The DeCCo project papers II

Z Specification of Asp

Susan Stepney and Ian T. Nabney

University of York Technical Report YCS-2002-359

June 2003

© Crown Copyright 2003

PERMITTED USES. This material may be accessed as downloaded onto electronic, magnetic, optical or similar storage media provided that such activities are for private research, study or in-house use only.

RESTRICTED USES. This material must not be copied, distributed, published or sold without the permission of the Controller of Her Britannic Majesty's Stationery Office.

Contents

1 Introduction **1**
1.1 Asp, AspAL, and XAspAL 1
1.2 Overview of the Asp device 1

2 Abstract syntax **3**
2.1 Instructions 4
2.2 Program 8

3 Concrete syntax **10**
3.1 Asp concrete syntax 10
3.2 AspAL concrete syntax 10

4 Asp and AspAL domains **12**
4.1 Store or Contents 12
4.2 State 12
4.3 Continuations 13
4.4 Environments 14
4.5 State updating functions 14
4.6 State retrieval functions 18

5 Dynamic semantics **20**
5.1 Meaning function 20
5.2 Load instructions 22
5.3 Logical instructions 25
5.4 Equality instructions 27
5.5 Rotation instructions 27
5.6 Carry bit instructions 28
5.7 Halt and Nop 29
5.8 Arithmetic operators 30
5.9 Comparison operators 33
5.10 Data address manipulations 34
5.11 Program address manipulations 36
5.12 Remaining XAspAL instructions 39
5.13 Program semantics 40

A General DeCCo Toolkit **43**

A.1 Raising to a power 43
A.2 Bits 44

B AspAL Toolkit definitions 46
B.1 Addresses and values 46
B.2 Value conversion 47

Preface

Historical background of the DeCCo project

In 1990 Logica's Formal Methods Team performed a study for RSRE (now QinetiQ) into how to develop a compiler for high integrity applications that is itself of high integrity. In that study, the source language was Spark, a subset of Ada designed for safety critical applications, and the target was Viper, a high integrity processor. Logica' Formal Methods Team developed a mathematical technique for specifying a compiler and proving it correct, and developed a small proof of concept prototype. The study is described in [Stepney *et al.* 1991], and the small case study is worked up in full, including all the proofs, in [Stepney 1993]. Experience of using the PVS tool to prove the small case study is reported in [Stringer-Calvert *et al.* 1997]. Futher developments to the method to allow separate compilation are described in [Stepney 1998].

Engineers at AWE read about the study and realised the technique could be used to implement a compiler for their own high integrity processor, called the ASP (Arming System Processor). They contacted Logica, and between 1992 and 2001 Logica used these techniques to deliver a high integrity compiler, integrated in a development and test environment, for progressively larger subsets of Pascal.

The full specifications of the final version of the DeCCo compiler are reproduced in these technical reports. These are written in the Z specification language. The variant of Z used is that supported by the *Z Specific Formaliser* tool [Formaliser], which was used to prepare and type-check all the DeCCo specifications. This variant is essentially the Z described in the *Z Reference Manual* [Spivey 1992] augmented with a few new constructs from *ISO Standard Z* [ISO-Z]. Additions to ZRM are noted as they occur in the text.

The DeCCo Reports

The DeCCo Project case study is detailed in the following technical reports (this preface is common to all the reports):

I. **Z Specification of Pasp**
The denotational semantics of the high level source language, Pasp. The definition is split into several static semantics (such as type checking) and a dynamic semantics (the meaningof executing a program). Later smeantics are not defined for those programs where the result of earlier semantics is error.

II. **Z Specification of Asp, AspAL and XAspAL**
The denotational semantics of the low level target assembly languages. XAspAL is the target of compilation of an individual Pasp module; it is AspAL extended with some cross-module instructions that are resolved at link time. The meaning of these extra instructions is given implicitly by the specification of the linker and hexer. AspAL is the target of linking a set of XAspAL modules, and also the target of compilation of a complete Pasp program. Asp is the non-relocatable assembly language of the chip, with AspAL's labels replaced by absolute program addresses. The semantics of programs with errors is not defined, because these defintions will only ever be used to define the meaning of correct, compiled programs.

III. **Z Specification of Compiler Templates**
The operational semantics of the Pasp source language, in the form of a set of XAspAL target language templates.

IV. **Z Specification of Linker and Hexer**
The linker combines compiled XAspAL modules into a single compiled AspAL program. The hexer converts a relocatable AspAL program into an Asp program located at a fixed place in memory.

V. **Compiler Correctness Proofs**
The compiler's operational semantics are demonstrated to be equivalent to the source language's denotational semantics, by calculating the meaning of each Pasp construct, and the corresponding meaning

of the AspAL template, and showing them to be equivalent. Thus the compiler transformation is *meaning preserving*, and hence the compiler is correct.

VI. **Z to Prolog DCTG translation guidelines**
The Z specifications of the Pasp semantics and compiler templates are translated into an executable Prolog DCTG implementation of a Pasp interpreter and Pasp-to-Asp compiler. The translation is done manually, following the stated guidelines.

Acknowledgements

We would like to thank the client team at AWE – Dave Thomas, Wilson Ifill, Alun Lewis, Tracy Bourne – for providing such an interesting development project to work on.

We would like to thank the rest of the development team at Logica: Tim Wentford, John Taylor, Roger Eatwell, Kwasi Ametewee.

1 Introduction

1.1 Asp, AspAL, and XAspAL

This document defines the abstract syntax and semantics of Asp and AspAL, and the abstract syntax of XAspAL (the semantics of the extral X commands are defined by the linker and hexer).

XAspAL is the target of compilation of an individual Pasp module. It is AspAL extended with some across-module instructions that are resolved at link time. AspAL is the target of linking a set of XAspAL modules. It is close in form to the Asp instruction set, but refers to the destination of jumps using labels rather than program addresses.

The subset of the full Asp processor instruction set that is modelled in AspAL is sufficient to be the target of a translation from Pasp, the source language.

Since Asp is the target language, it is used only in a well controlled manner. In particular, there is no need to define any static semantics, since if the source program is checked, and the translation is carried out correctly, then the target program is correct as well. Furthermore, we need to specify only a small number of the instructions available on the Asp processor since Pasp programs can be translated without using most of the instructions available.

1.2 Overview of the Asp device

The ASP is a RISC with separate data and program address spaces. Each address space is addressed with sixteen bits. To facilitate addressing there are two sixteen bit registers, for the current program location and D for the data address location. Each data location contains eight bits.

There are two eight bit data registers: the a register and the b register. Each of these has an associated carry bit, the a-carry and the b-carry respectively.

