Extended Update Plans

Jiří Mencák

A thesis submitted to the University of Huddersfield
in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

The University of Huddersfield

School of Computing and Engineering

May 2003

Abstract

Formal methods are gaining popularity as a way of increasing the reliability of systems through the use of mathematically based techniques. Their domain is no longer restricted to purely academic environments and examples, as they are slowly moving into industrial settings. The slow rate at which this transition takes place is mainly due to the perceived difficulty of formalising the behaviour of systems. While this is undoubtedly true, it is not the case with all formal methods.

Update Plans are a powerful formalism for the description of computer architectures and intermediate to low-level languages. They are a declarative specification language with an underlying imperative machine model. The descriptions using Update Plans are clear, compact, intuitive, unambiguous and simple to read. These characteristics allow for the minimisation of possible errors at early stages of the development process even before a verification takes place.

In this thesis an overview of the Update Plans formalism is given and a number of real-world applications is shown. The investigation of the application area focuses on computer architectures for which various specifications already exist. The comparison of Update Plan specifications to other specifications provides a useful insight into the strengths and shortcomings of the formalism. The shortcomings, in particular the lack of synchronisation primitives and modularity, are addressed by the development and evaluation of several syntactic and semantic extensions described in this thesis. The extended formalism is also compared to other specification languages and conclusions are drawn.

Acknowledgements

The research detailed in this thesis was funded by the Engineering and Physical Sciences Research Council (EPSRC) and partially by the School of Computing and Engineering of the University of Huddersfield. I wish to express my gratitude to both of these institutions for their financial support.

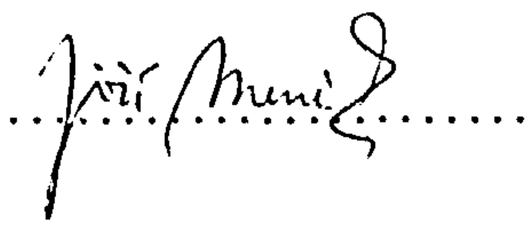
I am indebted to my director of studies Dr. Hugh R. Osborne and my supervisor Dr. Adrian R. Jackson, for their help and guidance over the past three years.

Special thanks belong to my parents and my girlfriend Kamila for their continuous support and encouragement.

Statement of Original Authorship

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Brno, 4th April 2004



Contents

Introduction

1 Context

The first recorded ideas about modern formal methods were those of Leibniz (1646–1716), who dreamt [47] about a machine called "calculus ratiocinator" which would decide any question given in a language "characteristica universalis" rich enough to describe any kind of phenomena.

Many years have passed since these ideas surfaced, and formal methods are today advocated as a means of increasing the reliability of systems as exhaustive testing through simulation is no longer possible due to the ever increasing complexity of integrated circuits. Furthermore, the gap between the performance of current systems used for simulation and the complexity of systems under development is increasing rather than shrinking.

The current uses of formal methods include but are not limited to safety and security-critical systems in domains such as transport, defence, banking and medical applications, where the cost of failure is unacceptably high. The losses suffered by bad microprocessor design in particular can be devastating. For example, the cost of Ariane 5 to the European Space Agency was $7,000 million and 10 years of design due to an overflow error during a conversion of a number from a 64-bit format to a 16-bit format [27]. The Pentium floating-point bug [63] in the division algorithm cost Intel an estimated $500 million. Perhaps not surprisingly the use of formal methods is spreading from government agencies such as NASA or MoD into a wider industrial sphere.

The "weakest link" in the development of a system is usually its specification. Many inconsistencies and ambiguities can be discovered just by going through the process of rigorous specification. By writing a formal specification and devising a verification strategy one is forced to think about the system in new ways, new questions are raised, new insights into the system are gained and thus the number of possible errors is minimised at early stages of the development.

While there are methods for formal development of high level software (e.g. [7, 14]), and for hardware description (e.g. [36, 57, 73]), there seems to be a shortage of specification and development formalisms at the intermediate level of machine architectures and instruction sets

as the semantics of low-level languages is still [48, 70, 77] unjustly assumed trivial. Although it is possible to describe the semantics of low-level languages by various methods [13, 16, 34, 66, 69], they either lack formal semantics themselves or the readability of specifications aimed at this level is generally bad.

Update Plans [60], on the other hand, aim to fill the gap among formal languages by providing a formal, clear, unambiguous and expressive way of describing intermediate to low-levels of machine architectures.

Many formal methods (some of which are briefly described and compared with Update Plans in this thesis) are difficult to grasp. The learning time for the production of usable designs with such methods is usually very long, and an efficient use of a specification language relies heavily on its detailed knowledge. This also results in all sorts of other difficulties. Firstly, it is very easy to formalise the wrong thing in this environment. Secondly, a hard to understand design description makes any verification more complicated.

Update Plans, on the other hand, have a short learning curve—they are easy to learn and understand as its basic concepts are very simple and confidence in using the formalism is achieved very soon. Designs written in Update Plans are highly communicable and compact formal descriptions.

2 Update Plans

Update Plans (UP for short) are a formalism for the description of (abstract) machines and algorithms. A specification produced by UP combines both the structural and behavioural model of a system. The structural descriptions are concrete, detailed and low-level (although it is easy to abstract away from irrelevant details). With respect to algorithmic structures the descriptions are abstract and high-level. These characteristics make UP particularly suitable as a specification language for the description of large classes of machine architectures [55, 58, 59, 61] and as a target language in compiler design [54, 60].

In [54] the Update Plans formalism (then called Update Schemes) was introduced by Hans Meijer as a target language in the framework of the design of a translator generator. Since then it has been extended to the description of machines and algorithms.

Hugh Osborne designed a formal semantics [58] and developed a powerful macro-like mechanism called *archetypes*. He also simplified the typing regime, introduced parallelism [60], constructed a prototype implementation based on [45], investigated a number of typical applications [59, 61] and approached the problem of the formal verification of UP specifications.

A brief description of most of the above-mentioned past areas of Update Plans research is given in the following points.

- **Semantics.** The presence of a formal semantics for any specification language is a necessity for any kind of formal reasoning about specifications written in that language. In UP

this has been done in [58] by defining the underlying imperative model and the referentially transparent semantics of UP.

• **Archetypes.** Archetypes are an abstraction and structure reuse mechanism that has been introduced into Update Plans in order to further increase their expressiveness [60]. Although the primary motivation was abstracting away the details of addressing modes, they can be used for a variety of other purposes where it is desirable to express multiple update schemes by a single archetype call.

• **Parallelism.** Parallelism is inherent in Update Plans. While there is (to some extent) no need for an explicit mechanism to describe it, it has been introduced [60] to increase legibility by providing a way to allow many update schemes to be combined into one atomic update.

• **Typing.** The type information introduced to Update Plans in [60] is not only important for type checking, but it also has serious consequences for the implementability and formal verification in Update Plans.

• **Verification.** Perhaps most importantly, the work on proving semantic equivalence between update plans (possibly representing various levels of description) has started [60].

• **Applications.** A number of machines varying from very abstract to concrete have been described. One of the first larger applications was a machine for a simple functional language [59]. Two lower level specifications have been given for a more realistic machine model, taken from [25]. The more abstract of these was a machine for a tree language, typical of intermediate code generated by a compiler. The more concrete was a PDP-11 style machine. A linearisation of the tree code (register allocation) was given transforming it to concrete machine code. A proof of the semantic equivalence of the two specifications under the transformation was shown, which verified the register allocation algorithm. A specification of a pipelined RISC processor has been given [61]. UP have also been applied to a (partial) specification of the Java Virtual Machine [55]. Using an abstraction mechanism similar to the archetype mechanism, elementary VLSI components have been specified [53].

Update Plans have also proved very useful in education. Apart from their didactic use in explaining the intricacies of addressing modes and the semantics of instructions of a computer architecture simulator [62], they can also be used as a compact means of explaining algorithms such as rebalancing operations in AVL trees [60].

2.1 Research questions

The main aim of this project is to develop UP as a specification and verification formalism, primarily at the level of low-level code and hardware. To achieve this a hierarchical modular specification and validation structure for UP needs to developed. This would provide a formalism in which multi-level specifications of architectures can be given, each level being a translation or transformation of its neighbouring levels. The formally defined translation

between levels will provide a proof of the semantic equivalence of the specifications.

A second aim is to extend the syntax and semantics of UP to cover a wide range of parallel paradigms and, in particular, their communication and synchronisation primitives.

To achieve this the problems of parallelism, modularity and formal verification need to be addressed.

- **Parallelism.** More general mechanisms for specifying parallel systems need to be introduced, allowing for the specification of asynchronous parallelism and related communication primitives. The semantics of the new constructs must be clearly defined. While some preliminary investigation of parallelism in UP has been done, this has not gone much further than replacing nondeterministic execution of an applicable update scheme with simultaneous execution of all applicable update schemes; so in order to maintain well-behaviour a host of artificial semaphore-like constructs have to be used.

- **Modularity.** A modular structure needs to be introduced into UP in order to provide the formalism with a mechanism for structure reuse, for information hiding and to encourage hierarchical specification. A specification at one level, e.g. that of the fetch/execute cycle could then be defined in terms of a lower level specification such as microcode, and in turn be used to specify higher level activities such as the instruction set.

- **Formal verification.** A proof system needs to be developed that, given UP specifications of two machine architectures and a translation, would prove the semantic equivalence of the specifications under the translation. This work should be based on the concepts of semantic equivalence, translation and irrelevance [60].

3 Organisation

This thesis contains an evaluation of Update Plans as a specification formalism for hardware architectures, and suggests extensions based on this evaluation. It is divided into two parts. Unless indicated otherwise by marginal notes giving forward references to sections of chapter 5, the first part uses basic Update Plans as defined by [60], and the suggested extensions are described in the second part. The first part of this thesis is an evaluation of basic Update Plans, and consists of three case studies involving instruction set specifications of machines differing in the degree of abstraction.

A tutorial on basic Update Plans is given in chapter 1. A lot of material in this chapter was adapted with minor amendments from [60] and [61].

The PDP-11 specification in chapter 2 is the first attempt to specify a large subset of a real machine's instruction set using Update Plans. It was chosen as a candidate for specification because some historically important formal specifications of the PDP-11 exist [20, 69]. Consideration of a more detailed specification at the fetch/execute cycle drove the development of some semantic extensions described in chapter 6.

The first chapters to make some advance use of the syntactic extensions to Update Plans described in chapter 5 are chapters 3 and 4. Both of these chapters feature modern, abstract, but very different architectures. Chapter 3 has the UP specification for the SPARC-V9 architecture. Again, there was already a partial specification of the SPARC-V9 architecture [65], but this is aimed at the specification of connections and communication rather than at a concise specification of the instruction set. In chapter 4 the Java Virtual Machine (JVM) was chosen to test UP's suitability for describing more abstract instruction sets using a wide variety of types.

The second part of this thesis proposes some extensions to Update Plans and contains a comparison of Update Plans to other formal methods.

Chapter 5 covers the syntactic extensions brought about during the development of the SPARC-V9 and JVM instruction set specifications in the first part of the thesis. In particular this chapter discusses the changes in the revised Update Plans grammar and the extensions to the typing mechanism which, in many cases, allow multi-level specifications.

The concept of sequential update schemes and archetypes is introduced in chapter 6 together with other semantic extensions to make the Update Plans formalism more expressive and consistent. More importantly, this chapter gives possible answers to the research questions of parallelism and modularity. The problem with synchronisation of parallel processes is solved by the introduction of a general synchronisation primitive (sequential update schemes) that augments non-deterministic model of Update Plans by explicitly stating the order in which updates will be applied. Sequential archetypes extend the possibilities for information hiding and structure reuse by encapsulating a series of synchronised updates at one level of abstraction into a modular update easily identifiable against a corresponding update at another level of description.

Chapter 7 contains a more comprehensive example of the application of sequential update schemes and archetypes in a specification of a theoretical model of computation—Parallel Random Access Machine (PRAM). PRAM provides a useful test-bed for the semantic extensions as it specifies strictly sequential operations (the fetch/execute cycle) within a massively parallel context.

The penultimate chapter evaluates Update Plans. Other existing formalisms which could be used for the specification of intermediate levels of hardware architectures are considered. A brief description of each of the formalisms is given and they are compared to Update Plans. This chapter uses a few simple examples to show the advantages the introduction of sequences brings to UP.

Finally, conclusions are drawn and future research directions are suggested in chapter 9.

4 Notational conventions

Throughout this thesis, the following notational conventions are observed.

- Production rules used to describe basic and Extended Update Plans grammars given in this thesis make use of the notational conventions of the specification formalism ASF+SDF [43, 74] in which the notation $\{S\ term\}^+$ indicates a list of one or more S's separated by the terminal *term*. If the $^+$ is replaced by a * the list may also be empty. The suffix *-opt* indicates zero or one occurrences of its nonterminal. More notational conventions used by the ASF+SDF specification formalism can be found in appendix A on page 121.
- Update plans given in this thesis use the `typewriter` font. This convention has two exceptions. Firstly, update plan comments and 'meta-variables' such as *lhs* use the *italic* typestyle. The second exception is update plan elements which have been commented out. They use the *`slanted typewriter`* font.
- All *emphasised* words appearing freely in the text are glossary terms explained in appendix F for quick reference. The only exception are references to update plan 'meta-variables' which use the *emphasised* (*italic*) typestyle.
- Whenever a reference to an instruction field is made, a sans serif typeface is used. The ‖ symbol designates concatenation of bit vectors, the % symbol is arithmetic modulo, and × is used for multiplication. Symbols &, |, and ^ are used for bit-wise AND, OR, and XOR operations respectively. And finally, symbols $=$, $\neq$, $<$, $>$, $\leq$ and $\geq$ have their usual mathematical meanings.
- Marginal notes used in the first part of this thesis are forward references to sections of chapter 5 where syntactic UP extensions are explained. References to the trivial extension introduced in section 4 of the same chapter are not made.
- The American spelling 'program' is used to denote computer software.

Part I

Basic Update Plans

Introduction

Part one of this thesis serves as an introduction to Update Plans with the focus on the description of hardware architectures. It shows three specifications of instruction sets of machines at various degrees of abstraction, ordered from the most concrete to the most abstract. Each of the specifications provide some interesting insights into the formalism and uncovers its various shortcomings, which are addressed in the second part of the thesis. Although the specifications use basic Update Plans as described in [60], some advance use (in chapters 3 and 4) is made of syntactic changes to the formalism described in chapter 5.

Chapter 1

Update Plans

This chapter gives an informal overview of basic Update Plans. Only the basic facts necessary to understand the specifications in the first part and the reasons for the extensions in the second part of the thesis are introduced. More information on Update Plans along with their complete formal syntactic and semantic definition can be found in [58–61].

1 Basic Update Plans

An update plan specifies state transitions in an abstract machine. This machine consists of a number of *stores*, each containing a linear countably infinite sequence of memory *cells* (e.g. bytes or machine words) between *locators* (addresses). Note that it is not the cells themselves that are addressed, but the boundaries between cells as shown in figure 1.1.

Triples $\alpha[\xi]\beta$ are called *locator expressions*, where α and β are locators, $\alpha \leq \beta$ and the cells between the addresses α and β contain (a particular representation of) the value of ξ. A set of locator expressions is *consistent* if there are no two or more expressions in the set which specify the contents of some cell to have different values. A consistent set of locator expressions describes a (sub)*configuration* of the machine.

State transitions in the abstract machine are described by *update schemes*, which are constructed from two sets of locator expressions forming the left-hand side (lhs) and the right-hand side (rhs), separated by a *guard* ($=\![\ \gamma\ ]\!\Rightarrow$) which carries an applicability condition γ. A guard whose condition is always true can be simply written as $\Longrightarrow$.

An *update plan* is a set of update schemes, each of which may contain unspecified values (variables). Variables are indicated by lower case words, constants are indicated either by a value or, symbolically, by upper case words. An update scheme containing no unspecified values is called an *update rule*. Update schemes yield update rules by instantiation. Both the left and right-hand sides of the resulting update rule must be self-consistent, i.e. all locator expressions in the left and right-hand side must be mutually consistent.

Traditional array addressing Update Plan addressing

Figure 1.1: Addressing in arrays and Update Plans

An update rule is *applicable* to a given configuration if its left-hand side is a subset of that configuration and its guard is true. The *memory* then may be minimally updated such that thereafter all locator expressions in its right-hand side are satisfied.

An *initial configuration*, which specifies the initial state of the memory before any update scheme is applied, and an update plan form an *update script*. The update plan is executed by repeatedly (non-deterministically) choosing an applicable update scheme from the plan, and applying it, until the configuration is such that no scheme is applicable. This *final configuration* is the result of the script.

The overview of the basic Update Plans syntax is given by the following grammar.

⟨*script*⟩ → ⟨*configuration*⟩ "." ⟨*plan*⟩

⟨*plan*⟩ → ⟨*item*⟩*

⟨*item*⟩ → ⟨*scheme*⟩ "."

⟨*configuration*⟩ → ⟨*locator expression*⟩*

⟨*scheme*⟩ → ⟨*configuration*⟩ ⟨*guard*⟩ ⟨*configuration*⟩

⟨*guard*⟩ → "=[" ⟨*term*⟩ "]⇛"

⟨*locator expression*⟩ → ⟨*locator*⟩ "[" ⟨*text*⟩ "]" ⟨*locator*⟩

⟨*locator*⟩ → ⟨*term*⟩

⟨*text*⟩ → ⟨*term*⟩*

A ⟨*term*⟩ is an expression built from constants, variables and operators. Note that the grammar presented here is a very simplified version of the Update Plans grammar to demonstrate the basic structure. This grammar will be extended at several places in this chapter, but will still not be presented in its entirety. The complete basic Update Plans grammar can be found in [60], and a complete and updated grammar for Extended Update Plans in appendix A.

The classic two-scheme update plan in example 1 demonstrates Euclid's algorithm for computing the GCD of the number initially between A and B and that initially between B and C. The constants A, B and C are fixed locators and x and y are unspecified values.

Example 1

$$\mathtt{A[x]B\ B[y]C} =[\ \mathtt{x} < \mathtt{y}\ ]\Rightarrow \mathtt{B[y-x]C}.$$
$$\mathtt{A[x]B\ B[y]C} =[\ \mathtt{x} > \mathtt{y}\ ]\Rightarrow \mathtt{A[x-y]B}.$$

If at any stage of the computation the machine configuration contains `A[9]B` and `B[6]C`, (only) the second update scheme (instantiating into an update rule) in example 1 is applicable, whereupon the `9` is replaced by a `3`.

Syntactic sugar

The following points describe notational simplifications which make update plans more legible.

- A superfluous locator may be omitted. A locator is superfluous if its removal does not lead to any confusion.
- Contiguous sequences may be concatenated. Two expressions $x[s]y$ and $y[t]z$ may be written as $x[s]y[t]z$.
- Locators may be also omitted when concatenating contiguous sequences so that $x[s]y[t]z$ may also be written as $x[s\ t]z$.
- Identical left-hand sides may be shared. A *repeat* of the previous left-hand side is indicated by the 'repeat' symbol '"'.
- Other notational conventions (I/O, the introduction of the program counter and alternatives) are described in sections of their own.

I/O

As input/output instructions are one of most frequently used operations, a mechanism for hiding the details and so sweetening the syntax is provided. Input streams which are described by an update scheme

$$\cdots\ IP[\mathtt{i}]\ \mathtt{i}[input]\mathtt{j}\ \cdots \Rightarrow\!\!\!\!\!\!=\!\{\ g\ \}\!\!\Rightarrow \cdots\ IP[\mathtt{j}]\ \cdots .$$

may be written as

$$\cdots\ ?IP[input]\ \cdots \Rightarrow\!\!\!\!\!\!=\!\{\ g\ \}\!\!\Rightarrow \cdots .$$

For the standard input stream the '?[*input*]' can be used when this is unambiguously defined.

Similarly, output streams

$$\cdots\ OP[\mathtt{o}]\ \cdots \Rightarrow\!\!\!\!\!\!=\!\{\ g\ \}\!\!\Rightarrow \cdots\ OP[\mathtt{p}]\ \mathtt{o}[output]\mathtt{p}\ \cdots .$$

may be written as

$$\cdots \Rightarrow\!\!\!\!\!\!=\!\{\ g\ \}\!\!\Rightarrow \cdots\ !OP[output]\ \cdots .$$

Again, the '![*output*]' can be used for the standard output stream, if it is defined.

Program counter

By acknowledging the existence of a *program counter* at a fixed locator (by convention `PC`), update schemes exhibiting the pattern

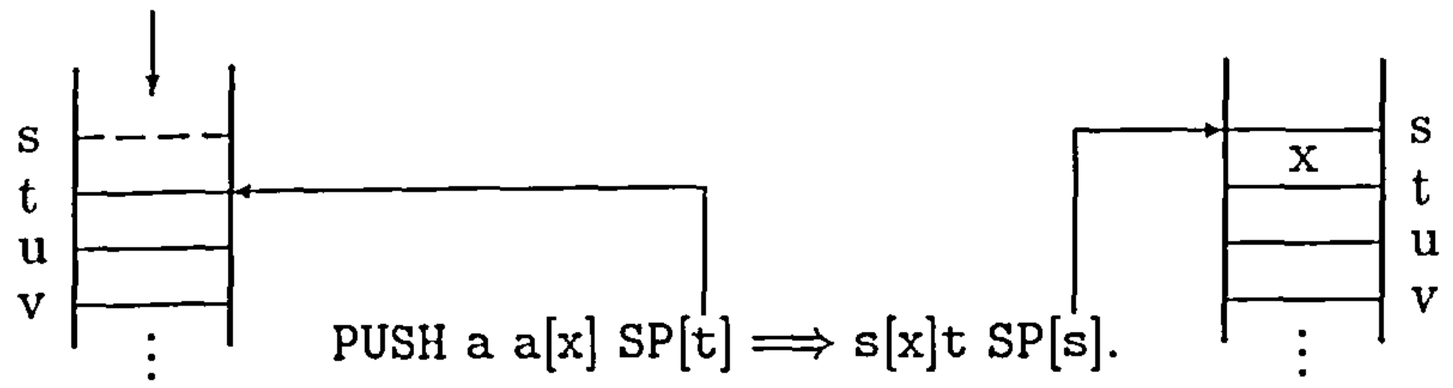


Figure 1.2: Visualisation of the PUSH operation on a stack machine

PC[pc] pc[*OP args*]qc *lc* ⫤ *g* ⊨ PC[pc′] pc′[*next*]qc *rc*.

may be written as so-called *commands*

OP args lc ⫤ *g* ⊨ *next rc*.

with the program counter hidden, where at least one of *OP args* and *next* is a non-empty *command sequence*, and *lc*/*rc* are left/right *contexts*. The first term of a command will in most cases be a constant, known as the *opcode*. An update plan in which all update schemes are commands is know as a *command driven* plan. A configuration is in *command form* if it contains a non-empty command sequence, or one in which the contents of the register PC are not specified.

Example 2

The following two commands may be part of a stack machine.

PUSH a a[x] SP[t] ⟹ s[x]t SP[s].
POP a SP[s] s[x]t ⟹ a[x] SP[t].

They will be desugared as

PC[pc] pc[PUSH a]qc a[x] SP[t] ⟹ PC[qc] s[x]t SP[s].
PC[pc] pc[POP a]qc SP[s] s[x]t ⟹ PC[qc] a[x] SP[t].

A graphical description of the PUSH operation is in figure 1.2.

Alternatives

A set of update schemes with mutually exclusive guards often performs some form of case analysis. Such a set

$$
\begin{aligned}
&lhs_1 \dashv\{\ g_1\ \} \models rhs_1.\\
&lhs_2 \dashv\{\ \neg g_1 \wedge g_2\ \} \models rhs_2.\\
&\ \ \vdots\\
&lhs_n \dashv\{\ (\bigwedge_{i=1}^{n-1} \neg g_i) \wedge g_n\ \} \models rhs_n.
\end{aligned}
$$

can then be written as a series of n update schemes known as *alternatives*

$$lhs_1 =\{\ g_1\ \}\Rightarrow rhs_1;$$
$$lhs_2 =\{\ g_2\ \}\Rightarrow rhs_2;$$
$$\vdots$$
$$lhs_n =\{\ g_n\ \}\Rightarrow rhs_n.$$

The update schemes are divided by semicolons and only the first applicable update scheme, reading from top to bottom, will be applied. The following production rules are added to the basic Update Plans grammar.

⟨*item*⟩ → ⟨*alternatives*⟩ "."

⟨*alternatives*⟩ → { ⟨*update scheme*⟩ ";" }$^+$

2 Typing

The basic type in Update Plans is the locator. In fact all objects appearing in an update plan are considered to be locators.

New types may be defined by combining existing types using any of the standard operators of regular expressions. Such a declaration is said to define a *type alias*. Store names are lower case words with an initial upper case letter, and must, therefore, contain at least two letters, in order to ensure that store names can be distinguished from constants. Stores are declared by listing them between braces, e.g.

{`Bit`, `Bool`, `Int`, `Stack`, `Heap`}.

Each store name is said to be a *type primitive*. Type aliases are declared similarly

{`Num` = `Byte` | `Short` | `Long`}.

Type aliases may not be recursive, either directly or indirectly. This ensures that any type alias can be expressed as a regular expression containing only type primitives. Every object appearing in an update plan must be typed. For some objects this may be done implicitly. Each symbolic constant is considered to have its own unique type, again unless indicated otherwise. Other objects—expressions for which no type can be determined automatically—must have their type indicated. This can be done by means of a global declaration valid throughout an update plan, e.g. '`v :: Store1.`'. A global type definition can be overridden by *casting* a term within an update scheme with a different type, 'e.g. (`v :: Store2`)' such a cast determining only the type of the term to which it is applied. The syntax of type declarations is obtained by adding the following rules to the basic UP grammar

⟨*item*⟩ → ⟨*store declaration*⟩ "." | ⟨*type declaration*⟩ "."

⟨*store declaration*⟩ → "{" {⟨*store*⟩ ","}* "}"

⟨*type declaration*⟩ → {⟨*store*⟩ ","}$^+$ "::" ⟨*store structure*⟩

⟨*term*⟩ → "(" ⟨*term*⟩ "::" ⟨*store structure*⟩ ")"

⟨*store*⟩ →
⟨*store name*⟩ |
⟨*store name*⟩ "=" ⟨*store structure*⟩

A ⟨*store name*⟩ is a lower case word with a leading upper case letter, and a ⟨*store structure*⟩ is a regular expression over the set of store names. Some extensions have been added to the typing mechanism, and can be found in chapter 5 alongside an updated type grammar.

A very important concept related to typing and implementability of update plans is grounding. It is extensively described in [60], but the following terminology is provided for convenience as it will be used in this thesis.

A *ground* expression is one for which a unique variable-free expression can be derived, possibly by instantiation of variables with respect to the current configuration; a *semi-ground* expression is one for which a finite number of such expressions can be derived.

Example 3

This example assumes that the length of an object of the type `Bit` is 1.
Given the following update plan

```
x, y :: Bit.
xs, ys :: (Bit)*.
A[x]b[xs]C[ys]d ⟹ B[x y]c.
```

`A`, `B` and `C` are ground as they are constants; `b` and `c` are also ground ($b = A + 1, c = B + 2$); `d` is non-ground, since the length of `ys` cannot be established; `x` is ground, since both of its locators are ground and its value can be determined by a reference to a configuration. The 'sequence' `xs` is ground for $b \leq C \leq b + 1$, *semi-ground otherwise. Finally, both* `ys` and `y` are non-ground.

3 Archetypes

The expressive power of Update Plans is greatly increased by the use of a macro-like mechanism known as *archetypes*. An update plan often contains a set of similar update schemes sharing certain parts of their left/right-hand sides. Such patterns can be moved out of update schemes into an archetype definition, and replaced by archetype calls. This will almost certainly mean a reduction in the number of update schemes necessary for a description as duplicate update schemes will be omitted from the update plan.

The primary motivation for the introduction of the archetype mechanism was the complexity of many machines due to the number of addressing modes they use. In an update plan, addressing modes can easily be replaced by a single archetype call, thus making it possible to express many update schemes as one.

Archetypes can also be viewed as a mechanism of abstraction, where certain frequently used actions are be grouped into "modular updates". This view will be reinforced in the second part of the thesis, where the concept of sequential update schemes and archetypes is introduced.

3.1 Syntax

The basic Update Plans grammar is updated by the following production rules.

⟨*item*⟩ → ⟨*archetype definition*⟩

⟨*archetype definition*⟩ →
⟨*basic archetype definition*⟩ |
⟨*command archetype definition*⟩

⟨*basic archetype definition*⟩ → ⟨*basic declaration*⟩ ⟨*basic definition*⟩$^+$

⟨*basic declaration*⟩ → ⟨*basic archetype name*⟩ ⟨*parameters*⟩

⟨*basic definition*⟩ → "=" ⟨*basic body*⟩ "."

⟨*basic body*⟩ →
⟨*configuration*⟩ ⟨*guard*⟩ ⟨*configuration*⟩ |
⟨*repeat*⟩ ⟨*guard*⟩ ⟨*configuration*⟩ |
⟨*configuration*⟩ |
⟨*text*⟩ "|" ⟨*context*⟩ ⟨*guard*⟩ ⟨*context*⟩ |
⟨*text*⟩ "|" ⟨*repeat*⟩ ⟨*guard*⟩ ⟨*context*⟩ |
⟨*text*⟩ "|" ⟨*context*⟩

⟨*command archetype definition*⟩ →
⟨*command declaration*⟩ ⟨*command definition*⟩$^+$

⟨*command declaration*⟩ →
⟨*command archetype name*⟩ ⟨*parameters*⟩ ⟨*text*⟩

⟨*command definition*⟩ → "=" ⟨*command body*⟩ "."

⟨*command body*⟩ →
⟨*context*⟩ ⟨*guard*⟩ ⟨*context*⟩ |
⟨*repeat*⟩ ⟨*guard*⟩ ⟨*context*⟩ |
⟨*context*⟩

⟨*parameters*⟩ → "(" {⟨*term*⟩ "," }* ")"

A *basic archetype name* is an identifier, a *command archetype name* is a constant. The syntax of an archetype call is basically that of an archetype declaration. The only difference is when archetype calls are 'coupled' by an index.

⟨*term*⟩ → ⟨*archetype call*⟩

⟨*archetype call*⟩ → ⟨*archetype name*⟩ ⟨*index*⟩-*opt* ⟨*parameters*⟩

⟨*archetype name*⟩ → ⟨*basic archetype name*⟩ | ⟨*command archetype name*⟩

⟨*index*⟩ → ⟨*number*⟩ | "{" ⟨*number*⟩ "}"

Again, the archetype grammar has been adapted with minor amendments from [60].

3.2 Expansion

The archetype expansion mechanism consists of two stages, the *textual expansion* and the *parameter resolution* stage.

The parameter resolution phase is necessary due to the fact that parameters do not have to be variables—they can be complex expressions. The aim of parameter resolution is to derive semi-ground expressions for non-ground terms. Parameters are resolved by using a resolution set, to which equations are added as rewriting takes place. If at any point in this process a non-trivial equation is derived relating two semi-ground expressions in an update scheme, it is added to the scheme's guard. The method is discussed in great detail in [60].

An archetype's body consists of a left and a right-hand side *expansion* and *context*. The expansion is the *text* that actually replaces the archetype call, the left/right-hand side expansion replacing the call on the left/right-hand side. The contexts are simply added to the call's scheme, again the left/right-hand side context is added to the left/right-hand side of the scheme. Archetype variables conflicting with variables of the call's scheme will be renamed before expansion.

See, for example, an archetype `autoinc` in example 4a which could define an autoincrement addressing mode. The (output) parameter `v` obtains its value by means of instantiation against the current configuration, and the contents of `r` are updated.

Example 4a

$$\texttt{autoinc(v)} = \texttt{AUTOINC r r[b] b[v]c} \Longrightarrow \texttt{r[c]}.$$

In the example, the left-hand side expansion is `AUTOINC r` and the left-hand side context `r[b] b[v]c`.

Archetype calls occur in indexed pairs, with one element of the pair on the left-hand side of the update scheme or archetype definition in which it occurs, and the other on the right-hand side. The resolution mechanism is demonstrated in example 4b, which gives a possible expansion of $\texttt{ADD r autoinc}_1\texttt{(x) autoinc}_2\texttt{(y)} \Longrightarrow \texttt{autoinc}_1\texttt{(x) autoinc}_2\texttt{(y) r[x + y]}$.

Example 4b

ADD r AUTOINC r_1 AUTOINC r_2

$r_1[b_1]\ r_2[b_2]\ b_1[v_1]c_1\ b_2[v_2]c_2 \Longrightarrow r[v_1 + v_2]\ r_1[c_1]\ r_2[c_2]$.

As will be noted in the following section, syntactic sugar allows one of the elements of an archetype call pair to be omitted if the corresponding expansion is empty. The indices may then also be omitted, as they are superfluous. The update plan from the previous example can then be rewritten as ADD r autoinc(x) autoinc(y) $\Longrightarrow$ r[x + y].

Archetype calls can be recursive. However, there are several limitations as to where recursive and non-recursive archetype calls may appear in an update plan. For example, a guard may not contain an archetype call and a locator and an archetype parameter may not contain a call of a recursive archetype.

3.3 Syntactic sugar

The bodies of archetype definitions inherit syntactic sugar as described in section 1. Additionally, syntactic sugar also makes it possible to

1. omit the guard and right-hand side from the body of an archetype definition, if the right-hand side and the guard are both empty
2. replace irrelevant parameters by the "don't care" symbol '_'
3. share identical archetype declarations (see the grammar)

Other forms of syntactic sugar are left/right-handed, ambidextrous and command archetypes.

Left/Right-handed archetypes

If the expansions (text) of all right/left-hand sides of a given archetype in all of its definitions are empty then the right/left-hand side of that archetype may be omitted. If all right/left-hand side calls are omitted in an update plan, such an archetype is called a *left/right-handed* archetype.

Left/Right-handed archetypes further contribute to sweetening of an update plan by the omission of archetype call's indices which are rendered superfluous as all calls of the same name appear only on one (left/right) side of an update scheme.

Ambidextrous archetypes

A left/right-handed archetypes are restricted in that they may only be called on the left/right-hand side. An *ambidextrous archetype* is a context independent archetype, which can be called on both sides of an update scheme.

A definition of an ambidextrous archetype a is equivalent to a pair of left and right archetype definitions a_l and a_r

$a_l(params) = text\ lc =\!\!\{\ g\ \}\!\!\Rightarrow rc.$
$a_r(params) = lc =\!\!\{\ g\ \}\!\!\Rightarrow text\ rc.$

which can be 'compressed' into one ambidextrous archetype definition

$a(params) = text \mid lc =\!\!\{\ g\ \}\!\!\Rightarrow rc.$

One use of ambidextrous archetypes can be to describe repetitive computations occurring on both sides of an update scheme.

Example 5

Consider the following recursive ambidextrous archetype pad(x) and an update scheme containing a call of this archetype. The archetype expands x padding bytes PAD to align address a of the JMP instruction so that a begins at an address that is a multiple of 4 bytes.

$\texttt{pad(0)} = .$
$\texttt{pad(n)} = \texttt{PAD pad(n} - 1\texttt{)} \mid =\!\!\{\ \texttt{n} > 0\ \}\!\!\Rightarrow .$
$\texttt{PC[pc] pc[JMP pad(4} - \texttt{(pc} + 1\texttt{)} \ \%\ 4\texttt{) a]} \Longrightarrow \texttt{PC[a]}.$

The purpose of making the pad(n) archetype ambidextrous is that it can be called on both sides of an update scheme without the need to define the following $\texttt{pad}_{[lr]}\texttt{(n)}$ pair

$\texttt{pad}_l\texttt{(n)} = \texttt{PAD pad(n} - 1\texttt{)} =\!\!\{\ \texttt{n} > 0\ \}\!\!\Rightarrow .$ *# To be called on the lhs*
$\texttt{pad}_r\texttt{(n)} = \ =\!\!\{\ \texttt{n} > 0\ \}\!\!\Rightarrow \texttt{PAD pad(n} - 1\texttt{).}$ *# To be called on the rhs*

in which case the $\texttt{pad}_l\texttt{(n)}$ and $\texttt{pad}_r\texttt{(n)}$ archetypes would have to be called on the left/right-hand side of an update scheme respectively, depending on the side padding is needed.

Command archetypes

An ambidextrous archetype *a* whose text part begins with a constant can be further compressed by using this constant as a name of a *command archetype*, unless this constant has already been used for this purpose. The command archetype

$CONST(params)\ text = lc =\!\!\{\ g\ \}\!\!\Rightarrow rc.$

is the sugared version of

$a(params) = CONST\ text \mid lc =\!\!\{\ g\ \}\!\!\Rightarrow rc.$

with calls of *a* replaced throughout the update plan by calls of *CONST*. Note that the syntax of ambidextrous archetypes changed (see chapter 5) in Extended Update Plans to be more consistent with command archetypes.

4 Parallelism

Parallelism is inherent in Update Plans, as application of an update rule is atomic and many cells may be changed simultaneously as a result. However, relying purely on this mechanism may create update schemes of unmanageable length. The solution is to have a set of non-related update schemes, instantiations of which will be applied simultaneously in a *parallel block*. These instantiations will only be applied if the conditions for application are satisfied for every single update rule and their right-hand sides are mutually consistent. An alternative view of parallel blocks will be given in chapter 6.

Parallel blocks are delimited by the *open parallel block symbol*, '(||', and the *close parallel block symbol*, '||)'. The use of the double pipeline symbol '||' is encouraged to separate the individual update schemes in a parallel block. An m update scheme parallel block can then be written as

$$
\begin{array}{l}
(\!\| \; lhs_1 \mathrel{=\!\!\{} g_1 \mathrel{\}\!\!\Rightarrow} rhs_1. \\
\;\| \; lhs_2 \mathrel{=\!\!\{} g_2 \mathrel{\}\!\!\Rightarrow} rhs_2. \\
\qquad \vdots \\
\;\| \; lhs_m \mathrel{=\!\!\{} g_m \mathrel{\}\!\!\Rightarrow} rhs_m. \\
\;\|\!)
\end{array}
$$

or making use of typesetting possibilities as

$$
\begin{array}{l}
\| \; lhs_1 \mathrel{=\!\!\{} g_1 \mathrel{\}\!\!\Rightarrow} rhs_1. \\
\| \; lhs_2 \mathrel{=\!\!\{} g_2 \mathrel{\}\!\!\Rightarrow} rhs_2. \\
\quad \vdots \\
\| \; lhs_m \mathrel{=\!\!\{} g_m \mathrel{\}\!\!\Rightarrow} rhs_m.
\end{array}
$$

To accommodate for parallel blocks, the following production rules need to be added to the basic Update Plans grammar.

⟨*item*⟩ → ⟨*parallel block*⟩

⟨*parallel block*⟩ → "(||" (⟨*alternatives*⟩ ".")$^+$ "||)"

Note that in Extended Update Plans not only alternatives can be in the body of a parallel block as will be shown in the second part of this thesis.

5 Examples

5.1 ADTs

The use of Update Plans is not limited to specification of hardware architectures although this is their primary aim. As has already been shown in this chapter, Update Plans can also be used to describe algorithms. This section shows their use in the description and implementation of abstract data types (ADTs), in particular singly linked lists.

$$
\begin{array}{l}
\texttt{LIST[head] head[data}_1\ \texttt{item}_2\texttt{] item}_2\texttt{[data}_2\ \texttt{item}_3\texttt{] item}_3\texttt{[data}_3\ \texttt{NULL]}. \\
\texttt{is\_empty(list)} = \texttt{list[item]}\ (\texttt{item} = \texttt{NULL}) =\!\![\ \texttt{list} \neq \texttt{NULL}\ ]\!\Rightarrow . \\
\texttt{insert(item}_1\texttt{, item}_2\texttt{, data)} = =\!\![\ \texttt{item}_2 \neq \texttt{NULL}\ ]\!\Rightarrow \texttt{item}_2\texttt{[data item}_1\texttt{]}. \\
\texttt{insert(item}_1\texttt{, item}_2\texttt{, data)} = \texttt{item}_1\texttt{[\_ next]} =\!\![\ \texttt{item}_1 \neq \texttt{NULL} \wedge \texttt{item}_2 \neq \texttt{NULL}\ ]\!\Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{item}_1\texttt{[\_ item}_2\texttt{] item}_2\texttt{[data next]}. \\
\texttt{remove(item)} = \texttt{item}_0\texttt{[\_ item] item[\_ next]} =\!\![\ \texttt{item}_0 \neq \texttt{NULL} \wedge \texttt{item} \neq \texttt{NULL}\ ]\!\Rightarrow \\
\qquad\qquad\qquad\quad \texttt{item}_0\texttt{[\_ next]}. \\
\texttt{length(NULL)} = 0. \\
\texttt{length(item)} = \texttt{item[\_ next] length(next)}+1 =\!\![\ \texttt{item} \neq \texttt{NULL}\ ]\!\Rightarrow .
\end{array}
$$

Figure 1.3: ADT list operations

An item of a list can be described as $\texttt{item[data item}_f\texttt{]}$. It consists of a payload `data` left of locator `item` and locator $\texttt{item}_f$ addressing a following item. To make a list, items can be simply linked in a list of items, and a reference to its head will be added as shown in figure 1.3.

The easiest operation on a list is to find about its emptiness. The archetype `is_empty` simply checks whether the head of a list points to the constant `NULL` and expands the appropriate truth value.

Two versions of `insert` item operation are provided. The difference is only in the placement of $\texttt{item}_1$. Note that an additional update scheme inserting an item at the start of the list would be required to complete the ADT if a list without a dummy head and the second version of `insert` archetype was to be used.

Similarly to the the second `insert` item archetype, the `remove` item archetype doesn't take into account removal of the first item in a list of linked items. Thus a similar function now removing the first item in a list would be required if a list without a dummy head was to be used.

Finally, the recursive `length` archetype calculates length of a list of items. The remaining ADT operations such as "retrieve n-th item" can be defined in a similar fashion.

5.2 Archetype expansion

The following example uses archetypes and an update scheme from chapter 2 to illustrate the archetype expansion mechanism. In this example the update scheme

$$[2.2]\ \texttt{arithm(x, r}_1\texttt{) dop(a, x)} \Longrightarrow \texttt{a[r}_1\texttt{] cc(r}_1\texttt{)}.$$

will be expanded into one update scheme showing the effects of the `INC` instruction in direct autodecrement mode from the PDP-11 instruction set.

The command scheme 2.2 expands as shown in figure 1.4. The archetypes and their index numbers from chapter 2 are included in figure 1.4 as comments for convenience. After substitution using equations derived at each step the full expansion of our update scheme is

$\texttt{arithm}(\texttt{x}, \texttt{r}_1)\ \texttt{dop}(\texttt{a}, \texttt{x}) \Longrightarrow \texttt{a}[\texttt{r}_1]\ \texttt{cc}(\texttt{r}_1)$.
\# *[2.2]* $arithm(x, x+1) = INC$. $x = x, r_1 = x + 1$

$\texttt{INC dop}(\texttt{a}, \texttt{x}) \Longrightarrow \texttt{a}[\texttt{r}_1]\ \texttt{cc}(\texttt{r}_1)$.
\# *[1.4]* $dop(a, v_1) = AUTODEC(a, v_1)$. $a = a, x = v_1$

$\texttt{INC AUTODEC}(\texttt{a}, \texttt{v}_1) \Longrightarrow \texttt{a}[\texttt{r}_1]\ \texttt{cc}(\texttt{r}_1)$.
\# *[1.1]* $AUTODEC(a, v_1)\ \ r = r[b]\ \ a[v_1]b \Longrightarrow r[a]$. $a = a, v_1 = v_1$

$\texttt{INC AUTODEC r r[b] a}[\texttt{v}_1]\texttt{b} \Longrightarrow \texttt{r[a] a}[\texttt{r}_1]\ \texttt{cc}(\texttt{r}_1)$.
\# *[2.1]* $cc(v_2) = CC_N[(v_2 <_s 0)\ \ (v_2 = 0)\ \ (\neg(MIN_s \leq_s v_2 \leq_s MAX_s))$
\# $(v_2 >_u MAX_u)]\ |.$ $r_1 = v_2$

$\texttt{INC AUTODEC r r[b] a}[\texttt{v}_1]\texttt{b} \Longrightarrow \texttt{r[a] a}[\texttt{r}_1]\ \texttt{CC}_\texttt{N}[(\texttt{v}_2 <_\texttt{s} 0)\ (\texttt{v}_2 = 0)$
$(\neg(\texttt{MIN}_\texttt{s} \leq_\texttt{s} \texttt{v}_2 \leq_\texttt{s} \texttt{MAX}_\texttt{s}))\ (\texttt{v}_2 >_\texttt{u} \texttt{MAX}_\texttt{u})]$.

Figure 1.4: Example of archetype expansion

$\texttt{INC AUTODEC r r[b] a[x]b} \Longrightarrow \texttt{r[a] a}[\texttt{x} + 1]$
$\texttt{CC}_\texttt{N}[((\texttt{x} + 1) <_\texttt{s} 0)\ ((\texttt{x} + 1) = 0)\ (\neg(\texttt{MIN}_\texttt{s} \leq_\texttt{s} (\texttt{x} + 1) \leq_\texttt{s} \texttt{MAX}_\texttt{s}))\ ((\texttt{x} + 1) >_\texttt{u} \texttt{MAX}_\texttt{u})]$.

A slightly more complex example of archetype expansion can also be found in chapter 3.

Chapter 2

PDP-11

The PDP-11 is one of the most widely known and used machines in the history of computing science, and its importance in mission-critical applications can hardly be questioned. Although one might argue the relevance of a specification of a historically dead machine (although sales of hardware products for the PDP-11 continued until as recently as September 1997), the very fact of its historical importance serves as a counterargument.

The instruction set of the PDP-11 was designed to provide a clean, general, symmetric instruction set. Word length is 16 bits with the leftmost, most significant bit being bit 15. There are eight general registers of 16 bits each. Register 7 is the program counter (PC) and, by convention, register 6 is the stack pointer (SP). There is also a Processor Status Register/Word (PSW) which among other things indicates the 4 condition code bits (N, Z, V, C).

In this chapter, a formal specification (based on [21, 23, 69]) of the DEC PDP-11 machine instruction set is given using the Update Plans formalism. Although some formal specifications of the PDP-11 instruction set already exist [20, 69], this chapter gives a clear, compact, unambiguous [60], and easy to follow specification which can serve as a comparison to those already existing.

1 Addressing modes

Basic addressing modes of the PDP-11 machine are given in 1.1. They define two results—the effective address of the value and the value (v) itself. The following list describes PDP-11 addressing modes informally. "The register" refers to one of eight general purpose registers (r), identified by an instruction word. "The displacement" refers to a word in the memory stored after an instruction (d).

- **register.** Direct access to a general register. The content of the selected register is taken as the value.

- **autoincrement.** At the start of an instruction's execution, the register contains the address of the value, and after the value is accessed and the instruction is executed, the address is incremented.
- **autodecrement.** The register has been decremented, before the value is accessed at this new address.
- **index.** The content of the register is added to the displacement to produce the address of the value.
- **register deferred.** The address of the value is stored in the register.
- **autoincrement deferred.** The register contains a pointer to the address of the value. The pointer is automatically incremented after the value is retrieved.
- **autodecrement deferred.** The content of the register is first decremented and then it contains a pointer to the address of the value.
- **index deferred.** The displacement is added to the base address stored in the register. The result is a pointer to the address of the value.

```
[1.1] REG(r,v) r          = r[v].                          # register mode
      AUTOINC(b,v) r      = r[b] b[v]c ⟹ r[c].             # autoincrement mode
      AUTODEC(a,v) r      = r[b] a[v]b ⟹ r[a].             # autodecrement mode
      INDEX(b+d,v) r d    = r[b] b+d[v].                   # index mode
      REGDEF(b,v) r       = r[b] b[v].                     # register deferred mode
      AUTOINCDEF(b2,v) r  = r[b1] b1[b2]c1 b2[v]c2 ⟹ r[c1]. # autoincrement deferred mode
      AUTODECDEF(a2,v) r  = r[b1] a1[a2]b1 a2[v]b2 ⟹ r[a1]. # autodecrement deferred mode
      INDEXDEF(b2,v) r d  = r[b1] b1+d[b2] b2[v].          # index deferred mode
```

In addition to these basic addressing modes there are 4 other special PC addressing modes. These addressing modes (1.2) come into effect when referencing the PC (register 7). Again, they define (apart from the immediate mode) two results—the effective address of the value and the value itself. "The word" refers to the contents of the location following the instruction.

