language so a description can be realistically synthesized.

This research is motivated by the possibility to dramatically reduce hardware design time through the elevation of hardware design abstractions and greater involvement of automated synthesis and verification tools. In the design flow on Figure 1, we envision an architect focusing the majority of his attention in producing an architectural specification using high-level constructs in Bluespec. A specification is fully debugged interactively at this high level of abstraction using automatically generated simulators and computer-aided proof systems. With this specification as the blueprint, lower-level Bluespec descriptions of optimized implementation specifications can be generated by either automatic or human-driven means. The correctness of the implementation specifications can be formally verified against the initial architectural specification without a lengthy hit-and-miss validation process. Beyond this stage, human involvement is limited to high-level guidance, such as selecting one of the verified designs for the final stages of synthesis. The potential reduction in time, effort and risk will enable hardware solutions to become competitive in many applications that we currently endure in software as an engineering compromise.

To understand the issues in mapping TRS to hardware structures, we studied TRS descriptions of two processors. The first is a simple single-cycle, non-pipelined processor, and the other is a processor with machineries for register renaming and out-of-order execution. The two TRS descriptions were developed as part of a research into modeling hardware systems for formal verification[1, 4]. Two exercises were conducted by this study. In the first exercise, we adopted the syntax of Parallel Haskel (or $pH$)[3], a functional language with TRS semantics, to encode the TRS's of the two processor models. Based on this experience, we developed Bluespec$_{pH}$ from a stylized subset of $pH$'s syntax and semantics. By carefully constraining the features of the language, a Bluespec$_{pH}$ description not only uses a high-level of programming abstraction, but it also has a clear mapping to hardware structures. As an added benefit, a Bluespec$_{pH}$ description is a simulator of itself when executed as a $pH$ program. With newly gained understanding on language requirements, in a second exercise, we investigated Bluespec$_{C}$ which uses the syntax of $C[2]$, but provides its own TRS-based semantics. A Bluespec$_{C}$ description can also be interpreted directly as $C$ code for simulation, but a description must be augmented by additional wrapper $C$ codes to execute.
Bluespec is a work in progress. This paper documents the findings from two preliminary exercises to develop the Bluespec architectural description language on top of the syntax from two well-known languages. The emphasis of this paper is on the language requirements for efficient hardware description and synthesis. Section 2 first reviews prior work in modeling microprocessor architecture using TRS’s. Using the two TRS examples from Section 2, the subsequent sections identify issues encountered in our first study involving BluespecPH. Section 3 first discusses the description and extraction of hardware state elements in TRS’s. In Sections 4 thru 6, we then discuss the issues in describing combinational logic. In Section 7, we turn the attention to our experience with BluespecC. The paper concludes with a few closing remarks in Section 8. Complete BluespecPH and BluespecC sources produced in this study are included as appendices.

2 Modeling Hardware with TRS’s

The development of Bluespec grew out of the research by Arvind and Shen [1, 4] in modeling complex architecture mechanisms for verification\(^\text{1}\). Their modeling research is based on the formalism of term rewriting systems. A TRS consists of a set of terms and a set of rewriting rules. In the architectural context, terms usually represent the state of a system and the rules specify state transitions. The general structure of rewriting rules is as follows, where \(S\) and \(S'\) are terms, and \(p\) is a predicate:

\[
S \text{ if } p(S) \rightarrow S'
\]

Starting from an initial system term, rewriting rules are applied to transform the system configuration dynamically. A rule can be used to rewrite a term if the rule’s pattern, \(S\), matches the term or one of its sub-terms, and the corresponding predicate is true. The term resulting from the rule’s application is specified in \(S'\). If several rules are applicable, then any applicable rule can be applied.

TRS’s have been used extensively to define the operational semantics of programming languages. Their appeal for describing architectures stems from the fact that they can be used to model parallelism and non-deterministic behavior in a straightforward manner. Arvind and Shen have shown that TRS’s can be used to model concisely very complex architectural features and to prove important equivalences and behavioral properties. A dynamically scheduled processors with out-of-order and speculative instruction execution can be modeled in a TRS with less than 20 rules.

\(^{1}\)The material in this section is taken from [1] with authors’ permission
inst \equiv r := \text{Loadc}(v) \quad \text{Load-constant Instruction} \\
\quad | \quad r := \text{Loadpc} \quad \text{Load-program-counter Instruction} \\
\quad | \quad r := \text{Op}(r_1,r_2) \quad \text{Arithmetic-operation Instruction} \\
\quad | \quad Jz(r_1,r_2) \quad \text{Branch Instruction} \\
\quad | \quad r := \text{Load}(r_1) \quad \text{Load Instruction} \\
\quad | \quad \text{Store}(r_1, r_2) \quad \text{Store Instruction}

Figure 2: \mathcal{AX}, A Simple RISC Instruction Set. For the Load and Store instructions, register \( r_1 \) contains the data address; for the Jz instruction, register \( r_2 \) contains the branch target address.

To give insight to how TRS can be used in modeling hardware behavior, a highlight of two TRS's are presented. The first models a single-cycle, non-pipelined processor \( \mathcal{P}_B \), and the other models a dynamically scheduled processor with register renaming, \( \mathcal{P}_R \). Both processors implement \( \mathcal{AX} \), a simple RISC instruction set described in Figure 2. Refer to [1, 4] for a complete presentation of this material.

The state of the system is modeled by a processor and a memory. The memory consists of a set of cells with addresses and values. In the \( \mathcal{P}_B \) model, the processor state is described as a term \( \text{Proc}(pc, rf, prog) \), where \( pc, rf \) and \( prog \) represent the program counter, the register file, and the program memory, respectively. To give a flavor of how the rewriting rules can model processor execution, the rewriting rule that corresponds to the execution of an arithmetic instruction is given below:

\textbf{Op Rule}
\[ \text{Proc}(i, a, rf, prog) \quad \text{if} \quad \text{prog}[i] = r := \text{Op}(r_1,r_2) \]
\[ \rightarrow \quad \text{Proc}(i+1, r[f := v], \text{prog}) \quad \text{where} \quad v = \text{Op}(rf[r_1], rf[r_2]) \]

This rule is applicable to states in which the current instruction is a binary arithmetic operation such as addition. (\( \text{Op}(rf[r_1], rf[r_2]) \) means the value or the result of the operation applied to the two operands.) The rule updates the destination register in the register file so that it contains the result from performing operation on the two source registers. It also increments the program counter.

There is a separate rule to model the execution of each instruction type. The remaining rules for \( \mathcal{P}_B \) are the following:

\textbf{Loadc Rule}
\[ \text{Proc}(i, a, rf, prog) \quad \text{if} \quad \text{prog}[i] = r := \text{Loadc}(v) \]
\[ \rightarrow \quad \text{Proc}(i+1, r[f := v], \text{prog}) \]

\textbf{Loadpc Rule}
\[ \text{Proc}(i, a, rf, prog) \quad \text{if} \quad \text{prog}[i] = r := \text{Loadpc} \]
\[ \rightarrow \quad \text{Proc}(i+1, r[f := a], \text{prog}) \]

\textbf{Jz-Jump Rule}
\[ \text{Proc}(i, a, rf, prog) \quad \text{if} \quad \text{prog}[i] = \text{Jz}(r_1,r_2) \quad \text{and} \quad rf[r_1] = 0 \]
\[ \rightarrow \quad \text{Proc}(r[rf_2], r, \text{prog}) \]

\textbf{Jz-NoJump Rule}
\[ \text{Proc}(i, a, rf, prog) \quad \text{if} \quad \text{prog}[i] = \text{Jz}(r_1,r_2) \quad \text{and} \quad rf[r_1] \neq 0 \]
\[ \rightarrow \quad \text{Proc}(i+1, r, \text{prog}) \]

3
Load Rule
Sys(m, Proc(ia, rf, prog)) if prog[ia] = r := Load(r₁)

→ Sys(m, Proc(ia+1, rf[r := m[a]], prog)) where a = rf[r₁]

Store Rule
Sys(m, Proc(ia, rf, prog)) if prog[ia] = Store(r₁, r₂)

→ Sys(m[a := rf[r₁]], Proc(ia+1, rf, prog)) where a = rf[r₁]

In the \( P_R \) model, the processor implements register renaming and uses instruction template buffers, itbs, to hold the state of partially executed instructions. An instruction is assigned a register renaming tag at the time it is issued and the renaming tag is stored in the register file itself. The issued instruction is enqueued in the instruction template as described by the following \( P_R \)-Op-Issue rule:

\( P_R \)-Op-Issue Rule
Proc(ia, rf, itbs, prog) if prog[ia] = r := Op(r₁, r₂)

→ Proc(ia+1, rf[r := t], itbs \oplus ITB(ia, t := Op(rf[r₁], rf[r₂])), prog)

Notice the definition of processor terms has been extended to include itbs. An instruction template buffer contains an instruction in which each register name has been replaced by either its renaming tag or the corresponding value. Any instruction in itbs can be executed if all of its operands are available (there are some additional restrictions on memory access instructions). A natural consequence of register renaming is that instructions can be executed in a different order from the “program order”. itbs is typically maintained as an ordered queue. The queue is represented using the constructor ‘\( \oplus \)’, which is associative but not commutative. Initially, itbs is empty.

There are a total of 14 rules in the TRS for \( P_R \). These rules correspond to computing the result of an instruction, propagating the result to other instructions in itbs, and committing the result to the register file. The following rule for computing the result of an arithmetic instruction states that an arithmetic operation in itbs can be performed if both operands are available, i.e., tags for both operands have been replaced by values:

\( P_R \)-Op Rule
Proc(pc, rf, itbs₁ \oplus ITB(ia₁, t := Op(v₁, v₂)) \oplus itbs₂, prog)

→ Proc(pc, rf, itbs₁ \oplus ITB(ia₁, t := Op(v₁, v₂)) \oplus itbs₂, prog)

The effect of applying the above rule is that a tag in itbs is assigned a value. There are several other rules that also assign values to tags. The following two rules propagate the effect of these rules by forwarding the value of the tag to other instruction templates and the register that may contain this tag. The notation itbs₂[v/t] means that all instances of tag t in itbs₂ are replaced by value v.