There is a sixteen bit data register: the C register. It may be used to store and load the contents of the a and b registers, as well as store and load the contents of the memory locations pointed to by the data address.

Errors, for example arithmetic or memory addressing overflow, halt the Asp. Parallel I/O is possible only via memory-mapping.

2 Abstract syntax

In this section we define the abstract syntaxes for the ASP Assembly Language, Asp, used by the hexer, and the eXtended Asp Assembly Language (XAspAL) used by the compiler and the linker. The syntax required for XAspAL is a combination of the syntax for the AspAL language (the target of linking a set of modules) along with additional syntax required to support linking separately compiled modules.

All the Asp instructions are specified. In choosing a subset of Asp instructions to model in AspAL, we have tried to balance the desire to specify as small a subset as possible with the desire for achieving a reasonably efficient compiler and linker (which leads to the definition of a larger number of instructions). The additional XAspAL instructions are provided purely to allow the linker to properly link together the pre-compiled modules.

There are two syntactic categories in Asp: the instruction and the program. There are no labels as such in Asp. Jumps and gotos are implemented by instructions the manipulate the program counter directly. Therefore each Asp instruction is labelled with the corresponding program address.

The code output from the compiler is relocatable; addresses are resolved when the individual modules are linked. Labels rather than actual program addresses are therefore used within the compiler output. While most uses of labels are 'fixed', there are some uses where the label can be determined only at run time. A jump or goto to a label is known at compile time and so is fixed (this is true for all jumps and gotos inside the templates for operators, if and while statements and procedure and function calls). The gotos providing the return from functions and procedures can be resolved only at run time.

The relocation of memory is treated as an offset from a given data address. There are three distinct areas of memory: the heap for variables, assign Addr and for storing return program addresses; the stack, for evaluating expressions; and the operator scratchpad. The way this works is discussed in the compiler specification.

Some of the instructions defined below are required because the result of tests (such as whether two values are equal) is stored in the carry bit of the

relevant data register. We use register rotation instructions to access these bits. The instructions for manipulating the carry bit are required to perform logical operations on (Pasp) boolean values.

Some definitions of types, constants, and common functions are included as an appendix.

2.1 Instructions

Asp instructions are divided into two classes: immediate and indirect. Immediate instructions are followed by *BYTE* data; indirect instructions access *BYTE* data contained in the memory location referenced by the data address register *D*, if any is required for the instruction.

2.1.1 Common Asp and XAspAL Instructions

The common immediate instructions are:

IMMED_INSTR ::=
  `aldi` | `bldi` | `aori` | `bori` | `aani` | `bani` | `axoi`
  | `bxoi` | `aeqi` | `beqi` | `ausi` | `busi` | `auai` | `buai` | `auci`
  | `buci` | `aumi` | `audi` | `dix` | `dxi` | `dpi`

Most immediate instructions have an indirect counterpart that has the same mnemonic except that final *i* is replaced by a *d*. The common indirect instructions are:

INDIRECT_INSTR ::=
  `aldb` | `blda` | `aldd` | `bldd` | `astd` | `bstd`
  | `daldab` | `cldab` | `abldc` | `abldd` | `abstd`
  | `cldda` | `daldc` | `cldd` | `cstd`
  | `aord` | `bord` | `aand` | `band` | `axod` | `bxod` | `aeqd` | `beqd`
  | `lra` | `lrb` | `rra` | `rrb` | `cca` | `ccb` | `nca` | `ncb` | `sca`
  | `scb` | `caldcb` | `cbldca` | `hlt` | `hltca` | `hltcb` | `nop` | `ausd`
  | `busd` | `auad` | `buad` | `aucd` | `bucd` | `aumd` | `audd`

2.1.2 Asp only Instructions

There are some Asp instructions that do not occur in XAspAL, because they are jumps to program locations, which XAspAL abstracts away to program labels.

The Asp-only immediate instructions are:

$$ASP_IMMED_INSTR ::= \texttt{ppi} \mid \texttt{cappi} \mid \texttt{cbppi}$$

The Asp-only indirect instructions are:

$$ASP_INDIRECT_INSTR ::= \texttt{pab} \mid \texttt{capab} \mid \texttt{cbpab}$$

2.1.3 XAspAL only, X Instructions

The language extensions for XAspAL are indicated by the first character being an x or X.

2.1.3.1 X Indirect Instruction

$$X_INDIRECT_INSTR ::= xag$$

- xag, equivalent of "`pab`": indirect absolute goto, go to label in the ab registers.

2.1.3.2 X Label Instructions

$$X_LABEL_INSTR ::= xrja \mid xrjb \mid xrjal \mid xrjbl \mid xrg \mid xrgs \mid xll \mid xli$$

- $xrja$, equivalent of "`cappi, pab`": jump to immediate label, conditional on a-carry.
- $xrjb$, equivalent of "`cbppi, pab`": jump to immediate label, conditional on b-carry.

  The following two instructions allow an unlimited 'long' jump to any program address, but the contents of a and b are overwritten.

- *xrjal*, equivalent of "`cappi`": jump to immediate label, conditional on *a*- carry.
- *xrjbl*, equivalent of "`cbppi`": jump to immediate label, conditional on *b*- carry.
- *xrgs*, equivalent of "`ppi`": goto immediate label. This instruction allows a relative jump of up to 255 instructions, and preserves the *a* and *b* registers.
- *xrg*, equivalent of "`pab`": goto immediate label. This instruction allows an unlimited goto to any program address, but the contents of *a* and *b* are overwritten.
- *xll*: load "immediate" label into *ab*.
- *xli*: mark the position of a label in the code.

2.1.3.3 X Data Instructions

$$X_DATA_INSTR ::= xsdrh \mid xsdrs \mid xsdro \mid xsdr \mid xlada \mid xlrda$$

- *xsdrh*, *xsdrs*, *xsdro*, *xsdr*: set data register relative to heap, stack and op starts, and absolute address.
- *xlada*, *xlrda*: load the contents of the data address into *ab*, using an absolute, and relative, address.

