Superscalar Processors
via Automatic Microarchitecture Transformations

by

Mieszko N. Lis

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Copyright © 2000 Mieszko N. Lis. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science
May 12, 2000

Certified by

Arvind
Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Superscalar Processors
via Automatic Microarchitecture Transformations
by
Mieszko N. Lis

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2000, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract
Modern microprocessors issue multiple instructions per cycle to achieve high performance. But developing a multiple-issue processor is much harder than designing a single-issue version: the added constraints on scheduling and hazard resolution render it a far more time-consuming and bug-prone process. In this thesis, we present a method for automatically deriving a multiple-issue processor from its single-issue specification using a Term Rewriting Systems (TRS) formalism. Implemented as a stage in a TRS synthesis tool, this technique derives a provably correct implementation given only the original specification and the desired issue width. We then discuss a successful application of our method to pipeline buffer elision, an important step in efficient pipeline verification.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science and Engineering
Acknowledgments

I am deeply indebted to Professor Arvind for his time and guidance, and for the opportunity to do fascinating research.

Members of the Computation Structures Group have created a uniquely stimulating research environment. I am especially indebted to James C. Hoe for his time and contributions to this research, and to Lennart Augustsson for research contributions and the many instructive discussions. I will always cherish the countless fascinating and entertaining conversations with Jan-Willem Maessen and other fellow students.

The students, faculty, and staff of the Massachusetts Institute of Technology have helped me learn and grow in many unobtrusive yet significant ways. I could not have envisioned a better education.

I want to thank the Intel Corporation and the Intel Foundation for supporting the research presented here.

My debt to Lynn Suwatanapongched for patiently keeping me sane and happy exceeds my ability to express. She has been a refuge and an inspiration.

Finally, I am immensely grateful to my parents and my little sister for their love and support, and for having sacrificed so much for my education. They deserve far more gratitude than I could ever express.
For my parents,
Wiesława K. and Norbert Lis
# Contents

1 Introduction .................................................. 11
   1.1 Background ............................................. 11
   1.2 Contributions .......................................... 11
   1.3 Related Work ........................................... 11

2 ATS Semantics .................................................. 13
   2.1 Example: Fibonacci ....................................... 13
   2.2 Rule Semantics .......................................... 13
       2.2.1 Rules ............................................. 13
       2.2.2 Values ........................................... 14
       2.2.3 Expressions ...................................... 14
       2.2.4 Actions .......................................... 14
   2.3 State Semantics ......................................... 15
       2.3.1 Registers ......................................... 15
       2.3.2 Arrays ........................................... 15
       2.3.3 Queues ........................................... 15
       2.3.4 Input registers .................................. 16
       2.3.5 Output registers ................................ 16
   2.4 Synthesis Issues ......................................... 16

3 Rule Composition ............................................... 17
   3.1 Example: SuperFibonacci .................................. 17
   3.2 Definitions .............................................. 18
   3.3 The Composition Algorithm ............................... 19
       3.3.1 Composing expressions and actions ................ 19
       3.3.2 The predicate .................................... 20
       3.3.3 The actions ...................................... 20
   3.4 Composition Details ..................................... 20
       3.4.1 Empty actions .................................... 20
       3.4.2 Register operations ................................ 20
       3.4.3 Array operations .................................. 20
       3.4.4 Queue operations .................................. 21
       3.4.5 Output operations .................................. 21
       3.4.6 Input operations ................................... 21

4 Cross-product Transformations ................................. 23
   4.1 Terminology .............................................. 23
   4.2 Example: Multiple Loads .................................. 23
   4.3 Cross Product Details .................................... 24
   4.4 Efficiency ................................................ 24
   4.5 Synthesis Issues ......................................... 25
Chapter 1

Introduction

1.1 Background

Advances in VLSI technology have enabled designers to produce increasingly complex microprocessors: as the decreasing size of transistors has boosted the chip area available for logic, modern processors have exploited instruction level parallelism by issuing multiple instructions per cycle [2]. This improvement in speed comes at a steep cost, however: design and verification have become significantly more complex and time-consuming.

Recently, Term Rewriting Systems (TRS) have been used to describe and verify microprocessors [1, 5] and cache coherence protocols [9]. Hardware synthesis from TRS [3] has enabled quick architecture exploration, as TRSs can be quickly pipelined and superscaled [4]. But, while synthesis has been successfully automated, architectural transformations have remained a largely manual task.

1.2 Contributions

We first present an algorithm for composing rules in a synthesizable TRS framework. Next, we employ composition in a technique for automatically deriving multiple-issue systems—including microprocessors—from their single-issue TRS descriptions, and in a method for eliminating pipeline buffers, an important step in pipeline verification [6].

The techniques presented here have been implemented as source-to-source transformation stages in a TRS compiler. They require no user guidance and generate provably correct hardware.

1.3 Related Work

The foundations for this thesis lie in research on employing Term Rewriting Systems to specify and verify microprocessors [1] and for hardware synthesis [3]. In particular, Hoe and Arvind [4] develop a methodology for deriving superscalar processors in TRS.

Semi-automatic pipeline verification by collapsing pipeline stages has been implemented by Levitt and Olukotun [6]. Matthews and Launchbury [7] employ this idea to derive an elegant microarchitecture algebra for the Hawk specification language, and use it to collapse a microprocessor pipeline.
Chapter 2

ATS Semantics

This chapter presents a brief introduction to Abstract Transition Systems (ATS), a language for non-deterministic finite automata. The ATS syntax described here is a straightforward extension of an intermediate representation used in hardware synthesis from TRS; extensions include a queue reduction operator and multi-value operations. A more thorough treatment of ATS, including algorithms for synthesis and translation from TRS, may be found in [3].

2.1 Example: Fibonacci

Figure 2-1 shows an ATS that iteratively computes the fifth Fibonacci number. The state specifies what storage is required at runtime, and the rules describe how this state may be modified during execution.

When the Fibonacci ATS is synthesized in hardware, n, prev, and ans are instantiated as 32-bit registers. The Fibonacci rule turns into combinational logic that updates the state.

The ATS is executed by repeatedly applying the Fibonacci rewrite rule. As is customary in synchronized hardware, all registers are read at the beginning of each application and written at the end; the assignments (←) are thus performed in parallel. Executing the ATS yields the fifth Fibonacci number (underlined) in four transitions (the tuple \((n, \text{prev}, \text{ans})\) represents the ATS state):

\[(5, 0, 1) \rightarrow (4, 1, 1) \rightarrow (3, 1, 2) \rightarrow (2, 2, 3) \rightarrow (1, 3, 5).\]

When \(n = 1\), the Fibonacci rule no longer applies and execution halts; register ans holds the result.

More complex examples of ATS descriptions may be found in the Appendices.

2.2 Rule Semantics

Each ATS comprises an initial state and a set of rewrite rules. We first describe how rules modify the state, and then expound on the various kinds of state.

2.2.1 Rules

Each rule \((\rho)\) may be divided into two parts: the predicate \((\pi)\), which decides whether to apply the rule, and the set of actions \((\delta)\), which describe the new state; symbolically, \(\rho = (\pi, \delta)\). In a denotational semantics, each rule is a function on a state \(s\):

\[\rho = \lambda s . \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s.\]
We shall call a rule applicable if its predicate is satisfied.

In ATS, a computation is a sequence of rule applications called transitions. Rules are atomic: during each transition, a rule is either applied entirely or not at all. Computations in which one rule is applied in every transition completely specify the valid behavior of the ATS, and are obtained as follows:

1. Let $s$ be the initial state;
2. If no rules are applicable, halt;
3. Let $\rho$ be an applicable rule;
4. Let $s$ be $\rho s$;
5. Go to step 2.

Note that step 3 is free to choose any applicable rule; if there is more than one, the ATS is non-deterministic and may have many legal behaviors. Moreover, because step 2 only halts if there are no applicable rules, the ATS may not terminate. These qualities are, in fact, desirable for describing hardware: pipelined microprocessor ATSs are neither deterministic nor terminating.

While valid behavior is specified by single-rule transitions, an ATS implementation may choose to apply multiple rules per transition provided that sequential semantics are preserved. A processor pipeline ATS (see Appendix B) is a good example: rules in separate pipeline stages are usually independent of each other and, in an efficient hardware implementation, fire simultaneously [3].

### 2.2.2 Values

A value in ATS is a bit vector of a specific width. The language is strongly typed: the type of a value is its bit width. In practice, the bits are frequently treated as numbers or boolean values; the predicate of each rule, for example, computes a one-bit value interpreted in a boolean context.

### 2.2.3 Expressions

ATS expressions are similar to their mathematical counterparts: each expression has one value and may contain other expressions. In a hardware implementation, expressions are rendered as combinational logic.

The simplest expression is a constant. A constant specifies a value as well as a bit width: the constant $77_8$ stands for 01001101, the 8-bit representation of 77.

Expressions may also represent the values of state elements. In Figure 2-1, for example, the expression $n$ stands for the value of register $n$. These values are read at the beginning of each transition and are not affected by any state changes within that transition. Other ATS expressions that read state are described in detail in Section 2.3 and summarized in Figure 2-2.

As in mathematics, many expressions may be combined into one with operators. The ATSs in this thesis employ several primitive operators, including addition ($5_4 + 9_4 = 14_4$), bitwise negation ($\neg 11_2 = 01_2$) and conjunction ($11_1 \land 01_1 = 01_1$), bit selection ($5_2|0_1 = 1_2$), and bit concatenation ($11_1 \| 01_1 = 2_2$). Most operator types are parametrized by bit width.

### 2.2.4 Actions

Actions are state transformers: they describe how the state is modified when a rule is applied. Each rule $(\pi, \delta)$ has one action $\delta.s$ for every state term $s$; in the Fibonacci rule, for example, the action $\delta.n$ is "$n \leftarrow n - 1$" (Figure 2-1).

The empty action (e) is the simplest ATS action: it specifies that the state term remains unmodified. (ATS listings in this thesis omit empty actions for brevity.) Other actions and their semantics are described below.
### 2.3 State Semantics

An ATS may contain a number of named state elements, each of which is a register, an array, a queue, an input register, or an output register. State may not be added or removed during execution.

Some state elements are exported from the ATS. Exported elements may be read—and perhaps modified—independently of ATS rules; in a hardware implementation, they correspond to wires interfacing the ATS to other elements. Semantically, exported elements are subject to external rules that may fire during any transition. Among ATS elements presented here, input registers and output registers are exported.

Figure 2-2 shows a summary of ATS state elements; detailed semantics follow.

#### 2.3.1 Registers

Registers are internal locations that store values of a specific bit width. Because registers are not exported, each holds its value until explicitly modified by some ATS rule. In rules, registers may be read or written, but each register may be written only once per transition.

#### 2.3.2 Arrays

Arrays store a fixed number of elements, all of which are values of the same type, addressed via indices of a specific type. Like registers, elements may be read and written in rules, and preserve their values unless explicitly modified. An arbitrary number of distinct array elements may be read and written during the same rule.

#### 2.3.3 Queues

By far the most complex state elements, queues store values with FIFO semantics. The number of slots in a queue is finite but not defined and not accessible in the ATS; such decoupling enforces useful abstraction and is essential in verification. Expressions and actions in rules only imply a minimum number of slots for each queue.

Rules may test if the queue has a specified number of free slots (free?) or full slots (full?), and, after ensuring enough full slots, read an element indexed from the head of the queue (head). Expressions may also be reduced over the queue, from the oldest element to the youngest (fold); for example, the expression

\[
q.\text{fold}(\lambda elt . \lambda sofar . \text{if } elt > sofar \text{ then } elt \text{ else sofar}, 0_{32})
\]

finds the largest element of \( q \) by successively assigning each element to \( elt \) and the largest element so far, starting from 0, to sofar.
A queue may be modified by adding some elements to the tail in left-to-right order (enq), removing a number of elements from the head (deq), or removing all elements (deqall). For example,

\[ q.\text{enq}(1_{32}, 2_{32}, 3_{32}) \]

adds the 32-bit values 1, 2, and 3 to a queue \( q \) (1 first, then 2, and finally 3).

### 2.3.4 Input registers

Input registers are exported read-only registers. They may only be read, and their values may change between transitions on account of external rewrites.

### 2.3.5 Output registers

Output registers work much like ordinary registers: they may be read or written in rules. Unlike registers, however, they are exported, and may be read outside of the ATS.

### 2.4 Synthesis Issues

In theory, every ATS can be turned into a circuit. In practice, however, the cost of physical resources limits ATS elements that can be synthesized in hardware.

Arrays, for example, are synthesized as register files. Because registered storage consumes many gates, only relatively small arrays are feasible; large memories are usually accessed via external interfaces. Similarly, the number of ports available on a register file drastically increases cost; therefore, the practical number of coincident array reads and writes is limited, as is the number of simultaneously accessed queue elements.

For a complete discussion of hardware synthesis, see [3].
Chapter 3

Rule Composition

In the previous chapter, we have described how ATSs are used to specify synthesizable systems, and noted that an efficient hardware implementation might fire multiple independent rules in one transition. For the transformations described in the following chapters, however, we also require the effect of simultaneously applying multiple rules that are not independent. This chapter describes an algorithm for composing such ATS rules while preserving their sequential semantics.

3.1 Example: SuperFibonacci

The Fibonacci ATS from last chapter (Figure 2-1) computes the \( n \)th Fibonacci number in \( n - 1 \) transitions. Using composition, we derive a SuperFibonacci ATS that requires only half the transitions of the original for the same task.*

By composing the original Fibonacci rule with itself, we derive a SuperFibonacci rule that encapsulates the semantics of applying Fibonacci twice:

\[
\varphi_1 = \text{when } n > 1_{32} \\
\quad n \leftarrow n - 1_{32} \\
\quad \text{prev} \leftarrow \text{ans} \\
\quad \text{ans} \leftarrow \text{prev} + \text{ans},
\]

\[
\varphi_2 = \text{when } n > 1_{32} \\
\quad n \leftarrow n - 1_{32} \\
\quad \text{prev} \leftarrow \text{ans} \\
\quad \text{ans} \leftarrow \text{prev} + \text{ans}.
\]

First, to preserve the sequential semantics of applying \( \varphi_2 \) after \( \varphi_1 \), we replace expressions in \( \varphi_2 \) with the values they acquired in \( \varphi_1 \): \( n \) with \( n - 1_{32} \), \( \text{prev} \) with \( \text{ans} \), and \( \text{ans} \) with \( \text{prev} + \text{ans} \) (the substitutions are set in bold):

\[
\varphi_1 = \text{when } n > 1_{32} \\
\quad n \leftarrow n - 1_{32} \\
\quad \text{prev} \leftarrow \text{ans} \\
\quad \text{ans} \leftarrow \text{prev} + \text{ans},
\]

\[
\varphi'_2 = \text{when } n - 1_{32} > 1_{32} \\
\quad n \leftarrow n - 1_{32} - 1_{32} \\
\quad \text{prev} \leftarrow \text{prev} + \text{ans} \\
\quad \text{ans} \leftarrow \text{ans} + \text{prev} + \text{ans}.
\]

(For readability, we’ll simplify “\( n - 1_{32} > 1_{32} \)” to “\( n > 2_{32} \),” and “\((n - 1_{32}) - 1_{32}\)” to “\( n - 2_{32} \)”)

As we already said, SuperFibonacci should apply whenever the original Fibonacci rule would apply twice. In terms of the predicates of \( \varphi_1 \) and \( \varphi'_2 \),

\[
\pi_c = \pi_1 \land \pi'_2 = (n > 1_{32}) \land (n > 2_{32}) = (n > 2_{32}).
\]

Finally, we compose the actions on terms \( n \), \( \text{prev} \), and \( \text{ans} \). Because all three are registers, the values written by \( \varphi'_2 \) prevail. Combining the new predicate and the composite actions, we obtain the SuperFibonacci rule:

\[
\varphi_c = \text{when } \pi_c \rightarrow \\
\quad n \leftarrow n - 1_{32} \\
\quad \text{prev} \leftarrow \text{ans} \\
\quad \text{ans} \leftarrow \text{prev} + \text{ans}.
\]

* \( \lceil \frac{n - 1}{2} \rceil \) transitions, to be precise.
\[ \varphi_c = \text{when } n > 2^{32} \]
\[ n \leftarrow n - 2^{32} \]
\[ \text{prev} \leftarrow \text{prev} + \text{ans} \]
\[ \text{ans} \leftarrow \text{ans} + \text{prev} + \text{ans}. \]

Execution of the SuperFibonacci ATS yields the fifth Fibonacci number (underlined) in the desired two transitions:

\[ (5, 0, 1) \rightarrow (3, 1, 2) \rightarrow (1, 3, 5). \]

### 3.2 Definitions

We noted earlier that a composition \( \varphi_c = \varphi_2 \circ \varphi_1 \) should preserve the semantics of applying \( \varphi_1 \) and \( \varphi_2 \) sequentially. This is because we expect \( \varphi_c \) to do the job of \( \varphi_1 \) and \( \varphi_2 \) (Chapter 4), and even to replace them altogether (Chapter 5). We therefore require that:

1. applying \( \varphi_c \) always yield the same state as consecutively applying \( \varphi_1 \) and \( \varphi_2 \);
2. \( \varphi_c \) be applicable if and only if \( \varphi_1 \) and \( \varphi_2 \) may be applied consecutively;*
3. no exported state be modified by both \( \varphi_1 \) and \( \varphi_2 \).

Not every pair of rules has a valid composition. Consider, for example, composing executeJzTaken and executeLoad from the four-stage AX pipeline (Appendix C):

**executeJzTaken:**
\[
\text{when } \text{xbuf.full?(1)} \\
\land \text{xbuf.head(1)_{1:0} = 2_2} \\
\land \text{xbuf.head(1)_{3:42} = 2_32} \\
\text{pc} \leftarrow \text{xbuf.head(1)_{4:10}} \\
\text{ibuf.deqall} \\
\text{xbuf.deqall},
\]

**executeLoad:**
\[
\text{when } \text{xbuf.full?(1)} \land \text{wbuf.free?(1)} \\
\land \text{xbuf.head(1)_{1:0} = 3_2} \\
\land \text{xbuf.deq(1)} \\
\text{wbuf.enq(xbuf.head(1)_{6:2}} \\
\text{wbuf.deqall},
\]

\[ \text{++ dmem[xbuf.head(1)_{3:42}]} \].

Immediately after **executeJzTaken** fires, queue \( \text{xbuf} \) is empty, so **executeLoad** cannot apply. What, then, of composing **executeLoad** with **executeJzTaken**? The composition algorithm returns the special value **NOTHING**, indicating that the rules never apply in a sequence and their composition is empty. We shall call such compositions **impotent**.