\( P_R \)-Value-Forward Rule
Proc(pc, rf, itbs₁ \oplus ITB(ia₁, t := v) \oplus itbs₂, prog) if t \in itbs₂

→ Proc(pc, rf, itbs₁ \oplus ITB(ia₁, t := v) \oplus itbs₂[v/t], prog)

\( P_R \)-Value-Commit Rule
Proc(pc, rf, itbs₁ \oplus (ia₁: t := v) \oplus itbs₂, prog) if t = rf[r]

→ Proc(pc, rf[r := v], itbs₁ \oplus (ia₁: t := v) \oplus itbs₂, prog)

The following rule retires a renaming tag and frees the associated instruction buffer:

\( P_R \)-Value-Discard Rule
Proc(pc, rf, itbs₁ \oplus ITB(ia₁, t := -) \oplus itbs₂, prog) if t \notin rf, itbs₂

→ Proc(pc, rf, itbs₁ \oplus itbs₂, prog)
The equivalence between the $\mathcal{P}_S$ and $\mathcal{P}_R$ models can be proven by showing that if an initial term can get into a state using the $\mathcal{P}_S$ rules, then it can also get there using the $\mathcal{P}_R$ rules. However, the more interesting result is that $\mathcal{P}_S$ can simulate the state transitions of $\mathcal{P}_R$ as well. These types of theorems often provide a lot more insight into micro-architectures than the actual implementations.

3 Identifying State Elements from TRS’s

In the last section we have shown that TRS is capable of modeling the behavior of hardware mechanisms in a succinct and precise manner. However, not all TRS’s can be successfully mapped into a functionally equivalent hardware implementation. In general, emulating an arbitrary TRS can require infinite or dynamically varying number of state elements. Intuitively, the class of TRS’s that can be synthesized must resemble a finite state transition system where state elements in the system neither grow nor diminish during execution. Although asynchronous finite state machines more closely resemble TRS’s, for the sake of implementation, we will limit our discussion to mapping TRS’s into synchronous finite state machines (FSM’s).

One approach to identify state elements in a TRS is to construct a dependence graph whose nodes are leaf terms in the syntax tree of the TRS grammar. (In this case, we assume the TRS given for synthesis has a fixed and finite syntax tree.) A directed edge from node $t_i$ to node $t_j$ is added to the graph if there exists a rule $S$ \( p(S) \rightarrow S' \) such that $t_i \in S$ and $t_j \in S$, $t'_i$ and $t'_j$ are corresponding terms for $t_i$ and $t_j$ in $S'$, and either

\[
t'_j \neq t_j, \quad \text{and} \quad t_i \text{ is not a don't care in } P, \quad \text{or}
\]

$t_i$ is in the domain to compute $t'_j$.

If a node is not on any cyclic path in the resulting graph, then its corresponding term is only a combinational value. A valid state extraction can be constructed by selecting a sufficient number of nodes $n_1..n_n$ such that after removing all edges that enter these nodes, the graph is acyclic. If the corresponding terms $t_1..t_n$ are mapped to state elements, then the remaining terms can be combinational functions of $t_1..t_n$. A trivial choice is to select all the terms. An optimal choice would require some balance between minimizing the size of state elements and the combinational delay of the next-state logic. Instead of solving this involved multi-dimensional optimization during compilation, BlueSpec $\text{pH}$ gives the architect, who has greater insight into the design, the means to identify the state elements explicitly. If the architect fails to identify all state elements, the compiler can then issue a warning and proceed with a naive approach to identify the remaining state elements.

Under BlueSpec $\text{pH}$’s typing system (adopted from $\text{pH}$), a state element must be declared as a special mutable type identified by the type constructor keyword MCell. For example:

\[
\text{type} \quad \text{Pc} \quad = \quad \text{MCell} \quad \text{InstAddress}
\]
\[
\text{type} \quad \text{InstAddress} \quad = \quad \text{Int}
\]

In the declaration above, Pc is the type for a state element containing an InstAddress. In contrast, InstAddress is simply an integer type. Instantiating a term as Pc type explicitly identifies the term as a state element. Whereas, declaring a term as InstAddress leaves the decision open to the compiler. Besides requiring the special MCell type constructor in its declaration, once declared as a MCell type, a state term can only be referenced using the special operators mStore and mFetch.

An array of registers can be declared as:

\[
\text{data} \quad \text{RegFile} \quad = \quad \text{Regfile} \quad (\text{Array} \ \text{Int} \ \text{RegCell})
\]
\[
\text{data} \quad \text{RegCell} \quad = \quad \text{MCell} \quad \text{Int}
\]
1 opSubRule (Sys memory proc prog) =
2     let (Proc pc rf) = proc
3     in
4         if (imemload prog (mFetch pc)) ==
5             RR Sub rd r1 r2 then
6                 let
7                     v1 = rflookup rf r1
8                     v2 = rflookup rf r2
9                     action1 = mStore pc ((mFetch pc)+1)
10                    action2 = rfupdate rf rd (v1-v2)
11                 in
12                     ()

Figure 3: The \( \mathcal{P}_B \) Op-sub Rule in Bluespec\( _{pH} \) Syntax

As declared, RegFile\(^2\) is a fixed-size array\(^3\) of RegCell’s where each RegCell is a state element that holds an integer. Declaration of a dynamically sized structure is not allowed in Bluespec\( _{pH} \).

Within Bluespec\( _{pH} \)’s type system, hierarchical structures can be defined. For example:

data System = System Memory Processor InstMemory
data Processor = Proc Pc RegFile

In this example, a structure of type System is composed of Memory, Processor, and InstMemory. A substructure, Processor, is also defined. Analogous to TRS grammars, hierarchical type definitions describe the organization of state elements within a system.

4 Format of a Rule in Bluespec

Given that terms in TRS’s map to state elements in FSM’s, rewriting rules naturally correspond to state transition logic. Recall a rewriting rule is typically expressed as:

\[ S \text{ if } p(S) \rightarrow S' \]

In order for such a rule to be synthesizable as fixed combinational logic, the terms \( S \) and \( S' \) must be the same type. This ensures that no Bluespec\( _{pH} \) rule can represent an addition or removal of state elements. Bluespec\( _{pH} \) syntactically enforces this property by requiring a rule to be expressed as:

\[ S \text{ if } p(S) \rightarrow \delta(S) \]

where \( \delta(S) \) can only specify the changes in the values of existing state terms in \( S \). Bluespec\( _{pH} \) statements that correspond to \( p(S) \) can represent arbitrary combinational logics, but only statements corresponding to \( \delta(S) \) can cause updates to mutable MCell terms in \( S \). A Bluespec\( _{pH} \) rewrite rule can always be restated in the conventional syntax:

\[ S \text{ if } p(S) \rightarrow \delta(S)(S) \]

\( \mathcal{P}_B \)’s Op Rule for subtraction is stated in Bluespec\( _{pH} \) in Figure 3. The pattern term, \( S \), containing the entire system states is stated on line 1 following the declaration of the rule’s name, opSubRule. Lines 2 thru 13 contain a \( pH \) let block. The Proc structure is deconstructed in line 3 to expose its internal structures for use in the body of the let block (lines 5 thru 13). The let

\(^2\)In \( pH \) syntax, the first symbol, e.g. RegFile, in the right-hand side of a type definition is a user defined type constructor for that type.

\(^3\)The symbol Int in the RegFile type definition gives the type of the array index.
block body begins with a predicate at lines 5 and 6 to check whether if the instruction memory location selected by the processor's pc contains a “RR Sub” (i.e., a register-to-register subtraction instruction). The memload function, used in the predicate, cannot have any side effects. If the predicate is true, the state changes specified in the body of the if statement should be applied. In this example, the state changes include incrementing the the state element pc by 1 (line 10), and updating the destination register with the result of subtracting the contents of two operand registers (line 11). The temporary variables action1 and action2, and the empty parentheses () are place holders to allow Bluespec_pH to share Parallel Haskel's parser and type checker.

5 Identifying Mutually Exclusive Rules

The simple $\mathcal{P}_B$ TRS given in Section 2 contains seven rules that appear to be independent. However, the rules are in fact mutually exclusive - at any instance, only one rule's predicate can be true. Except for the two $Jz$ Rules, each rule's predicate tests for a different instruction type at the current program location. Clearly, the current pc can only point to one instruction, and in the case of $Jz$, it must either jump or not jump.

Mutual exclusion of this type can help identify combinational logic resource that can be shared. For example, an ALU capable of both addition and subtraction can be shared by the Add and Sub Rules since $\mathcal{P}_B$ will never add and subtract simultaneously. In a small TRS, this type of mutual exclusion can certainly be deduced statically. However, such analysis would be prohibitively expensive as TRS’s becomes large and convoluted. Thus, Bluespec_pH supports a case construct to enable the architect to group and identify mutually exclusive rules of this kind.

The seven individual $\mathcal{P}_B$ TRS rules from Section 2 are restated in Figure 4 as a single case statement. The current instruction word is tested by a case (Line 15) and switches to one of the six branches, each corresponding to an instruction type. For brevity, only the two branches for the Op Rule(Sub) (Lines 36 thru 43) and JzRules (Lines 44 thru 59) are shown here. The $Jz$ branch contains both $Jz$-Jump and $Jz$-NoJump Rules. This excerpt is taken directly from the Bluespec_pH $\mathcal{P}_B$ description (axpb.hs) in Appendix A.2.

6 Support for Modular Design

A module in Bluespec_pH is an abstract data type that can be imported into a top-level description. The description of a module is provided separately from the main description, but it is also stated in the form of a Bluespec_pH description. In fact, every Bluespec_pH description itself is a module that can be imported by other modules. Interaction with an imported child module is only allowed through a set of well-defined interfaces exported by the child module. The actual structure and operations inside the child module is not visible externally. Under our current synthesis strategy, each Bluespec_pH module would be synthesized as an independent FSM. From a hardware design perspective, importing a module into a Bluespec_pH description is analogous to incorporating an independently encapsulated slave FSM to run cooperatively with the primary FSM of the parent module. At the design entry stage, the use of modules allows different parts of a design to be developed independently. This modular organization of descriptions also encourages reuse of source codes. At the synthesis level, partitioning a large design into multiple modules has the effect of replacing a single large FSM by multiple smaller cooperating FSM’s. This can result in a significant reduction of the combinational state transition logic.

Figure 5 contains the module description of the register file used in $\mathcal{P}_B$ (regfile.hs in Appendix C.3). A module description starts with the declaration of the exported interfaces on line 1.
<< excerpt from axpb.hs >>

11 axpbRule (Sys memory proc prog) =
12    let
13    (Proc pc rf)=proc
14    in
15    case (imemload prog (mFetch pc)) of
16    R Sub rd r1 r2 ->
17        let
18            v1 = rflookup rf r1
19            v2 = rflookup rf r2
20            action1 = mStore pc ((mFetch pc)+1)
21            action2 = rfupdate rf rd (v1-v2)
22            in
23            ()
24    Jz rc rt ->
25        let
26            vc= rflookup rf rc
27            vt= rflookup rf rt
28            in
29            case (rflookup rf rc) of
30                0 ->
31                    let
32                    action1 = mStore pc vt
33                    in
34                    ()
35                _ ->
36                let
37                    action1 = mStore pc ((mFetch pc)+1)
38                    in
39                    ()
40    . . . . . . . .

