LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

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.

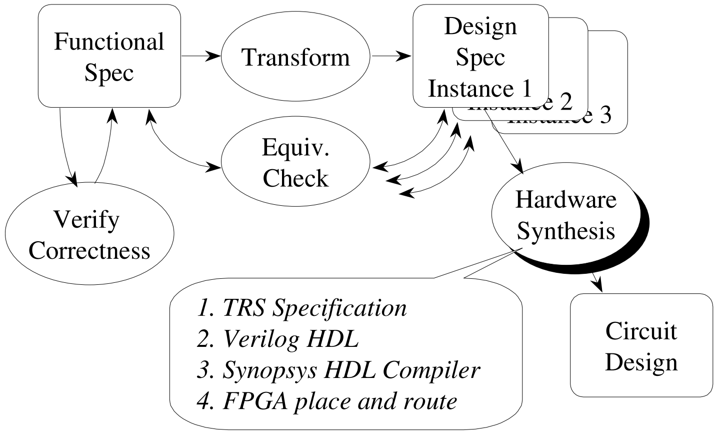

Figure 1: A Highly Automated Hardware Design Flow using High-level Abstractions

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 Bluespec$_{pH}$. Section 3 first discusses the the description and extraction of hardware state elements in TRS's. In Sections 4 thru 6, we then discusses the issues in describing combinational logic. In Section 7, we turn the attention to our experience with Bluespec$_C$. The paper concludes with a few closing remarks in Section 8. Complete Bluespec$_{pH}$ and Bluespec$_C$ 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[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:

$S \ if \ p(S) \rightarrow S'$

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

$$
\begin{array}{llll}
\mathsf{Inst} & \equiv & \mathsf{r} := \mathsf{Loadc}(\mathsf{v}) & \textit{Load-constant Instruction} \\
 & [\!] & \mathsf{r} := \mathsf{Loadpc} & \textit{Load-program-counter Instruction} \\
 & [\!] & \mathsf{r} := \mathsf{Op}(\mathsf{r}_1, \mathsf{r}_2) & \textit{Arithmetic-operation Instruction} \\
 & [\!] & \mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2) & \textit{Branch Instruction} \\
 & [\!] & \mathsf{r} := \mathsf{Load}(\mathsf{r}_1) & \textit{Load Instruction} \\
 & [\!] & \mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2) & \textit{Store Instruction}
\end{array}
$$

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 $\mathcal{P}_{\mathcal{B}}$, and the other models a dynamically scheduled processor with register renaming, $\mathcal{P}_{\mathcal{R}}$. 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 a set of cells with addresses and values. In the $\mathcal{P}_{\mathcal{B}}$ model, the processor state is described as a term 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:

Op Rule

$$
\begin{array}{ll}
 & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{r} := \mathsf{Op}(\mathsf{r}_1, \mathsf{r}_2) \\
\longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{v}],\ \mathsf{prog}) \quad \textit{where} \quad \mathsf{v} = \underline{\mathsf{Op}}(\mathsf{rf}[\mathsf{r}_1], \mathsf{rf}[\mathsf{r}_2])
\end{array}
$$

This rule is applicable to states in which the current instruction is a binary arithmetic operation such as addition. ($\underline{\mathsf{Op}}(\mathsf{rf}[\mathsf{r}_1], \mathsf{rf}[\mathsf{r}_2])$ 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 $\mathcal{P}_{\mathcal{B}}$ are the following:

Loadc Rule

$$
\begin{array}{ll}
 & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{r} := \mathsf{Loadc}(\mathsf{v}) \\
\longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{v}],\ \mathsf{prog})
\end{array}
$$

Loadpc Rule

$$
\begin{array}{ll}
 & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{r} := \mathsf{Loadpc} \\
\longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{ia}],\ \mathsf{prog})
\end{array}
$$

Jz-Jump Rule

$$
\begin{array}{ll}
 & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2) \quad \textit{and} \quad \mathsf{rf}[\mathsf{r}_1] = 0 \\
\longrightarrow & \mathsf{Proc}(\mathsf{rf}[\mathsf{r}_2],\ \mathsf{rf},\ \mathsf{prog})
\end{array}
$$

Jz-NoJump Rule

$$
\begin{array}{ll}
 & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2) \quad \textit{and} \quad \mathsf{rf}[\mathsf{r}_1] \neq 0 \\
\longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf},\ \mathsf{prog})
\end{array}
$$

Load Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{r := Load(r_1)} \\ \longrightarrow & \mathsf{Sys(m,\ \ Proc(ia{+}1,\ rf[r := m[a]],\ prog))} \quad \textit{where} \quad \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

Store Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{Store(r_1, r_2)} \\ \longrightarrow & \mathsf{Sys(m[a := rf[r_1]],\ \ Proc(ia{+}1,\ rf,\ prog))} \quad \textit{where} \quad \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

In the $\mathcal{P}_\mathcal{R}$ 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

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ itbs,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{r := Op(r_1, r_2)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf[r := t],\ itbs} \oplus \mathsf{ITB(ia,\ t := Op(rf[r_1], rf[r_2])),\ prog)} \end{array}$$

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 $\mathcal{P}_\mathcal{R}$. 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

$$\begin{array}{ll} & \mathsf{Proc(pc,\ rf,\ itbs_1} \oplus \mathsf{ITB(ia_1, t := Op(v_1, v_2))} \oplus \mathsf{itbs_2,\ prog)} \\ \longrightarrow & \mathsf{Proc(pc,\ rf,\ itbs_1} \oplus \mathsf{ITB(ia_1, t := \underline{Op}(v_1, v_2))} \oplus \mathsf{itbs_2,\ prog)} \end{array}$$

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 $\mathsf{itbs_2[v/t]}$ means that all instances of tag t in $\mathsf{itbs_2}$ are replaced by value v.