- **PC immediate.** The word is the resulting value itself.
- **PC absolute.** The word is interpreted an address of the value.
- **PC relative.** The value's address is calculated by adding the word to the PC.
- **PC relative deferred.** A pointer to the value's address is calculated by adding the word to the PC.

```
[1.2] IMM(v) v          = .                          # immediate mode
      ABS(a,v) a        = a[v].                      # absolute mode
      REL(pc+d,v) d     = PC[pc] pc+d[v].            # relative mode
      RELDEF(a,v) d     = PC[pc] pc+d[a] a[v].       # relative deferred mode
```

Archetypes in 1.3 and 1.4 define two classes of addressing modes—the source (sop), and the destination (dop) operand.

[1.3] sop(v) = REG(_, v).
= AUTOINC(_, v).
= AUTODEC(_, v).
= INDEX(_, v).
= REGDEF(_, v).
= AUTOINCDEF(_, v).
= AUTODECDEF(_, v).
= INDEXDEF(_, v).
= IMM(_, v).
= ABS(_, v).
= REL(_, v).
= RELDEF(_, v).

[1.4] dop(a, v) = REG(a, v).
= AUTOINC(a, v).
= AUTODEC(a, v).
= INDEX(a, v).
= REGDEF(a, v).
= AUTOINCDEF(a, v).
= AUTODECDEF(a, v).
= INDEXDEF(a, v).
= ABS(a, v).
= REL(a, v).
= RELDEF(a, v).

2 Instructions

2.1 Single operand instructions

In order to make the Update Plans specification closer to the real implementation, the following data types are defined. Promotions and conversions between these individual data types are considered to be defined elsewhere.

{Nibble = Bit Bit Bit Bit, Byte = Nibble Nibble, Word = Byte Byte}.

The following text shows the layout of condition codes in the PSW register. We make use of the $CC_{[NZVC]}$ locators later on in this specification. The registers $CC_{[NZVC]}$ carry the negative, zero, overflow and carry bits respectively.

b_n, b_z, b_v, b_c :: Bit.
nibble :: Nibble.
byte :: Byte.
$PSW[byte\ nibble]CC_N[b_n]CC_Z[b_z]CC_V[b_v]CC_C[b_c]$.

Archetype 2.1 sets the condition codes according to the value of the parameter v.

[2.1] $cc(v) = CC_N[(v <_s 0)\ (v = 0)\ (\neg(MIN_s \leq_s v \leq_s MAX_s))\ (v >_u MAX_u)]$ |.

The constants MIN_s/MAX_s are the smallest/largest negative 16-bit two's complement signed integer (-32768/32767). MAX_u is the largest 16-bit unsigned integer (65535). The operators $<_s$, $\leq_s$ (signed comparison operators), $>_u$ (unsigned comparison operator) and = are assumed to be defined elsewhere.

It is assumed that the standard arithmetic operators are already defined. The arithmetic instructions are then defined as archetypes.

[2.2] $\texttt{arithm}(\_, 0) = \texttt{CLR}.$ # *clear destination*
$\texttt{arithm}(x, \neg x) = \texttt{COM}.$ # *complement destination*
$\texttt{arithm}(x, x+1) = \texttt{INC}.$ # *increment destination*
$\texttt{arithm}(x, x-1) = \texttt{DEC}.$ # *decrement destination*
$\texttt{arithm}(x, -x) = \texttt{NEG}.$ # *negate destination*
$\texttt{arithm}(x, x+c) = \texttt{ADC CC}_C[c].$ # *add carry to destination*
$\texttt{arithm}(x, x-c) = \texttt{SBC CC}_C[c].$ # *subtract carry from destination*
$\texttt{arithm}(x, x/2) = \texttt{ASR}.$ # *arithmetic shift right destination*
$\texttt{arithm}(x, x \times 2) = \texttt{ASL}.$ # *arithmetic shift left destination*
$\texttt{arithm}(\_, -1 \times n) = \texttt{STX CC}_N[n].$ # *sign extend destination*
$\texttt{arithm}(x, r)\ \texttt{dop}(a, x) \Longrightarrow a[r]\ \texttt{cc}(r).$

This specification covers mainly integer operand instructions. The byte operand instructions could easily be defined. For example SWAB is defined by

[2.3] $v_1, v_0 :: \texttt{Byte}.$
$\texttt{SWAB dop}(a, \_) = a[v_1\ v_0] \Longrightarrow a[v_0\ v_1].$ # *swap bytes of destination*

The specification of INCB is slightly more complicated. Here is an example of how the specification would change if we were to specify the instruction set including the byte operand instructions.

[2.4] $\texttt{arithm}(x, x+1) = \texttt{INC} \Longrightarrow \texttt{W}.$ # *increment destination*
$\texttt{arithm}(x, x+1) = \texttt{INCB} \Longrightarrow \texttt{B}.$ # *increment destination byte*
$\texttt{arithm}(x, r)\ \texttt{dop}(a, x) \Longrightarrow \texttt{rcc}(a, r, \texttt{arithm}(\_, \_)).$

The other byte/word operand pairs can be defined similarly. The constants W and B are used to pass typing information to the new condition code archetype rcc. Note that the left-hand side call of arithm occurs in the text part of the left-hand side, while the right-hand side call is a parameter of the rcc archetype so that, e.g. the INC in update scheme 2.4 will appear in the text of the expansion, while the constant W will be passed to rcc on the right-hand side. A small example is shown in figure 2.1. As this is a simple example (of a partial expansion), parameters are resolved every time an archetype is expanded. Some extra archetypes, which are used for byte operand instructions are shown in 2.5.

[2.5] $\texttt{type}(W) = \texttt{Word}\ |.$
$\texttt{type}(B) = \texttt{Byte}\ |.$
$\texttt{rcc}(a, v, w) = \texttt{cc}(v, w)\ | \Longrightarrow a[(v :: \texttt{type}(w))].$
$\texttt{cc}(v, W) = \texttt{CC}_N[(v <_s 0)\ (v = 0)\ (\neg(\texttt{MIN}_s \leq_s v \leq_s \texttt{MAX}_s))\ (v >_u \texttt{MAX}_u)]\ |.$
$\texttt{cc}(v, B) = \texttt{CC}_N[(v <_{bs} 0)\ (v = 0)\ (\neg(\texttt{MIN\_B}_s \leq_{bs} v \leq_{bs} \texttt{MAX\_B}_s))\ (v >_{bu} \texttt{MAX\_B}_u)]\ |.$

$\texttt{arithm(x,r) dop(a,x)} \Longrightarrow \texttt{rcc(a,r,arithm(\_,\_))}$. # *[2.4]*
 # [2.4] $\textit{arithm}(x, x+1) = \textit{INCB} \Longrightarrow B$. $x = x, r = x+1$

$\texttt{INCB dop(a,x)} \Longrightarrow \texttt{rcc(a,x+1,B)}$.
 # [2.5] $\texttt{rcc(a,v,w)} = \texttt{cc(v,w)} \mid \Longrightarrow \texttt{a[(v :: type(w))]}$. $a = a, x+1 = v, B = w$

$\texttt{INCB dop(a,x)} \Longrightarrow \texttt{cc(x+1,B) a[(x+1 :: type(B))]}$.
 # [2.5] $cc(v,B) = CC_N[(v <_{bs} 0)\ (v = 0)\ (\neg(MIN\_B_s \leq_{bs} v \leq_{bs} MAX\_B_s))$
$(v >_{bu} MAX\_B_u)] \mid$. $x+1 = v$

$\texttt{INCB dop(a,x)} \Longrightarrow \texttt{CC}_\texttt{N}\texttt{[(x+1 <}_\texttt{bs}\texttt{ 0) (x+1 = 0) (}\neg\texttt{(MIN\_B}_\texttt{s}\texttt{ <}_\texttt{bs}\texttt{ x+1 <}_\texttt{bs}\texttt{ MAX\_B}_\texttt{s}\texttt{))}$
$\texttt{(x+1 >}_\texttt{bu}\texttt{ MAX\_B}_\texttt{u}\texttt{)] a[(x+1 :: type(B))]}$.

Figure 2.1: An example of archetype expansion

Again, the byte constants $\texttt{MIN\_B}_\texttt{s}$, $\texttt{MAX\_B}_\texttt{s}$, $\texttt{MAX\_B}_\texttt{u}$, and the byte operators $<_{bs}$, $\leq_{bs}$, $>_{bu}$ have similar meaning to their integer (word) counterparts, and are assumed to be defined elsewhere.

[2.6] $\texttt{TST dop(\_,x)} \Longrightarrow \texttt{cc(x)}$. # *test destination*
$\texttt{ROR dop(a,x) CC}_\texttt{C}\texttt{[c]} \Longrightarrow \texttt{a[(x>>1)|(c<<15)] CC}_\texttt{C}\texttt{[(x\&1 :: Bit)]}$.
$\texttt{ROL dop(a,x) CC}_\texttt{C}\texttt{[c]} \Longrightarrow \texttt{a[(x<<1)|c] CC}_\texttt{C}\texttt{[(x>>15)\&1 :: Bit)]}$.

The TST instruction sets the condition codes according to the contents of the destination word.

ROR/ROL: the PSW C-bit and the destination word (considered as a 17-bit "word" is rotated right/left one bit. The vacated high/low-order bit (bit 15/0) is loaded with the contents of the C-bit. The low/high-order bit of the destination (bit 0/15) is loaded into the C-bit.

2.2 Double operand instructions

The MOV instruction moves a copy of the source word contents to the destination. Its command archetype definition is given in 2.7.

[2.7] $\texttt{MOV sop(v) dop(a,\_)} \Longrightarrow \texttt{a[v]}$.

The following arithmetic and bit-wise instructions perform the specified operations, and set the destination operand and the condition codes in accordance with the result.

[2.8] $\texttt{abrithm(x,y,x+y)} = \texttt{ADD}$. # *add source to destination*
$\texttt{abrithm(x,y,x-y)} = \texttt{SUB}$. # *subtract source from destination*
$\texttt{abrithm(x,y,}\neg\texttt{x\&y)} = \texttt{BIC}$. # *bit clear destination from source*
$\texttt{abrithm(x,y,x|y)} = \texttt{BIS}$. # *bit set destination from source*
$\texttt{abrithm(x,y,r) sop(x) dop(a,y)} \Longrightarrow \texttt{a[r] cc(r)}$.

The comparison instructions are similar to the above instructions, however, the destination operand is not affected.

[2.9] cmps(x, y, x − y) = CMP. # *compare source to destination*
cmps(x, y, x&y) = BIT. # *bit test source and destination bytes*
cmps(x, y, r) sop(x) dop(_, y) ⟹ cc(r).

2.3 Condition code and program flow operations

The condition code operations SE*c* and CL*c*, where *c* is any of the condition codes N, Z, V, and C are shown in 2.10. The SE*c* operations set the condition codes, and CL*c* clear the condition code. TRUE and FALSE are predefined constants of the type Bit, values of which are 1 and 0 respectively.

[2.10] SEN ⟹ CC_N[TRUE]. CLN ⟹ CC_N[FALSE].
SEZ ⟹ CC_Z[TRUE]. CLZ ⟹ CC_Z[FALSE].
SEV ⟹ CC_V[TRUE]. CLV ⟹ CC_V[FALSE].
SEC ⟹ CC_C[TRUE]. CLC ⟹ CC_C[FALSE].
SCC ⟹ CC_N[TRUE] CC_Z[TRUE] CC_V[TRUE] CC_C[TRUE].
CCC ⟹ CC_N[FALSE] CC_Z[FALSE] CC_V[FALSE] CC_C[FALSE].

The full suite of program flow instructions is given in the following definitions. First, the conditional branches are given (2.11), and then the unconditional jump is defined in 2.12.

[2.11]
branch(true) = BR. # *branch always*
branch(¬z) = BNE CC_Z[z]. # ≠ 0
branch(z) = BEQ CC_Z[z]. # = 0
branch(¬(n ^ v)) = BGE CC_N[n] CC_V[v]. # ≥ 0
branch(n ^ v) = BLT CC_N[n] CC_V[v]. # < 0
branch(¬(z|(n ^ v))) = BGT CC_Z[z] CC_N[n] CC_V[v]. # > 0
branch(z|(n ^ v)) = BLE CC_Z[z] CC_N[n] CC_V[v]. # ≤ 0
branch(¬n) = BPL CC_N[n]. # +
branch(n) = BMI CC_N[n]. # −
branch(¬(c|z)) = BHI CC_C[c] CC_Z[z]. # *higher (unsigned comparison)*
branch(¬c) = BCC CC_C[c]. # *carry clear*
= BHIS CC_C[c]. # *higher or same (unsigned compar.)*
branch(c) = BCS CC_C[c]. # *carry set*
= BLO CC_C[c]. # *lower (unsigned comparison)*
branch(c|z) = BLOS CC_C[c] CC_Z[z]. # *lower or same (unsigned compar.)*
branch(¬v) = BVC CC_V[v]. # *overflow clear*
branch(v) = BVS CC_V[v]. # *overflow set*
PC[pc] pc[branch(c) d]cp =[c]⇒ PC[cp + d]; # *branch*
" ⟹ PC[cp]. # *no branch*

All branch instructions jump relative to the contents of the program counter based on condition codes. The displacement d is contained in the instruction word. The target absolute address of the jump instruction defined by the following command is dependent on an addressing mode encoded in the instruction word.

[2.12] JMP dop(_, a) $\Longrightarrow$ PC[a].

The following register addressing archetype is defined for convenience. Note that use can not be made of the register addressing command archetype defined in 1.1, since it expands the additional REG text.

[2.13] reg(r, v) = r[v] |.

JSR calculates the destination address (a), saves the contents (v) of the source register (r) on the hardware stack, and saves the address of the following instruction in the source register. (This saving of the PC provides the linkage from the subroutine back to the calling program.) Finally, the PC is given the destination address, which produces the actual jump to the subroutine. Return to the calling routine is provided by the companion instruction RTS. The contents of the specified register are loaded into the PC, and the top of the hardware stack is popped into the register. Note that register 6 is by convention the stack pointer.

[2.14] PC[pc] pc[JSR r dop(a, _)]qc reg(r, v) reg(6, tp)
$\Longrightarrow$ PC[a] reg(6, sp) sp[v]tp reg(r, qc).
RTS r reg(r, pc) reg(6, sp) sp[v]tp $\Longrightarrow$ PC[pc] reg(6, tp) reg(r, v).

2.4 Interrupts

Software interrupt instructions perform a trap through the vector at (fixed) memory locations. The effect of all of these instructions is the same: stack the contents of the PC and PSW (program status word), and then load the PC and PSW with the contents of a fixed memory location. The following definitions specify interrupt traps.

[2.15] trap(14_8, 16_8) = BPT. # *breakpoint trap*
trap(20_8, 22_8) = IOT. # *input/output trap*
trap(30_8, 32_8) = EMT. # *emulator trap*
trap(34_8, 36_8) = TRAP. # *TRAP*
trap(pc_a, psw_a) reg(6, tp) pc_a[pc] psw_a[psw] PC[pc_s] PSW[psw_s]
$\Longrightarrow$ sp[pc_s psw_s]tp reg(6, sp) PC[pc] PSW[psw].

Returns from interrupts, on the other hand, restore the original contents (just before an interrupt) of the PC and PSW registers from stack.

[2.16] ret() = RTI. # *return from interrupt*
= RTT. # *return from trap*
ret() reg(6, sp) sp[pc psw]tp $\Longrightarrow$ PC[pc] PSW[psw] reg(6, tp).

2.5 Other instructions

The NOP instruction is the standard *no operation* instruction. It produces no effect, but occupies one word of memory and requires a complete execution cycle for execution.

[2.17] NOP $\Longrightarrow$. # *no operation*

3 Conclusions

A significant subset of the PDP-11 instruction set has been described using the Update Plans formalism. Although this specification adheres strictly to the Update Plans formalism as described in [60], it already raises several issues addressed in the second part of this thesis. In particular, it prompted the development of sequential update schemes and archetypes as explained in chapter 6. A new, more complex specification would be based on the concepts described in the same chapter, possibly on a fetch/execute cycle level description. This specification is rather an abstract one. Use of the syntactic extensions described in the second part of the thesis could be made, together with additional typing information to define the precise layout of instruction words, as is done, for example, in the following chapter.

Chapter 3

SPARC-V9

The SPARC[1] has been implemented in processors used in a range of computers from laptops to supercomputers, and today boasts over 8,000 software application programs. SPARC-V9, like its predecessor SPARC-V8, is a modern microprocessor specification created by the SPARC Architecture Committee of SPARC International. SPARC-V9 is not a specific chip. It is an architectural specification that can be implemented as a microprocessor by those obtaining a license from SPARC International.

In this chapter, a formal specification of the SPARC-V9 abstract machine instruction set is given using the Update Plans formalism. Although a partial specification of the SPARC-V9 architecture already exists [50], this is aimed at a different level of description.

The structure of this chapter is as follows. Section 1 defines types and constants used throughout the specification. The SPARC-V9 has a rich set of registers using a relatively new concept of windowing which is both informally described and then specified using Update Plans in section 2. The most important instructions of this architecture are specified in section 3. An example taken from the specification is shown in section 4. Finally, conclusions and some future research suggestions are given section 5.

1 Types and constants

Throughout the specification, the following types are used. The remaining types used in this specification which are not displayed here are declared in sections of their immediate use. `Bit` is considered to be a primitive type having the usual meaning.

$\{\texttt{Bit}\}.$

$\{\texttt{Byte}(01_2) = \{\texttt{Bit}\}8, \texttt{Halfword}(10_2) = \{\texttt{Bit}\}16, \texttt{Word}(00_2) = \{\texttt{Bit}\}32,$ 5:3.1, 3.2

$\texttt{Extended\_word} = \{\texttt{Bit}\}64\}.$ 5:3.2

$\{\texttt{Fp\_single} = \{\texttt{Bit}\}32, \texttt{Fp\_double} = \{\texttt{Bit}\}64, \texttt{Fp\_quad} = \{\texttt{Bit}\}128\}.$ 5:3.2

[1]Scalable Processor ARChitecture

In Update Plans, constants are uppercase words. However, as will be shown in chapter 5 any sequence of letters (store) can be made into a constant by assigning it a type `Constant`. The main reason for listing these constants is to show their bit values, supply grounding information where necessary, and use names easily identifiable against the official (informal) description of the SPARC-V9 architecture [77].

`nPC :: Constant.` 5:3.1

$\texttt{IMM13}(1_2)$ :: `Bit`. 5:3.1

$\texttt{OP:ARM}(10_2), \texttt{OP:LS}(11_2), \texttt{OP:BRS}(00_2), \texttt{OP:CLL}(01_2)$ :: {`Bit`}2. 5:3.1, 3.2

$\texttt{BRZ}(01_2), \texttt{BRLEZ}(10_2), \texttt{BRLZ}(11_2)$ :: {`Bit`}2. 5:3.1, 3.2

$\texttt{ADD}(000_2), \texttt{SUB}(100_2), \texttt{SAVE}(100_2), \texttt{RESTORE}(101_2)$ :: {`Bit`}3. 5:3.1, 3.2

$\texttt{BPA}(1000_2), \texttt{BPN}(0000_2), \texttt{BPNE}(1001_2), \texttt{BPE}(0001_2)$ :: {`Bit`}4. 5:3.1, 3.2

$\texttt{LDSTUB}(00\,11\,01_2)$ :: {`Bit`}6. 5:3.1, 3.2

$\texttt{FADD}(0\,01\,00\,00_2), \texttt{FSUB}(0\,01\,00\,01_2)$ :: {`Bit`}7. 5:3.1, 3.2

$\texttt{REGRES}(0\,00\,00\,00\,00_2)$ :: {`Bit`}9. 5:3.1, 3.2

2 Registers

A SPARC-V9 processor includes two types of registers: general-purpose, or working data registers, and control/status registers. In the following sections only integer and floating point registers are described. For the structure of control/status registers, please refer to the informal specification [77].

2.1 General purpose `r` registers

An implementation of the instruction unit may contain anything from 64 to 528 general-purpose 64-bit `r` registers. At any time, an instruction can access the first 8 global registers (`r`[0–7]), and a 24-register window into the `r` registers. A register window comprises the 8 'in' and 8 'local' registers, together with the 8 'in' registers of an adjacent register set, which are addressable from the current window as 'out' registers. See figure 3.1 for more graphic explanation.

The following two sets of archetypes are provided to facilitate reading (2.1) and writing (2.2) of `r` registers, and expansion of a 5-bit (`a` :: {`Bit`}5) register address in an instruction word. 5:3.2

[2.1]
$$\begin{aligned}
\texttt{rr}(0,0) &= . && && \#\ \textit{reading } \texttt{r[0]} \textit{ yields 0}\\
\texttt{rr}(a,v) &= \texttt{a}[v] && \models\!\!\!\mid\ 1 \le a < 8 \ \mid\!\!\!\models . && \#\ \textit{global registers}\\
&= \texttt{CWP}[w]\ \texttt{outr}(a-8,v,w) && \models\!\!\!\mid\ 8 \le a < 16 \ \mid\!\!\!\models .\\
&= \texttt{CWP}[w]\ \texttt{locr}(a-16,v,w) && \models\!\!\!\mid\ 16 \le a < 24 \ \mid\!\!\!\models .\\
&= \texttt{CWP}[w]\ \texttt{inr}(a-24,v,w) && \models\!\!\!\mid\ 24 \le a < 32 \ \mid\!\!\!\models .
\end{aligned}$$

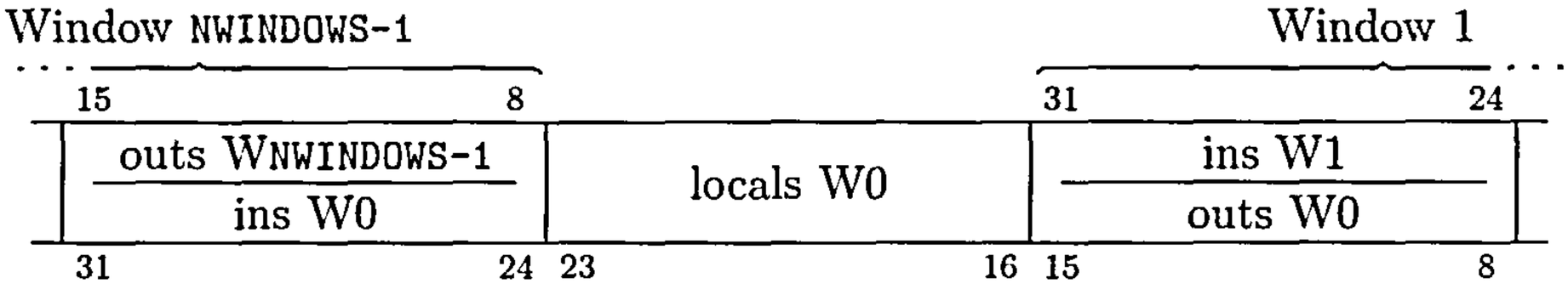


Figure 3.1: Register Window 0

CWP is a constant locator pointing to the left boundary of the CWP (Current Window Pointer) register containing a 5-bit ($w :: \{Bit\}5$) address of the current 24 r register window. 5:3.2

A typical use of archetypes rr is through archetypes sop1 and sop2 (3.1) both of which will be described in section 3.1.

[2.2] $rw(a, v)\ a = CWP[w]\ rw(a, v, w).$ 5:2.2

$$\begin{aligned} rw(a, v, w) &= =\{\ 0 = a \}\Rightarrow . && \#\ \textit{writing r[0] has no effect} \\ &= =\{\ 1 \le a < 8 \}\Rightarrow a[v]. && \#\ \textit{global registers} \\ &= =\{\ 8 \le a < 16 \}\Rightarrow outw(a-8, v, w). \\ &= =\{\ 16 \le a < 24 \}\Rightarrow locw(a-16, v, w). \\ &= =\{\ 24 \le a < 32 \}\Rightarrow inw(a-24, v, w). \end{aligned}$$

Global register zero r[0] always reads as zero (archetype 2.1), and writes to it have no program-visible effect (archetype 2.2).

[2.3]
$$\begin{aligned} outr(a, v, w) &= ilocr(a, v, w). \\ outw(a, v, w) &= \Longrightarrow ilocw(a, v, w). \\ locr(a, v, w) &= ilocr(a+8, v, w). \\ locw(a, v, w) &= \Longrightarrow ilocw(a+8, v, w). \\ inr(a, v, w) &= ilocr(a, v, w-1). \\ inw(a, v, w) &= \Longrightarrow ilocw(a, v, w-1). \end{aligned}$$

Archetypes outr/outw, locr/locw and inr/inw (used to read/write access 'out', 'local' and 'in' registers) are defined in terms of archetypes ilocr/ilocw (2.4) which calculate left locators for these 3 types of r registers.

[2.4]
$$\begin{aligned} ilocr(a, v, w) &= (NWINDOWS - 1 - (w \,\%\, NWINDOWS)) \times 16 + a[v]. \\ ilocw(a, v, w) &= \Longrightarrow (NWINDOWS - 1 - (w \,\%\, NWINDOWS)) \times 16 + a[v]. \end{aligned}$$

The number of windows or register sets, NWINDOWS, is implementation-dependent and ranges from 3 to 32. Note that r register with address o, where $8 \le o < 16$, refers to exactly the same register as $o + 16$ does after the CWP is incremented by 1 (% NWINDOWS). Likewise, a register with address i, where $24 \le i < 32$, refers to exactly the same register as address $i - 16$ does after the CWP is decremented by 1 (% NWINDOWS).

2.2 Floating-point f registers

The floating-point unit contains

- 32 single-precision (32-bit) floating-point registers, numbered f[0], f[1], ..., f[31]
- 32 double-precision (64-bit) floating-point registers, numbered f[0], f[2], ..., f[62]
- 16 quad-precision (128-bit) floating-point registers, numbered f[0], f[4], ..., f[60]

The floating-point registers are arranged so that some of them overlap, that is, are aliased. Layout and numbering of the floating-point registers is apparent from the following archetypes.

[2.5] $\texttt{fr(a,v,s)}\ \texttt{a} = \texttt{a[(v :: Fp\_single)]} \ =\!\{\ \texttt{s} = 01_2\ \}\!\Rightarrow .$ 5:2.2

$= \texttt{a[(v :: Fp\_double)]} \ =\!\{\ \texttt{s} = 10_2\ \}\!\Rightarrow .$

$= \texttt{a[(v :: Fp\_quad)]} \ =\!\{\ \texttt{s} = 11_2\ \}\!\Rightarrow .$

$\texttt{fw(a,v,s)}\ \texttt{a} = =\!\{\ \texttt{s} = 01_2\ \}\!\Rightarrow \texttt{a[(v :: Fp\_single)]}.$ 5:2.2

$= =\!\{\ \texttt{s} = 10_2\ \}\!\Rightarrow \texttt{a[(v :: Fp\_double)]}.$

$= =\!\{\ \texttt{s} = 11_2\ \}\!\Rightarrow \texttt{a[(v :: Fp\_quad)]}.$

Archetypes (2.5) are provided to encapsulate reading (fr) and writing (fw) of different types of floating-point values into f registers. Similarly to r registers, address a of an f register is 5-bit wide (a :: {Bit}5). 5:3.2

Unlike the windowed r registers, all of the floating-point registers are accessible at any time.

3 Instructions

3.1 Arithmetic and logical operations

Arithmetic and logical instructions (and some others) use the instruction word format shown in detail in figure 3.2. The value (major opcode) of two highest bits of all instructions falling into this category is 10, for which a constant OP:ARM is used. Other fields in the instruction word are 5-bit addresses of source (rs1, rs2) and destination (rd) registers, minor opcode op3, and if bit 13 is set to 1, a 13-bit immediate value simm13 always sign-extended to 64 bits.

In order to make our specification as compact and clear as possible, a frequent use of the archetype mechanism is made. Firstly, we define archetypes for accessing values in source registers, which at the same time describe the layout of the instruction word.

[3.1] $\texttt{sop1(v)} = \texttt{rr(a,v)}\ \texttt{a}.$

$\texttt{sop2(v)} = \texttt{REGRES(v)}.$

$= \texttt{IMM13(v)}.$

Command archetypes REGRES and IMM13 help in defining the instruction word format and return a value of a source operand in variable v. The function sign_ext sign-extends an integer value and is assumed to be defined elsewhere.

[3.2] REGRES(v) a = rr(a, v). # *REGRES* = 0_2 $00\,00\,00\,00_2$ 5:2.2
IMM13(sign_ext(v)) = (v :: {Bit}13). # *IMM13* = 1_2 5:3.2

Secondly, there must be a way of reading and writing carry bit in the condition codes register. Archetypes 3.3 and 3.4 do exactly that. Note that arithmetic and logical instructions always read the carry bit from integer condition code register (ICC), whereas they write to both ICC and the extended integer condition code (XCC) registers.

[3.3] iccr(c) = CCR:ICC:C[c].
iccrb(c) = 1_2 iccr(c).
iccrb(0) = 0_2.

Archetype iccrb reads the carry flag to c, if the carry bit in the op3 field (bit 22) is set. Similarly, ccwb sets the condition code registers XCC and ICC (only) if the ccwb bit (23) in the instruction is 1.

[3.4] ccw(v) $\Longrightarrow$ CCR:XCC[(v $<_{4s}$ 0) (v = 0) ($\neg$(MIN$_{4s}$ $\leq_{4s}$ v $\leq_{4s}$ MAX$_{4s}$)) (v $>_{4u}$ MAX$_{4u}$)] 5:2.2
CCR:ICC[(v $<_{2s}$ 0) (v = 0) ($\neg$(MIN$_{2s}$ $\leq_{2s}$ v $\leq_{2s}$ MAX$_{2s}$)) (v $>_{2u}$ MAX$_{2u}$)].
ccwb(v) = 0_2.
= $1_2 \Longrightarrow$ ccw(v).

The constants MIN$_{ns}$/MAX$_{ns}$ are the smallest/largest negative n-byte two's complement signed integer. MAX$_{nu}$ is the largest n-byte unsigned integer. The n-byte operators $<_{ns}$, $\leq_{ns}$ (signed comparison operators), $>_{nu}$ (unsigned comparison) and = are assumed to be defined elsewhere.

[3.5] c :: Bit.
aloper(x, y, x + y + c) = 0_2 ccwb($3) iccrb(c) ADD. # *ADD* = 000_2 5:2.4
aloper(x, y, x − y − c) = 0_2 ccwb($3) iccrb(c) SUB. # *SUB* = 100_2 5:2.4

Archetype aloper (3.5) shows the behaviour of 8 different arithmetic instructions and defines the layout of their op3 field making use of the previously defined archetypes iccrb and ccwb. These instructions include add/subtract, add/subtract and modify condition codes, add/subtract with carry, and add/subtract with carry and modify condition codes. The $\$n$ symbol used in archetype bodies is used to reference the archetype's nth parameter. Thus the $3 of the first aloper archetype is merely shorthand for saying 'x + y + c'.

Since arithmetic and logical instructions fall into the same category as regards their instruction field layout, the following archetypes (and in particular the aloper archetype) are provided to specify both the op3 field and the appropriate actions for logical instructions.

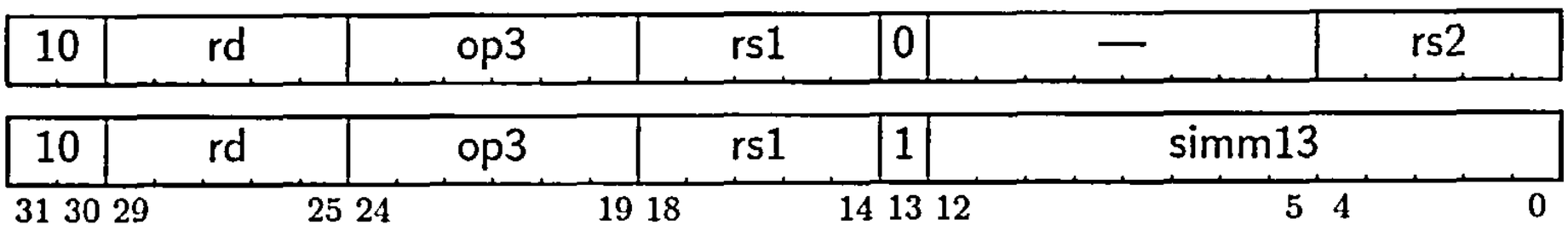


Figure 3.2: Format of arithmetic, logical, and SAVE and RESTORE instructions

[3.6] n :: Bit.
op :: {Bit}2. 5:3.2
neg(v) $= \neg =[\, v = 1_2 \,]\Rightarrow;$
$\Longrightarrow$. # *empty body if* v $\neq 1_2$
lop(op) = & $=[\, op = 01_2 \,]\Rightarrow$. # *bit-wise and*
| $=[\, op = 10_2 \,]\Rightarrow$. # *bit-wise or*
^ $=[\, op = 11_2 \,]\Rightarrow$. # *bit-wise xor*
aloper(x, y, neg(n) (x lop(op) y)) = 0_2 ccwb($3) 0_2 n op. 5:2.4

If the n bit is set it inverts (by the means of the neg archetype) the logical operation determined by the op field. Note that archetype neg expands to empty right and left-hand sides, if the value of the bit n is 0. The archetype lop simply decides on one of 3 logical operations based on value of its two-bit parameter op. Again, condition codes are set or not by the archetype ccwb based on the result of logical operations (third argument of archetype aloper) in the same way as for arithmetical operations.

Many update schemes in this specification are commands. As explained in [60], every command whose left and right program counters are hidden can be prefixed by '*pc*() =', where *pc*() is a unique archetype declaration. As there are two program counters in the SPARC-V9, PC—containing the address of the current instruction and nPC—containing the address of the next instruction, the following update scheme can be defined and all commands instantiated through the expansion in this scheme.

[3.7] PC[pc] pc[*pc*()]qc nPC[pc′] $\Longrightarrow$ PC[pc′] pc′[*pc*()]qc nPC[pc′ + 4].

Finally, the command update scheme 3.8 specifies the instruction field layout of all arithmetic and logical instructions defined in this section.

[3.8] OP:ARM rw(_, r) aloper(x, y, r) sop1(x) sop2(y) $\Longrightarrow$. # *OP:ARM* $= 10_2$

3.2 Register window manipulation instructions

All r registers (apart from the first eight) are windowed. This means that there must be a mechanism to access registers in various windows. There are two instructions that do that—SAVE and RESTORE.

The SAVE instruction provides the routine executing it with a new register window. The 'out' registers from the old window become the 'in' registers of the new window.

The RESTORE instruction restores the register window saved by the last SAVE instruction executed by the current process. The 'in' registers of the old window become the 'out' registers of the new window. The 'in' and 'local' registers in the new window contain the previous values.

[3.9] $\texttt{sr}(\texttt{x},\texttt{y},\texttt{x}+\texttt{y}+\texttt{c},\texttt{w}+1) =$
$1_0\ \texttt{ccwb}(\$3)\ \texttt{iccr}(\texttt{c})\ \texttt{SAVE}\ \texttt{CWP}[\texttt{w}] \Longrightarrow .$ $\quad \#\mathit{SAVE} = 100_2$ 5: 2.4
$\texttt{sr}(\texttt{x},\texttt{y},\texttt{x}+\texttt{y}+\texttt{c},\texttt{w}-1) =$
$1_0\ \texttt{ccwb}(\$3)\ \texttt{iccr}(\texttt{c})\ \texttt{RESTORE}\ \texttt{CWP}[\texttt{w}] \Longrightarrow .\ \#\mathit{RESTORE} = 101_2$ 5: 2.4
$\texttt{OP:ARM}\ \texttt{rw}(\texttt{a},\texttt{r},\texttt{w})\ \texttt{a}\ \texttt{sr}(\texttt{x},\texttt{y},\texttt{r},\texttt{w})\ \texttt{sop1}(\texttt{x})\ \texttt{sop2}(\texttt{y}) \Longrightarrow \texttt{CWP}[\texttt{w}\ \%\ \texttt{NWINDOWS}].$

Furthermore, SAVE and RESTORE behave like an ordinary ADD instruction, except that the source operands r[rs1] and/or r[rs2] are read from the old window (that is, the window addressed by the original CWP) and the sum is written into r[rd] of the new window (that is, the window addressed by the new CWP).

3.3 Load/Store instructions

3.3.1 Load-store unsigned byte

The load-store unsigned byte instruction copies a byte from memory into r[rd], and then rewrites the addressed byte in memory to all ones. The fetched byte is right-justified in the destination register r[rd] and zero-filled on the left. The format of this instruction is the same as the format of arithmetic instructions, with the exception of the major opcode.

[3.10] $\texttt{r} :: \texttt{Byte}.$
$\texttt{ldstub}(\texttt{a},\texttt{r}) = \texttt{LDSTUB}\ \texttt{a}[\texttt{r}] \Longrightarrow \texttt{a}[11111111_2].$ $\quad \#\mathit{LDSTUB} = 00\,11\,01_2$
$\texttt{OP:LS}\ \texttt{rw}(\_,\texttt{r})\ \texttt{ldstub}(\texttt{x}+\texttt{y},\texttt{r})\ \texttt{sop1}(\texttt{x})\ \texttt{sop2}(\texttt{y}) \Longrightarrow .\ \#\mathit{OP{:}LS} = 11_2$

3.3.2 Load integer from alternate space

The load integer from alternate space instructions copy a byte, a halfword, a word or an extended word from memory into r[rd]. A fetched byte, halfword, or word is right-justified in the destination register; it is either sign-extended or zero-filled on the left (to 64 bits), depending on whether the opcode specifies a signed or unsigned operation, respectively.

For each instruction access and each normal data access, the IU appends an 8-bit address space identifier (ASI) to the 64-bit memory address. Load/Store alternate instructions can provide an arbitrary ASI with their data addresses (if bit 13 of the instruction word = 0), or use the ASI value contained in the ASI register (if bit 13 of the instruction word = 1).

Archetypes defined in 3.11, 3.13 and the type alias declaration in 3.12 are provided in order to make the final command update scheme in 3.14 more compact.

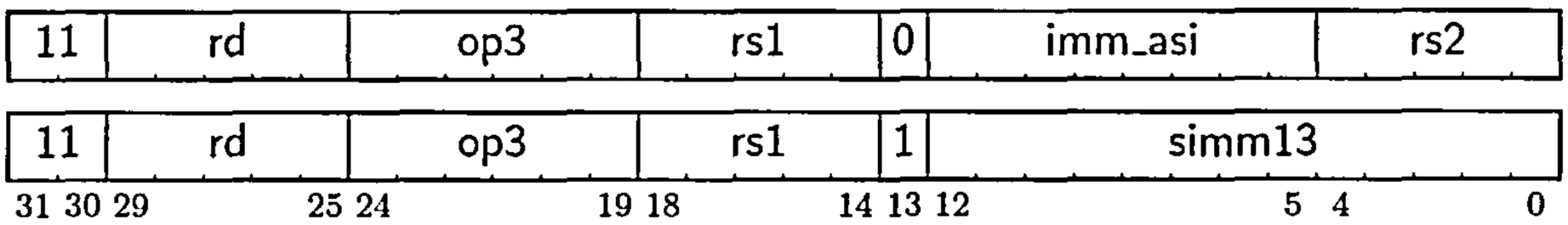


Figure 3.3: Format of load/store integer or from/into alternate space instructions

[3.11] $\texttt{signed(s)} = \texttt{zero\_fill} \dashv\{\ \texttt{s} = 0\ \}\vdash .$
$= \texttt{sign\_ext} \dashv\{\ \texttt{s} = 1\ \}\vdash .$

[3.12] $\{\texttt{Integer} = \texttt{Byte} \mid \texttt{Halfword} \mid \texttt{Word}\}.$ 5:3.1

The archetype 3.13 makes sure that we get the correct value of ASI regardless of the value of bit 13 in the instruction field, fetches values x and y from source registers, and sets the format of the instruction word for bits 0–18.

[3.13] $\texttt{asi} :: \texttt{Byte}.$
$\texttt{sopasi(x, y, asi)} = \texttt{sop1(x)}\ 0_2\ \texttt{asi}\ \texttt{sop1(y)}.$
$= \texttt{sop1(x)}\ \texttt{IMM13(y)}\ \texttt{ASI[asi]}.$

Load/Store instructions use rather inconsistently different bit values to denote the type Extended_word, and that is why a separate archetype lias is provided for this type. The asi:a locator points to a normal or alternative address area.

[3.14] $\texttt{s} :: \texttt{Bit}.$
$\texttt{lias(asi, a, signed(s) (r))} = \texttt{s}\ 0_2\ \texttt{Integer}\ \texttt{asi:a[(r} :: \texttt{Integer)]}.$ 5:3.1
$\texttt{lias(asi, a, r)} = 1011_2\ \texttt{asi:a[(r} :: \texttt{Extended\_word)]}.$
$\texttt{OP:LS}\ \texttt{rw(\_, r)}\ 01_2\ \texttt{lias(asi, x + y, r)}\ \texttt{sopasi(x, y, asi)} \Longrightarrow .$

3.3.3 Store integer into alternate space

The store integer into alternate space instructions copy the whole extended (64-bit) integer, the less-significant word, the least-significant halfword, or the least-significant byte of r[rd] into memory. Again, due to the inconsistency described in the previous section a separate archetype sias is provided for the type Extended_word.

[3.15] $\texttt{sias(asi, a, r)} = 01_2\ \texttt{Integer} \Longrightarrow \texttt{asi:a[(r} :: \texttt{Integer)]}.$ 5:3.1
$\texttt{sias(asi, a, r)} = 1110_2 \Longrightarrow \texttt{asi:a[(r} :: \texttt{Extended\_word)]}.$
$\texttt{OP:LS}\ \texttt{rr(\_, r)}\ 01_2\ \texttt{sias(asi, x + y, r)}\ \texttt{sopasi(x, y, asi)} \Longrightarrow .$

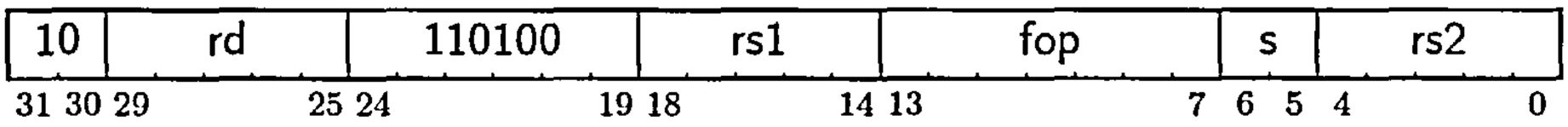


Figure 3.4: Format of the floating-point ADD and SUB instructions

3.4 Floating-point instructions

3.4.1 Floating-point ADD and SUB instructions

The floating-point add instructions add the floating-point register specified by the rs1 field and the floating-point register specified by the rs2 field, and write the sum into the floating-point register specified by the rd field. The floating-point subtract instructions subtract the floating-point register specified by the rs2 field from the floating-point register specified by the rs1 field, and write the difference into the floating-point register specified by the rd field. Note that the command archetype in 3.16 makes use of archetypes `fr` defined in section 2.2 which read appropriate floating-point type values (based on the value in the two-bit cell `s`) into variables `x` and `y`. The archetype `fw` writes the result (`r`) into a floating-point register specified by the rd field, again, its type is based on the value of `s`.

[3.16] $\texttt{s} :: \{\texttt{Bit}\}2.$ 5:3.2

$\texttt{farithm}(\texttt{x},\texttt{y},\texttt{x}+\texttt{y}) = \texttt{FADD}.\ \#\ \textit{FADD} = 0\,01\,00\,00_2$

$\texttt{farithm}(\texttt{x},\texttt{y},\texttt{x}-\texttt{y}) = \texttt{FSUB}.\ \#\ \textit{FSUB} = 0\,01\,00\,01_2$

$\texttt{OP:ARM}\ \texttt{fw}(\_,\texttt{r},\texttt{s})\ 110100_2\ \texttt{fr}(\_,\texttt{x},\texttt{s})\ \texttt{farithm}(\texttt{x},\texttt{y},\texttt{r})\ \texttt{s}\ \texttt{fr}(\_,\texttt{y},\texttt{s}) \Longrightarrow .$

3.4.2 Floating-point compare

These instructions compare the floating-point register specified by the rs1 field with the floating-point register specified by the rs2 field, and set the selected floating-point condition code (fccn) as specified by the 3.17 archetype.

[3.17] $\texttt{fcmp}(\texttt{x},\texttt{y}) = \ =\!\{\,\texttt{x}=\texttt{y}\,\}\!\Rightarrow 00_2.$

$=\!\{\,\texttt{x}<\texttt{y}\,\}\!\Rightarrow 01_2.$

$=\!\{\,\texttt{x}>\texttt{y}\,\}\!\Rightarrow 10_2.$

$=\!\{\,\texttt{x}\ ?\ \texttt{y}\,\}\!\Rightarrow 11_2.\ \#\ \textit{unordered (x or/and y is NaN)}$

Archetypes `fccnr` and `fccnw` read/write a selected floating point condition code field, and finally, update scheme 3.19 makes use of all the archetypes from this section to specify floating-point compare instruction and its field layout.

[3.18] $\texttt{v} :: \{\texttt{Bit}\}2.$ 5:3.2

$\texttt{fccnr}(\texttt{fcc},\texttt{v}) = \texttt{FSR:FCC:fcc[v]}.$

$\texttt{fccnw}(\texttt{fcc},\texttt{v}) = \ \Longrightarrow \texttt{FSR:FCC:fcc[v]}.$

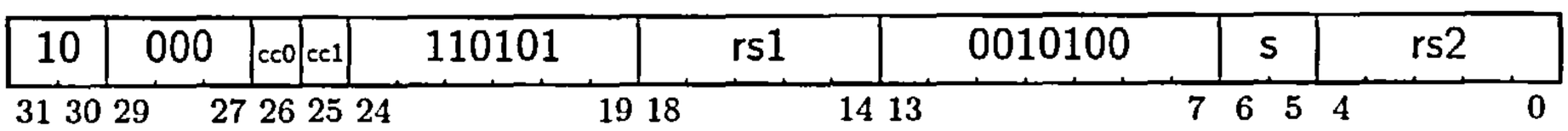


Figure 3.5: Format of the floating-point compare instructions

[3.19] OP:ARM 000_2 cc1 cc0 110101_2 fr(_, x, s) 0010100_2 s fr(_, y, s) $\Longrightarrow$
fccnw(cc1 ‖ cc0, fcmp(x, y)).

3.4.3 Load floating-point from alternate space

The load single floating-point from alternate space instruction LDFA (see figure 3.6) copies a word from memory into f[rd]. The load doubleword floating-point from alternate space instruction (LDDFA) copies a word-aligned double-word from memory into a double-precision floating-point register. The load quad floating-point from alternate space instruction (LDQFA) copies a word-aligned quadword from memory into a quad-precision floating-point register. The command update scheme in 3.20 makes use of archetypes defined earlier in 2.5 and 3.13.

[3.20] ldfa(asi, a, r, 01) = 00_2 asi:a[(r :: Fp_single)]. # *LDFA*
ldfa(asi, a, r, 10) = 11_2 asi:a[(r :: Fp_double)]. # *LDDFA*
ldfa(asi, a, r, 11) = 10_2 asi:a[(r :: Fp_quad)]. # *LDQFA*
OP:LS fw(_, r, s) 1100_2 ldfa(asi, x + y, r, s) sopasi(x, y, asi) $\Longrightarrow$.

3.4.4 Store floating-point into alternate space

The store single floating-point into alternate space instruction (STFA) copies f[rd] into memory. The store double floating-point into alternate space instruction (STDFA) copies a doubleword from a double floating-point register into a word-aligned doubleword in memory. The store quad floating-point into alternate space instruction (STQFA) copies the contents of a quad floating-point register into a word-aligned quadword in memory.

Similarly to load floating-point instructions from ASI (and load/store integer from/into ASI) store floating-point instructions into ASI contain the ASI to be used for the load in the imm_asi field if bit 13 = 0 in the instruction word, or in the ASI register if the same bit (i) reads as 1. The effective address for these instructions is r[rs1] + r[rs2] if $i = 0$, or r[rs1] + sign_ext(simm13) if $i = 1$.

[3.21] stfa(asi, a, r, 01) = $00_2 \Longrightarrow$ asi:a[(r :: Fp_single)]. # *STFA*
stfa(asi, a, r, 10) = $11_2 \Longrightarrow$ asi:a[(r :: Fp_double)]. # *STDFA*
stfa(asi, a, r, 11) = $10_2 \Longrightarrow$ asi:a[(r :: Fp_quad)]. # *STQFA*
OP:LS fr(_, r, s) 1101_2 stfa(asi, x + y, r, s) sopasi(x, y, asi) $\Longrightarrow$.