Figure 4: An Excerpt from the BluespecpH PB Description
<< excerpt from regfile.hs >>

1 module RegFileModule (RegFile, regFile, rflookup, rfupdate) where

2 import Array
3 import "isa"
4 import "mcell"

5 type RegCell a = MCell a
6 data RegFile a = RegFile (Array RegName (RegCell a))

......

10 rflookup :: RegFile a -> RegName -> a
11 rflookup (RegFile rf) r = (mFetch (rf!r))

12 rfupdate :: RegFile a -> RegName -> a -> ()
13 rfupdate (RegFile rf) r v =
14     let dummy=(mStore (rf!r) v)
15     in
16 ()

Figure 5: Module Definition of P_B's Register File

Between the keywords, module and where, the module name is given, followed by a list of items
that the module exports. Lines 2 thru 4 are examples of importing child modules into a description.
Lines 5 and 6 declares the module's data type, RegFile, as an array indexed by RegName. Each
cell of the array is a MCell type and, thus, is a state element.

Each interface description is composed of its type signature and its operational definition. For
example, on line 10, the type signature of rflookup is given as receiving a RegFile and a RegName
and returning an Int. Line 11 states rflookup should index the register file rf by the register
name r, and use mFetch to lookup the value of the selected state element. It should be obvious that
invoking the rflookup interface does not imply any side effects on the internal states of RegFile. In
contrast, the definition of rfupdate does imply a state change as indicated by the mStore operator
on line 15. According to the constraints set in Section 4, this interface cannot be invoked from the
predicate portion of a Bluespec rule. The signature of rfupdate is given on line 12, followed by the
operational definition in lines 13 thru 17. Line 15 states that invoking rfupdate causes the
state element, selected by indexing rf with r, to take on the new value, v.

By restricting interactions through a well-defined module interface, one module can be easily
substituted by another, as long as the same interface is maintained. This clean abstraction provides
a means for instantiate circuit elements from a design library. Certain circuit structures, like mem-
ory and arithmetic units, have been investigated extensively and are available as highly-optimized
drop-in circuits. An exact description of the circuits' internals is irrelevant to the simulation and
verification of the main design. Instead, their functionality can be quickly captured in a functionally
equivalent module to serve as a place holder until synthesis. One only needs to show the behavioral
equivalence between the place holder module and the actual circuits at the interface boundary.
<< excerpt from tag.h >>

2 typedef struct struct_TagValue {
  3 int _flag:1;
  4 union {
  5    Tag _tag;
  6    Value _value;
  7  } _tag_or_value;
  8 } TagValue;

<< excerpt from tagregfile.h >>

2 typedef TagValue TaggedRegFile[NumRegName];

<< excerpt from axpr.h >>

8 TaggedRegFile regFile;

Figure 6: Declaration and Instantiation of a Tagged Register File in \( \mathcal{P}_R \)

7 **Bluespec\(_C\): A TRS with the C Language Syntax**

Until now, Bluespec’s syntax has borrowed from a functional language with TRS semantics itself. However, to support the subset of TRS’s that we are interested in synthesizing, it is feasible to implement Bluespec in the syntax of many other popular languages, provided some TRS-specific extensions are made to the languages’ original semantics. In this section, we will briefly describe our experience with Bluespec\(_C\), a TRS language with the C language syntax.

As an synthesizable architectural description language, Bluespec\(_C\) has the same first-order requirements as Bluespec\(_PH\). Bluespec\(_C\) allows explicit identification of state elements in TRS’s. In a Bluespec\(_C\) description, TRS terms intended to be state elements are declared as global variables under C’s syntax. Bluespec\(_C\) also makes use of C’s type definition facilities to provide the means for organizing and defining hierarchical state structures. To illustrate this, the Bluespec\(_C\) code, from Appendix E, for declaring and instantiating the tagged register file used by \( \mathcal{P}_R \) is given in Figure 6. A register file of TaggedRegFile type is instantiated on line 8 of axpr.h. Since regFile is declared as a global variable, it is explicitly identified as a state element in the system. The structure of TaggedRegFile is user defined to be an array of TagValue’s. TagValue itself is another user-defined structure that can contain a piece of datum of either Tag or Value type.

Like in Bluespec\(_PH\), the format of a Bluespec\(_C\) rule is also syntactically enforced to be:

\[ S \text{ if } p(S) \rightarrow \delta(S) \]

The Bluespec\(_PH\) restriction that limits updating of state element to within \( \delta(S) \) also applies to Bluespec\(_C\) descriptions. Figure 7 contains the equivalent Bluespec\(_C\) translation of the Op Rule. (For comparison, the same Op Rule was explained in Section 2 and then translated into Bluespec\(_PH\) in Section 4.) A rule in Bluespec\(_C\) takes on the syntax of a C function. A rule begins with a declaration of the rule’s name and its return value type. When a rule is typed to return a non-void
1 void opSubRule() {
2     Instruction inst = imemload (imemory, pc);
3     if ( (majorOp(inst)==RR) && (minorOp(inst)==Sub) ) {
4         RegName rd, rs, rt;
5         rd = RegD(inst);
6         rs = RegS(inst);
7         rt = RegT(inst);
8         rfupdate(regFile, rd,
9                     rflookup(regFile, rs) - rflookup(regFile, rt));
10         pc = pc+1;
11     }
12 }

Figure 7: The \( \mathcal{P}_B \) Op-Sub Rule in Bluespec\(_C\) Syntax

typedef Value RegFile[NumRegName];
1 extern Value rflookup(RegFile, RegName);
2 extern void rfupdate(RegFile, RegName, Value);

Figure 8: Bluespec\(_C\) Header File (regfile.h) for a Register File Module

value, it implies the rule represents combinational logic and should not modify any state elements. Rules representing a state transition must return a void. The pattern term, \( S \), of a rule is not directly represented in the syntax of the rule itself. Instead, Bluespec\(_C\) assumes that all rules have the same pattern, which corresponds to the list of all state terms in the system\(^4\).

Without pattern matching, the condition for applying a rule is purely decided by the rule’s predicate, \( p \). This does not reduce the power of TRS’s in Bluespec\(_C\) since the conditions described by a pattern match can always be restated as a predicate. The predicate for \texttt{opSubRule} tests whether the current instruction location contains a register-to-register subtraction instruction (lines 2 and 3). When this predicate is true, the state changes specified by \( \delta(S) \) (lines 8 and 9) are applied. The state changes of Op-Sub Rule include updating \texttt{regFile} by the result of the subtraction and incrementing \texttt{pc} by 1. The code segment shown in Figure 7 represents the Op Rule of \( \mathcal{P}_B \) as a single independent rule. As discussed in Section 5, to help identify the mutually exclusive relationships between the rules, the Bluespec\(_C\) description of \( \mathcal{P}_B \) in Appendix D.4 actually groups all seven \( \mathcal{P}_B \) rules into a single \texttt{case} statement.

A Bluespec\(_C\) module is composed of a pair of \texttt{.h} and \texttt{.c} files. Figures 8 and 9 contain \texttt{regfile.h} (Appendix D.5) and \texttt{regfile.c} (Appendix C.3), the header and the body of the register file module used by \( \mathcal{P}_B \). The \texttt{.h} header file declares interfaces exported by the register file module. The \texttt{.c} file defines the module’s content, which includes declaration of state elements, rewrite rules for a module’s internal operation , and the rules to support the operations of the interfaces. As discussed in the context of Bluespec\(_PH\) in Section 6, a Bluespec\(_C\) module also enables modular development of a description.

An interesting issue arises when one wishes to execute a Bluespec\(_C\) description as a C program

\(^{4}\)In the \( \mathcal{P}_B \) example, this pattern is made up of \texttt{imemory, memory, regFile} and \texttt{pc}, which are all declared as global variables elsewhere.
Figure 9: A BluespecC Description (regfile.c) of a Register File Module

to simulate the design. Whereas, when a Bluespec\textsubscript{pH} description is executed as a \textit{pH} program, the \textit{pH} run-time system automatically handles the selection and application of rules because a \textit{pH} program is inherently a TRS. In Bluespec\textsubscript{C}, however, although a rule fits the syntax of a C function, TRS’s execution semantics is not inherent to C’s execution model. In order to simulate a Bluespec\textsubscript{C} description, a top level execution loop needs to be inserted. This execution loop is responsible for repeatedly invoking each “rule” in the system to test its predicate. A rule is applied (i.e. state changes takes place) when its predicate is true at the time of invocation. To model the asynchronous nature of TRS’s, the execution loop can randomize the order in which rules are invoked.

8 Concluding Remarks

In this paper, we have presented our results from specifying two microprocessor architecture (\(P_B\) and \(P_R\)) in synthesizable TRS’s. Two versions of the Bluespec TRS architectural description language were developed and evaluated in this study. The Bluespec\textsubscript{pH} source files to describe \(P_B\) and \(P_R\) are in Appendices A and B. Appendix C contains the source files for the shared modules used by both \(P_B\) and \(P_R\). Appendices D, E, and F contain the corresponding source files in Bluespec\textsubscript{C}.

These examples have illustrated that architectural description using TRS’s can be very similar to programming in a standard high-level programming language, whether functional (e.g. \textit{pH}) or declarative (e.g. C or Java). The important differences between Bluespec and a standard programming language are the introduction of rule constructs and the identification of helpful programming styles that make it possible to efficiently extract a term rewriting system and the corresponding hardware. These language features enable program analysis and compilation algorithms to compile from a very high level abstract description down to hardware.