2.1.4 All Asp Instructions

An Asp instruction *INSTR* is either an immediate instruction with its byte of immediate data, or an indirect instruction.

$$\begin{aligned} INSTR ::= {} & both_immediate\langle\langle IMMED_INSTR \times BYTE\rangle\rangle \\ & \mid asp_immediate\langle\langle ASP_IMMED_INSTR \times BYTE\rangle\rangle \\ & \mid both_indirect\langle\langle INDIRECT_INSTR\rangle\rangle \\ & \mid asp_indirect\langle\langle ASP_INDIRECT_INSTR\rangle\rangle \end{aligned}$$

2.1.5 All XAspAL Instructions

To communicate between modules, we need identifiers to name functions within modules. We also define ID here, as it is required for the following declaration.

$$[ID]$$

The complete set of X_INSTR instructions is as follows. A number of these instructions are the extra instructions in the XAspAL language. These are resolved and replaced by the linker.

$$\begin{aligned} X_INSTR ::= {} & immediate\langle\langle IMMED_INSTR \times BYTE\rangle\rangle \\ & \mid indirect\langle\langle INDIRECT_INSTR\rangle\rangle \\ & \mid xIndirect\langle\langle X_INDIRECT_INSTR\rangle\rangle \\ & \mid xLabel\langle\langle X_LABEL_INSTR \times LABEL\rangle\rangle \\ & \mid xData\langle\langle X_DATA_INSTR \times DATA_ADDRESS\rangle\rangle \\ & \mid xBvar\langle\langle ID \times \operatorname{seq} \mathbb{N}\rangle\rangle \mid xConst\langle\langle ID\rangle\rangle \\ & \mid xCall\langle\langle ID\rangle\rangle \mid xCallOp\langle\langle LABEL\rangle\rangle \\ & \mid xProc\langle\langle ID \times LABEL\rangle\rangle \mid xPa\langle\langle ID \times ID\rangle\rangle \end{aligned}$$

- *immediate*: all the AspAL immediate instructions
- *indirect*: all the AspAL indirect instructions
- *xIndirect*: the XAspAL indirect instruction
- *xLabel*: the XAspAL label instructions
- *xData*: XAspAL data address modifying instructions
- *xBvar*: identifies B-style imported variable
- *xConst*: identifies imported constant
- *xCall*: function calling
- *xCallOp*: optimised operator calling

- *xProc*: identifies procedure and function
- *xPa*: parameter passing

The question of uniqueness of labels across modules is solved by the linker, which interprets all labels as an offset from an initial label and modifies the labels (and environment) accordingly.

2.2 Program

For a full definition of the Asp and AspAL languages there would need to be a much tighter definition of what constitutes a legal program, so that whenever the program register is set to a new value (either through direct manipulation or through the normal increment after an instruction) there is an instruction at that address. However, it is not possible in general to carry out all such checks statically, as the program register can be set to the contents of the a and b registers.

2.2.1 Asp Program

An Asp program consists of

- the address of the first instruction to be executed
- the instructions, each labelled with the address in the program data space where it is located

$$
\begin{array}{l}
ASP_PROGRAM == \\
\quad \{ \delta : PROG_ADDRESS;\ I : PROG_ADDRESS \nrightarrow INSTR \mid \\
\qquad\qquad \delta \in \operatorname{dom} I \land \operatorname{dom} I \in \operatorname{ran}(_ \ldots _) \\
\qquad\qquad \land\ I(max(\operatorname{dom} I)) = both_indirect\ \mathtt{hlt}\ \}
\end{array}
$$

The start address is one of the instructions; the instructions occupy a contiguous area of program data space; the last instruction is a halt.

2.2.2 AspAL Program

The abstract syntax of an AspAL program is straightforward. An AspAL program consists of a sequence of instructions. As indicated above these instructions can consist of a combination of AspAL instructions and various types of label.

It is convenient to introduce two syntactic categories at this stage; a module and a program. Although they are syntactically equivalent there are some semantic differences between the two constructs.

An $ASPAL_MODULE$ is the XAspAL produced by the compilation of a Pasp module.

$$ASPAL_MODULE == \mathrm{seq}\, X_INSTR$$

An $ASPAL_PROGRAM$ is reserved for the AspAL resulting from the linking together of a set of XAspAL modules into a single program corresponding to a Pasp program.

$$ASPAL_PROGRAM == \mathrm{seq}\, X_INSTR$$

3 Concrete syntax

The assembly language mnemonic for an instruction is essentially the same as the abstract syntactic name for that instruction. For immediate instructions, the language mnemonic is followed by the immediate data in the form of two hexadecimal digits. The function $\mathcal{C}_{HEX}$ carries out the required mapping from bytes to two-digit numbers.

$$\mathcal{C}_{HEX} : BYTE \rightarrow \mathrm{seq}\, ALPHANUM$$
$$\forall b : BYTE \bullet \#(\mathcal{C}_{HEX}\ b) = 2$$

3.1 Asp concrete syntax

Let us suppose that the functions $\mathcal{C}_{Abin}$ and $\mathcal{C}_{Abim}$ and so on map abstract syntax instruction names and labels to their concrete string equivalents. Then the function $\mathcal{C}_{AspAI}$ that maps abstract syntax to concrete may be defined as follows.

$$\mathcal{C}_{AspAI} : INSTR \rightarrow String$$
$$\begin{array}{l}
\forall i : IMMED_INSTR;\ b : BYTE \bullet \\
\quad \mathcal{C}_{AspAI}(both_immediate(i, b)) = \mathcal{C}_{Abim}\ i \frown \mathcal{C}_{HEX}\ b \\
\forall i : ASP_IMMED_INSTR;\ b : BYTE \bullet \\
\quad \mathcal{C}_{AspAI}(asp_immediate(i, b)) = \mathcal{C}_{Aaim}\ i \frown \mathcal{C}_{HEX}\ b \\
\forall i : INDIRECT_INSTR \bullet \mathcal{C}_{AspAI}(both_indirect\ i) = \mathcal{C}_{Abin}\ i \\
\forall i : ASP_INDIRECT_INSTR \bullet \mathcal{C}_{AspAI}(asp_indirect\ i) = \mathcal{C}_{Aain}\ i
\end{array}$$

3.2 AspAL concrete syntax

Let us suppose that the functions $\mathcal{C}_{in}$ and $\mathcal{C}_{im}$ and so on map abstract syntax instruction names and labels to their concrete string equivalents. Then the function $\mathcal{C}_{AI}$ that maps abstract syntax to concrete may be defined as follows.