Even so, some rules which happily fire in a sequence cannot be composed. For instance, **executeLoad** (above) is ordinarily followed by **writeback**:

**writeback:**
\[
\text{when } \text{wbuf.full?(1)} \\
\land \text{rf[wbuf.head(1)_{36:32} = wbuf.head(1)_{31:0}]} \\
\land \text{wbuf.deq(1)}.
\]

In fact, the correctness of the AX pipeline demands it. To compose them, we need to compose head with enq:

\[ q.\text{head}(n) \circ q.\text{enq}(x_1, \ldots, x_m) \Rightarrow \begin{cases} 
\text{head}(n) & \text{if } n \leq |q|, \\
\text{x}_{n-|q|} & \text{if } n > |q| \land n \leq m + |q|,
\end{cases} \]

where \( |q| \) denotes the number of elements in \( q \). But the ATS queue abstraction does not (and should not) allow access to \( |q| \), and the rules cannot be composed without it. The composition of **writeback** with **executeLoad** is therefore **undefined**.

---

*The attentive reader is sure to ask, "shouldn't we make sure that no other rules fire between \( \varphi_1 \) and \( \varphi_2 \)?" The answer is yes, but not here: the issue at hand is not of how composition is done, but rather of how it is used. The composition algorithm only knows about the two rules it is composing, and promises only to capture the semantics of \( \varphi_1 \) and \( \varphi_2 \) being applied in a sequence. An in-depth discussion may be found in Chapter 5.

**In Chapter 5 we explain how to compose **writeback** with **executeLoad**.
Other rule pairs cannot be composed because they violate the I/O semantics of ATS. Consider composing rules with actions that modify an output register \( o \) in different ways:

\[
o \leftarrow 5 \circ o \leftarrow 7.
\]

We cannot compose them to

\[
o \leftarrow 7,
\]
as we would actions on ordinary registers, because ATS semantics require that all exported values appear in their original order.

Yet other compositions fail only because ATS lacks an operation that embodies some behavior. The `decodeLoadPC` and `executeJzTaken` rules, for example, cannot be composed because ATS syntax does not provide an action that combines `enq` with `deqall`:

\[
\begin{align*}
\text{executeJzTaken:} & \quad \text{decodeLoadPC:} \\
\text{when } xbuf.full?(1) \land xbuf.head(1)_{1..0} = 0_2 & \quad \text{when } ibuf.full?(1) \land xbuf.free?(1) \\
\land xbuf.head(1)_{73..42} = 0_{32} & \land ibuf.head(1)_{27..25} = 1_3 \\
\text{pc } \leftarrow xbuf.head(1)_{11..10} & \quad \text{ibuf.deqall(1)} \\
\text{ibuf.deqall} & \quad xbuf.enq(ibuf.head(1)_{63..32} ++ 0_{32} ++ 3_3 \\
\text{xbuf.deqall}, & \quad ++ \text{ibuf.head(1)_{20..16} ++ 2_2}).
\end{align*}
\]

In all such cases, the composition is undefined and the algorithm returns the special value `UNDEFINED`, indicating that the rules may fire sequentially but a composite rule cannot be constructed.

### 3.3 The Composition Algorithm

Given two rules, \( \varphi_1 \) and \( \varphi_2 \), the composition algorithm produces either (1) a composite rule \( \varphi_c \), (2) the special value `NOTHING`, or (3) the special value `UNDEFINED`. In the first two cases, the result may be used to replace all sequential applications of \( \varphi_1 \) and \( \varphi_2 \).

First, we update expressions and actions in \( \varphi_2 \) to reflect the effects of \( \varphi_1 \). Next, we compute the composite predicate \( \pi_c \) and the composite actions \( \delta_c \). Finally, we combine the two to form \( \varphi_c \). If, at any point, the composition is discovered to be impotent or undefined, an exceptional value—`NOTHING` or `UNDEFINED`, respectively—is returned.

### 3.3.1 Composing expressions and actions

Composing an expression \( x \) with a set of actions \( \delta \) produces an expression equivalent to evaluating \( x \) after applying \( \delta \). To compose \( x \) with \( \delta \),

1. let \( x' \) be \( x \) with all of its subexpressions recursively composed with \( \delta \);
2. if any composition returned `UNDEFINED`, return `UNDEFINED`;
3. if any composition returned `NOTHING`, return `NOTHING`;
4. if \( x' \) accesses some state element \( s \), return \((x' \circ \delta.s)\); otherwise return \( x' \).

Similarly, composing an action \( \alpha \) on a state element \( s \) with a set of actions \( \delta \) produces an action with the effect of applying \( \alpha \) after \( \delta \). To compose \( \alpha \) with \( \delta \),

1. let \( \alpha' \) be \( \alpha \) with all of its subexpressions recursively composed with \( \delta \);
2. if any composition returned `UNDEFINED`, return `UNDEFINED`;
3. if any composition returned `NOTHING`, return `NOTHING`;
4. return \((\alpha' \circ \delta.s)\).

Section 3.4 defines \( x \circ \delta.s \) and \( \alpha \circ \delta.s \) for every ATS state element \( s \).
3.3.2 The predicate

The composite rule $\varrho_c$ should be applicable whenever (1) $\varrho_1$ is applicable, and (2) $\varrho_2$ is applicable after $\varrho_1$ fires. The former condition is the predicate $\pi_1$; the latter is the predicate $\pi_2$ composed with $\delta_1$ using the composition procedure above:

$$\pi_c = \pi_1 \land (\pi_2 \circ \delta_1).$$

3.3.3 The actions

Actions in the composite rule must reflect state modifications from both rules. They are formed by composing actions from $\delta_2$ with $\delta_1$ piecewise for all state elements $s$:

$$\forall s : \delta_c.s = \delta_2.s \circ \delta_1.$$

Detailed composition rules follow.

3.4 Composition Details

Composition is trivial when the action—or one of the actions—is $\varepsilon$. In general, however, composition is dictated by the semantics of the modified state element: a register, for example, sees only the later of two writes, while an array sees all writes to distinct elements.

3.4.1 Empty actions

Composition with the empty action $\varepsilon$ preserves the expression:

$$x \circ \varepsilon \Rightarrow x,$$

or the other action:

$$\alpha \circ \varepsilon \Rightarrow \alpha,$$

$$\varepsilon \circ \alpha \Rightarrow \alpha.$$

3.4.2 Register operations

All expressions in $\varrho_2$ that read a register $r$ must be bypassed with $r$'s new value—if any—acquired in $\varrho_1$:

$$r \circ (r \leftarrow x_1) \Rightarrow x_1.$$

Actions in the composite rule must also reflect the latest value written to $r$:

$$(r \leftarrow x_2) \circ (r \leftarrow x_1) \Rightarrow (r \leftarrow x_2).$$

3.4.3 Array operations

Array operations may be thought of as operations on indexed collections of registers, and are composed very much like registers. Because the indices are only known at runtime, however, $\varrho_c$ must compare each index referenced in $\varrho_2$ against those of all elements written in $\varrho_1$:

$$a[i_2] \circ (a[i_1] \leftarrow x_1) \Rightarrow \text{if } i_2 = i_1 \text{ then } x_1 \text{ else } a[i_2],$$

and in actions:

$$(a[i_2] \leftarrow x_2) \circ (a[i_1] \leftarrow x_1) \Rightarrow \text{if } i_2 = i_1 \text{ then } a[i_2] \leftarrow x_2 \text{ else } (a[i_1] \leftarrow x_1; a[i_2] \leftarrow x_2).$$
3.4.4 Queue operations

Composing queues is much more complicated than composing registers and arrays. In particular, some compositions are impotent or undefined. For brevity, the descriptions below include only defined compositions.

After deqall

Since after q.deqall the queue is empty, reading or removing any elements is impotent, and tests for full slots evaluate to false. Reduction over q yields its zero.

\[ q.\text{full?}(n) \circ q.\text{deqall} \Rightarrow 0_1 \]
\[ q.\text{head}(n) \circ q.\text{deqall} \Rightarrow \text{NOTHING} \]
\[ q.\text{fold}(f,z) \circ q.\text{deqall} \Rightarrow z \]
\[ q.\text{deq}(n) \circ q.\text{deqall} \Rightarrow \text{NOTHING} \]
\[ q.\text{deqall}(\ldots) \circ q.\text{deqall} \Rightarrow q.\text{deqall}. \]

After deq

Operations reading from q, testing for full or free slots, or removing elements from q are adjusted to bypass the removed elements.

\[ q.\text{full?}(n) \circ q.\text{deq}(k) \Rightarrow q.\text{full?}(n + k) \]
\[ q.\text{free?}(n) \circ q.\text{deq}(k) \Rightarrow \begin{cases} 0_1 & \text{if } n \leq k \\ q.\text{free?}(n - k) & \text{if } n > k \end{cases} \]
\[ q.\text{head}(n) \circ q.\text{deq}(k) \Rightarrow q.\text{head}(n + k) \]
\[ q.\text{deq}(n) \circ q.\text{deq}(k) \Rightarrow q.\text{deq}(n + k) \]
\[ q.\text{deqall} \circ q.\text{deq}(k) \Rightarrow q.\text{deqall}. \]

After enq

Tests to detect the number of full or empty slots and reductions must be updated to account for the additional elements; enqueue operations must combine all elements to be added.

\[ q.\text{full?}(n) \circ q.\text{enq}(x_1, \ldots, x_k) \Rightarrow \begin{cases} 1_1 & \text{if } n \leq k \\ q.\text{full?}(n - k) & \text{if } n > k \end{cases} \]
\[ q.\text{free?}(n) \circ q.\text{enq}(x_1, \ldots, x_k) \Rightarrow q.\text{free?}(n + k) \]
\[ q.\text{fold}(f,z) \circ q.\text{enq}(x_1, \ldots, x_k) \Rightarrow f(x_k, \ldots f(x_1, q.\text{fold}(f,z)) \ldots) \]
\[ q.\text{enq}(y_1, \ldots, y_n) \circ q.\text{enq}(x_1, \ldots, x_k) \Rightarrow q.\text{enq}(x_1, \ldots, x_k, y_1, \ldots, y_n). \]

3.4.5 Output operations

Expressions reading an output register are bypassed:

\[ o \circ (o \leftarrow x) \Rightarrow x. \]

Writes, however, cannot be composed because ATS semantics require all exported writes to appear in their original sequence.

3.4.6 Input operations

Since ATS rules cannot modify input registers, no composition rules are required.
Chapter 4

Cross-product Transformations

Issuing multiple instructions in one cycle while maintaining the semantics of single issue has become the norm in high-performance processor implementation. But the gain in performance comes at a significant cost in design effort: the added complexity of scheduling and hazard resolution render design and verification a time-consuming and bug-prone process.

This chapter describes an algorithm which transforms an ATS by adding "cross-product" rules. As we shall see, this technique derives an n-way superscalar microprocessor from its single-issue version.

4.1 Terminology

Hennessy and Patterson define a *superscalar* processor to be "a machine that executes multiple instructions per clock cycle with instruction issue controlled dynamically by hardware" [2]. This definition, while appropriate for microprocessors, excludes other systems that process several similar pieces of information at a time with sequential semantics: neither a memory management unit that processes multiple requests in one cycle nor a packet router that processes multiple packets in one cycle are "superscalar," even though both require hardware similar to that of superscalar processors.

Instead of abusing this definition, we shall use the term *cross product* to refer to systems generated by our algorithm.

4.2 Example: Multiple Loads

We have already seen a simple example of the cross-product transformation: *SuperFibonacci* in Section 3.1 is a cross-product ATS does the equivalent of issuing two instructions in one transition. In a more real-world example, we show how the cross-product algorithm derives a superscalar version of the four-stage pipeline from Appendix C.

The *executeLoad* rule removes a load instruction template from the execute buffer *xbuf* and adds the result to the writeback buffer *wbuf*:

\[
\text{executeLoad:}
\text{when } \text{xbuf.full}?(1) \land \text{wbuf.free}?(1) \land \text{xbuf.head}(1)_{1:0} = 3_2
\]
\[
\text{xbuf.deq}(1)
\]
\[
\text{wbuf.enq(xbuf.head}(1)_{6:2} + + \text{dmem}[\text{xbuf.head}(1)_{73:42}]).
\]

Using the composition algorithm from Chapter 3, we compose *executeLoad* with *executeLoad* to obtain a rule that executes two contiguous loads (for readability, some algebraic simplifications have been applied to the composite rule):
executeLoad and executeLoad:
when $\text{xbuf.full?(2)} \land \text{wbbuf.free?(2)} \land \text{xbuf.head(1)_{1:0}} = 3_2$
\begin{align*}
\land \text{xbuf.head(2)_{1:0}} &= 3_2 \\
\text{xbuf.deq}(2) \\
\text{wbbuf.enq}(\text{xbuf.head(1)_{6:2}} + \text{dmem}[\text{xbuf.head(1)_{73:42}}]) , \\
\text{xbuf.head(2)_{6:2}} + \text{dmem}[\text{xbuf.head(2)_{73:42}}]) .
\end{align*}

By adding executeLoad and executeLoad—and rules for fetching, decoding, executing, and committing other two-instruction sequences—to the original ATS, we obtain the 2-way superscalar 4-stage pipeline shown in Appendix E.

With sufficient chip area available, there is no reason to stop at a 2-way superscalar AX. We can compose the new executeLoad and executeLoad rule with executeLoad to obtain an execution rule for three contiguous loads:

executeLoad and executeLoad and executeLoad:
when $\text{xbuf.full?(3)} \land \text{wbbuf.free?(3)} \land \text{xbuf.head(1)_{1:0}} = 3_2$
\begin{align*}
\land \text{xbuf.head(2)_{1:0}} &= 3_2 \land \text{xbuf.head(3)_{1:0}} = 3_2 \\
\text{xbuf.deq}(3) \\
\text{wbbuf.enq}(\text{xbuf.head(1)_{6:2}} + \text{dmem}[\text{xbuf.head(1)_{73:42}}]) , \\
\text{xbuf.head(2)_{6:2}} + \text{dmem}[\text{xbuf.head(2)_{73:42}}]) , \\
\text{xbuf.head(3)_{6:2}} + \text{dmem}[\text{xbuf.head(3)_{73:42}}]) .
\end{align*}

Space considerations prevent us from including the full 3-way superscalar 4-stage AX pipeline in this thesis.

4.3 Cross Product Details

As we have seen, combining all possible rule pairs yields a two-way cross-product ATS. Formally, a set of rules $R = \{\theta_1, \ldots, \theta_n\}$, composed with itself, results in two-way cross-product rules $R^2$:

\[
R^2 = R \circ R = \{ \theta_i \circ \theta_j \mid 1 \leq i \leq n, 1 \leq j \leq n \}.
\]

Repeated composition with $R$ produces $k$-way cross-product rules $R^k$:

\[
R^k = R \circ R^{k-1} = R \circ \cdots \circ R.
\]

The full set of rules in a $k$-way cross-product ATS is then formed from all $R^i$ such that $i \leq k$:

\[
R_{ATS} = R \cup R^2 \cup \cdots \cup R^k .
\]

This ensures that cycles do not go to waste in cases where a cross-product rule does not apply but the original rule does (e.g., the next instruction is cached, but the one after is not).

Because cross-product rules can always fall back on the original rules, we do not require that the rule compositions be defined; if a composition is undefined, it is ignored.

4.4 Efficiency

A full cross product composes all possible rule pairs. While this approach results in a correct ATS, it is hardly efficient for most systems: many of the cross-product rules combine rules that are independent and hence co-scheduled anyway. While this does not change the semantics of the ATS, extra rules mean additional hardware in the synthesized circuit.

To avoid combining these rules, we need to examine how independent rules are scheduled. The ATS compiler co-schedules rules which it considers "conflict-free" [3]; intuitively, two rules may be co-scheduled if they do not modify the same state in different ways and an application of one does
not affect the applicability of the other. Most rules in different pipeline stages are good examples of conflict-free rules.

Because conflict-free rules are already co-scheduled, new rules should be limited to compositions with state conflicts. Rules $r_1 = (\pi_1, \delta_1)$ and $r_2 = (\pi_2, \delta_2)$ have a state conflict if there is a state $s$ such that

$$\delta_2(\delta_1(s)) \neq \delta_1(\delta_2(s)).$$

Ignoring all but state-conflicting rules in a cross-product transformation drastically cuts the number of rules. For deep pipelines, the cross-product ATS generated with state conflicts in mind is significantly smaller than the naively generated cross-product. Figure 4-1 illustrates the effectiveness of this optimization: up to 41% of rules are eliminated in multiple-issue derivations of a four-stage AX pipeline. Perhaps more importantly, the remaining cross-product rules combine rules from the same pipeline stages—exactly the rules a human designer would compose to obtain a superscalar processor.

Although detecting state conflicts precisely is a difficult proposition, a simple observation allows us to compute conflicts efficiently. In real-world ATSs, it turns out, rules are rarely written to cancel each other's effects. As a result, two rules that modify the same state term in any way are very likely to have a state conflict. This approximation was sufficient for all cross-product transformations included in this thesis.

### 4.5 Synthesis Issues

Even though the user does not specify a depth when declaring a queue, the synthesis tool must choose a depth to implement the queue in hardware. (In most cases, the maximum depth read in any rule is a good choice, since it generates the fewest number of logic gates while providing enough resources for all rules to fire.) In a k-way cross-product ATS, the minimum queue depth increases $k$ times.

Arrays, rendered in hardware as register files, exhibit a similar problem. The compiler infers the minimum required number of read and write ports in the register file from the rules, and, since additional ports rapidly increase cost and complexity, does not generate extra ports. A k-way cross-product transformation increases demands on read and write ports $k$-fold.

Because register files grow quadratically with the number of ports [8], arrays dominate chip area as $k$ increases. Fortunately, this is easily solved by adding a post-processing stage to remove cross-product rules that violate some resource constraints. A designer may, for example, prefer savings in chip area to simultaneously executing rare instruction sequences, and might limit the number of write ports on synthesized arrays.

Cross-product rules tend to increase the critical path and dilate the clock cycle in the synthesized circuit. While increasing the critical path somewhat is inevitable, cross-product rules which increase it beyond the designer's demands can be removed from the ATS.

<table>
<thead>
<tr>
<th>WIDTH</th>
<th>SIMPLE</th>
<th>OPTIMIZED</th>
<th>SAVINGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>98</td>
<td>74</td>
<td>24%</td>
</tr>
<tr>
<td>3</td>
<td>631</td>
<td>409</td>
<td>35%</td>
</tr>
<tr>
<td>4</td>
<td>4,112</td>
<td>2,442</td>
<td>41%</td>
</tr>
<tr>
<td>5</td>
<td>32,323</td>
<td>19,055</td>
<td>41%</td>
</tr>
</tbody>
</table>

Figure 4-1: Sizes of superscalar ATSs derived from the 4-stage AX
executeAdd and writeback:
when xbuf.full?(1) \land xbuf.head(1)_{1:0} = 2_2 \land xbuf.head(1)_{8:7} = 3_3
rf[xbuf.head(1)_{6:2}] \leftarrow xbuf.head(1)_{7:3} + xbuf.head(1)_{41:10}
xbuf.deq(1).

Similarly, we remove the execute buffer xbuf and compose the decodeLoadPC rule with the new executeAdd and writeback rule:

decodeLoadPC and executeAdd and writeback:
when ibuf.full?(1) \land ibuf.head(1)_{27:25} = 1_3
rf[ibuf.head(1)_{20:16}] \leftarrow ibuf.head(1)_{63:32}.