We have compiled and executed both the Bluespec\textsubscript{pH} and Bluespec\textsubscript{C} source codes as simulators in their respective language environments. We have also compiled (by hand, following a mechanical procedure) the corresponding TRS’s into Verilog\textsuperscript{[6]} descriptions at the register-transfer level. The resulting Verilog descriptions have also been compiled successfully into FPGA’s by Synopsys Design Compiler\textsuperscript{[5]}. These experiences further reinforces the validity of our approach to architectural description and synthesis.
References


A Bluespec\textsubscript{pH} Sources for a Simple $\mathcal{AX}$ Instruction Set Processor

A.1 $\mathcal{AX}$ ISA Definition Module: (isa.hs)

\begin{verbatim}
1 --
2 -- ax ISA data types
3 --
4 data Op = Add | Sub deriving (Eq)
5 execOp :: Op -> Value -> Value -> Value
6 execOp op v1 v2 = 0

7 -- data Value = ndu Int, Bool, Address, InstAddress
8 type Value = Int
9 type Address=Int
10 type InstAddress=Int

11 --
12 -- ax register names
13 --
14 data RegName = R0 | R1 | R2 | R3 deriving (Ix,Ord,Eq)

15 type RegD=RegName -- destination register name
16 type RegC=RegName -- branch condition register name
17 type RegT=RegName -- branch target register name
18 type RegA=RegName -- load and store address
19 type RegV=RegName -- store value register name
20 type Reg1=RegName -- 1st argument register name
21 type Reg2=RegName -- 2st argument register name

22 --
23 -- ax instruction set
24 --
25 data AXInstruction = Loadc RegD Value
26   | Loadpc RegD
27   | RR Op RegD Reg1 Reg2
28   | Jz RegC RegT
29   | Load RegA RegD
30   | Store RegA RegV
\end{verbatim}

A.2 $\mathcal{PB}$ Processor Module: (axpb.hs)

\begin{verbatim}
1 module AXPB() where

2 import "isa"
3 import "regfile"
4 import "memory"
5 import "imemory"
6 import "mcell"

7 type Pc = MCell InstAddress
8 data Processor = Proc Pc (RegFile Value)
9 type Prog=IMemory
10 data System = Sys Memory Processor Prog
\end{verbatim}
axpBRule (Sys memory proc prog) =
  let (Proc pc rf)=proc
  in
  case (imemload prog (mFetch pc)) of
    Loadc rd v ->
      let
        action1 = mStore pc ((mFetch pc)+1)
        action2 = rfupdate rf rd v
      in
        ()
    Loadpc rd ->
      let
        action1 = mStore pc ((mFetch pc)+1)
        action2 = rfupdate rf rd (mFetch pc)
      in
        ()
    RR Add rd r1 r2 ->
      let
        v1 = rflookup rf r1
        v2 = rflookup rf r2
        action1 = mStore pc ((mFetch pc)+1)
        action2 = rfupdate rf rd (v1+v2)
      in
        ()
    RR Sub rd r1 r2 ->
      let
        v1 = rflookup rf r1
        v2 = rflookup rf r2
        action1 = mStore pc ((mFetch pc)+1)
        action2 = rfupdate rf rd (v1-v2)
      in
        ()
    Jz rc rt ->
      let
        vc= rflookup rf rc
        vt= rflookup rf rt
        case (rflookup rf rc) of
          0 ->
            let
              action1 = mStore pc vt
              in
              ()
        _ ->
          let
            action1 = mStore pc ((mFetch pc)+1)
            in
            ()
      in
    Load ra rd ->
      let
        va = rflookup rf ra
        vd = memload memory va
      in
        ()
action1 = rfupdate rf rd vd

Store ra rv ->

let

va=rflookup rf ra
vv=rflookup rf rv

action1 = memstore memory va vv

in

()
B Bluespec$_{PH}$ Sources for a Renaming/Reordering AX Processor

B.1 $\mathcal{P}_R$ Processor Module: (axpr.hs)

```hs
1 module AXPR() where

2 import "isa"
3 import "tag"
4 import "tagregfile"
5 import "itbs"
6 import "memory"
7 import "imemory"
8 import "mcell"

9 data PCell = RealPc InstAddress | StallPc
10 type Pc = MCell PCell
11 data Processor = Proc Pc TaggedRegFile ITBs
12 type Prog=IMemory
13 data System = Sys Memory Processor Prog

14 -- Ax instruction issue rules

15 issueRule:: System -> ()
16 issueRule (Sys memory (Proc pc rf itbs) prog) =
17     if (not (itbsfull itbs)) then
18         case (mFetch pc) of
19             RealPc ia ->
20                 case (imemload prog ia) of
21                     Loadc rd v ->
22                     let
23                         t = enqueue itbs (Loadc' rd (Tvalue v)) ia
24                         in
25                         action1 = markDone itbs t v
26                         action2 = trfupdateT rf rd t
27                         action3 = mStore pc (RealPc (ia+1))
28                     in
29                     ()
30             Loadpc rd ->
31                     let
32                         t = enqueue itbs (Loadpc' rd (Tvalue ia)) ia
33                         in
34                         action1 = markDone itbs t ia
35                         action2 = trfupdateT rf rd t
36                         action3 = mStore pc (RealPc (ia+1))
37                     in
38                     ()
39             RR op rd r1 r2 ->
40                     let
41                         tv1 = trfllookup rf r1
42                         tv2 = trfllookup rf r2
43                         t = enqueue itbs (RR' op rd tv1 tv2) ia
44                         action1 = trfupdateT rf rd t
45                         action2 = mStore pc (RealPc (ia+1))
46                     in
```