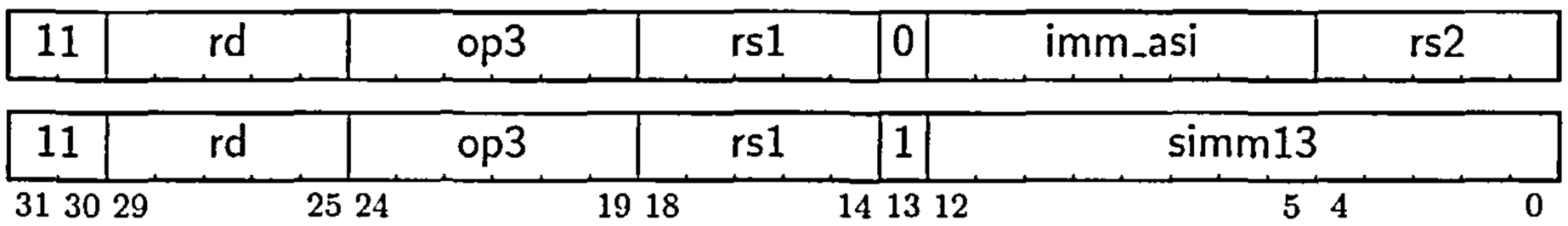


Figure 3.6: Load/Store floating-point from/into alternate space

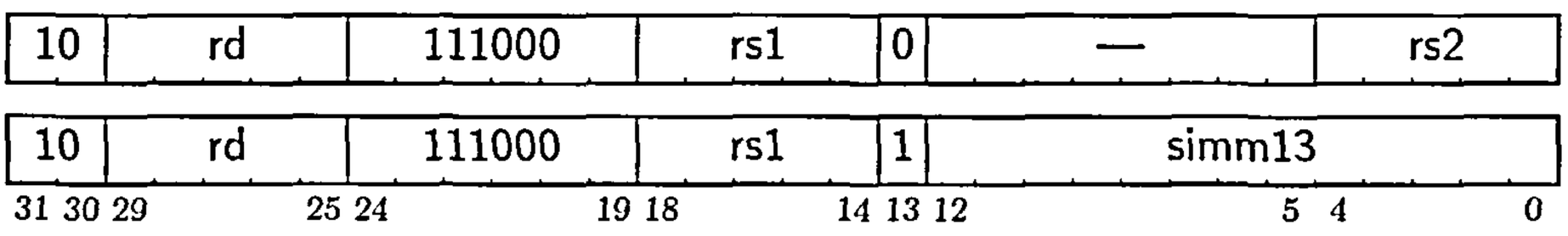


Figure 3.7: Format of the JMPL instruction

3.5 Control transfer instructions

3.5.1 Jump and link

One of the simplest control transfer instructions is the JMPL instruction. The JMPL writes the contents of the PC (program counter), which points to the JMPL instruction itself, into the r[rd] register[2] and then causes a delayed transfer of control (the register nPC [next PC] is written) to the address given by r[rs1] + r[rs2] or r[rs1] + sign_ext(simm13) based on the value of bit 13 of the instruction word. Note that its format is the same as that of arithmetic instructions (figure 3.2). The value 111000_2 is the content of the op3 instruction field and simply distinguishes the JMPL instruction from other (arithmetic) instructions.

[3.22] PC[pc] nPC[npc] pc[OP:ARM rw(_,pc) 111000_2 sop1(x) sop2(y)] $\Longrightarrow$ PC[npc] nPC[x+y].

3.5.2 Branch on integer register with prediction (BPr)

These instructions branch based on the contents of r[rs1]. They treat the register contents as a signed integer value. A BPr instruction examines all 64 bits of r[rs1] according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken; that is, the instruction causes a PC-relative, delayed control transfer to the address PC + (4 × sign_ext(d16hi ‖ d16lo)). If FALSE, the branch is not taken. If the branch is taken, the delay instruction is always executed, regardless of the value of the annul bit. If the branch is not taken and the annul bit (a) is 1, the delay instruction is annulled (not executed)—see update scheme 3.25.

The predict bit (p) is used to give the hardware a hint about whether the branch is expected to be taken. A 1 in the p bit indicates that the branch is expected to be taken; a 0 indicates that the branch is expected not to be taken.

First, let us define some stores which will be used later in the specification.

[2]The value written into the register is visible to the instruction in the delay slot.

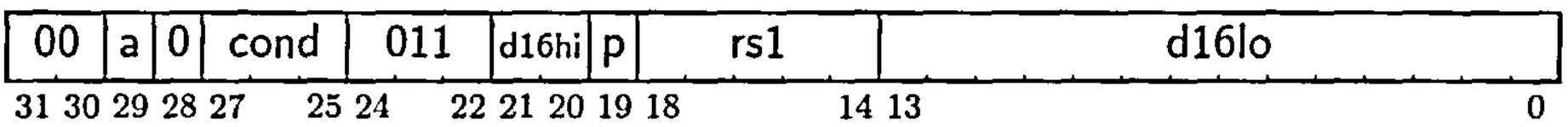


Figure 3.8: Format of BPr instructions

a, p, n :: Bit. # *annul, prediction and "negate condition" bits*
d16hi :: {Bit}2. # *2-bit PC-relative displacement* 5:3.2
rs1 :: {Bit}5. # *address of the 1st source register* 5:3.2
d16lo :: {Bit}14. # *14-bit PC-relative displacement* 5:3.2

The values of constants BRZ, BRLEZ, BRLZ and OP:BRS can be found in section 1.

[3.23] bpr_cond(v, neg(n) (v = 0)) = n BRZ. # *Branch on Register Zero*
bpr_cond(v, neg(n) (v ≤ 0)) = n BRLEZ. # *" Register Less Than or Equal to Zero*
bpr_cond(v, neg(n) (v < 0)) = n BRLZ. # *" Register Less Than Zero*

Note that the archetypes bpr_cond can expand into 6 different update schemes, based on the value of bit 27 (n) of the instruction word. The archetype neg is defined in 3.6 on page 35.

Ambidextrous archetypes ea are used throughout the specification, and expand an effective jump address to branches.

[3.24] ea(pc, displ) pc + (4 × sign_ext(displ)) = . 5:2.2

[3.25] PC[pc] nPC[npc] pc[OP:BRS a 0_2 bpr_cond(v, c) 011_2 d16hi p rw(rs1, v) d16lo]
=[c]⇛ PC[npc] nPC[ea(pc, d16hi ‖ d16lo)].
" =[¬c ∧ a = 0]⇛ PC[npc] nPC[npc + 4]. # *instruction in delay slot executed*
" =[¬c ∧ a = 1]⇛ PC[npc + 4] nPC[npc + 8]. # *instruction in delay slot annulled*

3.5.3 Branch on integer condition codes with prediction (BPcc)

Unconditional Branches (BPA, BPN). A BPN (Branch Never with Prediction) in no case causes a transfer of control to take place. This instruction may be treated as a NOP by an implementation.

BPA (Branch Always with Prediction) causes an unconditional PC-relative, delayed control transfer to the address PC + (4 × sign_ext(disp19)).

If the BPN's or BPA's a field is 1, the following (delay) instruction is not executed. If the a field is 0, the following instruction is executed.

Conditional Branches. Conditional BPcc instructions (except BPA and BPN) evaluate one of the two integer condition codes (icc or xcc), as selected by cc0 and cc1, according to the cond field of the instruction, producing either a TRUE or FALSE result.

If TRUE, the branch is taken; that is, the instruction causes a PC-relative, delayed control transfer to the address PC + (4 × sign_ext(disp19)). If FALSE, the branch is not taken. If a

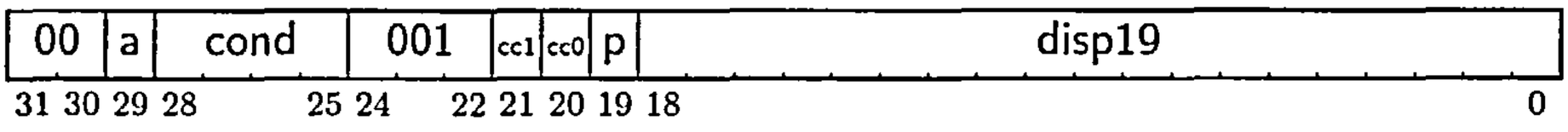


Figure 3.9: Format of BPcc instructions

conditional branch is taken, the delay instruction is always executed regardless of the value of the annul (a) field. If a conditional branch is not taken and the a field is 1, the delay instruction is not executed.

Again, let us define some stores which will be used later in the following sections of this specification.

```
n, z, v, c :: Bit.        # bits of the ICC/XCC register
cc1, cc0 :: Bit.          # condition codes selection
cond :: {Bit}4.           # the condition field                    5:3.2
disp19 :: {Bit}19.        # branch's PC-relative displacement
```

```
[3.26] bp_cond(_, TRUE) = BPA.                                  # Branch Always
       bp_cond(_, FALSE) = BPN.                                 # " Never
       bp_cond(ncc, ¬z) = CCR:ncc[n z v c] BPNE.                # " on ≠
       bp_cond(ncc, z) = CCR:ncc[n z v c] BPE.                  # " on =
       bp_cond(ncc, ¬(z ∨ (n ^ v))) = CCR:ncc[n z v c] BPG.     # " on >
       bp_cond(ncc, z ∨ (n ^ v)) = CCR:ncc[n z v c] BPLE.       # " on ≤
       bp_cond(ncc, ¬(n ^ v)) = CCR:ncc[n z v c] BPGE.          # " on ≥
       bp_cond(ncc, n ^ v) = CCR:ncc[n z v c] BPL.              # " on <
       bp_cond(ncc, ¬(c ∨ z)) = CCR:ncc[n z v c] BPGU.          # " on > Unsigned
       bp_cond(ncc, c ∨ z) = CCR:ncc[n z v c] BPLEU.            # " on ≤ Unsigned
       bp_cond(ncc, ¬c) = CCR:ncc[n z v c] BPCC.                # " on ≥ Unsigned
       bp_cond(ncc, c) = CCR:ncc[n z v c] BPCS.                 # " on < Unsigned
       bp_cond(ncc, ¬n) = CCR:ncc[n z v c] BPPOS.               # " on Positive
       bp_cond(ncc, n) = CCR:ncc[n z v c] BPNEG.                # " on Negative
       bp_cond(ncc, ¬v) = CCR:ncc[n z v c] BPVC.                # " on Overflow Clear
       bp_cond(ncc, v) = CCR:ncc[n z v c] BPVS.                 # " on Overflow Set
```

The bp_cond archetypes are used for reading condition code values, and making a decision about a condition's validity. It also defines the cond field in an instruction word (the 4-bit values of the constants BPA, BPN, ..., BPVS are assumed to be defined elsewhere). COND is a 3-bit offset into the instruction word.

```
[3.27] PC[pc] nPC[npc] pc:COND[cond]
       pc[OP:BRS a]pc:COND[bp_cond(cc1 ‖ cc0, c) 001₂ cc1 cc0 p disp19]
           =⟦ ¬c ∧ a = 0 ⟧⇛ PC[npc] nPC[npc + 4].
       " =⟦ ¬c ∧ a = 1 ⟧⇛ PC[npc + 4] nPC[npc + 8].
```

$$\text{"} \rightleftharpoons [\, \mathtt{c} \wedge \mathtt{a} = 0 \,] \Rightarrow \mathtt{PC[npc]}\ \mathtt{nPC[ea(pc, disp19)]}.$$
$$\text{"} \rightleftharpoons [\, \mathtt{c} \wedge \mathtt{a} = 1 \wedge \mathtt{cond} = \mathtt{BPA} \,] \Rightarrow \mathtt{PC[ea(pc, disp19)]}\ \mathtt{nPC[ea(pc, disp19) + 4]}.$$
$$\text{"} \rightleftharpoons [\, \mathtt{c} \wedge \mathtt{a} = 1 \wedge \mathtt{cond} \neq \mathtt{BPA} \,] \Rightarrow \mathtt{PC[npc]}\ \mathtt{nPC[ea(pc, disp19)]}.$$

Note that the value of the CCR:ncc locator (CCR:ICC or CCR:XCC) depends on the values of condition codes cc1 and cc0 ($00_2 \approx$ XCC, $10_2 \approx$ ICC). The reason for using so many guards in update scheme 3.27 is mainly the different effects of the annul bit on conditional and unconditional branches described earlier in this section.

3.5.4 Branch on floating-point condition codes with prediction (FBPfcc)

The effects of these instructions are exactly the same as those of their integer counterparts described in section 3.5.3 as far as control transfer and annulment of the instruction in a delay slot is concerned. See figure 3.10 for their format.

The following set of archetypes reads a floating-point condition register selected by a 2-bit value fcc and compares it to constants: $\mathtt{E}(00_2), \mathtt{L}(01_2), \mathtt{G}(10_2), \mathtt{U}(11_2) :: \{\mathtt{Bit}\}2$. As a result TRUE or FALSE value is instantiated as the archetype's second parameter. Again, the content of the cond field of the instruction word is defined by 4-bit constants (FBPA, FBPN, ..., FBP0) which are assumed to be defined elsewhere. 5:3.1,3.2

[3.28] fbp_cond(_, TRUE) = FBPA. # *Branch Always*
fbp_cond(_, FALSE) = FBPN. # " *Never*
fbp_cond(fcc, v = U) = fccnr(fcc, v) FBPU. # " *on Unordered*
fbp_cond(fcc, v = G) = fccnr(fcc, v) FBPG. # " *on* >
fbp_cond(fcc, v = U ∨ v = G) = fccnr(fcc, v) FBPUG. # " *on Unordered or* >
fbp_cond(fcc, v = L) = fccnr(fcc, v) FBPL. # " *on* <
fbp_cond(fcc, v = U ∨ v = L) = fccnr(fcc, v) FBPUL. # " *on Unordered or* <
fbp_cond(fcc, v = L ∨ v = G) = fccnr(fcc, v) FBPLG. # " *on* < *or* >
fbp_cond(fcc, v = ¬E) = fccnr(fcc, v) FBPNE. # " *on* ≠
fbp_cond(fcc, v = E) = fccnr(fcc, v) FBPE. # " *on* =
fbp_cond(fcc, v = U ∨ v = E) = fccnr(fcc, v) FBPUE. # " *on Unordered or* =
fbp_cond(fcc, v = G ∨ v = E) = fccnr(fcc, v) FBPGE. # " *on* ≥
fbp_cond(fcc, v = ¬L) = fccnr(fcc, v) FBPUGE. # " *on Unordered or* ≥
fbp_cond(fcc, v = L ∨ v = E) = fccnr(fcc, v) FBPLE. # " *on* ≤
fbp_cond(fcc, v = ¬G) = fccnr(fcc, v) FBPULE. # " *on Unordered or* ≤
fbp_cond(fcc, v = ¬U) = fccnr(fcc, v) FBP0. # " *on Ordered*

A more complex, but concise alternative to archetypes 3.28 is given in 3.29.

[3.29] e :: Bit. lgu :: {Bit}3. 5:3.2
fbp_cond(fcc, neg(e) (*dm*(¬v) & (¬lgu + 1))) = fccnr(fcc, v) e lgu.

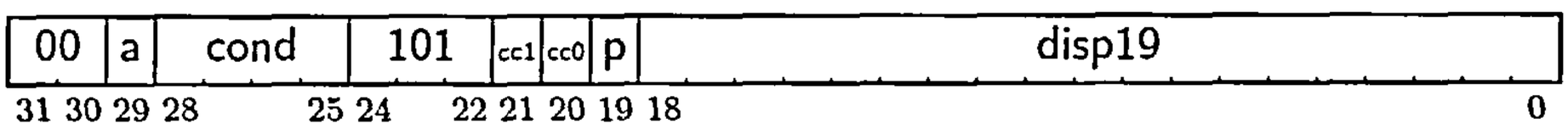


Figure 3.10: Format of FBPfcc instructions

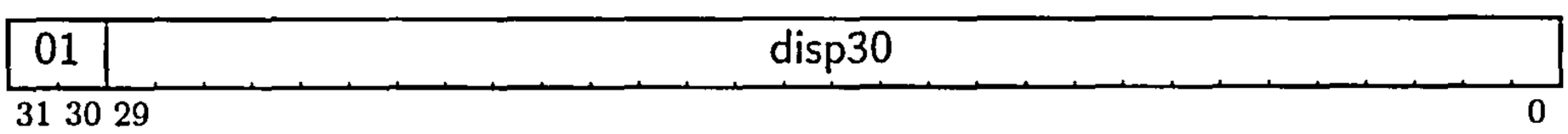


Figure 3.11: Format of the Call and link instruction

The function *dm* is an ordinary demultiplexor, which could be defined using the arithmetic shift left operator $\ll$ as $dm(v)$: $1_2 \ll v \ \{v \mid v \in \mathcal{N}_0\}$.

[3.30] PC[pc] nPC[npc] pc:COND[cond]
pc[OP:BRS a]pc:COND[fbp_cond(cc1 ‖ cc0, c) 101_2 cc1 cc0 p disp19]
$=\!\!\{ \neg c \wedge a = 0 \}\!\!\Rightarrow$ PC[npc] nPC[npc + 4].
" $=\!\!\{ \neg c \wedge a = 1 \}\!\!\Rightarrow$ PC[npc + 4] nPC[npc + 8].
" $=\!\!\{ c \wedge a = 0 \}\!\!\Rightarrow$ PC[npc] nPC[ea(pc, disp19)].
" $=\!\!\{ c \wedge a = 1 \wedge cond = FBPA \}\!\!\Rightarrow$ PC[ea(pc, disp19)] nPC[ea(pc, disp19) + 4].
" $=\!\!\{ c \wedge a = 1 \wedge cond \neq FBPA \}\!\!\Rightarrow$ PC[npc] nPC[ea(pc, disp19)].

Again, the annul bit has a different effect for conditional branches than it does for unconditional branches (in this case opcode FBPA). If the annul bit (a) is set, it annuls the delay instruction for unconditional branches always, whereas for conditional branches it annuls the delay only if the branch is not taken. Please refer to [77] for details.

3.5.5 Call and link

The CALL instruction writes the contents of the PC into r[15] ('out' register[3] 7) and then causes an unconditional delayed transfer of control to a PC-relative effective address.

[3.31] disp30 :: {Bit}30. 5:3.2
PC[pc] nPC[npc] CWP[w] pc[OP:CLL disp30] # OP:CLL = 01_2
$\Longrightarrow$ PC[npc] nPC[ea(pc, disp30)] rw(15, pc, w).

3.5.6 Return

The RETURN instruction causes a delayed transfer of control to the target address and has the window semantics of a RESTORE instruction; that is, it restores the register window prior to the last SAVE instruction. The target address is r[rs1] + r[rs2] if bit 13 of the instruction word (i) is 0, or r[rs1] + sign_ext(simm13) if $i = 1$.

[3] see footnote 2 on page 40

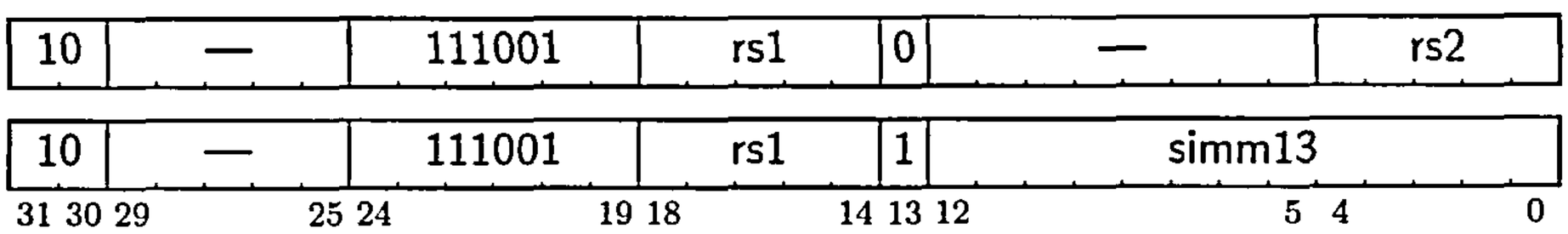


Figure 3.12: Format of the RETURN instruction

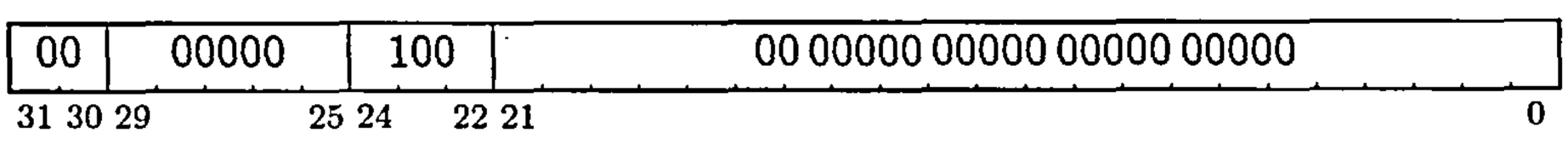


Figure 3.13: Format of the NOP instruction

[3.32] PC[pc] nPC[npc] CWP[w] pc[OP:ARM 00000_2 111001_2 sop1(x) sop2(y)]
$\Longrightarrow$ PC[npc] nPC[x + y] CWP[(w − 1) % NWINDOWS].

Note that registers r[rs1] and r[rs2] come from the old window.

3.6 Miscellaneous instructions

3.6.1 No operation

The NOP instruction changes no program-visible state (except the PC and nPC registers).

[3.33] OP:BRS 00000_2 100_2 00_2 00000_2 00000_2 00000_2 00000_2 $\Longrightarrow$.

4 An example

The following example uses archetypes from sections 2 and 3 to illustrate the archetype expansion mechanism. In this example the update scheme

[3.22] PC[pc] nPC[npc] pc[OP:ARM rw(_,pc) 111000_2 sop1(x) sop2(y)] $\Longrightarrow$ PC[npc] nPC[x+y].

will be expanded into one update scheme showing the effects of the JMPL (Jump and Link) instruction described in section 3.5.1. For maximum simplicity, the instruction uses as its destination and first source operands global r registers ($1 \leq r < 8$) and as its second source operand an immediate value carried by the instruction itself.

The arithmetic command scheme 3.22 expands as shown in figure 3.14. The archetypes and their index numbers from sections 2 and 3 are included as comments for convenience. After substitution using equations derived at each step the full expansion of our update scheme is

PC[pc] nPC[npc] CWP[w] a_1[x]	# *initial configuration*	
pc[OP:ARM _ 111000_2 a_1 IMM13 (v_3 :: {Bit}13)]	# *field format of the current instruction*	5:3.2
$\Rightarrow\!\!\!\!\mid$ $(1 \leq \_ < 8) \wedge (1 \leq a_1 < 8)$ $\mid\!\!\!\Rightarrow$	# *conditions for application*	
PC[npc] nPC[x + sign_ext(v_3)] _[pc].	# *effects of instruction on configuration*	

PC[pc] nPC[npc] pc[OP:ARM rw(_,pc) 111000_2 sop1(x) sop2(y)]
$\Longrightarrow$ PC[npc] nPC[x + y]. *# [3.22]*
 # [2.2] $rw(a, v)$ a = $CWP[w]$ $rw(a, v, w)$. $\_ = a, pc = v$

PC[pc] nPC[npc] CWP[w] pc[OP:ARM rw(a, v, w) a 111000_2 sop1(x) sop2(y)]
$\Longrightarrow$ PC[npc] nPC[x + y].
 # [3.1] $sop1(v_1) = rr(a_1, v_1)$ a_1. $x = v_1$

PC[pc] nPC[npc] CWP[w] pc[OP:ARM rw(a, v, w) a 111000_2 rr(a_1, v_1) a_1 sop2(y)]
$\Longrightarrow$ PC[npc] nPC[v_1 + y].
 # [3.1] $sop2(v_2) = IMM13(v_2)$. $y = v_2$

PC[pc] nPC[npc] CWP[w] pc[OP:ARM rw(a, v, w) a 111000_2 rr(a_1, v_1) a_1 IMM13(v_2)]
$\Longrightarrow$ PC[npc] nPC[$v_1 + v_2$].
 # [2.2] $rw(a, v, w) = =\{\ 1 \leq a < 8\ \}\Rightarrow a[v]$.

PC[pc] nPC[npc] CWP[w] pc[OP:ARM a 111000_2 rr(a_1, v_1) a_1 IMM13(v_2)]
$=\{\ 1 \leq a < 8\ \}\Rightarrow$ PC[npc] nPC[$v_1 + v_2$] a[v].
 # [2.1] $rr(a_1, v_1) = a_1[v_1] =\{\ 1 \leq a_1 < 8\ \}\Rightarrow$.

PC[pc] nPC[npc] CWP[w] $a_1[v_1]$ pc[OP:ARM a 111000_2 a_1 IMM13(v_2)]
$=\{\ (1 \leq a < 8) \wedge (1 \leq a_1 < 8)\ \}\Rightarrow$ PC[npc] nPC[$v_1 + v_2$] a[v].
 # [3.2] $IMM13(sign\_ext(v_3)) = (v_3 :: \{Bit\}13)$. $v_2 = sign\_ext(v_3)$

PC[pc] nPC[npc] CWP[w] $a_1[v_1]$ pc[OP:ARM a 111000_2 a_1 IMM13 (v_3 :: {Bit}13)]
$=\{\ (1 \leq a < 8) \wedge (1 \leq a_1 < 8)\ \}\Rightarrow$ PC[npc] nPC[$v_1 + v_2$] a[v].

Figure 3.14: Example of archetype expansion

For a slightly longer example of archetype expansion please refer to appendix C, where the same instruction uses as its destination operand one of 7 global `r` registers, as its first source operand one of 8 'in' registers, and finally as its second source operand an immediate value.

5 Conclusions

This chapter gives a (partial) specification of the SPARC-V9 CPU at the instruction set level. The main achievement is its compactness (the number of update schemes necessary to specify the whole instruction set of this processor is less then the number of opcodes appearing in the informal specification [77]) and clarity. The next step could be a specification of a SPARC-V9 compliant specific chip (such as UltraSPARC IIi [70]) and proof of their (partial) equivalence. Furthermore, since the same formalism can be used to specify lower level representations (such as the microcode), it would be possible to prove transformations between these levels and reason about the specifications.

Chapter 4

Java Virtual Machine

The Java Virtual Machine (JVM) is a platform independent abstract computing machine. The JVM has a bytecoded instruction set designed to be compact and easily interpreted in either software or hardware. In this chapter, a formal specification of a subset of the JVM instructions is given. Instructions are described using the basic Update Plans formalism briefly introduced in chapter 1. This is not the first attempt at a formal description of Java bytecode semantics as a similar work has already been done [11], however, the following description of JVM instruction semantics reflects the readability and flexibility of Update Plans for this purpose and also demonstrates some of the new Update Plans features introduced to the formalism by this thesis. The specification is based on informal specifications [19, 48, 75] and a part of it was already published in [55].

1 Types, constants, variables

1.1 Types

The following types are used throughout the specification. Store `Bit` is considered to be a primitive type having the usual meaning.

[1.1] `{Bit}`.

```
{Boolean = {Bit}32, Byte = {Bit}32, Char = {Bit}32, Short = {Bit}32,                5:3.2
 Int = {Bit}32, Float = {Bit}32, Reference = {Bit}32,                               5:3.2
 ReturnAddress = {Bit}32, Long = {Bit}64, Double = {Bit}64}.                        5:3.2
```

Since many JVM instructions are defined in terms of so called category types [48], the following type aliases are provided.

[1.2]
```
{Category1  = Boolean|Byte|Char|Short|Int|Float|Reference|ReturnAddress,
 Category2  = Long|Double,
 Category12 = Category1|Category2}.
```

For the purposes of this specification, additional types Signed/Unsigned Byte/Short are declared. The main reason is to supply grounding information for Update Plans. Signed values use two's-complement encoding similarly to Int and Long types.

[1.3] {UnsignedByte = {Bit}8, SignedByte = {Bit}8, 5:3.2
UnsignedShort = {Bit}16, SignedShort = {Bit}16}. 5:3.2

In order to make the definitions of the instructions as compact and clear as possible, an alias type Word is also defined, which is used mainly by stack manipulation instructions.

[1.4] {Word = Category1 Category1 | Category2}.

1.2 Constants

This section contains a list of all JVM opcodes used in this specification with their values.

POP(87), POP2(88), DUP(89), DUP_X1(90), DUP_X2(91), DUP2(92) 5:3.1
DUP2_X1(93), DUP2_X2(94), SWAP(95), 5:3.1
ILOAD(21), ILOAD_0(26), ILOAD_1(27), ILOAD_2(28), ILOAD_3(29), 5:3.1
ISTORE(54), ISTORE_0(59), ISTORE_1(60), ISTORE_2(61), ISTORE_3(62), 5:3.1
IADD(96), ISUB(100), IMUL(104), IDIV(108), IREM(112), 5:3.1
IAND(126), IOR(128), IXOR(130), ISHL(120), ISHR(122), IUSHR(124), 5:3.1
INEG(116), IINC(132), 5:3.1
ICONST_M1(2), ICONST_0(3), ICONST_1(4), ICONST_2(5), ICONST_3(6), ICONST_4(7), 5:3.1
ICONST_5(8), 5:3.1
BIPUSH(16), SIPUSH(17), 5:3.1
IFEQ(153), IFNE(154), IFLT(155), IFGE(156), IFGT(157), IFLE(158), 5:3.1
IF_ICMPEQ(159), IF_ICMPNE(160), IF_ICMPLT(161), IF_ICMPGE(162), IF_ICMPGT(163), 5:3.1
IF_ICMPLE(164), 5:3.1
GOTO(167), GOTO_W(200), JSR(168), JSR_W(201), RET(169), 5:3.1
WIDE(196) :: UnsignedByte. 5:3.1

1.3 Variables

Unless stated otherwise, throughout the whole document the following variables, used for description of instruction parameters, are assumed to have the following types

b_s :: SignedByte.
b_u, opc :: UnsignedByte.
s_s, δ_s :: SignedShort.
s_u :: UnsignedShort.
i, j, δ_i :: Int.
z :: Category12.

2 Instructions

2.1 Operand stack management

A number of instructions are provided for direct manipulation of the operand stack in the current frame: `POP`, `POP2`, `DUP`, `DUP_X1`, `DUP2_X1`, `DUP_X2`, `DUP2_X2` and `SWAP`. One of many possible UP definitions is as follows

[2.1] `POP` $\quad$ SP[t] s[v]t $\Longrightarrow$ SP[s].

`POP2` $\quad$ SP[t] s[w]t $\Longrightarrow$ SP[s].

`DUP` $\quad$ SP[t] s[v]t $\Longrightarrow$ SP[t′] s[v v]t′.

`DUP2` $\quad$ SP[t] s[w]t $\Longrightarrow$ SP[t′] s[w w]t′.

`DUP_X1` $\quad$ SP[t] s[v_1 v_0]t $\Longrightarrow$ SP[t′] s[v_0 v_1 v_0]t′.

`DUP2_X1` SP[t] s[v_1 w_0]t $\Longrightarrow$ SP[t′] s[w_0 v_1 w_0]t′.

`DUP_X2` $\quad$ SP[t] s[w_1 v_0]t $\Longrightarrow$ SP[t′] s[v_0 w_1 v_0]t′.

`DUP2_X2` SP[t] s[w_1 w_0]t $\Longrightarrow$ SP[t′] s[w_0 w_1 w_0]t′.

`SWAP` $\quad$ SP[t] s[v_1 v_0]t $\Longrightarrow$ SP[t′] s[v_0 v_1]t.

where SP denotes a stack pointer in the current frame and the s, t and t′ variables are stack locators. In the specification above, the variables v, w, and their indexed variants, are of data types

[2.2] v :: Category1.
w :: Word.

where variables of the type `Category1` and `Word` occupy one and two stack items on the operand stack respectively.

Variables w, w_0 and w_1 in 2.1 can be viewed as two variables of the type `Category1`. Thus (for example) the definition of the instruction `DUP2_X2` also covers definitions such as

[2.3] `DUP2_X2` SP[t] s[v_2 v_1 w_0]t $\Longrightarrow$ SP[t′] s[w_0 v_2 v_1 w_0]t′.
`DUP2_X2` SP[t] s[w_2 v_1 v_0]t $\Longrightarrow$ SP[t′] s[v_1 v_0 w_2 v_1 v_0]t′.
`DUP2_X2` SP[t] s[v_3 v_2 v_1 v_0]t $\Longrightarrow$ SP[t′] s[v_1 v_0 v_3 v_2 v_1 v_0]t′.

The above specification assumes that the memory for the JVM stack is contiguous and as such is in fact implementation dependent. One of the ways around this particular problem is to use generic push and pop archetypes whose exact definitions would be defined later on in terms of specific/implementation dependent archetypes. For example, assuming there is a generic popsh(pop, psh) archetype combining the effects of pop and push instructions we can simply say that

[2.4] push(v) = popsh(, v). 5:2.1
pop(v) = popsh(v,). 5:2.1

The push and pop archetypes from 2.4 are frequently used in the following sections as a means of defining other JVM instructions which make changes to the operand stack.

2.2 Local variable access

The JVM instruction set includes a number of instructions for accessing local variables of different types in the current frame. The following ambidextrous archetype is used throughout this chapter to denote accesses to a local variable at index n.

[2.5] $\texttt{var}(\texttt{n},\texttt{z})\ \texttt{n[z]} = .$ 5:2.2

Note that the value z right of the locator n can be either of type `Category1` or `Category2` as declared in section 1.3. As the usual size of local variables is 32 bits (the width of the computational type (see [48]) `Int`), value z is stored in one or two local variables based on its size.

Many JVM instructions can be prefixed by the `WIDE` instruction, which results in different interpretation of the instruction modified in this way. The `WIDE` instruction takes one of two formats, depending on the instruction being modified. Archetypes in 2.6 are provided to illustrate the modifications. Two non-wide instruction formats are immediately followed by two wide formats. α and δ helper archetypes reconstruct (un)signed integer values from bytes of data.

[2.6] $\alpha(\texttt{b}_{u1},\texttt{b}_{u2})\ (((\texttt{b}_{u1} \ll 8)|\texttt{b}_{u2}) :: \texttt{UnsignedShort}) = .$ 5:2.2

$\delta(\texttt{b}_{u1},\texttt{b}_{u2})\ (((\texttt{b}_{u1} \ll 8)|\texttt{b}_{u2}) :: \texttt{SignedShort}) = .$ 5:2.2

$\delta(\texttt{b}_{u1},\texttt{b}_{u2},\texttt{b}_{u3},\texttt{b}_{u4})\ ((\texttt{b}_{u1} \ll 24)|(\texttt{b}_{u2} \ll 16)|(\texttt{b}_{u3} \ll 8)|\texttt{b}_{u4}) = .$ 5:2.2

$\texttt{wide}(\texttt{opc},\texttt{b}_u) = \texttt{opc}\ \texttt{b}_u.$

$\texttt{wide}(\texttt{opc},\texttt{b}_u,\texttt{b}_s) = \texttt{opc}\ \texttt{b}_u\ \texttt{b}_s.$

$\texttt{wide}(\texttt{opc},\alpha(\texttt{b}_{u1},\texttt{b}_{u2})) = \texttt{WIDE opc}\ \texttt{b}_{u1}\ \texttt{b}_{u2}.$

$\texttt{wide}(\texttt{opc},\alpha(\texttt{b}_{u1},\texttt{b}_{u2}),\delta(\texttt{b}_{u3},\texttt{b}_{u4})) = \texttt{WIDE opc}\ \texttt{b}_{u1}\ \texttt{b}_{u2}\ \texttt{b}_{u3}\ \texttt{b}_{u4}.$

The most common instructions that access local variables are load and store instructions which transfer values between JVM local variables and the operand stack. The following archetypes describe the instruction formats of all `Int` load/store instructions.

[2.7] $\texttt{i3}(0) = 0.\quad \texttt{i3}(1) = 1.\quad \texttt{i3}(2) = 2.\quad \texttt{i3}(3) = 3.$

$\texttt{iload(i)} = \texttt{wide(ILOAD, i)}.$

$\texttt{iload(i)} = \texttt{ILOAD\_\~{}i3(i)}.$ 5:3.1

$\texttt{istore(i)} = \texttt{wide(ISTORE, i)}.$

$\texttt{istore(i)} = \texttt{ISTORE\_\~{}i3(i)}.$ 5:3.1

The purpose of the `i3` archetypes is to condense the specification which would otherwise had to explicitly list all `ILOAD_`n and `ISTORE_`n instructions that implicitly access variables n. Finally, update schemes in 2.8 show the effects that these instruction have on the JVM machine.

[2.8] $\texttt{iload(n) var(n,v) push(v)} \Longrightarrow .$
$\texttt{istore(n) pop(v)} \Longrightarrow \texttt{var(n,v)}.$

All the instructions described in this section operate on the data type Int. Similar archetypes and update schemes could be specified for the instructions operating on the data types Float, Long, Double, and Reference; that is, instructions FLOAD, LLOAD, DLOAD, and ALOAD respectively, plus their implicit immediate operand variants FLOAD_n, LLOAD_n, DLOAD_n, and ALOAD_n, where $n \in \{0,1,2,3\}$.

2.3 Arithmetic instructions

The Java Virtual Machine set offers a wide range of arithmetic operators. It is assumed that the standard arithmetic operators are defined in the standard environment, so only these less common arithmetic operators are informally explained: shift left (<<), shift right (>>), and unsigned shift right (>>>). Archetypes and an update scheme for all the JVM's double operand Int arithmetic instructions are given in 2.9.

[2.9]
$$
\begin{aligned}
&\texttt{iarithm\_binary}(i_1, i_2, i_1 + i_2) &&= \texttt{IADD}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 - i_2) &&= \texttt{ISUB}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \times i_2) &&= \texttt{IMUL}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 / i_2) &&= \texttt{IDIV} \dashv\!\![\; i_2 \neq 0 \;]\!\!\Rightarrow .\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \,\%\, i_2) &&= \texttt{IREM} \dashv\!\![\; i_2 \neq 0 \;]\!\!\Rightarrow .\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \,\&\, i_2) &&= \texttt{IAND}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \mid i_2) &&= \texttt{IOR}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \,\hat{}\, i_2) &&= \texttt{IXOR}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \ll i_2) &&= \texttt{ISHL}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \gg i_2) &&= \texttt{ISHR}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_1 \ggg i_2) &&= \texttt{IUSHR}.\\
&\texttt{iarithm\_binary}(i_1, i_2, i_r)\ \texttt{popsh}(i_1\ i_2, i_r) \Longrightarrow .
\end{aligned}
$$

Similarly, the only single operand[1] Int arithmetic instruction present in the JVM instruction set INEG can be defined as

[2.10]
$$
\begin{aligned}
&\texttt{iarithm\_monadic}(i_1, -i_1) = \texttt{INEG}.\\
&\texttt{iarithm\_monadic}(i, i_r)\ \texttt{popsh}(i, i_r) \Longrightarrow .
\end{aligned}
$$

Note that because of the two's-complement representation used for negative numbers, negation of the minimum value of the type Int produces the same value, not the maximum value of this type as one might expect.

[1]The Int bitwise negation is not present in the JVM instruction set, and is typically carried out by the instruction IXOR with the constant ICONST_M1.

And finally the last arithmetic instruction defined in this section is the `Int` increment of a local variable by a constant. It is the only arithmetic instruction, which can take the wide instruction format.

[2.11] $\texttt{wide(IINC, s}_u\texttt{, s}_s\texttt{) var(s}_u\texttt{, i)} \Longrightarrow \texttt{var(s}_u\texttt{, i} + \texttt{s}_s\texttt{)}$.

All the arithmetic instructions specified here operated on the data type `Int`. The instruction set of the Java Virtual Machine contains similar instructions for data types `Long`, `Float`, and `Double`, and so their formal description would not be beneficial to explanation of their semantics, or as a demonstration of the UP formalism.

2.4 Immediate operands

The instruction set of the Java Virtual Machine includes a number of instructions pushing an immediate operand onto the operand stack.

The simplest form of these instructions are pushes of implicit immediate operands. In a way similar to the `i3` archetypes from section 2.2, `i5` archetypes are used in an effort to make the specification as succinct as possible.

[2.12] $\texttt{i5}(-1) = \texttt{M1}.\quad \texttt{i5}(0) = 0.\quad \texttt{i5}(1) = 1.\quad \texttt{i5}(2) = 2.\quad \texttt{i5}(3) = 3.\quad \texttt{i5}(4) = 4.\quad \texttt{i5}(5) = 5.$
$\texttt{iconst(i)} = \texttt{ICONST\_}\tilde{}\texttt{i5(i)}.$ 5:3.1

The rest of the instructions that push `Int` values onto the operand stack have explicit immediate operands.

[2.13] $\texttt{iconst}(\texttt{b}_s) = \texttt{BIPUSH b}_s.$
$\texttt{iconst}(\delta(\texttt{b}_{u1}, \texttt{b}_{u2})) = \texttt{SIPUSH b}_{u1}\ \texttt{b}_{u2}.$
$\texttt{iconst(i) push(i)} \Longrightarrow .$

Please note that there is an implicit promotion of (un)signed types `Byte` and `Short` to `Int` described by [48].

Again, all the instructions described in this section operated on the data type `Int`. Similar archetypes and update schemes could be specified for the instructions operating on data types

- `Float`: `FCONST_`n, where $n \in \{0, 1, 2\}$
- `Long`: `LCONST_`n, where $n \in \{0, 1\}$
- `Double`: `DCONST_`n, where $n \in \{0, 1\}$
- `Reference`: `ACONST_NULL` pushing a special `NULL` value

2.5 Control transfer

The control transfer instructions conditionally or unconditionally make the Java Virtual Machine continue execution with an instruction other than the one following the control transfer instruction. These instructions can be divided into three categories

- conditional branches (e.g. IFEQ δ, IF_ICMPEQ δ) comparing a top stack item against zero or comparing two topmost stack items against each other
- compound conditional branches (TABLESWITCH and LOOKUPSWITCH) with a variable number of operands (branches)
- unconditional branches (GOTO δ_s, GOTO_W δ_i, JSR δ_s, JSR_W δ_i, RET b_u and WIDE RET s_u)

As has already been pointed out, there are two groups of conditional branches comparing Int type values. The first one compares its parameter against an implicit immediate operand 0 and the other one compares its two parameters. Both versions are listed on the same line for the same type of conditional jump. The δ archetype defined in section 2.2, is used.

[2.14]
$$\begin{aligned}
&\texttt{jmpc(i, i} = 0, \delta(b_1, b_2)) = \texttt{IFEQ } b_1, b_2. && \texttt{jmpc(i, j, i} = \texttt{j}, \delta(b_1, b_2)) = \texttt{IF_ICMPEQ } b_1\ b_2.\\
&\texttt{jmpc(i, i} \neq 0, \delta(b_1, b_2)) = \texttt{IFNE } b_1\ b_2. && \texttt{jmpc(i, j, i} \neq \texttt{j}, \delta(b_1, b_2)) = \texttt{IF_ICMPNE } b_1\ b_2.\\
&\texttt{jmpc(i, i} < 0, \delta(b_1, b_2)) = \texttt{IFLT } b_1\ b_2. && \texttt{jmpc(i, j, i} < \texttt{j}, \delta(b_1, b_2)) = \texttt{IF_ICMPLT } b_1\ b_2.\\
&\texttt{jmpc(i, i} \geq 0, \delta(b_1, b_2)) = \texttt{IFGE } b_1\ b_2. && \texttt{jmpc(i, j, i} \geq \texttt{j}, \delta(b_1, b_2)) = \texttt{IF_ICMPGE } b_1\ b_2.\\
&\texttt{jmpc(i, i} > 0, \delta(b_1, b_2)) = \texttt{IFGT } b_1\ b_2. && \texttt{jmpc(i, j, i} > \texttt{j}, \delta(b_1, b_2)) = \texttt{IF_ICMPGT } b_1\ b_2.\\
&\texttt{jmpc(i, i} \leq 0, \delta(b_1, b_2)) = \texttt{IFLE } b_1\ b_2. && \texttt{jmpc(i, j, i} \leq \texttt{j}, \delta(b_1, b_2)) = \texttt{IF_ICMPLE } b_1\ b_2.
\end{aligned}$$

The list of conditional branches in 2.14 is not complete. Again, only instructions comparing Int type values are included. There are four other conditional branches comparing values of the type Reference: IF_ACMPEQ δ_s, IF_ACMPNE δ_s, IFNULL δ_s and IFNULL δ_s; however, the semantics of these instructions is very similar to that of the instructions already defined in 2.14.

[2.15]
$$\begin{aligned}
&\texttt{jmpu(TRUE, } \delta(b_1, b_2)) && = \texttt{GOTO } b_1\ b_2.\\
&\texttt{jmpu(TRUE, } \delta(b_1, b_2, b_3, b_4)) && = \texttt{GOTO_W } b_1\ b_2\ b_3\ b_4.
\end{aligned}$$

Note that the only difference between the instructions GOTO δ_s and GOTO_W δ_i is the size of their operands. The wide variant of the unconditional jump takes a four-byte offset, whereas the standard variant takes only a two-byte offset. The aim of the archetype definitions above is to compress both the conditional and unconditional branches into one update scheme 2.17.

[2.16]
$$\begin{aligned}
\texttt{jump(cond, } \delta_i) &= \texttt{jmpc(i, cond, } \delta_i) \texttt{ pop(i).}\\
&= \texttt{jmpc(i, j, cond, } \delta_i) \texttt{ pop(i j).}\\
&= \texttt{jmpu(cond, } \delta_i).
\end{aligned}$$

[2.17]
$$\begin{aligned}
\texttt{PC[pc] pc[jump(cond, } \delta_i)]\texttt{qc} &= \{ \texttt{ cond } \} \Rightarrow \quad \texttt{PC[pc} + \delta_i].\\
\text{"} \qquad\qquad &= \{ \neg\texttt{cond } \} \Rightarrow \texttt{PC[qc]}.
\end{aligned}$$

PC is the JVM program counter, which contains the address of the current instruction. Note that the target address of a GOTO(_W) instruction must be that of an opcode of an instruction within the method that contains this instruction. This is not dealt with in this specification, since this is the responsibility of the JVM bytecode verifier, which is not specified here.

The format of JSR(_W) instructions is exactly the same as the format of GOTO(_W) instructions. However, contrary to the GOTO(_W) instructions, the JSR(_W) instructions also push the return address on the operand stack as shown in 2.19.

[2.18] $\texttt{jsr}(\delta(\texttt{b}_1, \texttt{b}_2)) = \texttt{JSR b}_1\ \texttt{b}_2.$
$\texttt{jsr}(\delta(\texttt{b}_1, \texttt{b}_2, \texttt{b}_3, \texttt{b}_4)) = \texttt{JSR\_W b}_1\ \texttt{b}_2\ \texttt{b}_3\ \texttt{b}_4.$

Note that the asymmetry of JSR and RET instructions is intentional. After a JSR instruction, the return address is stored into a local variable by one of ASTORE instructions, and this local variable may in turn be used by a RET instruction.

[2.19] $\texttt{PC[pc] pc[jsr}(\delta_\texttt{i})\texttt{]qc push(qc)} \Longrightarrow \texttt{PC[pc} + \delta_\texttt{i}\texttt{]}.$
$\texttt{PC[pc] pc[wide(RET, n)]qc var(n, a)} \Longrightarrow \texttt{PC[a]}.$

3 Conclusions

The specification of the subset of the JVM instruction set is the third and last demonstration of the original Update Plans, using only minor syntactic extensions from Extended Update Plans introduced in the second part of this thesis. It provides a proof that even more abstract instruction sets using a wide variety of types can easily be described by UP. An interesting exercise for the future could be a specification of a real Java processor (used widely in mobile devices) which usually implements one of several existing JVM subsets.

Part II

Extended Update Plans

Introduction

The first part of the thesis concentrated mainly on Update Plans as proposed by [60]. Several case studies have been performed testing the formalism on real examples, and a comparison of UP against other existing formal methods has been made. This prompted a number of research questions some of which are addressed in this second part of the thesis. This second part contains a number of syntactic and semantic extensions to Update Plans called Extended Update Plans (EUP). The main contribution of EUP is a concept of sequential update scheme and archetypes and improved consistency of the formalism in some areas. Finally, various other formal methods with similar application domains are examined and comparisons with the Update Plans formalism are drawn.

Chapter 5

Syntactic Extensions

The whole Update Plans grammar has been revised, and undergone changes to make it more compact and consistent in its structure and terminology. As a result, the formalism became much more simple to use without the need for frequent reference to its grammar. As an additional benefit, the implementation will be more transparent. Some syntactic changes, such as parallel blocks in archetypes introduced new possibilities in Update Plans and (inevitably) resulted in the need to define their semantics. These changes along with a completely new concept of sequential update schemes and archetypes will be discussed in chapter 6.

This chapter describes most of the important changes and additions to the grammar and the semantic impact of less significant syntactic changes. The complete revised grammar of Extended Update Plans can be found in appendix A.

1 Everything is an update

Most 'inconsistencies' in the original Update Plans grammar stemmed from the fact that there was no unifying concept of an 'update'. In the new grammar, an *update* is either a parallel block, a sequential block or a set of alternatives. In other words, changes to a configuration are always caused by an update. Another significant inconsistency was the use of a dot '.' to separate all (top-level) items other than parallel blocks. Strictly only (top-level) items are separated by a dot now. One of the consequences is that sugared multiple archetype definitions such as

$$\begin{aligned} a(params) &= update_1 \\ &= update_2 \\ &= \quad \vdots \\ &= update_n. \end{aligned}$$

are perceived as one item. They are separated from other items in an update plan by the dot behind the last update $\texttt{update}_\texttt{n}$ and from individual updates of the archetype by the '=' sign.