P_R-Value-Forward Rule

$$\begin{array}{ll} & \mathsf{Proc(pc,\ rf,\ itbs_1} \oplus \mathsf{ITB(ia_1, t := v)} \oplus \mathsf{itbs_2,\ prog)} \quad \textit{if} \quad \mathsf{t} \in \mathsf{itbs_2} \\ \longrightarrow & \mathsf{Proc(pc,\ rf,\ itbs_1} \oplus \mathsf{ITB(ia_1, t := v)} \oplus \mathsf{itbs_2[v/t],\ prog)} \end{array}$$

P_R-Value-Commit Rule

$$\begin{array}{ll} & \mathsf{Proc(pc,\ rf,\ itbs_1} \oplus \mathsf{(ia_1\!:\ t := v)} \oplus \mathsf{itbs_2,\ prog)} \quad \textit{if} \quad \mathsf{t} = \mathsf{rf[r]} \\ \longrightarrow & \mathsf{Proc(pc,\ rf[r := v],\ itbs_1} \oplus \mathsf{(ia_1\!:\ t := v)} \oplus \mathsf{itbs_2,\ prog)} \end{array}$$

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

P_R-Value-Discard Rule

$$\begin{array}{ll} & \mathsf{Proc(pc,\ rf,\ itbs_1} \oplus \mathsf{ITB(ia_1, t := \text{-})} \oplus \mathsf{itbs_2,\ prog)} \quad \textit{if} \quad \mathsf{t} \notin \mathsf{rf,\ itbs_2} \\ \longrightarrow & \mathsf{Proc(pc,\ rf,\ itbs_1} \oplus \mathsf{itbs_2,\ prog)} \end{array}$$

The equivalence between the $\mathcal{P}_{\mathcal{B}}$ and $\mathcal{P}_{\mathcal{R}}$ models can be proven by showing that if an initial term can get into a state using the $\mathcal{P}_{\mathcal{B}}$ rules, then it can also get there using the $\mathcal{P}_{\mathcal{R}}$ rules. However, the more interesting result is that $\mathcal{P}_{\mathcal{B}}$ can simulate the state transitions of $\mathcal{P}_{\mathcal{R}}$ 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 $S\ if\ p(S) \rightarrow S'$ 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

$t_j' \neq t_j$, and t_i is not a *don't care* in P, 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, than 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$_{pH}$ 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$_{pH}$'s typing system (adopted from pH), a state element must be declared as a special mutable type identified by the type constructor keyword `MCell`. For example:

```
type Pc           =  MCell InstAddress
type InstAddress  =  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:

```
data RegFile  =  Regfile (Array Int RegCell)
data RegCell  =  MCell Int
```

```
 1 opSubRule (Sys memory proc prog) =
 2         let
 3                 (Proc pc rf)=proc
 4         in
 5                 if ((imemload prog (mFetch pc))==
 6                     RR Sub rd r1 r2) then
 7                         let
 8                                 v1 = rflookup rf r1
 9                                 v2 = rflookup rf r2
10                                 action1 = mStore pc ((mFetch pc)+1)
11                                 action2 = rfupdate rf rd (v1-v2)
12                         in
13                                 ()
```

Figure 3: The $\mathcal{P}_{\mathcal{B}}$ *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:

$S\ if\ p(S) \rightarrow S'$

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:

$S\ if\ p(S) \rightarrow \delta(S)$

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:

$S\ if\ p(S) \rightarrow \delta(S)(S)$.

$\mathcal{P}_{\mathcal{B}}$'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 `imemload` 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 $\mathcal{P}_{\mathcal{B}}$ 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 $\mathcal{P}_{\mathcal{B}}$ 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 $\mathcal{P}_{\mathcal{B}}$ 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$_{pH}$ $\mathcal{P}_{\mathcal{B}}$ 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 $\mathcal{P}_{\mathcal{B}}$ (`regfile.hs` in Appendix C.3). A module description starts with the declaration of the exported interfaces on line 1.

```
 << excerpt from axpb.hs >>

axpbRule (Sys memory proc prog) =
    let
        (Proc pc rf)=proc
    in
        case (imemload prog (mFetch pc)) of
             . . . . . . . . . .
            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
                in
                    case (rflookup rf rc) of
                        0 ->
                            let
                                action1 = mStore pc vt
                            in
                                ()
                        _ ->
                            let
                                 action1 = mStore pc ((mFetch pc)+1)
                            in
                                ()
                     . . . . . . . . . .
```

Figure 4: An Excerpt from the Bluespec$_{pH}$ $\mathcal{P}_{\mathcal{B}}$ 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
15                 dummy=(mStore (rf!r) v)
16         in
17                 ()
```

Figure 5: Module Definition of $\mathcal{P}_{\mathcal{B}}$'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$_{pH}$ 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 memory 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 $\mathcal{P}_{\mathcal{R}}$

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 $\mathcal{P}_{\mathcal{R}}$ 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:
$S\ if\ p(S) \rightarrow \delta(S)$

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,
                     rflookup(regFile, rs) - rflookup(regFile, rt));
9           pc = pc+1;
10      }
11  }
```

Figure 7: The $\mathcal{P}_{\mathcal{B}}$ *Op-Sub Rule* in Bluespec$_C$ Syntax

```
typedef Value RegFile[NumRegName];

extern Value rflookup(RegFile, RegName);
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 `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 `regFile` by the result of the subtraction and incrementing `pc` by `1`. The code segment shown in Figure 7 represents the *Op Rule* of $\mathcal{P}_{\mathcal{B}}$ 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 $\mathcal{P}_{\mathcal{B}}$ in Appendix D.4 actually groups all seven $\mathcal{P}_{\mathcal{B}}$ rules into a single `case` statement.