17
```
47 | Jz rc rt -> ()
48 |     let
49 |     tvc = trflookup rf rc
50 |     tvt = trflookup rf rt
51 |     t = enqueue itbs (Jz' tvc tvt) ia
52 |     action = mStore pc StallPc
53 |     in
54 |()
55 | Load ra rd -> ()
56 |     let
57 |     tva = trflookup rf ra
58 |     t = enqueue itbs (Load' tva rd) ia
59 |     action1 = trfupdateT rf rd t
60 |     action2 = mStore pc (RealPc (ia+1))
61 |     in
62 |()
63 | Store ra rv -> ()
64 |     let
65 |     tva = trflookup rf ra
66 |     tvv = trflookup rf rv
67 |     t = enqueue itbs (Store' tva tvv) ia
68 |     action = mStore pc (RealPc (ia+1))
69 |     in
70 |()
71 | else ()
72 | axprExecuteRule (Sys memory (Proc pc rf itbs) prog) =
73 |     case (instReady itbs) of
74 |         (True, t, ia, (RR'' op v1 v2)) -> ()
75 |         let
76 |             action1 = markExec itbs t
77 |             v = execOp op v1 v2
78 |             ---
79 |             action2 = markDone itbs t v
80 |             in
81 |()
82 |         (True, t, ia, (Jz'' c tia)) | c==0 -> ()
83 |         let
84 |             action1 = markExec itbs t
85 |             action2 = mStore pc (RealPc tia)
86 |             ---
87 |             action3 = markDone itbs t 0
88 |             in
89 |()
90 |         (True, t, ia, (Jz'' c iat)) | not (c==0) -> ()
91 |         let
92 |             action1 = markExec itbs t
93 |             action2 = mStore pc (RealPc (ia+1))
94 |             ---
95 |             action3 = markDone itbs t 0
96 |             in
97 |()
98 |```

99  ()
100  (True, t, ia, (Load,’’ a)) ->
101    let
102      action1 = markExec itbs t
103      v = (memload memory a)
104      ---
105      action2 = markDone itbs t v
106    in
107    ()
108  (True, t, ia, (Store,’’ a v)) ->
109    let
110      action1 = markExec itbs t
111      ---
112      action2 = (memstore memory a v)
113      ---
114      action3 = markDone itbs t 0
115    in
116    ()

117  axprCommitRule ( Sys memory (Proc pc rf itbs) prog ) =
118  case (dequeue itbs) of
119    (True, (Done,’’ t r v)) ->
120      let
121        action = (trfupdateV rf r t v )
122      in
123      ()
124

B.2 Instruction Template Buffer Module: (itbs.hs)

1  module ITBsModule(ITBs, itbs, itbsfull, enqueue, instReady,  
2    markExec, markDone, dequeue,  
4    Inst’’, RR’’, Jz’’, Load’’, Store’’, Done’’,  
5    OpCode) where

6  import Array
7  import "isa"
8  import "tag"
9  import "mcell"
10  import "qmcell"

11  -- ITBs interface data types
12  data Inst’ = Loadc’ RegD TV  
13    | Loadpc’ RegD TV  
14    | RR’ Op RegD TV1 TV2  
15    | Jz’ TVC TVT  
16    | Load’ TVA RegD  
17    | Store’ TVA TVV
18  data Inst’’ = RR’’ Op Value Value  
19    | Jz’’ Value InstAddress
20 | Load'' Address
21 | Store'' Address Value
22 | Done'' Tag RegD Value
23 | Nop''

24 data OpCode =
25 | Loadc0p
26 | Loadpc0p
27 | RR0p 0p
28 | Jz0p
29 | Load0p
30 | Store0p
31 deriving (Eq)

32 data Status = Invalid | Valid | Executing | Done deriving (Eq)

33 data ITBs = ITBs (Array Tag IBuf) IdxY Idx0
34 -- - an array of entries, plus
35 -- - IdxY, Idx0 which index the youngest
36 -- and oldest entries

37 data IBuf = IBuf (MCell InstAddress) ICell (MCell Status)
38 data ICell = ICell (MCell OpCode) (MCell RegD) (MCell TV1) (MCell TV2)
39 type Idx = MCell Tag
40 type IdxY = Idx
41 type Idx0 = Idx

42 opDecode :: Inst' -> OpCode
43 opDecode inst =
44 case inst of
45 (Loadc' _ _) -> Loadc0p
46 (Loadpc' _ _) -> Loadpc0p
47 (RR' op _ _) -> RR0p op
48 (Jz' _ _)  -> Jz0p
49 (Load' _ _) -> Load0p
50 (Store' _ _) -> Store0p

51 rdDecode :: Inst' -> RegName
52 rdDecode inst =
53 case inst of
54 (Loadc' rd _)  -> rd
55 (Loadpc' rd _)  -> rd
56 (RR' _ rd _ _)  -> rd
57 (Load' _ rd )  -> rd

58 tv1Decode :: Inst' -> TV
59 tv1Decode inst =
60 case inst of
61 (Loadc' _ _)  -> (Tvalue 0)
62 (Loadpc' _ tv) -> tv
63 (RR' _ tv _ _)  -> tv
64 (Jz' tv _ _)  -> tv
65 (Load' tv _ )  -> tv
66 (Store' tv _ )  -> tv
tv2Decode :: Inst' -> TV
tv2Decode inst =
  case inst of
    (Loadc' _ _) -> (Tvalue 0)
    (Loadpc' _ _) -> (Tvalue 0)
    (RR' _ _ tv) -> tv
    (Jz' _ tv) -> tv
    (Load' _ _) -> (Tvalue 0)
    (Store' _ tv) -> tv

itbs :: ITBs
itbs = ITBs (listArray (T0,T3) (replicate 4
  (IBuf (mReplace (mCell()) 0)
   (ICell (mReplace (mCell()) DomeOp)
   (mReplace (mCell()) R0)
   (mReplace (mCell()) (Tvalue 0))
   (mReplace (mCell()) (Tvalue 0))
   (mReplace (mCell()) Invalid)))
(mReplace (mCell()) T0)
(mReplace (mCell()) T0)

itbsfull :: ITBs -> Bool
itbsfull itbs =
case itbs of
  (ITBs _ idxY idx0) | (nextTag (mFetch idxY)) == (mFetch idx0) -> False
_ -> True

enqueue :: ITBs -> Inst' -> InstAddress -> Tag
enqueue itbs inst cia =
case itbs of
  (ITBs ibuf idxY idx0) ->
  let
    idxTemp = mFetch idxY
    action1 = mStore idxY (nextTag idxTemp)
    (IBuf ia iCell status)=ibuf!idxTemp
    (ICell op rd tv1 tv2)=icell
    action2 = mStore ia cia
    action3 = mStore status Valid
    action4 = mStore op (opDecode inst)
    action5 = mStore rd (rdDecode inst)
    action6 = mStore tv1 (tv1Decode inst)
    action7 = mStore tv2 (tv2Decode inst)
in
  idxTemp

dequeue :: ITBs -> (Bool, Inst')
dequeue itbs =
case itbs of
  (ITBs ibuf idxY idx0) | (mFetch idxY) == (mFetch idx0) ->
    (False, Nop')
(ITBs ibuf idxY idx0) ->

let

(IBuf _ icell status)=ibuf!((mFetch idx0))

actionX = mStore status Invalid

(ICell _ rd tv _)=icell

in

if (((mFetch status))==Done) then

let

idxTemp=(mFetch idx0)

action=mStore idx0

(nextTag idxTemp)

(Tvalue v)=mFetch tv

in

(TRUE, (Done'' idxTemp (mFetch rd) v))

else

(False, Nop'')

enabled :: ICell -> Bool

enabled icell =

let

(ICell _ _ tv1 tv2)= icell

in

case (mFetch tv1, mFetch tv2) of

  ((Tvalue _), (Tvalue _)) -> True

  _ -> False

encode :: ICell -> Inst''

encode (ICell op rd tv1 tv2) =

let

(Tvalue v1)=mFetch tv1

(Tvalue v2)=mFetch tv2

in

case mFetch op of

  RROp op -> RR'' op v1 v2

  LoadOp -> Load'' v1

  JzOp -> Jz'' v1 v2

  StoreOp -> Store'' v1 v2

noYoungerMemOp :: ITBs -> Tag -> Bool

noYoungerMemOp itbs t =

let

(ITBs ibuf idxY idx0)=itbs

(IBuf _ (ICell op rd tv1 tv2) status)=ibuf!t

in

if (((mFetch status))==Invalid) then

 TRUE

else if ( ((mFetch op)==LoadOp) ||

  (mFetch op)==StoreOp ) ) \&\&

  not ( (mFetch status)==Done) ) then

  False

else

  noYoungerMemOp itbs (prevTag t)
instReady :: ITBs -> (Bool, Tag, InstAddress, Inst')

instReady itbs =
  let
    idxS = qMCell T0
    in
      case itbs of
        (ITBs ibuf idxY idx0) | (mFetch idxY) == (mFetch idx0) ->
          let
            action = qMStore idxS (mFetch idx0)
          in
            (False, T0, 0, Nop')
        (ITBs ibuf idxY idx0) ->
          let
            (IBuf is icell status) = ibuf!((qMFetch idxS))
            (ICell op rd tvi tv2)=icell
            in
              if ((mFetch status)==Invalid) then
                let
                  action = qMStore idxS (mFetch idx0)
                in
                  (False, T0, 0, Nop')
              else
                let idxTemp = ((qMFetch idxS))
                  action = qMStore idxS (nextTag idxTemp)
                in
                  if (((mFetch status)==Done) ||
                    ((mFetch status)==Executing) ||
                    not (enabled icell) ) then
                    (False, T0, 0, Nop')
                else
                  if ((mFetch op)==LoadOp) then
                    if (noYoungerMemOp itbs (prevTag idxTemp)) then
                      (True, idxTemp, (mFetch ia), (encode icell))
                    else
                      (False, T0, 0, Nop')
                  else if ((mFetch op)==StoreOp) then
                    if (noYoungerMemOp itbs (prevTag idxTemp)) then
                      (True, idxTemp, (mFetch ia), (encode icell))
                    else
                      (False, T0, 0, Nop')
                  else
                    (True, idxTemp, (mFetch ia), (encode icell))

forward :: ITBs -> Tag -> Value -> ()
forward itbs t v = forwardNext itbs t v T0

forwardNext :: ITBs -> Tag -> Value -> Tag -> ()
forwardNext itbs t v t' =
  let
    action = forwardEach itbs t v t'
  in
    if (moreTag t') then
      forwardNext itbs t v (nextTag t')
else ()

case icell of 
  (ICell _ _ tv0 tv1) | ((mFetch tv0)==(Ttag t)) && ((mFetch tv1)==(Ttag t)) -> 
    let 
      action1=(mStore tv0 (Tvalue v)) 
      action2=(mStore tv1 (Tvalue v)) 
    in 
    ()

  (ICell _ _ tv0 tv1) | ((mFetch tv0)==(Ttag t)) -> 
    let 
      action=(mStore tv0 (Tvalue v)) 
    in 
    ()

markExec :: ITBs -> Tag -> ()
markExec itbs t = 
  let 
    (ITBs ibuf _) = itbs 
    (IBuf _ _ status) = ibuf!t 
    actionX = mStore status Executing 
  in 
  ()

markDone :: ITBs -> Tag -> Value -> ()
markDone itbs t v = 
  let 
    (ITBs ibuf _) = itbs 
    (IBuf _ icell status) = ibuf!t 
    (ICell _ rd tv1 _) = icell 
    action1 = mStore tv1 (Tvalue v) 
    action2 = forward itbs t v 
    action3 = mStore status Done 
  in 
  ()