A similar consideration applies to parallel blocks. The use of the double pipeline symbol '||' to separate alternatives in a parallel block was entirely optional. As only items are now separated by a dot, it is necessary to separate individual updates in parallel blocks. This is done by changing the production rule for parallel blocks to

⟨*parblock*⟩ → "(||" {⟨*update*⟩ "||"}$^+$ "||)"

2 Archetypes

2.1 Grammar

The following grammar tries to be as compact as possible while preserving all features offered by the original grammar.

⟨*item*⟩ → ⟨*archetype definition*⟩

⟨*archetype definition*⟩ →
⟨*basic archetype definition*⟩ |
⟨*ambidextrous archetype definition*⟩

⟨*basic archetype definition*⟩ → ⟨*basic declaration*⟩ ⟨*basic definition*⟩$^+$

⟨*basic declaration*⟩ → ⟨*basic archetype name*⟩ ⟨*parameters*⟩

⟨*ambidextrous archetype definition*⟩ →
⟨*ambidextrous declaration*⟩ ⟨*basic definition*⟩$^+$

⟨*ambidextrous declaration*⟩ →
⟨*archetype name*⟩ ⟨*parameters*⟩ ⟨*text*⟩

⟨*basic definition*⟩ → "=" ⟨*archetype body*⟩

⟨*archetype body*⟩ → ⟨*update*⟩ | ⟨*configuration*⟩

⟨*parameters*⟩ → "(" {⟨*text*⟩ "," }* ")"

A ⟨*basic archetype name*⟩ is an identifier, an ⟨*archetype name*⟩ is a symbolic constant or an identifier. Note that the pipeline symbol '|' is no longer used to denote ambidextrous archetypes. The use of this symbol by ambidextrous archetypes and other means of separating the text by command (also ambidextrous) archetypes was one of the main inconsistencies. The consequences of these simplifications are discussed in the following two sections. The syntax of archetype calls has been left unchanged and can be found in appendix A or in the original work [60].

2.2 Ambidextrous archetypes

As mentioned earlier, the '|' symbol no longer denotes the text to be expanded on the left or right-hand side of an update scheme during ambidextrous archetype expansion. Instead, the

text is placed between the parameters of an archetype and its definition, so that the archetype definitions

$$a_l(params) = text\ lc =\{\ g\ \}\Rightarrow rc.$$
$$a_r(params) = lc =\{\ g\ \}\Rightarrow text\ rc.$$

can be 'compressed' into one EUP ambidextrous archetype definition

$$a(params)\ text = lc =\{\ g\ \}\Rightarrow rc.$$

where *lc*/*rc* and *text* are left/right contexts and text shared among archetypes a_l, a_r and a. Note that every archetype whose left and right-hand side expansions are empty is an ambidextrous archetype, which was not the case in basic Update Plans.

If the archetype body of an ambidextrous archetype consists of alternatives

$$a(params)\ text = lc_1 =\{\ g_1\ \}\Rightarrow rc_1;\ lc_2 =\{\ g_2\ \}\Rightarrow rc_2;\ \ldots;\ lc_n =\{\ g_n\ \}\Rightarrow rc_n.$$

it is interpreted as syntactic sugar for

$$a(params)\ text = lc_1 =\{\ g_1\ \}\Rightarrow rc_1.$$
$$a(params)\ text = lc_2 =\{\ \neg g_1 \wedge g_2\ \}\Rightarrow rc_2.$$
$$\vdots$$
$$a(params)\ text = lc_n =\{\ (\bigwedge_{i=1}^{n-1} \neg g_i) \wedge g_n\ \}\Rightarrow rc_n.$$

Note that text in ambidextrous archetypes is not allowed to appear on the left or right-hand side of an archetype's body. The semantics of ambidextrous sequential archetypes is discussed in chapter 6.

2.3 Command archetypes

If a is an ambidextrous archetype and the expansion of all definitions of a begin with the same constant then that constant may be used as the archetype name, unless it has already been so used. The remainder of the expansion is then placed immediately after archetype's parameters. Such an archetype is called *command archetype*. The command archetype

$$CONST(params)\ text = update.$$

is the sugared version of

$$a(params)\ CONST\ text = update.$$

with calls of a replaced throughout the update plan by calls of *CONST*.

2.4 Archetype parameters

An archetype's parameters can be referenced in the archetype's body by the symbol '$' and a number directly corresponding to the position of the archetype's parameter where the parameters are numbered from the left starting from 1. For instance, the fourth parameter 'x + y' is referred to by $4 in the following archetype.

$\texttt{add(o, b, c, x + y)} = \texttt{A[o]a a[x] b[y]} \Longrightarrow \texttt{c[\$4]}.$

One of the changes to the archetype grammar is that the individual parameters of an archetype call can be empty. For example $\texttt{add(, b, c, r)}$ is a perfectly valid archetype call, and an "empty string" is in place of parameter \$1. No production rules are added to the grammar as the preprocessor can expand these references as simple macros.

2.5 Archetypes in guards

The last extension concerning archetypes concerns archetype calls appearing in guards. Archetype calls were not allowed in guards in basic Update Plans. As there is no reason for this restriction, archetype calls can be used in EUP guards unless they are recursive.

3 Types

3.1 Constants

Constants are uppercase words in Update Plans. An exception to this rule is when making an explicit declaration of constants using the predefined type `Constant`. Then it is possible to compose constants of lowercase letters. For instance, `c` and `nPC` are given "a constant status" by the following definition $\texttt{c, nPC} :: \texttt{Constant}$.

Variables, on the other hand, are denoted by lowercase words. However, even if a word uses uppercase letters, it can be assigned an explicit type. Furthermore, individual types can be viewed as constants when used as an ordinary text in an update scheme. This feature can, for example, compress the following two update schemes

$$\begin{aligned}\texttt{get\_r(a, r)} &= 001_2\ \texttt{a[(r :: Byte)]}.\\ &= 010_2\ \texttt{a[(r :: Halfword)]}.\end{aligned}$$

into only one

$$\texttt{get\_r(a, r)} = \texttt{Integer a[(r :: Integer)]}.$$

provided that a new type alias `Integer` is created, and that the constants `Byte` and `Halfword` are assigned symbolic values 001_2 and 010_2 respectively by an explicit list of types with their symbolic values.

$\{\texttt{Byte}(001_2), \texttt{Halfword}(010_2)\}.$
$\{\texttt{Integer} = \texttt{Byte} \mid \texttt{Halfword}\}.$

It is sometimes useful to initialise a variable or a constant to a value. The following example assumes that we want constants `E`, `L`, `G` and `U` to have symbolic values $00_2, 01_2, 10_2$

and 11_2 respectively. Note that this example makes use of the repeat construct introduced in section 3.2.

$\texttt{E}(00_2), \texttt{L}(01_2), \texttt{G}(10_2), \texttt{U}(11_2) :: \{\texttt{Bit}\}2.$

A new meaning is given to the symbol '~'. It is now used as the text concatenation symbol. For instance a sequence 'T~ E~ X~ T' is interpreted as a constant 'TEXT'. A more realistic example of its use can be found in section 2.4 of chapter 4.

An updated type grammar for these changes is given in the following section.

3.2 Type grammar

⟨*item*⟩ → ⟨*store declaration*⟩ | ⟨*type declaration*⟩

⟨*store declaration*⟩ → "{" {⟨*store*⟩ "," }$^+$ "}"

⟨*type declaration*⟩ → {(⟨*term*⟩ ⟨*const*⟩-*opt*) ","}$^+$ "::" ⟨*store structure*⟩

⟨*term*⟩ → "(" ⟨*term*⟩ "::" ⟨*store structure*⟩ ")"

⟨*store*⟩ →
    ⟨*store identifier*⟩ |
    ⟨*store identifier*⟩ "=" ⟨*store structure*⟩

⟨*store identifier*⟩ → ⟨*store name*⟩ ⟨*const*⟩-*opt*

⟨*const*⟩ → "(" ⟨*number*⟩ ")"

Note that ⟨*store structure*⟩ is a regular expression over a set of ⟨*store name*⟩s, i.e. lower case words with a leading upper case letter.

A new type can be declared reusing an old one provided that the structure of the new type contains a pattern identical to the structure of the old type. In order to express the exact number of repetitions of the old structure in the new type, the syntax of the original Update Plans needs to be extended by adding the following rules.

⟨*store structure*⟩ → "{" ⟨*store structure*⟩ "}" ⟨*number*⟩

For instance, assuming that the type `Bit` is already defined, the type `Byte` could be defined using the repeat construct as $\{\texttt{Byte} = \{\texttt{Bit}\}8\}$.

4 Comments

The last trivial but useful syntactic addition to basic Update Plans are comments. A comment is started by the symbol '#' and is terminated by the end of the line. They would be removed by a preprocessor during lexical analysis.

5 Conclusions

Although not significant, the changes described in this chapter contribute to the usability of the formalism in three ways. Firstly, the grammar has now a slightly less restrictive syntax. As a result several simple semantic rules had to be added to the formalism, and the more will follow in the following chapter. This, however, is a small price to pay for its increased consistency and more intuitive use of the whole formalism. Secondly, the introduction of a few simple conventions improves the type reuse mechanism and allows even more compact, multi-level UP specifications. And finally, the changes to the grammar open new possibilities to UP such as nesting of sequential/parallel blocks within an update which will be described in the following chapter.

Chapter 6

Semantic Extensions

While parallel blocks form a useful extension to basic Update Plans by making them more readable, they do not add much power to the formalism as they can be interpreted as mere syntactic sugar. Moreover, one of the drawbacks of a parallel block lies in the nondeterministic execution of its constituent update schemes which makes any synchronisation of these schemes impossible.

Not only do the extensions described in this chapter address these problems, but they also provide Update Plans with modularity, which with the exception of the archetype mechanism was not present in basic Update Plans.

The layout of this chapter is as follows. Firstly, a simple convention concerning parallel blocks making Extended Update Plans more consistent is introduced in section 1. Secondly, the syntax, semantics and the implementation of sequential update schemes is presented in section 2. An extension, known as sequential archetypes, closely related to the archetype mechanism makes the concept of sequential update schemes more powerful and is introduced in section 3. The EUP grammar allows (in contrast to the basic UP grammar) any kind of update to appear in the body of an archetype. The consequences of this change are considered in section 4. Finally, conclusions are drawn in section 5.

1 Parallel blocks

Parallel blocks were introduced in [60] as a way of expressing synchronous parallelism. In the original Update Plans, a parallel block is a set of independent alternatives which are applied to a configuration simultaneously. The application of a parallel block is naturally atomic, so all changes to the configuration contributed by any of its alternatives will appear simultaneously. The only way to share data between the individual alternatives is to use constant locators.

The introduction of a simple convention allowing variables to be shared across all updates in a parallel block not only simplifies sharing of data among these updates and makes a parallel block consistent in this respect with its sequential counterpart, but it also makes

transformation of parallel blocks into canonical form slightly easier as there is no need to rename unrelated variables of the same name among the parallel block's updates.

Example 1

Consider the following update plan, which best illustrates the difference between the semantics of parallel blocks in basic and Extended Update Plans.

```
A[0] B[1].          # initial configuration
|| A[a] ⟹ A[a + 1]  # basic Update Plans require '.' here
|| B[a] ⟹ B[a − 1].
```

In basic Update Plans, the first application of the parallel block will result in the configuration A[1] B[0]. In Extended Update Plans, the parallel block is not applicable, as the cells right of A and B have different values and as such the set of locator expressions on the left-hand side of the resulting update scheme is not consistent.

2 Sequential update schemes

2.1 Background/Motivation

Apart from improving modularity in Update Plans, the motivation for sequential update schemes had two sources. Firstly, at the formalism level, it was necessary to provide some synchronisation for update schemes, and on a related note, at the application level, the introduction of explicit sequential execution proved to be very useful as there is often a need to express a sequential execution of update schemes without resorting to techniques which would make the specification less transparent.

At the instruction level, consider any PDP-11-*like* instruction, but with three operands[1], e.g. `ADD z x y`, with `x` and `y` source operands and `z` the destination operand. The semantics of this instruction seems simple. Source operands `x` and `y` are added and the result is written into the destination `z`. Consider, however, what happens if two of the operands use addressing modes that address the same register, and change the value in that register—e.g. `x` is postincrement on register `R1` (`R1+`) and `y` is predecrement on the same register (`-R1`). It is then not clear what values are being addressed. Should `x` be addressed first and then `y`—i.e. get `x` from `@R1`, increment `R1` then decrement `R1` and get `y` from `@R1`, or is `y` addressed first and then `x`, or are they in some way accessed in parallel. In the original Update Plans such an instruction was illegal.

[1]PDP-11 instructions have a maximum of only 2 operands

The specification of the above-mentioned addressing modes and the arithmetic instruction is in 2.1.

[2.1] $\texttt{POSTINC(b,v) r} = \texttt{r[b] b[v]c} \Longrightarrow \texttt{r[c]}.$ # *postincrement mode*
$\texttt{PREDEC(a,v) r} = \texttt{r[b] a[v]b} \Longrightarrow \texttt{r[a]}.$ # *predecrement mode*
$\texttt{arithm(x,y,x+y)} = \texttt{ADD}.$ # *addition*
$\texttt{arithm(x,y,r) r}_3\ \texttt{POSTINC(b,x) PREDEC(a,y)} \Longrightarrow \texttt{r}_3\texttt{[r]}.$

In the following text the expansion of an `ADD R2 R1+ -R1` instruction is examined. First, before an archetype's expansion, it is necessary to rename all archetype's variables so that they do not conflict with variables within the scheme the archetype is being called from.

[2.2] $\texttt{POSTINC(b,v) r}_1 = \texttt{r}_1\texttt{[b] b[v]c} \Longrightarrow \texttt{r}_1\texttt{[c]}.$

The `POSTINC` archetype is now used and its parameters added to the resolution set.

[2.3] $\texttt{arithm(x,y,r) r}_3\ \texttt{POSTINC r}_1\ \texttt{r}_1\texttt{[b] b[v]c PREDEC(a,y)} \Longrightarrow \texttt{r}_3\texttt{[r] r}_1\texttt{[c]}.$
$\#\{\texttt{b} = \texttt{b}, \texttt{x} = \texttt{v}\}$

Similarly, all conflicting local variables of the `PREDEC` archetype are renamed.

[2.4] $\texttt{PREDEC(a,v}_1\texttt{) r}_2 = \texttt{r}_2\texttt{[b}_1\texttt{] a[v}_1\texttt{]b}_1 \Longrightarrow \texttt{r}_2\texttt{[a]}.$

Finally, after the `PREDEC`'s archetype expansion, the original update scheme becomes

[2.5] $\texttt{arithm(x,y,r) r}_3\ \texttt{POSTINC r}_1\ \texttt{r}_1\texttt{[b] b[v]c PREDEC r}_2\ \texttt{r}_2\texttt{[b}_1\texttt{] a[v}_1\texttt{]b}_1 \Longrightarrow \texttt{r}_3\texttt{[r] r}_1\texttt{[c] r}_2\texttt{[a]}.$
$\#\{\texttt{a} = \texttt{a}, y = v_1\}$

If $\texttt{r}_1$ and $\texttt{r}_2$ refer to the same register (`R1`) then $\texttt{b} = \texttt{b}_1$, so it must be the case that $\texttt{a} \neq \texttt{c}$. This is therefore a conflict between the cells $\texttt{r}_1\texttt{[c]}$ and $\texttt{r}_2\texttt{[a]}$, i.e. `R1[c]` and `R1[a]` (the right-hand side of the update scheme 2.5 is inconsistent). Although the update scheme is not yet fully expanded, we can already recognise the inconsistency at this early stage of archetype expansion.

Although there is a good reason for making these instructions illegal, there are situations when instruction operands are accessed sequentially in a clearly defined order during a f/e cycle. While even such cases can be specified using basic Update Plans, there would be a significant loss of clarity in the specification.

On most PDP-11 implementations, operands of an instruction are accessed in some sequence. In this example operand access order of the `ADD z x y` instruction is x, then y and finally z. Using the extension to the UP formalism introduced in this chapter, the behaviour of our arithmetic instruction can be described in terms of sequential update scheme S in 2.6.

[2.6] $\texttt{S}\left|\begin{array}{ll} 3 & \texttt{arithm(x,y,r) r}_3 \Longrightarrow \texttt{r}_3\texttt{[r]} \\ 1 & \texttt{POSTINC(b,x)} \Longrightarrow \\ 2 & \texttt{PREDEC(a,y)} \Longrightarrow . \end{array}\right.$

Informally, the above definition says: expand archetypes `POSTINC`, `PREDEC`, and `arithm` and apply their locator expressions to the configuration in this exact order (the application order). However, the ordering of text expanded from sequence's archetypes is from left to right/top to bottom (the textual order).

2.2 Syntax

Similarly to parallel blocks [60], *sequential blocks* are delimited by the *open sequential block symbol*, '(*SEQ*|' which takes an identifier (*sequencer*) here '*SEQ*', and the *close sequential block symbol*, '|)'. The reason for using the identifier in the open sequential block symbol will become clear in section 3 where sequential archetypes are discussed.

While the double pipeline symbol '||' is entirely optional (with the exception of the open/close parallel block symbol) in basic Update Plans, the (single) pipeline symbol is essential in sequential blocks, in order to separate updates in these blocks[2]. The pipeline symbol takes an additional number which is the order in which updates nested in the sequential block are applied to a configuration.

A basic notation for *sequential update schemes* (*sequences* for short) is

(*SEQ*|a $update_{a}$
|b $update_{b}$
⋮
|m $update_{m}$
|)

or making use of typesetting possibilities

$$SEQ \left|\begin{array}{ll} \mathtt{a} & update_{a} \\ \mathtt{b} & update_{b} \\ & \vdots \\ \mathtt{m} & update_{m} \end{array}\right.$$

where *update*s are either alternatives, a parallel block, or a sequence.

Sequences either have all *stages* (sequence numbers/indicators; in the above generic sequential update scheme a, b, ..., m) in a sequential block tagged, or all stages left untagged. Such 'untagged' sequences are so-called *stageless*. A sequence with no sequential block identifier is referred to as an *anonymous sequence*. Sequencers do not have to be unique, but if they are not, the risk of creating an incorrect specification by an oversight is increased as a sequential archetype can expand into two or more unrelated sequences as will be seen in section 3.3.

The following production rules need to be added to the UP grammar.

[2]Note that in EUP the double pipeline symbol is no longer optional.

⟨*item*⟩ → ⟨*seqblock*⟩

⟨*seqblock*⟩ → "(" ⟨*seqblock id*⟩-*opt* "|" {(⟨*stage*⟩-*opt* ⟨*update*⟩) "|"}$^{+}$ "|)"

⟨*seqblock id*⟩ → ⟨*symb-const*⟩

⟨*stage*⟩ → ⟨*symb-const*⟩ | ⟨*variable*⟩

The complete grammar of Extended Update Plans can be found in appendix A.

2.3 Semantics

Consider the generic sequential update scheme from section 2.2. There are $n = |\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{m}\}|$ updates in the sequence, where the variables $\mathtt{a}, \mathtt{b}, \ldots, \mathtt{m}$ are stages of application. All the stage-representing variables can be grouped into a countable set of variables $\mathcal{V} = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{m}\}$. Every variable $v \in \mathcal{V}$ instantiates a value $i \in \mathcal{I}$, where $\mathcal{I}$ is a countable instantiation set. The mapping between $\mathcal{V}$ and $\mathcal{I}$ is a many-to-one mapping, in other words, two or more stages in a sequential block can have equal instantiations, which allows for non-deterministic sequential updates schemes. The stages represent the *application order* for all the updates within a sequence and for every instantiation α

$\{\alpha \,|\, \alpha \in (\mathcal{I} - \{\omega\})\}$, where $\omega = \max(\mathcal{I})$

of these variables there is a direct successor function $succ(\alpha)$. If for any $\{\alpha \,|\, succ(\alpha) \notin \mathcal{I}\}$, the sequence is always only partially applicable, as explained later. If for all $\{\alpha \,|\, succ(\alpha) \in \mathcal{I}\}$, stages do not restrict the applicability of the sequence, and it can be fully applicable depending on the applicability of its updates. The application/temporal order of updates in a sequence is defined as

$$\min(\mathcal{I}), succ(\min(\mathcal{I})), succ(succ(\min(\mathcal{I}))), \ldots, succ^{n-1}(\min(\mathcal{I})) \equiv \max(\mathcal{I})$$

where $\min(\mathcal{I})$ is the first stage, and $succ^{n-1}(\min(\mathcal{I}))$ is the last stage of application.

Although the application of a sequence takes place in individual stages (steps), it is still atomic (as is for example the application of a parallel block) in relation to the rest of the updates outside the sequence. The syntax and the semantics of any of these n updates (including their atomicity of application) is naturally preserved. The semantics of sequential blocks nested in parallel blocks is slightly more complicated, and is explained in section 2.4.

Unlike parallel blocks, which are either applicable as a whole or simply not applicable at all, we define a *full* and *partial applicability* of sequences. A sequence is fully applicable if all of its constituent updates are applicable in the application order. A sequence is partially applicable, if one or more but less than n updates are applicable in the application order. For example, sequence 2.1 in chapter 7 will always be only partially applicable unless some sequential archetype expands its stage two. The effects of a sequential update scheme $\mathcal{S}$ which is only partially applicable are permanent, unless there was another update which was applicable (in the case of a sequential update scheme this may be either fully or partially) before the

application of the scheme $\mathcal{S}$. This behaviour is especially useful when simulating fatal failures of a system, when the system retains the state when the failure occurred. Breaking the application order by omitting a stage $\{\alpha \mid succ(\alpha) \notin \mathcal{I}\}$ results in a sequential block which is always only partially applicable.

Since text can be expanded as a result of a sequence's application, we need to define the placement of the text for the sequence as a whole, and also an internal *textual ordering.* The placement of text expanded as a result of a *top-level* sequence's application fits nicely with the concept of a command driven update plan. An update plan's top-level update is also an item in the update plan. In other words, a top-level update is always separated from other update plan's items by the '.' symbol. Since the application of a sequence can be viewed as an atomic action, it can also be viewed as a command update scheme

$$ltxt\ lc \Rightarrow\!\!\!\!\mid\ g\ \mid\!\!\!\!\Rightarrow rtxt\ rc$$

which can be desugared for top-level sequences as

$$\texttt{PC[pc]}\ \texttt{pc}[ltxt]\texttt{qc}\ lc \Rightarrow\!\!\!\!\mid\ g\ \mid\!\!\!\!\Rightarrow \texttt{PC[pc']}\ \texttt{pc'}[rtxt]\texttt{qc}\ rc.$$

where $ltxt/rtxt$ or both are non-empty left/right-hand side texts, and lc/rc is the left/right-hand side context before/after application of a sequence, and PC is a program counter.

The (internal) textual ordering of left/right-hand side text $ltxt_i/rtxt_i,\ i \in \{\texttt{a},\texttt{b},\ldots,\texttt{m}\}$ from updates of the form

$$ltxt_i\ lc_i \Rightarrow\!\!\!\!\mid\ g_i\ \mid\!\!\!\!\Rightarrow rtxt_i\ rc_i$$

is then

$$ltxt_{\texttt{a}}\ ltxt_{\texttt{b}} \cdots\ ltxt_{\texttt{m}}\ lc \Rightarrow\!\!\!\!\mid\ g\ \mid\!\!\!\!\Rightarrow rtxt_{\texttt{a}}\ rtxt_{\texttt{b}} \cdots\ rtxt_{\texttt{m}}\ rc$$

which can again be desugared for top-level sequences as

$$\texttt{PC[pc]}\ \texttt{pc}[ltxt_{\texttt{a}}\ ltxt_{\texttt{b}} \cdots\ ltxt_{\texttt{m}}]\texttt{qc}\ lc \Rightarrow\!\!\!\!\mid\ g\ \mid\!\!\!\!\Rightarrow \texttt{PC[pc']}\ \texttt{pc'}[rtxt_{\texttt{a}}\ rtxt_{\texttt{b}} \cdots\ rtxt_{\texttt{m}}]\texttt{qc}\ rc.$$

Not only are variables shared between left and right-hand sides of an update scheme, but they are also shared across all updates of a sequence. This further reduces the complexity of an update plan as it is possible to share data between updates directly rather than using additional constant locators. However, as will be shown in section 3.7, no forward references to variables are allowed.

The temporal ordering of stageless sequences is equivalent to its textual ordering. In other words, the stageless sequence

$$\begin{array}{l} SEQ \\ \left|\begin{array}{l} update_{\texttt{a}} \\ update_{\texttt{b}} \\ \quad\vdots \\ update_{\texttt{m}} \end{array}\right. \end{array}$$

is syntactic sugar for

$$SEQ \begin{cases} \min(\mathcal{I}) & update_a \\ succ(\min(\mathcal{I})) & update_b \\ \vdots & \vdots \\ succ^{n-1}(\min(\mathcal{I})) \equiv \max(\mathcal{I}) & update_m \end{cases}$$

2.4 Canonical form

A sequential block is said to have a *canonical form* if all of its updates are update schemes. The canonical form of sequential blocks is introduced to simplify the implementation of sequential update schemes and to define the semantics of sequential blocks nested in parallel blocks.

Every sequential block has its canonical form. The following text shows how a sequential update scheme can be translated into its canonical form.

One of the requirements for transformation of a sequential update scheme into canonical form is that all its stages and stages of all nested sequences must be ground. At first sight this might seem as a limitation, but as temporal ordering is also required for the implementation, and transformation into canonical form is primarily required as the first step of the implementation, this is not an issue.

The algorithm can be divided into two independent parts. Firstly, the sequential block is 'linearised' by three transformations so that all of updates in the block are alternatives or parallel blocks. Then, in the second part, any parallel blocks and alternatives are replaced by update schemes. Due to the possible presence of alternatives (or alternatives nested in parallel blocks), this step (conversions 4 and 5) may produce two or more sequential update schemes in canonical form.

The algorithm

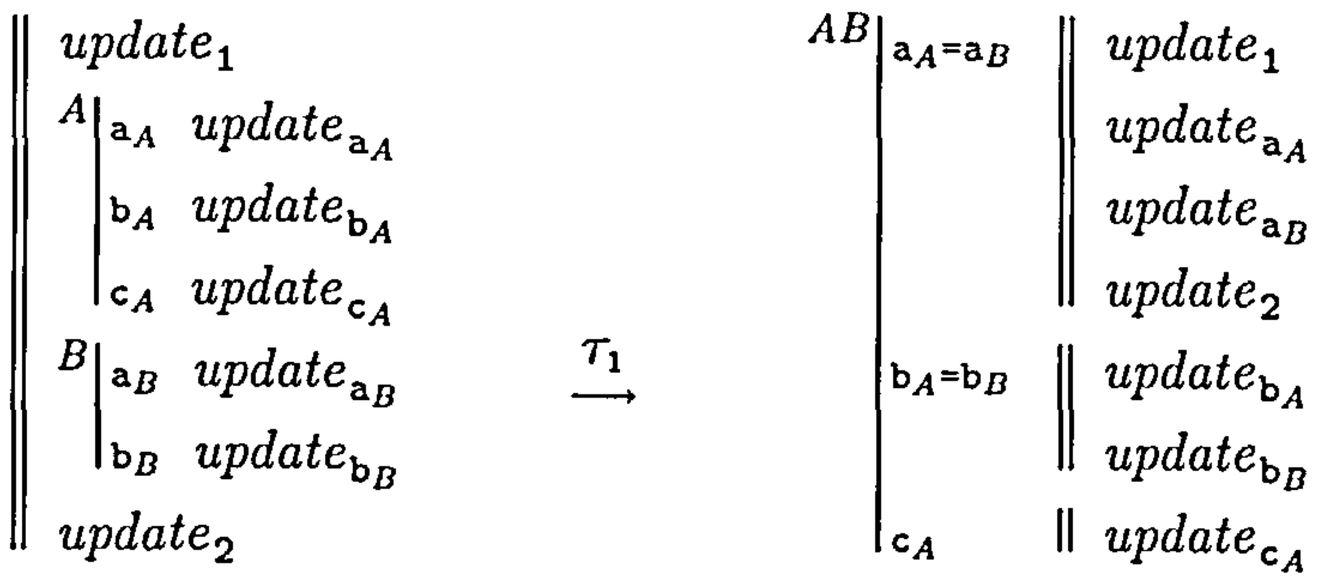
while(sequential block contains other sequential blocks) {
  convert:
    1) sequential blocks in || blocks /* τ_1 */
    2) sequential blocks in sequential blocks /* τ_2 */
    3) || blocks in || blocks /* τ_3 */
}
/ assertion: the only type of nested updates are || blocks or alternatives */*
convert:
  4) all || blocks into update schemes /* τ_{4a}, τ_{4b} */
  5) all alternatives into update schemes /* τ_5 */

All five individual steps of the algorithm are explained in more detail in the following sections.

Sequential blocks in a parallel block

Of all the transformations used in the algorithm the transformation of sequential blocks nested in a parallel block has the most noticeable impact on the structure of the resulting sequential

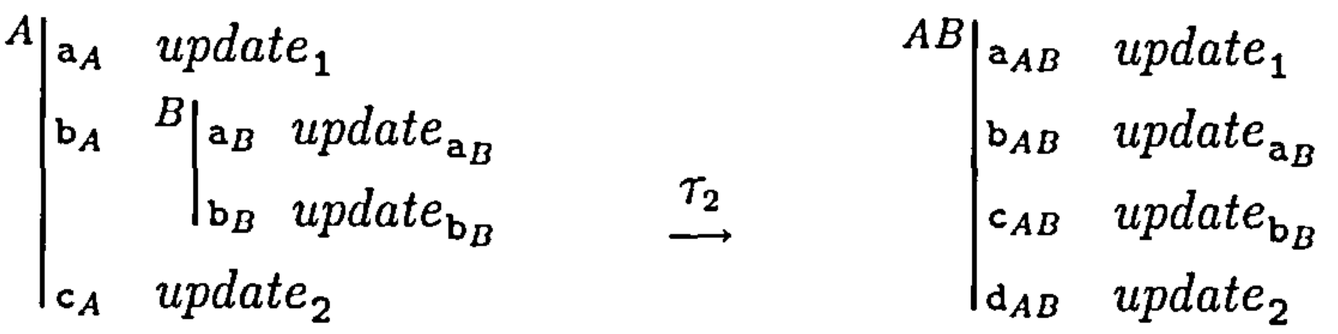
block. As a side effect, the transformation also shows the semantics of sequential blocks nested in a parallel block. The main idea is demonstrated by the following example. There are two sequential updates A and B, and two non-sequential updates $update_1$ and $update_2$ in the parallel block. Let there be instantiation sets $\mathcal{I}_A$ and $\mathcal{I}_B$ containing instantiations of sequential blocks' A and B stages. The parallel block can then be replaced by a sequential block containing n parallel blocks/stages, where $n = |\mathcal{I}_A \cup \mathcal{I}_B|$. Each of these parallel blocks/stages will contain updates to be applied at the same time. All non-sequential updates are grouped in the first stage of the newly created sequential block.



Provided stages $\mathtt{a}_A = \mathtt{a}_B$ and $\mathtt{b}_A = \mathtt{b}_B$, the parallel block can be split into three separate parallel blocks as demonstrated above (τ_1). As all updates in a parallel block are guaranteed to be desugared, no special care needs to be taken about textual ordering.

Sequential blocks in a sequential block

Let there be a sequential block A with a nested sequential block B in stage $\mathtt{b}_A$. Then it is possible to transform (τ_2) the original sequential block A into a sequential block AB by simply splitting B's stages and adding them to the the newly created sequential block.



Values of stages of the new block AB must preserve the application order to ensure the semantic equivalence of sequential blocks A and AB. Values of all stages temporally preceding stage $\mathtt{b}_A$ are left unchanged. For instance, assuming that $\mathtt{b}_A$ temporally follows both the $\mathtt{a}_A$ and $\mathtt{c}_A$ stages, $\mathtt{a}_{AB} = \mathtt{a}_A$ and $\mathtt{d}_{AB} = \mathtt{c}_A$. All stages $\mathcal{B}$ introduced from the block B will have their value changed to $\mathtt{b}_A + b - \min(\mathcal{B})$, where $b \in \mathcal{B}$.

Values of all stages temporally following stage $\mathtt{b}_A$ need to be incremented by the difference between the last and the first stage to be applied in block B, i.e. $\max(\mathcal{B}) - \min(\mathcal{B})$. Textual ordering is naturally preserved.

Parallel blocks in a parallel block

The third step of conversion of a sequential block into its canonical form is the simplest one. A parallel block in the sequential block $\mathcal{S}$ having as its updates other parallel blocks will be transformed into a new parallel block using a simple manipulation demonstrated by the following figure.

$$\left\| \begin{array}{l} update_1 \\ \left\| \begin{array}{l} update_2 \\ update_3 \end{array} \right. \\ update_4 \end{array} \right. \quad \xrightarrow{\tau_3} \quad \left\| \begin{array}{l} update_1 \\ update_2 \\ update_3 \\ update_4 \end{array} \right.$$

Parallel blocks

In contrast to the three transformations described above, transformation of parallel blocks into canonical form can produce two or more updates (update schemes) derived from a single parallel block. This fact is due to the possible presence of alternatives, which will now be separated into update schemes.

A parallel block $\mathcal{P}$ containing m alternatives with a variable amount of update schemes (us_{rc}) for each of the alternatives

$$\left\| \begin{array}{l} us_{11};\ us_{12};\ \ldots;\ us_{1a} \\ us_{21};\ us_{22};\ \ldots;\ us_{2b} \\ \quad \vdots \quad \ddots \\ us_{m1};\ us_{m2};\ \ldots;\ us_{mn} \end{array} \right.$$

can be transformed (τ_{4a}) into $x = a \times b \times \cdots \times n$ parallel blocks

$$\left\| \begin{array}{l} us_{11} \\ us_{21} \\ \vdots \\ us_{m1} \end{array} \right. \left\| \begin{array}{ll} us_{11} & \\ us_{21} & \\ \vdots & \ddots \\ us_{m2} & \end{array} \right. \left\| \begin{array}{l} us_{11} \\ us_{21} \\ \vdots \\ us_{mn} \end{array} \right. \left\| \begin{array}{ll} us_{11} & \\ us_{22} & \ldots \\ \vdots & \ddots \\ us_{m1} & \end{array} \right. \left\| \begin{array}{l} us_{11} \\ us_{2b} \\ \vdots \\ us_{mn} \end{array} \right. \left\| \begin{array}{ll} us_{12} & \ldots \\ us_{21} & \ldots \\ \vdots & \ddots \\ us_{m1} & \end{array} \right. \left\| \begin{array}{l} us_{1a} \\ us_{2b} \\ \vdots \\ us_{mn} \end{array} \right.$$

This means that the sequential block containing the parallel block $\mathcal{P}$ will be replaced by multiple (x) instances of the sequential block where $\mathcal{P}$'s occurrence will be replaced in each of the instances by a parallel block transformed as shown above.

Every parallel block consisting of only update schemes

$$\left\| \begin{array}{l} lhs_1 \,\text{=}\!\{\, g_1 \,\}\!\Rightarrow rhs_1 \\ lhs_2 \,\text{=}\!\{\, g_2 \,\}\!\Rightarrow rhs_2 \\ \quad \vdots \\ lhs_m \,\text{=}\!\{\, g_m \,\}\!\Rightarrow rhs_m \end{array} \right.$$

is said to be in *canonical form* and can be rewritten (τ_{4b}) as a single update scheme simply by taking the unions of all left/right-hand sides and combining the guards.

$$lhs_1\ lhs_2\ \cdots\ lhs_m =\!\!\{\ g_1 \wedge g_2 \wedge \cdots \wedge g_m\ \}\!\!\Rightarrow rhs_1\ rhs_2\ \cdots\ rhs_m$$

Alternatives

Linearised sequential blocks containing alternatives will be normalised (τ_5) into multiple sequential blocks using exactly the same principal as described in the previous section.

An alternative $\mathcal{A}$ containing n update schemes

$$lhs_1 =\!\!\{\ g_1\ \}\!\!\Rightarrow rhs_1;\ \ldots;\ lhs_n =\!\!\{\ g_n\ \}\!\!\Rightarrow rhs_n.$$

can be rewritten as n update schemes

$$\begin{array}{l} lhs_1 =\!\!\{\ g_1\ \}\!\!\Rightarrow rhs_1. \\ \quad\vdots \\ lhs_n =\!\!\{\ (\bigwedge_{i=1}^{n-1} \neg g_i) \wedge g_n\ \}\!\!\Rightarrow rhs_n. \end{array}$$

The sequential block containing alternative $\mathcal{A}$ will then be replaced by n instances of the sequential block where $\mathcal{A}$'s occurrence will be replaced in each of the instances by the individual update schemes shown above.

2.5 Implementation

In this section a simple translation of sequential update schemes to basic update schemes is presented. The advantages of such a translation are obvious. As an algorithm for implementation of basic Update Plans has already been described [60], it is not necessary to redesign the whole implementation. In other words, the proposed translation is a front-end, independent of the actual implementation of basic UP. The implementation is divided in three sections. Firstly, the main idea is presented in the following section. Secondly, the algorithm is described in section 2.5.2. Finally, section 2.5.3 gives an example of the implementation.

2.5.1 Preliminaries

The algorithm described in the following section assumes that the sequential update scheme under transformation is already in its canonical form and that all of its sequential archetypes have been expanded. For the expansion of sequential archetypes please refer to section 3.3. The main idea is to translate all sequential update schemes at compile-time into basic update schemes and adding a mechanism to synchronise the translated update schemes and existing top-level updates. Synchronisation is made possible by the introduction of a shared central update plan *synchroniser* `UP:SEQ`, which is a locator addressing the synchroniser's stack. The role of the stack is twofold. Firstly, it serves as an extra guard to allow the application of only a particular update scheme from a sequential block, and secondly, if the application of a sequential block finished, or has yet to start, it allows the application of only top-level updates.

2.5.2 The algorithm

For the purposes of the implementation, the following popsh archetype is defined. It is a two-parameter stack management archetype, which combines the effects of the classical pop and push operations.

[2.7] $\texttt{popsh}(\texttt{v}_1, \texttt{v}_2) = \texttt{UP:SEQ[q]}\ \texttt{p}[\texttt{v}_1]\texttt{q} \Longrightarrow \texttt{UP:SEQ[q']}\ \texttt{p}[\texttt{v}_2]\texttt{q}'$. *# pop and push*

In order to prevent existing top-level update schemes from interfering with half-completed sequential update schemes, it is necessary to add an archetype call popsh(UP:TOP, UP:TOP) to every top-level update scheme in an update plan. The initial synchroniser's configuration is UP:SEQ[p] [UP:TOP]p, which enables the application of all top-level update schemes. The algorithm for the actual translation of sequential update schemes into basic update schemes is described below.

Consider the following generic top-level canonical sequential update scheme S

[2.8] $$S \left| \begin{array}{ll} \texttt{a} & ltxt_{\texttt{a}}\ lc_{\texttt{a}} =\!\{\ g_{\texttt{a}}\ \}\!\Rightarrow rtxt_{\texttt{a}}\ rc_{\texttt{a}} \\ \texttt{b} & ltxt_{\texttt{b}}\ lc_{\texttt{b}} =\!\{\ g_{\texttt{b}}\ \}\!\Rightarrow rtxt_{\texttt{b}}\ rc_{\texttt{b}} \\ & \vdots \\ \texttt{m} & ltxt_{\texttt{m}}\ lc_{\texttt{m}} =\!\{\ g_{\texttt{m}}\ \}\!\Rightarrow rtxt_{\texttt{m}}\ rc_{\texttt{m}} \end{array} \right.$$

where $ltxt_i/rtxt_i$ is the left/right-hand side text and lc_i/rc_i is the left/right-hand side context of an update scheme $i \in \{\texttt{a}, \texttt{b}, \ldots, \texttt{m}\}$.

As already explained in section 2.3, not only are variables shared between left and right-hand sides of an update scheme, but they are also shared across all updates in a sequence. As the aim is to convert a sequential block into top-level update schemes, and the only way to share data across top-level updates is using ground (in other words constant) locators, additional ground locators will be used as addresses of the shared variables.

Let there be a set $\mathcal{A}$ of update schemes which share a variable v in the sequence S. Let there also be a universe of locators $\mathcal{U}$ used in the transformed update plan and a locator $\mathcal{V}$, where $\mathcal{V} \notin \mathcal{U}$. Then all update schemes from $\mathcal{A}$ need to be adapted by adding a locator expression $\mathcal{V}[v]$ using the following two rules.

1. An update scheme from $\mathcal{A}$ in which v is ground is subject to transformation τ_1

$$lhs =\!\{\ g\ \}\!\Rightarrow rhs \quad \xrightarrow{\tau_1} \quad lhs =\!\{\ g\ \}\!\Rightarrow rhs\ \mathcal{V}[v]$$

2. An update scheme from $\mathcal{A}$ in which v is not ground is subject to transformation τ_2

$$lhs =\!\{\ g\ \}\!\Rightarrow rhs \quad \xrightarrow{\tau_2} \quad lhs\ \mathcal{V}[v] =\!\{\ g\ \}\!\Rightarrow rhs$$

It is also necessary to make the textual ordering of a sequence's text explicit, to conform to the semantics of sequences described in 2.3. Let there be an ordering $\mathcal{B}$ of n update schemes in the sequence S ordered textually in the order text is arranged in a configuration

after expansion as described in section 2.3. Then every update scheme from that ordering is assigned a number t in the range 1 to n representing the position it appears in $\mathcal{B}$ and is subject to transformation

$$ltxt\ lc \Rightarrow\!\!\!\!\!\!\{\ g\ \}\!\!\!\!\!\!\Rightarrow\ rtxt\ rc$$

$$\xrightarrow{\tau_3}$$

$$\texttt{PC[pc]}\ (\texttt{pc} + \sum_{j=1}^{t-1} |ltxt_j|)[ltxt]\ lc \Rightarrow\!\!\!\!\!\!\{\ g\ \}\!\!\!\!\!\!\Rightarrow\ (\texttt{pc} + \delta - \sum_{j=t}^{n} |rtxt_j|)[rtxt]\ rc\ pc$$

where $\delta = \sum_{j=1}^{n} |ltxt_j|$ and $pc \equiv \texttt{PC[pc} + \delta - \sum_{j=t}^{n} |rtxt_j|]$ if the update scheme under transformation is in the last (temporal) stage of S, otherwise pc is empty. The '$|\cdot|$' symbol is the length operator.

Lastly, the application order of update schemes within S needs to be made explicit together with an explicit statement of its entering and leaving.

Rule 1. The first update scheme μ_f of the sequence S can only be applied if there is an appropriate unique value σ corresponding to μ_f on the synchroniser. The following update scheme needs to be added for (top-level) sequence S in order to enter sequential application of S's schemes.

[2.9] `popsh(UP:TOP, UP:TOP` σ`)` $\Longrightarrow$. # *preamble, the first update scheme (rule 1)*

Rule 2. Let there be a temporal ordering $\mathcal{C}$ of all sequence's S stages given by the successor function $succ(\alpha)$ as described in section 2.3. Then for every stage $\alpha \in \mathcal{C}$ there is an update scheme μ in that stage (already adapted by transformations τ_1, τ_2 and τ_3) which will be translated into a basic update scheme

[2.10] `popsh(`$\mathtt{v}_1, \mathtt{v}_2$`)` μ. # *sequential block itself (rule 2 and 3)*

where $\mathtt{v}_1$ and $\mathtt{v}_2$ are both unique constants—the first one identifying the update scheme in stage α, the second one identifying the update scheme in stage $succ(\alpha)$. The uniqueness of these constants in combination with the `popsh` archetype ensures sequentiality of application of all update schemes within the sequence S and no interference with other updates.

Rule 3. The update scheme μ_l in the last stage of S's application—$succ^{n-1}(\min(\mathcal{I}))$, where $\mathcal{I}$ is a countable instantiation set of the sequence S as explained in section 2.3—is not subject to rule 2, as its successor function is not defined. There is, however, a need to transfer control back to (or enable execution of) top-level updates, in other words exit the sequential application of S. This is ensured by adding an update scheme 2.10 where $\mu \equiv \mu_l$, $\mathtt{v}_2$ = '`UP:TOP`' and $\mathtt{v}_1$ = '`UP:TOP` ψ', where ψ is the constant which was used by rule 2 as $\mathtt{v}_2$ for μ in stage $succ^{n-2}(\min(\mathcal{I}))$.

Anonymous and stageless sequences are mere syntactic sugar for ordinary sequences as described in sections 2.2 and 2.3. As such they do not need to be considered as a special case for implementation.

2.5.3 Example

Continuing with the 'PDP-11' example introduced in this chapter as one of the motivations for sequential update schemes, the implementation of the sequential update scheme S shown in section 2.1 on page 65 is now examined. It is given again for convenience in 2.11.

[2.11] $^{S}\Big|_{1}^{3}$ `arithm(x,y,r)` $r_3 \Longrightarrow r_3[r]$
`POSTINC(b,x)` $\Longrightarrow$
`PREDEC(a,y)` $\Longrightarrow$.

As the sequential update scheme is already in canonical form, we can start with transformations τ_1 and τ_2 and thus make sharing of variables between update schemes explicit.

[2.12] $^{S}\Big|_{1}^{3}$ `arithm(x,y,r)` r_3 `X[x] Y[y]` $\Longrightarrow r_3[r]$ $\#\tau_2,\tau_2$ *(x and y not ground by* `arithm`*)*
`POSTINC(b,x)` $\Longrightarrow$ `X[x]` $\#\tau_1$ *(archetype POSTINC grounds x)*
`PREDEC(a,y)` $\Longrightarrow$ `Y[y]`. $\#\tau_1$ *(archetype PREDEC grounds y)*

Note that variables `a`, `b`, r_3 and `r` are not shared among the individual update schemes of S, and as such are not subject to transformations τ_1 and τ_2.

As mentioned in section 2.5.1, this is a compile-time transformation and so all text-expanding archetypes of a sequence must be expanded before the sequence can be translated into basic update schemes. This step is demonstrated by the update scheme 2.13.

[2.13] $^{S}\Big|_{1}^{3}$ `ADD` r_3 `X[x] Y[y]` $\Longrightarrow r_3[r]$ $\#\{x=x, y=y, r=x+y\}$
`POSTINC` r_1 r_1`[b] b[v]c` $\Longrightarrow$ `X[x]` r_1`[c]` $\#\{b=b, x=v\}$
`PREDEC` r_2 $r_2[b_1]$ $a[v_1]b_1 \Longrightarrow$ `Y[y]` r_2`[a]`. $\#\{a=a, y=v_1\}$

Note that the variables of archetype definitions have been renamed before the expansion of archetype calls as in section 2.1. As a sequential update scheme shares variables among its individual updates, variables in all updates within a sequence have to be taken into account during the renaming process.

Assuming that $|$`ADD` $r_3|=16$, $|$`POSTINC` $r_1|=8$, and $|$`PREDEC` $r_2|=8$, internal and external textual orderings are made explicit by the transformation τ_3 (see section 2.3) as shown by 2.14.

[2.14] $^{S}\Big|_{1}^{3}$ `PC[pc] pc[ADD` r_3`] X[x] Y[y]` $\Longrightarrow r_3$`[r] pc+32[] PC[pc + 32]` $\#t=1, n=3$
`PC[pc] pc+16[POSTINC` r_1`]` r_1`[b] b[v]c` $\Longrightarrow$ `X[x]` r_1`[c] pc+32[]` $\#t=2$
`PC[pc] pc+24[PREDEC` r_2`]` $r_2[b_1]$ $a[v_1]b_1 \Longrightarrow$ `Y[y]` r_2`[a] pc+32[].` $\#t=3$

Finally, S can be split into individual top-level update schemes using rules 1–3 from the previous section. As no text is expanded on any of the right-hand sides the 'empty' locator expressions `pc+32[]` are discarded for better readability.