A Bluespec$_C$ module is composed of a pair of `.h` and `.c` files. Figures 8 and 9 contain `regfile.h` (Appendix D.5) and `regfile.c` (Appendix C.3), the header and the body of the register file module used by $\mathcal{P}_{\mathcal{B}}$. The `.h` header file declares interfaces exported by the register file module. The `.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 $\mathcal{P}_{\mathcal{B}}$ example, this pattern is made up of `imemory`, `memory`, `regFile` and `pc`, which are all declared as global variables elsewhere.

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

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

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

Figure 9: A Bluespec$_C$ Description (regfile.c) of a Register File Module

to simulate the design. Whereas, when a Bluespec$_{pH}$ description is executed as a pH program, the pH run-time system automatically handles the selection and application of rules because a pH program is inherently a TRS. In Bluespec$_C$, 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$_C$ 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 ($\mathcal{P}_\mathcal{B}$ and $\mathcal{P}_\mathcal{R}$) in synthesizable TRS's. Two versions of the Bluespec TRS architectural description language were developed and evaluated in this study. The Bluespec$_{pH}$ source files to describe $\mathcal{P}_\mathcal{B}$ and $\mathcal{P}_\mathcal{R}$ are in Appendices A and B. Appendix C contains the source files for the shared modules used by both $\mathcal{P}_\mathcal{B}$ and $\mathcal{P}_\mathcal{R}$. Appendices D, E, and F contain the corresponding source files in Bluespec$_C$.

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. 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$_{pH}$ and Bluespec$_C$ 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[6] descriptions at the register-transfer level. The resulting Verilog descriptions have also been compiled successfully into FPGA's by Synopsys Design Compiler[5]. These experiences further reinforces the validity of our approach to architectural description and synthesis.

References

[1] Arvind and Xiaowei Shen. Specification of memory models and design of provably correct cache coherence protocols. Technical Report CSG-MEMO-398, MIT Lab. for Comp. Sci., June 1997. Work-in-progress: Not for Distribution without Authors' Permission.

[2] S. P. Harrison and G. L. Steele, Jr. *C: A Reference Manual.* Prentice Hall, 1995.

[3] R. S. Nikhil and Arvind. Programming in pH: A parallel programming language. Book in Progress. Draft [V5], September 1997.

[4] Xiaowei Shen. Processor models. Technical Report CSG-MEMO-400, MIT Lab. for Comp. Sci., July 1997. Work-in-progress: Not for Distribution without Authors' Permission.

[5] Synopsis, Inc. *HDL Compiler for Verilog Reference Manual.*

[6] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

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

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

```
--
-- ax ISA data types
--
data Op = Add | Sub deriving (Eq)
execOp :: Op -> Value -> Value -> Value
execOp op v1 v2 = 0

-- data Value = ndu Int, Bool, Address, InstAddress
type Value = Int
type Address=Int
type InstAddress=Int

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

type RegD=RegName                -- destination register name
type RegC=RegName                -- branch condition register name
type RegT=RegName                -- branch target register name
type RegA=RegName                -- load and store address
type RegV=RegName                -- store value register name
type Reg1=RegName                -- 1st argument register name
type Reg2=RegName                -- 2st argument register name

--
-- ax instruction set
--
data AXInstruction =                Loadc RegD Value
                        |        Loadpc RegD
                        |        RR Op RegD Reg1 Reg2
                        |        Jz RegC RegT
                        |        Load RegA RegD
                        |        Store RegA RegV
```

A.2 $\mathcal{P}_{\mathcal{B}}$ Processor Module: (axpb.hs)

```
module AXPB() where

import "isa"
import "regfile"
import "memory"
import "imemory"
import "mcell"

type Pc = MCell InstAddress
data Processor = Proc Pc (RegFile Value)
type Prog=IMemory
data System = Sys Memory Processor Prog
```

```
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
                        in
                                case (rflookup rf rc) of
                                0 ->
                                        let
                                                action1 = mStore pc vt
                                        in
                                                ()
                                _ ->
                                        let
                                                action1 = mStore pc ((mFetch pc)+1)
                                        in
                                                ()
                Load ra rd ->
                        let
                                va = rflookup rf ra
                                vd = memload memory va
```

```
                                action1 = rfupdate rf rd vd
                        in
                                ()
                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 $\mathcal{AX}$ Processor

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

```
module AXPR() where

import "isa"
import "tag"
import "tagregfile"
import "itbs"
import "memory"
import "imemory"
import "mcell"

data PcCell = RealPc InstAddress | StallPc
type Pc = MCell PcCell
data Processor = Proc Pc TaggedRegFile ITBs
type Prog=IMemory
data System = Sys Memory Processor Prog