Finally, we add the fetch rule and remove the instruction buffer ibuf to complete the elision:

fetch and decodeLoadPC and executeAdd and writeback:
when imem[pc]_{27:25} = 1_3
pc \leftarrow pc + 1_{32}
rf[imem[pc]_{20:16}] \leftarrow pc.

As the final rule is identical (modulo the name) to the LoadPC rule in the reference AX, we conclude that the four-stage pipeline implements loadPC correctly.

For an example of collapsing complete ATSs, see Appendices F and G. Visually comparing the resulting ATSs shows that they correctly implement the reference AX from Appendix A.

5.2 Requirements
To be successfully elided, a queue must be empty in its initial state. Moreover, depending on its effect on the queue, every rule must belong to one of three categories:

1. **producers** add elements to the queue after ensuring there is sufficient space and may fold over the queue;
2. **consumers** remove elements from the queue after ensuring that the queue is sufficiently full and may read from or fold over the queue;
3. **dummies** do not read or modify the queue; dummies and consumers must be conflict-free.

An ATS with rules that do not fit any of these categories cannot be elided.

5.3 Elision Details
As we have seen, collapsing two pipeline stages is accomplished by fusing rules that add or remove elements on either side of an initially empty queue; the composites then replace the original rules. Because no remaining rules add or remove elements, the queue is always empty and can be safely removed.

5.3.1 Composition
To ensure completeness in eliding a queue q, every consumer rule must be composed with every producer rule. For this, we shall employ the composition algorithm developed in Chapter 3.

First, however, producer rule expressions that refer to q must be modified to reflect the empty queue. In particular,

\[
q.\text{free}(1) \Rightarrow 1_1
\]
\[
q.\text{fold}(f, z) \Rightarrow z.
\]

Replacing \textit{fold} with its zero in the composition algorithm simplifies the composition

\[
q.\text{fold}(f, z) \circ q.\text{enq}(x) \Rightarrow f(x, z),
\]
Chapter 5

Queue Elision

As processor pipelines become deeper and more complex, verification against the original instruction set architecture becomes much more difficult. Yet such verification in early stages of design is essential, since the cost of fixing design errors skyrockets as the design process advances.

Levitt and Olukotun [6] attempt to simplify verification by semi-automatically collapsing pipeline stages until a single stage is obtained and generating an inductive correctness proof. Hoe and Arvind [4] suggest that this process can be automated in TRS, where pipelines are typically implemented as queues.

In this chapter, we employ the rule composition algorithm in a fully automated elision technique that applies to any pipelined ATS. In spite of its limitations, this method successfully handles reasonable single-issue pipelines.

5.1 Example

To develop an intuition for elision, we examine how the four-stage AX pipeline (Appendix C) processes the loadPC instruction and remove the pipeline buffers to verify the implementation of loadPC.

The four-stage AX pipeline employs rules fetch, decodeLoadPC, executeAdd, and writeback to process each loadPC:

```plaintext
fetch:
    when ibuf.free?(1)
      pc ← pc + 1_{32}
      ibuf.enq(pc ++ imem[pc])

decodeLoadPC:
    when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 1_{3}
      ibuf.deq(1)
      xbuf.enq(ibuf.head(1)_{63:32} ++ 0_{32} ++ 3_{3} ++ ibuf.head(1)_{20:16} ++ 2_{2})

executeAdd:
    when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1)_{1:0} = 2_{2} ∧ xbuf.head(1)_{9:7} = 3_{3}
      xbuf.deq(1)
      wbuf.enq(xbuf.head(1)_{6:2} ++ xbuf.head(1)_{73:42} + xbuf.head(1)_{41:10})

writeback:
    when wbuf.full?(1)
      rf[wbuf.head(1)_{36:32}] ← wbuf.head(1)_{31:0}
      wbuf.deq(1).
```

We first combine the executeAdd and writeback rules by removing the writeback buffer wbuf. Under the assumption that wbuf contains no elements when executeAdd applies and one element—the result of applying executeAdd—when writeback applies, we eliminate wbuf and employ the composition algorithm to fuse the rules:
and enables two new compositions:

\[ q.\text{head}(n) \circ q.\text{enq}(x) \Rightarrow x \]
\[ q.\text{deq}(1) \circ q.\text{enq}(x) \Rightarrow \varepsilon. \]

The resulting composite rule contains no references to \( q \).

Unlike the cross-product transformation, elision uses the composite rules to replace the original rules. All compositions must therefore satisfy completeness: they must result in a composite rule or be impotent. If a composition is undefined, completeness is not preserved and elision must fail.

5.3.2 Elision

If all compositions succeed, the new ATS is constructed: producers and consumers are removed and replaced with composite rules. Because \( q \) is always empty and never accessed, it can be removed as well. The resulting ATS does not contain \( q \) and preserves the semantics of the original.

5.3.3 Simplification

While all of the composite rules are correct and preserve the original semantics, many of them may never fire: their predicates are always false. This is often the case in processor pipelines, where producer rules for one instruction type are followed by consumer rules for that instruction type. In the AX two-stage and four-stage pipelines, for example, a decoded ALU operation never triggers the load or store execute rules.

In our implementation, such rules are eliminated by a simplification stage; the result is a more compact ATS and a smaller circuit. During the process of verifying the four-stage AX pipeline, simplification removed 23 out of 30 rules, leaving precisely the rules that make up the AX reference implementation.
Chapter 6

Conclusion

6.1 Summary
We have described ATS, an intermediate representation of TRS, and shown how ATS rules may be automatically composed. We have also demonstrated how rule composition is used to form cross-product systems like multiple-issue microprocessors, and to elide pipeline buffers for verification. As a part of this thesis, these techniques have been implemented as source-to-source transformation stages in a TRS synthesis tool.

6.2 Future Work
We believe that further investigation is required to render the techniques presented here practical. Some research directions we consider important are outlined below.

More complex state elements
Modern out-of-order and speculative processors employ complex state elements for register renaming, instruction reordering, and branch prediction [2]. To tackle modern processors, ATS and composition techniques must be extended with such state elements.

Non-processor systems
While this thesis has focused on well-understood general purpose microprocessor examples, the techniques presented here are not specific to microprocessors. But implications on systems other than processors are not well understood, and further research is needed.

Elision in formal verification
Further investigation is required to apply the queue elision technique in formal verification, and to automate general techniques for TRS verification.

Complexity metrics
The number of rules in an ATS appears to be a poor metric for complexity, as many expressions are shared by several rules. This phenomenon is particularly striking in cross-product transformations. Better complexity metrics would help evaluate efficiency without the need for synthesis.
Appendix A

AX: Reference Implementation

state
   pc : register 32
   rf : array [5] 32
   imem : array [8] 32
   dmem : array [8] 32

rules
Add:
   when \text{imem}[pc][27:25] = 03
   \hspace{1em} pc \leftarrow pc + 1_{32}
   \hspace{1em} rf[imem[pc][20:16] \leftarrow rf[imem[pc][15:11]] + rf[imem[pc][10:6]]

LoadC:
   when \text{imem}[pc][27:25] = 23
   \hspace{1em} pc \leftarrow pc + 1_{32}
   \hspace{1em} rf[imem[pc][20:16] \leftarrow 0_{16} + imem[pc][15:0]

LoadPC:
   when \text{imem}[pc][27:25] = 13
   \hspace{1em} pc \leftarrow pc + 1_{32}
   \hspace{1em} rf[imem[pc][20:16] \leftarrow pc

Load:
   when \text{imem}[pc][27:25] = 33
   \hspace{1em} pc \leftarrow pc + 1_{32}
   \hspace{1em} rf[imem[pc][20:16] \leftarrow dmem[rf[imem[pc][15:11]]]

Store:
   when \text{imem}[pc][27:25] = 53
   \hspace{1em} pc \leftarrow pc + 1_{32}
   \hspace{1em} dmem[rf[imem[pc][15:11]] \leftarrow rf[imem[pc][10:6]]

JzTaken:
   when \text{imem}[pc][27:25] = 43 \land rf[imem[pc][15:11]] = 0_{32}
   \hspace{1em} pc \leftarrow rf[imem[pc][10:6]]

JzNotTaken:
   when \text{imem}[pc][27:25] = 43 \land \neg (rf[imem[pc][15:11] = 0_{32})
   \hspace{1em} pc \leftarrow pc + 1_{32}
Appendix B

AX: 2-stage Pipeline

state

- pc : register 32
- rf : array [5] 32
- imem : array [8] 32
- dmem : array [8] 32
- xbuf : fifo 74

rules

decodeALU:
when xbuf.free?(1) \& imem[pc]_{27:25} = 03
  \& xbuf.fold(\lambda x . \lambda y . (if imem[pc]_{15:11} = x_{6:2} \lor imem[pc]_{10:8} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)
  pc \leftarrow pc + 1_{32}
  xbuf.enq((rf[imem[pc]_{15:11}] \oplus rf[imem[pc]_{10:8}] \oplus imem[pc]_{24:22} \oplus imem[pc]_{20:16} \oplus 2_2)

decodeLoadC:
when xbuf.free?(1) \& imem[pc]_{27:25} = 23
  pc \leftarrow pc + 1_{32}
  xbuf.enq(pc + 0_{32} + 3_3 + imem[pc]_{20:16} + 2_1)

decodeLoadPC:
when xbuf.free?(1) \& imem[pc]_{27:25} = 13
  pc \leftarrow pc + 1_{32}
  xbuf.enq((pc + 0_{32} + 3_3 + imem[pc]_{20:16} + 2_1)

decodeLoad:
when xbuf.free?(1) \& imem[pc]_{27:25} = 33
  \& xbuf.fold(\lambda x . \lambda y . (if imem[pc]_{15:11} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)
  pc \leftarrow pc + 1_{32}
  xbuf.enq((rf[imem[pc]_{15:11}] + 0_{32} + 3_3 + imem[pc]_{20:16} + 2_3)

decodeStore:
when xbuf.free?(1) \& imem[pc]_{27:25} = 53
  \& xbuf.fold(\lambda x . \lambda y . (if imem[pc]_{15:11} = x_{6:2} \lor imem[pc]_{10:8} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)
  pc \leftarrow pc + 1_{32}
  xbuf.enq((rf[imem[pc]_{15:11}] + rf[imem[pc]_{10:8}] + 3_3 + 0_5 + 1_2)

decodeJzTaken:
when imem[pc]_{27:25} = 43 \& rf[imem[pc]_{15:11}] = 0_{32}
  pc \leftarrow rf[imem[pc]_{10:8}]

decodeJzNotTaken:
when imem[pc]_{27:25} = 43 \& \neg(rf[imem[pc]_{15:11}] = 0_{32})
  pc \leftarrow pc + 1_{32}

executeAdd:
when $xbuf.full(1)$ \land xbuf.head(1)_{1:0} = 2 \land xbuf.head(1)_{9:7} = 3$
\[
rf[xbuf.head(1)_{6:2}] \leftarrow xbuf.head(1)_{73:42} + xbuf.head(1)_{41:10}
\]
\[
xbuf.deq(1)
\]

executeLoad:
when $xbuf.full(1)$ \land xbuf.head(1)_{1:0} = 3
\[
rf[xbuf.head(1)_{6:2}] \leftarrow dmem[xbuf.head(1)_{73:42}]
\]
\[
xbuf.deq(1)
\]

executeStore:
when $xbuf.full(1)$ \land xbuf.head(1)_{1:0} = 1
\[
dmem[xbuf.head(1)_{73:42}] \leftarrow xbuf.head(1)_{41:10}
\]
\[
xbuf.deq(1)
\]
Appendix C

AX: 4-stage Pipeline

state
\[pc: \text{register } 32\]
\[rf: \text{array } [5] \ 32\]
\[imem: \text{array } [8] \ 32\]
\[dmem: \text{array } [8] \ 32\]
\[ibuf: \text{fifo } 64\]
\[xbuf: \text{fifo } 74\]
\[wbuf: \text{fifo } 37\]

rules
fetch:
when \(ibuf.\text{free}(1)\)
\[pc \leftarrow pc + 1_{32}\]
\(ibuf.\text{enq}(pc ++ \text{imem}[pc])\)

decodeALU:
when \(ibuf.\text{full}(1) \land xbuf.\text{free}(1) \land ibuf.\text{head}(1)_{27:25} = 0_{3}\)
\[\land xbuf.\text{fold}(\lambda x . \lambda y . (if \ ibuf.\text{head}(1)_{15:11} = x_{6:2} \lor ibuf.\text{head}(1)_{10:6} = x_{6:2} \ \text{then } 0_{1} \ \text{else } y_{1}), 1_{1})\]
\[\land wbuf.\text{fold}(\lambda x . \lambda y . (if \ ibuf.\text{head}(1)_{15:11} = x_{6:2} \lor ibuf.\text{head}(1)_{10:6} = x_{6:2} \ \text{then } 0_{1} \ \text{else } y_{1}), 1_{1})\]
\(ibuf.\text{deq}(1)\)
\(xbuf.\text{enq}(rf[ibuf.\text{head}(1)_{15:11}] + rf[ibuf.\text{head}(1)_{10:6}] + ibuf.\text{head}(1)_{24:22} + ibuf.\text{head}(1)_{20:16} + 2_{2})\)

decodeLoadC:
when \(ibuf.\text{full}(1) \land xbuf.\text{free}(1) \land ibuf.\text{head}(1)_{27:25} = 2_{3}\)
\(ibuf.\text{deq}(1)\)
\(xbuf.\text{enq}(0_{16} ++ ibuf.\text{head}(1)_{15:0} + 0_{32} + 3_{3} + ibuf.\text{head}(1)_{20:16} + 2_{2})\)

decodeLoadPC:
when \(ibuf.\text{full}(1) \land xbuf.\text{free}(1) \land ibuf.\text{head}(1)_{27:25} = 1_{3}\)
\(ibuf.\text{deq}(1)\)
\(xbuf.\text{enq}(ibuf.\text{head}(1)_{63:32} + 0_{32} + 3_{3} + ibuf.\text{head}(1)_{20:16} + 2_{2})\)

decodeLoad:
when \(ibuf.\text{full}(1) \land xbuf.\text{free}(1) \land ibuf.\text{head}(1)_{27:25} = 3_{3}\)
\[\land xbuf.\text{fold}(\lambda x . \lambda y . (if \ ibuf.\text{head}(1)_{15:11} = x_{6:2} \ \text{then } 0_{1} \ \text{else } y_{1}), 1_{1})\]
\[\land wbuf.\text{fold}(\lambda x . \lambda y . (if \ ibuf.\text{head}(1)_{15:11} = x_{6:2} \lor ibuf.\text{head}(1)_{10:6} = x_{6:2} \ \text{then } 0_{1} \ \text{else } y_{1}), 1_{1})\]
\(ibuf.\text{deq}(1)\)
\(xbuf.\text{enq}(rf[ibuf.\text{head}(1)_{15:11}] + 0_{32} + 3_{3} + ibuf.\text{head}(1)_{20:16} + 2_{2})\)

decodeStore:
when \(ibuf.\text{full}(1) \land xbuf.\text{free}(1) \land ibuf.\text{head}(1)_{27:25} = 5_{3}\)
\[\land xbuf.\text{fold}(\lambda x . \lambda y . (if \ ibuf.\text{head}(1)_{15:11} = x_{6:2} \lor ibuf.\text{head}(1)_{10:6} = x_{6:2} \ \text{then } 0_{1} \ \text{else } y_{1}), 1_{1})\]
\[\land wbuf.\text{fold}(\lambda x . \lambda y . (if \ ibuf.\text{head}(1)_{15:11} = x_{6:2} \lor ibuf.\text{head}(1)_{10:6} = x_{6:2} \ \text{then } 0_{1} \ \text{else } y_{1}), 1_{1})\]
\(ibuf.\text{deq}(1)\)
\$$xbuf\.enq(rf[ibuf\.head(1)_{15:11}] + rf[ibuf\.head(1)_{10:6}] + 3 + 0_5 + 1_2)\$$

**decodeJz:**
- when \(ibuf\.full?(1) \land xbuf\.free?(1) \land ibuf\.head(1)_{27:25} = 4_3\)
- \(xbuf\.fold(\lambda x : \lambda y . (if \ ibuf\.head(1)_{15:11} = x_{6:2} \lor ibuf\.head(1)_{10:6} = x_{6:2} \ then \ 0_1 \ else \ y), 1_1)\)
- \(wbuf\.fold(\lambda x : \lambda y . (if \ ibuf\.head(1)_{15:11} = x_{6:2} \lor ibuf\.head(1)_{10:6} = x_{6:2} \ then \ 0_1 \ else \ y), 1_1)\)
- \(ibuf\.deq(1)\)
- \(xbuf\.enq(rf[ibuf\.head(1)_{15:11}] + rf[ibuf\.head(1)_{10:6}] + 0_5 + 0_5 + 0_2)\)

**executeAdd:**
- when \(xbuf\.full?(1) \land wbuf\.free?(1) \land xbuf\.head(1)_{1:0} = 2_2 \land xbuf\.head(1)_{9:7} = 3_3\)
- \(xbuf\.deq(1)\)
- \(wbuf\.enq(xbuf\.head(1)_{6:2} + xbuf\.head(1)_{73:42} + xbuf\.head(1)_{41:10})\)

**executeLoad:**
- when \(xbuf\.full?(1) \land wbuf\.free?(1) \land xbuf\.head(1)_{1:0} = 3_2\)
- \(xbuf\.deq(1)\)
- \(wbuf\.enq(xbuf\.head(1)_{6:2} + dmem[xbuf\.head(1)_{73:42}])\)

**executeStore:**
- when \(xbuf\.full?(1) \land xbuf\.head(1)_{1:0} = 1_2\)
- \(dmem[xbuf\.head(1)_{73:42}] \leftarrow xbuf\.head(1)_{41:10}\)
- \(xbuf\.deq(1)\)

**executeJzTaken:**
- when \(xbuf\.full?(1) \land xbuf\.head(1)_{1:0} = 0_2 \land xbuf\.head(1)_{73:42} = 0_32\)
- \(pc \leftarrow xbuf\.head(1)_{41:10}\)
- \(ibuf\.deqall\)
- \(xbuf\.deqall\)

**executeJzNotTaken:**
- when \(xbuf\.full?(1) \land xbuf\.head(1)_{1:0} = 0_2 \land \neg(xbuf\.head(1)_{73:42} = 0_32)\)
- \(xbuf\.deq(1)\)

**writeback:**
- when \(wbuf\.full?(1)\)
- \(rf[wbuf\.head(1)_{38:32}] \leftarrow wbuf\.head(1)_{31:0}\)
- \(wbuf\.deq(1)\)
Appendix D