$$\mathcal{C}_{AI} : X_INSTR \rightarrow String$$

$$\forall i : IMMED_INSTR;\ b : BYTE \bullet$$
$$\quad \mathcal{C}_{AI}(immediate(i, b)) = \mathcal{C}_{im}\ i \frown \mathcal{C}_{HEX}\ b$$
$$\forall i : INDIRECT_INSTR \bullet \mathcal{C}_{AI}(indirect\ i) = \mathcal{C}_{in}\ i$$
$$\forall i : X_INDIRECT_INSTR \bullet \mathcal{C}_{AI}(xIndirecti) = \mathcal{C}_{Xin}\ i$$
$$\forall i : X_LABEL_INSTR;\ l : LABEL \bullet$$
$$\quad \mathcal{C}_{AI}(xLabel(i, l)) = \mathcal{C}_{Xl}\ i \frown \mathcal{C}_{l}\ l$$
$$\forall i : X_DATA_INSTR;\ \delta : DATA_ADDRESS \bullet$$
$$\quad \mathcal{C}_{AI}(xData(i, \delta)) = \mathcal{C}_{Xd}\ i \frown \mathcal{C}_{\delta}\ \delta$$
$$\forall \xi : IDENTIFIER \bullet \mathcal{C}_{AI}(xCall\ \xi) = \mathcal{C}_{\xi}\ \xi$$
$$\forall l : LABEL \bullet \mathcal{C}_{AI}(xCallOp\ l) = \mathcal{C}_{l}\ l$$
$$\forall \xi : IDENTIFIER;\ l : LABEL \bullet \mathcal{C}_{AI}(xProc(\xi, l)) = \mathcal{C}_{\xi}\ \xi \frown \mathcal{C}_{l}\ l$$
$$\forall \xi, \xi' : IDENTIFIER \bullet \mathcal{C}_{AI}(xPa(\xi, \xi')) = \mathcal{C}_{\xi}\ \xi \frown \mathcal{C}_{\xi}\ \xi'$$

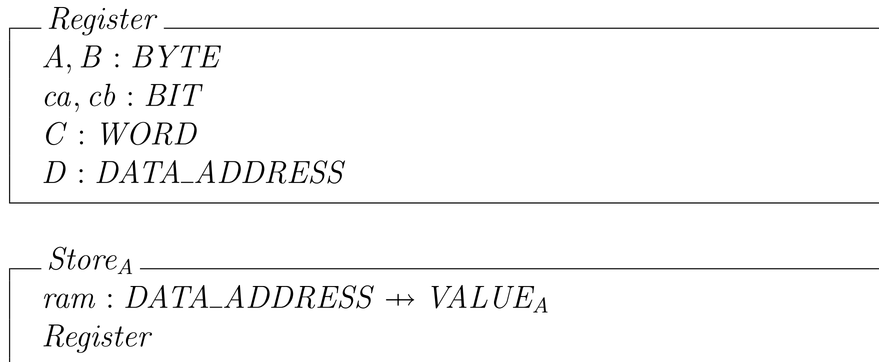
4 Asp and AspAL domains

This section describes the domains required for an Asp or an AspAL program. We have already defined the address domains in the abstract syntax.

4.1 Store or Contents

Because Asp addresses are not bijective with Pasp locations, it improves the clarity of the operational semantics and proofs to call the store by another name, $Store_A$, as there is a store defined as a domain in the translation context.

The content of every *ram* address and the *a* and *b* data registers is a *BYTE*. The *ca* and *cb* are carry *BIT*s. The *C* register holds a *WORD*. The data address register is sixteen bits, and the values it contains are modelled as *WORD*s. Because of the complexity of the Asp storage compared to that of Pasp, we group its components into a schema.



Register

$A, B : BYTE$
$ca, cb : BIT$
$C : WORD$
$D : DATA_ADDRESS$

$Store_A$

$ram : DATA_ADDRESS \pfun VALUE_A$
$Register$

These schemas may be regarded as analogous to a structure in C, or a record in Pascal.

4.2 State

The input and output from a program are modelled as *Bstreams* attached to data addresses. The locations of these streams are given by sets *Iadd* and

Oadd:

$$Iadd == \mathbb{P}\, DATA_ADDRESS$$
$$Oadd == \mathbb{P}\, DATA_ADDRESS$$

The Asp state is analogous to the dynamic state of Pasp, in that it consists of the store, the input addresses, and the output addresses.

$$\begin{array}{l} State_A == \\ \quad \{\ Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd \mid \\ \qquad\qquad \mathsf{disjoint}\langle \Delta i, \Delta o \rangle \\ \qquad\qquad \land\ (\ \forall \delta : \operatorname{dom} ram \bullet \\ \qquad\qquad\qquad ram\ \delta \in \operatorname{ran} Bstream \Leftrightarrow \delta \in \Delta i \cup \Delta o\ ) \bullet \\ \qquad\quad (\theta\, Store_A, \Delta i, \Delta o)\ \} \end{array}$$

No address can be both input and output; a ram address is a stream precisely when it is an input or output.

The query functions $storeOf_A$ and $outOf_A$ return the $Store_A$ component and output streams of the state respectively.

$$storeOf_A, outOf_A : State_A \rightarrow Store_A$$

$$\begin{array}{l} \forall\, Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd \bullet \\ \qquad storeOf_A(\theta\, Store_A, \Delta i, \Delta o) = \theta\, Store_A \\ \qquad \land\ outOf_A(\theta\, Store_A, \Delta i, \Delta o) \\ \qquad\qquad = (\mu\, Store_A{}' \mid ram' = \Delta o \lhd ram \land \theta Register' = \theta Register) \end{array}$$

4.3 Continuations

Because of jumps and gotos, Asp instructions do not always compose sequentially; the dynamic semantics of Asp are not as straightforward as those of Pasp. For example, the meaning of the sequence $\langle jump, store \rangle$ is not the meaning of *jump* followed by the meaning of *store*; the jump command bypasses the following store command. The standard way to solve this problem is to use *continuations*.

A continuation is the computation that follows an instruction if control is not passed elsewhere. It is the state transition of the remainder of the program from that point on.

$$Cont == State_A \pfun State_A$$

The function is partial since continuations is defined for legal Asp and AspAL programs and states.

4.4 Environments

4.4.1 Asp's environment

The Asp environment is a map from program instruction addresses to what they denote, which is the computation that follows from jumping to that address.

$$Env_A == PROG_ADDRESS \pfun Cont$$

4.4.2 AspAL's environment

The AspAL environment is a map from 'labels' (abstract representation of some program instruction addresses) to what they denote, which is the computation that follows from jumping to that label.

$$Env_X == LABEL \pfun Cont$$

4.5 State updating functions

It is convenient to define a number of functions to update the values of components of the store. Although Z has a notation for accessing schema components, there is no notation for explicitly altering their values or defining their bindings. This makes the function definitions somewhat cumbersome.

The functions are defined as maps between states as this is most convenient for the definition of the dynamic semantics. The functions take natural numbers as arguments and truncate them as appropriate.