-- Ax instruction issue rules

issueRule:: System -> ()
issueRule (Sys memory (Proc pc rf itbs) prog) =
    if (not (itbsfull itbs)) then
        case (mFetch pc) of
           RealPc ia ->
             case (imemload prog ia) of
                Loadc rd v ->
                        let
                                t = enqueue itbs (Loadc' rd (Tvalue v)) ia
                                ---
                                action1 = markDone itbs t v
                                action2 = trfupdateT rf rd t
                                action3 = mStore pc (RealPc (ia+1))
                        in
                                ()
                Loadpc  rd ->
                        let
                                t = enqueue itbs (Loadpc' rd (Tvalue ia)) ia
                                ---
                                action1 = markDone itbs t ia
                                action2 = trfupdateT rf rd t
                                action3 = mStore pc (RealPc (ia+1))
                        in
                                ()
                RR op rd r1 r2 ->
                        let
                                tv1 = trflookup rf r1
                                tv2 = trflookup rf r2
                                t = enqueue itbs (RR' op rd tv1 tv2) ia
                                action1 = trfupdateT rf rd t
                                action2 = mStore pc (RealPc (ia+1))
                        in
```

```
                                ()
                Jz rc rt ->
                        let
                                tvc = trflookup rf rc
                                tvt = trflookup rf rt
                                t = enqueue itbs (Jz' tvc tvt) ia
                                action = mStore pc StallPc
                        in
                                ()
                Load ra rd ->
                        let
                                tva = trflookup rf ra
                                t = enqueue itbs (Load' tva rd) ia
                                action1 = trfupdateT rf rd t
                                action2 = mStore pc (RealPc (ia+1))
                        in
                                ()
                Store ra rv ->
                        let
                                tva = trflookup rf ra
                                tvv = trflookup rf rv
                                t = enqueue itbs (Store' tva tvv) ia
                                action = mStore pc (RealPc (ia+1))
                        in
                                ()

     else
        ()

axprExecuteRule (Sys memory (Proc pc rf itbs) prog) =
    case (instReady itbs) of
        (True, t, ia, (RR'' op v1 v2)) ->
            let
                action1 = markExec itbs t
                v = execOp op v1 v2
                ---
                action2 = markDone itbs t v
            in
                ()
        (True, t, ia, (Jz'' c tia)) | c==0 ->
            let
                action1 = markExec itbs t
                action2 = mStore pc (RealPc tia)
                ---
                action3 = markDone itbs t 0
            in
                ()
        (True, t, ia, (Jz'' c iat)) | not (c==0) ->
            let
                action1=markExec itbs t
                action2 = mStore pc (RealPc (ia+1))
                ----
                action3 = markDone itbs t 0
            in
```

```
                ()
        (True, t, ia, (Load'' a)) ->
            let
                action1 = markExec itbs t
                v = (memload memory a)
                ---
                action2 = markDone itbs t v
            in
                ()
        (True, t, ia, (Store'' a v)) ->
            let
                action1 = markExec itbs t
                ---
                action2 = (memstore memory a v)
                ---
                action3 = markDone itbs t 0
            in
                ()
```

```
axprCommitRule ( Sys memory (Proc pc rf itbs) prog ) =
        case (dequeue itbs) of
        (True, (Done'' t r v)) ->
                let
                        action = (trfupdateV rf r t v )
                in
                        ()

```

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

```
module ITBsModule(ITBs, itbs, itbsfull, enqueue, instReady,
                 markExec, markDone, dequeue,
                 Inst', Loadc', Loadpc', RR', Jz', Load', Store',
                 Inst'', RR'', Jz'', Load'', Store'', Done'',
                 OpCode) where

import Array
import "isa"
import "tag"
import "mcell"
import "qmcell"

-- ITBs interface data types
data Inst' =            Loadc' RegD TV
                |       Loadpc' RegD TV
                |       RR' Op RegD TV1 TV2
                |       Jz' TVC TVT
                |       Load' TVA RegD
                |       Store' TVA TVV

data Inst'' =           RR'' Op Value Value
                |       Jz'' Value InstAddress
```

```
                |       Load'' Address
                |       Store'' Address Value
                |       Done'' Tag RegD Value
                |       Nop''

data OpCode =           LoadcOp
                |       LoadpcOp
                |       RROp Op
                |       JzOp
                |       LoadOp
                |       StoreOp
                |       DoneOp
                deriving (Eq)

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

data ITBs = ITBs (Array Tag IBuf) IdxY IdxO
-- - an array of entries, plus
-- - IdxY, IdxO which index the youngest
--   and oldest entries

data IBuf = IBuf (MCell InstAddress) ICell (MCell Status)
data ICell = ICell (MCell OpCode) (MCell RegD) (MCell TV1) (MCell TV2)
type Idx = MCell Tag
type IdxY = Idx
type IdxO = Idx

opDecode :: Inst' -> OpCode
opDecode inst =
        case inst of
                (Loadc' _ _) -> LoadcOp
                (Loadpc' _ _) -> LoadpcOp
                (RR' op _ _ _) -> RROp op
                (Jz' _ _) -> JzOp
                (Load' _ _) -> LoadOp
                (Store' _ _) -> StoreOp

rdDecode :: Inst' -> RegName
rdDecode inst =
        case inst of
                (Loadc' rd _) -> rd
                (Loadpc' rd _) -> rd
                (RR' _ rd _ _) -> rd
                (Load' _ rd ) -> rd

tv1Decode :: Inst' -> TV
tv1Decode inst =
        case inst of
                (Loadc' _ _) -> (Tvalue 0)
                (Loadpc' _ tv) -> tv
                (RR' _ _ tv _) -> tv
                (Jz' tv _) -> tv
                (Load' tv _) -> tv
                (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()) DoneOp)
                                     (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 idxO) | (nextTag (mFetch idxY)) == (mFetch idxO) -> False
        _ -> True

enqueue :: ITBs -> Inst' -> InstAddress -> Tag
enqueue itbs inst cia =
        case itbs of
{-            (ITBs _ idxY idxO) | (nextTag (mFetch idxY)) == (mFetch idxO) ->
                        ERROR "Buffer Overflow"                        -}
             (ITBs ibuf idxY idxO) ->
                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 idxO) | (mFetch idxY) == (mFetch idxO) ->
                (False, Nop'')
```