AX: 2-issue 2-stage Pipeline

state
- pc : register 32
- rf : array [5] 32
- imem : array [8] 32
- dmem : array [8] 32
- xbuf : fifo 74

rules

decodeALU:
- when xbuf.free?(1) ∧ imem[pc]_{27:25} = 03
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} \lor imem[pc]_{10:6} = x_{6:2} then 01 else y), 1_1)
  pc ← pc + 1_{32}
  xbuf.enq(rf[imem[pc]_{15:11}] ++ rf[imem[pc]_{10:6}] ++ imem[pc]_{24:22} ++ imem[pc]_{20:16} ++ 2_2)

decodeLoadC:
- when xbuf.free?(1) ∧ imem[pc]_{27:25} = 2_3
  pc ← pc + 1_{32}
  xbuf.enq(0_{16} ++ imem[pc]_{15:0} ++ 0_{32} ++ 3_3 ++ imem[pc]_{20:16} ++ 2_2)

decodeLoadPC:
- when xbuf.free?(1) ∧ imem[pc]_{27:25} = 1_3
  pc ← pc + 1_{32}
  xbuf.enq(pc ++ 0_{32} ++ 3_3 ++ imem[pc]_{20:16} ++ 2_2)

decodeLoad:
- when xbuf.free?(1) ∧ imem[pc]_{27:25} = 3_3
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} then 01 else y), 1_1)
  pc ← pc + 1_{32}
  xbuf.enq(rf[imem[pc]_{15:11}] ++ 0_{32} ++ 3_3 ++ imem[pc]_{20:16} ++ 3_2)

decodeStore:
- when xbuf.free?(1) ∧ imem[pc]_{27:25} = 5_3
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} \lor imem[pc]_{10:6} = x_{6:2} then 01 else y), 1_1)
  pc ← pc + 1_{32}
  xbuf.enq(rf[imem[pc]_{15:11}] ++ rf[imem[pc]_{10:6}] ++ 3_3 ++ 0_{5} ++ 1_2)

decodeJzTaken:
- when imem[pc]_{27:25} = 4_3 ∧ rf[imem[pc]_{15:11}] = 0_{32}
  pc ← rf[imem[pc]_{10:6}]

decodeJzNotTaken:
- when imem[pc]_{27:25} = 4_3 ∧ ¬(rf[imem[pc]_{15:11}] = 0_{32})
  pc ← pc + 1_{32}
executeAdd:
  when xbuf.full?(1) \&\& xbuf.head(1:0) = 2_2 \&\& xbuf.head(1:7) = 3_3
  rf[xbuf.head(1):6:2] \leftarrow xbuf.head(1:73:42 + xbuf.head(1:41:10)
  xbuf.deq(1)
executeLoad:
  when xbuf.full?(1) \&\& xbuf.head(1:0) = 3_2
  rf[xbuf.head(1):6:2] \leftarrow dmem[xbuf.head(1:73:42]
  xbuf.deq(1)
executeStore:
  when xbuf.full?(1) \&\& xbuf.head(1:0) = 1_2
  dmem[xbuf.head(1:73:42] \leftarrow xbuf.head(1:41:10)
  xbuf.deq(1)
decodeALU and decodeALU:
  when xbuf.free?(1) \&\& imem[pc]_{27:25} = 0_3
    \land xbuf.fold(\lambda x . \lambda y . (if imem[pc]_{15:11} = x_6:2 \lor imem[pc]_{10:6} = x_6:2 then 0_1 else y), 1_1)
    \land xbuf.free?(2) \&\& imem[pc] + 1_{32} = 2_{32}
  \land (if imem[pc] + 1_{32} = imem[pc]_{20:16} \lor imem[pc] + 1_{32} = 0_{16} then 0_1
  \lor else xbuf.fold(\lambda x . \lambda y . (if imem[pc] + 1_{32} = x_6:2 \lor imem[pc] + 1_{32} = x_6:2 then 0_1
  \lor else y), 1_1))
  pc \leftarrow pc + 2_{32}
xbuf.enq(rf[imem[pc]_{15:11}] + rf[imem[pc]_{10:6}] + imem[pc]_{24:22} + imem[pc]_{20:16} + 2_2,
  rf[imem[pc] + 1_{32}]_{15:11} + rf[imem[pc] + 1_{32}]_{10:6} + imem[pc] + 1_{32} = 2_{24:22}
  + imem[pc + 1_{32}]_{20:16} + 2_2)
decodeALU and decodeLoadC:
  when xbuf.free?(1) \&\& imem[pc]_{27:25} = 0_3
    \land xbuf.fold(\lambda x . \lambda y . (if imem[pc]_{15:11} = x_6:2 \lor imem[pc]_{10:6} = x_6:2 then 0_1 else y), 1_1)
    \land xbuf.free?(2) \&\& imem[pc] + 1_{32} = 2_{32}
    pc \leftarrow pc + 2_{32}
xbuf.enq(rf[imem[pc]_{15:11}] + rf[imem[pc]_{10:6}] + imem[pc]_{24:22} + imem[pc]_{20:16} + 2_2,
    0_{16} + imem[pc] + 1_{32} = 0_{32} + 3_3 + imem[pc] + 1_{32} = 2_{20:16} + 2_2)
decodeALU and decodeLoadPC:
  when xbuf.free?(1) \&\& imem[pc]_{27:25} = 0_3
    \land xbuf.fold(\lambda x . \lambda y . (if imem[pc]_{15:11} = x_6:2 \lor imem[pc]_{10:6} = x_6:2 then 0_1 else y), 1_1)
    \land xbuf.free?(2) \&\& imem[pc] + 1_{32} = 2_{32}
    pc \leftarrow pc + 2_{32}
xbuf.enq(rf[imem[pc]_{15:11}] + rf[imem[pc]_{10:6}] + imem[pc]_{24:22} + imem[pc]_{20:16} + 2_2,
    pc + 1_{32} + 0_{32} + 3_3 + imem[pc] + 1_{32} = 2_{20:16} + 2_2)
decodeALU and decodeLoad:
  when xbuf.free?(1) \&\& imem[pc]_{27:25} = 0_3
    \land xbuf.fold(\lambda x . \lambda y . (if imem[pc]_{15:11} = x_6:2 \lor imem[pc]_{10:6} = x_6:2 then 0_1 else y), 1_1)
    \land xbuf.free?(2) \&\& imem[pc] + 1_{32} = 2_{32} \land (if imem[pc] + 1_{32} = imem[pc]_{20:16}
    then 0_1 else xbuf.fold(\lambda x . \lambda y . (if imem[pc] + 1_{32} = x_6:2 then 0_1 else y), 1_1))
    pc \leftarrow pc + 2_{32}
xbuf.enq(rf[imem[pc]_{15:11}] + rf[imem[pc]_{10:6}] + imem[pc]_{24:22} + imem[pc]_{20:16} + 2_2,
    rf[imem[pc] + 1_{32}]_{15:11} + 0_{32} + 3_3 + imem[pc] + 1_{32} = 2_{20:16} + 3_2)
decodeLoadC and decodeLoadPC:
when $xbuf\text{.free}?(1) \land \text{imem}[pc]_{27:25} = 0_3$
\[ \land xbuf\text{.fold}(\lambda x . \lambda y . (if \ imem[pc]_{15:11} = x_{6:2} \lor \ imem[pc]_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1) \land (if \ imem[pc] + 132_{15:11} = \text{imem}[pc]_{20:16} \lor \ imem[pc] + 132_{10:6} = \text{imem}[pc]_{20:16} \text{ then } 0_1 \text{ else } xbuf\text{.fold}(\lambda x . \lambda y . (if \ imem[pc] + 132_{15:11} = x_{6:2} \lor \ imem[pc] + 132_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)) \]
\[ \begin{array}{l}
   pc \leftarrow pc + 2_{32} \\
   xbuf\text{.enq}(rf[imem[pc]_{15:11}] + rf[imem[pc]_{16:6}] + imem[pc]_{24:22} + imem[pc]_{20:16} + 2_2, rf[imem[pc] + 132_{15:11}] + rf[imem[pc] + 132_{10:6}] + 3_3 + 0_5 + 1_2) 
\end{array} \]

decodeLoadC and decodeLoadC:
when $xbuf\text{.free}?(1) \land \text{imem}[pc]_{27:25} = 0_3$
\[ \land xbuf\text{.fold}(\lambda x . \lambda y . (if \ imem[pc]_{15:11} = x_{6:2} \lor \ imem[pc]_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1) \land \text{imem}[pc + 132]_{27:25} = 4_3 \land rf[imem[pc + 132]_{15:11}] = 0_3 \land pc \leftarrow rf[imem[pc + 132]_{10:6}] \\
   xbuf\text{.enq}(rf[imem[pc]_{15:11}] + rf[imem[pc]_{10:6}] + imem[pc]_{24:22} + imem[pc]_{20:16} + 2_2) 
\]

decodeLoadC and decodeLoadC:
when $xbuf\text{.free}?(1) \land \text{imem}[pc]_{27:25} = 2_3 \land \text{imem}[pc + 132]_{27:25} = 0_3$
\[ \land (if \ imem[pc + 132]_{15:11} = \text{imem}[pc]_{20:16} \lor \ imem[pc + 132]_{10:6} = \text{imem}[pc]_{20:16} \text{ then } 0_1 \text{ else } xbuf\text{.fold}(\lambda x . \lambda y . (if \ imem[pc + 132]_{15:11} = x_{6:2} \lor \ imem[pc + 132]_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)) \]
\[ \begin{array}{l}
   pc \leftarrow pc + 2_{32} \\
   xbuf\text{.enq}(0_16 + imem[pc]_{15:6} + 0_32 + 3_3 + imem[pc]_{20:16} + 2_2, rf[imem[pc + 132]_{15:11}] + rf[imem[pc + 132]_{10:6}] + imem[pc + 132]_{24:22} + imem[pc + 132]_{20:16} + 2_2) 
\end{array} \]
decodeLoadC and decodeStore:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 23 \& \text{xbuf}.\text{free}\)?(2) \& \( \text{imem}[pc + 132]_{27:25} = 53 \)
\& (if \( \text{imem}[pc + 132]_{15:11} = \text{imem}[pc]_{20:16} \& \text{imem}[pc + 132]_{10:6} = \text{imem}[pc]_{20:16} \) then 01
else \( \text{xbuf}.\text{fold}(\lambda x . \lambda y . (if \( \text{imem}[pc + 132]_{15:11} = x_{6:2} \& \text{imem}[pc + 132]_{10:6} = x_{6:2} \) then 01
else \( y), 11)\))
\( pc \leftarrow pc + 232 \)
\( \text{xbuf}.\text{enq}(016 \& \& \text{imem}[pc]_{15:0} \& \& \text{imem}[pc]_{20:16} \& \& 23, \)
\( \text{rf}[\text{imem}[pc + 132]_{15:11} \& \& \text{imem}[pc + 132]_{10:6} \& \& 33 \& \& 05 \& \& 12) \)

decodeLoadC and decodeJzTaken:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 23 \& \text{imem}[pc + 132]_{27:25} = 43 \)
\& \( \text{rf}[\text{imem}[pc + 132]_{15:11} = 032 \)
\( pc \leftarrow \text{rf}[\text{imem}[pc + 132]_{10:6}] \)
\( \text{xbuf}.\text{enq}(016 \& \& \text{imem}[pc]_{15:0} \& \& \text{imem}[pc]_{20:16} \& \& 23) \)

decodeLoadC and decodeJzNotTaken:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 23 \& \text{imem}[pc + 132]_{27:25} = 43 \)
\& \( \neg(\text{rf}[\text{imem}[pc + 132]_{15:11} = 032) \)
\( pc \leftarrow pc + 232 \)
\( \text{xbuf}.\text{enq}(016 \& \& \text{imem}[pc]_{15:0} \& \& \text{imem}[pc]_{20:16} \& \& 23) \)

decodeLoadPC and decodeALU:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 13 \& \text{xbuf}.\text{free}\)?(2) \& \( \text{imem}[pc + 132]_{27:25} = 03 \)
\& (if \( \text{imem}[pc + 132]_{15:11} = \text{imem}[pc]_{20:16} \& \text{imem}[pc + 132]_{10:6} = \text{imem}[pc]_{20:16} \) then 01
else \( \text{xbuf}.\text{fold}(\lambda x . \lambda y . (if \( \text{imem}[pc + 132]_{15:11} = x_{6:2} \& \text{imem}[pc + 132]_{10:6} = x_{6:2} \) then 01
else \( y), 11)\))
\( pc \leftarrow pc + 232 \)
\( \text{xbuf}.\text{enq}(pc \& \& 032 \& \& 33 \& \& \text{imem}[pc]_{20:16} \& \& 23, \text{rf}[\text{imem}[pc + 132]_{15:11} \& \& \text{imem}[pc + 132]_{10:6} \& \& 33 \& \& 05 \& \& 12) \)

decodeLoadPC and decodeLoadC:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 13 \& \text{xbuf}.\text{free}\)?(2) \& \( \text{imem}[pc + 132]_{27:25} = 23 \)
\( pc \leftarrow pc + 232 \)
\( \text{xbuf}.\text{enq}(pc \& \& 032 \& \& 33 \& \& \text{imem}[pc]_{20:16} \& \& 23, \)
\( 0_{16} \& \& \text{imem}[pc + 132]_{15:0} \& \& 032 \& \& 33 \& \& \text{imem}[pc + 132]_{20:16} \& \& 23) \)

decodeLoadPC and decodeLoadPC:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 13 \& \text{xbuf}.\text{free}\)?(2) \& \( \text{imem}[pc + 132]_{27:25} = 13 \)
\( pc \leftarrow pc + 232 \)
\( \text{xbuf}.\text{enq}(pc \& \& 032 \& \& 33 \& \& \text{imem}[pc]_{20:16} \& \& 23, \)
\( pc + 132 \& \& 032 \& \& 33 \& \& \text{imem}[pc + 132]_{20:16} \& \& 23) \)

decodeLoadPC and decodeLoadLoad:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 13 \& \text{xbuf}.\text{free}\)?(2) \& \( \text{imem}[pc + 132]_{27:25} = 33 \)
\& (if \( \text{imem}[pc + 132]_{15:11} = \text{imem}[pc]_{20:16} \) then 01
else \( \text{xbuf}.\text{fold}(\lambda x . \lambda y . (if \( \text{imem}[pc + 132]_{15:11} = x_{6:2} \) then 01
else \( y), 11)\))
\( pc \leftarrow pc + 232 \)
\( \text{xbuf}.\text{enq}(pc \& \& 032 \& \& 33 \& \& \text{imem}[pc]_{20:16} \& \& 23, \text{rf}[\text{imem}[pc + 132]_{15:11} \& \& \text{imem}[pc + 132]_{10:6} \& \& 33 \& \& 05 \& \& 12) \)

decodeLoadPC and decodeStore:
when \( \text{xbuf}.\text{free}\)?(1) \& \( \text{imem}[pc]_{27:25} = 13 \& \text{xbuf}.\text{free}\)?(2) \& \( \text{imem}[pc + 132]_{27:25} = 53 \)
\& (if \( \text{imem}[pc + 132]_{15:11} = \text{imem}[pc]_{20:16} \& \text{imem}[pc + 132]_{10:6} = \text{imem}[pc]_{20:16} \) then 01
else \( \text{xbuf}.\text{fold}(\lambda x . \lambda y . (if \( \text{imem}[pc + 132]_{15:11} = x_{6:2} \) then 01
else \( y), 11)\))
\( pc \leftarrow pc + 232 \)
\( \text{xbuf}.\text{enq}(pc \& \& 032 \& \& 33 \& \& \text{imem}[pc]_{20:16} \& \& 23, \text{rf}[\text{imem}[pc + 132]_{15:11} \& \& \text{imem}[pc + 132]_{10:6} \& \& 33 \& \& 05 \& \& 12) \)

42
decodeLoadPC and decodeJzTaken:
when \(xbuf.free?(1) \land imem[pc]_{27:25} = 13 \land imem[pc + 132]_{27:25} = 43\)
\& rf[imem[pc + 132]_{15:11}] = 0_{32}
\& pc ← rf[imem[pc + 132]_{10:6}]
\& xbuf.enq(pc + 0_{32} + 3_{3} + imem[pc]_{20:16} + 2_{2})

decodeLoadPC and decodeJzNotTaken:
when \(xbuf.free?(1) \land imem[pc]_{27:25} = 13 \land imem[pc + 132]_{27:25} = 43\)
\& \neg(rf[imem[pc + 132]_{15:11}] = 0_{32})
\& pc ← pc + 2_{32}
\& xbuf.enq(pc + 0_{32} + 3_{3} + imem[pc]_{20:16} + 2_{2})

decodeLoad and decodeALU:
when \(xbuf.free?(1) \land imem[pc]_{27:25} = 33\)
\& xbuf.fold(Ax . Ay . (if imem[pc]_{15:11} = \text{X6:2} then 0_{1} else y), 1_{1}) \land xbuf.free?(2)
\& imem[pc + 132]_{27:25} = 0_{32}
\& (if imem[pc + 132]_{15:11} = imem[pc]_{20:16} \lor imem[pc + 132]_{10:6} = imem[pc]_{20:16} then 0_{1}
\& else xbuf.fold(\lambda x . Ay . (if imem[pc + 132]_{15:11} = \text{X6:2} \lor imem[pc + 132]_{10:6} = \text{X6:2} then 0_{1}
\& else y), 1_{1}))
\& pc ← pc + 2_{32}
\& xbuf.enq(rf[imem[pc]_{15:11}] + 0_{32} + 3_{3} + imem[pc]_{20:16} + 3_{2},
\& rf[imem[pc + 132]_{15:11}] + 0_{16} + imem[pc + 132]_{10:6} + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 2_{2})

decodeLoad and decodeLoadC:
when \(xbuf.free?(1) \land imem[pc]_{27:25} = 33\)
\& xbuf.fold(\lambda x . Ay . (if imem[pc]_{15:11} = \text{X6:2} then 0_{1} else y), 1_{1}) \land xbuf.free?(2)
\& imem[pc + 132]_{27:25} = 3_{3}
\& imem[pc + 132]_{15:11} = imem[pc]_{20:16} + 3_{2},
\& 0_{16} + imem[pc + 132]_{10:6} + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 2_{2})

decodeLoad and decodeLoadPC:
when \(xbuf.free?(1) \land imem[pc]_{27:25} = 33\)
\& xbuf.fold(\lambda x . Ay . (if imem[pc]_{15:11} = \text{X6:2} then 0_{1} else y), 1_{1}) \land xbuf.free?(2)
\& imem[pc + 132]_{27:25} = 3_{3}
\& pc ← pc + 2_{32}
\& xbuf.enq(rf[imem[pc]_{15:11}] + 0_{32} + 3_{3} + imem[pc]_{20:16} + 3_{2},
\& pc + 132 + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 2_{2})

decodeLoad and decodeLoad:
when \(xbuf.free?(1) \land imem[pc]_{27:25} = 33\)
\& xbuf.fold(\lambda x . Ay . (if imem[pc]_{15:11} = \text{X6:2} then 0_{1} else y), 1_{1}) \land xbuf.free?(2)
\& imem[pc + 132]_{27:25} = 3_{3}
\& (if imem[pc + 132]_{15:11} = imem[pc]_{20:16} \lor imem[pc + 132]_{10:6} = imem[pc]_{20:16} then 0_{1}
\& else xbuf.fold(\lambda x . Ay . (if imem[pc + 132]_{15:11} = \text{X6:2} \lor imem[pc + 132]_{10:6} = \text{X6:2} then 0_{1}
\& else y), 1_{1}))
\& pc ← pc + 2_{32}
\& xbuf.enq(rf[imem[pc]_{15:11}] + 0_{32} + 3_{3} + imem[pc]_{20:16} + 3_{2},
\& rf[imem[pc + 132]_{15:11}] + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 2_{2})

decodeLoad and decodeStore:
when \(xbuf.free?(1) \land imem[pc]_{27:25} = 33\)
\& xbuf.fold(\lambda x . Ay . (if imem[pc]_{15:11} = \text{X6:2} then 0_{1} else y), 1_{1}) \land xbuf.free?(2)
\& imem[pc + 132]_{27:25} = 3_{3}
\& (if imem[pc + 132]_{15:11} = imem[pc]_{20:16} \lor imem[pc + 132]_{10:6} = imem[pc]_{20:16} then 0_{1}
\& else xbuf.fold(\lambda x . Ay . (if imem[pc + 132]_{15:11} = \text{X6:2} \lor imem[pc + 132]_{10:6} = \text{X6:2} then 0_{1}
\& else y), 1_{1}))
\& pc ← pc + 2_{32}
\& xbuf.enq(rf[imem[pc]_{15:11}] + 0_{32} + 3_{3} + imem[pc]_{20:16} + 3_{2},
\& rf[imem[pc + 132]_{15:11}] + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 2_{2})