[2.15] `popsh(UP:TOP,UP:TOP S1)` $\Longrightarrow$. $\#$ *rule 1*
`popsh(S1,S2) PC[pc] pc+16[POSTINC` r_1`]` r_1`[b] b[v]c` $\Longrightarrow$ `X[x]` r_1`[c].` $\#$ *rule 2*
`popsh(S2,S3) PC[pc] pc+24[PREDEC` r_2`]` $r_2[b_1]$ $a[v_1]b_1 \Longrightarrow$ `Y[y]` r_2`[a].` $\#$ *rule 2*
`popsh(UP:TOP S3,UP:TOP) PC[pc] pc[ADD` r_3`] X[x] Y[y]` $\Longrightarrow r_3$`[r] PC[pc + 32].` $\#$ *rule 3*

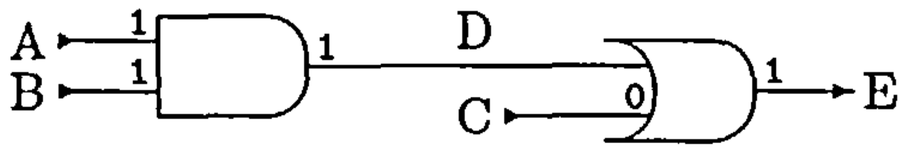


Figure 6.1: A simple logic circuit

Since in our example r_1 and r_2 are identical registers, r_2 can be substituted by r_1. Also, based on the resolution set derived in 2.13, v, v_1 and r variables are replaced by x, y and x + y respectively.

[2.16] popsh(UP:TOP, UP:TOP S1) $\Longrightarrow$.
popsh(S1, S2) PC[pc] pc+16[POSTINC r_1] r_1[b] b[x]c $\Longrightarrow$ X[x] r_1[c].
popsh(S2, S3) PC[pc] pc+24[PREDEC r_1] r_1[b_1] a[y]b_1 $\Longrightarrow$ Y[y] r_1[a].
popsh(UP:TOP S3, UP:TOP) PC[pc] pc[ADD r_3] X[x] Y[y] $\Longrightarrow$ r_3[x + y] PC[pc + 32].

To complete the example, all popsh archetypes are expanded.

[2.17] UP:SEQ[q] p[UP:TOP]q $\Longrightarrow$ UP:SEQ[q′] p[UP:TOP S1]q′.
UP:SEQ[q′] q[S1]q′ PC[pc] pc+16[POSTINC r_1] r_1[b] b[x]c $\Longrightarrow$
X[x] r_1[c] UP:SEQ[q′] q[S2]q′.
UP:SEQ[q′] q[S2]q′ PC[pc] pc+24[PREDEC r_1] r_1[b_1] a[y]b_1 $\Longrightarrow$
Y[y] r_1[a] UP:SEQ[q′] q[S3]q′.
UP:SEQ[q′] p[UP:TOP S3]q′ PC[pc] pc[ADD r_3] X[x] Y[y] $\Longrightarrow$
r_3[x + y] PC[pc + 32] UP:SEQ[q] p[UP:TOP]q.

Note that b_1 = c, and since x and y are of the same type, x = y (and b = a). Also note that the value b in r_1[b] does not change with respect to the value before application of the update scheme 2.6 on page 65.

3 Sequential archetypes

3.1 Background/Motivation

While sequential update schemes have already proved to be a good step towards more transparent and hierarchical UP specifications, they still provide only a limited improvement on the original model.

Consider the example of a simple logic circuit in figure 6.1. A basic Update Plans specification for AND and OR gates is as follows.

[3.1] and(a, b, c) = a[0] b[0] $\Longrightarrow$ c[0].
= a[0] b[1] $\Longrightarrow$ c[0].
= a[1] b[0] $\Longrightarrow$ c[0].
= a[1] b[1] $\Longrightarrow$ c[1].

or(a, b, c) = a[0] b[0] $\Longrightarrow$ c[0].
= a[0] b[1] $\Longrightarrow$ c[1].
= a[1] b[0] $\Longrightarrow$ c[1].
= a[1] b[1] $\Longrightarrow$ c[1].

To connect the output D of the AND gate to the input D of the OR gate, sequential update scheme 3.2 is defined.

[3.2] $^{S}\Big|_{2}^{1}$ and(A, B, D)
or(D, C, E).

In the situation shown in figure 6.1 only one of 16 possible expansions is applicable

[3.3] $^{S}\Big|_{2}^{1}$ A[1] B[1] $\Longrightarrow$ D[1] *# AND gate*
D[1] C[0] $\Longrightarrow$ E[1]. *# OR gate*

This example demonstrates that archetypes can be used in sequential update schemes in the same manner as in basic update schemes. Note that it would be possible to assign delays (or other metrics) to the individual and and or gates/archetypes, and that the overall delay of the logic circuit could then be easily calculated simply by adding maximum delays occurring in every stage of the sequential update scheme.

However, the main limitation of using basic archetypes in sequential update schemes is that sequences would be unduly complicated to encapsulate into non-sequential archetypes. Not only do sequential archetypes address this issue, but they also make it possible to explicitly predetermine locations of sequential archetypes' updates in the hierarchy of the top-level sequential update scheme these archetypes are called from. They also provide the formalism with additional information-hiding facility with the help of which modular specifications can easily be defined.

3.2 Syntax

The syntax of *sequential archetype* definitions is that of ordinary archetype definitions where the ⟨*update*⟩ (see page 58) appearing in the archetype body is a sequential block. The syntax of archetype calls is the same for all types of archetypes. Using typesetting possibilities a sequential archetype a is defined

$$a(params) = {}^{SEQ}\Big|_{\mathtt{m}}^{\mathtt{a}}\ update_{\mathtt{a}}\ update_{\mathtt{b}}\ \vdots\ update_{\mathtt{m}}.$$

The new archetype grammar can be found in chapter 5, or alternatively refer to appendix A for the revised complete Extended Update Plans grammar.

3.3 Semantics

Syntacticly, sequential archetype calls are equivalent to basic archetype calls. A sequential archetype definition, on the other hand, has a sequential block in its body instead of an update scheme. The individual stages (together with their updates) of this sequential block are then

matched during expansion against corresponding stages of the sequential update scheme they are called from.

As matching is done using sequencers and stages, both of these must be ground in the archetype body and in the sequential update scheme before any expansion can take place. The sequencer/stage pair has a similar purpose to that of indices in basic archetype calls, as described in [60], i.e. to ensure the correct placement of archetype bodies during an expansion.

In a sense, sequential archetypes are not as 'dynamic' as normal archetypes, and serve only as a framework to place update schemes in their predetermined locations of a sequential update scheme. Nevertheless, this is still a powerful concept, examples of which will be shown in chapter 7.

Expansion

If

$$(\lambda x.(^{SEQ}\Big|_{\begin{array}{ll}\mathtt{a_1} & update_{a_1}\\ \mathtt{a_2} & update_{a_2}\\ & \vdots\\ \mathtt{a_m} & update_{a_m}\end{array}})(a(params))$$

is equivalent to a (top-level or nested) sequential update scheme $\mathcal{S}$ containing one or more calls of the archetype $a(params)$ and the definition of this archetype is

$$a(params) = {}^{SEQ}\Big|_{\begin{array}{ll}\mathtt{b_1} & update_{\mathtt{b_1}}\\ \mathtt{b_2} & update_{\mathtt{b_2}}\\ & \vdots\\ \mathtt{b_n} & update_{\mathtt{b_n}}\end{array}}$$

then the result of expanding the archetype in $\mathcal{S}$ is

$$^{SEQ}\Big|_{\begin{array}{ll}\mathtt{c_1} & update_{\mathtt{c_1}}\\ \mathtt{c_2} & update_{\mathtt{c_2}}\\ & \vdots\\ \mathtt{c_o} & update_{\mathtt{c_o}}\end{array}}$$

where $\{\mathtt{c_1}, \mathtt{c_2}, \ldots, \mathtt{c_o}\} = \{\mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_m}\} \cup \{\mathtt{b_1}, \mathtt{b_2}, \ldots, \mathtt{b_n}\}$, and the individual $update_{\mathtt{c}_j}$ updates, where $j \in \{1, \ldots, \mathtt{o}\}$ are constructed as described in the following four sections.

As has already been shown in the introduction to this section, expansion of non-sequential archetypes in sequential blocks is allowed and they expand in the same manner as they do in the original UP.

3.3.1 Non-matching sequencers

If the archetype's sequencer does not match any of the sequencers of the top-level sequential update scheme it is called from, it is matched against the sequencer of the closest sequence in which the archetype call $a(params)$ appears—i.e. the lowest level sequence containing the

archetype call. Note that due to the syntactic sugar introduced in section 3.6.2, this is in effect an in situ expansion.

3.3.2 Non-matching stages

If during the archetype expansion any $a(params)$ b-stage does not match any of the (sequencer matched) sequential update scheme's $\mathcal{S}$ stages, it is added temporally after the previous stage of $\mathcal{S}$.

Example 2

The sequential archetype a

$$\mathtt{a}() = \begin{array}{|ll} \mathtt{ID}|_3 & update_3 \\ |_2 & update_2. \end{array}$$

in a sequential update scheme

$$\mathtt{ID}|_1 \quad \mathtt{a}()\ lhs \Longrightarrow rhs.$$

expands as

$$\begin{array}{|lll} \mathtt{ID}|_1 & lhs \Longrightarrow rhs & \\ |_2 & update_2 & \#1 < 2 \\ |_3 & update_3. & \#2 < 3 \end{array}$$

Although it might seem attractive to preserve textual ordering as it is in the sequential block of an archetype, this is not the primary aim. The primary aim is an accurate placement of a sequential archetype's updates in a sequence regardless of the order in which sequential archetypes expand (resulting in the normal form). This kind of expansion is also more consistent with the way placement of matching updates is performed.

3.3.3 $update_{\mathtt{a}}$ and $update_{\mathtt{b}}$ are alternatives

In case sequencer and stage matched updates $update_{\mathtt{a}}$ and $update_{\mathtt{b}}$ are both alternatives, the expansion mechanism is identical to the expansion mechanism in basic Update Plans. Consider the following sequential archetype and update scheme containing its calls.

$$a(params) = \begin{array}{|ll} SEQ|_1 & ltxt_1\ lc_1 =\![\ g_1\ ]\!\Rightarrow rtxt_1\ rc_1 \\ |_2 & ltxt_2\ lc_2 =\![\ g_2\ ]\!\Rightarrow rc_2 \\ & \vdots \\ |_{\mathtt{m}} & ltxt_{\mathtt{m}}\ lc_{\mathtt{m}} =\![\ g_{\mathtt{m}}\ ]\!\Rightarrow rtxt_{\mathtt{m}}\ rc_{\mathtt{m}}. \end{array}$$

$$SEQ\begin{cases} 1 & s\_lhs_1\ a(params) =\!\{\ s\_g_1\ \}\!\Rightarrow a(params)\ s\_rhs_1 \\ 2 & a(params)\ s\_lhs_2 =\!\{\ s\_g_2\ \}\!\Rightarrow s\_rhs_2 \\ & \vdots \\ n & s\_lhs_n \qquad\qquad =\!\{\ s\_g_n\ \}\!\Rightarrow s\_rhs_n. \end{cases}$$

Note the intentional omission of an archetype call $a(params)$ on the right-hand side of the alternative in stage 2. The archetype call can be omitted on the right/left-hand side provided the corresponding right/left expansion of the sequential archetype is empty. This is equivalent to the notion of left/right-handed archetypes introduced in [60].

Provided m does not match any stage in the sequential update scheme and m > n the expansion is

$$SEQ\begin{cases} 1 & s\_lhs_1\ ltxt_1\ lc_1 =\!\{\ s\_g_1 \wedge g_1\ \}\!\Rightarrow rtxt_1\ s\_rhs_1\ rc_1 \\ 2 & ltxt_2\ s\_lhs_2\ lc_2 =\!\{\ s\_g_2 \wedge g_2\ \}\!\Rightarrow s\_rhs_2\ rc_2 \\ & \vdots \\ n & s\_lhs_n \qquad\qquad =\!\{\ s\_g_n\ \}\!\Rightarrow s\_rhs_n \\ m & ltxt_m\ lc_m \qquad\qquad =\!\{\ g_m\ \}\!\Rightarrow rtxt_m\ rc_m. \end{cases}$$

Note that the text in stage m does not require any archetype call to be present since there is no matching stage m in the sequential update scheme and the stage is simply added as described earlier in section 3.3.2.

Although alternatives, all a and b *update*s in this section were in fact only plain update schemes. The expansion of true alternatives, however, is no more complicated than the expansion presented here as every sequential update block containing alternatives can be replaced by multiple instances of the sequential block containing only update schemes as was described in section 2.4.

3.3.4 $update_a$ or $update_b$ is a parallel or a sequential block

If one of sequencer and stage matched updates $update_a$ or $update_b$ is a parallel or a sequential block, the newly constructed $update_c$ will consist of both updates arranged in parallel in the matching stage. For example, see the expansion of the xor archetype in example 4 in section 3.6.2.

3.4 Special types of sequential archetypes

3.4.1 Ambidextrous sequential archetypes

Ambidextrous sequential archetypes have a slightly different semantics to ambidextrous archetypes from basic Update Plans. Again, consider the definition of the archetype *a* and the sequential update scheme *SEQ* containing its calls.

$$a(params)\ text = {}^{SEQ}\Big|_{\mathtt{m}}^{\mathtt{1}}\ ltxt_{\mathtt{1}}\ lc_{\mathtt{1}} =\!\{\ g_{\mathtt{1}}\ \}\!\Rightarrow rtxt_{\mathtt{1}}\ rc_{\mathtt{1}}$$
$$\vdots$$
$$ltxt_{\mathtt{m}}\ lc_{\mathtt{m}} =\!\{\ g_{\mathtt{m}}\ \}\!\Rightarrow rtxt_{\mathtt{m}}\ rc_{\mathtt{m}}.$$

$${}^{SEQ}\Big|_{\mathtt{n}}^{\mathtt{1}}\ s\_lhs_{\mathtt{1}}\ a(params) =\!\{\ s\_g_{\mathtt{1}}\ \}\!\Rightarrow a(params)\ s\_rhs_{\mathtt{1}}$$
$$a(params)\ s\_lhs_{\mathtt{2}} =\!\{\ s\_g_{\mathtt{2}}\ \}\!\Rightarrow s\_rhs_{\mathtt{2}}$$
$$\vdots$$
$$s\_lhs_{\mathtt{n}} =\!\{\ s\_g_{\mathtt{n}}\ \}\!\Rightarrow s\_rhs_{\mathtt{n}}.$$

Ambidextrous sequential archetypes have an additional *text*, which is substituted for every instance of its call. This text precedes any text expanded as a result of expansion of left/right-hand side of an update scheme in a particular stage. Apart from this, the expansion mechanism of ambidextrous sequential archetypes is equivalent to the expansion mechanism of sequential archetypes described in the previous section.

Again, provided m does not match any stage in the sequential update scheme and $\mathtt{m} > \mathtt{n}$ the expansion is

$${}^{SEQ}\Big|_{\mathtt{m}}^{\mathtt{1}}\ s\_lhs_{\mathtt{1}}\ text\ ltxt_{\mathtt{1}}\ lc_{\mathtt{1}} =\!\{\ s\_g_{\mathtt{1}} \wedge g_{\mathtt{1}}\ \}\!\Rightarrow text\ rtxt_{\mathtt{1}}\ s\_rhs_{\mathtt{1}}\ rc_{\mathtt{1}}$$
$$text\ ltxt_{\mathtt{2}}\ s\_lhs_{\mathtt{2}}\ lc_{\mathtt{2}} =\!\{\ s\_g_{\mathtt{2}} \wedge g_{\mathtt{2}}\ \}\!\Rightarrow s\_rhs_{\mathtt{2}}\ rc_{\mathtt{2}}$$
$$\vdots$$
$$s\_lhs_{\mathtt{n}} =\!\{\ s\_g_{\mathtt{n}}\ \}\!\Rightarrow s\_rhs_{\mathtt{n}}$$
$$ltxt_{\mathtt{m}}\ lc_{\mathtt{m}} =\!\{\ g_{\mathtt{m}}\ \}\!\Rightarrow rtxt_{\mathtt{m}}\ rc_{\mathtt{m}}.$$

3.4.2 Command sequential archetypes

Command archetypes in general are a special case of ambidextrous archetypes and their semantics has already been explained in section 2.3 on page 59.

3.5 Parameters

Parameters of sequential and parallel archetypes have to be ground expressions before archetype expansion. The primary reason for this restriction is that the structure of a sequential block that is to be expanded must be known before transformation into canonical form which is an important part of the implementation of sequential update schemes.

Example 3

The following archetype definition is perfectly valid as long as n is known (ground) before archetype expansion and $\mathtt{n} > 0$.

$$\mathtt{a}(0) = .$$
$$\mathtt{a}(\mathtt{n}) = {}^{\mathtt{s}}|_{\mathtt{n}}\ \mathtt{b}(\mathtt{n})\ \mathtt{a}(\mathtt{n}-1).$$

Parameter resolution of archetype calls inside a sequential block is slightly complicated by the fact that variables are shared across individual stages of the sequential block application and the parameter resolution set has to reflect this.

3.6 Syntactic sugar

3.6.1 Update schemes in sequential and parallel blocks

If the right-hand side of an update scheme in a sequential or parallel block is empty and its guard is true, then the transition symbol '$\Longrightarrow$' can be omitted.

3.6.2 An update is a sequential update

Every update *update* (excluding the updates which form archetype bodies) can be viewed as syntactic sugar for an anonymous one-stage sequential update $|\,$*update*.

The impact of this syntactic sugar is bigger than it may at first sight seem. It is significant as sequential archetypes do not necessarily have to be called from within sequential update schemes. The intention is to use this sugar in conjunction with the rule from section 3.3.1.

Example 4

The following update plan defines a half adder.

$$\texttt{xor(x,y,s)}\ \texttt{and(x,y,c)} \Longrightarrow .\ \#\ \textit{half adder}$$

It can be viewed as syntactic sugar for

$$|\ \texttt{xor(x,y,s)}\ \texttt{and(x,y,c)} \Longrightarrow .\ \#\ \textit{half adder}$$

Given an archetype definition

$$\texttt{xor(x,y,s)} = \left|\begin{array}{ll} 1 & \|\texttt{not(x,}w_2\texttt{)} \\ & \|\texttt{not(y,}w_1\texttt{)} \\ 2 & \|\texttt{and(x,}w_1\texttt{,}w_3\texttt{)} \\ & \|\texttt{and(}w_2\texttt{,y,}w_4\texttt{)} \\ 3 & \|\texttt{or(}w_3\texttt{,}w_4\texttt{,s).} \end{array}\right.$$

expanding the xor archetype in the definition of a half adder using the rule from section 3.3.4 gives

$$\left|\left\|\begin{array}{ll} 1 & \|\texttt{not(x,}w_2\texttt{)} \\ & \|\texttt{not(y,}w_1\texttt{)} \\ 2 & \|\texttt{and(x,}w_1\texttt{,}w_3\texttt{)} \\ & \|\texttt{and(}w_2\texttt{,y,}w_4\texttt{)} \\ 3 & \|\texttt{or(}w_3\texttt{,}w_4\texttt{,s)} \\ \multicolumn{2}{l}{\texttt{and(x,y,c).}} \end{array}\right.\right. \quad \text{which can be transformed as} \quad \left|\begin{array}{ll} 1 & \|\texttt{and(x,y,c)} \\ & \|\texttt{not(x,}w_2\texttt{)} \\ & \|\texttt{not(y,}w_1\texttt{)} \\ 2 & \|\texttt{and(x,}w_1\texttt{,}w_3\texttt{)} \\ & \|\texttt{and(}w_2\texttt{,y,}w_4\texttt{)} \\ 3 & \|\texttt{or(}w_3\texttt{,}w_4\texttt{,s).} \end{array}\right.$$

3.7 Limitations

With the introduction of sequential update schemes a new set of problems needs to be addressed. The most obvious one stems from allowing variables to be shared among all stages of a sequential block. Consider example 5.

Example 5

This rather forced example uses two sequential archetypes and a (sugared) sequential update scheme which calls these archetypes. One of these archetypes (r) reads a value of variable v, and the second one (w) writes a value of variable v.

$$\mathtt{r(v)} = {}^{\mathtt{S}}|_{2}\ \mathtt{A[v]}.$$
$$\mathtt{w(v)} = {}^{\mathtt{S}}|_{1}\ \Longrightarrow \mathtt{B[v]}.$$
$$\mathtt{r(v)} \Longrightarrow \mathtt{w(v)}.$$

Clearly, the expansion of these archetypes the results in a "forward reference" to the variable v (v is not ground in stage 1), which is not a valid specification.

$$\begin{array}{|ll} 1 & \Longrightarrow \mathtt{B[v]} \\ 2\ \mathtt{A[v]} & \Longrightarrow . \end{array}$$

However, problems such as these are not exclusive to Extended UP, they were also present in basic UP, and can easily be detected by a trivial data flow analysis [1]. Any update scheme containing such a forward reference is illegal.

4 Special cases of archetype expansion

4.1 Alternatives

As a result of syntactic changes, alternatives can now (in Extended Update Plans) be bodies of archetypes. A single update scheme in which such an archetype is called will, assuming there are n update schemes in the archetype's alternative, expand into n update schemes. In other words, the archetype definition of such an archetype can be viewed as n archetype definitions of the same archetype containing update schemes with their guards adapted as shown by the transformation τ_5 in section 2.4. Again, consistency is the primary reason for introducing alternatives into archetype's bodies.

4.2 Parallel blocks

The original Update Plans formalism does not allow the use of parallel blocks in archetypes. While this may seem unnecessary in the original Update Plans, in Extended Update Plans

parallel blocks can appear in archetypes not only as a part of a sequential update scheme, but also entirely on its own. This increases consistency and adds a degree of modularity and information hiding to the formalism.

An archetype definition

$$a(params) = \left\| \begin{array}{l} update_1 \\ \quad \vdots \\ update_n. \end{array} \right.$$

is syntactic sugar for

$$a(params) = \left| \left\| \begin{array}{l} update_1 \\ \quad \vdots \\ update_n. \end{array} \right. \right.$$

that is the same parallel block is embedded in a one-stage anonymous sequence. Thanks to the existence of the expansion rule introduced in section 3.3.1 and the syntactic sugar in section 3.6.2, the expansion of this kind of archetype is governed by the rule in section 3.3.4, which is thus in effect an in situ expansion.

5 Conclusions

In this chapter sequential update schemes and sequential archetypes were introduced as semantic extensions to basic Update Plans. They contribute to the usability of the formalism in two major areas.

Firstly, the obvious area of synchronisation. Synchronisation, or more precisely explicit temporal ordering of update schemes, was already possible in basic Update Plans. However, this required an introduction of artificial constructs unrelated to the architecture under description and made the whole specification less readable and elegant. Sequential update schemes augment the non-deterministic model of execution of basic Update Plans by explicitly stating the order in which updates will be applied and expressing a number of consecutive updates as one.

Secondly, the introduction of sequential archetypes extended the possibility for information hiding and structure reuse by encapsulating a series of synchronised updates rather than just a single atomic update. As a series of actions can be encapsulated into a module (a sequential archetype), it is possible to provide multiple definitions for exactly the same series of actions, but perhaps on a different level of abstraction. The next step is to design a mechanism proving equivalence of these levels or ideally a refinement methodology to automatically derive provably correct levels of description.

Overall, Extended Update Plans present a real improvement in readability of descriptions where sequential behaviour is required. These may include fetch/execute cycle and lower-level descriptions such as gate-level models of (a)synchronous circuits.

Chapter 7

PRAM

To demonstrate the power, compactness and intuitiveness of the use of sequential update schemes and archetypes, a specification of the parallel random access machine (PRAM) has been developed. The PRAM specification has been chosen in particular not only because of its historical importance as one of the first models of parallel computing [22], but also because of its ongoing relevance and the interest of researchers in this model [2, 40, 46, 51, 76].

A PRAM is characterised by its "memory models" which determine the PRAM's behaviour when two or more random access machines (RAMs) attempt to read from or write to the same memory all. An informal description of these memory models is given in section 1. Section 2 gives an EUP specification of the memory models for PRAMs with two RAMs. More general n-RAM PRAM memory models are defined in section 3. The EUP specifications are used in section 4 where they are placed in the context of the remainder of a PRAM's instruction set. Conclusions are drawn in section 5. The full specification for n-PRAM can be found in appendix E.

1 Informal description

The parallel random access machine is a theoretical uniform memory access shared memory model (UMA/SMP) [35] of parallel computing. This description and the formal specification is based on the informal PRAM specification given in [30]. PRAM consists of n random access machines [67, 72] with infinite shared memory and a common clock. Each of the RAMs can access shared memory independently from any other RAM in constant time.

Figure 7.1 shows a RAM consisting of R general-purpose registers, a program counter (`PC`), a signature register (`SIG`), an accumulator (`ACC`) and a memory address register (`MAR`). Note that the signature register is only used in the priority model, which will be explained later in this section.

A PRAM's instruction set can be divided into four classes: arithmetic/logic instructions, load/store instructions, flow control instructions and read/write instructions. These instruc-

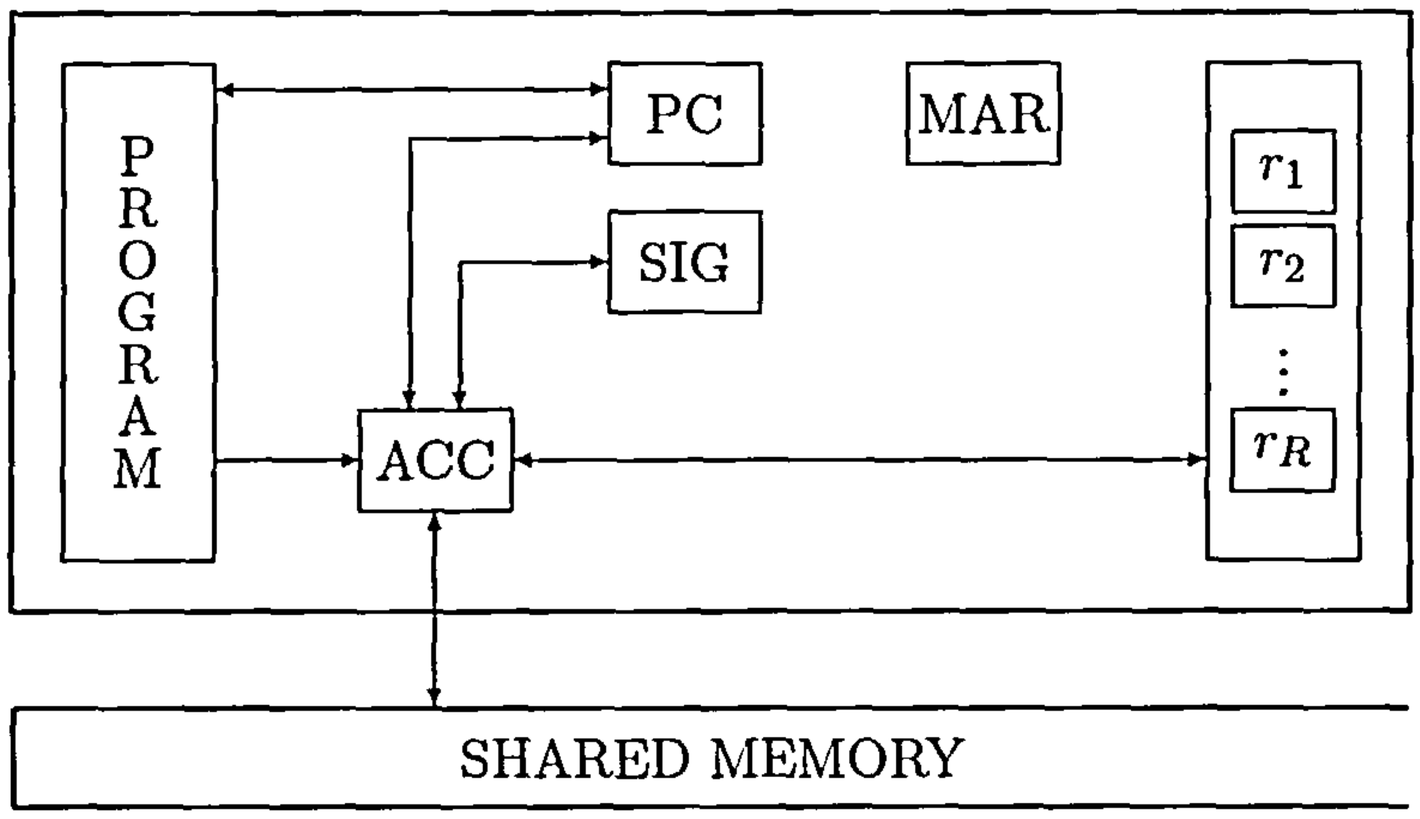


Figure 7.1: The structure of a PRAM's RAM and data flow inside it

tions are specified in section 4, but it is worth noting that only the read/write instructions access shared memory.

All RAMs execute the same program, but each one can be executing a separate code segment within the program. Execution of any instruction on the PRAM machine takes one clock cycle, i.e. a constant unit of time. One clock cycle is divided into four sequential phases (in a sense a f/e cycle) which are synchronous between processors (RAMs). In the first phase program counters of all processors that are not yet halted are increased to point to the following instruction. The second phase is a register read/write access phase and instruction execution phase. A read/write instruction will, on execution, prepare for shared memory access in the next two phases by loading the memory address register. All other instructions will update the contents of local registers. Finally the third and the fourth phases are shared memory read and write access phases respectively.

Depending on whether simultaneous reads of a same shared memory cell are allowed, two read models, the concurrent read model (CR), and the exclusive read model (ER) can be defined.

Similarly, two memory write models exist—a concurrent write (CW), and an exclusive write (EW). However, concurrent write models need further consideration. The result of two or more processors trying to write simultaneously into a same shared memory cell c has to be defined. There are many different write conflict resolution rules. Some of the most common ones are as follows.

WEAK Simultaneously writing the value zero to c by two or more RAMs is allowed and the value zero is stored into the cell. Simultaneously writing any other value to the cell by

two or more RAMs is forbidden and the execution of the PRAM ends in a write conflict if such a write is attempted.

COMMON Simultaneously writing a common value to c by two or more RAMs is allowed and the common value is stored into the cell. Writing two or more different values simultaneously to c by two or more RAMs is forbidden and the execution of the PRAM ends in a write conflict if such a write is attempted.

TOLERANT If two or more RAMs simultaneously try to write to c, then the value of the cell is not changed. The value of c is changed only if just one RAM is writing to it at the time.

COLLISION If two or more RAMs simultaneously try to write to c, then a special collision symbol (`COLL`) is written to the cell, even if they are writing the same value.

COLLISION+ If two or more RAMs simultaneously try to write two or more different values to c, then a special collision symbol (`COLL`) is written to the cell. The value of c is changed normally if the RAMs are writing the same value.

ARBITRARY If two or more RAMs simultaneously try to write to c, then an arbitrarily chosen RAM writing to the cell succeeds in writing. There is no way to determine prior to the write which RAM will succeed, or determine after the write which RAM succeeded.

PRIORITY If two or more RAMs simultaneously try to write to c, then the RAM with the smallest RAM identifier, i.e. with the smallest value of the `SIG` register, succeeds in writing. In other words, RAM identifiers define an unequivocal and RAM-wise order of priority of RAMs so that the RAM with the smallest RAM identifier has the highest priority.

2 2-PRAM memory models

The aim of this section is to show a compact and elegant specification of a 2-RAM PRAM using EUP. This specification will serve as the basis for an n-PRAM specification presented in the following section. The EUP RAM instruction set specification is shared by 2- and n-PRAM specifications, and can be found in section 4.

As already mentioned in the previous section, the execution of an instruction on a PRAM machine is divided into four stages. The following sequential update scheme defines three of these stages. Phase two of the f/e cycle (the register read/write phase) is defined in terms of the `instr` archetype which is part of the definition of the `pc` archetype—i.e. when the `instr` archetype is expanded, appropriate actions for (not only) phase two of the f/e cycle will be added to the specification. The `pc`, `shr`, and `shw` archetypes are defined later in this section.

[2.1] $^{\texttt{FE}}\Big|_{4}^{1,3}$ `pc(1) pc(2)` *# update PCs, block (stage 2) if neither P1 or P2 is running*
`shr()` *# read shared memory*
`shw().` *# write shared memory*

The `pc` archetype in 2.2 increases program counters on both RAMs. `C` is the number of instructions in the program each of the RAMs are executing. A RAM having the value of its program counter `pc` $<$ `0` or `pc` $\geq$ `C` is halted. The constants `P1` and `P2` are used to distinguish between processor (RAM) 1 and 2. Note the use of the symbol '~' to expand these constants. For example, for `P` = 1, `P~p` expands to `P~1` which is equivalent to `P1`. The 'empty' update scheme in stage two of the `pc` archetype prevents unwanted blocking behaviour of the `FE` sequence caused by the absence of the register access stage.

[2.2] $\texttt{pc(p)} = {}^{\texttt{FE}}\Big|_{2}^{1}$ $\texttt{P~p:PC[pc] pc[instr(p)]qc} \Rightarrow \{\ \texttt{0} \leq \texttt{pc} < \texttt{C}\ \} \Rightarrow \texttt{P~p:PC[qc]}$ *# running*
$\Longrightarrow$. *# don't block*
$\texttt{pc(p)} = \texttt{P~p:PC[pc]} \Rightarrow \{\ \texttt{0} > \texttt{pc} \vee \texttt{pc} \geq \texttt{C}\ \} \Rightarrow$. *# not running*

In order to make the specification more concise, operators $\ominus$ and $\oslash$ are introduced.

$\{a_1 \ominus a_2 \mid a_1 = a_2 \wedge a_1 \neq \texttt{NULL}\}$
$\{a_1 \oslash a_2 \mid a_1 \neq a_2 \vee a_1 = \texttt{NULL} \vee a_2 = \texttt{NULL}\}$

Memory read conflicts occur only if the PRAM's memory read model (given by the memory read model register `MR`) is ER, and both RAMs are trying to read the same shared memory cell. The `shr` (shared memory read) archetype is defined to read values from shared memory to the processors' accumulators, and to act as a barrier to halt the entire ER PRAM should a read conflict occur.

[2.3] $\texttt{shr()} = \texttt{PRAM:MR[mr] P1:MAR[a}_1\texttt{] P2:MAR[a}_2\texttt{]} \Rightarrow \{\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mr} = \texttt{ER}\ \} \Rightarrow \texttt{P1:PC[C] P2:PC[C];}$
" $\Longrightarrow \texttt{read(1, a}_1\texttt{) read(2, a}_2\texttt{).}$

If no read conflicts occur and the current instruction is a memory read instruction, a value is read from shared memory into a RAM's accumulator by the following archetype.

[2.4] $\texttt{read(p, a)} = \texttt{a[v]} \Rightarrow \{\ \texttt{a} \neq \texttt{NULL}\ \} \Rightarrow \texttt{P~p:ACC[v];}$ *# read value v from shared memory*
$\Longrightarrow$. *# not a memory read instruction*

The `MAR` on every RAM contains a special value `NULL` when the currently executed instruction doesn't access shared memory.

The `shw` performs similar function to the `shr` archetype, but it writes values into shared memory. The PRAM's memory write model is given by the memory write model register (`MW`).

$$
\begin{aligned}
[2.5]\ \texttt{shw}() &= \texttt{PRAM:MW[mw] P1:MAR[a}_1\texttt{] P2:MAR[a}_2\texttt{] P1:ACC[v}_1\texttt{] P2:ACC[v}_2\texttt{] P1:SIG[s}_1\texttt{] P2:SIG[s}_2\texttt{]} \\
&\quad =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{EW}\ ]\!\Rightarrow \texttt{P1:PC[C] P2:PC[C]}. \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_WEAK} \wedge ((\texttt{v}_1 = 0) \wedge (\texttt{v}_2 = 0))\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, 0). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_WEAK} \wedge ((\texttt{v}_1 \neq 0) \vee (\texttt{v}_2 \neq 0))\ ]\!\Rightarrow \texttt{P1:PC[C] P2:PC[C]}. \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_COMMON} \wedge \texttt{v}_1 = \texttt{v}_2\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{v}_1). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_COMMON} \wedge \texttt{v}_1 \neq \texttt{v}_2\ ]\!\Rightarrow \texttt{P1:PC[C] P2:PC[C]}. \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_TOLERANT}\ ]\!\Rightarrow . \ \#\ \textit{value not changed} \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_COLLISION}\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{COLL}). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_COLLISIONP} \wedge \texttt{v}_1 = \texttt{v}_2\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{v}_1). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_COLLISIONP} \wedge \texttt{v}_1 \neq \texttt{v}_2\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{COLL}). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_ARBITRARY}\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{v}_1). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_ARBITRARY}\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{v}_2). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_PRIORITY} \wedge \texttt{s}_1 < \texttt{s}_2\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{v}_1). \\
&= \text{"} =\![\ \texttt{a}_1 \ominus \texttt{a}_2 \wedge \texttt{mw} = \texttt{CW\_PRIORITY} \wedge \texttt{s}_1 > \texttt{s}_2\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{v}_2). \\
&= \text{"} =\![\ \texttt{a}_1 \oslash \texttt{a}_2\ ]\!\Rightarrow \texttt{write}(\texttt{a}_1, \texttt{v}_1)\ \texttt{write}(\texttt{a}_2, \texttt{v}_2). \ \#\ \textit{no conflict}
\end{aligned}
$$

Note that writing the constant C into a p's program counter register is effectively a HALT instruction.

If the addresses in MAR registers are not the same, there are clearly no write conflicts and both values are written to shared memory using the following archetype.

$$
\begin{aligned}
[2.6]\ \texttt{write}(\texttt{a}, \texttt{v}) = &=\![\ \texttt{a} \neq \texttt{NULL}\ ]\!\Rightarrow \texttt{a[v]};\ \#\ \textit{write value v into shared memory} \\
&\Longrightarrow . \qquad\qquad\qquad\qquad \#\ \textit{not a memory write instruction}
\end{aligned}
$$

3 n-PRAM memory models

This section has a specification for an n-RAM PRAM. It is a relatively high-level specification abstracting away implementation details and giving only two memory write models. The complete EUP specification can be found in appendix E.

We first define the instruction cycle, as we did for the 2-PRAM. This time the PRAM is defined by the archetype pram (3.1), rather than just a sequential update scheme as for the 2-PRAM (2.1). The number of processors and memory read/write models are parameters of the specification rather than of the machine configuration. Again, only three out of four stages are included in the FE sequence, as the second (execution) stage of the instruction execution will be expanded by the pc archetype which was defined in section 2. As will be explained later, the enforced blocking behaviour when no RAM is running is intentional. The pcs, shr and shw archetypes now take additional arguments. The variable n is the number of RAMs in the PRAM machine ($n > 0$), rm and wm are the memory read and write models respectively. For example, a 5-PRAM with concurrent read and common concurrent write would be specified by pram(5, CR, CW_COMMON).

$$[3.1]\ \texttt{pram(n,rm,wm)} = {}^{\texttt{FE}}\Big|\begin{array}{ll}1\ \texttt{pcs(n)} & \#\ \textit{update PCs, block (stage 2) if no RAM is running}\\ 3\ \texttt{shr(n,rm)} & \#\ \textit{shared memory reads}\\ 4\ \texttt{shw(n,wm).} & \#\ \textit{shared memory writes}\end{array}$$

As mentioned at the start of this section, this specification is relatively abstract. It uses the 'limits' notation (e.g. in 3.2) to define 'multi-processor' archetypes, and the (multi)set notation (e.g. in 3.6) to simplify guards. This notation can easily be implemented as recursive archetypes as shown in appendix E. For example, the pcs archetype (which expresses the first stage of an instruction execution) can be defined using the limits notation as

$$[3.2]\ \texttt{pcs(n)} = \overset{\texttt{n}}{\underset{p=1}{\texttt{pc}(p)}}.$$

which is shorthand for 'pc(1) pc(2) $\cdots$ pc(n).'. The same archetype in 3.3 is an example of a recursive implementation. It increases the PC on every running RAM p using the archetype 2.2 from the previous section. If there are no running RAMs, execution is blocked (the FE sequence is only partially applicable), and the whole PRAM stops.

$$[3.3]\ \texttt{pcs(0)} = .\ \#\ \textit{0 RAMs}$$
$$\texttt{pcs(p)} = \texttt{pc(p) pcs(p}-1\texttt{)} =\!\!\{\ \texttt{p} > 0\ \}\!\!\Rightarrow .$$

The following halt archetype serves to stop a specific RAM and also the whole PRAM. This archetype is used by shr/shw archetypes in terminal conflict models when a read/write conflict occurs.

$$[3.4]\ \texttt{halt(p)} = \Longrightarrow \texttt{P\~{}p:PC[C]}.\ \#\ \overset{\texttt{n}}{\underset{p=1}{\texttt{halt}(p)}}\ \textit{stops the whole PRAM}$$

The shr archetype checks simultaneous reads from shared memory. It uses set theory in order to discover read accesses to the same memory location. Addresses waiting to be read in shared memory are stored in each one of RAM's MAR registers. The archetype shr checks a RAM's (non-NULL) MAR register value against the values in all the other RAM's MARs, and if it finds a conflict, the entire PRAM is halted. The read archetype is defined in 2.4. The shrd archetype in 3.5 provides the address ($\texttt{a}_p$) from the memory address register of processor p.

$$[3.5]\ \texttt{shrd(p,a)} = \texttt{P\~{}p:MAR[a]}.$$

The notation $\{\!|\ \texttt{a}_p\ |\!\}$ is the bag or multiset containing the elements $\texttt{a}_p$, and an element $\texttt{a}_p^{\overline{\texttt{NULL}}} = \texttt{a}_p$ if and only if $\texttt{a}_p \neq \texttt{NULL}$, otherwise it is equivalent to an empty/non-existent element.

$$[3.6]\ \texttt{shr(n,CR)} = \overset{\texttt{n}}{\underset{p=1}{\texttt{shrd}(p,\texttt{a}_p)}}\ \overset{\texttt{n}}{\underset{p=1}{\texttt{read}(p,\texttt{a}_p)}}.$$
$$\texttt{shr(n,ER)} = \overset{\texttt{n}}{\underset{p=1}{\texttt{shrd}(p,\texttt{a}_p)}}\ \overset{\texttt{n}}{\underset{p=1}{\texttt{halt}(p)}} \quad =\!\!\left\{\left\{\!\middle|\ \texttt{a}_p^{\overline{\texttt{NULL}}}\ \middle|\!\right\}\neq\left\{\ \texttt{a}_p^{\overline{\texttt{NULL}}}\ \right\}\right\}\!\!\Rightarrow \quad \#\ \textit{read conflict}$$
$$= \overset{\texttt{n}}{\underset{p=1}{\texttt{shrd}(p,\texttt{a}_p)}}\ \overset{\texttt{n}}{\underset{p=1}{\texttt{read}(p,\texttt{a}_p)}} =\!\!\left\{\left\{\!\middle|\ \texttt{a}_p^{\overline{\texttt{NULL}}}\ \middle|\!\right\}=\left\{\ \texttt{a}_p^{\overline{\texttt{NULL}}}\ \right\}\right\}\!\!\Rightarrow .\ \#\ \textit{no read conflict}$$

A similar approach will be used to determine shared memory write conflicts, but simple detection of the fact that 1 or more write conflicts are due to occur is not sufficient. Different actions need to be taken in different memory models.

The number of RAMs (n) and the memory write model are now parameters of `shw` archetypes. The `write` archetype is defined in 2.6. The `shwr` archetype reads the address (a_p) from the memory address register of processor p and the value (v_p) to be written at this address in shared memory.

[3.7] `shwr(p,a,v)` = `P~p:MAR[a] P~p:ACC[v]`.

An archetype is defined for each memory write model. Only the exclusive write and the weak concurrent write models are described in detail here. The full specification is in the appendix E.

[3.8] `shw(n,EW)`

$$= \overset{n}{\underset{p=1}{\texttt{shwr}}}(p, a_p, \_)\ \overset{n}{\underset{p=1}{\texttt{halt}}}(p) \qquad \Rightarrow\!\left|\left\{\left|\, a_p^{\overline{\texttt{NULL}}} \,\right|\right\} \neq \left\{\, a_p^{\overline{\texttt{NULL}}} \,\right\}\right|\!\Rightarrow \quad \#\ \textit{write conflict}$$

$$= \overset{n}{\underset{p=1}{\texttt{shwr}}}(p, a_p, v_p)\ \overset{n}{\underset{p=1}{\texttt{write}}}(a_p, v_p) \Rightarrow\!\left|\left\{\left|\, a_p^{\overline{\texttt{NULL}}} \,\right|\right\} = \left\{\, a_p^{\overline{\texttt{NULL}}} \,\right\}\right|\!\Rightarrow . \ \#\ \textit{no write conflict}$$

`shw(n,CW_WEAK)`

$$= \overset{n}{\underset{p=1}{\texttt{shwr}}}(p, a_p, v_p)\ \overset{n}{\underset{p=1}{\texttt{halt}}}(p) \qquad \#\ \textit{write conflict}$$

$$\Rightarrow\!\left|\, \exists (a_p, v_p), (a_q, v_q) \in \left\{ (a_p^{\overline{\texttt{NULL}}}, v_p) \right\} \,|\, p \neq q \wedge a_p = a_q \wedge (v_p \neq 0 \vee v_q \neq 0) \,\right|\!\Rightarrow$$

$$= \overset{n}{\underset{p=1}{\texttt{shwr}}}(p, a_p, v_p)\ \overset{n}{\underset{p=1}{\texttt{write}}}(a_p, v_p) \qquad \#\ 0\ \textit{or no write conflict}$$

$$\Rightarrow\!\left|\, \forall (a_p, v_p), (a_q, v_q) \in \left\{ (a_p^{\overline{\texttt{NULL}}}, v_p) \right\} \,|\, a_p = a_q \Rightarrow (v_p = v_q = 0) \,\right|\!\Rightarrow .$$

4 Instructions

4.1 Addressing modes

4.1.1 Implicit modes

There are three implicit addressing modes on the PRAM—i.e. addressing modes that are implicit in the opcode, rather than being explicitly provided as an operand. Access to these addressing modes always takes place in the register read/write phase. The three implicit addressing modes are accumulator addressing,

[4.1] `accr(p,v)` = $^{\texttt{FE}}|_2$ `P~p:ACC[v]`. # *accumulator read*

`accw(p,v)` = $^{\texttt{FE}}|_2$ $\Longrightarrow$ `P~p:ACC[v]`. # *accumulator write*

signature register addressing

[4.2] `sigr(p,v)` = $^{FE}|_2$ `P~p:SIG[v]`. # *signature register read*
`sigw(p,v)` = $^{FE}|_2$ $\Longrightarrow$ `P~p:SIG[v]`. # *signature register write*

and program counter addressing.

[4.3] `pcr(p,v)` = $^{FE}|_2$ `P~p:PC[v]`. # *program counter read*
`pcw(p,v)` = $^{FE}|_2$ $\Longrightarrow$ `P~p:PC[v]`. # *program counter write*

4.1.2 Register modes

We first define two 'textless' register addressing archetypes that can be used to read/write values from/to local registers.

[4.4] `regr(p,r,v)` = $^{FE}|_2$ `P~p:r[v]`. # *local register read*
`regw(p,r,v)` = $^{FE}|_2$ $\Longrightarrow$ `P~p:r[v]`. # *local register write*

These archetypes can, among other things, be used to define the operand register addressing modes. These are direct register addressing mode

[4.5] `DR(p,r,v) r` = `regr(p,r,v)`. # *direct register (read)*

and indirect register addressing mode.

[4.6] `IR(p,`r_i`,v) r` = `regr(p,r,`r_i`) regr(p,`r_i`,v)`. # *indirect register (read)*

Note that archetypes in 4.5 and in 4.6 are command archetypes generating addressing mode mnemonic for a PRAM instruction. These two addressing modes are known as register modes.

[4.7] `rm(p,a,v)` = `DR(p,a,v)` # *register modes*
= `IR(p,a,v)`.

Parameter `p` is the processor number, `a` is the effective (register) address, and `v` is the value accessed. Register modes are the only addressing modes that may be used by a `STORE` instruction (the `STORE` instruction must have an effective (register) address in which to store the value—see section 4.3).

4.1.3 Immediate/Register modes

Most of the remaining instructions (jump instructions, arithmetic/logical instructions, the `LOAD` instruction) may also take an immediate value. Adding this to the register modes defines the set of immediate/register modes.

[4.8] `irm(p,v)` = `IMM v` # *immediate mode and register modes*
`rm(p,_,v)`.

4.1.4 Memory Modes

Memory read and memory write accesses can be direct or indirect. A direct memory read is defined by

$$[4.9]\ \texttt{DMR(p) m} = {}^{\texttt{FE}}\Big|_{3}^{2} \begin{array}{l} \Longrightarrow \texttt{P\~{}p:MAR[m]} \quad \# \textit{ direct memory read} \\ \Longrightarrow \texttt{P\~{}p:MAR[NULL]}. \end{array}$$

The MAR is loaded with the address in phase two (register read/write), and cleared in phase three (memory read).