```
        (ITBs ibuf idxY idxO) ->
                let
                        (IBuf _ icell status)=ibuf!((mFetch idxO))
                        actionX = mStore status Invalid
                        (ICell _ rd tv _)=icell
                in
                        if (((mFetch status))==Done) then
                                let
                                        idxTemp=(mFetch idxO)
                                        action=mStore idxO
                                                      (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 idxO)=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)
```

```haskell
instReady :: ITBs -> (Bool, Tag, InstAddress, Inst'')
instReady itbs =
     let
        idxS = QMCell T0
     in
        case itbs of
            (ITBs ibuf idxY idxO) | (mFetch idxY) == (mFetch idxO) ->
                let
                     action = qMStore idxS (mFetch idxO)
                in
                     (False, T0, 0, Nop'')
            (ITBs ibuf idxY idxO) ->
                let
                        (IBuf ia icell status) = ibuf!((qMFetch idxS))
                        (ICell op rd tv1 tv2)=icell
                in
                    if ((mFetch status)==Invalid) then
                        let
                            action = qMStore idxS   (mFetch idxO)
                        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
                ()

forwardEach :: ITBs -> Tag -> Value -> Tag -> ()
forwardEach itbs t v t' =
     let
            (ITBs ibuf idxY idxO)=itbs
            (IBuf ia icell status)=ibuf!t'
     in
        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
                    ()
             (ICell _ _ tv0 tv1)| ((mFetch tv1)==(Ttag t)) ->
                let
                        action=(mStore tv1 (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
```

```
import "isa"
--
-- AXPR-specific internal data types
--
data Tag= T0 | T1 | T2 | T3
        deriving (Ix,Ord,Eq)
data TV = Tvalue Value | Ttag Tag deriving (Eq)
type TVC=TV                -- branch condition tag/value
type TVT=TV                -- branch target tag/value
type TVA=TV                -- branch condition tag/value
type TVV=TV                -- store value tag/value
type TV1=TV                -- 1st argument tag/value
type TV2=TV                -- 2st argument tag/value


moreTag :: Tag->Bool
moreTag t = not (t==T3)

nextTag :: Tag->Tag
nextTag t = t

prevTag :: Tag->Tag
prevTag t = t
```

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

```
module TaggedRegFileModule (TaggedRegFile, taggedRegFile, trflookup,
                            trfupdateT,trfupdateV) where

import Array
import "isa"
import "mcell"
import "tag"
import "regfile"

type TaggedRegFile = RegFile TV

taggedRegFile :: TaggedRegFile
taggedRegFile = regFile (Tvalue 0)

trflookup :: TaggedRegFile -> RegName -> TV
trflookup rf r = (rflookup rf r)

trfupdateT :: TaggedRegFile -> RegName -> Tag -> ()
trfupdateT rf r t =
        let
                action=rfupdate rf r (Ttag t)
        in
                ()

trfupdateV :: TaggedRegFile -> RegName -> Tag -> Value -> ()
trfupdateV rf r t v =
        case (rflookup rf r) of
        (Ttag tlast) | (tlast == t) ->
```

```
                 rfupdate rf r (Tvalue v)
         _ ->
                 ()
```

C Bluespec$_{pH}$ Sources for Library Modules shared by $\mathcal{P}_{\mathcal{B}}$ and $\mathcal{P}_{\mathcal{R}}$

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

```
module IMemory(IMemory,imemory,imemload) where

import Array
import "isa"
import "mcell"

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

imemory :: IMemory
imemory = IMemory (listArray (1,3) [(mReplace (mCell()) (Loadc R0 0)),
                                    (mReplace (mCell()) (Loadc R0 0)),
                                    (mReplace (mCell()) (Loadc R0 0)),
                                    (mReplace (mCell()) (Loadc R0 0))])

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

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

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

import Array
import "isa"
import "mcell"

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

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

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

memstore :: Memory -> Address -> Value -> ()
memstore (Memory mem) a v =
        let
                dummy = (mStore (mem!a) v)
        in
                ()
```

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

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

import Array
import "isa"
import "mcell"
```

```
type RegCell a = MCell a
data RegFile a = RegFile (Array RegName (RegCell a))

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

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

rfupdate :: RegFile a -> RegName -> a -> ()
rfupdate (RegFile rf) r v =
        let
                dummy=(mStore (rf!r) v)
        in
                ()
```