43
decodeLoad and decodeJzTaken:
when xbuf.free?(1) ∧ imem[pc]_{27:25} = 3
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} then 0_{1} else y), 1) ∧ imem[pc + 132]_{27:25} = 43
  ∧ rf [imem[pc + 132]_{15:11}] = 0_{32}
  pc ← rf [imem[pc + 132]_{10:6}]
xbuf.enq(rf [imem[pc]_{15:11}] + 0_{32} + 3_{3} + imem[pc]_{20:16} + 3_{2})

decodeLoad and decodeJzNotTaken:
when xbuf.free?(1) ∧ imem[pc]_{27:25} = 3
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} then 0_{1} else y), 1) ∧ imem[pc + 132]_{27:25} = 43
  ∧ ¬(rf [imem[pc + 132]_{15:11}] = 0_{32})
  pc ← pc + 2_{32}
xbuf.enq(rf [imem[pc]_{15:11}] + 0_{32} + 3_{3} + imem[pc]_{20:16} + 3_{2})

decodeStore and decodeALU:
when xbuf.free?(1) ∧ imem[pc]_{27:25} = 53
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} ∨ imem[pc]_{10:6} = x_{6:2} then 0_{1} else y), 1)
  ∧ ¬(xbuf.free?(2) ∧ imem[pc + 132]_{27:25} = 0_{3})
  ∧ (if imem[pc + 132]_{15:11} = 0_{5} ∨ imem[pc + 132]_{10:6} = 0_{5} then 0_{1}
else xbuf.fold(λx . λy . (if imem[pc + 132]_{15:11} = x_{6:2} ∨ imem[pc + 132]_{10:6} = x_{6:2} then 0_{1}
else y), 1))
  pc ← pc + 2_{32}
xbuf.enq(rf [imem[pc]_{15:11}] + rf [imem[pc]_{16:6}] + 3_{3} + 0_{5} + 1_{2}, rf [imem[pc + 132]_{15:11}]
  + rf [imem[pc + 132]_{16:6}] + imem[pc + 132]_{24:22} + imem[pc + 132]_{20:16} + 2_{2})

decodeStore and decodeLoadC:
when xbuf.free?(1) ∧ imem[pc]_{27:25} = 53
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} ∨ imem[pc]_{10:6} = x_{6:2} then 0_{1} else y), 1)
  ∧ xbuf.free?(2) ∧ imem[pc + 132]_{27:25} = 2_{3}
  pc ← pc + 2_{32}
xbuf.enq(rf [imem[pc]_{15:11}] + rf [imem[pc]_{16:6}] + 3_{3} + 0_{5} + 1_{2},
  0_{16} + imem[pc + 132]_{15:0} + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 2_{2})

decodeStore and decodeLoadPC:
when xbuf.free?(1) ∧ imem[pc]_{27:25} = 53
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} ∨ imem[pc]_{10:6} = x_{6:2} then 0_{1} else y), 1)
  ∧ xbuf.free?(2) ∧ imem[pc + 132]_{27:25} = 1_{3}
  pc ← pc + 2_{32}
xbuf.enq(rf [imem[pc]_{15:11}] + rf [imem[pc]_{16:6}] + 3_{3} + 0_{5} + 1_{2},
  pc + 1_{32} + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 2_{2})

decodeStore and decodeLoad:
when xbuf.free?(1) ∧ imem[pc]_{27:25} = 53
  ∧ xbuf.fold(λx . λy . (if imem[pc]_{15:11} = x_{6:2} ∨ imem[pc]_{10:6} = x_{6:2} then 0_{1} else y), 1)
  ∧ xbuf.free?(2) ∧ imem[pc + 132]_{27:25} = 3_{2} ∧ (if imem[pc + 132]_{15:11} = 0_{5} then 0_{1}
else xbuf.fold(λx . λy . (if imem[pc + 132]_{15:11} = x_{6:2} then 0_{1} else y), 1))
  pc ← pc + 2_{32}
xbuf.enq(rf [imem[pc]_{15:11}] + rf [imem[pc]_{10:6}] + 3_{3} + 0_{5} + 1_{2},
  rf [imem[pc + 132]_{15:11}] + 0_{32} + 3_{3} + imem[pc + 132]_{20:16} + 3_{2})
decodeStore and decodeStore:
when xbuf.free?(1) ∧ imem[pc][27:25] = 5₃
∧ xbuf.free?(2) ∧ imem[pc + 1₃₂][27:25] = 5₃
∧ (if imem[pc + 1₃₂][15:11] = 0₅ ∨ imem[pc + 1₃₂][10:6] = 0₅ then 0₁
else y), 1₁))
   pc ← pc + 2₃²
   xbuf.enq(rf[imem[pc][15:11]] ++ rf[imem[pc][10:6]] ++ 3₃ ++ 0₅ ++ 1₂,
    rf[imem[pc + 1₃₂][15:11]] ++ rf[imem[pc + 1₃₂][10:6]] ++ 3₃ ++ 0₅ ++ 1₂)

decodeStore and decodeJzTaken:
when xbuf.free?(1) ∧ imem[pc][27:25] = 5₃
∧ imem[pc + 1₃₂][27:25] = 4₃ ∧ rf[imem[pc + 1₃₂][15:11]] = 0₃₂
   pc ← rf[imem[pc + 1₃₂][10:6]]
   xbuf.enq(rf[imem[pc][15:11]] ++ rf[imem[pc][10:6]] ++ 3₃ ++ 0₅ ++ 1₂)

decodeStore and decodeJzNot Taken:
when xbuf.free?(1) ∧ imem[pc][27:25] = 5₃
∧ imem[pc + 1₃₂][27:25] = 4₃ ∧ ¬(rf[imem[pc + 1₃₂][15:11]] = 0₃₂)
   pc ← pc + 2₃²
   xbuf.enq(rf[imem[pc][15:11]] ++ rf[imem[pc][10:6]] ++ 3₃ ++ 0₅ ++ 1₂)

decodeJzTaken and decodeALU:
when imem[pc][27:25] = 4₃ ∧ rf[imem[pc][13:11]] = 0₃₂ ∧ xbuf.free?(1)
   pc ← rf[imem[pc][10:6] + 1₃²

decodeJzTaken and decodeLoadC:
when imem[pc][27:25] = 4₃ ∧ rf[imem[pc][15:11]] = 0₃₂ ∧ xbuf.free?(1)
   pc ← rf[imem[pc][10:6] + 1₃²
    ++ 2₁)

decodeJzTaken and decodeLoadPC:
when imem[pc][27:25] = 4₃ ∧ rf[imem[pc][15:11]] = 0₃₂ ∧ xbuf.free?(1)
   pc ← rf[imem[pc][10:6] + 1₃²
   xbuf.enq(rf[imem[pc][10:6]] ++ 0₃₂ ++ 3₃ ++ imem[rf[imem[pc][10:6]][20:16] ++ 2₁)

decodeJzTaken and decodeLoad:
when imem[pc][27:25] = 4₃ ∧ rf[imem[pc][15:11]] = 0₃₂ ∧ xbuf.free?(1)
∧ xbuf.fold(λx . A y : (if imem[rf[imem[pc][10:6]][15:11] = x6:2 then 0₁ else y), 1₁)
   pc ← rf[imem[pc][10:6] + 1₃²
decodeJzTaken and decodeStore:
   ∧ imem[rf[imem[pc]10:6]]27:25 = 53
   then 0₁ else y₁), 1₁)
   ++ 1₂)
decodeJzTaken and decodeJzTaken:
decodeJzTaken and decodeJzNot Taken:
decodeJzNotTaken and decodeALU:
   ∧ xbuf.fold(λx.λy. (if imem[pc + 132]15:11 = x₆:₂ ∨ imem[pc + 132]10:6 = x₆:₂ then 0₁
   else y₁), 1₁)
   pc ← pc + 232
   ++ imem[pc + 132]20:16 ++ 2₂)
decodeJzNotTaken and decodeLoadC:
   pc ← pc + 232
decodeJzNotTaken and decodeLoadPC:
   pc ← pc + 232
   xbuf.enq(pc + 132 ++ 0₃₂ ++ 3₃ ++ imem[pc + 132]₁₉:₁₆ ++ 2₂)
decodeJzNotTaken and decodeLoad:
   ∧ xbuf.fold(λx.λy. (if imem[pc + 132]15:11 = x₆:₂ then 0₁ else y₁), 1₁)
   pc ← pc + 232
decodeJzNotTaken and decodeLoadStore:
   ∧ xbuf.fold(λx.λy. (if imem[pc + 132]15:11 = x₆:₂ ∨ imem[pc + 132]10:6 = x₆:₂ then 0₁
   else y₁), 1₁)
   pc ← pc + 232
decodeJzNotTaken and decodeJzTaken:
   pc ← rf[imem[pc + 132]10:6]
decodeJzNotTaken and decodeJzNot Taken:
   ∧ ¬(rf[imem[pc + 132]15:11] = 032)
   pc ← pc + 232
executeAdd and executeAdd (1):
when xbuf.full?(1) \land xbuf.head(1,0) = 2 \land xbuf.head(1,9,7) = 3 \land xbuf.full?(2) \land xbuf.head(2,1,0) = 2 \land xbuf.head(2,9,7) = 3 \land \neg(xbuf.head(1,6,2) = xbuf.head(2,6,2))
rf [xbuf.head(1,6,2)] \leftarrow xbuf.head(2,73,42) + xbuf.head(2,41,10)
\text{executeAdd and executeAdd (2)}:
when xbuf.full?(1) \land xbuf.head(1,0) = 2 \land xbuf.head(1,9,7) = 3 \land xbuf.full?(2)
\land xbuf.head(2,1,0) = 2 \land xbuf.head(2,9,7) = 3 \land \neg(xbuf.head(1,6,2) = xbuf.head(2,6,2))
rf [xbuf.head(1,6,2)] \leftarrow xbuf.head(1,73,42) + xbuf.head(1,41,10)
\text{executeAdd and executeLoad (1)}:
when xbuf.full?(1) \land xbuf.head(1,0) = 2 \land xbuf.head(1,9,7) = 3 \land xbuf.full?(2)
\land xbuf.head(2,1,0) = 32 \land xbuf.head(1,6,2) = xbuf.head(2,6,2)
rf [xbuf.head(1,6,2)] \leftarrow dmem[xbuf.head(2,73,42)]
xbuf.deq(2)
\text{executeAdd and executeLoad (2)}:
when xbuf.full?(1) \land xbuf.head(1,0) = 2 \land xbuf.head(1,9,7) = 3 \land xbuf.full?(2)
\land xbuf.head(2,1,0) = 32 \land xbuf.head(1,6,2) = xbuf.head(2,6,2)
rf [xbuf.head(1,6,2)] \leftarrow xbuf.head(1,73,42) + xbuf.head(1,41,10)
\text{executeAdd and executeStore}:
when xbuf.full?(1) \land xbuf.head(1,0) = 2 \land xbuf.head(1,9,7) = 3 \land xbuf.full?(2)
\land xbuf.head(2,1,0) = 12
rf [xbuf.head(1,6,2)] \leftarrow xbuf.head(1,73,42) + xbuf.head(1,41,10)
dmem[xbuf.head(2,73,42)] \leftarrow xbuf.head(2,41,10)
xbuf.deq(2)
\text{executeLoad and executeAdd (1)}:
when xbuf.full?(1) \land xbuf.head(1,0) = 3 \land xbuf.full?(2) \land xbuf.head(2,1,0) = 2
\land xbuf.head(2,9,7) = 3 \land xbuf.head(1,6,2) = xbuf.head(2,6,2)
rf [xbuf.head(1,6,2)] \leftarrow xbuf.head(2,73,42) + xbuf.head(2,41,10)
xbuf.deq(2)
\text{executeLoad and executeAdd (2)}:
when xbuf.full?(1) \land xbuf.head(1,0) = 3 \land xbuf.full?(2) \land xbuf.head(2,1,0) = 2
\land xbuf.head(2,9,7) = 3 \land \neg(xbuf.head(1,6,2) = xbuf.head(2,6,2))
rf [xbuf.head(1,6,2)] \leftarrow dmem[xbuf.head(1,73,42)]
r[xbuf.head(2,6,2)] \leftarrow xbuf.head(2,73,42) + xbuf.head(2,41,10)
xbuf.deq(2)
\text{executeLoad and executeLoad (1)}:
when xbuf.full?(1) \land xbuf.head(1,0) = 3 \land xbuf.full?(2) \land xbuf.head(2,1,0) = 3
\land xbuf.head(2,9,7) = 3 \land xbuf.head(1,6,2) = xbuf.head(2,6,2)
rf [xbuf.head(1,6,2)] \leftarrow dmem[xbuf.head(2,73,42)]
xbuf.deq(2)
\text{executeLoad and executeLoad (2)}:
when xbuf.full?(1) \land xbuf.head(1,0) = 3 \land xbuf.full?(2) \land xbuf.head(2,1,0) = 3
\land \neg(xbuf.head(1,6,2) = xbuf.head(2,6,2))
rf [xbuf.head(1,6,2)] \leftarrow dmem[xbuf.head(1,73,42)]
r[xbuf.head(2,6,2)] \leftarrow dmem[xbuf.head(2,73,42)]
xbuf.deq(2)
executeLoad and executeStore:
when \( xbuf . full?(1) \land xbuf . head(1)_{1:0} = 3_2 \land xbuf . full?(2) \land xbuf . head(2)_{1:0} = 1_2 \)
\[ rf[xbuf . head(1)_{6:2}] \leftarrow \text{dmem}[xbuf . head(1)_{73:42}] \]
\[ \text{dmem}[xbuf . head(2)_{73:42}] \leftarrow xbuf . head(2)_{41:10} \]
\[ xbuf . deq(2) \]

executeStore and executeAdd:
when \( xbuf . full?(1) \land xbuf . head(1)_{1:0} = 1_2 \land xbuf . full?(2) \land xbuf . head(2)_{1:0} = 2_2 \)
\[ rf[xbuf . head(2)_{9:7}] = 3_1 \]
\[ rf[xbuf . head(2)_{6:2}] \leftarrow xbuf . head(2)_{73:42} + xbuf . head(2)_{41:10} \]
\[ \text{dmem}[xbuf . head(1)_{73:42}] \leftarrow xbuf . head(1)_{41:10} \]
\[ xbuf . deq(2) \]

executeStore and executeLoad (1):
when \( xbuf . full?(1) \land xbuf . head(1)_{1:0} = 1_2 \land xbuf . full?(2) \land xbuf . head(2)_{1:0} = 3_2 \)
\[ rf[xbuf . head(2)_{6:2}] \leftarrow \text{dmem}[xbuf . head(2)_{73:42}] \]
\[ \text{dmem}[xbuf . head(1)_{73:42}] \leftarrow xbuf . head(1)_{41:10} \]
\[ xbuf . deq(2) \]

executeStore and executeLoad (2):
when \( xbuf . full?(1) \land xbuf . head(1)_{1:0} = 1_2 \land xbuf . full?(2) \land xbuf . head(2)_{1:0} = 3_2 \)
\[ rf[xbuf . head(2)_{6:2}] \leftarrow xbuf . head(1)_{41:10} \]
\[ \text{dmem}[xbuf . head(1)_{73:42}] \leftarrow xbuf . head(1)_{41:10} \]
\[ xbuf . deq(2) \]

executeStore and executeStore (1):
when \( xbuf . full?(1) \land xbuf . head(1)_{1:0} = 1_2 \land xbuf . full?(2) \land xbuf . head(2)_{1:0} = 1_2 \)
\[ rf[xbuf . head(2)_{6:2}] \leftarrow xbuf . head(1)_{73:42} \]
\[ \text{dmem}[xbuf . head(1)_{73:42}] \leftarrow xbuf . head(2)_{41:10} \]
\[ xbuf . deq(2) \]

executeStore and executeStore (2):
when \( xbuf . full?(1) \land xbuf . head(1)_{1:0} = 1_2 \land xbuf . full?(2) \land xbuf . head(2)_{1:0} = 1_2 \)
\[ rf[xbuf . head(2)_{6:2}] \leftarrow \text{dmem}[xbuf . head(1)_{73:42}] \]
\[ \text{dmem}[xbuf . head(2)_{73:42}] \leftarrow xbuf . head(2)_{41:10} \]
\[ xbuf . deq(2) \]
Appendix E