The indirect memory read is

$$[4.10]\ \texttt{IMR(p) r} = {}^{\texttt{FE}}\Big|_{3}^{2\ \texttt{P\~{}p:r[m}_\texttt{i}\texttt{]}} \begin{array}{l} \Longrightarrow \texttt{P\~{}p:MAR[m}_\texttt{i}\texttt{]} \quad \# \textit{ indirect memory read} \\ \Longrightarrow \texttt{P\~{}p:MAR[NULL]}. \end{array}$$

The addressing modes in 4.9 and 4.10 are combined to form a set of memory read modes for use with the READ instruction—see section 4.5.

$$[4.11]\ \texttt{mrm(p)} = \begin{array}{l} \texttt{DMR(p)} \\ \texttt{IMR(p)}. \end{array}$$

The memory write modes are defined similarly.

$$[4.12]\ \texttt{DMW(p) m} = {}^{\texttt{FE}}\Big|_{4}^{3} \begin{array}{l} \Longrightarrow \texttt{P\~{}p:MAR[m]} \quad \# \textit{ direct memory write} \\ \Longrightarrow \texttt{P\~{}p:MAR[NULL]}. \end{array}$$

$$\texttt{IMW(p) r} = {}^{\texttt{FE}}\Big|_{3}^{2\ \texttt{P\~{}p:r[m}_\texttt{i}\texttt{]}}{}_{4} \begin{array}{l} \Longrightarrow \quad \# \textit{ indirect memory write} \\ \Longrightarrow \texttt{P\~{}p:MAR[m}_\texttt{i}\texttt{]} \\ \Longrightarrow \texttt{P\~{}p:MAR[NULL]}. \end{array}$$

These two addressing modes form the memory write mode set.

$$[4.13]\ \texttt{mwm(p)} = \begin{array}{l} \texttt{DMW(p)} \\ \texttt{IMW(p)}. \end{array}$$

A read from shared memory will take place in the memory read phase (phase three), and a write to shared memory will take place in the memory write phase (phase four). A processor will read from memory if and only if its MAR contains a non-NULL address at the onset of phase three, and write to memory if and only if its MAR contains a non-NULL address at the onset of phase four. The mrm and mwm addressing modes are designed to load the processors' MARs with non-NULL addresses at the correct stage of the f/e cycle.

4.2 Accumulator loading instructions

There are two groups of (local) accumulator loading instructions—the arithmetic/logic instructions and the load instructions. We will first define the way in which these instructions access the value to be written to the accumulator.

Arithmetic instructions

All arithmetic/logical instructions use only (local) registers (the `imr` addressing mode set), and hence take place wholly in phase two (the register read/write phase).

Arithmetic/logical operators can be distinguished as being binary operators

[4.14]
$$\begin{aligned}
&\texttt{binary}(x, y, x + y) = \texttt{ADD}. && \#\ \textit{addition}\\
&\texttt{binary}(x, y, x - y) = \texttt{SUB}. && \#\ \textit{subtraction}\\
&\texttt{binary}(x, y, x \times y) = \texttt{MUL}. && \#\ \textit{multiplication}\\
&\texttt{binary}(x, y, x/y) = \texttt{DIV}. && \#\ \textit{division}\\
&\texttt{binary}(x, y, x \,\%\, y) = \texttt{MOD}. && \#\ \textit{modulo}\\
&\texttt{binary}(x, y, x << y) = \texttt{SHIFT}. && \#\ \textit{shift left}\\
&\texttt{binary}(x, y, x \,\&\, y) = \texttt{AND}. && \#\ \textit{bitwise and}\\
&\texttt{binary}(x, y, x \mid y) = \texttt{OR}. && \#\ \textit{bitwise or}\\
&\texttt{binary}(x, y, x \,\hat{}\, y) = \texttt{XOR}. && \#\ \textit{bitwise xor}
\end{aligned}$$

or monadic operators

[4.15]
$$\begin{aligned}
&\texttt{monadic}(x, \log(x)) = \texttt{LOG}. && \#\ \textit{logarithm}\\
&\texttt{monadic}(x, \text{not}(x)) = \texttt{NOT}. && \#\ \textit{bitwise not}
\end{aligned}$$

Both binary and monadic operators access one argument from the instruction (immediate/register addressed) operand. Binary operators access their other (first) argument from the processor's accumulator.

[4.16]
$$\begin{aligned}
\texttt{arlog}(p, r) &= \texttt{binary}(x, y, r)\ \texttt{irm}(p, y)\ \texttt{accr}(p, x)\\
&= \texttt{monadic}(y, r)\ \texttt{irm}(p, y).
\end{aligned}$$

Load instructions

There are three load instructions on the PRAM. They all load the processor's accumulator with a value. We will first define the value access and corresponding opcode. Note that `LOADINDEX` and `LOADPC` are zero operand instructions.

[4.17]
$$\begin{aligned}
\texttt{load}(p, v) &= \texttt{LOAD}\ \texttt{irm}(p, v)\\
&= \texttt{LOADINDEX}\ \texttt{sigr}(p, v)\\
&= \texttt{LOADPC}\ \texttt{pcr}(p, v).
\end{aligned}$$

Writing the result

We now combine these two sets of instructions and define the write to the accumulator.

[4.18]
$$\begin{aligned}
\texttt{toacc}(p, v) &= \texttt{arlog}(p, v) \Longrightarrow \texttt{accw}(p, v)\\
&= \texttt{load}(p, v) \Longrightarrow \texttt{accw}(p, v).
\end{aligned}$$

4.3 General purpose register loading instructions

The store instruction writes the contents of the accumulator to a local register. The following update scheme defines the value and effective address access.

[4.19] store(p, r, v) = STORE rm(p, r, _) accr(p, v).

The write to the register is defined by

[4.20] toreg(p, r, v) = store(p, r, v) $\Longrightarrow$ regw(p, r, v).

4.4 Program counter loading instructions

The jump family of instructions will load a processor's program counter. The value to be loaded is defined by

[4.21] jump(p, v > 0, a) = JPOS irm(p, a) accr(p, v).
jump(p, v = 0, a) = JZERO irm(p, a) accr(p, v).
jump(p, TRUE, a) = JUMP irm(p, a). # *unconditional jump*
jump(_, TRUE, C) = HALT.

The program counter is only updated if the condition in the second parameter of jump is TRUE.

[4.22] topc(p) = jump(p, cond, a) =[cond]⇒ pcw(p, a)
= jump(p, cond, a) =[¬cond]⇒ .

4.5 Memory read instructions

The READ instruction moves a value from shared memory into a RAM's accumulator. As mentioned earlier, the values are read from shared memory in phase three of the f/e cycle by the shr archetype, which checks memory read conflicts. The read archetype only generates the instruction and addressing mode mnemonic, and moves shared memory address into the MAR for the shr archetype.

[4.23] read(p) = READ mrm(p).

4.6 Memory write instructions

The WRITE instruction stores values from a RAM's accumulator into shared memory. Similarly to the READ instruction, the actual write is done by the shw archetype, which checks and resolves memory write conflicts.

[4.24] write(p) = WRITE mwm(p).

4.7 The instruction set

Finally, all 5 types of instructions are added to the instruction set.

[4.25] `instr(p)` = `toacc(p,_)`
= `toreg(p,_,_)`
= `topc(p,_)`
= `read(p)`
= `write(p)`.

5 Conclusion

Although [61] contains a specification of a parallel machine, this was only of pipelining in a relatively simple RISC processor. This specification, on the other hand, is a simple demonstration that a massively parallel system can easily be formally described with EUP. It could also be relatively easily adapted to describe more recent PRAM models such as [76].

While a similar specification of the PRAM machine would be possible using basic Update Plans, this could only be done at a more concrete level of description by enforcing sequential behaviour through constructs unspecified by the PRAM model—i.e. effectively preempting design decisions about the implementation of parallelism which the PRAM model was developed to avoid. The Extended Update Plan specification preserves the level of abstraction of the PRAM model, with the `shr` and `shw` archetypes containing all the characteristics of the various access models.

In addition the Extended Update Plan specification encapsulates most of the PRAM instructions in single sequential archetype definitions—corresponding to the descriptions with which a user would be familiar—even though the effects of these instructions take place across several clock cycles. This greatly simplifies the task of verification between multiple levels of specifications as a direct correspondence between these levels can easily be found.

Chapter 8

Other Methods

This chapter provides an overview of the most frequently used formal methods in the area of formal specification of hardware architectures. It is almost impossible to provide a complete list of methods that serve this purpose. For a fuller picture the reader is referred to comprehensive surveys in [18, 29, 42, 52, 68, 80], where methods for specification and verification, not only of computer hardware, have been described.

A group of specification methods (e.g. [31]) which is not considered in this chapter is based on process algebras. Although they are based on rigorous, well developed mathematical theories providing a variety of techniques for proving and verifying properties, this approach is still in its infancy and readability of such specifications is generally poor.

Several examples that demonstrate most of the formalisms and compare them to UP are given. Although the examples are too basic to exercise a wide range of language features, or to give a true test of ease of expressibility, they should provide some insight into the formalisms.

1 Specification methods

It is difficult to separate various formalisms into distinct classes as some of them overlap and every generalisation is bound to introduce some inaccuracies. Nevertheless, they will be categorised here by their area of predominant use.

1.1 Hardware

Hardware Description Languages (HDLs) have been used in the industry since the 1960s to document and simulate designs, mainly at the circuitry level of machine architectures. The most widely used HDLs are VHDL, Verilog and ELLA.

Verilog	Update Plans
`module half_adder(a, b, s, c);`	`half_adder(a, b, s, c) = xor(a, b, s) and(a, b, c).`
`  input a, b;`	
`  output s, c;`	
`  xor g1(a, b, s);`	
`  and g2(a, b, c);`	
`endmodule`	

Figure 8.1: Verilog vs. UP

1.1.1 VHDL

VHDL [3, 36] stands for VHISC (Very High Speed Integrated Circuit) HDL—an international IEEE standard (1987) specification language for describing digital hardware.

Each VHDL description contains three main parts: the `ENTITY` section describing the entity interface, the `ARCHITECTURE` section of which there may be several instances for a particular entity and the `CONFIGURATION` section which defines the particular architecture to be used for an entity by its environment.

A rich set of tools and has been written for VHDL which aid the development, synthesis, testing and verification of hardware designs. There are also IEEE standard libraries for some pre-defined components.

Several methods have been developed to translate a VHDL subset into formats suitable for formal verification by both model checking and theorem proving [49, 71]. However, as VHDL semantics (in contrast to UP) is not formally defined, any such translations must be taken as only 'provisional'.

1.1.2 Verilog

Verilog HDL [73] and VHDL are essentially identical in function, however, Verilog is simpler (less general) than VHDL, and syntactically different. Its programming constructs are based on C, while those of VHDL are based on ADA. Verilog was made IEEE standard 8 years after VHDL in 1995. Similarly to VHDL, Verilog has a large library of predefined components.

Figure 8.1 shows a structural specification of a half adder in both Verilog and the UP formalism. An example of a sequential logic circuit is given in 8.2, now comparing Verilog to Extended UP as described in chapter 6.

The attempts [28] to formalise Verilog in order to facilitate formal verification are (in common with VHDL) still in very early stages. Unlike UP, Verilog is (in common with VHDL and ELLA) a deterministic language, so non-determinism must be emulated.

Verilog	Extended Update Plans
`module full_adder(a,b,i,s,o);`	`full_adder(a,b,i,s,o) =`
`  input a,b,i;`	`  half_adder(a,b,w1,w2)`
`  output s,o;`	`  half_adder(w1,i,s,w3)`
`  wire w1,w2,w3;`	`  or(w2,w3,o).`
`  half_adder g1(a,b,w1,w2);`	
`  half_adder g2(w1,i,s,w3);`	
`  or g3(w2,w3,o);`	
`endmodule`	

Figure 8.2: *Verilog vs. EUP*

ELLA	Update Plans
`FN MUX = (bit: c i1 i2) -> bit:`	
`  CASE c`	
`  OF hi: i1,`	`mux(c,i1,i2,i1) = c[HI].`
`     lo: i2`	`mux(c,i1,i2,i2) = c[LO].`
`  ESAC.`	

Figure 8.3: ELLA vs. UP

1.1.3 ELLA

ELLA [57] has been developed for circuit design, with the aim of supporting automatic synthesis from high-level behavioural descriptions to low-level structural descriptions. ELLA is a parallel language and describes circuit behaviour by defining nodes, connections and signal flows in that circuit.

One of the marked differences between ELLA and VHDL is that in VHDL the structural, behavioural and procedural design descriptions must be separate, whereas in ELLA they can be freely mixed. Also, ELLA is primarily a functional language, whereas VHDL relies on state transitions. It is not only the regional use[1] and features described in this section that separate VHDL and ELLA. For a fuller comparison between VHDL and ELLA the reader is referred to [79].

As with other HDLs, though a complete formal semantics for ELLA does not currently exist, various verification strategies (based on model checking and theorem proving) for a subset of this specification language have been developed [5, 6, 12].

Two simple examples are provided to compare specification capabilities of ELLA and (E)UP. The first one (figure 8.3) is a two-bit multiplexer, the second one (figure 8.4) is a sequential parity checker. Note the clear arrangement of the parity checker modules in the EUP definition which makes the layout and the connections between the modules immediately obvious.

[1]VHDL is the standard language for the US DoD, ELLA is likely to become its equivalent for the UK MoD

ELLA	Extended Update Plans
FN PARITY_IMP = (bit: in) -> bit: BEGIN MAKE INV: l1, MUX: l3 out, REG: l2 l4. JOIN (in,l1,l2) -> l3, hi -> l4, (l4,l3,hi) -> out, out -> l2, l2 -> l1. OUTPUT out END.	parity_imp(in, out) = inv(l2, l1) ‖ mux(in, l1, l2, l3) ‖ reg(HI, l4) mux(l4, l3, HI, out) reg(out, l2).

Figure 8.4: ELLA vs. EUP

1.2 Concrete machines and instruction sets

1.2.1 RTL, ISPS

Historically, the best known formalisms specifically aimed at the description of instruction sets are register transfer languages (RTL) and Instruction Set Processor Specification (ISPS) [69].

There is a variety of slightly differing RTL notations for describing the workings of computers at the register level. They all, however, semi-formally describe behaviour of computers as stepwise transformations on register contents, where variables correspond to hardware registers, using some FORTRAN, PASCAL or C constructs.

ISPS has its origins in the Instruction Set Processor (ISP) notation and has been frequently used in the past as a design tool to cover a wide area of applications. One of the most significant applications is the PDP-11 specification [20]. However, the specification was ambiguous and various implementations of the PDP-11 machine exhibited differing behaviour of several instructions[2]. Formal semantics of the Update Plans formalism ensures that only unambiguous specifications are written.

1.2.2 RAPIDE

RAPIDE is a high level, event-based, concurrent, object-oriented specification and simulation language. It was designed for prototyping system architectures. Instruction sets are modelled by communicating modules forming an 'architecture'. The formalism started as an effort to complement often over-specific and hard to understand HDL descriptions.

Unfortunately, RAPIDE lacks a formal semantics, which makes any rigorous formal verification methodology impossible. The only way to verify properties of a system under description

[2]For example, the `MOV Rn,(Rn)+` instruction resulted in `Rn` having either the original value or the original value+2 depending on the processor.

RAPIDE	Update Plans
type Producer **is interface**	
action out Send(data: integer);	RECEIVED[0] START[1] READY[0].
action in Ready();	# *Producer*
behavior	START[s] READY[v] =[s ∨ v]⊨
function GenData() **return** integer;	SEND[gendata()] START[0] READY[0].
begin	
Start ⇒ Send(GenData());;	# *Consumer*
Ready ⇒ Send(GenData());;	RECEIVED[r] SEND[data] READY[0] ⟹
constraint	RECEIVED[r + 1] READY[1] r[data].
observe from Start, Send, Ready	
match (Start → Send) → [* **rel** →]	
(Ready → Send);	
end	
end Producer;	

Figure 8.5: RAPIDE vs. UP

is traditional simulation. The authors [50, 66] argue that the strength of the RAPIDE formalism lies in partially ordered sets of events (posets) which are generated by executing a RAPIDE model. Where other concurrent event-based simulation languages produce linear traces of events, RAPIDE simulation shows dependency between events. However, the same effect (building posets) could easily be achieved by an UP simulator by annotating special "communication cells".

The power of Update Plans to describe concrete data manipulations in a parallel environment is demonstrated in figure 8.5 which shows the classic "producer/consumer" problem. The very first 'Send' is initiated by 'Start' and subsequent 'Send' signals causally follow 'Ready' (acknowledgement) signals. The RAPIDE specification of the 'Consumer' interface is not shown here for the sake of brevity, as its description is even longer than that of 'Producer'. On the other hand, the UP specifications of both the 'Producer' and 'Consumer' processes are included. The 'Consumer' simply stores data sent by the 'Producer' into a buffer.

1.3 Parallelism

1.3.1 UNITY

The UNITY formalism [16] consists of two parts: a programming language based on transition systems, and a specification language, based on a linear temporal logic. Similarly to other formalisms, such us UP and CHAM, the UNITY computational model is liberated from control management.

A minimal UNITY program consists of three parts: a collection of variable declarations called the 'declare' section, a set of initial conditions called the 'initially' section, and a

UNITY	Update Plans
program *TrafficLight*	
declare	
$ns, ew : \{red, grn, yel\}$	
initially	
$ns, ew = \{red, red\}$	EW[RED] NS[RED].
assign	
$ns := grn$ if $(ns = red) \wedge (ew = red)$	NS[RED] EW[RED] $\Longrightarrow$ NS[GRN].
$[\!]$ $ns := yel$ if $(ns = grn)$	NS[GRN] $\Longrightarrow$ NS[YEL].
$[\!]$ $ns := red$ if $(ns = yel)$	NS[YEL] $\Longrightarrow$ NS[RED].
$[\!]$ $ew := grn$ if $(ns = red) \wedge (ew = red)$	NS[RED] EW[RED] $\Longrightarrow$ EW[GRN].
$[\!]$ $ew := yel$ if $(ew = grn)$	EW[GRN] $\Longrightarrow$ EW[YEL].
$[\!]$ $ew := red$ if $(ew = yel)$	EW[YEL] $\Longrightarrow$ EW[RED].
end *TrafficLight*	

Figure 8.6: UNITY vs. UP program for a traffic light controller.

finite set of statements in the 'assign' section. The standard UNITY execution model is a non-deterministic, fair interleaved selection of all statements from the 'assign' section.

An example of a program for a traffic light controller in both UNITY and UP is given in figure 8.6. Although very similar to UP, UNITY lacks an appropriate abstraction mechanism and is more suited for the description of concurrent programs, rather than architectures. Also, as opposed to Extended UP, as described in the second part of this thesis, there is no concept of a sequence which is often needed to describe hardware architectures.

1.3.2 Γ, CHAM

The Chemical Abstract Machine (CHAM) model of computation [9, 10, 37] is fashioned on chemicals and chemical reactions. The model is build upon the Γ language [4] for parallel programming. Γ computation is a set of reactions that consume elements of a multiset and produce new ones according to the rules that constitute the program. As reactions can take place in any order (or even simultaneously), the model is inherently parallel in common with other formalisms such as UP and Petri Nets.

The Γ language is extended by the CHAM formalism by the provision of a classification scheme for reaction rules and a membrane construct which extends the use of multisets in such a way that they can form parts of molecules. The second extension allows the formalism to deal with abstraction and hierarchical programming, as a membrane can be porous to allow communication between an encapsulated solution (multisets of molecules) and its environment. Perhaps an overused example, but one which effectively demonstrates the philosophy of CHAM is that a solution originally made of all integers from 2 to n along with a rule that any integer destroys its multiple will result in a solution of prime numbers between 2 and n.

CHAM, as opposed to UP, is predominantly aimed at formally specifying and analysing

software achitectures. In contrast to CHAM, UP specifications are also reasonably simple to implement on concrete machines.

1.4 Protocols

There are many competing, well established protocol description languages/environments. The main contributors are SDL, Estelle and LOTOS.

SDL stands for Specification and Description Language [15]. It is an object-oriented, formal language standardised by The International Telecommunications Union–Telecommunications Standardization Sector (formerly CCITT). It was developed to describe real-time and distributed communicating systems. The major drawback in using SDL for protocol specification, however, is that both the graphical and textual protocol specification descriptions tend to be large, and therefore difficult to understand and maintain.

Estelle (Extended State Transition Language) [38] is an ISO standardised, partly formalised technique for the specification of distributed and concurrent processing systems based on an extended state transition model (non-deterministic finite state machine augmented by the addition of variables). Estelle has been successfully used on the specification and analysis of many real protocols.

LOTOS and especially DILL deserve further attention as they have been used for the description of hardware, specifically sequential digital circuits.

1.4.1 LOTOS, DILL

LOTOS (Language Of Temporal Ordering Specification) [39] is a by-product of the effort of standardisation of the Open Systems Interconnection (OSI) within ISO. It is a standardised formal description technique designed to describe distributed concurrent information processing systems, in particular the OSI architecture and the related standards.

DILL (Digital Logic in LOTOS) [33, 34] uses LOTOS to formally specify digital hardware making use of a library of typical components and is realised through translation into LOTOS. The analysis and verification of properties takes place at the LOTOS level.

For a comparison between DILL and UP see figure 8.7. The basic component of a logic circuit (a NAND gate) is described using (only) a behavioural style in DILL, and both the behavioural and stuctural styles in UP. It turns out that great care must be taken to avoid non-deterministic behaviour of components when writing DILL specifications. Even if such care has been taken there is another problem—the DILL model is not suitable for circuits containing cyclic connections. Although the NAND gate is modelled in the manner "when all inputs arrive, then output happens" as in UP, the DILL specification will deadlock [33] if there is a cyclic connection within each stage.

Another disadvantage of this approach is that DILL only has a limited set of components in the hardware library and the construction of new components requires knowledge of both

DILL	Update Plans (behavioural description)
process Nand2 [Ip1, Ip2, Op] : **noexit** :=	ip1, ip2 :: Bit.
(Ip1 ?dtIp1 : Bit; **exit** (dtIp1, **any** Bit)	nand(ip1, ip2, ¬(ip1 & ip2)).
‖	
Ip2 ?dtIp2 : Bit; **exit** (**any** Bit, dtIp2))	Update Plans (structural description)
>> **accept** dtIp1, dtIp2 : Bit **in**	nand(ip1, ip2, op) = ip1[0] ip2[0] ⟹ op[1].
(Op !(dtIp1 nand dtIp2);	= ip1[0] ip2[1] ⟹ op[0].
Nand2 [Ip1, Ip2, Op])	= ip1[1] ip2[0] ⟹ op[0].
endproc (* Nand2 *)	= ip1[1] ip2[1] ⟹ op[0].

Figure 8.7: DILL vs. UP modelling a NAND gate

DILL and LOTOS. However, it seems to be convenient for giving higher-level architecture specifications.

1.5 Z/VDM

The list of formal methods would not be complete without at least mentioning the pioneering Z notation and the Vienna Development Method (VDM) [32] both of which still have a large user base. Z and VDM are based on set theory and first order predicate calculus. ISO standards exist for both of them.

Z is a non-executable specification language developed mainly by the Programming Research Group at the Oxford University Computing Laboratory from the late 1970s. The most noticeable difference between Z and VDM are structures called schemas, by which programs are described.

VDM, being a method rather than just a notation for expressing software specification design and development, also contains an inference system for constructing correctness proofs and a methodology for developing software from a specification in a formally verifiable manner.

While different in syntax and structure, Z and VDM do not differ radically from one another. Unfortunately, neither Z nor VDM can handle concurrent systems [78].

Although there has been some work [24] in describing computer architectures using both these methods, they are used primarily for software requirements specification and program development.

2 Verification

While formal specification of a system often forms an important part of its design, it is sometimes not sufficient and needs to be complemented by some sort of a verification strategy. This section gives a short overview of different verification techniques.

The traditional method of trying to ensure correctness is through simulation and testing.

However, as Moore's prediction[3] turned out to be very accurate, these techniques have their limitations as with ever-increasing complexity of hardware it is impossible to simulate all inputs or sequences of inputs. Formal techniques, on the other hand, are better able to scale with complexity by using various mathematical methodologies rather than exhaustive testing.

While there have been some attempts at the "ideal solution"—correct-by-construction synthesis [64], designers are still using hand-crafted custom design [44] in an effort to optimise performance of systems, which necessitates some post-design verification.

Formal post-design verification involves the use of analytical methods to prove that the implementation of a system conforms to the specification. Formal proofs are based on establishing that universal properties about the design hold independently of any particular set of inputs, or on showing the equivalence between several layers of a system specification with differing degree of abstraction. Verification methods can be classified into two major groups: model-checking and theorem-proving, based on these criteria.

In model-checking, the implementation description (model) is given as (or transformed to) a finite state machine (FSM), and the specification description by properties given in some kind of temporal logic. Correctness is then established by showing whether some property holds in the FSM model, or if not, a counter example is provided. Model-checking tools are fully automated, and perform exhaustive searches through the state space of the model. Unfortunately, this method is not scalable to larger circuits due to the state explosion problem. Various methodologies have been developed to alleviate the state explosion [17] usually by treating the sets of states symbolically or replacing the system to be checked by a simpler one in which irrelevant details are suppressed.

Some specification methods use first-order (Boyer-Moore/ACL2) or higher-order logic (HOL, PVS) for both the specification and implementation description. Theorem-proving then tries to establish whether the specification and implementation are equivalent or the language representing the implementation is contained in the language representing the specification. The main advantages of this approach are that the formal proof can be mechanically checked and, in contrast to model checking, theorem proving can deal directly with infinite state spaces. However, as opposed to model-checking the derivation of a formal proof is extremely tedious, as the theorem-proving tools are semi-automatic and need a large degree of expertise for efficient use. Moreover, a theorem-prover is not guaranteed to always give an answer because of decidability problems.

2.1 ACL2

One integrated specification and verification method for computer architectures that stands out of the rest mainly because of its use in the industry, is A Computational Logic for Applicative Common Lisp (ACL2) [13, 41]. ACL2 is a re-implementation of the Boyer-Moore

[3] In 1965 Gordon Moore predicted that complexity of hardware devices would double every 18 months.

system Nqthm. Like Nqthm, ACL2 supports a Lisp-like, first-order mathematical logic.

Most of the Common Lisp functions were axiomatised or defined as functions or macros in ACL2. In contrast to Common Lisp, all functions in ACL2 are total.

There are two large scale verification projects based on ACL2. The first one is a formal executable specification of the Motorola CAP [26] digital signal processor. The second, perhaps even more important project was the application of ACL2 to formalise and prove the correctness of the microcode for the kernel of the floating point division operation used on the AMD5$_K$86 microprocessor [56].

As ACL2 is based on theorem proving, it is more reliant on the user. Authors themselves admit [13] that ACL2 proofs require many skills including great familiarity and insight into the applications areas, engineering issues, mathematics, formal logic, the workings of the ACL2 proof tool, and a lot of persistence and dedication.

3 Conclusions

This chapter briefly described several specification and verification methods relevant to the area of the Update Plan formalism.

3.1 Integrated specification/verification methodologies

The most significant challenge to UP are integrated specification and verification methodologies such as ACL2. However these require a great deal of skill from the user, both in designing the specification in order to make it amenable to verification, and in performing the verification itself. A verification methodology is only useful if it is used. While Update Plans are not, as yet, embedded in a verification formalism (e.g. HOL) there do not seem to be any major obstacles to achieving this. This would provide a useful mechanism for producing intuitive, structured and verifiable specifications.

3.2 Specification methods

It is noticeable that all of the Update Plan specifications in section 1 are more compact but, arguably, more easily understood than the corresponding specifications using alternative methods. In contrast to many of the methods in section 1 the Update Plans formalism has a formal semantics, and can be applied to the specification of both hardware and software architectures.

3.2.1 Hardware

None of the HDLs described in section 1.1 has a formal semantics which has hindered the development of formal verification methodologies [79]. Although some pioneering work has

been done on verification of HDL specifications, this was either only a preliminary work to inspire further research [28], or developed verification methodologies for only a restricted subset/core of that language [5]. UP on the other hand has a simple but formally defined semantics.

The HDLs in section 1.1 are also generally not not very well suited to specification of software architectures.

Furthermore they are deterministic. This makes it relatively simple to specify sequences and modules consisting of sequentially connected elements, but difficult to specify parallel and/or synchronous processes—in general these must be emulated. On the other hand non-determinism is an inherent part of the semantics of Update Plans making the specification of parallel processes relatively easy. Indeed basic Update Plans were somewhat weak in specifying sequential behaviour. The introduction of sequential schemes and archetypes in Extended Update Plans has solved this, and sequential systems can now be specified with ease.

3.2.2 Concrete machines and instruction sets

Again, none of the methods described in section 1.2 has a formal semantics—and RTL in particular does not even have a standard syntax, as can be seen from the number of differing RTL notations. ISPS dates from the 1970s and is essentially an imperative procedural language with all the difficulties in formal reasoning that this entails. Though both RTLs and ISPS can provide detailed descriptions of the hardware/software interface neither is particularly suited to either the specification of hardware or the specification of software at a more abstract level. Nor are they strong in the description of parallel processes.

Though RAPIDE is a more recent development it still lacks a formal semantics, and verification can only be achieved through simulation. RAPIDE specifications also tend to be considerably longer than the corresponding UP specifications.

3.2.3 Parallelism

None of the specification methods discussed so far would be the tool of choice for defining parallel processes. The methods discussed in section 1.3 (UNITY, Γ and CHAM) were designed with parallelism explicitly in mind. However they are all far more suitable for the description of parallel programs rather than hardware. They also lack mechanisms, such as sequential Update Schemes and archetypes, for imposing the sequentiality and synchronisation that is almost always an essential part of hardware behaviour. In addition Γ and CHAM specifications are considerable harder to implement on real machines than the corresponding UP specifications.

3.2.4 Protocols

Of the specification methods discussed in section 1.4 LOTOS and DILL are the most suited to the description of systems across the hardware/software interface. LOTOS provides the higher level (protocol) specification, while DILL uses LOTOS to provide a limited hardware specification facility. This means, however, that writing non-trivial DILL specifications requires a good knowledge of both LOTOS and DILL. Also DILL is limited in the class of systems that it can specify—it cannot even be used to specify a simple RS flip-flop, since this contains cyclical connections. It also tends to introduce unexpected non-determinism unless used carefully.

3.2.5 Z/VDM

These methods are again more suited to higher level (software) specifications rather than hardware. In contrast to UP they do not produce executable specifications.

3.3 Summary

Though there is a large number of specification methods across the application area of Update Plans none is as suited to describing systems at the hardware/software interface as Update Plans. Almost all of the methods tend to be either hardware or software oriented. Most of the methods lack a formal semantics making verification of the systems specified difficult. Also the Extended Update Plans formalism can be used equally well for the specification of sequential and parallel systems, and the interaction between sequential and parallel processes. The other methods surveyed here are mostly suitable for describing sequential or parallel systems, but not both. In general Extended Update Plans provide a powerful method for specifying and reasoning about all the most common processes and systems that exist at the hardware/software interface.

Chapter 9

Conclusions and Future Research

Update Plans constitute a very flexible, clean formalism in which clear, elegant, compact, intuitive, simple to read and unambiguous low level specifications can be written. Contributions to the Update Plans formalism presented in this thesis can be divided into two major areas.

Firstly, in the application domain, instruction sets of four different machines (ranging from concrete to purely mathematical models) have been described using both original Update Plans and UP with the extensions described in the second part of the thesis.

Secondly, syntactic and semantic extensions to Update Plans made the whole formalism even more consistent and expressive. The semantic extensions in particular provide a way to make UP descriptions more compact and readable. This is mainly due to the advances in the area of synchronisation between parallel and sequential processes and the provision of a tool to design specifications with a better degree of modularity.

1 Update Plans applications

Update Plans applications featuring both in the first and second part of the thesis provide a good deal of evaluation of the Update Plans formalism which drove both the syntactic and semantic extensions. Specifications were produced for a variety of concrete and abstract machines.

The PDP-11 machine instruction set specification provided a useful comparison of an UP specification to the historically important formal specifications in [20, 69]. A significant subset of the PDP-11 instruction set, including the complete set of addressing modes, single and double operand instructions, condition code and program flow instruction, and the interrupt mechanism, was developed. This specification was unambiguous (in contrast to, for example, [20]), more compact than [20, 69] and significantly clearer. Consideration of a more detailed specification at the fetch/execute cycle drove the development of sequential update schemes—a semantic extension to the Update Plan model.

The SPARC-V9 was chosen to test UP's suitability for specifying modern RISC architectures. A partial specification of the SPARC-V9 architecture exists [50, 65], but this is aimed more at the specification of connections and communication than at a concise specification of the instruction set. The UP specification covers the SPARC-V9 register architecture (general purpose and floating-point registers, the SPARC-V9 register window mechanism and instructions); arithmetic, logical and floating-point instructions; data transfer instructions; and control transfer instructions. The main achievement of this specification, apart from its clarity and compactness, is that the extensions to the Update Plan formalism allowed simultaneous specification of the SPARC-V9's assembly language and machine code.

The Java Virtual Machine (JVM) was chosen to test Update Plans' suitability for describing more abstract instruction sets using a wide variety of types. Again a compact and readable specification of the JVM instruction set subset was produced. This specification exercise lead to some syntactic changes to the original Update Plans formalism.

The Parallel Random Access Machine (PRAM) [22] was chosen as a specification target because it provided a useful test-bed for the semantic extensions to Update Plans described in chapter 6. It was also a test of the expressive power of Update Plans in that the specification should be able to describe all of the PRAM's memory models within one specification. The "n-PRAM" specification is thus a family of specifications, parameterised by the number of RAMs, and the particular read/write model under consideration.

2 Update Plans extensions

The second class of advances achieved by the thesis are the extensions to the previous Update Plans formalism.

The syntactic extensions have a more significant impact than might at first sight seem. They now allow, in many cases, for specifications at multiple levels of abstraction within one update plan. The equivalence of the specifications is then implicit in that single specification, rather than requiring explicit proof through transformation. The SPARC-V9 specification is an example of such a specification, since it describes both SPARC-V9 assembly language and machine code.

The typing mechanism has been extended to allow a clearer match between "hardware types" (e.g. bits, bytes, words, etc.; registers, memory, etc.) and "software types" (e.g. integers, booleans, etc.; opcodes, operands, condition codes, instruction words, etc.). The JVM specification, for example, defines both the abstract and the physical structure of Java bytecode.

The problem in specifying parallel processes is often not the parallelism in itself, but the interaction between, and the synchronisation of, sequential and parallel processes. In the PRAM, for example, the massive parallelism of the memory access model must be syn-

chronised with the sequentiality of the fetch/execute cycles of each of the individual RAMs in the PRAM. When this research started parallel system could be specified using Update Plans, but synchronisation of the parallel processes thus specified required the introduction of artificial constructs unrelated to the architecture under consideration, leading to inelegant specifications that were hard to read. The introduction of sequential update schemes provides the formalism with a general synchronisation primitive that augments the non-deterministic model of Update Plans by explicitly stating the order in which updates will be applied. As a result it is now also possible to give more abstract descriptions in EUP than in UP.

The development of sequential update schemes led naturally to the development of sequential archetypes. These greatly extend the possibilities for information hiding and structure reuse by encapsulating a series of synchronised updates into a "module" (a sequential archetype). A sequential archetype can express an atomic action at one level of abstraction (e.g. an instruction execution) as a series of atomic actions at a lower level of specification (e.g. phases of the f/e cycle). This simplifies the task of proving the equivalence of the two levels of specification.

3 Future considerations

Suggestions for further UP-related work can be classified into three different areas. Firstly, the work on theoretical aspects of the formalism itself. Secondly, more research should be carried out on the application area of Update Plans. Finally, the implementation for the Update Plans formalism consisting of appropriate CAD tools should be developed.

3.1 Theory

3.1.1 Metrics

An annotation to indicate cost of applying an update or expanding an archetype coupled with a cost-derivation methodology should be developed. This (with a working UP simulator) would not only make the application of Update Plans to compiler optimisations possible as suggested in [60], but it would also prove useful in a variety of other applications such as automatic calculation of delays of (networks of) components in a logic circuit, simulation of latencies in a network environment or answering other performance-related questions.

3.1.2 Modularity

It would be useful to have a mechanism for restricting access of updates to only a selected set of cells in the memory by means of some kind of a modular structure or simply by restricting the domain of variables in some way. Not only would this automatically protect accesses to undesired parts of memory, but the current set of grounding rules [60] could be further

extended. The impact on the compactness and readability of UP specifications would in some cases be significant.

3.1.3 Verification

Verification, and possibly transformation methods should be developed, together with appropriate heuristic rules. Although a first step in this direction has been taken [60], further work is necessary.

On a higher level of abstraction, it is more likely that complex data manipulations will be described by a well-defined external function rather than Update Plans. On a lower level of abstraction, this function will in some cases be specified in terms of Update Plans. Therefore it will be necessary to find such a formalism and develop a methodology of proving semantic equivalence or containment between these two descriptions.

The majority of successful general-purpose proof checking systems that have proved themselves on several industrial-scale microprocessors are based on theorem proving in first or second order logic. Therefore a promising approach seems to be the formalisation of Update Plans in such a system (e.g. ACL2, PVS, HOL, Isabelle, IMPS or Nuprl).

Alternatively, use of existing model-checking tools (e.g. SPIN) could be made after the development of methodologies to transform UP descriptions into specification languages the tools use. It would then be possible to use the tool's logic to specify and check desired properties.

As no method or tool is general enough and appropriate for all the varied, and sometimes conflicting requirements of hardware architecture description and analysis, a combination of existing methodologies will almost certainly need to be used. UP should be used in conjunction with other formal techniques and the traditional approaches such as testing through simulation.

Finally, methodologies to translate specification languages used by general theorem provers to hardware specification languages have been developed [8]. A similar approach could be taken to facilitate automatic synthesis and implementation of chips.

3.2 Applications

Although there have already been a lot of UP-based specifications of concrete and abstract machines, further validation of the Update Plans formalism is needed. This effort should concentrate on the specification of one of the concrete machines already specified, but on a lower level, e.g. microcode. These two levels should then be subject to formal verification for equivalence or containment by one of the methods suggested in section 3.1.3. As Extended Update Plans introduced a convenient way to describe sequences, libraries of components to describe the behaviour of basic building blocks starting from basic elements of logic circuits to blocks such as multipliers for an Update Plans simulator could be defined.

A true test of the (Extended) Update Plans formalism will be more sophisticated models of parallel computation such as F-PRAM [76], which introduce a communication network with latency and a synchronisation barrier among asynchronously running processors. On a slightly related note, attempts to enter into neighbouring domains such as communication protocol specification could be tried.

3.3 Implementation

A prototype implementation of Update Plans [60] which has been successfully used as a didactic aid has been developed. Completion of this work will greatly enhance the usefulness of the formalism not only in automatic derivation of specified properties, but also in prototyping and validation. A full implementation of UP (an UP simulator) will lead to better popularisation of UP, and helping to prompt more research in certain features of Update Plans and their refinement.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] A. G. Alexandrakis, A. V. Gerbessiotis, D. S. Lecomber, and C. J. Siniolakis. Bandwidth, space and computation efficient PRAM programming: The BSP approach. In *Proceedings of the SUP'EUR '96 Conference*, Sep 1996.

[3] J. H. Aylor, R. Waxman, and C. Scarratt. VHDL – feature description and analysis. *IEEE Design and Text of Computers*, 3(2), April 1986.

[4] J.-P. Banâtre and D. L. Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[5] H. Barringer, G. Gough, T. Longshaw, B. Monahan, M. Peim, and A. Williams. A semantics and verification framework for ELLA. Technical Report UMCS-92-4-6, Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK, March 1992. URL `ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-92-4-6.ps.Z` [cited May 2003].

[6] H. Barringer, G. Gough, B. Monahan, and A. Williams. A process algebra foundation for reasoning about core ELLA. Technical Report UMCS-94-12-1, Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK, December 1994. URL `ftp://ftp.cs.man.ac.uk/pub/TR/UMCS-94-12-1.ps.Z` [cited May 2003].

[7] F. L. Bauer. *The Munich Project CIP*, volume 183 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin/Heidelberg/New York, 1985.

[8] C. Berg, S. Beyer, C. Jacobi, D. Kröning, and D. Leinenbach. Formal verification of the VAMP microprocessor (project status). In *Symposium on the Effectiveness of Logic in Computer Science (ELICS02)*, pages 31–36, September 2002. URL `http://busserver.cs.uni-sb.de/publikationen/BBJKL02.pdf` [cited May 2003].

[9] G. Berry. The chemical abstract machine. 1998. URL `http://www-sop.inria.fr/meije/personnel/Gerard.Berry/cham.ps` [cited May 2003].

[10] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[11] P. Bertelsen. Semantics of Java byte code. Technical report, 1997. URL `ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/jvm-semantics.ps.gz` [cited May 2003].

[12] R. Boulton, M. Gordon, J. Herbert, and J. V. Tassel. The HOL verification of ELLA designs. Technical Report TR199, University of Cambridge Computer Laboratory, Cambridge, UK, 1999. URL `http://www.ftp.cl.cam.ac.uk/ftp/papers/reports/TR199-rjb-mjcg-jmjh-jvt-HOL-verification-ELLA.ps.gz` [cited May 2003].

[13] B. Brock, M. Kaufmann, and J. S. Moore. ACL2 theorems about commercial microprocessors. 1166:275–293, 1996. URL `http://www.cs.utexas.edu/users/moore/acl2/v2-1/reports/bkm96.ps` [cited May 2003].

[14] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, Jan 1977.

[15] CCITT. *Red book – Functional Specification and Description Language (SDL). Recommendations Z.101-Z.104*, volume VI. CCITT, Geneva, 1985.

[16] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[17] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science 2000*, pages 176–194, 2001.

[18] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *Journal of the ACM*, 28(1):626–643, Dec 1996.

[19] R. M. Cohen. The Defensive Java Virtual Machine specification, version 0.53. Technical report, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

[20] *PDP-11 Processor Handbook.* Digital Equipment Corporation, 1971.

[21] H. R. Eckhouse, Jr. and L. R. Morris. *Minicomputer Systems – Organization, Programming and Applications (PDP-11).* Prentice-Hall, 1979. ISBN 0-13-583914-9.

[22] S. Fortune and J. Willie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[23] T. S. Frank. *Introduction to the PDP-11 and its Assembly Language.* Prentice-Hall, Le Moyne College, Syracuse, New York, 1983. ISBN 0-13-491704-9.

[24] S. Gerhart, D. Craigen, and T. Ralston. Observations on industrial practice using formal methods. In *15th International Conference on Software Engineering (ICSE)*, Baltimore, Maryland, USA, May 1993.

[25] R. Giegerich. *Implementierung von Programmiersprachen.* Technische Fakultät, Universität Bielefeld, Postfach 86 40, 4800 Bielefeld 1, BRD, 1992. Lecture notes for a course in compiler construction. (In German).

[26] S. Gilfeather, J. Gehman, and C. Harrison. Architecture of a complex arithmetic processor for communication signal processing. In *SPIE Proceedings, International Symposium on Optics, Imaging, and Instrumentation*, pages 624–625, March 1994. URL `http://www.cli.com/hardware/cap.eps` [cited May 2003].

[27] J. Gleick. A bug and a crash: Sometimes a bug is more than a nuisance. 1996. URL `http://www.around.com/ariane.html` [cited May 2003].

[28] M. J. C. Gordon. The semantic challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 136–145, 1995. URL `http://www.math.chalmers.se/~gpace/HDL/papers/V.ps` [cited May 2003].

[29] A. Gupta. Formal hardware verification methods: A survey. In *Formal Methods in System Design*, volume 1, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, 1992. Kluwer Academic Publishers, Hingham, MA, USA.

[30] P. Hämäläinen. PRAM emulator, user's manual. Technical Report B-1992-2, University of Joensuu, PO Box 111, 80101 Joensuu, Finland, 1992. URL `ftp://cs.joensuu.fi/pub/Reports/B-1992-2.ps` [cited May 2003].

[31] E. Harcourt, J. Mauney, and T. Cook. Formal specification and simulation of instruction-level parallelism. In *Proceedings of the 1994 European Design Automation Conference.* IEEE Press, October 1994.

[32] A. Harry. *Formal Methods: VDM and Z.* Wiley, 1996. ISBN 0-471-95857-3.

[33] J. He and K. J. Turner. Modelling and verifying synchronous circuits in DILL. Technical Report CSM-152, Department of Computing Science and Mathematics, University of Stirling, Scotland, April 1999. URL `ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/sync-dill.ps.gz` [cited May 2003].

[34] J. He and K. J. Turner. Specification and verification of synchronous hardware using LOTOS. October 1999. URL `ftp://ftp.cs.stir.ac.uk/pub/staff/kjt/research/pubs/sync-lot.ps.gz` [cited May 2003].

[35] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 3rd edition, June 2002. ISBN 1-55860-724-2.

[36] *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, 1988. IEEE Std. 1076–1987.

[37] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[38] ISO. *Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique based on an Extended State Transition Model.* ISO 9074. International Organization for Standardization, Geneva, 1989.

[39] ISO/IEC. *Information Processing Systems – Open Systems Interconnection* – LOTOS *– A Formal Description Technique based on the Temporal Ordering of Observational Behaviour.* ISO/IEC 8807. International Organization for Standardization, Geneva, 1989.

[40] S. Juvaste. *Modelling Parallel Shared Memory Computations.* PhD thesis, University of Joensuu, 1998. URL `ftp://ftp.cs.joensuu.fi:/pub/Dissertations/juvaste.ps.gz` [cited May 2003].

[41] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *Software Engineering*, 23(4):203–213, March 1997. URL `http://www.cs.utexas.edu/users/moore/publications/km97.ps.Z` [cited May 2003].

[42] C. Kern and M. R. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.

[43] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993. ISSN 1049-331X.

[44] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin. Verity—a formal verification program for custom CMOS circuits. *IBM Journal of Research and Development*, 39(1/2):149–165, March 1995.

[45] W. Lamain. Update Plans, implementatie aspecten. Master's thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1992. (In Dutch).

[46] D. S. Lecomber, K. R. Sujithan, and J. M. D. Hill. Architecture-independent locality analysis and efficient PRAM simulations. In *HPCN'97*, Vienna, April 1997. Springer-Verlag.

[47] G. W. Leibniz. *Œuvre philosophiques, latines et françaises, de feu Mr. de Leibniz, tirées de ses manuscrits, qui se conservent dans la bibliothéque royale à Hanovre et publiées par M. Rud. Eric Raspe.* Amsterdam/Leipzig, 1765.

[48] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition).* Addison-Wesley, 1999. URL `http://java.sun.com/docs/books/vmspec/index.html` [cited May 2003].

[49] J. Lohse, J. Bormann, M. Payer, and G. Venzl. VHDL-translation for BDD-based formal verification. 1994. URL `http://tech-www.informatik.uni-hamburg.de/vhdl/papers/verification/vhdl2fsm.ps.gz` [cited May 2003].

[50] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using RAPIDE. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[51] M. Marin. Binary tournaments and priority queues: PRAM and BSP. Technical Report PRG-TR-7-97, Oxford University, January 1997.

[52] M. C. McFarland. Formal verification of sequential hardware: A tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5), May 1993.

[53] E. Meijer. *Calculating Compilers.* PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1992.

[54] H. Meijer. *Programmar: A Translator Generator.* PhD thesis, University of Nijmegen, Toernooiveld 1, Nijmegen, The Netherlands, 1986.

[55] J. Mencák. Java target code optimization. Master's thesis, Department of Computer Science and Engineering, Brno University of Technology, The Czech Republic, 1999.

[56] J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5$_K$86 floating point division algorithm. March 1996. URL `http://www.cli.com/news/divide.ps` [cited May 2003].

[57] J. D. Morison and A. S. Clarke. *ELLA200: A Language for Electronic System Design.* McGraw-Hill, 1993.

[58] H. R. Osborne. The semantics and syntax of Update Schemes. In *Code Generation — Concepts, Tools, Techniques (Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20-24 May 1991)*, Workshops in Computing, pages 210–223. Springer Verlag, 1992. URL `http://scom.hud.ac.uk/scomhro/Papers/CODE91/code91.ps` [cited May 2003].

[59] H. R. Osborne. Update Plans. In *Proceedings of the 25th Hawaii International on System Sciences* (Volume II: *Software Technology*), pages 488–496. IEEE Computer Society Press, 1992. URL `http://scom.hud.ac.uk/scomhro/Papers/HICSS25/hicss25.ps` [cited May 2003].

[60] H. R. Osborne. *Update Plans – A High Level Low Level Specification Language.* PhD thesis, University of Nijmegen, 1995. URL `http://scom.hud.ac.uk/scomhro/Papers/PhD/phd.ps` [cited May 2003].

[61] H. R. Osborne. Update Plans for parallel architectures. In *Abstract Machine Models for Parallel and Distributed Computing*, pages 79–90, Amsterdam, 1996. IOS Press. URL `http://scom.hud.ac.uk/scomhro/Papers/AMW/amw.ps` [cited May 2003].