D Bluespec$_C$ Sources for a Simple $\mathcal{AX}$ Processor

D.1 $\mathcal{AX}$ ISA Definition Header File: (isa.h)

```
typedef enum enum_MajorOp {
  Loadc, Loadpc, RR, Jz, Load, Store, NumMajorOp
} MajorOp;

typedef enum enum_MinorOp {
  Add, Sub, NumMinorOp
} MinorOp;

typedef int Value;
typedef int Address;
typedef int InstAddress;

typedef enum enum_RegName { R0, R1, R2, R3, NumRegName} RegName;

typedef struct struct_Instruction {
  MajorOp _majorOp : 4;
  RegName _RegD   : 2;
  union {
    struct { int _immediate : 16; } _immediate_instruction;
    struct { RegName _RegS : 2; RegName _RegT : 2; MinorOp _minorOp : 4; } _rr_instruction;
  } _instruction_format;
} Instruction;

#define majorOp(a) ((a)._majorOp)
#define RegD(a) ((a)._RegD)

#define immediate(a) ((a)._instruction_format._immediate_instruction._immediate)
#define RegS(a) ((a)._instruction_format._rr_instruction._RegS)
#define RegT(a) ((a)._instruction_format._rr_instruction._RegT)
#define minorOp(a) ((a)._instruction_format._rr_instruction._minorOp)

#define Reg1 RegS
#define Reg2 RegT
#define RegV RegT
#define RegA RegS
#define RegC RegS

extern Instruction makeInstruction(MajorOp, RegName, RegName, RegName, int, MinorOp);
```

D.2 $\mathcal{AX}$ ISA Definition Module: (isa.c)

```
#include <isa.h>

Instruction makeInstruction(MajorOp majorop,
  RegName rd, RegName rs, RegName rt, int immed, MinorOp minorop) {
  Instruction i;
  majorOp(i) = majorop;
  minorOp(i) = minorop;
  RegD(i) = rd;
  RegS(i) = rs;
```

```
  RegT(i) = rt;
  if (majorop == Loadc) {
    immediate(i) = immed;
  }
  return(i);
}
```

D.3 $\mathcal{P}_{\mathcal{B}}$ Processor Header File: (axpb.h)

```
extern RegFile regFile;
extern Memory memory;
extern Imemory imemory;
```

D.4 $\mathcal{P}_{\mathcal{B}}$ Processor Module: (axpb.c)

```
#include <isa.h>
#include <regfile.h>
#include <memory.h>
#include <imemory.h>
#include <main.h>

RegFile regFile;
Memory  memory;
Imemory imemory;
Address pc;

void
rules() {
  Instruction i;
  i = imemload(imemory, pc);
  switch (majorOp(i)) {
    case Loadc : {
      RegName rd;
      Value v;
      rd = RegD(i);
      v = immediate(i);

      rfupdate(regFile, rd, v);
      pc = pc+1;
      break;
    };
    case Loadpc : {
      RegName rd;
      rd = RegD(i);

      rfupdate(regFile, rd, pc);
      pc = pc+1;
      break;
    };
    case RR : {
      switch (minorOp(i)) {
        case Add: {
          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 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;
        };
      }
      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;
    };
  };
}
```

```
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[114]=makeInstruction(RR,  R1, R0, R0, 0, Add);
imemory[115]=makeInstruction(RR,  R2, R0, R0, 0, Add);
imemory[116]=makeInstruction(RR,  R3, R0, R0, 0, Add);

// increment by 1
imemory[117]=makeInstruction(Loadc,  R3, R0, R0, 1, 0);
imemory[118]=makeInstruction(RR,  R1, R1, R3, 0, Add);

// compare to 10
imemory[119]=makeInstruction(Loadc,  R3, R0, R0, 10, 0);
imemory[120]=makeInstruction(RR,  R2, R3, R1, 0, Sub);

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

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

// function return
imemory[125]=makeInstruction(Loadc,  R3, R0, R0, 10, 0);
imemory[126]=makeInstruction(Load,  R3, R3, R0, 0, 0);
imemory[127]=makeInstruction(Jz,  R0, R0, R3, 0, 0);
```

```
imemory[128]=makeInstruction(RR,  R0, R0, R0, 0, Add);
imemory[129]=makeInstruction(RR,  R0, R0, R0, 0, Add);
}

main() {
  loadImemory();
  while(1) {
    rules();
  }
}
```

D.5 Generic Register File Header File: (regfile.h)

```
typedef Value RegFile[NumRegName];

extern Value rflookup(RegFile, RegName);
extern void  rfupdate(RegFile, RegName, Value);
```

D.6 Generic Register File Module: (regfile.c)

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

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

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

E Bluespec$_C$ Sources for a Renaming/Reordering $\mathcal{AX}$ Processor

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

```
extern TaggedRegFile regFile;
extern Memory memory;
extern Imemory imemory;
```