AX: 2-issue 4-stage Pipeline

state

\[ pc : \text{register} \ 32 \]
\[ rf : \text{array} \ [5] \ 32 \]
\[ imem : \text{array} \ [8] \ 32 \]
\[ dmem : \text{array} \ [8] \ 32 \]
\[ ibuf : \text{fifo} \ 64 \]
\[ xbuf : \text{fifo} \ 74 \]
\[ wbuf : \text{fifo} \ 37 \]

rules

fetch:
when \( \text{ibuf}.\text{free?}(1) \)
\[ pc \leftarrow pc + 1_{32} \]
\[ \text{ibuf}.\text{enq}(pc + imem[pc]) \]

decodeALU:
when \( \text{ibuf}.\text{full?}(1) \land \text{xbuf}.\text{free?}(1) \land \text{ibuf}.\text{head}(1)_{27:25} = 0_3 \)
\[ \land \text{xbuf}.\text{fold}(\lambda x . \lambda y . (\text{if ibuf}.\text{head}(1)_{15:11} = x_{6:2} \lor \text{ibuf}.\text{head}(1)_{10:6} = x_{6:2} \text{then } 0_1 \text{else } y), 1_1) \]
\[ \land \text{xbuf}.\text{fold}(\lambda x . \lambda y . (\text{if ibuf}.\text{head}(1)_{15:11} = x_{6:2} \lor \text{ibuf}.\text{head}(1)_{10:6} = x_{6:2} \text{then } 0_1 \text{else } y), 1_1) \]
\[ \text{ibuf}.\text{deq}(1) \]
\[ \text{xbuf}.\text{enq}(\text{rf}[\text{ibuf}.\text{head}(1)_{15:11}] + \text{rf}[\text{ibuf}.\text{head}(1)_{10:6}] + \text{ibuf}.\text{head}(1)_{24:22} \]
\[ + \text{ibuf}.\text{head}(1)_{20:16} + 2_2) \]

decodeLoadC:
when \( \text{ibuf}.\text{full?}(1) \land \text{xbuf}.\text{free?}(1) \land \text{ibuf}.\text{head}(1)_{27:25} = 2_3 \)
\[ \text{ibuf}.\text{deq}(1) \]
\[ \text{xbuf}.\text{enq}(0_{16} + \text{ibuf}.\text{head}(1)_{15:0} + 0_{32} + 3_3 + \text{ibuf}.\text{head}(1)_{20:16} + 2_2) \]

decodeLoadPC:
when \( \text{ibuf}.\text{full?}(1) \land \text{xbuf}.\text{free?}(1) \land \text{ibuf}.\text{head}(1)_{27:25} = 1_3 \)
\[ \text{ibuf}.\text{deq}(1) \]
\[ \text{xbuf}.\text{enq}(\text{ibuf}.\text{head}(1)_{63:32} + 0_{32} + 3_3 + \text{ibuf}.\text{head}(1)_{20:16} + 2_2) \]

decodeLoad:
when \( \text{ibuf}.\text{full?}(1) \land \text{xbuf}.\text{free?}(1) \land \text{ibuf}.\text{head}(1)_{27:25} = 3_3 \)
\[ \land \text{xbuf}.\text{fold}(\lambda x . \lambda y . (\text{if ibuf}.\text{head}(1)_{15:11} = x_{6:2} \text{then } 0_1 \text{else } y), 1_1) \]
\[ \land \text{xbuf}.\text{fold}(\lambda x . \lambda y . (\text{if ibuf}.\text{head}(1)_{15:11} = x_{6:2} \text{then } 0_1 \text{else } y), 1_1) \]
\[ \text{ibuf}.\text{deq}(1) \]
\[ \text{xbuf}.\text{enq}(\text{rf}[\text{ibuf}.\text{head}(1)_{15:11}] + 0_{32} + 3_3 + \text{ibuf}.\text{head}(1)_{20:16} + 3_2) \]
decodeStore:
  when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1,27:25) = 53
    ∧ xbuf.fold(λx . λy . (if ibuf.head(1,15:11) = x6:2 ∨ ibuf.head(1,10:6) = x6:2 then 01 else y), I1)
    ∧ wbuf.fold(λx . λy . (if ibuf.head(1,15:11) = x6:2 ∨ ibuf.head(1,10:6) = x6:2 then 01 else y), I1)
    ibuf.deq(1)
    xbuf.enq(rf[ibuf.head(1,15:11)] + rf[ibuf.head(1,10:6)] + 33 ± 05 4+ 02)
  xbuf.free?(1)
  ibuf.head(1,27:25) = 43
  xbuf.fold(λx . λy . (if ibuf.head(1,15:11) = x6:2 ∨ ibuf.head(1,10:6) = x6:2 then 01 else y), I1)
  when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1,27:25) = 43
    xbuf.fold(λx . λy . (if ibuf.head(1,15:11) = x6:2 ∨ ibuf.head(1,10:6) = x6:2 then 01 else y), I1)
    ibuf.deq(1)
    xbuf.enq(rf[ibuf.head(1,15:11)] + rf[ibuf.head(1,10:6)] + 03 ++ 05 ++ 12)
executeAdd:
  when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1,1,0) = 22 ∧ xbuf.head(1,9:7) = 33
    xbuf.deq(1)
    wbuf.enq(xbuf.head(1,6:2) ++ xbuf.head(1,73:42) + xbuf.head(1,41:10))
executeLoad:
  when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1,1,0) = 32
    xbuf.deq(1)
    wbuf.enq(xbuf.head(1,6:2) ++ dmem[xbuf.head(1,73:42)])
executeStore:
  when xbuf.full?(1) ∧ xbuf.head(1,1,0) = 12
    dmem[xbuf.head(1,73:42)] ← xbuf.head(1,41:10)
    xbuf.deq(1)
executeJzTaken:
  when xbuf.full?(1) ∧ xbuf.head(1,1,0) = 02 ∧ xbuf.head(1,73:42) = 032
    pc ← xbuf.head(1,41:10)
    ibuf.deqall
    xbuf.deqall
executeJzNotTaken:
  when xbuf.full?(1) ∧ xbuf.head(1,1,0) = 02 ∧ ¬(xbuf.head(1,73:42) = 032)
    xbuf.deq(1)
writeback:
  when wbuf.full?(1)
    rf[wbuf.head(1,36:32)] ← wbuf.head(1,31:0)
    wbuf.deq(1)
fetch and fetch:
  when ibuf.free?(1) ∧ ibuf.free?(2)
    pc ← pc + 232
    ibuf.enq(pc ++ imem[pc], pc + 132 ++ imem[pc + 132])
decodeALU and decodeALU:
when ibuf.full?(1) \&\& zbuf.free?(1) \&\& ibuf.head(1)_{27:25} = 0_{3}
\&\& zbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& wbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& ibuf.full?(2) \&\& zbuf.free?(2) \&\& ibuf.head(2)_{27:25} = 0_{3}
\&\& (if ibuf.head(2)_{15:11} = ibuf.head(1)_{20:16} \vee ibuf.head(2)_{10:6} = ibuf.head(1)_{20:16} then 0_1
else xbuf.fold(\lambda x. \lambda y. (if ibuf.head(2)_{15:11} = x_{6:2} \vee ibuf.head(2)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& wbuf.fold(\lambda x. \lambda y. (if ibuf.head(2)_{15:11} = x_{6:2} \vee ibuf.head(2)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
ibuf.deq(2)

xbuf.enq(rf[ibuf.head(1)_{15:11}] ++ rf[ibuf.head(1)_{10:6}] ++ ibuf.head(1)_{24:22}
++ ibuf.head(2)_{20:16} ++ 2_2, rf[ibuf.head(2)_{15:11}] ++ rf[ibuf.head(2)_{10:6}]
++ ibuf.head(2)_{24:22} ++ ibuf.head(2)_{20:16} ++ 2_2)

decodeALU and decodeLoadC:
when ibuf.full?(1) \&\& zbuf.free?(1) \&\& ibuf.head(1)_{27:25} = 0_{3}
\&\& zbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& wbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& ibuf.full?(2) \&\& zbuf.free?(2) \&\& ibuf.head(2)_{27:25} = 0_{3}
ibuf.deq(2)

xbuf.enq(rf[ibuf.head(1)_{15:11}] ++ rf[ibuf.head(1)_{10:6}] ++ ibuf.head(1)_{24:22}
++ ibuf.head(1)_{20:16} ++ 2_2,
0_{16} ++ ibuf.head(2)_{15:10} ++ 0_{32} ++ 3_3 ++ ibuf.head(2)_{20:16} ++ 2_2)

decodeALU and decodeLoadPC:
when ibuf.full?(1) \&\& zbuf.free?(1) \&\& ibuf.head(1)_{27:25} = 0_{3}
\&\& zbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& wbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& ibuf.full?(2) \&\& zbuf.free?(2) \&\& ibuf.head(2)_{27:25} = 0_{3}
ibuf.deq(2)

xbuf.enq(rf[ibuf.head(1)_{15:11}] ++ rf[ibuf.head(1)_{10:6}] ++ ibuf.head(1)_{24:22}
++ ibuf.head(2)_{20:16} ++ 2_2,
ibuf.head(2)_{63:32} ++ 0_{32} ++ 3_3 ++ ibuf.head(2)_{20:16} ++ 2_2)

decodeALU and decodeLoad:
when ibuf.full?(1) \&\& zbuf.free?(1) \&\& ibuf.head(1)_{27:25} = 0_{3}
\&\& zbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& wbuf.fold(\lambda x. \lambda y. (if ibuf.head(1)_{15:11} = x_{6:2} \vee ibuf.head(1)_{10:6} = x_{6:2} then 0_1 else y), 1_1)
\&\& ibuf.full?(2) \&\& zbuf.free?(2) \&\& ibuf.head(2)_{27:25} = 0_{3}
\&\& (if ibuf.head(2)_{15:11} = ibuf.head(1)_{20:16} then 0_1
else xbuf.fold(\lambda x. \lambda y. (if ibuf.head(2)_{15:11} = x_{6:2} then 0_1 else y), 1_1)
\&\& wbuf.fold(\lambda x. \lambda y. (if ibuf.head(2)_{15:11} = x_{6:2} then 0_1 else y), 1_1)
ibuf.deq(2)

xbuf.enq(rf[ibuf.head(1)_{15:11}] ++ rf[ibuf.head(1)_{10:6}] ++ ibuf.head(1)_{24:22}
++ ibuf.head(1)_{20:16} ++ 2_2,
rf[ibuf.head(2)_{15:11}] ++ 0_{32} ++ 3_3 ++ ibuf.head(2)_{20:16} ++ 2_2)
decodeALU and decodeStore:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)\_{27:25} = 03
∧ xbuf.fold(\(x, y\). (ibuf.head(1)\_{15:11} = x_6:2 ∨ ibuf.head(1)\_{10:6} = x_6:2 then 0_1 else y), 1_1)
∧ xbuf.fold(\(x, y\). (ibuf.head(1)\_{15:11} = x_6:2 ∨ ibuf.head(1)\_{10:6} = x_6:2 then 0_1 else y), 1_1)
∧ ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2)\_{27:25} = 53
∧ (if ibuf.head(2)\_{15:11} = ibuf.head(1)\_{20:16} ∨ ibuf.head(2)\_{10:6} = ibuf.head(1)\_{20:16} then 0_1
else xbuf.fold(\(x, y\). (if ibuf.head(2)\_{15:11} = x_6:2 ∨ ibuf.head(2)\_{10:6} = x_6:2 then 0_1 else y), 1_1))
∧ xbuf.fold(\(x, y\). (if ibuf.head(2)\_{15:11} = x_6:2 ∨ ibuf.head(2)\_{10:6} = x_6:2 then 0_1 else y), 1_1)
ibuf.deq(2)
xbuf.enq(\(rf[ibuf.head(1)\_{15:11}] + rf[ibuf.head(1)\_{10:6}] + ibuf.head(1)\_{24:22} + ibuf.head(1)\_{20:16} + 2_2, \) rf[ibuf.head(2)\_{15:11}] + rf[ibuf.head(2)\_{10:6}] + 3_3 + 0_5 ++ 1_2)

decodeALU and decodeJs:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)\_{27:25} = 03
∧ xbuf.fold(\(x, y\). (ibuf.head(1)\_{15:11} = x_6:2 ∨ ibuf.head(1)\_{10:6} = x_6:2 then 0_1 else y), 1_1)
∧ xbuf.fold(\(x, y\). (if ibuf.head(2)\_{15:11} = x_6:2 ∨ ibuf.head(2)\_{10:6} = x_6:2 then 0_1 else y), 1_1)
∧ ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2)\_{27:25} = 43
∧ (if ibuf.head(2)\_{15:11} = ibuf.head(1)\_{20:16} ∨ ibuf.head(2)\_{10:6} = ibuf.head(1)\_{20:16} then 0_1
else xbuf.fold(\(x, y\). (if ibuf.head(2)\_{15:11} = x_6:2 ∨ ibuf.head(2)\_{10:6} = x_6:2 then 0_1 else y), 1_1))
∧ xbuf.fold(\(x, y\). (if ibuf.head(2)\_{15:11} = x_6:2 ∨ ibuf.head(2)\_{10:6} = x_6:2 then 0_1 else y), 1_1)
ibuf.deq(2)
xbuf.enq(\(rf[ibuf.head(1)\_{15:11}] + rf[ibuf.head(1)\_{10:6}] + ibuf.head(1)\_{24:22} + ibuf.head(1)\_{20:16} + 2_2, \) rf[ibuf.head(2)\_{15:11}] + rf[ibuf.head(2)\_{10:6}] + 0_3 + 0_5 ++ 0_2)

decodeLoadC and decodeALU:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)\_{27:25} = 23 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)\_{27:25} = 03
∧ (if ibuf.head(2)\_{15:11} = ibuf.head(1)\_{20:16} ∨ ibuf.head(2)\_{10:6} = ibuf.head(1)\_{20:16} then 0_1
else xbuf.fold(\(x, y\). (if ibuf.head(2)\_{15:11} = x_6:2 ∨ ibuf.head(2)\_{10:6} = x_6:2 then 0_1 else y), 1_1))
∧ xbuf.fold(\(x, y\). (if ibuf.head(2)\_{15:11} = x_6:2 ∨ ibuf.head(2)\_{10:6} = x_6:2 then 0_1 else y), 1_1)
ibuf.deq(2)
xbuf.enq(0_16 + ibuf.head(1)\_{15:0} + 0_32 + 3_3 + ibuf.head(1)\_{20:16} + 2_2, \) rf[ibuf.head(2)\_{15:11}] + rf[ibuf.head(2)\_{10:6}] + rf[ibuf.head(2)\_{24:22} + rf[ibuf.head(2)\_{20:16} + 2_2]

decodeLoadC and decodeLoadC:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)\_{27:25} = 23 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)\_{27:25} = 23
ibuf.deq(2)
xbuf.enq(0_16 + ibuf.head(1)\_{15:0} + 0_32 + 3_3 + ibuf.head(1)\_{20:16} + 2_2, \) 0_16 + ibuf.head(2)\_{15:0} + 0_32 + 3_3 + ibuf.head(2)\_{20:16} + 2_2)

decodeLoadC and decodeLoadPC:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)\_{27:25} = 23 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)\_{27:25} = 13
ibuf.deq(2)
xbuf.enq(0_16 + ibuf.head(1)\_{15:0} + 0_32 + 3_3 + ibuf.head(1)\_{20:16} + 2_2, \) ibuf.head(2)\_{63:32} + 0_32 + 3_3 + ibuf.head(2)\_{20:16} + 2_2)
decodeLoadC and decodeLoad:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 23 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)_{27:25} = 3 ∧ (if ibuf.head(2)_{15:11} = ibuf.head(1)_{20:16} then 0₁
else xbuf.fold(x, λy. (if ibuf.head(2)_{15:11} = x₆:₂ then 0₁ else y), 1)₁)
∧ wbuf.fold(λx. λy. (if ibuf.head(2)_{15:11} = x₆:₂ then 0₁ else y), 1₁)
ibuf.deq(2)
xBuf.enq(0₁6 ++ ibuf.head(1)_{15:0} ++ 0₃₂ ++ 3₃ ++ ibuf.head(1)_{20:16} ++ 2₂,
rf[ibuf.head(2)_{15:11}] ++ 0₁6 ++ 3₃ ++ ibuf.head(2)_{20:16} ++ 2₃)

decodeLoadC and decodeStore:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 23 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)_{27:25} = 5₃
∧ (if ibuf.head(2)_{15:11} = ibuf.head(1)_{20:16} ∨ ibuf.head(2)_{10:6} = ibuf.head(1)_{20:16} then 0₁
else xbuf.fold(x, λy. (if ibuf.head(2)_{15:11} = x₆:₂ ∨ ibuf.head(2)_{10:6} = x₆:₂ then 0₁ else y), 1₁)
∧ wbuf.fold(λx. λy. (if ibuf.head(2)_{15:11} = x₆:₂ ∨ ibuf.head(2)_{10:6} = x₆:₂ then 0₁ else y), 1₁)
ibuf.deq(2)
xbuf.enq(0₁6 ++ ibuf.head(1)_{15:0} ++ 0₃₂ ++ 3₃ ++ ibuf.head(1)_{20:16} ++ 2₂,
rf[ibuf.head(2)_{15:11}] ++ rf[ibuf.head(2)_{10:6}] ++ 0₁6 ++ 0₅ ++ 0₂₁)

decodeLoadC and decodeJz:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 23 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)_{27:25} = 5₃
∧ (if ibuf.head(2)_{15:11} = ibuf.head(1)_{20:16} ∨ ibuf.head(2)_{10:6} = ibuf.head(1)_{20:16} then 0₁
else xbuf.fold(x, λy. (if ibuf.head(2)_{15:11} = x₆:₂ ∨ ibuf.head(2)_{10:6} = x₆:₂ then 0₁ else y), 1₁)
∧ wbuf.fold(λx. λy. (if ibuf.head(2)_{15:11} = x₆:₂ ∨ ibuf.head(2)_{10:6} = x₆:₂ then 0₁ else y), 1₁)
ibuf.deq(2)
xbuf.enq(0₁6 ++ ibuf.head(1)_{15:0} ++ 0₃₂ ++ 3₃ ++ ibuf.head(1)_{20:16} ++ 2₂,
rf[ibuf.head(2)_{15:11}] ++ rf[ibuf.head(2)_{10:6}] ++ 0₁6 ++ 0₅ ++ 0₂₁)

decodeLoadPC and decodeALU:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 1₃ ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)_{27:25} = 0₃
∧ (if ibuf.head(2)_{15:11} = ibuf.head(1)_{20:16} ∨ ibuf.head(2)_{10:6} = ibuf.head(1)_{20:16} then 0₁
else xbuf.fold(x, λy. (if ibuf.head(2)_{15:11} = x₆:₂ ∨ ibuf.head(2)_{10:6} = x₆:₂ then 0₁ else y), 1₁)
∧ wbuf.fold(λx. λy. (if ibuf.head(2)_{15:11} = x₆:₂ ∨ ibuf.head(2)_{10:6} = x₆:₂ then 0₁ else y), 1₁)
ibuf.deq(2)
xbuf.enq(ibuf.head(1)_{63:32} ++ 0₃₂ ++ 3₃ ++ ibuf.head(1)_{20:16} ++ 2₂,
rf[ibuf.head(2)_{15:11}] ++ rf[ibuf.head(2)_{10:6}] ++ 0₁6 ++ 0₅ ++ 0₂₁)

decodeLoadPC and decodeLoadC:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 1₃ ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)_{27:25} = 2₃
ibuf.deq(2)
xbuf.enq(ibuf.head(1)_{63:32} ++ 0₃₂ ++ 3₃ ++ ibuf.head(1)_{20:16} ++ 2₂,
0₁6 ++ ibuf.head(2)_{15:0} ++ 0₃₂ ++ 3₃ ++ ibuf.head(2)_{20:16} ++ 2₂)