B.3 Tagged Data Type Definition Module: (tag.hs)

module TagModule(Tag, TV, Tvalue, Ttag, TVC, TVA, TVT, TVV, TV1, TV2, T0, T1, T2, T3, moreTag, nextTag, prevTag) where
3 import "isa"
4 --
5 -- AXPR-specific internal data types
6 --
7 data Tag= T0 | T1 | T2 | T3
8 deriving (Ix,Ord,Eq)
9 data TV = Tvalue Value | Ttag Tag deriving (Eq)
10 type TVC=TV     -- branch condition tag/value
11 type TVT=TV     -- branch target tag/value
12 type TVA=TV     -- branch condition tag/value
13 type TVV=TV     -- store value tag/value
14 type TV1=TV     -- 1st argument tag/value
15 type TV2=TV     -- 2st argument tag/value

16 moreTag :: Tag->Bool
17 moreTag t = not (t==T3)
18 nextTag :: Tag->Tag
19 nextTag t = t
20 prevTag :: Tag->Tag
21 prevTag t = t

B.4 Tagged Register File Module: (tagregfile.hs)

1 module TaggedRegFileModule (TaggedRegFile, taggedRegFile, trflookup,
2 trfupdateT,trfupdateV) where

3 import Array
4 import "isa"
5 import "mcell"
6 import "tag"
7 import "regfile"

8 type TaggedRegFile = RegFile TV

9 taggedRegFile :: TaggedRegFile
10 taggedRegFile = regFile (Tvalue 0)
11 trflookup :: TaggedRegFile -> RegName -> TV
12 trflookup rf r = (rflookup rf r)
13 trfupdateT :: TaggedRegFile -> RegName -> Tag -> ()
14 trfupdateT rf r t =
15       let
16           action=rfupdate rf r (Ttag t)
17       in
18           ()
19
20 trfupdateV :: TaggedRegFile -> RegName -> Tag -> Value -> ()
21 trfupdateV rf r t v =
22       case (rflookup rf r) of
23           (Ttag tlast) | (tlast == t) ->
rfupdate rf r (Tvalue v)
_ ->
()
C  Bluespec\textsubscript{PH} Sources for Library Modules shared by \( \mathcal{P}_B \) and \( \mathcal{P}_R \)

C.1  Read-only Instruction Memory Module: (imemory.hs)

1 module IMemory(IMemory,imemory,imemload) where

2 import Array
3 import "isa"
4 import "mcell"

5 data IMemory = IMemory (Array InstAddress (MCell AXInstruction))

6 imemory :: IMemory
7 imemory = IMemory (listArray (1,3) [(mReplace (mCell())) (Loadc RO 0)),
8 (mReplace (mCell())) (Loadc RO 0)),
9 (mReplace (mCell())) (Loadc RO 0)),
10 (mReplace (mCell())) (Loadc RO 0))]

11 imemload :: IMemory -> InstAddress -> AXInstruction
12 imemload (IMemory imem) a = (mFetch (imem!a))

C.2  Read-Write Memory Module: (memory.hs)

1 module Memory(Memory,memory,memload,memstore) where

2 import Array
3 import "isa"
4 import "mcell"

5 data Memory = Memory (Array Address (MCell Value))

6 memory :: Memory
7 memory = Memory (listArray (1,3) [mCell(),mCell(),mCell(),mCell()])

8 memload :: Memory -> Address -> Value
9 memload (Memory mem) a = (mFetch (mem!a))

10 memstore :: Memory -> Address -> Value -> ()
11 memstore (Memory mem) a v =
12 let
13     dummy = (mStore (mem!a) v)
14 in
15 ()

C.3  Generic Register File Module: (regfile.hs)

1 module RegFileModule (RegFile, regFile, rflookup, rfupdate) where

2 import Array
3 import "isa"
4 import "mcell"
5 type RegCell a = MCell a
6 data RegFile a = RegFile (Array RegName (RegCell a))

7 regFile :: a -> RegFile a
8 regFile init = RegFile (listArray (R0,R3)
9     (replicate 4 (mReplace (mCell()) init)))

10 rflookup :: RegFile a -> RegName -> a
11 rflookup (RegFile rf) r = (mFetch (rf!r))
12 rfupdate :: RegFile a -> RegName -> a -> ()
13 rfupdate (RegFile rf) r v =
14     let dummy=(mStore (rf!r) v)
15     in ()
D Bluespec₃ Sources for a Simple ₃₄ Processor

D.1 ₃₄ ISA Definition Header File: (isa.h)

1 typedef enum enum_MajorOp {
2 Loadc, Loadpc, RR, Jz, Load, Store, NumMajorOp
3 } MajorOp;
4
typedef enum enum_MinorOp {
5 Add, Sub, NumMinorOp
6 } MinorOp;
7
typedef int Value;
8
typedef int Address;
9
typedef int InstAddress;
10
typedef enum enum_RegName { R0, R1, R2, R3, NumRegName} RegName;
11
typedef struct struct_Instruction {
12 MajorOp _majorOp : 4;
13 RegName _RegD : 2;
14 union {
15 struct { int _immediate : 16; } _immediate_instruction;
16 struct { RegName _RegS : 2; RegName _RegT : 2; MinorOp _minorOp : 4; } _rr_instruction;
17 } _instruction_format;
18 } Instruction;
19
#define majorOp(a) ((a)._majorOp)
20#define RegD(a) ((a)._RegD)
21
#define immediate(a) ((a)._instruction_format._immediate_instruction._immediate)
22#define RegS(a) ((a)._instruction_format._rr_instruction._RegS)
23#define RegT(a) ((a)._instruction_format._rr_instruction._RegT)
24#define minorOp(a) ((a)._instruction_format._rr_instruction._minorOp)
25
#define Reg1 RegS
26#define Reg2 RegT
27#define RegV RegT
28#define RegA RegS
29#define RegC RegS
30 extern Instruction makeInstruction(MajorOp, RegName, RegName, RegName, int, MinorOp);

D.2 ₃₄ ISA Definition Module: (isa.c)

1 #include <isa.h>
2
2 Instruction makeInstruction(MajorOp majorop,
3 RegName rd, RegName rs, RegName rt, int immed, MinorOp minorop) {
4 Instruction i;
5 majorOp(i) = majorop;
6 minorOp(i) = minorop;
7 RegD(i) = rd;
8 RegS(i) = rs;
9  RegT(i) = rt;
10  if (majorop == Loadc) {
11     immediate(i) = immed;
12  }
13  return(i);
14 }

D.3 \(P_B\) Processor Header File: (axpb.h)

1  extern RegFile regFile;
2  extern Memory memory;
3  extern Imemory imemory;

D.4 \(P_B\) Processor Module: (axpb.c)

1 #include <isa.h>
2 #include <regfile.h>
3 #include <memory.h>
4 #include <imemory.h>
5 #include <main.h>
6 RegFile regFile;
7 Memory memory;
8 Imemory imemory;
9 Address pc;
10 void
11 rules() {
12   Instruction i;
13   i = imemload(imemory, pc);
14   switch (majorOp(i)) {
15      case Loadc : {
16         RegName rd;
17         Value v;
18         rd = RegD(i);
19         v = immediate(i);
20         rfupdate(regFile, rd, v);
21         pc = pc+1;
22         break;
23      }
24      case Loadpc : {
25         RegName rd;
26         rd = RegD(i);
27         rfupdate(regFile, rd, pc);
28         pc = pc+1;
29         break;
30      }
31      case RR : {
32         switch (minorOp(i)) {
33            case Add : {
34               RegName rd, rs, rt;
35               rd = RegD(i);
rs = RegS(i);
rt = RegT(i);
rfupdate(regFile, rd, rflookup(regFile, rs) + rflookup(regFile, rt));
pc = pc+1;
break;
}
case Sub: {
    RegName rd, rs, rt;
    rd = RegD(i);
    rs = RegS(i);
    rt = RegT(i);
    rfupdate(regFile, rd, rflookup(regFile, rs) - rflookup(regFile, rt));
    pc = pc+1;
    break;
}
case Jz: {
    RegName rc, rt;
    Value vc, vt;
    rc = RegC(i);
    rt = RegT(i);
    vc = rflookup(regFile, rc);
    vt = rflookup(regFile, rt);
    if (vc == 0) {
        pc = vt;
    } else {
        pc = pc+1;
    }
    break;
}
case Load : {
    RegName rd, ra;
    Value v;
    rd = RegD(i);
    ra = RegA(i);
    v = memload(memory, rflookup(regFile, ra));
    rfupdate(regFile, rd, v);
    pc = pc+1;
    break;
}
case Store : {
    RegName ra, rv;
    ra = RegA(i);
    rv = RegV(i);
    memstore(memory, rflookup(regFile, ra), rflookup(regFile, rv));
    pc = pc+1;
    break;
}
}
88 void
89 loadMemory() {
90    // dead cycles
91    imemory[0]=makeInstruction(Loadc, R0, R0, R0, 0, 0);
92    imemory[1]=makeInstruction(Loadc, R0, R0, R0, 0, 0);
93
94    // clear registers
95    imemory[2]=makeInstruction(Loadc, R1, R0, R0, 0, 0);
96    imemory[3]=makeInstruction(Loadc, R2, R0, R0, 0, 0);
97    imemory[4]=makeInstruction(Loadc, R3, R0, R0, 0, 0);
98
99    // find branch target
100   imemory[5]=makeInstruction(Loadpc, R1, R0, R0, 0, 0);
101   imemory[6]=makeInstruction(Loadc, R3, R0, R0, 7, 0);
102   imemory[7]=makeInstruction(RR, R1, R3, R1, 0, Add);
103
104   // store branch target to imemory[10]
105   imemory[8]=makeInstruction(Loadc, R3, R0, R0, 10, 0);
106   imemory[9]=makeInstruction(Store, R0, R3, R1, 0, 0);
107
108   // jump to count to 10 function
109   imemory[10]=makeInstruction(Loadc, R3, R0, R0, 114, 0);
110   imemory[11]=makeInstruction(Jz, R0, R0, R3, 0, 0);
111
112   // branch back to function call
113   imemory[12]=makeInstruction(Loadc, R3, R0, R0, 10, 0);
114   imemory[13]=makeInstruction(Jz, R0, R0, R3, 0, 0);
115
116   // clear registers
117   imemory[14]=makeInstruction(RR, R1, R0, R0, 0, Add);
118   imemory[15]=makeInstruction(RR, R2, R0, R0, 0, Add);
119   imemory[16]=makeInstruction(RR, R3, R0, R0, 0, Add);
120
121   // increment by 1
122   imemory[17]=makeInstruction(Loadc, R3, R0, R0, 1, 0);
123   imemory[18]=makeInstruction(RR, R1, R1, R3, 0, Add);
124
125   // compare to 10
126   imemory[19]=makeInstruction(Loadc, R3, R0, R0, 10, 0);
127   imemory[20]=makeInstruction(RR, R2, R3, R1, 0, Sub);
128
129   // if r1=10 then return
130   imemory[21]=makeInstruction(Loadc, R3, R0, R0, 125, 0);
131   imemory[22]=makeInstruction(Jz, R0, R2, R3, 0, 0);
132
133   // else jump back
134   imemory[23]=makeInstruction(Loadc, R3, R0, R0, 117, 0);
135   imemory[24]=makeInstruction(Jz, R0, R0, R3, 0, 0);
136
137   // function return
138   imemory[25]=makeInstruction(Loadc, R3, R0, R0, 10, 0);
139   imemory[26]=makeInstruction(Loadc, R3, R0, R0, 0, 0);
140   imemory[27]=makeInstruction(Jz, R0, R0, R3, 0, 0);
```c
130  imemory[128]=makeInstruction(RR, RO, RO, RO, 0, Add);
131  imemory[129]=makeInstruction(RR, RO, RO, RO, 0, Add);
132  }