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

```
#include <isa.h>
#include <tag.h>
#include <regfile.h>
#include <memory.h>
#include <imemory.h>
#include <itbs.h>
#include <main.h>

TaggedRegFile regFile;
Memory  memory;
Imemory imemory;
Address pc;
int stalled;
Itbs itbs;

void
issueRules() {
  Tag t;
  Instruction i;
  Template tmp;

  t = itbsCurrent(itbs);
  if (t != NoTag) {
    if (!stalled) {
      i = imemload(imemory, pc);
      switch (majorOp(i)) {
        case Loadc : {
          RegName rd;
          rd = RegD(i);

  setInstAddress(tmp,pc);
          templateSetMajorOp(tmp, Loadc);
          setValueS(tmp, immediate(i));
  templateSetRegD(tmp,rd);

  enqueue(itbs,tmp);
  trfupdateT(regFile,rd,t);
  pc=pc+1;
  break;
}
        case Loadpc : {
          RegName rd;
          rd = RegD(i);
```

```
  setInstAddress(tmp,pc);
          templateSetMajorOp(tmp, Loadpc);
          setValueS(tmp, pc);
  templateSetRegD(tmp,rd);

  enqueue(itbs,tmp);
  trfupdateT(regFile,rd,t);
  pc=pc+1;
  break;
}
        case RR : {
  switch (minorOp(i)) {
  case Add: {
    RegName rd,rs,rt;
    rd = RegD(i);
    rs = RegS(i);
    rt = RegT(i);

    setInstAddress(tmp,pc);
    templateSetMajorOp(tmp, RR);
    templateSetMinorOp(tmp, Add);
    templateSetRegD(tmp,rd);
    setTVS(tmp, trflookup(regFile,rs));
    setTVT(tmp, trflookup(regFile,rt));

    enqueue(itbs,tmp);
    trfupdateT(regFile,rd,t);
    pc=pc+1;
    break;
  }
  case Sub: {
    RegName rd,rs,rt;
    rd = RegD(i);
    rs = RegS(i);
    rt = RegT(i);

    setInstAddress(tmp,pc);
    templateSetMajorOp(tmp, RR);
    templateSetMinorOp(tmp, Sub);
    templateSetRegD(tmp,rd);
    setTVS(tmp, trflookup(regFile,rs));
    setTVT(tmp, trflookup(regFile,rt));

    enqueue(itbs,tmp);
    trfupdateT(regFile,rd,t);
    pc=pc+1;
    break;
  }
  }
  break;
}
case Jz: {
  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);


  setInstAddress(tmp,pc);
  templateSetMajorOp(tmp, Store);
  setTVS(tmp, trflookup(regFile,ra));
  setTVT(tmp, trflookup(regFile,rv));

  enqueue(itbs,tmp);
  pc=pc+1;
  break;
}
}
    }
  }
}

void
executeRules() {
  Tag t;
```

```
  Instruction i;
  DispatchTemplate tmp;

  tmp=instReady(itbs);

  switch (dispatchTemplateMajorOp(tmp)) {
  case RR: {
    switch (dispatchTemplateMinorOp(tmp)) {
    case Add: {
      Value v= dispatchTemplateValueS(tmp)+dispatchTemplateValueT(tmp);
      markDone(itbs,dispatchTemplateTag(tmp),v);
      break;
    }
    case Sub: {
      Value v= dispatchTemplateValueS(tmp)-dispatchTemplateValueT(tmp);
      markDone(itbs,dispatchTemplateTag(tmp),v);
      break;
    }
    }
    break;
  }

  case Jz: {
    if (dispatchTemplateValueS(tmp)) {
      pc=dispatchTemplateInstAddress(tmp)+1;
      stalled=0;
      markDone(itbs,dispatchTemplateTag(tmp),0);
    } else {
      pc=dispatchTemplateValueT(tmp);
      stalled=0;
      markDone(itbs,dispatchTemplateTag(tmp),0);
    }
    break;
  }
  case Load: {
    Value v=memload(memory,dispatchTemplateValueS(tmp));

    markDone(itbs,dispatchTemplateTag(tmp),v);
    break;
  }
  case Store: {
    Address a=dispatchTemplateValueS(tmp);
    Value v=dispatchTemplateValueT(tmp);

    memstore(memory,a,v);

    markDone(itbs,dispatchTemplateTag(tmp),0);
    break;
  }
  }
}

void
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[114]=makeInstruction(RR,  R1, R0, R0, 0, Add);
imemory[115]=makeInstruction(RR,  R2, R0, R0, 0, Add);
imemory[116]=makeInstruction(RR,  R3, R0, R0, 0, Add);

// increment by 1
imemory[117]=makeInstruction(Loadc,  R3, R0, R0, 1, 0);
imemory[118]=makeInstruction(RR,  R1, R1, R3, 0, Add);
```

```
// compare to 10
imemory[119]=makeInstruction(Loadc,  R3, R0, R0, 10, 0);
imemory[120]=makeInstruction(RR,  R2, R3, R1, 0, Sub);

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

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

// function return
imemory[125]=makeInstruction(Loadc,  R3, R0, R0, 10, 0);
imemory[126]=makeInstruction(Load,  R3, R3, R0, 0, 0);
imemory[127]=makeInstruction(Jz,  R0, R0, R3, 0, 0);

imemory[128]=makeInstruction(RR,  R0, R0, R0, 0, Add);
imemory[129]=makeInstruction(RR,  R0, R0, R0, 0, Add);
}

main() {
  int round=0;
  loadImemory();
  resetRegisters(regFile);

  while(1) {
    round++;
    issueRules();
    executeRules();
    commitRule();


    ruleRetire(itbs);
  }
}
```

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

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

typedef struct struct_CommitTemplate {
  Tag _tag;
  RegName _reg;
  Value _value;
} CommitTemplate;

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

#define commitTemplateTag(d) ((d)._tag)
#define commitTemplateReg(d) ((d)._reg)
#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 dispatchTemplateMajorOp(d) ((d)._majorOp)
#define dispatchTemplateMinorOp(d) ((d)._minorOp)
#define dispatchTemplateInstAddress(d) ((d)._instAddress)

#define dispatchTemplateSetTag(d, v) ((d)._tag = (v))
#define dispatchTemplateSetMajorOp(d, v) ((d)._majorOp = (v))
#define dispatchTemplateSetMinorOp(d, v) ((d)._minorOp = (v))
#define dispatchTemplateSetInstAddress(d, v) ((d)._instAddress = (v))

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

#define dispatchTemplateValueS(d) ((d)._valueS)
#define dispatchTemplateValueT(d) ((d)._valueT)

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 templateMajorOp(t) ((t)._majorOp)
#define templateMinorOp(t) ((t)._minorOp)
#define templateRegD(t) ((t)._RegD)