4.5.1 Ram updating

The function *updateRam* updates the *ram* location referenced by the *addr* with the byte b. If the address is the location of an output stream, then the value is added to the end of the stream.

$$
\begin{array}{l}
updateRam : State_A \times DATA_ADDRESS \times BYTE \rightarrow State_A \\
\hline
\forall Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ \delta : DATA_ADDRESS;\ b : BYTE \bullet \\
\quad \exists Store_A{}';\ v : VALUE_A \mid \\
\qquad \theta Register' = \theta Register \\
\qquad v = \mathbf{if}\ \delta \in \Delta o \\
\qquad\quad \mathbf{then}\ Bstream(Bstream^{\sim}\ (ram\ \delta) \frown \langle b \rangle) \\
\qquad\quad \mathbf{else}\ byte\ b \\
\qquad \wedge\ ram' = ram \oplus \{\delta \mapsto v\} \bullet \\
\quad updateRam\ ((\theta Store_A, \Delta i, \Delta o), \delta, b) = (\theta Store_A{}', \Delta i, \Delta o)
\end{array}
$$

The function $\stackrel{R}{\leftarrow}$ updates the *ram* location referenced by the data address register D.

$$\mathbf{function}\ 30\ \mathbf{leftassoc}(_ \stackrel{R}{\leftarrow} _)$$

$$
\begin{array}{l}
_ \stackrel{R}{\leftarrow} _ : State_A \times \mathbb{N} \rightarrow State_A \\
\hline
\forall \sigma : State_A;\ n : \mathbb{N} \bullet \\
\quad \exists Store_A \mid \theta Store_A = storeOf_A\ \sigma \bullet \\
\qquad \sigma \stackrel{R}{\leftarrow} n = updateRam(\sigma, D, n \bmod 256)
\end{array}
$$

The function $\stackrel{W}{\leftarrow}$ updates the *ram* locations referenced by the data address register and the next location (that is, $D+1$). If either of these locations are output streams, then the relevant value is added to the end of their stream.

function 30 **leftassoc**$(_ \overset{W}{\leftarrow} _)$

$$
\begin{array}{l}
_ \overset{W}{\leftarrow} _ : State_A \times (\mathbb{N} \times \mathbb{N}) \rightarrow State_A \\
\hline
\forall \sigma : State_A;\ lo, hi : \mathbb{N} \bullet \\
\quad \exists Store_A \mid \theta Store_A = storeOf_A\ \sigma \bullet \\
\quad\quad \sigma \overset{W}{\leftarrow} (hi, lo) = \\
\quad\quad\quad updateRam(updateRam(\sigma, D, lo \bmod 256), \\
\quad\quad\quad\quad D + 1, hi \bmod 256)
\end{array}
$$

4.5.2 Register updating

These functions update the relevant part of the store with the data passed. All the functions are used in the definitions of the dynamic semantics of the Asp instructions.

The functions $\overset{A}{\leftarrow}$, $\overset{B}{\leftarrow}$ and $\overset{AB}{\leftarrow}$ update the values of the a register, the b register, and both registers, respectively.

function 30 **leftassoc**$(_ \overset{A}{\leftarrow} _)$

$$
\begin{array}{l}
_ \overset{A}{\leftarrow} _ : State_A \times \mathbb{N} \rightarrow State_A \\
\hline
\forall Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ n : \mathbb{N} \bullet \\
\quad \exists Store_A{}' \mid ram' = ram \wedge A' = n \bmod 256 \wedge B' = B \\
\quad\quad\quad \wedge\ ca' = ca \wedge cb' = cb \wedge C' = C \wedge D' = D \bullet \\
\quad\quad (\theta Store_A, \Delta i, \Delta o) \overset{A}{\leftarrow} n = (\theta Store_A{}', \Delta i, \Delta o)
\end{array}
$$

function 30 **leftassoc**$(_ \overset{B}{\leftarrow} _)$

$$
\begin{array}{l}
_ \overset{B}{\leftarrow} _ : State_A \times \mathbb{N} \rightarrow State_A \\
\hline
\forall Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ n : \mathbb{N} \bullet \\
\quad \exists Store_A{}' \mid ram' = ram \wedge A' = A \wedge B' = n \bmod 256 \\
\quad\quad\quad \wedge\ ca' = ca \wedge cb' = cb \wedge C' = C \wedge D' = D \bullet \\
\quad\quad (\theta Store_A, \Delta i, \Delta o) \overset{B}{\leftarrow} n = (\theta Store_A{}', \Delta i, \Delta o)
\end{array}
$$

function 30 **leftassoc**$(_ \stackrel{AB}{\leftarrow} _)$

$$_ \stackrel{AB}{\leftarrow} _ : State_A \times WORD \rightarrow State_A$$

$$\forall \sigma : State_A;\ w : WORD \bullet$$
$$\quad \sigma \stackrel{AB}{\leftarrow} w = \sigma \stackrel{A}{\leftarrow} theHiByte\ w \stackrel{B}{\leftarrow} theLoBytew$$

The functions $\stackrel{Ac}{\leftarrow}$ and $\stackrel{Bc}{\leftarrow}$ update the values of the a-carry and the b-carry bits respectively.

function 30 **leftassoc**$(_ \stackrel{Ac}{\leftarrow} _)$

$$_ \stackrel{Ac}{\leftarrow} _ : State_A \times \mathbb{N} \rightarrow State_A$$

$$\forall Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ n : \mathbb{N} \bullet$$
$$\quad \exists Store_A{}' \mid ram' = ram \land A' = A \land B' = B$$
$$\qquad \land ca' = n \bmod 2 \land cb' = cb \land C' = C \land D' = D \bullet$$
$$\qquad (\theta\, Store_A, \Delta i, \Delta o) \stackrel{Ac}{\leftarrow} n = (\theta\, Store_A{}', \Delta i, \Delta o)$$

function 30 **leftassoc**$(_ \stackrel{Bc}{\leftarrow} _)$

$$_ \stackrel{Bc}{\leftarrow} _ : State_A \times \mathbb{N} \rightarrow State_A$$

$$\forall Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ n : \mathbb{N} \bullet$$
$$\quad \exists Store_A{}' \mid ram' = ram \land A' = A \land B' = B$$
$$\qquad \land ca' = ca \land cb' = n \bmod 2 \land C' = C \land D' = D \bullet$$
$$\qquad (\theta\, Store_A, \Delta i, \Delta o) \stackrel{Bc}{\leftarrow} n = (\theta\, Store_A{}', \Delta i, \Delta o)$$

The function $\stackrel{C}{\leftarrow}$ updates the contents of the C register.