decodeLoadPC and decodeLoadPC:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 1₃ ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)_{27:25} = 1₃
ibuf.deq(2)
xbuf.enq(ibuf.head(1)_{63:32} ++ 0₃₂ ++ 3₃ ++ ibuf.head(1)_{20:16} ++ 2₂,
ibuf.head(2)_{63:32} ++ 0₃₂ ++ 3₃ ++ ibuf.head(2)_{20:16} ++ 2₂)
decodeLoadPC and decodeLoad:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)27:25 = 13 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)27:25 = 3g ∧ (if ibuf.head(2)15:11 = ibuf.head(1)20:16 then 01
else xbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_1_2 then 01 else y), 1))
∧ wbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_2_2 then 01 else y), 1)
ibuf.deq(2)
xbuf.enq(ibuf.head(1)63:32 ++ 032 ++ 33 ++ ibuf.head(1)20:16 ++ 23, rf[ibuf.head(2)15:11] ++ 032 ++ 33 ++ ibuf.head(2)20:16 ++ 32)

decodeLoadPC and decodeStore:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)27:25 = 13 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)27:25 = 3g
∧ (if ibuf.head(2)15:11 = ibuf.head(1)20:16 ∨ ibuf.head(2)10:6 = ibuf.head(1)20:16 then 01
else xbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_1_2 ∨ ibuf.head(2)10:6 = x_2_2 then 01 else y), 1))
∧ wbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_1_2 ∨ ibuf.head(2)10:6 = x_2_2 then 01 else y), 1)
ibuf.deq(2)
xbuf.enq(ibuf.head(1)63:32 ++ 032 ++ 33 ++ ibuf.head(1)20:16 ++ 23, rf[ibuf.head(2)10:6] ++ 33 ++ 01 ++ 12)

decodeLoadPC and decodeJz:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)27:25 = 13 ∧ ibuf.full?(2) ∧ xbuf.free?(2)
∧ ibuf.head(2)27:25 = 3g
∧ (if ibuf.head(2)15:11 = ibuf.head(1)20:16 ∨ ibuf.head(2)10:6 = ibuf.head(1)20:16 then 01
else xbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_1_2 ∨ ibuf.head(2)10:6 = x_2_2 then 01 else y), 1))
∧ wbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_1_2 ∨ ibuf.head(2)10:6 = x_2_2 then 01 else y), 1)
ibuf.deq(2)
xbuf.enq(ibuf.head(1)63:32 ++ 032 ++ 33 ++ ibuf.head(1)20:16 ++ 23, rf[ibuf.head(2)10:6] ++ 01 ++ 03 ++ 02)

decodeLoad and decodeALU:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)27:25 = 3g
∧ xbuf.fold(λx . λy . (if ibuf.head(1)15:11 = x_1_2 then 01 else y), 1)
∧ wbuf.fold(λx . λy . (if ibuf.head(1)15:11 = x_1_2 then 01 else y), 1) ∧ ibuf.full?(2)
∧ xbuf.free?(2) ∧ ibuf.head(2)27:25 = 03
∧ (if ibuf.head(2)15:11 = ibuf.head(1)20:16 ∨ ibuf.head(2)10:6 = ibuf.head(1)20:16 then 01
else xbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_1_2 ∨ ibuf.head(2)10:6 = x_2_2 then 01 else y), 1))
∧ wbuf.fold(λx . λy . (if ibuf.head(2)15:11 = x_1_2 ∨ ibuf.head(2)10:6 = x_2_2 then 01 else y), 1)
ibuf.deq(2)

decodeLoad and decodeLoadC:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)27:25 = 3g
∧ xbuf.fold(λx . λy . (if ibuf.head(1)15:11 = x_1_2 then 01 else y), 1)
∧ wbuf.fold(λx . λy . (if ibuf.head(1)15:11 = x_1_2 then 01 else y), 1) ∧ ibuf.full?(2)
∧ xbuf.free?(2) ∧ ibuf.head(2)27:25 = 23
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1)15:11] ++ 032 ++ 33 ++ ibuf.head(1)20:16 ++ 32, 01 ++ ibuf.head(2)15:0 ++ 032 ++ 33 ++ ibuf.head(2)20:16 ++ 23)
decodeLoad and decodeLoadPC:
when ibuf.full?(1) ∧ ibuf.free?(1) ∧ ibuf.head(1).27:25 = 3
∧ ibuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11)
∧ wbuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11) ∧ ibuf.full?(2)
∧ ibuf.free?(2) ∧ ibuf.head(2).27:25 = 1
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1).15:11] ++ 032 ++ 33 ++ ibuf.head(1).20:16 ++ 32,
ibuf.head(2).63:32 ++ 032 ++ 33 ++ ibuf.head(2).20:16 ++ 2)

decodeLoad and decodeLoad:
when ibuf.full?(1) ∧ ibuf.free?(1) ∧ ibuf.head(1).27:25 = 3
∧ ibuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11)
∧ wbuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11) ∧ ibuf.full?(2)
∧ ibuf.free?(2) ∧ ibuf.head(2).27:25 = 1
∧ (if ibuf.head(2).15:11 = ibuf.head(1).20:16 V ibuf.head(2).10:6 = ibuf.head(1).20:16 then 01
else xbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11))
∧ wbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11)
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1).15:11] ++ 032 ++ 33 ++ ibuf.head(1).20:16 ++ 32,
rf[ibuf.head(2).15:11] ++ 032 ++ 33 ++ ibuf.head(2).20:16 ++ 2)

decodeLoad and decodeStore:
when ibuf.full?(1) ∧ ibuf.free?(1) ∧ ibuf.head(1).27:25 = 3
∧ ibuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11)
∧ wbuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11) ∧ ibuf.full?(2)
∧ ibuf.free?(2) ∧ ibuf.head(2).27:25 = 1
∧ (if ibuf.head(2).15:11 = ibuf.head(1).20:16 V ibuf.head(2).10:6 = ibuf.head(1).20:16 then 01
else xbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11))
∧ wbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11)
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1).15:11] ++ 032 ++ 33 ++ ibuf.head(1).20:16 ++ 32,

decodeLoad and decodeJz:
when ibuf.full?(1) ∧ ibuf.free?(1) ∧ ibuf.head(1).27:25 = 3
∧ ibuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11)
∧ wbuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 then 01 else y), 11) ∧ ibuf.full?(2)
∧ ibuf.free?(2) ∧ ibuf.head(2).27:25 = 1
∧ (if ibuf.head(2).15:11 = ibuf.head(1).20:16 V ibuf.head(2).10:6 = ibuf.head(1).20:16 then 01
else xbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11))
∧ wbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11)
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1).15:11] ++ 032 ++ 33 ++ ibuf.head(1).20:16 ++ 32,

decodeStore and decodeALU:
when ibuf.full?(1) ∧ ibuf.free?(1) ∧ ibuf.head(1).27:25 = 3
∧ ibuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 V ibuf.head(1).10:6 = x6:2 then 01 else y), 11)
∧ wbuf.fold(ax . ay . (if ibuf.head(1).15:11 = x6:2 V ibuf.head(1).10:6 = x6:2 then 01 else y), 11)
∧ ibuf.full?(2) ∧ ibuf.free?(2) ∧ ibuf.head(2).27:25 = 1
∧ (if ibuf.head(2).15:11 = 05 V ibuf.head(2).10:6 = 05 then 01
else xbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11))
∧ wbuf.fold(ax . ay . (if ibuf.head(2).15:11 = x6:2 V ibuf.head(2).10:6 = x6:2 then 01 else y), 11)
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1).15:11] ++ rf[ibuf.head(1).10:6] ++ 33 ++ 05 ++ 12,
++ ibuf.head(2).20:16 ++ 2)
decodeStore and decodeLoadC:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1) //27 //25 = 53
∧ xbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ wbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2) //27 //25 = 23
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1) //15 //11] ++ rf[ibuf.head(1) //10 //6] ++ 33 ++ 05 ++ 12,
016 ++ ibuf.head(2) //20 //16 ++ 22)
decodeStore and decodeLoadPC:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1) //27 //25 = 53
∧ xbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ wbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2) //27 //25 = 13
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1) //15 //11] ++ rf[ibuf.head(1) //10 //6] ++ 33 ++ 05 ++ 12,
ibuf.head(2) //20 //16 ++ 33 ++ ibuf.head(2) //20 //16 ++ 22)
decodeStore and decodeLoad:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1) //27 //25 = 53
∧ xbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ wbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2) //27 //25 = 33 ∧ (if ibuf.head(2) //15 //11 = 05 then 01
else xbuf.fold(λ x . λ y . (if ibuf.head(2) //15 //11 = x //6 //2 then 01 else y), 11))
∧ wbuf.fold(λ x . λ y . (if ibuf.head(2) //15 //11 = x //6 //2 then 01 else y), 11)
∧ ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1) //15 //11] ++ rf[ibuf.head(1) //10 //6] ++ 33 ++ 05 ++ 12,
rf[ibuf.head(2) //15 //11] ++ 032 ++ 33 ++ ibuf.head(2) //20 //16 ++ 32)
decodeStore and decodeStore:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1) //27 //25 = 53
∧ xbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ wbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2) //27 //25 = 53
∧ (if ibuf.head(2) //15 //11 = 03 ∨ ibuf.head(2) //10 //6 = 03 then 01
else xbuf.fold(λ x . λ y . (if ibuf.head(2) //15 //11 = x //6 //2 ∨ ibuf.head(2) //10 //6 = x //6 //2 then 01 else y), 11))
∧ wbuf.fold(λ x . λ y . (if ibuf.head(2) //15 //11 = x //6 //2 ∨ ibuf.head(2) //10 //6 = x //6 //2 then 01 else y), 11)
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1) //15 //11] ++ rf[ibuf.head(1) //10 //6] ++ 33 ++ 05 ++ 12,
rf[ibuf.head(2) //15 //11] ++ 032 ++ 33 ++ ibuf.head(2) //20 //16 ++ 32)
decodeStore and decodeJz:
when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1) //27 //25 = 53
∧ xbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ wbuf.fold(λ x . λ y . (if ibuf.head(1) //15 //11 = x //6 //2 ∨ ibuf.head(1) //10 //6 = x //6 //2 then 01 else y), 11)
∧ ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2) //27 //25 = 43
∧ (if ibuf.head(2) //15 //11 = 05 ∨ ibuf.head(2) //10 //6 = 05 then 01
else xbuf.fold(λ x . λ y . (if ibuf.head(2) //15 //11 = x //6 //2 ∨ ibuf.head(2) //10 //6 = x //6 //2 then 01 else y), 11))
∧ wbuf.fold(λ x . λ y . (if ibuf.head(2) //15 //11 = x //6 //2 ∨ ibuf.head(2) //10 //6 = x //6 //2 then 01 else y), 11)
ibuf.deq(2)
xbuf.enq(rf[ibuf.head(1) //15 //11] ++ rf[ibuf.head(1) //10 //6] ++ 33 ++ 05 ++ 12,
rf[ibuf.head(2) //15 //11] ++ rf[ibuf.head(2) //10 //6] ++ 03 ++ 05 ++ 02)
decodeJz and decodeALU:
when ibuf.full?(1) \& xbuf.free?(1) \& ibuf.head(1)_{27:25} = 43
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1
  \& ybuf.full?(1) \& xbuf.free?(1) \& ibuf.head(2)_{27:25} = 03
  \& (if ibuf.head(2)_{15:11} = 0_3 \& (if ibuf.head(2)_{10:6} = 0_3 then 0_1
else xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(2)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1)
  \& xbuf.fold(1) \& ybuf.fold(1)_{15:11} = y_{6:2} \lor ybuf.head(2)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  ibuf.deq(2)
  xbuf.enq(2) \& (if ibuf.head(1)_{15:11} = 0_3 \& + ybuf.head(1)_{10:6} = 0_3 then 0_1
else ibuf.head(2)_{20:16} = 2_2)

decodeJz and decodeLoadC:
when ibuf.full?(1) \& xbuf.free?(1) \& ibuf.head(1)_{27:25} = 43
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& ybuf.full?(1) \& xbuf.free?(2) \& ibuf.head(2)_{27:25} = 03
  ibuf.deq(2)
  xbuf.enq(2) \& xbuf.head(1)_{15:11} = 0_3 \& + ybuf.head(1)_{10:6} = 0_3 then 0_1
else ibuf.head(2)_{20:16} = 2_2)

decodeJz and decodeLoadPC:
when ibuf.full?(1) \& xbuf.free?(1) \& ibuf.head(1)_{27:25} = 43
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& ybuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& ybuf.full?(1) \& xbuf.free?(2) \& ibuf.head(2)_{27:25} = 03
  ibuf.deq(2)
  xbuf.enq(2) \& xbuf.head(1)_{15:11} = 0_3 \& + ybuf.head(1)_{10:6} = 0_3 then 0_1
else ibuf.head(2)_{20:16} = 2_2)

decodeJz and decodeLoad:
when ibuf.full?(1) \& xbuf.free?(1) \& ibuf.head(1)_{27:25} = 43
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& ybuf.full?(1) \& xbuf.free?(2) \& ibuf.head(2)_{27:25} = 03
  ibuf.deq(2)
  xbuf.enq(2) \& xbuf.head(1)_{15:11} = 0_3 \& + ybuf.head(1)_{10:6} = 0_3 then 0_1
else ibuf.head(2)_{20:16} = 2_2)

decodeJz and decodeStore:
when ibuf.full?(1) \& xbuf.free?(1) \& ibuf.head(1)_{27:25} = 43
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(1)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  \& ybuf.full?(1) \& xbuf.free?(2) \& ibuf.head(2)_{27:25} = 03
  \& (if ibuf.head(2)_{15:11} = 0_3 \& (if ibuf.head(2)_{10:6} = 0_3 then 0_1
else xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(2)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1)
  \& xbuf.fold(2) \& ybuf.fold(1)_{15:11} = x_{6:2} \lor ybuf.head(2)_{10:6} = x_{6:2} then 0_1 else y_1, 1_1
  ibuf.deq(2)
  xbuf.enq(2) \& xbuf.head(1)_{15:11} = 0_3 \& + ybuf.head(1)_{10:6} = 0_3 then 0_1
else ibuf.head(2)_{20:16} = 2_2)
decodeJz and decodeJz:

when ibuf.full?(1) ∧ xbuf.free?(1) ∧ ibuf.head(1)_{27:25} = 43
\land xbuf.fold(λx. λy. (if ibuf.head(1)_{10:11} = x_{6:2} \lor ibuf.head(1)_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)
\land wbuf.fold(λx. λy. (if ibuf.head(1)_{15:11} = x_{6:2} \lor ibuf.head(1)_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)
\land ibuf.full?(2) ∧ xbuf.free?(2) ∧ ibuf.head(2)_{27:25} = 43
\land (if ibuf.head(2)_{15:11} = 0_6 \lor ibuf.head(2)_{10:6} = 0_5 \text{ then } 0_1
\text{ else } xbuf.fold(λx. λy. (if ibuf.head(2)_{15:11} = x_{6:2} \lor ibuf.head(2)_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1))
\land wbuf.fold(λx. λy. (if ibuf.head(2)_{15:11} = x_{6:2} \lor ibuf.head(2)_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1)

ibuf.deq(2)
xbuf.enq([rbuf.head(1)_{15:11}] ++ rbuf.head(1)_{10:6} ++ 0_3 ++ 0_5 ++ 0_2,

executeAdd and executeAdd:

when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1)_{1:0} = 2_2 ∧ xbuf.head(1)_{9:7} = 3_3 ∧ xbuf.full?(2)
\land wbuf.free?(2) ∧ xbuf.head(2)_{1:0} = 2_2 ∧ xbuf.head(2)_{9:7} = 3_3
\land xbuf.deq(2)
wbuf.enq(xbuf.head(1)_{6:2} ++ xbuf.head(1)_{7:3} + xbuf.head(1)_{4:1:10},
xbuf.head(2)_{6:2} ++ xbuf.head(2)_{7:3} + xbuf.head(2)_{4:1:10})

executeAdd and executeLoad:

when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1)_{1:0} = 2_2 ∧ xbuf.head(1)_{9:7} = 3_3 ∧ xbuf.full?(2)
\land wbuf.free?(2) ∧ xbuf.head(2)_{1:0} = 3_2
\land xbuf.deq(2)
wbuf.enq(xbuf.head(1)_{6:2} ++ xbuf.head(1)_{7:3} + xbuf.head(1)_{4:1:10},
xbuf.head(2)_{6:2} ++ dmem[xbuf.head(2)_{7:3}])

executeAdd and executeStore:

when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1)_{1:0} = 2_2 ∧ xbuf.head(1)_{9:7} = 3_3 ∧ xbuf.full?(2)
\land xbuf.head(2)_{1:0} = 1_2
dmem[xbuf.head(2)_{7:3}] ← xbuf.head(2)_{4:1:10}
\land xbuf.deq(2)
wbuf.enq(xbuf.head(1)_{6:2} ++ xbuf.head(1)_{7:3} + xbuf.head(1)_{4:1:10})

executeAdd and executeJzTaken:

when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1)_{1:0} = 2_2 ∧ xbuf.head(1)_{9:7} = 3_3 ∧ xbuf.full?(2)
\land xbuf.head(2)_{1:0} = 0_2 ∧ \neg(xbuf.head(2)_{7:3} = 0_3)
\land xbuf.deq(2)
\land xbuf.head(2)_{4:1:10}
ibuf.deqall
\land xbuf.deqall
wbuf.enq(xbuf.head(1)_{6:2} ++ xbuf.head(1)_{7:3} + xbuf.head(1)_{4:1:10})

executeAdd and executeJzNotTaken:

when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1)_{1:0} = 2_2 ∧ xbuf.head(1)_{9:7} = 3_3 ∧ xbuf.full?(2)
\land xbuf.head(2)_{1:0} = 0_2 ∧ \neg(xbuf.head(2)_{7:3} = 0_3)
\land xbuf.deq(2)
wbuf.enq(xbuf.head(1)_{6:2} ++ xbuf.head(1)_{7:3} + xbuf.head(1)_{4:1:10})

executeLoad and executeAdd:

when xbuf.full?(1) ∧ wbuf.free?(1) ∧ xbuf.head(1)_{1:0} = 3_2 ∧ xbuf.full?(2) ∧ wbuf.free?(2)
\land xbuf.head(2)_{1:0} = 2_2 ∧ xbuf.head(2)_{9:7} = 3_3
\land xbuf.deq(2)
wbuf.enq(xbuf.head(1)_{6:2} ++ dmem[xbuf.head(1)_{7:3}],
xbuf.head(2)_{6:2} ++ xbuf.head(2)_{7:3} + xbuf.head(2)_{4:1:10})
executeLoad and executeLoad:
  when xbuf.full?(1) \& wbuf.free?(1) \& xbuf.head(1):o = 3_2 \& xbuf.full?(2) \& wbuf.free?(2)
    \& xbuf.head(2):o = 3_2
    xbuf.deq(2)
    wbuf.enq(xbuf.head(1):o + dmem[xbuf.head(1):73,42],
    xbuf.head(2):o + dmem[xbuf.head(2):73,42])
executeLoad and executeStore:
  when xbuf.full?(1) \& wbuf.free?(1) \& xbuf.head(1):o = 3_2 \& xbuf.full?(2) \& xbuf.head(2):o = 1_2
    dmem[xbuf.head(2):73,42] <- xbuf.head(2):41,10
    xbuf.deq(2)
    wbuf.enq(xbuf.head(1):o + dmem[xbuf.head(1):73,42])
executeLoad and executeJzTaken:
  when xbuf.full?(1) \& wbuf.free?(1) \& xbuf.head(1):o = 3_2 \& xbuf.full?(2) \& xbuf.head(2):o = 0_2
    pc <- xbuf.head(2):41,10
    ibuf.deqall
    xbuf.deqall
    wbuf.enq(xbuf.head(1):o - xbuf.head(1):73,42 + xbuf.head(2):73,42)
executeLoad and executeJzNotTaken:
  when xbuf.full?(1) \& wbuf.free?(1) \& xbuf.head(1):o = 3_2 \& xbuf.full?(2) \& xbuf.head(2):o = 0_2
    \& -(xbuf.head(2):73,42 = 0_32)
    xbuf.deq(2)
    wbuf.enq(xbuf.head(1):o + dmem[xbuf.head(1):73,42])
executeStore and executeAdd:
  when xbuf.full?(1) \& xbuf.head(1):o = 1_2 \& xbuf.full?(2) \& wbuf.free?(1) \& xbuf.head(2):o = 2_2
    \& xbuf.head(2):9,7 = 3_3
    dmem[xbuf.head(1):73,42] <- xbuf.head(1):41,10
    xbuf.deq(2)
    wbuf.enq(xbuf.head(2):o + xbuf.head(2):73,42 + xbuf.head(2):41,10)
executeStore and executeLoad (1):
  when xbuf.full?(1) \& xbuf.head(1):o = 1_2 \& xbuf.full?(2) \& wbuf.free?(1) \& xbuf.head(2):o = 3_2
    \& -(xbuf.head(2):73,42 = xbuf.head(1):73,42)
    dmem[xbuf.head(1):73,42] <- xbuf.head(1):41,10
    xbuf.deq(2)
    wbuf.enq(xbuf.head(2):o + dmem[xbuf.head(2):73,42])
executeStore and executeLoad (2):
  when xbuf.full?(1) \& xbuf.head(1):o = 1_2 \& xbuf.full?(2) \& wbuf.free?(1) \& xbuf.head(2):o = 3_2
    \& xbuf.head(2):73,42 = xbuf.head(1):73,42
    dmem[xbuf.head(1):73,42] <- xbuf.head(1):41,10
    xbuf.deq(2)
    wbuf.enq(xbuf.head(2):o + xbuf.head(1):41,10)
executeStore and executeStore (1):
  when xbuf.full?(1) \& xbuf.head(1):o = 1_2 \& xbuf.full?(2) \& xbuf.head(2):o = 1_2
    \& xbuf.head(1):73,42 = xbuf.head(2):73,42
    dmem[xbuf.head(1):73,42] <- xbuf.head(2):41,10
    xbuf.deq(2)
executeStore and executeStore (2):
  when xbuf.full?(1) \& xbuf.head(1):o = 1_2 \& xbuf.full?(2) \& xbuf.head(2):o = 1_2
    \& -(xbuf.head(1):73,42 = xbuf.head(2):73,42)
    dmem[xbuf.head(1):73,42] <- xbuf.head(1):41,10
    dmem[xbuf.head(2):73,42] <- xbuf.head(2):41,10
    xbuf.deq(2)
executeStore and executeJzTaken:
  when xbuf.full?(1) ∧ xbuf.head(1,0) = 12 ∧ xbuf.full?(2) ∧ xbuf.head(2,1,0) = 02
                     ∧ xbuf.head(2,73,42) = 032
      pc ← xbuf.head(2,41,10)
      dmem[xbuf.head(1,73,42)] ← xbuf.head(1,41,10)
      ibuf.deqall
      xbuf.deqall

executeStore and executeJzNot Taken:
  when xbuf.full?(1) ∧ xbuf.head(1,0) = 12 ∧ xbuf.full?(2) ∧ xbuf.head(1,0) = 02
                     ∧ ¬(xbuf.head(2,73,42) = 032)
      dmem[xbuf.head(1,73,42)] ← xbuf.head(1,41,10)
      xbuf.deq(2)

executeJzNot Taken and executeAdd:
  when xbuf.full?(1) ∧ xbuf.head(1,0) = 02 ∧ ¬(xbuf.head(1,73,42) = 032) ∧ xbuf.full?(2)
                     ∧ wbuf.free?(1) ∧ xbuf.head(2,1,0) = 22 ∧ xbuf.head(2,9,7) = 33
      wbuf.deq(2)
      wbuf.enq(2)

executeJzNot Taken and executeLoad:
  when xbuf.full?(1) ∧ xbuf.head(1,0) = 02 ∧ ¬(xbuf.head(1,73,42) = 032) ∧ xbuf.full?(2)
                     ∧ wbuf.free?(1) ∧ xbuf.head(2,1,0) = 32
      xbuf.deq(2)
      wbuf.enq(2)

executeJzNot Taken and executeStore:
  when xbuf.full?(1) ∧ xbuf.head(1,0) = 02 ∧ ¬(xbuf.head(1,73,42) = 032) ∧ xbuf.full?(2)
                     ∧ xbuf.head(2,1,0) = 12
      dmem[xbuf.head(2,73,42)] ← xbuf.head(2,41,10)
      xbuf.deq(2)

executeJzNot Taken and executeJzTaken:
  when xbuf.full?(1) ∧ xbuf.head(1,0) = 02 ∧ ¬(xbuf.head(1,73,42) = 032) ∧ xbuf.full?(2)
                     ∧ xbuf.head(2,1,0) = 02 ∧ xbuf.head(2,73,42) = 032
      pc ← xbuf.head(2,41,10)
      ibuf.deqall
      xbuf.deqall

executeJzNot Taken and executeJzNot Taken:
  when xbuf.full?(1) ∧ xbuf.head(1,0) = 02 ∧ ¬(xbuf.head(1,73,42) = 032) ∧ xbuf.full?(2)
                     ∧ xbuf.head(2,1,0) = 02 ∧ ¬(xbuf.head(2,73,42) = 032)
      xbuf.deq(2)

writeback and writeback (1):
  when wbuf.full?(1) ∧ wbuf.full?(2) ∧ wbuf.head(1,36,32) = wbuf.head(2,36,32)
     rf[wbuf.head(1,36,32)] ← wbuf.head(2,31,0)
     wbuf.deq(2)

writeback and writeback (2):
  when wbuf.full?(1) ∧ wbuf.full?(2) ∧ ¬(wbuf.head(1,36,32) = wbuf.head(2,36,32))
     rf[wbuf.head(1,36,32)] ← wbuf.head(1,31,0)
     rf[wbuf.head(2,36,32)] ← wbuf.head(2,31,0)
     wbuf.deq(2)
Appendix F

Eliding AX: 2-stage Pipeline

state
- pc : register 32
- rf : array [5] 32
- imem : array [8] 32
- dmem : array [8] 32

rules
- decodeJzNotTaken:
  - when \( \text{imem}[pc]_{27:25} = 43 \land \neg (\text{rf}[\text{imem}[pc]_{15:11}] = 0_{32}) \)
  - \( pc \leftarrow pc + 1_{32} \)
- decodeJzTaken:
  - when \( \text{imem}[pc]_{27:25} = 43 \land \text{rf}[\text{imem}[pc]_{15:11}] = 0_{32} \)
  - \( pc \leftarrow \text{rf}[\text{imem}[pc]_{10:6}] \)
- decodeStore and executeStore:
  - when \( \text{imem}[pc]_{27:25} = 5_{3} \)
  - \( pc \leftarrow pc + 1_{32} \)
  - \( \text{dmem}[\text{rf}[\text{imem}[pc]_{15:11}]] \leftarrow \text{rf}[\text{imem}[pc]_{10:6}] \)
- decodeLoad and executeLoad:
  - when \( \text{imem}[pc]_{27:25} = 3_{3} \)
  - \( pc \leftarrow pc + 1_{32} \)
  - \( \text{rf}[\text{imem}[pc]_{20:16}] \leftarrow \text{dmem}[\text{rf}[\text{imem}[pc]_{15:11}]] \)
- decodeLoadPC and executeAdd:
  - when \( \text{imem}[pc]_{27:25} = 1_{3} \)
  - \( pc \leftarrow pc + 1_{32} \)
  - \( \text{rf}[\text{imem}[pc]_{20:16}] \leftarrow \text{pc} \)
- decodeLoadC and executeAdd:
  - when \( \text{imem}[pc]_{27:25} = 2_{3} \)
  - \( pc \leftarrow pc + 1_{32} \)
  - \( \text{rf}[\text{imem}[pc]_{20:16}] \leftarrow 0_{16} + \text{imem}[pc]_{15:0} \)
- decodeALU and executeAdd:
  - when \( \text{imem}[pc]_{27:25} = 0_{3} \land \text{imem}[pc]_{24:22} = 3_{3} \)
  - \( pc \leftarrow pc + 1_{32} \)
  - \( \text{rf}[\text{imem}[pc]_{20:16}] \leftarrow \text{rf}[\text{imem}[pc]_{15:11}] + \text{rf}[\text{imem}[pc]_{10:6}] \)
Appendix G

Eliding AX: 4-stage Pipeline

G.1 Merging stages 3 and 4: removing wbuf

state

\[ \begin{align*}
\text{pc} & : \text{register 32} \\
\text{rf} & : \text{array [5] 32} \\
\text{imem} & : \text{array [8] 32} \\
\text{dmem} & : \text{array [8] 32} \\
\text{ibuf} & : \text{fifo 64} \\
\text{xbuf} & : \text{fifo 74}
\end{align*} \]

rules

executeJzNotTaken:

when \( \text{wbuf} . \text{full?} (1) \land \text{wbuf} . \text{head}(1)_{1:0} = 0_2 \land \lnot (\text{xbuf} . \text{head}(1)_{73:42} = 0_32) \)

\( \text{xbuf} . \text{deq}(1) \)

executeJzTaken:

when \( \text{wbuf} . \text{full?} (1) \land \text{wbuf} . \text{head}(1)_{1:0} = 0_2 \land \text{xbuf} . \text{head}(1)_{73:42} = 0_32 \)

\[ \begin{align*}
\text{pc} & \leftarrow \text{xbuf} . \text{head}(1)_{41:10} \\
\text{ibuf} . \text{deqall} \\
\text{xbuf} . \text{deqall}
\end{align*} \]

executeStore:

when \( \text{wbuf} . \text{full?} (1) \land \text{wbuf} . \text{head}(1)_{1:0} = 1_2 \)

\[ \begin{align*}
\text{dmem}[\text{xbuf} . \text{head}(1)_{73:42}] & \leftarrow \text{xbuf} . \text{head}(1)_{41:10} \\
\text{xbuf} . \text{deq}(1)
\end{align*} \]

decodeJz:

when \( \text{ibuf} . \text{full?}(1) \land \text{wbuf} . \text{free?}(1) \land \text{ibuf} . \text{head}(1)_{27:25} = 4_3 \)

\[ \begin{align*}
\lnot \text{xbuf} . \text{fold}(\lambda x . \lambda y . (\text{if ibuf} . \text{head}(1)_{15:11} = x_{6:2} \lor \text{ibuf} . \text{head}(1)_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1) \\
\text{ibuf} . \text{deq}(1) \\
\text{xbuf} . \text{enq}[\text{rf}][\text{ibuf} . \text{head}(1)_{15:11}] + \text{rf}[[\text{ibuf} . \text{head}(1)_{16:6}] + 0_3 + 0_6 + 0_2]
\end{align*} \]

decodeStore:

when \( \text{ibuf} . \text{full?}(1) \land \text{wbuf} . \text{free?}(1) \land \text{ibuf} . \text{head}(1)_{27:25} = 5_3 \)

\[ \begin{align*}
\lnot \text{xbuf} . \text{fold}(\lambda x . \lambda y . (\text{if ibuf} . \text{head}(1)_{15:11} = x_{6:2} \lor \text{ibuf} . \text{head}(1)_{10:6} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1) \\
\text{ibuf} . \text{deq}(1) \\
\text{xbuf} . \text{enq}[\text{rf}][\text{ibuf} . \text{head}(1)_{15:11}] + \text{rf}[[\text{ibuf} . \text{head}(1)_{16:6}] + 3_3 + 0_6 + 1_2)
\end{align*} \]

decodeLoad:

when \( \text{ibuf} . \text{full?}(1) \land \text{wbuf} . \text{free?}(1) \land \text{ibuf} . \text{head}(1)_{27:25} = 3_3 \)

\[ \begin{align*}
\lnot \text{xbuf} . \text{fold}(\lambda x . \lambda y . (\text{if ibuf} . \text{head}(1)_{15:11} = x_{6:2} \text{ then } 0_1 \text{ else } y), 1_1) \\
\text{ibuf} . \text{deq}(1) \\
\text{xbuf} . \text{enq}[\text{rf}][\text{ibuf} . \text{head}(1)_{15:11}] + 0_32 + 3_3 + \text{ibuf} . \text{head}(1)_{20:16} + 3_2)
\end{align*} \]
decodeLoadPC:
  when ibuf.full?(1) \& xbuf.fre?(1) \& ibuf.head(1)_{27:25} = 1_{3}
  ibuf.deq(1)
xbuf.enq(ibuf.head(1)_{03:32} ++ 0_{32} ++ 3_{3} ++ ibuf.head(1)_{20:16} ++ 2_{2})

decodeLoadC:
  when ibuf.full?(1) \& xbuf.fre?(1) \& ibuf.head(1)_{27:25} = 2_{3}
  ibuf.deq(1)
xbuf.enq(0_{16} ++ ibuf.head(1)_{15:0} ++ 0_{32} ++ 3_{3} ++ ibuf.head(1)_{20:16} ++ 2_{2})

decodeALU:
  when ibuf.full?(1) \& xbuf.fre?(1) \& ibuf.head(1)_{27:25} = 0_{3}
  \& xbuf.fold(\lambda x \cdot \lambda y \cdot (if \ ibuf.head(1)_{15:11} = x_{6:2} \lor \ ibuf.head(1)_{10:6} = x_{6:2} \ then \ 0_{1} \ else \ y), 1_{1})
  ibuf.deq(1)
xbuf.enq(rf[ibuf.head(1)_{15:11}] ++ rf[ibuf.head(1)_{10:6}] ++ ibuf.head(1)_{24:22} + + ibuf.head(1)_{20:16} ++ 2_{2})

fetch:
  when ibuf.fre?(1)
  pc \leftarrow pc + 1_{32}
  ibuf.enq(pc ++ imem[pc])

executeLoad and writeback:
  when xbuf.full?(1) \& xbuf.head(1)_{1:0} = 3_{2}
  rf[xbuf.head(1)_{6:2}] \leftarrow dmem[xbuf.head(1)_{73:42}]
xbuf.deq(1)

executeAdd and writeback:
  when xbuf.full?(1) \& xbuf.head(1)_{1:0} = 2_{2} \& xbuf.head(1)_{6:7} = 3_{3}
  rf[xbuf.head(1)_{6:2}] \leftarrow xbuf.head(1)_{73:42} + xbuf.head(1)_{41:10}
xbuf.deq(1)
G.2 Merging stages 2 and 3: removing xbuf

state
- pc: register 32
- rf: array [5] 32
- imem: array [8] 32
- dmem: array [8] 32
- ibuf: fifo 64

rules
fetch:
  when ibuf.free?(1)
    pc ← pc + 132
    ibuf.enq(pc ++ imem[pc])

decodeALU and executeAdd and writeback:
  when ibuf.full?(1) ∧ ibuf.head(1)27:25 = 0_3 ∧ ibuf.head(1)24:22 = 3_3
  ibuf.deq(1)

decodeLoadC and executeAdd and writeback:
  when ibuf.full?(1) ∧ ibuf.head(1)27:25 = 2_3
    rf[ibuf.head(1)20:16] ← 0_{16} + ibuf.head(1)15:0
    ibuf.deq(1)

decodeLoadPC and executeAdd and writeback:
  when ibuf.full?(1) ∧ ibuf.head(1)27:25 = 1_3
    rf[ibuf.head(1)20:16] ← ibuf.head(1)63:32
    ibuf.deq(1)

decodeLoad and executeLoad and writeback:
  when ibuf.full?(1) ∧ ibuf.head(1)27:25 = 3_3
    rf[ibuf.head(1)20:16] ← dmem[rf[ibuf.head(1)15:11]]
    ibuf.deq(1)

decodeStore and executeStore:
  when ibuf.full?(1) ∧ ibuf.head(1)27:25 = 5_3
    dmem[rf[ibuf.head(1)15:11]] ← rf[ibuf.head(1)10:6]
    ibuf.deq(1)

decodeJz and executeJzTaken:
  when ibuf.full?(1) ∧ ibuf.head(1)27:25 = 4_3 ∧ rf[ibuf.head(1)15:11] = 0_{32}
    pc ← rf[ibuf.head(1)10:6]
    ibuf.deqall

decodeJz and executeJzNotTaken:
  when ibuf.full?(1) ∧ ibuf.head(1)27:25 = 4_3 ∧ ¬(rf[ibuf.head(1)15:11] = 0_{32})
    ibuf.deq(1)
G.3 Merging stages 1 and 2: removing *ibuf*

**state**
- `pc` : register 32
- `rf` : array [5] 32
- `imem` : array [8] 32
- `dmem` : array [8] 32

**rules**

*fetch and decode Jz and execute JzNot Taken:*
  - when `imem[pc][27:25] = 43 ∧ ¬(rf[imem[pc][15:11]] = 032)
    - `pc ← pc + 132`

*fetch and decode Jz and execute JzTaken:*
  - when `imem[pc][27:25] = 43 ∧ rf[imem[pc][15:11]] = 032`
    - `pc ← rf[imem[pc][10:6]]`

*fetch and decode Store and execute Store:*
  - when `imem[pc][27:25] = 53`
    - `pc ← pc + 132`
    - `dmem[rf[imem[pc][15:11]]] ← rf[imem[pc][10:6]]`

*fetch and decode Load and execute Load and writeback:*
  - when `imem[pc][27:25] = 33`
    - `pc ← pc + 132`
    - `rf[imem[pc][20:16]] ← dmem[rf[imem[pc][15:11]]]`

*fetch and decode LoadPC and execute Add and writeback:*
  - when `imem[pc][27:25] = 13`
    - `pc ← pc + 132`
    - `rf[imem[pc][20:16]] ← pc`

*fetch and decode LoadC and execute Add and writeback:*
  - when `imem[pc][27:25] = 23`
    - `pc ← pc + 132`
    - `rf[imem[pc][20:16]] ← 016 ++ imem[pc][15:0]`

*fetch and decode ALU and execute Add and writeback:*
    - `pc ← pc + 132`
    - `rf[imem[pc][20:16]] ← rf[imem[pc][15:11]] + rf[imem[pc][10:6]]`
Bibliography