#define templateSetMajorOp(t,v) ((t)._majorOp = (v))
#define templateSetMinorOp(t,v) ((t)._minorOp = (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)
```

```
#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 (templateMajorOp(t)) {
    case Loadc:
    case Loadpc: {
      status(t) = Done;
```

```
      setValueT(t, 0);
      break;
    }
    case Load: {
      setValueT(t, 0);
    }
    default: {
      status(t) = Valid;
      break;
    }
  }
  young(i) = nextTag(y);
  template(i,y) = t;
}

CommitTemplate
dequeue(Itbs i) {
  CommitTemplate c;
  Template t;
  Tag o;

  o = old(i);
  t = template(i,o);
  if (status(t) == Done) {
    commitTemplateSetTag(c, o);
    commitTemplateSetValue(c, getValueS(t));
    commitTemplateSetReg(c, templateRegD(t));
    return c;
  } else {
    commitTemplateSetTag(c, NoTag);
    return c;
  }
}

void
remove(Itbs i) {
  Tag o;
  o = old(i);
  setStatus(template(i,o), Invalid);
  setOld(i, nextTag(o));
}

void
ruleRetire(Itbs i) {
  CommitTemplate c;
  Template t;
  Tag o;

  o = old(i);
  t = template(i,o);
  if ((!itbsEmpty(i)) && (status(t) == Invalid)) {
    remove(i);
  }
}
```

```
DispatchTemplate
instReady(Itbs i) {
  Tag t;
  DispatchTemplate d;
  int found = 0;

  for (t = 0; t < NumTag; t++) {
    Template tmp;
    tmp = template(i,t);
    if ((!found) && (status(tmp) == Valid) && isValueT(tmp) && isValueS(tmp)) {
      if (((templateMajorOp(tmp) != Load) && (templateMajorOp(tmp) != Store)) ||
          (t == old(i))) {
        found = 1;
        dispatchTemplateSetTag(d, t);
        dispatchTemplateSetInstAddress(d, instAddress(tmp));
        dispatchTemplateSetMajorOp(d, templateMajorOp(tmp));
        dispatchTemplateSetMinorOp(d, templateMinorOp(tmp));
        dispatchTemplateSetValueS(d, getValueS(tmp));
        dispatchTemplateSetValueT(d, getValueT(tmp));
      }
    }
  }
  if (found) {
    return d;
  } else {
    dispatchTemplateSetTag(d, NoTag);
    return d;
  }
}

void
markExec(Itbs i, Tag t) {
  setStatus(template(i, t), Executing);
}

void
markDone(Itbs i, Tag t, Value v) {
  Template tmp;
  tmp = template(i, t);
  if ((templateMajorOp(tmp) == Store) ||
      (templateMajorOp(tmp) == Jz)) {
    setStatus(template(i, t), Invalid);
  } else {
    setStatus(template(i, t), Done);
    setValueS(template(i, t), v);
    forward(i, t, v);
  }
}

void
forward(Itbs i, Tag t, Value v) {
  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)

```
typedef enum enum_Tag { T0, T1, T2, T3, NumTag, NoTag } Tag;

typedef struct struct_TagValue {
  int _flag:1;
  union {
    Tag _tag;
    Value _value;
  } _tag_or_value;
} TagValue;

#define isTag(tv) ((tv)._flag)
#define isValue(tv) (!((tv)._flag))

#define setTag(tv, t) { \
  ((tv)._flag = 1); ((tv)._tag_or_value._tag=(t)); \
}
#define setValue(tv, v) { \
  ((tv)._flag = 0); ((tv)._tag_or_value._value=(v)); \
}

#define getTag(tv) ((tv)._tag_or_value._tag)
#define getValue(tv) ((tv)._tag_or_value._value)

#define moreTag(t) ((t) != (NumTag-1))
#define nextTag(t) (((t)+1)%NumTag)
#define prevTag(t) ((((t)+NumTag)-1)%NumTag)
```

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

```
typedef Value RegFile[NumRegName];
typedef TagValue TaggedRegFile[NumRegName];

extern Value rflookup(RegFile, RegName);
extern void  rfupdate(RegFile, RegName, Value);

extern void  trfupdateV(TaggedRegFile f, RegName r, Tag t, Value v);

extern void  trfupdateT(TaggedRegFile, RegName, Tag);
```

```
extern TagValue trflookup(TaggedRegFile, RegName);
```

E.7 Tagged Register File Module for $\mathcal{P}_{\mathcal{R}}$: (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
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$_C$ Sources for Library Modules shared by $\mathcal{P}_\mathcal{B}$ and $\mathcal{P}_\mathcal{R}$

F.1 Read-only Instruction Memory Header File: (imemory.h)

```
#define SizeImemory 0x10000

typedef Instruction Imemory[SizeImemory];

extern Instruction imemload(Imemory, InstAddress);
```

F.2 Read-only Instruction Memory Module: (imemory.c)

```
#include <isa.h>
#include <imemory.h>

Instruction
imemload(Imemory m, InstAddress a) {
  return m[a];
}
```

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

```
#define SizeMemory 0x10000

typedef Value Memory[SizeMemory];

extern Value memload(Memory, Address);
extern void  memstore(Memory, Address, Value);
```

F.4 Read-Write Memory Module: (memory.c)

```
#include <isa.h>
#include <memory.h>

Value
memload(Memory m, Address a) {
  return m[a];
}

void
memstore(Memory m, Address a, Value v) {
  m[a] = v;
}
```

Contents