function 30 **leftassoc**$(_ \stackrel{C}{\leftarrow} _)$

$$_ \overset{C}{\leftarrow} _ : State_A \times \mathbb{N} \rightarrow State_A$$

$$\begin{array}{l}
\forall\, Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ n : \mathbb{N} \bullet \\
\quad \exists\, Store_A{}' \mid ram' = ram \land A' = A \land B' = B \\
\qquad \land\ ca' = ca \land cb' = cb \land C' = n \bmod 2 \uparrow 16 \land D' = D \bullet \\
\qquad (\theta\, Store_A, \Delta i, \Delta o) \overset{C}{\leftarrow} n = (\theta\, Store_A{}', \Delta i, \Delta o)
\end{array}$$

The function $\overset{D}{\leftarrow}$ updates the value of the data address register.

$$\textbf{function}\, 30\, \textbf{leftassoc}(_ \overset{D}{\leftarrow} _)$$

$$_ \overset{D}{\leftarrow} _ : State_A \times \mathbb{N} \rightarrow State_A$$

$$\begin{array}{l}
\forall\, Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ n : \mathbb{N} \bullet \\
\quad \exists\, Store_A{}' \mid ram' = ram \land A' = A \land B' = B \\
\qquad \land\ ca' = ca \land cb' = cb \land C' = C \land D' = n \bmod 2 \uparrow 16 \bullet \\
\qquad (\theta\, Store_A, \Delta i, \Delta o) \overset{D}{\leftarrow} n = (\theta\, Store_A{}', \Delta i, \Delta o)
\end{array}$$

The value of the data address register is updated with the number passed. This is set to be word-valued so that it has the correct type for the register.

4.6 State retrieval functions

The *ramOf* function retrieves the contents of the memory location at *addr*. Just as in the case of the dynamic semantics of value references in Pasp, if the location is connected to an input stream, the state may change.

$$
\begin{array}{l}
ramOf : State_A \times DATA_ADDRESS \rightarrowtail State_A \times BYTE \\
\hline
\forall Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd;\ \delta : DATA_ADDRESS \mid \delta \in \Delta i \bullet \\
\quad \exists Store_A{}';\ n : \mathbb{N};\ bb : \mathrm{seq}\, BYTE \mid \\
\qquad\qquad Bstream\ bb = ram\ \delta \land n < \# bb \\
\qquad\qquad \land\ ram' = ram \oplus \{\delta \mapsto Bstream\ (tail^n\ bb)\} \\
\qquad\qquad \land\ \theta Register' = \theta Register \bullet \\
\qquad ramOf((\theta\, Store_A, \Delta i, \Delta o), \delta) = \\
\qquad\qquad ((\theta\, Store_A{}', \Delta i, \Delta o), bb\ (n+1)) \\
\forall \sigma : State_A;\ \delta : DATA_ADDRESS \mid \delta \notin \sigma.2 \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad ramOf(\sigma, \delta) = (\sigma, byte^{\sim}\ (ram\ \delta))
\end{array}
$$

The *byteRamOf* function retrieves the byte in the memory location referenced by the data address register.

$$
\begin{array}{l}
byteRamOf : State_A \rightarrowtail State_A \times BYTE \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad byteRamOf\ \sigma = ramOf(\sigma, D)
\end{array}
$$

The *wordRamOf* function retrieves 2 bytes of data (a word) from the contents of the memory location referenced by the data address register D and the following location $D+1$.

$$
\begin{array}{l}
wordRamOf : State_A \rightarrowtail State_A \times (BYTE \times BYTE) \\
\hline
\forall Store_A;\ \Delta i : Iadd;\ \Delta o : Oadd \bullet \\
\quad \exists Store_A{}';\ Store_A{}'';\ lo, hi : BYTE \mid \\
\qquad\qquad ((\theta\, Store_A{}', \Delta i, \Delta o), lo) = \\
\qquad\qquad\quad ramOf((\theta\, Store_A, \Delta i, \Delta o), D) \\
\qquad\qquad \land\ ((\theta\, Store_A{}'', \Delta i, \Delta o), hi) = \\
\qquad\qquad\quad ramOf((\theta\, Store_A{}', \Delta i, \Delta o), D+1) \bullet \\
\qquad wordRamOf(\theta\, Store_A, \Delta i, \Delta o) = \\
\qquad\qquad ((\theta\, Store_A{}'', \Delta i, \Delta o), (hi, lo))
\end{array}
$$

The values are returned as (high byte, low byte) that is, $(D+1, D)$.

5 Dynamic semantics

5.1 Meaning function

The meaning function for instructions is declared here, and defined in the following sections. An instruction causes a state change. So the instruction meaning function maps an instruction to the relevant state transition function, in the context of an environment and continuation.

The meaning of an instruction is extended to sequences in the usual inductive fashion.

5.1.1 Immediate instruction meaning function

All the common immediate instructions simply change the state.

$$\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \twoheadrightarrow State_A \twoheadrightarrow State_A$$

5.1.2 Indirect instruction meaning function

All the common inderect instructions, except halt, merely change the state.

$$\mathcal{I}_{ind} : INDIRECT_INSTR \twoheadrightarrow State_A \twoheadrightarrow State_A$$

5.1.3 Asp Meaning function

The common immediate and indirect instructions can be defined in terms of $\mathcal{I}_{imm}$ and $\mathcal{I}_{ind}$; the remaining halt, goto and jump instructions are defined later.

$$\mathcal{A}_I : PROG_ADDRESS \times INSTR \rightarrow Env_A \rightarrow Cont \rightarrow State_A \rightarrow State_A$$

$$\forall \delta : PROG_ADDRESS;\ i : IMMED_INSTR;\ b : BYTE;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet$$
$$\mathcal{A}_I(\delta, both_immediate(i, b))\rho\ \vartheta\ \sigma = \vartheta(\mathcal{I}_{imm}(i, b)\sigma)$$
$$\forall \delta : PROG_ADDRESS;\ i : INDIRECT_INSTR;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \mid i \notin \{\mathtt{hlt}, \mathtt{hltca}, \mathtt{hltcb}\} \bullet$$
$$\mathcal{A}_I(\delta, both_indirect\ i)\rho\ \vartheta\ \sigma = \vartheta(\mathcal{I}_{ind}\ i\ \sigma)$$

$$\mathcal{A}_{I^*} : \mathrm{seq}(PROG_ADDRESS \times INSTR) \rightarrow Env_A \rightarrow Cont \rightarrow State_A \rightarrow State_A$$

$$\forall i : PROG_ADDRESS \times INSTR;\ I, I' : \mathrm{seq}(PROG_ADDRESS \times INSTR);\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet$$
$$\mathcal{A}_{I^*}\langle\ \rangle\rho\ \vartheta\ \sigma = \sigma$$
$$\land\ \mathcal{A}_{I^*}\langle i\rangle\rho\ \vartheta\ \sigma = \mathcal{A}_I\ i\ \rho\ \vartheta\ \sigma$$
$$\land\ \mathcal{A}_{I^*}(I \frown I')\rho\ \vartheta\ \sigma = \mathcal{A}_{I^*}\ I'\ \rho\ \vartheta(\mathcal{A}_{I^*}\ I\ \rho\ \vartheta\ \sigma)$$