[62] H. R. Osborne. The Postroom Computer. *Journal of Educational Resources in Computing*, 1(4):81–110, December 2001. URL `http://scom.hud.ac.uk/scomhro/Papers/JERIC/jeric.ps` [cited May 2003].

[63] V. Pratt. Anatomy of the Pentium bug. In *TAPSOFT'95: Theory and Practice of Software Development*, pages 97–107. Springer Verlag, 1995. URL `http://boole.stanford.edu/pub/anapent.ps.gz` [cited May 2003].

[64] P. S. Rajan. Transformations in high-level synthesis: Formal specification and efficient mechanical verification. October 1994. URL `http://www.csl.sri.com/papers/c/s/csl-94-10/csl-94-10.ps.gz` [cited May 2003].

[65] A. Santoro, W. Park, and D. Luckham. SPARC-V9 architecture specification with RAPIDE. Technical Report CSL-TR-95-677, 1995. URL `ftp://pavg.stanford.edu/pub/Rapide-1.0/sparc.ps.Z` [cited May 2003].

[66] A. Santoro, W. Park, and D. Luckham. Specifying instruction set architectures with RAPIDE. Technical report, Computer Systems Lab, Stanford University, to appear.

[67] J. Savage. *Models of Computation. Exploring the Power of Computing.* Addison-Wesley, 1998.

[68] C.-J. H. Seger. An introduction to formal verification. Technical Report 92-13, Department of Computer Science, Vancouver, B.C., Canada, June 1992.

[69] D. P. Siewiorek, C. G. Bell, and A. Newell. *Computer Structures: Principles and Examples.* McGraw-Hill, 1982.

[70] *UltraSPARC IIi – User's Manual.* Sun Microelectronics, 1999. URL `http://www.sun.com/oem/products/manuals/805-0087.pdf` [cited May 2003].

[71] J. V. Tassel and D. Hemmendinger. Toward formal verification of VHDL specification. In L. Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 261–270, Amsterdam, November 1989. Elsevier Science Publishers.

[72] R. G. Taylor. *Models of Computation and Formal Languages.* Oxford University Press, 1998. ISBN 0-19-510983-X.

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

[74] M. G. J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual, Revision 1.125.* Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 2002. URL `http://www.cwi.nl/projects/MetaEnv/meta/doc/manual.ps.gz` [cited May 2003].

[75] B. Venners. *Inside the Java Virtual Machine.* McGraw-Hill, 1998.

[76] J. Veräjäntausta. An F-PRAM emulator. Technical Report B-1998-1, University of Joensuu, PO Box 111, 80101 Joensuu, Finland, 1998. URL `ftp://cs.joensuu.fi/pub/Reports/B-1998-1.ps.gz` [cited May 2003].

[77] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual.* Prentice Hall, 2000. ISBN 0-13-825001-4. URL `http://www.sparc.com/standards/v9.ps.Z` [cited May 2003].

[78] B. A. Wichmann. A personal view of formal methods. March 2000. URL `http://www.npl.co.uk/ssfm/download/documents/baw_fm.pdf` [cited May 2003].

[79] A. Williams. Comparison of ELLA and VHDL. Technical Report IED 4/1/1357, Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK, 1994. URL `ftp://ftp.cs.man.ac.uk/pub/hardware-verification/ELLA-PROJECT/D2.1b.ps.gz` [cited May 2003].

[80] P. J. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers*, 44(1), October 1995.

Appendix A

Extended Update Plans Grammar

This appendix gives a full context free grammar for Extended Update Plans as specified in this thesis. The grammar is given as an ASF+SDF [43, 74] specification, a specification formalism developed at the University of Amsterdam and the *Centrum voor Wiskunde en Informatica*. The specification can be read as a context free grammar, with production rules reading from right to left. The following notational conventions apply

- $S*$ defines zero or more repetitions of a symbol (non-terminal or literal) S;
- $S+$ defines one or more repetitions of a symbol S;
- $\{S\ sep\}*$ defines zero or more repetitions of a symbol S separated by the literal *sep*;
- $\{S\ sep\}+$ defines one or more repetitions of a symbol S separated by the literal *sep*;
- $(\cdot)$ defines grouping of two or more symbols '$\cdot$';
- $[s]$ represents any one of the literals in the string s;
- $\tilde{}[s]$ represents any character not in the string s;
- `\t` is the horizontal tabulation character;
- `\n` is the newline character.

The {**left**} and {**bracket**} annotation is applied to disambiguate parsing, as is the information provided under a **priorities** header. The symbols appearing in the **sorts** section denote start symbols for the grammar.

module Main

imports Layout Schemes Types Stores Archetypes Updates

exports

sorts SCRIPT

context-free syntax

CONFIGURATION "." PLAN → SCRIPT

(ITEM ".")* → PLAN

TYPE-DECLARATION → ITEM

STORE-DECLARATION → ITEM

ARCHETYPE-DEFINITION → ITEM

UPDATE → ITEM

module Updates

imports Alternatives ParallelBlocks SequentialBlocks

exports

context-free syntax

ALTERNATIVES → UPDATE

PARBLOCK → UPDATE

SEQBLOCK → UPDATE

module Alternatives

imports Schemes

exports

context-free syntax

{SCHEME ";"}+ → ALTERNATIVES

module ParallelBlocks

imports Updates

exports

context-free syntax

"(||" {UPDATE "||"}+ "||)" → PARBLOCK

In the production rules for sequential blocks layout is forbidden between the pipeline symbol '|' and the symbol/variable indicating a stage.

module SequentialBlocks

imports Lexicon Updates

exports

context-free syntax

"(" SEQBLOCK-ID-OPT "|" {(STAGE-OPT UPDATE) "|"}+ "|)" → SEQBLOCK

SYMB-CONST → SEQBLOCK-ID-OPT

SYMB-CONST | VARIABLE → STAGE-OPT

→ STAGE-OPT

In the production rules for update schemes layout is forbidden between a locator and the '[' or ']' of the cell sequence of which it is a locator.

module Schemes

imports Terms

exports

context-free syntax

CONFIGURATION GUARD CONFIGURATION → SCHEME

REPEAT GUARD CONFIGURATION → SCHEME

TEXT CONTEXT → CONFIGURATION

TERM* → TEXT

LOC-EXPR* → CONTEXT

"" → REPEAT

"=[" TERM "]=>" → GUARD

"==>" → GUARD

LEFT-SECTION+ LOCATOR → LOC-EXPR

"?" TERM-OPT "[" TEXT "]" → LOC-EXPR

"!" TERM-OPT "[" TEXT "]" → LOC-EXPR

LOCATOR "[" TEXT "]" → LEFT-SECTION

TERM-OPT → LOCATOR

module Archetypes

imports BasicArchetypes AmbidextrousArchetypes Lexicon Schemes Updates

exports

context-free syntax

BASIC-ARCHETYPE-DEFINITION	→ ARCHETYPE-DEFINITION
AMBID-ARCHETYPE-DEFINITION	→ ARCHETYPE-DEFINITION
"=" ARCHETYPE-BODY	→ BASIC-DEFINITION
UPDATE | CONFIGURATION	→ ARCHETYPE-BODY
ARCHETYPE-CALL	→ TERM
ARCHETYPE-NAME INDEX-OPT PARAMETERS	→ ARCHETYPE-CALL
INDEX	→ INDEX-OPT
	→ INDEX-OPT
"(" {TEXT ","}* ")"	→ PARAMETERS

module BasicArchetypes

imports Archetypes Lexicon

exports

context-free syntax

BASIC-DECLARATION BASIC-DEFINITION+	→ BASIC-ARCHETYPE-DEFINITION
BASIC-ARCHETYPE-NAME PARAMETERS	→ BASIC-DECLARATION

module AmbidextrousArchetypes

imports Archetypes Lexicon

exports

context-free syntax

AMBID-DECLARATION BASIC-DEFINITION+	→ AMBID-ARCHETYPE-DEFINITION
ARCHETYPE-NAME PARAMETERS TEXT	→ AMBID-DECLARATION

module Stores

imports Lexicon

exports

context-free syntax

STORE-NAME	→ STORE-STRUCTURE
"(" STORE-STRUCTURE ")"	→ STORE-STRUCTURE {bracket}
STORE-STRUCTURE "++" STORE-STRUCTURE	→ STORE-STRUCTURE {left}
STORE-STRUCTURE "|" STORE-STRUCTURE	→ STORE-STRUCTURE {left}
STORE-STRUCTURE "*"	→ STORE-STRUCTURE
"{" STORE-STRUCTURE "}" NUMBER	→ STORE-STRUCTURE
"{" {STORE ","}+ "}"	→ STORE-DECLARATION
STORE-IDENTIFIER	→ STORE
STORE-IDENTIFIER "=" STORE-STRUCTURE	→ STORE
STORE-NAME CONST-OPT	→ STORE-IDENTIFIER

context-free priorities

STORE-STRUCTURE "*" → STORE-STRUCTURE >
STORE-STRUCTURE "++" STORE-STRUCTURE → STORE-STRUCTURE >
STORE-STRUCTURE "|" STORE-STRUCTURE → STORE-STRUCTURE

module Types

imports Terms Lexicon Stores

exports

context-free syntax

{(TERM CONST-OPT) ","}+ "::" STORE-STRUCTURE → TYPE-DECLARATION

module Terms

imports Archetypes BasicTerms Arithmetic Logic Ordering Memory

exports

context-free syntax

"(" TERM "::" STORE-STRUCTURE ")"	→ TERM
TERM "*"	→ TERM
TERM "++" TERM	→ TERM {left}
TERM "|" TERM	→ TERM {left}
"(" TERM ")"	→ TERM {bracket}
TERM	→ TERM-OPT
	→ TERM-OPT

context-free priorities

TERM "*" → TERM > TERM "++" TERM → TERM > TERM "|" TERM → TERM

module BasicTerms

imports Lexicon

exports

context-free syntax

NUMBER → TERM
CHAR → TERM
SYMB-CONST → TERM
VARIABLE → TERM
DONTCARE → TERM

module Arithmetic

imports BasicTerms

exports

context-free syntax

TERM "+" TERM → TERM {**left**}
TERM "–" TERM → TERM {**left**}
TERM "×" TERM → TERM {**left**}
TERM "/" TERM → TERM {**left**}
TERM "%" TERM → TERM {**left**}
TERM "^" TERM → TERM {**left**}
"–" TERM → TERM

context-free priorities

"–" TERM → TERM > TERM "^" TERM → TERM >
{**left:** TERM "%" TERM →
TERM "/" TERM → TERM
TERM "×" TERM → TERM} >
{**left:** TERM "–" TERM → TERM
TERM "+" TERM → TERM }

module Logic

imports BasicTerms

exports

context-free syntax

TERM "∧" TERM → TERM {**left**}
TERM "∨" TERM → TERM {**left**}
"¬" TERM → TERM

context-free priorities

"¬" TERM → TERM > TERM "∧" TERM → TERM > TERM "∨" TERM → TERM

module Ordering

imports BasicTerms

exports

context-free syntax

TERM "<" TERM → TERM
TERM "≤" TERM → TERM
TERM "=" TERM → TERM
TERM "≠" TERM → TERM
TERM "≥" TERM → TERM
TERM ">" TERM → TERM

module Memory

imports Lexicon BasicTerms

exports

context-free syntax

"|" STORE-NAME "|" → TERM
"@" "(" TERM "," TERM ")" → TERM
"@" TERM → TERM

module Lexicon

exports

lexical syntax

[0-9]+ → NUMBER
"'" ~['\n] "'" → CHAR
[A-Z][A-Z'0-9_]* → SYMB-CONST
[a-z][a-z'0-9_]* → VARIABLE
"_" → DONTCARE

[A-Z][A-Z'0-9_]* [a-z][A-Za-z'0-9_]* → STORE-NAME

[a-z][a-z'0-9_]* → BASIC-ARCHETYPE-NAME
[A-Z][A-Z'0-9_]* → COMMAND-ARCHETYPE-NAME
BASIC-ARCHETYPE-NAME → ARCHETYPE-NAME
COMMAND-ARCHETYPE-NAME → ARCHETYPE-NAME

NUMBER → INDEX
"{" NUMBER "}" → INDEX

"(" NUMBER ")" → CONST-OPT
→ CONST-OPT

module Layout

exports

lexical syntax

```
[\\t\n]*          → LAYOUT
"#" [\n]* [\n] → LAYOUT
```

context-free restrictions

```
LAYOUT? -/- [\\t\n]
```

Appendix B

PDP-11

```
REG(r,v) r          = r[v].                          # register mode
AUTOINC(b,v) r      = r[b] b[v]c ⟹ r[c].             # autoincrement mode
AUTODEC(a,v) r      = r[b] a[v]b ⟹ r[a].             # autodecrement mode
INDEX(b+d,v) r d    = r[b] b+d[v].                   # index mode
REGDEF(b,v) r       = r[b] b[v].                     # register deferred mode
AUTOINCDEF(b2,v) r  = r[b1] b1[b2]c1 b2[v]c2 ⟹ r[c1]. # autoincrement deferred mode
AUTODECDEF(a2,v) r  = r[b1] a1[a2]b1 a2[v]b2 ⟹ r[a1]. # autodecrement deferred mode
INDEXDEF(b2,v) r d  = r[b1] b1+d[b2] b2[v].          # index deferred mode

IMM(v) v            = .                              # immediate mode
ABS(a,v) a          = a[v].                          # absolute mode
REL(pc+d,v) d       = PC[pc] pc+d[v].                # relative mode
RELDEF(a,v) d       = PC[pc] pc+d[a] a[v].           # relative deferred mode
```

```
sop(v) = REG(_,v).
       = AUTOINC(_,v).
       = AUTODEC(_,v).
       = INDEX(_,v).
       = REGDEF(_,v).
       = AUTOINCDEF(_,v).
       = AUTODECDEF(_,v).
       = INDEXDEF(_,v).
       = IMM(_,v).
       = ABS(_,v).
       = REL(_,v).
       = RELDEF(_,v).
```

```
dop(a,v) = REG(a,v).
         = AUTOINC(a,v).
         = AUTODEC(a,v).
         = INDEX(a,v).
         = REGDEF(a,v).
         = AUTOINCDEF(a,v).
         = AUTODECDEF(a,v).
         = INDEXDEF(a,v).
         = ABS(a,v).
         = REL(a,v).
         = RELDEF(a,v).
```

$\{\mathtt{Nibble} = \mathtt{Bit\ Bit\ Bit\ Bit}, \mathtt{Byte} = \mathtt{Nibble\ Nibble}, \mathtt{Word} = \mathtt{Byte\ Byte}\}.$

$\mathtt{b_n}, \mathtt{b_z}, \mathtt{b_v}, \mathtt{b_c} :: \mathtt{Bit}.$
$\mathtt{nibble} :: \mathtt{Nibble}.$
$\mathtt{byte} :: \mathtt{Byte}.$
$\mathtt{PSW[byte\ nibble]CC_N[b_n]CC_Z[b_z]CC_V[b_v]CC_C[b_c]}.$

$\mathtt{cc(v)} = \mathtt{CC_N}[(\mathtt{v} <_\mathtt{s} 0)\ (\mathtt{v} = 0)\ (\neg(\mathtt{MIN_s} \leq_\mathtt{s} \mathtt{v} \leq_\mathtt{s} \mathtt{MAX_s}))\ (\mathtt{v} >_\mathtt{u} \mathtt{MAX_u})]\ |.$

$\mathtt{arithm}(\_, 0) = \mathtt{CLR}.$ # *clear destination*
$\mathtt{arithm}(\mathtt{x}, \neg\mathtt{x}) = \mathtt{COM}.$ # *complement destination*
$\mathtt{arithm}(\mathtt{x}, \mathtt{x}+1) = \mathtt{INC}.$ # *increment destination*
$\mathtt{arithm}(\mathtt{x}, \mathtt{x}-1) = \mathtt{DEC}.$ # *decrement destination*
$\mathtt{arithm}(\mathtt{x}, -\mathtt{x}) = \mathtt{NEG}.$ # *negate destination*
$\mathtt{arithm}(\mathtt{x}, \mathtt{x}+\mathtt{c}) = \mathtt{ADC\ CC_C[c]}.$ # *add carry to destination*
$\mathtt{arithm}(\mathtt{x}, \mathtt{x}-\mathtt{c}) = \mathtt{SBC\ CC_C[c]}.$ # *subtract carry from destination*
$\mathtt{arithm}(\mathtt{x}, \mathtt{x}/2) = \mathtt{ASR}.$ # *arithmetic shift right destination*
$\mathtt{arithm}(\mathtt{x}, \mathtt{x} \times 2) = \mathtt{ASL}.$ # *arithmetic shift left destination*
$\mathtt{arithm}(\_, -1 \times \mathtt{n}) = \mathtt{STX\ CC_N[n]}.$ # *sign extend destination*
$\mathtt{arithm(x,r)\ dop(a,x)} \Longrightarrow \mathtt{a[r]\ cc(r)}.$

$\mathtt{v_1}, \mathtt{v_0} :: \mathtt{Byte}.$
$\mathtt{SWAB\ dop(a,\_)} = \mathtt{a[v_1\ v_0]} \Longrightarrow \mathtt{a[v_0\ v_1]}.$ # *swap bytes of destination*

$\mathtt{arithm(x, x+1)} = \mathtt{INC} \Longrightarrow \mathtt{W}.$ # *increment destination*
$\mathtt{arithm(x, x+1)} = \mathtt{INCB} \Longrightarrow \mathtt{B}.$ # *increment destination byte*
$\mathtt{arithm(x,r)\ dop(a,x)} \Longrightarrow \mathtt{rcc(a, r, arithm(\_,\_))}.$

$\mathtt{type(W)} = \mathtt{Word}\ |.$
$\mathtt{type(B)} = \mathtt{Byte}\ |.$
$\mathtt{rcc(a,v,w)} = \mathtt{cc(v,w)}\ | \Longrightarrow \mathtt{a[(v :: type(w))]}.$
$\mathtt{cc(v,W)} = \mathtt{CC_N}[(\mathtt{v} <_\mathtt{s} 0)\ (\mathtt{v} = 0)\ (\neg(\mathtt{MIN_s} \leq_\mathtt{s} \mathtt{v} \leq_\mathtt{s} \mathtt{MAX_s}))\ (\mathtt{v} >_\mathtt{u} \mathtt{MAX_u})]\ |.$
$\mathtt{cc(v,B)} = \mathtt{CC_N}[(\mathtt{v} <_\mathtt{bs} 0)\ (\mathtt{v} = 0)\ (\neg(\mathtt{MIN\_B_s} \leq_\mathtt{bs} \mathtt{v} \leq_\mathtt{bs} \mathtt{MAX\_B_s}))\ (\mathtt{v} >_\mathtt{bu} \mathtt{MAX\_B_u})]\ |.$

$\mathtt{TST\ dop(\_, x)} \Longrightarrow \mathtt{cc(x)}.$ # *test destination*
$\mathtt{ROR\ dop(a,x)\ CC_C[c]} \Longrightarrow \mathtt{a[(x>>1)|(c<<15)]\ CC_C[(x\&1 :: Bit)]}.$
$\mathtt{ROL\ dop(a,x)\ CC_C[c]} \Longrightarrow \mathtt{a[(x<<1)|c]\ CC_C[(x>>15)\&1 :: Bit)]}.$

$\mathtt{MOV\ sop(v)\ dop(a,\_)} \Longrightarrow \mathtt{a[v]}.$

$\texttt{abrithm(x,y,x+y)} = \texttt{ADD.}$ # *add source to destination*
$\texttt{abrithm(x,y,x-y)} = \texttt{SUB.}$ # *subtract source from destination*
$\texttt{abrithm(x,y,}\neg\texttt{x\&y)} = \texttt{BIC.}$ # *bit clear destination from source*
$\texttt{abrithm(x,y,x|y)} = \texttt{BIS.}$ # *bit set destination from source*
$\texttt{abrithm(x,y,r) sop(x) dop(a,y)} \Longrightarrow \texttt{a[r] cc(r).}$

$\texttt{cmps(x,y,x-y)} = \texttt{CMP.}$ # *compare source to destination*
$\texttt{cmps(x,y,x\&y)} = \texttt{BIT.}$ # *bit test source and destination bytes*
$\texttt{cmps(x,y,r) sop(x) dop(\_,y)} \Longrightarrow \texttt{cc(r).}$

$\texttt{SEN} \Longrightarrow \texttt{CC}_\texttt{N}\texttt{[TRUE].}$ $\texttt{CLN} \Longrightarrow \texttt{CC}_\texttt{N}\texttt{[FALSE].}$
$\texttt{SEZ} \Longrightarrow \texttt{CC}_\texttt{Z}\texttt{[TRUE].}$ $\texttt{CLZ} \Longrightarrow \texttt{CC}_\texttt{Z}\texttt{[FALSE].}$
$\texttt{SEV} \Longrightarrow \texttt{CC}_\texttt{V}\texttt{[TRUE].}$ $\texttt{CLV} \Longrightarrow \texttt{CC}_\texttt{V}\texttt{[FALSE].}$
$\texttt{SEC} \Longrightarrow \texttt{CC}_\texttt{C}\texttt{[TRUE].}$ $\texttt{CLC} \Longrightarrow \texttt{CC}_\texttt{C}\texttt{[FALSE].}$
$\texttt{SCC} \Longrightarrow \texttt{CC}_\texttt{N}\texttt{[TRUE]}\ \texttt{CC}_\texttt{Z}\texttt{[TRUE]}\ \texttt{CC}_\texttt{V}\texttt{[TRUE]}\ \texttt{CC}_\texttt{C}\texttt{[TRUE].}$
$\texttt{CCC} \Longrightarrow \texttt{CC}_\texttt{N}\texttt{[FALSE]}\ \texttt{CC}_\texttt{Z}\texttt{[FALSE]}\ \texttt{CC}_\texttt{V}\texttt{[FALSE]}\ \texttt{CC}_\texttt{C}\texttt{[FALSE].}$

$\texttt{branch(true)}$	$= \texttt{BR.}$	# *branch always*
$\texttt{branch(}\neg\texttt{z)}$	$= \texttt{BNE CC}_\texttt{Z}\texttt{[z].}$	$\# \neq 0$
$\texttt{branch(z)}$	$= \texttt{BEQ CC}_\texttt{Z}\texttt{[z].}$	$\# = 0$
$\texttt{branch(}\neg\texttt{(n \^{} v))}$	$= \texttt{BGE CC}_\texttt{N}\texttt{[n] CC}_\texttt{V}\texttt{[v].}$	$\# \geq 0$
$\texttt{branch(n \^{} v)}$	$= \texttt{BLT CC}_\texttt{N}\texttt{[n] CC}_\texttt{V}\texttt{[v].}$	$\# < 0$
$\texttt{branch(}\neg\texttt{(z\|(n \^{} v)))}$	$= \texttt{BGT CC}_\texttt{Z}\texttt{[z] CC}_\texttt{N}\texttt{[n] CC}_\texttt{V}\texttt{[v].}$	$\# > 0$
$\texttt{branch(z\|(n \^{} v))}$	$= \texttt{BLE CC}_\texttt{Z}\texttt{[z] CC}_\texttt{N}\texttt{[n] CC}_\texttt{V}\texttt{[v].}$	$\# \leq 0$
$\texttt{branch(}\neg\texttt{n)}$	$= \texttt{BPL CC}_\texttt{N}\texttt{[n].}$	$\# +$
$\texttt{branch(n)}$	$= \texttt{BMI CC}_\texttt{N}\texttt{[n].}$	$\# -$
$\texttt{branch(}\neg\texttt{(c\|z))}$	$= \texttt{BHI CC}_\texttt{C}\texttt{[c] CC}_\texttt{Z}\texttt{[z].}$	# *higher (unsigned comparison)*
$\texttt{branch(}\neg\texttt{c)}$	$= \texttt{BCC CC}_\texttt{C}\texttt{[c].}$	# *carry clear*
	$= \texttt{BHIS CC}_\texttt{C}\texttt{[c].}$	# *higher or same (unsigned compar.)*
$\texttt{branch(c)}$	$= \texttt{BCS CC}_\texttt{C}\texttt{[c].}$	# *carry set*
	$= \texttt{BLO CC}_\texttt{C}\texttt{[c].}$	# *lower (unsigned comparison)*
$\texttt{branch(c\|z)}$	$= \texttt{BLOS CC}_\texttt{C}\texttt{[c] CC}_\texttt{Z}\texttt{[z].}$	# *lower or same (unsigned compar.)*
$\texttt{branch(}\neg\texttt{v)}$	$= \texttt{BVC CC}_\texttt{V}\texttt{[v].}$	# *overflow clear*
$\texttt{branch(v)}$	$= \texttt{BVS CC}_\texttt{V}\texttt{[v].}$	# *overflow set*
$\texttt{PC[pc] pc[branch(c) d]cp} \dashv\!\!\!\!\!\!\{\ \texttt{c}\ \}\!\!\!\!\!\!\Rightarrow \texttt{PC[cp + d];}$		# *branch*
"	$\Longrightarrow \texttt{PC[cp].}$	# *no branch*

$\texttt{JMP dop(\_,a)} \Longrightarrow \texttt{PC[a].}$

$\texttt{reg(r,v)} = \texttt{r[v]}\ |.$

PC[pc] pc[JSR r dop(a, _)]qc reg(r, v) reg(6, tp)
$\Longrightarrow$ PC[a] reg(6, sp) sp[v]tp reg(r, qc).
RTS r reg(r, pc) reg(6, sp) sp[v]tp $\Longrightarrow$ PC[pc] reg(6, tp) reg(r, v).

trap(14_8, 16_8) = BPT. # *breakpoint trap*
trap(20_8, 22_8) = IOT. # *input/output trap*
trap(30_8, 32_8) = EMT. # *emulator trap*
trap(34_8, 36_8) = TRAP. # *TRAP*
trap(pc_a, psw_a) reg(6, tp) pc_a[pc] psw_a[psw] PC[pc_s] PSW[psw_s]
$\Longrightarrow$ sp[pc_s psw_s]tp reg(6, sp) PC[pc] PSW[psw].

ret() = RTI. # *return from interrupt*
= RTT. # *return from trap*
ret() reg(6, sp) sp[pc psw]tp $\Longrightarrow$ PC[pc] PSW[psw] reg(6, tp).

NOP $\Longrightarrow$. # *no operation*

Appendix C

SPARC-V9

$\{\texttt{Bit}\}.$
$\{\texttt{Byte}(01_2) = \{\texttt{Bit}\}8, \texttt{Halfword}(10_2) = \{\texttt{Bit}\}16, \texttt{Word}(00_2) = \{\texttt{Bit}\}32,$
$\texttt{Extended\_word} = \{\texttt{Bit}\}64\}.$
$\{\texttt{Fp\_single} = \{\texttt{Bit}\}32, \texttt{Fp\_double} = \{\texttt{Bit}\}64, \texttt{Fp\_quad} = \{\texttt{Bit}\}128\}.$

$\texttt{nPC} :: \texttt{Constant}.$
$\texttt{IMM13}(1_2) :: \texttt{Bit}.$
$\texttt{OP:ARM}(10_2), \texttt{OP:LS}(11_2), \texttt{OP:BRS}(00_2), \texttt{OP:CLL}(01_2) :: \{\texttt{Bit}\}2.$
$\texttt{BRZ}(01_2), \texttt{BRLEZ}(10_2), \texttt{BRLZ}(11_2) :: \{\texttt{Bit}\}2.$
$\texttt{ADD}(000_2), \texttt{SUB}(100_2), \texttt{SAVE}(100_2), \texttt{RESTORE}(101_2) :: \{\texttt{Bit}\}3.$
$\texttt{BPA}(1000_2), \texttt{BPN}(0000_2), \texttt{BPNE}(1001_2), \texttt{BPE}(0001_2) :: \{\texttt{Bit}\}4.$
$\texttt{LDSTUB}(00\,11\,01_2) :: \{\texttt{Bit}\}6.$
$\texttt{FADD}(0\,01\,00\,00_2), \texttt{FSUB}(0\,01\,00\,01_2) :: \{\texttt{Bit}\}7.$
$\texttt{REGRES}(0\,00\,00\,00\,00_2) :: \{\texttt{Bit}\}9.$

$\texttt{rr}(0,0) = .$ *# reading r[0] yields 0*
$\texttt{rr}(a,v) = a[v] \quad \dashv\{\ 1 \le a < 8\ \}\Rightarrow .$ *# global registers*
$\quad = \texttt{CWP}[\texttt{w}]\ \texttt{outr}(a-8,v,\texttt{w}) \quad \dashv\{\ 8 \le a < 16\ \}\Rightarrow .$
$\quad = \texttt{CWP}[\texttt{w}]\ \texttt{locr}(a-16,v,\texttt{w}) \quad \dashv\{\ 16 \le a < 24\ \}\Rightarrow .$
$\quad = \texttt{CWP}[\texttt{w}]\ \texttt{inr}(a-24,v,\texttt{w}) \quad \dashv\{\ 24 \le a < 32\ \}\Rightarrow .$

$\texttt{rw}(a,v)\ a = \texttt{CWP}[\texttt{w}]\ \texttt{rw}(a,v,\texttt{w}).$
$\texttt{rw}(a,v,\texttt{w}) = \dashv\{\ 0 = a\ \}\Rightarrow .$ *# writing r[0] has no effect*
$\quad = \dashv\{\ 1 \le a < 8\ \}\Rightarrow a[v].$ *# global registers*
$\quad = \dashv\{\ 8 \le a < 16\ \}\Rightarrow \texttt{outw}(a-8,v,\texttt{w}).$
$\quad = \dashv\{\ 16 \le a < 24\ \}\Rightarrow \texttt{locw}(a-16,v,\texttt{w}).$
$\quad = \dashv\{\ 24 \le a < 32\ \}\Rightarrow \texttt{inw}(a-24,v,\texttt{w}).$

$\mathtt{outr}(a, v, w) = \mathtt{ilocr}(a, v, w).$
$\mathtt{outw}(a, v, w) = \Longrightarrow \mathtt{ilocw}(a, v, w).$
$\mathtt{locr}(a, v, w) = \mathtt{ilocr}(a + 8, v, w).$
$\mathtt{locw}(a, v, w) = \Longrightarrow \mathtt{ilocw}(a + 8, v, w).$
$\mathtt{inr}(a, v, w) = \mathtt{ilocr}(a, v, w + 1).$
$\mathtt{inw}(a, v, w) = \Longrightarrow \mathtt{ilocw}(a, v, w + 1).$

$\mathtt{ilocr}(a, v, w) = (\mathtt{NWINDOWS} - 1 - (w \,\%\, \mathtt{NWINDOWS})) \times 16 + a[v].$
$\mathtt{ilocw}(a, v, w) = \Longrightarrow (\mathtt{NWINDOWS} - 1 - (w \,\%\, \mathtt{NWINDOWS})) \times 16 + a[v].$

$\mathtt{fr}(a, v, s)\ a = a[(v :: \mathtt{Fp\_single})] =\!\!\{\ s = 01_2\ \}\!\!\Rightarrow .$
$= a[(v :: \mathtt{Fp\_double})] =\!\!\{\ s = 10_2\ \}\!\!\Rightarrow .$
$= a[(v :: \mathtt{Fp\_quad})] =\!\!\{\ s = 11_2\ \}\!\!\Rightarrow .$
$\mathtt{fw}(a, v, s)\ a = =\!\!\{\ s = 01_2\ \}\!\!\Rightarrow a[(v :: \mathtt{Fp\_single})].$
$= =\!\!\{\ s = 10_2\ \}\!\!\Rightarrow a[(v :: \mathtt{Fp\_double})].$
$= =\!\!\{\ s = 11_2\ \}\!\!\Rightarrow a[(v :: \mathtt{Fp\_quad})].$

$\mathtt{sop1}(v) = \mathtt{rr}(a, v)\ a.$
$\mathtt{sop2}(v) = \mathtt{REGRES}(v).$
$= \mathtt{IMM13}(v).$

$\mathtt{REGRES}(v)\ a = \mathtt{rr}(a, v).$ $\quad \#\ \textit{REGRES} = 0_2\ 00\,00\,00\,00_2$
$\mathtt{IMM13}(\mathtt{sign\_ext}(v)) = v :: \{\mathtt{Bit}\}13.$ $\#\ \textit{IMM13} = 1_2$

$\mathtt{iccr}(c) = \mathtt{CCR{:}ICC{:}C}[c].$
$\mathtt{iccrb}(c) = 1_2\ \mathtt{iccr}(c).$
$\mathtt{iccrb}(0) = 0_2.$

$\mathtt{ccw}(v) = \Longrightarrow \mathtt{CCR{:}XCC}[(v <_{4s} 0)\ (v = 0)\ (\neg(\mathtt{MIN}_{4s} \leq_{4s} v \leq_{4s} \mathtt{MAX}_{4s}))\ (v >_{4u} \mathtt{MAX}_{4u})]$
$\mathtt{CCR{:}ICC}[(v <_{2s} 0)\ (v = 0)\ (\neg(\mathtt{MIN}_{2s} \leq_{2s} v \leq_{2s} \mathtt{MAX}_{2s}))\ (v >_{2u} \mathtt{MAX}_{2u})].$
$\mathtt{ccwb}(v) = 0_2.$
$= 1_2 \Longrightarrow \mathtt{ccw}(v).$

$c :: \mathtt{Bit}.$

$\mathtt{aloper}(x, y, x + y + c) = 0_2\ \mathtt{ccwb}(\$3)\ \mathtt{iccrb}(c)\ \mathtt{ADD}.$ $\#\ \textit{ADD} = 000_2$
$\mathtt{aloper}(x, y, x - y - c) = 0_2\ \mathtt{ccwb}(\$3)\ \mathtt{iccrb}(c)\ \mathtt{SUB}.$ $\#\ \textit{SUB} = 100_2$

n :: Bit.

op :: {Bit}2.

neg(v) $= \neg \rightleftharpoons\!\lfloor\, v = 1_2 \,\rfloor\!\Rightarrow$;

$\Longrightarrow$. # *empty body if* $v \neq 1_2$

lop(op) = & $\rightleftharpoons\!\lfloor\, op = 01_2 \,\rfloor\!\Rightarrow$. # *bit-wise and*

| $\rightleftharpoons\!\lfloor\, op = 10_2 \,\rfloor\!\Rightarrow$. # *bit-wise or*

^ $\rightleftharpoons\!\lfloor\, op = 11_2 \,\rfloor\!\Rightarrow$. # *bit-wise xor*

aloper(x, y, neg(n) (x lop(op) y)) = 0_2 ccwb($3) 0_2 n op.

PC[pc] pc[*pc*()]qc nPC[pc′] $\Longrightarrow$ PC[pc′] pc′[*pc*()]qc nPC[pc′ + 4].

OP:ARM rw(_, r) aloper(x, y, r) sop1(x) sop2(y) $\Longrightarrow$. # *OP:ARM* $= 10_2$

sr(x, y, x + y + c, w + 1) =

1_0 ccwb($3) iccr(c) SAVE CWP[w] $\Longrightarrow$. # *SAVE* $= 100_2$

sr(x, y, x + y + c, w − 1) =

1_0 ccwb($3) iccr(c) RESTORE CWP[w] $\Longrightarrow$. # *RESTORE* $= 101_2$

OP:ARM rw(a, r, w) a sr(x, y, r, w) sop1(x) sop2(y) $\Longrightarrow$ CWP[w % NWINDOWS].

r :: Byte.

ldstub(a, r) = LDSTUB a[r] $\Longrightarrow$ a[11111111_2]. # *LDSTUB* $= 00\,11\,01_2$

OP:LS rw(_, r) ldstub(x + y, r) sop1(x) sop2(y) $\Longrightarrow$. # *OP:LS* $= 11_2$

signed(s) = zero_fill $\rightleftharpoons\!\lfloor\, s = 0 \,\rfloor\!\Rightarrow$.

= sign_ext $\rightleftharpoons\!\lfloor\, s = 1 \,\rfloor\!\Rightarrow$.

{Integer = Byte | Halfword | Word}.

asi :: Byte.

sopasi(x, y, asi) = sop1(x) 0_2 asi sop1(y).

= sop1(x) IMM13(y) ASI[asi].

s :: Bit.

lias(asi, a, signed(s) (r)) = s 0_2 Integer asi:a[(r :: Integer)].

lias(asi, a, r) = 1011_2 asi:a[(r :: Extended_word)].

OP:LS rw(_, r) 01_2 lias(asi, x + y, r) sopasi(x, y, asi) $\Longrightarrow$.

sias(asi, a, r) = 01_2 Integer $\Longrightarrow$ asi:a[(r :: Integer)].

sias(asi, a, r) = 1110_2 $\Longrightarrow$ asi:a[(r :: Extended_word)].

OP:LS rr(_, r) 01_2 sias(asi, x + y, r) sopasi(x, y, asi) $\Longrightarrow$.

s :: {Bit}2.
farithm(x, y, x + y) = FADD. # *FADD* = $0\,01\,00\,00_2$
farithm(x, y, x − y) = FSUB. # *FSUB* = $0\,01\,00\,01_2$
OP:ARM fw(_, r, s) 110100_2 fr(_, x, s) farithm(x, y, r) s fr(_, y, s) ⟹ .

fcmp(x, y) = =[x = y]⇛ 00_2.
=[x < y]⇛ 01_2.
=[x > y]⇛ 10_2.
=[x ? y]⇛ 11_2. # *unordered (x or/and y is NaN)*

v :: {Bit}2.
fccnr(fcc, v) = FSR:FCC:fcc[v].
fccnw(fcc, v) = ⟹ FSR:FCC:fcc[v].

OP:ARM 000_2 cc1 cc0 110101_2 fr(_, x, s) 0010100_2 s fr(_, y, s) ⟹
fccnw(cc1 ‖ cc0, fcmp(x, y)).

ldfa(asi, a, r, 01) = 00_2 asi:a[(r :: Fp_single)]. # *LDFA*
ldfa(asi, a, r, 10) = 11_2 asi:a[(r :: Fp_double)]. # *LDDFA*
ldfa(asi, a, r, 11) = 10_2 asi:a[(r :: Fp_quad)]. # *LDQFA*
OP:LS fw(_, r, s) 1100_2 ldfa(asi, x + y, r, s) sopasi(x, y, asi) ⟹ .

stfa(asi, a, r, 01) = 00_2 ⟹ asi:a[(r :: Fp_single)]. # *STFA*
stfa(asi, a, r, 10) = 11_2 ⟹ asi:a[(r :: Fp_double)]. # *STDFA*
stfa(asi, a, r, 11) = 10_2 ⟹ asi:a[(r :: Fp_quad)]. # *STQFA*
OP:LS fr(_, r, s) 1101_2 stfa(asi, x + y, r, s) sopasi(x, y, asi) ⟹ .

PC[pc] nPC[npc] pc[OP:ARM rw(_, pc) 111000_2 sop1(x) sop2(y)] ⟹ PC[npc] nPC[x+y].

a, p, n :: Bit. # *annul, prediction and "negate condition" bits*
d16hi :: {Bit}2. # *2-bit PC-relative displacement*
rs1 :: {Bit}5. # *address of the 1st source register*
d16lo :: {Bit}14. # *14-bit PC-relative displacement*

bpr_cond(v, neg(n) (v = 0)) = n BRZ. # *Branch on Register Zero*
bpr_cond(v, neg(n) (v ≤ 0)) = n BRLEZ. # *" Register Less Than or Equal to Zero*
bpr_cond(v, neg(n) (v < 0)) = n BRLZ. # *" Register Less Than Zero*

ea(pc, displ) pc + (4 × sign_ext(displ)) = .

PC[pc] nPC[npc] pc[OP:BRS a 0_2 bpr_cond(v, c) 011_2 d16hi p rw(rs1, v) d16lo]
$\quad =\!\!\!\!\!\!\{$ c $\}\!\!\!\!\!\!\Rightarrow$ PC[npc] nPC[ea(pc, d16hi ‖ d16lo)].
" $=\!\!\!\!\!\!\{$ $\neg$c $\wedge$ a = 0 $\}\!\!\!\!\!\!\Rightarrow$ PC[npc] nPC[npc + 4]. # *instruction in delay slot executed*
" $=\!\!\!\!\!\!\{$ $\neg$c $\wedge$ a = 1 $\}\!\!\!\!\!\!\Rightarrow$ PC[npc + 4] nPC[npc + 8]. # *instruction in delay slot annulled*

n, z, v, c :: Bit. # *bits of the ICC/XCC register*
cc1, cc0 :: Bit. # *condition codes selection*
cond :: {Bit}4. # *the condition field*
disp19 :: {Bit}19. # *branch's PC-relative displacement*

bp_cond(_, TRUE) = BPA. # *Branch Always*
bp_cond(_, FALSE) = BPN. # " *Never*
bp_cond(ncc, $\neg$z) = CCR:ncc[n z v c] BPNE. # " *on* $\neq$
bp_cond(ncc, z) = CCR:ncc[n z v c] BPE. # " *on* =
bp_cond(ncc, $\neg$(z $\vee$ (n ^ v))) = CCR:ncc[n z v c] BPG. # " *on* >
bp_cond(ncc, z $\vee$ (n ^ v)) = CCR:ncc[n z v c] BPLE. # " *on* $\leq$
bp_cond(ncc, $\neg$(n ^ v)) = CCR:ncc[n z v c] BPGE. # " *on* $\geq$
bp_cond(ncc, n ^ v) = CCR:ncc[n z v c] BPL. # " *on* <
bp_cond(ncc, $\neg$(c $\vee$ z)) = CCR:ncc[n z v c] BPGU. # " *on* > *Unsigned*
bp_cond(ncc, c $\vee$ z) = CCR:ncc[n z v c] BPLEU. # " *on* $\leq$ *Unsigned*
bp_cond(ncc, $\neg$c) = CCR:ncc[n z v c] BPCC. # " *on* $\geq$ *Unsigned*
bp_cond(ncc, c) = CCR:ncc[n z v c] BPCS. # " *on* < *Unsigned*
bp_cond(ncc, $\neg$n) = CCR:ncc[n z v c] BPPOS. # " *on Positive*
bp_cond(ncc, n) = CCR:ncc[n z v c] BPNEG. # " *on Negative*
bp_cond(ncc, $\neg$v) = CCR:ncc[n z v c] BPVC. # " *on Overflow Clear*
bp_cond(ncc, v) = CCR:ncc[n z v c] BPVS. # " *on Overflow Set*

PC[pc] nPC[npc] pc:COND[cond]
pc[OP:BRS a]pc:COND[bp_cond(cc1 ‖ cc0, c) 001_2 cc1 cc0 p disp19]
$\quad =\!\!\!\!\!\!\{$ $\neg$c $\wedge$ a = 0 $\}\!\!\!\!\!\!\Rightarrow$ PC[npc] nPC[npc + 4].
" $=\!\!\!\!\!\!\{$ $\neg$c $\wedge$ a = 1 $\}\!\!\!\!\!\!\Rightarrow$ PC[npc + 4] nPC[npc + 8].
" $=\!\!\!\!\!\!\{$ c $\wedge$ a = 0 $\}\!\!\!\!\!\!\Rightarrow$ PC[npc] nPC[ea(pc, disp19)].
" $=\!\!\!\!\!\!\{$ c $\wedge$ a = 1 $\wedge$ cond = BPA $\}\!\!\!\!\!\!\Rightarrow$ PC[ea(pc, disp19)] nPC[ea(pc, disp19) + 4].
" $=\!\!\!\!\!\!\{$ c $\wedge$ a = 1 $\wedge$ cond $\neq$ BPA $\}\!\!\!\!\!\!\Rightarrow$ PC[npc] nPC[ea(pc, disp19)].

fbp_cond(_, TRUE) = FBPA. # *Branch Always*
fbp_cond(_, FALSE) = FBPN. # " *Never*
fbp_cond(fcc, v = U) = fccnr(fcc, v) FBPU. # " *on Unordered*
fbp_cond(fcc, v = G) = fccnr(fcc, v) FBPG. # " *on* >
fbp_cond(fcc, v = U $\vee$ v = G) = fccnr(fcc, v) FBPUG. # " *on Unordered or* >

$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \texttt{L}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPL}.$ # " *on* $<$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \texttt{U} \vee \texttt{v} = \texttt{L}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPUL}.$ # " *on Unordered or* $<$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \texttt{L} \vee \texttt{v} = \texttt{G}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPLG}.$ # " *on* $<$ *or* $>$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \neg\texttt{E}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPNE}.$ # " *on* $\neq$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \texttt{E}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPE}.$ # " *on* $=$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \texttt{U} \vee \texttt{v} = \texttt{E}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPUE}.$ # " *on Unordered or* $=$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \texttt{G} \vee \texttt{v} = \texttt{E}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPGE}.$ # " *on* $\geq$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \neg\texttt{L}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPUGE}.$ # " *on Unordered or* $\geq$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \texttt{L} \vee \texttt{v} = \texttt{E}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPLE}.$ # " *on* $\leq$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \neg\texttt{G}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPULE}.$ # " *on Unordered or* $\leq$
$\texttt{fbp\_cond}(\texttt{fcc}, \texttt{v} = \neg\texttt{U}) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{FBPO}.$ # " *on Ordered*

$\#\,\texttt{e} :: \texttt{Bit}.\ \texttt{lgu} :: \{\texttt{Bit}\}3.$
$\#\,\texttt{fbp\_cond}(\texttt{fcc}, \texttt{neg}(\texttt{e})\ (\mathit{dm}(\neg\texttt{v})\ \&\ (\neg\texttt{lgu} + 1))) = \texttt{fccnr}(\texttt{fcc}, \texttt{v})\ \texttt{e}\ \texttt{lgu}.$

$\texttt{PC[pc]}\ \texttt{nPC[npc]}\ \texttt{pc:COND[cond]}$
$\texttt{pc[OP:BRS a]pc:COND[fbp\_cond(cc1} \parallel \texttt{cc0, c)}\ 101_2\ \texttt{cc1 cc0 p disp19]}$
$\quad =\!\!\{\ \neg\texttt{c} \wedge \texttt{a} = 0\ \}\!\!\Rightarrow \texttt{PC[npc]}\ \texttt{nPC[npc} + 4\texttt{]}.$
$\quad \text{"} =\!\!\{\ \neg\texttt{c} \wedge \texttt{a} = 1\ \}\!\!\Rightarrow \texttt{PC[npc} + 4\texttt{]}\ \texttt{nPC[npc} + 8\texttt{]}.$
$\quad \text{"} =\!\!\{\ \texttt{c} \wedge \texttt{a} = 0\ \}\!\!\Rightarrow \texttt{PC[npc]}\ \texttt{nPC[ea(pc, disp19)]}.$
$\quad \text{"} =\!\!\{\ \texttt{c} \wedge \texttt{a} = 1 \wedge \texttt{cond} = \texttt{FBPA}\ \}\!\!\Rightarrow \texttt{PC[ea(pc, disp19)]}\ \texttt{nPC[ea(pc, disp19)} + 4\texttt{]}.$
$\quad \text{"} =\!\!\{\ \texttt{c} \wedge \texttt{a} = 1 \wedge \texttt{cond} \neq \texttt{FBPA}\ \}\!\!\Rightarrow \texttt{PC[npc]}\ \texttt{nPC[ea(pc, disp19)]}.$

$\texttt{disp30} :: \{\texttt{Bit}\}30.$
$\texttt{PC[pc]}\ \texttt{nPC[npc]}\ \texttt{CWP[w]}\ \texttt{pc[OP:CLL disp30]} \qquad \#\,\texttt{OP:CLL} = 01_2$
$\quad \Longrightarrow \texttt{PC[npc]}\ \texttt{nPC[ea(pc, disp30)]}\ \texttt{rw(15, pc, w)}.$

$\texttt{PC[pc]}\ \texttt{nPC[npc]}\ \texttt{CWP[w]}\ \texttt{pc[OP:ARM}\ 00000_2\ 111001_2\ \texttt{sop1(x)}\ \texttt{sop2(y)]}$
$\quad \Longrightarrow \texttt{PC[npc]}\ \texttt{nPC[x} + \texttt{y]}\ \texttt{CWP[(w} - 1) \,\%\, \texttt{NWINDOWS]}.$

$\texttt{OP:BRS}\ 00000_2\ 100_2\ 00_2\ 00000_2\ 00000_2\ 00000_2\ 00000_2 \Longrightarrow .$

```
1  PC[pc] nPC[npc] pc[OP:ARM rw(_,pc) 111000₂ sop1(x) sop2(y)]
2          ⟹ PC[npc] nPC[x + y]. # [3.22]
3  # [2.2] rw(a, v)  a = CWP[w]  rw(a, v, w).                    _ = a, pc = v

4  PC[pc] nPC[npc] CWP[w] pc[OP:ARM rw(a, v, w) a 111000₂ sop1(x) sop2(y)]
5          ⟹ PC[npc] nPC[x + y].
6  # [3.1] sop1(v₁) = rr(a₁, v₁)  a₁.                            x = v₁

7  PC[pc] nPC[npc] CWP[w] pc[OP:ARM rw(a, v, w) a 111000₂ rr(a₁, v₁) a₁ sop2(y)]
8          ⟹ PC[npc] nPC[v₁ + y].
9  # [3.1] sop2(v₂) = IMM13(v₂).                                 y = v₂

10 PC[pc] nPC[npc] CWP[w] pc[OP:ARM rw(a, v, w) a 111000₂ rr(a₁, v₁) a₁ IMM13(v₂)]
11         ⟹ PC[npc] nPC[v₁ + v₂].
12 # [2.2] rw(a, v, w) = =⌊ 1 ≤ a < 8 ⌋⇛ a[v].

13 PC[pc] nPC[npc] CWP[w] pc[OP:ARM a 111000₂ rr(a₁, v₁) a₁ IMM13(v₂)]
14         =⌊ 1 ≤ a < 8 ⌋⇛ PC[npc] nPC[v₁ + v₂] a[v].
15 # [2.1] rr(a₁, v₁) = CWP[w₁]  inr(a₁ − 24, v₁, w₁) =⌊ 24 ≤ a₁ < 32 ⌋⇛.

16 PC[pc] nPC[npc] CWP[w] CWP[w₁] pc[OP:ARM a 111000₂ inr(a₁ − 24, v₁, w₁) a₁ IMM13(v₂)]
17         =⌊ (1 ≤ a < 8) ∧ (24 ≤ a₁ < 32) ⌋⇛ PC[npc] nPC[v₁ + v₂] a[v].
18 # [3.2] IMM13(sign_ext(v₃)) = (v₃ :: {Bit}13).                v₂ = sign_ext(v₃)

19 PC[pc] nPC[npc] CWP[w] CWP[w₁]
20 pc[OP:ARM a 111000₂ inr(a₁ − 24, v₁, w₁) a₁ IMM13 (v₃ :: {Bit}13)]
21         =⌊ (1 ≤ a < 8) ∧ (24 ≤ a₁ < 32) ⌋⇛ PC[npc] nPC[v₁ + v₂] a[v].
22 # [2.3] inr(a₂, v₄, w₁) = ilocr(a₂, v₄, w₁ + 1).              a₁−24 = a₂, v₁ = v₄

23 PC[pc] nPC[npc] CWP[w] CWP[w₁]
24 pc[OP:ARM a 111000₂ ilocr(a₂, v₄, w₁ + 1) a₁ IMM13 (v₃ :: {Bit}13)]
25         =⌊ (1 ≤ a < 8) ∧ (24 ≤ a₁ < 32) ⌋⇛ PC[npc] nPC[v₁ + v₂] a[v].
26 # [2.4] ilocr(a₂, v₄, w₂) = (NWINDOWS − 1 − (w₂ % NWINDOWS)) × 16 + a₂[v₄].
27 #                                                             w₁ + 1 = w₂

28 PC[pc] nPC[npc] CWP[w] CWP[w₁] (NWINDOWS − 1 − (w₂ % NWINDOWS)) × 16 + a₂[v₄]
29 pc[OP:ARM a 111000₂ a₁ IMM13 (v₃ :: {Bit}13)]
30         =⌊ (1 ≤ a < 8) ∧ (24 ≤ a₁ < 32) ⌋⇛ PC[npc] nPC[v₁ + v₂] a[v].

31 # after resolution and rearrangement of locators and text
32 PC[pc] nPC[npc] CWP[w] (NWINDOWS − 1 − ((w + 1) % NWINDOWS)) × 16 + (a₁ − 24)[x]
33 pc[OP:ARM _ 111000₂ a₁ IMM13 (v₃ :: {Bit}13)] # field format of current instruction
34         =⌊ (1 ≤ _ < 8) ∧ (24 ≤ a₁ < 32) ⌋⇛  # conditions for application
35 PC[npc] nPC[x + sign_ext(v₃)] _[pc]. # effects of the instruction on configuration
```

Figure C.1: Example of archetype expansion

Appendix D

Java Virtual Machine

```
{Bit}.
{Boolean = {Bit}32, Byte = {Bit}32, Char = {Bit}32, Short = {Bit}32,
 Int = {Bit}32, Float = {Bit}32, Reference = {Bit}32,
 ReturnAddress = {Bit}32, Long = {Bit}64, Double = {Bit}64}.

{Category1  = Boolean|Byte|Char|Short|Int|Float|Reference|ReturnAddress,
 Category2  = Long|Double,
 Category12 = Category1|Category2}.

{UnsignedByte = {Bit}8, SignedByte = {Bit}8,
 UnsignedShort = {Bit}16, SignedShort = {Bit}16}.

{Word = Category1 Category1|Category2}.

POP(87), POP2(88), DUP(89), DUP_X1(90), DUP_X2(91), DUP2(92)
DUP2_X1(93), DUP2_X2(94), SWAP(95),
ILOAD(21), ILOAD_0(26), ILOAD_1(27), ILOAD_2(28), ILOAD_3(29),
ISTORE(54), ISTORE_0(59), ISTORE_1(60), ISTORE_2(61), ISTORE_3(62),
IADD(96), ISUB(100), IMUL(104), IDIV(108), IREM(112),
IAND(126), IOR(128), IXOR(130), ISHL(120), ISHR(122), IUSHR(124),
INEG(116), IINC(132),
ICONST_M1(2), ICONST_0(3), ICONST_1(4), ICONST_2(5), ICONST_3(6), ICONST_4(7),
ICONST_5(8),
BIPUSH(16), SIPUSH(17),
IFEQ(153), IFNE(154), IFLT(155), IFGE(156), IFGT(157), IFLE(158),
IF_ICMPEQ(159), IF_ICMPNE(160), IF_ICMPLT(161), IF_ICMPGE(162), IF_ICMPGT(163),
IF_ICMPLE(164),
GOTO(167), GOTO_W(200), JSR(168), JSR_W(201), RET(169),
WIDE(196) :: UnsignedByte.
```

b_s :: SignedByte.

b_u, opc :: UnsignedByte.

s_s, δ_s :: SignedShort.

s_u :: UnsignedShort.

i, j, δ_i :: Int.

z :: Category12.