133  main() {
134    loadImemory();
135    while(1) {
136      rules();
137    }
138  }

D.5  Generic Register File Header File: (regfile.h)
1  typedef Value RegFile[NumRegName];
2  extern Value rflookup(RegFile, RegName);
3  extern void rfupdate(RegFile, RegName, Value);

D.6  Generic Register File Module: (regfile.c)
1  #include <isa.h>
2  #include <regfile.h>

3  Value
4  rflookup(RegFile f, RegName r) {
5    return f[r];
6  }

7  void
8  rfupdate(RegFile f, RegName r, Value v) {
9    f[r] = v;
10  }
```
E  Bluespec$_C$ Sources for a Renaming/Reordering $\mathcal{AX}$ Processor

E.1  $\mathcal{P}_R$ Processor Header File: (axpr.h)

1 extern TaggedRegFile regFile;
2 extern Memory memory;
3 extern Imemory imemory;

E.2  $\mathcal{P}_R$ Processor Module: (axpr.c)

1 #include <isa.h>
2 #include <tag.h>
3 #include <regfile.h>
4 #include <memory.h>
5 #include <imemory.h>
6 #include <itbs.h>
7 #include <main.h>

8 TaggedRegFile regFile;
9 Memory memory;
10 Imemory imemory;
11 Address pc;
12 int stalled;
13 Itbs itbs;

14 void
15 issueRules() {
16   Tag t;
17   Instruction i;
18   Template tmp;

19   t = itbsCurrent(itbs);
20   if (t != NoTag) {
21     if (!stalled) {
22       i = imemload(imemory, pc);
23       switch (majorOp(i)) {
24         case Loadc : {
25           RegName rd;
26           rd = RegD(i);
27           setInstAddress(tmp, pc);
28           templateSetMajorOp(tmp, Loadc);
29           setValueS(tmp, immediate(i));
30           templateSetRegD(tmp, rd);
31         }
32         case Loadpc : {
33           RegName rd;
34           rd = RegD(i);
35         }
36         break;
37       }
38     }
39   }
40   pc=pc+1;
41   break;
42      }
39    setInstAddress(tmp, pc);
40        templateSetMajorOp(tmp, Loadpc);
41    setValueS(tmp, pc);
42    templateSetRegD(tmp, rd);
43
44    enqueue(itbs, tmp);
45    trfupdateT(regFile, rd, t);
46    pc=pc+1;
47    break;
48
case RR : {
49    switch (minorOp(i)) {
50        case Add: {
51            RegName rd, rs, rt;
52            rd = RegD(i);
53            rs = RegS(i);
54            rt = RegT(i);
55
56            setInstAddress(tmp, pc);
57            templateSetMajorOp(tmp, RR);
58            templateSetMinorOp(tmp, Add);
59            templateSetRegD(tmp, rd);
60            setTVS(tmp, trflookup(regFile, rs));
61            setTVT(tmp, trflookup(regFile, rt));
62
63            enqueue(itbs, tmp);
64            trfupdateT(regFile, rd, t);
65            pc=pc+1;
66            break;
67        }
68        case Sub: {
69            RegName rd, rs, rt;
70            rd = RegD(i);
71            rs = RegS(i);
72            rt = RegT(i);
73
74            setInstAddress(tmp, pc);
75            templateSetMajorOp(tmp, RR);
76            templateSetMinorOp(tmp, Sub);
77            templateSetRegD(tmp, rd);
78            setTVS(tmp, trflookup(regFile, rs));
79            setTVT(tmp, trflookup(regFile, rt));
80
81        }
82    }
83    break;
84    }
85    case Jz : {
86        RegName rc, rt;
TagValue tvc, tvt;
rc = RegC(i);
rt = RegT(i);
tvc = trflookup(regFile, rc);
tvt = trflookup(regFile, rt);
setInstAddress(tmp, pc);
templateSetMajorOp(tmp, Jz);
setTVS(tmp, tvc);
setTVT(tmp, tvt);
enqueue(itbs, tmp);
pc=pc+1;
stalled=1;
break;
case Load : {
    RegName rd, ra;
    Value v;
    rd = RegD(i);
    setInstAddress(tmp, pc);
templateSetMajorOp(tmp, Load);
templateSetRegD(tmp, rd);
setTVS(tmp, trflookup(regFile, ra));
enqueue(itbs, tmp);
trfupdateT(regFile, rd, t);
pc=pc+1;
break;
}
case Store : {
    RegName ra, rv;
    ra = RegA(i);
    rv = RegV(i);
s
    setInstAddress(tmp, pc);
templateSetMajorOp(tmp, Store);
setTVS(tmp, trflookup(regFile, ra));
setTVT(tmp, trflookup(regFile, rv));
enqueue(itbs, tmp);
pc=pc+1;
break;
}
134 Instruction i;
135 DispatchTemplate tmp;
136
137 tmp=instReady(itbs);
138
139 switch (dispatchTemplateMajorOp(tmp)) {
140    case RR: {
141        switch (dispatchTemplateMinorOp(tmp)) {
142            case Add: {
143                Value v = dispatchTemplateValueS(tmp)+dispatchTemplateValueT(tmp);
144                markDone(itbs,dispatchTemplateTag(tmp),v);
145                break;
146            }
147            case Sub: {
148                Value v = dispatchTemplateValueS(tmp)-dispatchTemplateValueT(tmp);
149                markDone(itbs,dispatchTemplateTag(tmp),v);
150                break;
151            }
152            break;
153        }
154    case Jz: {
155        if (dispatchTemplateValueS(tmp)) {
156            pc=dispatchTemplateInstAddress(tmp)+1;
157            stalled=0;
158            markDone(itbs,dispatchTemplateTag(tmp),0);
159        } else {
160            pc=dispatchTemplateValueT(tmp);
161            stalled=0;
162            markDone(itbs,dispatchTemplateTag(tmp),0);
163        }
164        break;
165    }
166    case Load: {
167        Value v=memload(memory,dispatchTemplateValueS(tmp));
168        markDone(itbs,dispatchTemplateTag(tmp),v);
169        break;
170    }
171    case Store: {
172        Address a=dispatchTemplateValueS(tmp);
173        Value v=dispatchTemplateValueT(tmp);
174        memstore(memory,a,v);
175        markDone(itbs,dispatchTemplateTag(tmp),0);
176        break;
177    }
178    }
179    break;
180
181    void
182    commitRule() {
CommitTemplate tmp;

tmp=dequeue(itbs);

if (commitTemplateTag(tmp)!="NoTag") {
  RegName r;
  Tag t;
  Value v;
  r=commitTemplateReg(tmp);
  t=commitTemplateTag(tmp);
  v=commitTemplateValue(tmp);
  trfupdateV(regFile, r, t, v);
  remove(itbs, t);
}

void loadImemory() {
  // dead cycles
  imemory[0]=makeInstruction(Loadc,  R0, R0,  R0,  0, 0);
  imemory[1]=makeInstruction(Loadc,  R0, R0,  R0,  0, 0);

  // clear registers
  imemory[2]=makeInstruction(Loadc,  R1, R0,  R0,  0, 0);
  imemory[3]=makeInstruction(Loadc,  R2, R0,  R0,  0, 0);
  imemory[4]=makeInstruction(Loadc,  R3, R0,  R0,  0, 0);

  // find branch target
  imemory[5]=makeInstruction(Loadpc, R1, R0,  R0,  0, 0);
  imemory[6]=makeInstruction(Loadc,  R3, R0,  R0,  7, 0);
  imemory[7]=makeInstruction(RR,  R1, R3,  R1, 0,  Add);

  // store branch target to imemory[10]
  imemory[8]=makeInstruction(Loadc,  R3, R0,  R0, 10, 0);
  imemory[9]=makeInstruction(Store, R0,  R3,  R1, 0, 0);

  // jump to count to 10 function
  imemory[10]=makeInstruction(Loadc,  R3, R0,  R0, 114, 0);
  imemory[11]=makeInstruction(Jz, R0,  R0,  R3, 0, 0);

  // branch back to function call
  imemory[12]=makeInstruction(Loadc,  R3, R0,  R0, 10, 0);
  imemory[13]=makeInstruction(Jz, R0,  R0,  R3, 0, 0);

  // clear registers
  imemory[14]=makeInstruction(RR,  R1, R0,  R0, 0,  Add);
  imemory[15]=makeInstruction(RR,  R2, R0,  R0, 0,  Add);
  imemory[16]=makeInstruction(RR,  R3, R0,  R0, 0,  Add);

  // increment by 1
  imemory[17]=makeInstruction(Loadc,  R3, R0,  R0, 1, 0);
  imemory[18]=makeInstruction(RR,  R1, R1,  R3, 0,  Add);
229 // compare to 10
230  imemory[119]=makeInstruction(Loadc, R3, R0, R0, 10, 0);
231  imemory[120]=makeInstruction(RR, R2, R3, R1, 0, Sub);

232 // if r1==10 then return
233  imemory[121]=makeInstruction(Loadc, R3, R0, R0, 125, 0);
234  imemory[122]=makeInstruction(Jz, R0, R2, R3, 0, 0);

235 // else jump back
236  imemory[123]=makeInstruction(Loadc, R3, R0, R0, 117, 0);
237  imemory[124]=makeInstruction(Jz, R0, R0, R3, 0, 0);

238 // function return
239  imemory[125]=makeInstruction(Loadc, R3, R0, R0, 10, 0);
240  imemory[126]=makeInstruction(Load, R3, R3, R0, 0, 0);
241  imemory[127]=makeInstruction(Jz, R0, R0, R3, 0, 0);
242  imemory[128]=makeInstruction(RR, R0, R0, R0, 0, Add);
243  imemory[129]=makeInstruction(RR, R0, R0, R0, 0, Add);
244 }

245 main() {
246  int round=0;
247  loadMemory();
248  resetRegisters(regFile);

249 while(1) {
250   round++;
251   issueRules();
252   executeRules();
253   commitRule();
254   
255   ruleRetire(itbs);
256 }
257 }

E.3 Instruction Template Buffer Header File: (itbs.h)

1 typedef enum enum_Status { Invalid, Valid, Executing, Done, NumStatus } Status;

2 typedef struct struct_Chiptemplate {
3   Tag _tag;
4   RegName _reg;
5   Value _value;
6 } CommitTemplate;

7 #define commitTemplateSetTag(d, v) ((d)._tag = (v))
8 #define commitTemplateSetReg(d, v) ((d)._reg = (v))
9 #define commitTemplateSetValue(d, v) ((d)._value = (v))

10 #define commitTemplateTag(d) ((d)._tag)
11 #define commitTemplateReg(d) ((d)._reg)
12 #define commitTemplateValue(d) ((d)._value)
typedef struct struct_DispatchTemplate {
    Tag _tag;
    InstAddress _instAddress;
    MajorOp _majorOp;
    MinorOp _minorOp;
    Value _valueS;
    Value _valueT;
} DispatchTemplate;

#define dispatchTemplateTag(d) ((d)._tag)
#define dispatchTemplateMajor0p(d) ((d)._major0p)
#define dispatchTemplateMinor0p(d) ((d)._minor0p)
#define dispatchTemplateInstAddress(d) ((d)._instAddress)

#define dispatchTemplateSetTag(d, v) ((d)._tag = (v))
#define dispatchTemplateSetMajor0p(d, v) ((d)._major0p = (v))
#define dispatchTemplateSetMinor0p(d, v) ((d)._minor0p = (v))
#define dispatchTemplateSetInstAddress(d, v) ((d)._instAddress = (v))

#define dispatchTemplateSetValueS(d, v) ((d)._valueS = (v))
#define dispatchTemplateSetValueT(d, v) ((d)._valueT = (v))

typedef struct struct_Template {
    InstAddress _instAddress;
    Status _status;
    MajorOp _majorOp : 4;
    RegName _RegD : 2;
    TagValue _tagValueS;
    TagValue _tagValueT;
    MinorOp _minorOp : 4;
} Template;

#define instAddress(t) ((t)._instAddress)
#define status(t) ((t)._status)

#define setInstAddress(t,v) ((t)._instAddress = (v))
#define setStatus(t,v) ((t)._status = (v))

#define templateMajor0p(t) ((t)._major0p)
#define templateMinor0p(t) ((t)._minor0p)
#define templateRegD(t) ((t)._RegD)

#define templateSetMajor0p(t,v) ((t)._major0p = (v))
#define templateSetMinor0p(t,v) ((t)._minor0p = (v))
#define templateSetRegD(t,v) ((t)._RegD = (v))

#define isTagS(tv) isTag(tv._tagValueS)
#define isValueS(tv) isValue(tv._tagValueS)

#define setTVS(t, tv) ((t)._tagValueS=tv)
```c
#define setTVT(t, tv) ((t)._tagValueT=tv)
#define setTagS(tv, t) setTag(tv._tagValueS, (t))
#define setValueS(tv, t) setValue(tv._tagValueS, (t))

#define getTagS(tv) getTag(tv._tagValueS)
#define getValueS(tv) getValue(tv._tagValueS)

#define isTagT(tv) isTag(tv._tagValueT)
#define isValueT(tv) isValue(tv._tagValueT)

#define setTagT(tv, t) setTag(tv._tagValueT, (t))
#define setValueT(tv, t) setValue(tv._tagValueT, (t))

#define getTagT(tv) getTag(tv._tagValueT)
#define getValueT(tv) getValue(tv._tagValueT)

typedef struct s_Itbs {
    Template _templates[NumTag];
    Tag _young;
    Tag _old;
} structItbs, Itbs[1];

define template(ib, index) (((ib)->_templates)[(index)])
define young(i) (((i)->_young)
define old(i) (((i)->_old)

define setYoung(i,v) (((i)->_young = (v))
define setOld(i,v) (((i)->_old = (v))

define itbsFull(i) (nextTag(young(i)) == old(i))
define itbsEmpty(i) (young(i) == old(i))
define itbsCurrent(i) (itbsFull(i) ? NoTag : young(i))

void forward(Itbs i, Tag t, Value v);
DispatchTemplate instReady(Itbs i);
CommitTemplate dequeue(Itbs i);