5.1.4 AspAL Meaning function

The common immediate and indirect instructions can be defined in terms of $\mathcal{I}_{imm}$ and $\mathcal{I}_{ind}$; the remaining halt and X instructions are defined later.

$$\mathcal{X}_I : X_INSTR \rightarrow Env_X \rightarrow Cont \rightarrow State_A \rightarrow State_A$$

$$\forall i : IMMED_INSTR;\ b : BYTE;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet$$
$$\mathcal{X}_I(immediate(i, b))\rho\ \vartheta\ \sigma = \vartheta(\mathcal{I}_{imm}(i, b)\sigma)$$
$$\forall i : INDIRECT_INSTR;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \mid i \notin \{\mathtt{hlt}, \mathtt{hltca}, \mathtt{hltcb}\} \bullet$$
$$\mathcal{X}_I(indirect\ i)\rho\ \vartheta\ \sigma = \vartheta(\mathcal{I}_{ind}\ i\ \sigma)$$

$$
\begin{array}{|l}
\mathcal{X}_{I^*} : \operatorname{seq} X_INSTR \rightarrow Env_X \rightarrow Cont \rightarrow State_A \rightarrow State_A \\
\hline
\forall\, i : X_INSTR;\ I, I' : \operatorname{seq} X_INSTR;\ \rho : Env_X;\ \vartheta : Cont; \\
\qquad\qquad \sigma : State_A \bullet \\
\qquad \mathcal{X}_{I^*}\langle\ \rangle \rho\ \vartheta\ \sigma = \sigma \\
\qquad \wedge\ \mathcal{X}_{I^*}\langle i \rangle \rho\ \vartheta\ \sigma = \mathcal{X}_I\ i\ \rho\ \vartheta\ \sigma \\
\qquad \wedge\ \mathcal{X}_{I^*}(I \frown I')\rho\ \vartheta\ \sigma = \mathcal{X}_{I^*}\ I'\ \rho\ \vartheta(\mathcal{X}_{I^*}\ I\ \rho\ \vartheta\ \sigma)
\end{array}
$$

5.2 Load instructions

This section defines instructions that allow values to be loaded from memory locations into specified registers. These fall into two categories, those which load values into single byte and two byte registers.

5.2.1 Load register with immediate

Load the a or b register with an byte of immediate data.

$$
\begin{array}{|l}
\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \rightarrow State_A \rightarrow State_A \\
\hline
\forall\, b : BYTE;\ \sigma : State_A \bullet \\
\qquad \mathcal{I}_{imm}(\mathtt{aldi}, b)\sigma = \sigma \stackrel{A}{\leftarrow} b \\
\qquad \wedge\ \mathcal{I}_{imm}(\mathtt{bldi}, b)\sigma = \sigma \stackrel{B}{\leftarrow} b
\end{array}
$$

5.2.2 Load register from register

Load the a register from the b register; load the b register from the a register.

$$
\begin{array}{|l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrow State_A \rightarrow State_A \\
\hline
\forall\, \sigma : State_A \bullet \\
\qquad \exists\, Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad\qquad \mathcal{I}_{ind}\ \mathtt{aldb}\ \sigma = \sigma \stackrel{A}{\leftarrow} B \\
\qquad\qquad \wedge\ \mathcal{I}_{ind}\ \mathtt{blda}\ \sigma = \sigma \stackrel{B}{\leftarrow} A
\end{array}
$$

5.2.3 Load register from memory

Two load functions are defined: one for each of the a and b registers. These instructions load the byte at the memory location given by the data address D into the relevant register.

$$\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrow\!\!\!\!\!\!+ State_A \rightarrow\!\!\!\!\!\!+ State_A$$

$$\forall \sigma : State_A \bullet$$
$$\quad \exists \sigma' : State_A;\ b : BYTE \mid (\sigma', b) = byteRamOf\ \sigma \bullet$$
$$\quad\quad \mathcal{I}_{ind}\ \texttt{aldd}\ \sigma = \sigma' \stackrel{A}{\leftarrow} b$$
$$\quad\quad \wedge\ \mathcal{I}_{ind}\ \texttt{bldd}\ \sigma = \sigma' \stackrel{B}{\leftarrow} b$$

5.2.4 Load memory from register

These instructions store the byte in the a or b register into the memory location pointed to by the data address register D.

$$\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrow\!\!\!\!\!\!+ State_A \rightarrow\!\!\!\!\!\!+ State_A$$

$$\forall \sigma : State_A \bullet$$
$$\quad \exists Store_A \mid \theta\ Store_A = storeOf_A\ \sigma \bullet$$
$$\quad\quad \mathcal{I}_{ind}\ \texttt{astd}\ \sigma = \sigma \stackrel{R}{\leftarrow} A$$
$$\quad\quad \wedge\ \mathcal{I}_{ind}\ \texttt{bstd}\ \sigma = \sigma \stackrel{R}{\leftarrow} B$$

5.2.5 C register storing and loading

There are two instructions which have the effect of loading or storing the value in the C register from or to the a and b registers. The C register is a word in size (16 bits) and therefore the operations use the concatenation of the a and b registers for updating.

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrowtail State_A \rightarrowtail State_A \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta Store_A = storeOf_A \; \sigma \bullet \\
\quad\quad \mathcal{I}_{ind} \; \texttt{cldab} \; \sigma = \sigma \stackrel{C}{\leftarrow} (A \dagger B) \\
\quad\quad \land \mathcal{I}_{ind} \; \texttt{abldc} \; \sigma = \sigma \stackrel{AB}{\leftarrow} C
\end{array}
$$

5.2.6 Load word into registers from memory

These functions are similar to the single byte versions however they load a word of data (sixteen bits). The values loaded into the registers are those at the memory locations given by the data address, and the following location (that is, $D+1$). The functions are defined for the concatenation of the a and b registers, and for the C register. The a,b function loads, in one operation, the value at the memory location given by the data address D (low byte) into the b register, and the following location $D+1$ (high byte) into the a register.