POP $SP[t]\ s[v]t \Longrightarrow SP[s]$.

POP2 $SP[t]\ s[w]t \Longrightarrow SP[s]$.

DUP $SP[t]\ s[v]t \Longrightarrow SP[t']\ s[v\ v]t'$.

DUP2 $SP[t]\ s[w]t \Longrightarrow SP[t']\ s[w\ w]t'$.

DUP_X1 $SP[t]\ s[v_1\ v_0]t \Longrightarrow SP[t']\ s[v_0\ v_1\ v_0]t'$.

DUP2_X1 $SP[t]\ s[v_1\ w_0]t \Longrightarrow SP[t']\ s[w_0\ v_1\ w_0]t'$.

DUP_X2 $SP[t]\ s[w_1\ v_0]t \Longrightarrow SP[t']\ s[v_0\ w_1\ v_0]t'$.

DUP2_X2 $SP[t]\ s[w_1\ w_0]t \Longrightarrow SP[t']\ s[w_0\ w_1\ w_0]t'$.

SWAP $SP[t]\ s[v_1\ v_0]t \Longrightarrow SP[t']\ s[v_0\ v_1]t$.

v :: Category1.

w :: Word.

$push(v) = popsh(, v)$.

$pop(v) = popsh(v,)$.

$var(n, z)\ n[z] = .$

$\alpha(b_{u1}, b_{u2})\ (((b_{u1} << 8)|b_{u2}) :: \text{UnsignedShort}) = .$

$\delta(b_{u1}, b_{u2})\ (((b_{u1} << 8)|b_{u2}) :: \text{SignedShort}) = .$

$\delta(b_{u1}, b_{u2}, b_{u3}, b_{u4})\ ((b_{u1} << 24)|(b_{u2} << 16)|(b_{u3} << 8)|b_{u4}) = .$

$wide(opc, b_u) = opc\ b_u$.

$wide(opc, b_u, b_s) = opc\ b_u\ b_s$.

$wide(opc, \alpha(b_{u1}, b_{u2})) = \text{WIDE}\ opc\ b_{u1}\ b_{u2}$.

$wide(opc, \alpha(b_{u1}, b_{u2}), \delta(b_{u3}, b_{u4})) = \text{WIDE}\ opc\ b_{u1}\ b_{u2}\ b_{u3}\ b_{u4}$.

$i3(0) = 0.\quad i3(1) = 1.\quad i3(2) = 2.\quad i3(3) = 3.$

$iload(i) = wide(\text{ILOAD}, i)$.

$iload(i) = \text{ILOAD\_}\tilde{}i3(i)$.

$istore(i) = wide(\text{ISTORE}, i)$.

$istore(i) = \text{ISTORE\_}\tilde{}i3(i)$.

$iload(n)\ var(n, v)\ push(v) \Longrightarrow .$

$istore(n)\ pop(v) \Longrightarrow var(n, v)$.

$\texttt{iarithm\_binary}(i_1, i_2, i_1 + i_2) = \texttt{IADD}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 - i_2) = \texttt{ISUB}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 \times i_2) = \texttt{IMUL}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1/i_2) = \texttt{IDIV} \rightleftharpoons\{ i_2 \neq 0 \}\Rightarrow .$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 \,\%\, i_2) = \texttt{IREM} \rightleftharpoons\{ i_2 \neq 0 \}\Rightarrow .$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 \,\&\, i_2) = \texttt{IAND}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 \mid i_2) = \texttt{IOR}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 \,\hat{}\, i_2) = \texttt{IXOR}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 << i_2) = \texttt{ISHL}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 >> i_2) = \texttt{ISHR}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_1 >>> i_2) = \texttt{IUSHR}.$
$\texttt{iarithm\_binary}(i_1, i_2, i_r)\ \texttt{popsh}(i_1\ i_2, i_r) \Longrightarrow .$

$\texttt{iarithm\_monadic}(i_1, -i_1) = \texttt{INEG}.$
$\texttt{iarithm\_monadic}(i, i_r)\ \texttt{popsh}(i, i_r) \Longrightarrow .$

$\texttt{wide}(\texttt{IINC}, s_u, s_s)\ \texttt{var}(s_u, i) \Longrightarrow \texttt{var}(s_u, i + s_s).$

$\texttt{i5}(-1) = \texttt{M1}.\quad \texttt{i5}(0) = 0.\quad \texttt{i5}(1) = 1.\quad \texttt{i5}(2) = 2.\quad \texttt{i5}(3) = 3.\quad \texttt{i5}(4) = 4.\quad \texttt{i5}(5) = 5.$
$\texttt{iconst}(i) = \texttt{ICONST\_}\tilde{}\texttt{i5}(i).$

$\texttt{iconst}(b_s) = \texttt{BIPUSH}\ b_s.$
$\texttt{iconst}(\delta(b_{u1}, b_{u2})) = \texttt{SIPUSH}\ b_{u1}\ b_{u2}.$
$\texttt{iconst}(i)\ \texttt{push}(i) \Longrightarrow .$

$\texttt{jmpc}(i, i = 0, \delta(b_1, b_2)) = \texttt{IFEQ}\ b_1, b_2.\quad \texttt{jmpc}(i, j, i = j, \delta(b_1, b_2)) = \texttt{IF\_ICMPEQ}\ b_1\ b_2.$
$\texttt{jmpc}(i, i \neq 0, \delta(b_1, b_2)) = \texttt{IFNE}\ b_1\ b_2.\quad \texttt{jmpc}(i, j, i \neq j, \delta(b_1, b_2)) = \texttt{IF\_ICMPNE}\ b_1\ b_2.$
$\texttt{jmpc}(i, i < 0, \delta(b_1, b_2)) = \texttt{IFLT}\ b_1\ b_2.\quad \texttt{jmpc}(i, j, i < j, \delta(b_1, b_2)) = \texttt{IF\_ICMPLT}\ b_1\ b_2.$
$\texttt{jmpc}(i, i \geq 0, \delta(b_1, b_2)) = \texttt{IFGE}\ b_1\ b_2.\quad \texttt{jmpc}(i, j, i \geq j, \delta(b_1, b_2)) = \texttt{IF\_ICMPGE}\ b_1\ b_2.$
$\texttt{jmpc}(i, i > 0, \delta(b_1, b_2)) = \texttt{IFGT}\ b_1\ b_2.\quad \texttt{jmpc}(i, j, i > j, \delta(b_1, b_2)) = \texttt{IF\_ICMPGT}\ b_1\ b_2.$
$\texttt{jmpc}(i, i \leq 0, \delta(b_1, b_2)) = \texttt{IFLE}\ b_1\ b_2.\quad \texttt{jmpc}(i, j, i \leq j, \delta(b_1, b_2)) = \texttt{IF\_ICMPLE}\ b_1\ b_2.$

$\texttt{jmpu}(\texttt{TRUE}, \delta(b_1, b_2)) = \texttt{GOTO}\ b_1\ b_2.$
$\texttt{jmpu}(\texttt{TRUE}, \delta(b_1, b_2, b_3, b_4)) = \texttt{GOTO\_W}\ b_1\ b_2\ b_3\ b_4.$

$\begin{aligned}\texttt{jump}(\texttt{cond}, \delta_i) &= \texttt{jmpc}(i, \texttt{cond}, \delta_i)\ \texttt{pop}(i).\\ &= \texttt{jmpc}(i, j, \texttt{cond}, \delta_i)\ \texttt{pop}(i\ j).\\ &= \texttt{jmpu}(\texttt{cond}, \delta_i).\end{aligned}$

$\begin{aligned}\texttt{PC}[pc]\ \texttt{pc}[\texttt{jump}(\texttt{cond}, \delta_i)]qc &\rightleftharpoons\{ \texttt{cond} \}\Rightarrow \quad \texttt{PC}[pc + \delta_i].\\ \text{"} &\rightleftharpoons\{ \neg\texttt{cond} \}\Rightarrow \texttt{PC}[qc].\end{aligned}$

$\texttt{jsr}(\delta(\texttt{b}_1, \texttt{b}_2)) \quad = \texttt{JSR b}_1\ \texttt{b}_2.$

$\texttt{jsr}(\delta(\texttt{b}_1, \texttt{b}_2, \texttt{b}_3, \texttt{b}_4)) = \texttt{JSR\_W b}_1\ \texttt{b}_2\ \texttt{b}_3\ \texttt{b}_4.$

$\texttt{PC[pc] pc[jsr}(\delta_\texttt{i})\texttt{]qc push(qc)} \Longrightarrow \texttt{PC[pc} + \delta_\texttt{i}\texttt{]}.$

$\texttt{PC[pc] pc[wide(RET, n)]qc var(n, a)} \Longrightarrow \texttt{PC[a]}.$

Appendix E

n-PRAM

$\{\texttt{MrModel} = \texttt{Int}, \texttt{MwModel} = \texttt{Int}\}$. # *type aliases for the memory read and write models*

$\{\texttt{ER}, \texttt{CR} :: \texttt{MrModel},$
$\texttt{mr} :: \texttt{MrModel}\}.$
$\{\texttt{EW}, \texttt{CW\_WEAK}, \texttt{CW\_COMMON}, \texttt{CW\_TOLERANT}, \texttt{CW\_COLLISION}, \texttt{CW\_COLLISIONP},$
$\texttt{CW\_ARBITRARY}, \texttt{CW\_PRIORITY} :: \texttt{MwModel},$
$\texttt{mw} :: \texttt{MwModel}\}.$

$$\texttt{pram(n, rm, wm)} = {}^{\texttt{FE}}\Big|_{4}^{1,3} \begin{array}{ll} \texttt{pcs(n)} & \#\ \textit{update PCs, block (stage 2) if no RAM is running} \\ \texttt{shr(n, n, rm, } \emptyset\texttt{)} & \#\ \textit{shared memory reads} \\ \texttt{shw(n, n, wm, } \emptyset\texttt{).} & \#\ \textit{shared memory writes} \end{array}$$

$\texttt{pcs(0)} = .$ # *0 RAMs*
$\texttt{pcs(p)} = \texttt{pc(p)}\ \texttt{pcs(p} - 1\texttt{)} \Leftarrow\!\!\lfloor\ \texttt{p} > 0\ \rfloor\!\!\Rightarrow .$

$$\texttt{pc(p)} = {}^{\texttt{FE}}\Big|_{2}^{1} \begin{array}{lll} \texttt{P}\tilde{}\texttt{p:PC[pc]}\ \texttt{pc[instr(p)]qc} \Leftarrow\!\!\lfloor\ 0 \le \texttt{pc} < \texttt{C}\ \rfloor\!\!\Rightarrow\ \texttt{P}\tilde{}\texttt{p:PC[qc]} & \# \textit{running} \\ \Longrightarrow . & \# \textit{don't block} \\ \texttt{pc(p)} = \texttt{P}\tilde{}\texttt{p:PC[pc]} \Leftarrow\!\!\lfloor\ 0 > \texttt{pc} \vee \texttt{pc} \ge \texttt{C}\ \rfloor\!\!\Rightarrow . & \# \textit{not running} \end{array}$$

$\texttt{halt(0)} = .$ # *halt 0 RAMs*
$\texttt{halt(p)} = \texttt{halt(p} - 1\texttt{)} \Leftarrow\!\!\lfloor\ \texttt{p} > 0\ \rfloor\!\!\Rightarrow \texttt{P}\tilde{}\texttt{p:PC[C]}.$ # *halt p RAMs*

$\#\,\texttt{shrd(p, a)} = \texttt{P}\tilde{}\texttt{p:MAR[a]}.$

$$\#\,\texttt{shr(n, CR)} = \overset{n}{\underset{p=1}{\texttt{shrd}}}(p, \texttt{a}_p)\ \overset{n}{\underset{p=1}{\texttt{read}}}(p, \texttt{a}_p).$$

$$\#\,\texttt{shr(n, ER)} = \overset{n}{\underset{p=1}{\texttt{shrd}}}(p, \texttt{a}_p)\ \overset{n}{\underset{p=1}{\texttt{halt}}}(p) \quad \Leftarrow\!\!\left\lfloor\ \left\{\left|\, \texttt{a}_p^{\overline{\texttt{NULL}}} \,\right|\right\} \neq \left\{\, \texttt{a}_p^{\overline{\texttt{NULL}}} \,\right\}\ \right\rfloor\!\!\Rightarrow \quad \# \textit{read conflict}$$

$$\# \quad = \overset{n}{\underset{p=1}{\texttt{shrd}}}(p, \texttt{a}_p)\ \overset{n}{\underset{p=1}{\texttt{read}}}(p, \texttt{a}_p) \Leftarrow\!\!\left\lfloor\ \left\{\left|\, \texttt{a}_p^{\overline{\texttt{NULL}}} \,\right|\right\} = \left\{\, \texttt{a}_p^{\overline{\texttt{NULL}}} \,\right\}\ \right\rfloor\!\!\Rightarrow . \ \# \textit{no read conflict}$$

$\mathtt{shr}(0,\_,\_,\_) \quad = .$ *# 0 RAMs, no conflicts*

$\mathtt{shr}(p,n,CR,\_) = P\tilde{}p{:}MAR[a]\ \mathtt{shr}(p-1,n,CR,\_)\ \mathtt{read}(p,a)$

$\qquad\qquad =\!\!\lfloor\ p > 0\ \rfloor\!\!\Rightarrow .$ *# CR, no conflicts*

$\mathtt{shr}(p,n,ER,s) = P\tilde{}p{:}MAR[a]\ \mathtt{shr}(p-1,n,ER,s)\ \mathtt{halt}(n)$

$\qquad\qquad =\!\!\lfloor\ a \neq \mathtt{NULL} \wedge a \in s \wedge p > 0\ \rfloor\!\!\Rightarrow .$ *# conflict*

$\mathtt{shr}(p,n,ER,s) = P\tilde{}p{:}MAR[a]\ \mathtt{shr}(p-1,n,ER,s \cup \{a\})\ \mathtt{read}(p,a)$

$\qquad\qquad =\!\!\lfloor\ (a = \mathtt{NULL} \vee a \notin s) \wedge p > 0\ \rfloor\!\!\Rightarrow .$ *# no conflict*

$\mathtt{read}(p,a) = a[v] =\!\!\lfloor\ a \neq \mathtt{NULL}\ \rfloor\!\!\Rightarrow P\tilde{}p{:}ACC[v];$ *# read value v from shared memory*

$\qquad\qquad \Longrightarrow .$ *# not a memory read instruction*

$\mathtt{shw}(0,\_,\_,\_) = .$ *# 0 RAMs, no write conflicts*

$\mathtt{shw}(p,n,wm,s) = P\tilde{}p{:}MAR[a]\ \mathtt{shw}(p-1,n,wm,s) =\!\!\lfloor\ a \in s \wedge p > 0\ \rfloor\!\!\Rightarrow$ *# a resolved*

$\qquad\qquad = P\tilde{}p{:}MAR[a]\ \mathtt{shw}(p-1,n,wm,s \cup \{a\})\ \mathtt{sw}(p,n,wm) =\!\!\lfloor\ a \notin s \wedge p > 0\ \rfloor\!\!\Rightarrow .$

$\mathtt{sw}(p,n,EW)$ *# exclusive write*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{ew}(p,a,v,n,n).$

$\mathtt{sw}(p,n,CW\_WEAK)$ *# concurrent write, weak model*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{weak}(p,a,v,n,n).$

$\mathtt{sw}(p,n,CW\_COMMON)$ *# concurrent write, common model*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{common}(p,a,v,n,n).$

$\mathtt{sw}(p,n,CW\_TOLERANT)$ *# concurrent write, tolerant model*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{tolerant}(p,a,v,n,n).$

$\mathtt{sw}(p,n,CW\_COLLISION)$ *# concurrent write, collision model*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{collision}(p,a,v,n,n).$

$\mathtt{sw}(p,n,CW\_COLLISIONP)$ *# concurrent write, collision+ model*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{collisionp}(p,a,v,n,n).$

$\mathtt{sw}(p,n,CW\_ARBITRARY)$ *# concurrent write, arbitrary model*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{arbitrary}(p,a,v,n,n).$

$\mathtt{sw}(p,n,CW\_PRIORITY)$ *# concurrent write, priority model*

$\quad = P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ P\tilde{}p{:}SIG[s]\ \mathtt{priority}(p,a,v,s,n,n).$

$\#\ \{a_1 \ominus a_2 \mid a_1 = a_2 \wedge a_1 \neq \mathtt{NULL}\}$

$\#\ \{a_1 \oslash a_2 \mid a_1 \neq a_2 \vee a_1 = \mathtt{NULL} \vee a_2 = \mathtt{NULL}\}$

$\#\ \mathtt{shwr}(p,a,v) = P\tilde{}p{:}MAR[a]\ \ P\tilde{}p{:}ACC[v].$

$\#\ shw(n,EW)$

$\#\quad = \overset{n}{\underset{p=1}{shwr}}(p,a_p,\_)\ \overset{n}{\underset{p=1}{halt}}(p) \qquad =\!\!\lfloor\!\{\!|\ a_p^{\overline{NULL}}\ |\!\}\!\neq\!\{\ a_p^{\overline{NULL}}\ \}\!\rfloor\!\!\Rightarrow$ *# write conflict*

$\#\quad = \overset{n}{\underset{p=1}{shwr}}(p,a_p,v_p)\ \overset{n}{\underset{p=1}{write}}(a_p,v_p) =\!\!\lfloor\!\{\!|\ a_p^{\overline{NULL}}\ |\!\}\!=\!\{\ a_p^{\overline{NULL}}\ \}\!\rfloor\!\!\Rightarrow .$ *# no write conflict*

$\mathtt{ew}(\_, \mathtt{a}, \mathtt{v}, 0, \_) = \mathtt{write}(\mathtt{a}, \mathtt{v}).$

$\mathtt{ew}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}, \mathtt{n})$

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{halt(n)}$

$\quad =\!\!\{\ \mathtt{p_t} \neq \mathtt{p} \wedge \mathtt{a_t} \ominus \mathtt{a} \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow \quad$ *# write conflict*

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{ew}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}-1, \mathtt{n})$

$\quad =\!\!\{\ (\mathtt{p_t} = \mathtt{p} \vee \mathtt{a_t} \oslash \mathtt{a}) \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow .$ *# no conflict*

 # shw$(\mathtt{n}, \mathit{CW\_WEAK})$

$\quad = \overset{n}{\underset{p=1}{\mathtt{shwr}}}(p, \mathtt{a}_p, \mathtt{v}_p)\ \overset{n}{\underset{p=1}{\mathtt{halt}}}(p) \qquad$ *# write conflict*

$\quad =\!\!\{\ \exists(\mathtt{a}_p, \mathtt{v}_p), (\mathtt{a}_q, \mathtt{v}_q) \in \left\{ (\mathtt{a}_p^{\overline{NULL}}, \mathtt{v}_p) \right\} \mid p \neq q \wedge \mathtt{a}_p = \mathtt{a}_q \wedge (\mathtt{v}_p \neq 0 \vee \mathtt{v}_q \neq 0)\ \}\!\!\Rightarrow$

$\quad = \overset{n}{\underset{p=1}{\mathtt{shwr}}}(p, \mathtt{a}_p, \mathtt{v}_p)\ \overset{n}{\underset{p=1}{\mathtt{write}}}(\mathtt{a}_p, \mathtt{v}_p)$ *# 0 or no write conflict*

$\quad =\!\!\{\ \forall(\mathtt{a}_p, \mathtt{v}_p), (\mathtt{a}_q, \mathtt{v}_q) \in \left\{ (\mathtt{a}_p^{\overline{NULL}}, \mathtt{v}_p) \right\} \mid \mathtt{a}_p = \mathtt{a}_q \Rightarrow (\mathtt{v}_p = \mathtt{v}_q = 0)\ \}\!\!\Rightarrow .$

$\mathtt{weak}(\_, \mathtt{a}, \mathtt{v}, 0, \_) = \mathtt{write}(\mathtt{a}, \mathtt{v}).$

$\mathtt{weak}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}, \mathtt{n})$

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{P\tilde{}p{:}ACC[v]}\ \mathtt{weak}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}-1, \mathtt{n})$

$\quad =\!\!\{\ \mathtt{p_t} \neq \mathtt{p} \wedge \mathtt{a_t} \ominus \mathtt{a} \wedge \mathtt{v_t} = 0 \wedge \mathtt{v} = 0 \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow \quad$ *# write 0 conflict*

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{P\tilde{}p{:}ACC[v]}\ \mathtt{halt(n)}$

$\quad =\!\!\{\ \mathtt{p_t} \neq \mathtt{p} \wedge \mathtt{a_t} \ominus \mathtt{a} \wedge (\mathtt{v_t} \neq 0 \vee \mathtt{v} \neq 0) \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow$ *# conflict, halt*

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{weak}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}-1, \mathtt{n})$

$\quad =\!\!\{\ (\mathtt{p_t} = \mathtt{p} \vee \mathtt{a_t} \oslash \mathtt{a}) \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow .$ *# no conflict, check further*

 # shw$(\mathtt{n}, \mathit{CW\_COMMON})$

$\quad = \overset{n}{\underset{p=1}{\mathtt{shwr}}}(p, \mathtt{a}_p, \mathtt{v}_p)\ \overset{n}{\underset{p=1}{\mathtt{halt}}}(p)$

$\quad =\!\!\{\ \exists(\mathtt{a}_p, \mathtt{v}_p), (\mathtt{a}_q, \mathtt{v}_q) \in \left\{ (\mathtt{a}_p^{\overline{NULL}}, \mathtt{v}_p) \right\} \mid \mathtt{a}_p = \mathtt{a}_q \wedge \mathtt{v}_p \neq \mathtt{v}_q\ \}\!\!\Rightarrow$

$\quad = \overset{n}{\underset{p=1}{\mathtt{shwr}}}(p, \mathtt{a}_p, \mathtt{v}_p)\ \overset{n}{\underset{p=1}{\mathtt{write}}}(\mathtt{a}_p, \mathtt{v}_p)$

$\quad =\!\!\{\ \forall(\mathtt{a}_p, \mathtt{v}_p), (\mathtt{a}_q, \mathtt{v}_q) \in \left\{ (\mathtt{a}_p^{\overline{NULL}}, \mathtt{v}_p) \right\} \mid \mathtt{a}_p = \mathtt{a}_q \Rightarrow (\mathtt{v}_p = \mathtt{v}_q = 0)\ \}\!\!\Rightarrow .$

$\mathtt{common}(\_, \mathtt{a}, \mathtt{v}, 0, \_) = \mathtt{write}(\mathtt{a}, \mathtt{v}).$

$\mathtt{common}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}, \mathtt{n})$

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{P\tilde{}p{:}ACC[v]}\ \mathtt{common}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}-1, \mathtt{n})$

$\quad =\!\!\{\ \mathtt{p_t} \neq \mathtt{p} \wedge \mathtt{a_t} \ominus \mathtt{a} \wedge \mathtt{v_t} = \mathtt{v} \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow$ *# conflict, same value, proceed*

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{P\tilde{}p{:}ACC[v]}\ \mathtt{halt(n)}$

$\quad =\!\!\{\ \mathtt{a_t} \ominus \mathtt{a} \wedge \mathtt{v_t} \neq \mathtt{v} \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow \qquad$ *# conflict, halt*

$= \mathtt{P\tilde{}p{:}MAR[a]}\ \mathtt{common}(\mathtt{p_t}, \mathtt{a_t}, \mathtt{v_t}, \mathtt{p}-1, \mathtt{n})$

$\quad =\!\!\{\ (\mathtt{p_t} = \mathtt{p} \vee \mathtt{a_t} \oslash \mathtt{a}) \wedge \mathtt{p} > 0\ \}\!\!\Rightarrow . \qquad$ *# no conflict*

$$\#\,shw(n, CW\_TOLERANT) = \overset{n}{\underset{p=1}{shwr}}(p, a_p, v_p)\ \overset{n}{\underset{p=1}{write}}(a_{wp}, v_p)$$

$$\#\,\dashv\!\!\!\dashv\ \exists(a_p, v_p), (a_q, v_q) \in \left\{ (a_p^{\overline{NULL}}, v_p) \right\} \mid (a_{wp} := (p \neq q \wedge a_p = a_q)\,?\,NULL : a_p)\ \vdash\!\!\!\vdash .$$

$\mathtt{tolerant}(\_, a, v, 0, \_) = \mathtt{write}(a, v).$

$\mathtt{tolerant}(p_t, a_t, v_t, p, n)$

$= P\tilde{}p{:}MAR[a]$

$\dashv\!\!\!\dashv\ p_t \neq p \wedge a_t \ominus a \wedge p > 0\ \vdash\!\!\!\vdash$ *# conflict, value not changed*

$= P\tilde{}p{:}MAR[a]\ \mathtt{tolerant}(p_t, a_t, v_t, p-1, n)$ *# no conflict*

$\dashv\!\!\!\dashv\ (p_t = p \vee a_t \oslash a) \wedge p > 0\ \vdash\!\!\!\vdash .$

$$\#\,shw(n, CW\_COLLISION) = \overset{n}{\underset{p=1}{shwr}}(p, a_p, v_p)\ \overset{n}{\underset{p=1}{write}}(a_p, v_{wp})$$

$$\#\,\dashv\!\!\!\dashv\ \exists(a_p, v_p), (a_q, v_q) \in \left\{ (a_p^{\overline{NULL}}, v_p) \right\} \mid (v_{wp} := (p \neq q \wedge a_p = a_q)\,?\,COLL : v_p)\ \vdash\!\!\!\vdash .$$

$\mathtt{collision}(\_, a, v, 0, \_) = \mathtt{write}(a, v).$

$\mathtt{collision}(p_t, a_t, v_t, p, n)$

$= P\tilde{}p{:}MAR[a]$

$\dashv\!\!\!\dashv\ p_t \neq p \wedge a_t \ominus a \wedge p > 0\ \vdash\!\!\!\vdash \mathtt{write}(a_t, COLL)$ *# conflict, write a collision symbol*

$= P\tilde{}p{:}MAR[a]\ \mathtt{collision}(p_t, a_t, v_t, p-1, n)$

$\dashv\!\!\!\dashv\ (p_t = p \vee a_t \oslash a) \wedge p > 0\ \vdash\!\!\!\vdash .$ *# no conflict*

$$\#\,shw(n, CW\_COLLISIONP) = \overset{n}{\underset{p=1}{shwr}}(p, a_p, v_p)\ \overset{n}{\underset{p=1}{write}}(a_p, v_{wp})$$

$$\#\,\dashv\!\!\!\dashv\ \exists(a_p, v_p), (a_q, v_q) \in \left\{ (a_p^{\overline{NULL}}, v_p) \right\} \mid (v_{wp} := (a_p = a_q \wedge v_p \neq v_q)\,?\,COLL : v_p)\ \vdash\!\!\!\vdash .$$

$\mathtt{collisionp}(\_, a, v, 0, \_) = \mathtt{write}(a, v).$

$\mathtt{collisionp}(p_t, a_t, v_t, p, n) =$

$= P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]\ \mathtt{collisionp}(p_t, a_t, v_t, p-1, n)$

$\dashv\!\!\!\dashv\ p_t \neq p \wedge a_t \ominus a \wedge v_t = v \wedge p > 0\ \vdash\!\!\!\vdash$ *# conflict, same value, proceed*

$= P\tilde{}p{:}MAR[a]\ P\tilde{}p{:}ACC[v]$

$\dashv\!\!\!\dashv\ a_t \ominus a \wedge v_t \neq v \wedge p > 0\ \vdash\!\!\!\vdash \mathtt{write}(a_t, COLL)$ *# conflict, write a collision symbol*

$= P\tilde{}p{:}MAR[a]\ \mathtt{collisionp}(p_t, a_t, v_t, p-1, n)$

$\dashv\!\!\!\dashv\ (p_t = p \vee a_t \oslash a) \wedge p > 0\ \vdash\!\!\!\vdash .$ *# no conflict*

$$\#\,shw(n, CW\_ARBITRARY) = \overset{n}{\underset{p=1}{shwr}}(p, a_p, v_p)\ \overset{n}{\underset{p=1}{write}}(a_p, v_{wp})$$

$$\#\,\dashv\!\!\!\dashv\ \exists(a_p, v_p), (a_q, v_q) \in \left\{ (a_p^{\overline{NULL}}, v_p) \right\} \mid (v_{wp} := (p \neq q \wedge a_p = a_q)\,?\,\mathtt{rand}(\{v_p^{a_p}\}) : v_p)\ \vdash\!\!\!\vdash .$$

$\texttt{arbitrary}(\_, \texttt{a}, \texttt{v}, 0, \_) = \texttt{write}(\texttt{a}, \texttt{v}).$

$\texttt{arbitrary}(\texttt{p}_t, \texttt{a}_t, \texttt{v}_t, \texttt{p}, \texttt{n})$

$= \texttt{P\~{}p:MAR[a]}\ \texttt{arbitrary}(\texttt{p}_t, \texttt{a}_t, \texttt{v}_t, \texttt{p}-1, \texttt{n})$

$= \{\!|\ \texttt{p}_t \neq \texttt{p} \wedge \texttt{a}_t \ominus \texttt{a} \wedge \texttt{p} > 0\ |\!\}\!\Rightarrow$ *# conflict, choose* $\texttt{v}_t$

$= \texttt{P\~{}p:MAR[a]}\ \texttt{P\~{}p:ACC[v]}\ \texttt{arbitrary}(\texttt{p}_t, \texttt{a}_t, \texttt{v}, \texttt{p}-1, \texttt{n})$

$= \{\!|\ \texttt{p}_t \neq \texttt{p} \wedge \texttt{a}_t \ominus \texttt{a} \wedge \texttt{p} > 0\ |\!\}\!\Rightarrow$ *# conflict, choose v*

$= \texttt{P\~{}p:MAR[a]}\ \texttt{arbitrary}(\texttt{p}_t, \texttt{a}_t, \texttt{v}_t, \texttt{p}-1, \texttt{n})$

$= \{\!|\ (\texttt{p}_t = \texttt{p} \vee \texttt{a}_t \oslash \texttt{a}) \wedge \texttt{p} > 0\ |\!\}\!\Rightarrow .$ *# no conflict*

$\#\,shw(n, CW\_PRIORITY) = \overset{n}{\underset{p=1}{shwr}}(p, \texttt{a}_p, \texttt{v}_p, s_p)\ \overset{n}{\underset{p=1}{write}}(\texttt{a}_p, \texttt{v}_{wp})$

$\# \{\!|\ \exists(\texttt{a}_p, \texttt{v}_p), (\texttt{a}_q, \texttt{v}_q) \in \left\{ (\texttt{a}_p^{\overline{NULL}}, \texttt{v}_p) \right\} | (\texttt{v}_{wp} := (p \neq q \wedge \texttt{a}_p = \texttt{a}_q)\,?\,min_{s_p}(\{\texttt{v}_p^{\texttt{a}_p}\}) : \texttt{v}_p)\ |\!\}\!\Rightarrow .$

$\texttt{priority}(\_, \texttt{a}, \texttt{v}, \_, 0, \_) = \texttt{write}(\texttt{a}, \texttt{v}).$

$\texttt{priority}(\texttt{p}_t, \texttt{a}_t, \texttt{v}_t, \texttt{s}_t, \texttt{p}, \texttt{n})$

$= \texttt{P\~{}p:MAR[a]}\ \texttt{P\~{}p:ACC[v]}\ \texttt{P\~{}p:SIG[s]}\ \texttt{priority}(\texttt{p}_t, \texttt{a}_t, \texttt{v}_t, \texttt{s}_t, \texttt{p}-1, \texttt{n})$

$= \{\!|\ \texttt{p}_t \neq \texttt{p} \wedge \texttt{a}_t \ominus \texttt{a} \wedge \texttt{s}_t < \texttt{s} \wedge \texttt{p} > 0\ |\!\}\!\Rightarrow$ *# conflict, p has lower priority*

$= \texttt{P\~{}p:MAR[a]}\ \texttt{P\~{}p:ACC[v]}\ \texttt{P\~{}p:SIG[s]}\ \texttt{priority}(\texttt{p}_t, \texttt{a}_t, \texttt{v}, \texttt{s}, \texttt{p}-1, \texttt{n})$

$= \{\!|\ \texttt{p}_t \neq \texttt{p} \wedge \texttt{a}_t \ominus \texttt{a} \wedge \texttt{s}_t > \texttt{s} \wedge \texttt{p} > 0\ |\!\}\!\Rightarrow$ *# conflict, p has higher priority*

$= \texttt{P\~{}p:MAR[a]}\ \texttt{P\~{}p:ACC[v]}\ \texttt{P\~{}p:SIG[s]}\ \texttt{priority}(\texttt{p}_t, \texttt{a}_t, \texttt{v}_t, \texttt{s}_t, \texttt{p}-1, \texttt{n})$

$= \{\!|\ (\texttt{p}_t = \texttt{p} \vee \texttt{a}_t \oslash \texttt{a}) \wedge \texttt{p} > 0\ |\!\}\!\Rightarrow .$ *# no conflict*

$\texttt{write}(\texttt{a}, \texttt{v}) = \{\!|\ \texttt{a} \neq \texttt{NULL}\ |\!\}\!\Rightarrow \texttt{a[v]};$ *# write value* v *into shared memory*

$\Longrightarrow .$ *# not a memory write instruction*

$\texttt{accr}(\texttt{p}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \texttt{P\~{}p:ACC[v]}.$ *# accumulator read*

$\texttt{accw}(\texttt{p}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \Longrightarrow \texttt{P\~{}p:ACC[v]}.$ *# accumulator write*

$\texttt{sigr}(\texttt{p}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \texttt{P\~{}p:SIG[v]}.$ *# signature register read*

$\texttt{sigw}(\texttt{p}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \Longrightarrow \texttt{P\~{}p:SIG[v]}.$ *# signature register write*

$\texttt{pcr}(\texttt{p}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \texttt{P\~{}p:PC[v]}.$ *# program counter read*

$\texttt{pcw}(\texttt{p}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \Longrightarrow \texttt{P\~{}p:PC[v]}.$ *# program counter write*

$\texttt{regr}(\texttt{p}, \texttt{r}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \texttt{P\~{}p:r[v]}.$ *# local register read*

$\texttt{regw}(\texttt{p}, \texttt{r}, \texttt{v}) = {}^{\texttt{FE}}|_2\ \Longrightarrow \texttt{P\~{}p:r[v]}.$ *# local register write*

$\texttt{DR}(\texttt{p}, \texttt{r}, \texttt{v})\ \texttt{r} = \texttt{regr}(\texttt{p}, \texttt{r}, \texttt{v}).$ *# direct register (read)*

$\texttt{IR}(\texttt{p}, \texttt{r}_i, \texttt{v})\ \texttt{r} = \texttt{regr}(\texttt{p}, \texttt{r}, \texttt{r}_i)\ \texttt{regr}(\texttt{p}, \texttt{r}_i, \texttt{v}).$ *# indirect register (read)*

```
rm(p,a,v) = DR(p,a,v)  # register modes
          = IR(p,a,v).

irm(p,v) = IMM v       # immediate mode and register modes
         rm(p,_,v).

DMR(p) m = FE|2  ⟹ P~p:MAR[m]      # direct memory read
           |3  ⟹ P~p:MAR[NULL].

IMR(p) r = FE|2 P~p:r[m_i] ⟹ P~p:MAR[m_i]    # indirect memory read
           |3              ⟹ P~p:MAR[NULL].

mrm(p) = DMR(p)
         IMR(p).

DMW(p) m = FE|3  ⟹ P~p:MAR[m]      # direct memory write
           |4  ⟹ P~p:MAR[NULL].
IMW(p) r = FE|2 P~p:r[m_i] ⟹                # indirect memory write
           |3              ⟹ P~p:MAR[m_i]
           |4              ⟹ P~p:MAR[NULL].

mwm(p) = DMW(p)
         IMW(p).

binary(x,y,x+y) = ADD.      # addition
binary(x,y,x-y) = SUB.      # subtraction
binary(x,y,x×y) = MUL.      # multiplication
binary(x,y,x/y) = DIV.      # division
binary(x,y,x % y) = MOD.    # modulo
binary(x,y,x<<y) = SHIFT.   # shift left
binary(x,y,x & y) = AND.    # bitwise and
binary(x,y,x | y) = OR.     # bitwise or
binary(x,y,x ^ y) = XOR.    # bitwise xor

monadic(x,log(x)) = LOG.    # logarithm
monadic(x,not(x)) = NOT.    # bitwise not

arlog(p,r) = binary(x,y,r) irm(p,y) accr(p,x)
           = monadic(y,r) irm(p,y).

load(p,v) = LOAD irm(p,v)
          = LOADINDEX sigr(p,v)
          = LOADPC pcr(p,v).
```

$$\texttt{toacc(p,v)} = \texttt{arlog(p,v)} \Longrightarrow \texttt{accw(p,v)}$$
$$= \texttt{load(p,v)} \Longrightarrow \texttt{accw(p,v)}.$$

$$\texttt{store(p,r,v)} = \texttt{STORE rm(p,r,\_) accr(p,v)}.$$

$$\texttt{toreg(p,r,v)} = \texttt{store(p,r,v)} \Longrightarrow \texttt{regw(p,r,v)}.$$

$$\texttt{jump(p,v} > 0\texttt{,a)} = \texttt{JPOS irm(p,a) accr(p,v)}.$$
$$\texttt{jump(p,v} = 0\texttt{,a)} = \texttt{JZERO irm(p,a) accr(p,v)}.$$
$$\texttt{jump(p,TRUE,a)} = \texttt{JUMP irm(p,a)}. \qquad \# \textit{unconditional jump}$$
$$\texttt{jump(\_,TRUE,C)} = \texttt{HALT}.$$

$$\texttt{topc(p)} = \texttt{jump(p,cond,a)} =\!\{\ \texttt{cond}\ \}\!\Rightarrow \texttt{pcw(p,a)}$$
$$= \texttt{jump(p,cond,a)} =\!\{\ \neg\texttt{cond}\ \}\!\Rightarrow .$$

$$\texttt{read(p)} = \texttt{READ mrm(p)}.$$

$$\texttt{write(p)} = \texttt{WRITE mwm(p)}.$$

$$\texttt{instr(p)} = \texttt{toacc(p,\_)}$$
$$= \texttt{toreg(p,\_,\_)}$$
$$= \texttt{topc(p,\_)}$$
$$= \texttt{read(p)}$$
$$= \texttt{write(p)}.$$

Appendix F

Glossary

This glossary contains basic terminology for Extended Update Plans. Many terms have been adopted from the original work on Update Plans [60], others have been changed to accommodate for changes introduced in this thesis, and there is also a number of completely new terms exclusive to EUP.

alternatives, 14
: a series of **update schemes**; the first **applicable** scheme in the series is applied

ambidextrous archetype, 18
: a pair of **archetypes** of the same name, one **left-handed**, the other **right-handed**

anonymous sequence, 67
: a **sequence** with no sequential block identifier

applicable, 11
: an **update rule** is applicable if its left-hand side is **consistent** with the current **configuration** and its **guard** evaluates to true

application order, 68
: the order **updates** are applied to a **configuration** in a **sequence**

archetype, 15
: a macro-like mechanism

archetype expansion, 17
: the mechanism by which **archetypes** are expanded to give **update schemes**

canonical form, 70
: a **sequential** or a **parallel block** is in canonical form if it has only one type of **updates**—**update schemes**

casting, 14

forcing the value of a term to be of a different type than the type globally declared for that term

cell, 10

an element of **configuration** or **memory**

command, 13

an **update scheme** in which both the left and right-hand sides are in **command form**

command archetype, 19

an **ambidextrous archetype** whose name starts with a constant

command driven, 13

an **update plan** in which all **update schemes** are **commands**

command form, 13

a **configuration** containing a non-empty **command sequence**, or one in which the contents of the **register** PC are not specified

command sequence, 13

a sequence of terms

configuration, 10

a partial function from **locators** to values; a **consistent** set of **locator expressions**

consistent, 10

a set of **locator expressions** is consistent if it does not specify conflicting contents for one and the same **cell**

context, 17

the non-**text** part of a **configuration**, in particular in an **archetype** body

expansion, 17

see **archetype expansion**; the **text** that actually replaces the **archetype** call

final configuration, 11

a **configuration** to which none of the **update schemes** in an **update plan** are **applicable**

full applicability, 68

a **sequence** is fully **applicable** if all of its constituent **updates** are **applicable** in the **application order**

ground term, 15

a term for which a unique variable free value can be derived

guard, 10
a condition for **applicability**

initial configuration, 11
specifies the initial state of the **memory** before any **update scheme** is applied

left-handed archetype, 18
an **archetype** having an empty right-hand side **expansion**

locator, 10
an index to a **memory**

locator expression, 10
a sequence of **cells** delimited on the left and right by a **locator**

memory, 11
a function from **locators** to values

opcode, 13
the first element of a **command sequence**

parallel block, 20
a set of **update schemes** to be applied simultaneously

parallel block symbol (open), 20
'(||': indicates the start of a **parallel block**

parallel block symbol (close), 20
'||)': indicates the end of a **parallel block**

parameter resolution, 17
the mechanism by which parameters of an **archetype** are rewritten to evaluable expressions

partial applicability, 68
a **sequence** is partially **applicable**, if more than one but not all its constituent **updates** are **applicable** in the **application order**

program counter, 12
the **register** PC

register, 12
a constant **locator**

repeat, 12
'"': indicates a repeat of the previous left-hand side

right-handed archetype, 18
an **archetype** having an empty left-hand side **expansion**

semi-ground term, 15
a term for which a finite number of variable free values can be derived

sequence, 67
a synonym for a **sequential update scheme**

sequencer, 67
a synonym for the sequential block identifier

sequential archetype, 78
an **archetype** whose body is a **sequential block**

sequential block, 67
a set of **updates** to be applied sequentially

sequential block symbol (open), 67
'(S|': indicates the start of a **sequential block** S

sequential block symbol (close), 67
'|)': indicates the end of a **sequential block**

sequential update scheme, 67
a **top-level sequential block**

stage, 67
one step of a **sequence**

stageless sequence, 67
a **sequence** with all of its **stages** left untagged

store, 10
a two-way countably infinite set of **cells**

store structure, 15
a regular expression over **store** names

synchroniser, 73
central update plan synchroniser (UP:SEQ), a constant used in the implementation of **sequences**

text, 17
a sequence of terms

textual expansion, 17
the replacement **text** of an **archetype** call, before **parameter resolution**

textual ordering, 69
the way **text** expanded in individual **stages** of a **sequence** is arranged

top-level, 69
an **update plan**'s top-level **update** is also an item in the **update plan**

type alias, 14
a type declaration

type primitive, 14
a type name not appearing as the left-hand side of a **type alias**

update, 58
a **parallel block**, a **sequential block** or **alternatives**

update plan, 10
a set of **updates**, type and **store** *declarations*

update rule, 10
an **update scheme** which contains no variables and both its left and right-hand sides are self-**consistent**

update scheme, 10
consists of a left-hand side, a right-hand side (both **configurations**) and a **guard**

update script, 11
an **update plan** and an **initial configuration**