E.4 Instruction Template Buffer Module: (itbs.c)

#include <isa.h>
#include <tag.h>
#include <itbs.h>

void
enqueue(Itbs i, Template t) {
    Tag y;

    y = young(i);
    switch (templateMajorUp(t)) {
    case Loadc:
    case Loadp:
        status(t) = Done;
    ```
12 setValuenT(t, 0);
13 break;
14 }
15 case Load: {
16 setValuenT(t, 0);
17 }
18 default: {
19 status(t) = Valid;
20 break;
21 }
22 }
23 young(i) = nextTag(y);
24 template(i, y) = t;
25 }
26 CommitTemplate
27 dequeue(Tbs i) {
28 CommitTemplate c;
29 Template t;
30 Tag o;
31 o = old(i);
32 t = template(i, o);
33 if (status(t) == Done) {
34 commitTemplateSetTag(c, o);
35 commitTemplateSetValue(c, getValueS(t));
36 commitTemplateSetReg(c, templateRegD(t));
37 return c;
38 } else {
39 commitTemplateSetTag(c, NoTag);
40 return c;
41 }
42 }
43 void
44 remove(Tbs i) {
45 Tag o;
46 o = old(i);
47 setStatus(template(i, o), Invalid);
48 setOld(i, nextTag(o));
49 }
50 void
51 ruleRetire(Tbs i) {
52 CommitTemplate c;
53 Template t;
54 Tag o;
55 o = old(i);
56 t = template(i, o);
57 if (!itbsEmpty(i) && (status(t) == Invalid)) {
58 remove(i);
59 }
60 }
61 DispatchTemplate
62 instReady(Itbs i) {
63    Tag t;
64    DispatchTemplate d;
65    int found = 0;
66
67    for (t = 0; t < NumTag; t++) {
68        Template tmp;
69        tmp = template(i, t);
70        if (!found && (status(tmp) == Valid) && isValueT(tmp) && isValueS(tmp)) {
71            if (((templateMajor0p(tmp) != Load) && (templateMajor0p(tmp) != Store)) ||
72                (t == old(i))) {
73                found = 1;
74                dispatchTemplateSetTag(d, t);
75                dispatchTemplateSetInstAddress(d, instAddress(tmp));
76                dispatchTemplateSetMajorOp(d, templateMajorOp(tmp));
77                dispatchTemplateSetMinorOp(d, templateMinorOp(tmp));
78                dispatchTemplateSetValueS(d, getValueS(tmp));
79                dispatchTemplateSetValueT(d, getValueT(tmp));
80            }
81        }
82    } else {
83        dispatchTemplateSetTag(d, NoTag);
84        return d;
85    }
86    } return d;
87 }
88
90 void
91 markExec(Itbs i, Tag t) {
92    setStatus(template(i, t), Executing);
93 }
94
95 void
96 markDone(Itbs i, Tag t, Value v) {
97    Template tmp;
98    tmp = template(i, t);
99    if (((templateMajorOp(tmp) == Store) ||
100         (templateMajorOp(tmp) == Jz)) {
101        setStatus(template(i, t), Invalid);
102    } else {
103        setStatus(template(i, t), Done);
104        setValueS(template(i, t), v);
105        forward(i, t, v);
106    }
107 }
108
109 forward(Itbs i, Tag t, Value v) {
110    Tag c;
for (c = 0; c < NumTag; c++) {
    Template tmp;
    tmp = template(i, c);
    if (status(tmp) == Valid) {
        if (isTagS(tmp) && (getTagS(tmp) == t)) {
            setValueS(template(i, c), v);
        }
        if (isTagT(tmp) && (getTagT(tmp) == t)) {
            setValueT(template(i, c), v);
        }
    }
}

E.5 Tagged Data Type Definition Header File: (tag.h)

1 typedef enum enum_Tag { T0, T1, T2, T3, NumTag, NoTag } Tag;
2
typedef struct struct_TagValue {
3    int .flag:1;
4    union {
5        Tag .tag;
6        Value .value;
7    } .tag_or_value;
8 } TagValue;

9 #define isTag(tv) ((tv)->.flag)
10 #define isValue(tv) (!((tv)->.flag))

11 #define setTag(tv, t) { \ 
12     ((tv)->.flag = 1); ((tv)->tag_or_value.tag)=(t); \ 
13 }
14 #define setValue(tv, v) { \ 
15     ((tv)->.flag = 0); ((tv)->tag_or_value.value=(v)); \ 
16 }

17 #define getTag(tv) ((tv)->tag_or_value.tag)
18 #define getValue(tv) ((tv)->tag_or_value.value)

19 #define moreTag(t) ((t) !≡ (NumTag-1))
20 #define nextTag(t) (((t)+1)%NumTag)
21 #define prevTag(t) (((t)+NumTag-1)%NumTag)

E.6 Tagged Register File Header File: (tagregfile.h)

1 typedef Value RegFile[NumRegName];
2 typedef TagValue TaggedRegFile[NumRegName];

3 extern Value rflookup(RegFile, RegName);
4 extern void rupdate(RegFile, RegName, Value);
5 extern void trfupdateV(TaggedRegFile f, RegName r, Tag t, Value v);
6 extern void trfupdateT(TaggedRegFile, RegName, Tag);
extern TagValue trflookup(TaggedRegFile, RegName);

E.7 Tagged Register File Module for $P_R$: (tagregfile.c)

```
#include <isa.h>
#include <tag.h>
#include <regfile.h>

Value
rflookup(RegFile f, RegName r) {
    return f[r];
}

void
rfupdate(RegFile f, RegName r, Value v) {
    f[r] = v;
}

TagValue
TagValue Trflookup(TaggedRegFile f, RegName r) {
    return f[r];
}

void
TrfupdateV(TaggedRegFile f, RegName r, Tag t, Value v) {
    if (getTag(f[r]) == t) {
        setValue(f[r], v);
    }
}

void
TrfupdateT(TaggedRegFile f, RegName r, Tag t) {
    setTag(f[r], t);
}

void resetRegisters(TaggedRegFile f) {
    RegName i;
    for (i = 0; i < NumRegName; i++) {
        setValue(f[i], 0);
    }
}
```
F  Bluespec\textsubscript{C} Sources for Library Modules shared by $\mathcal{P}_B$ and $\mathcal{P}_R$

F.1  Read-only Instruction Memory Header File: (imemory.h)
1 #define SizeImemory 0x10000
2 typedef Instruction Imemory[SizeImemory];
3 extern Instruction imemload(Imemory, InstAddress);

F.2  Read-only Instruction Memory Module: (imemory.c)
1 #include <isa.h>
2 #include <imemory.h>
3 Instruction
4 imemload(Imemory m, InstAddress a) {
5    return m[a];
6 }

F.3  Read-Write Memory Header File: (memory.h)

1 #define SizeMemory 0x10000
2 typedef Value Memory[SizeMemory];
3 extern Value memload(Memory, Address);
4 extern void memstore(Memory, Address, Value);

F.4  Read-Write Memory Module: (memory.c)
1 #include <isa.h>
2 #include <memory.h>
3 Value
4 memload(Memory m, Address a) {
5    return m[a];
6 }
7 void
8 memstore(Memory m, Address a, Value v) {
9    m[a] = v;
10 }
# Contents

1 Introduction ....................................................... 1

2 Modeling Hardware with TRS's .................................... 2

3 Identifying State Elements from TRS's .......................... 5

4 Format of a Rule in Bluespec ....................................... 6

5 Identifying Mutually Exclusive Rules ............................... 7

6 Support for Modular Design ......................................... 7

7 Bluespec\(_C\): A TRS with the C Language Syntax ............. 10

8 Concluding Remarks .................................................. 12

A Bluespec\(_{PH}\) Sources for a Simple \(AX\) Instruction Set Processor ......................... 14
   A.1 \(AX\) ISA Definition Module: (isa.hs) ......................... 14
   A.2 \(PB\) Processor Module: (axpb.hs) .......................... 14

B Bluespec\(_{PH}\) Sources for a Renaming/Reordering \(AX\) Processor ......................... 17
   B.1 \(PR\) Processor Module: (axpr.hs) .......................... 17
   B.2 Instruction Template Buffer Module: (itbs.hs) .............. 19
   B.3 Tagged Data Type Definition Module: (tag.hs) ............. 24
   B.4 Tagged Register File Module: (tagregfile.hs) .............. 25

C Bluespec\(_{PH}\) Sources for Library Modules shared by \(PB\) and \(PR\) ......................... 27
   C.1 Read-only Instruction Memory Module: (imemory.hs) ....... 27
   C.2 Read-Write Memory Module: (memory.hs) ................... 27
   C.3 Generic Register File Module: (regfile.hs) .................. 27

D Bluespec\(_C\) Sources for a Simple \(AX\) Processor ................ 29
   D.1 \(AX\) ISA Definition Header File: (isa.h) .................... 29
   D.2 \(AX\) ISA Definition Module: (isa.c) ....................... 29
   D.3 \(PB\) Processor Header File: (axpb.h) ....................... 30
   D.4 \(PB\) Processor Module: (axpb.c) ....................... 30
   D.5 Generic Register File Header File: (regfile.h) ............ 33
   D.6 Generic Register File Module: (regfile.c) .................. 33

E Bluespec\(_C\) Sources for a Renaming/Reordering \(AX\) Processor ......................... 34
   E.1 \(PR\) Processor Header File: (axpr.h) ....................... 34
   E.2 \(PR\) Processor Module: (axpr.c) ....................... 34
   E.3 Instruction Template Buffer Header File: (itbs.h) ......... 39
   E.4 Instruction Template Buffer Module: (itbs.c) ............... 41
   E.5 Tagged Data Type Definition Header File: (tag.h) .......... 44
   E.6 Tagged Register File Header File: (tagregfile.h) .......... 44
   E.7 Tagged Register File Module for \(PR\): (tagregfile.c) ...... 45

47
F  Bluespec$C$ Sources for Library Modules shared by $P_B$ and $P_R$  

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>F.1</td>
<td>Read-only Instruction Memory Header File: (imemory.h)</td>
<td>46</td>
</tr>
<tr>
<td>F.2</td>
<td>Read-only Instruction Memory Module: (imemory.c)</td>
<td>46</td>
</tr>
<tr>
<td>F.3</td>
<td>Read-Write Memory Header File: (memory.h)</td>
<td>46</td>
</tr>
<tr>
<td>F.4</td>
<td>Read-Write Memory Module: (memory.c)</td>
<td>46</td>
</tr>
</tbody>
</table>