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrowtail State_A \rightarrowtail State_A \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists \sigma' : State_A ; \; lo, hi : BYTE \mid (\sigma', (lo, hi)) = wordRamOf \; \sigma \bullet \\
\quad\quad \mathcal{I}_{ind} \; \texttt{ablдd} \; \sigma = \sigma' \stackrel{AB}{\leftarrow} (hi \dagger lo) \\
\quad\quad \land \mathcal{I}_{ind} \; \texttt{cldd} \; \sigma = \sigma' \stackrel{C}{\leftarrow} (hi \dagger lo)
\end{array}
$$

5.2.7 Load word into memory from registers

These instructions store the word value (16 bits) contained within either the C register or the concatenation of the a and b registers. The low byte (b register or low byte of the C register) is stored at the memory location pointed to by the data address register D. The high byte is stored at the next location, at $D+1$.

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrowtail State_A \rightarrowtail State_A \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad \mathcal{I}_{ind}\ \texttt{abstd}\ \sigma = \sigma \overset{W}{\leftarrow} (A, B) \\
\qquad \wedge\ \mathcal{I}_{ind}\ \texttt{cstd}\ \sigma = \sigma \overset{W}{\leftarrow} (theHiByte\ C, theLoByte\ C)
\end{array}
$$

5.3 Logical instructions

Recall that the function *NatToBits* maps natural numbers to sequences of bits. These sequences are of variable length with no leading zeroes. This enables us to define logical operators on bit sequences generically, and then to restrict the domain so as to model the mapping on words, bytes, etc. (See the appendix for the definition of the operators *OR*, *XOR*, and *AND*.)

5.3.1 Logical OR

First we define the *OR* operators on the data registers.

$$
\begin{array}{l}
\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \rightarrowtail State_A \rightarrowtail State_A \\
\hline
\forall b : BYTE;\ \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad \mathcal{I}_{imm}\ (\texttt{aori}, b)\sigma = \\
\qquad\quad \sigma \overset{A}{\leftarrow} BitsToNat(NatToBits\ A\ OR\ NatToBits\ b) \\
\qquad \wedge\ \mathcal{I}_{imm}(\texttt{bori}, b)\sigma = \\
\qquad\quad \sigma \overset{B}{\leftarrow} BitsToNat(NatToBits\ B\ OR\ NatToBits\ b)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrowtail State_A \rightarrowtail State_A \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists \sigma' : State_A;\ b : BYTE \mid (\sigma', b) = byteRamOf\ \sigma \bullet \\
\qquad \mathcal{I}_{ind}\ \texttt{aord}\ \sigma = \mathcal{I}_{imm}(\texttt{aori}, b)\sigma' \\
\qquad \wedge\ \mathcal{I}_{ind}\ \texttt{bord}\ \sigma = \mathcal{I}_{imm}(\texttt{bori}, b)\sigma'
\end{array}
$$

The instructions take the values, converts them into a bit representation, applies the bit operations, and then converts the bit value back into the natural number representation.

5.3.2 Logical AND

The same principle is used to define the *AND* operators.

$$
\begin{array}{l}
\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \nrightarrow State_A \nrightarrow State_A \\
\hline
\forall b : BYTE;\ \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta Store_A = storeOf_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{imm}\ (\mathtt{aani}, b)\sigma = \\
\quad\quad\quad \sigma \stackrel{A}{\leftarrow} BitsToNat(NatToBits\ A\ AND\ NatToBits\ b) \\
\quad\quad \wedge\ \mathcal{I}_{imm}(\mathtt{bani}, b)\sigma = \\
\quad\quad\quad \sigma \stackrel{B}{\leftarrow} BitsToNat(NatToBits\ B\ AND\ NatToBits\ b)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \nrightarrow State_A \nrightarrow State_A \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists \sigma' : State_A;\ b : BYTE \mid (\sigma', b) = byteRamOf\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{aand}\ \sigma = \mathcal{I}_{imm}(\mathtt{aani}, b)\sigma' \\
\quad\quad \wedge\ \mathcal{I}_{ind}\ \mathtt{band}\ \sigma = \mathcal{I}_{imm}(\mathtt{bani}, b)\sigma'
\end{array}
$$

5.3.3 Logical XOR

The same principle is used to define the *XOR* operators.

$$
\begin{array}{l}
\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \nrightarrow State_A \nrightarrow State_A \\
\hline
\forall b : BYTE;\ \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta Store_A = storeOf_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{imm}(\mathtt{axoi}, b)\sigma = \\
\quad\quad\quad \sigma \stackrel{A}{\leftarrow} BitsToNat(NatToBits\ A\ XOR\ NatToBits\ b) \\
\quad\quad \wedge\ \mathcal{I}_{imm}(\mathtt{bxoi}, b)\sigma = \\
\quad\quad\quad \sigma \stackrel{B}{\leftarrow} BitsToNat(NatToBits\ B\ XOR\ NatToBits\ b)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}_{ind} : \mathit{INDIRECT_INSTR} \rightarrowtail \mathit{State}_A \rightarrowtail \mathit{State}_A \\
\hline
\forall \sigma : \mathit{State}_A \bullet \\
\quad \exists \sigma' : \mathit{State}_A;\ b : \mathit{BYTE} \mid (\sigma', b) = \mathit{byteRamOf}\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \texttt{axod}\ \sigma = \mathcal{I}_{imm}(\texttt{axoi}, b)\sigma' \\
\quad\quad \wedge \mathcal{I}_{ind}\ \texttt{bxod}\ \sigma = \mathcal{I}_{imm}(\texttt{bxoi}, b)\sigma'
\end{array}
$$

5.4 Equality instructions

These compare the contents of the specified register and the provided data, and test for equality of their values. The carry bit for the register is set to the result of the comparison.

$$
\begin{array}{l}
\mathcal{I}_{imm} : \mathit{IMMED_INSTR} \times \mathit{BYTE} \rightarrowtail \mathit{State}_A \rightarrowtail \mathit{State}_A \\
\hline
\forall b : \mathit{BYTE};\ \sigma : \mathit{State}_A \bullet \\
\quad \exists \mathit{Store}_A \mid \theta\, \mathit{Store}_A = \mathit{storeOf}_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{imm}(\texttt{aeqi}, b)\sigma = \\
\quad\quad\quad \sigma \overset{Ac}{\leftarrow} (\textbf{if}\ A = b\ \textbf{then}\ \mathit{btrue}\ \textbf{else}\ \mathit{bfalse}) \\
\quad\quad \wedge \mathcal{I}_{imm}(\texttt{beqi}, b)\sigma = \\
\quad\quad\quad \sigma \overset{Bc}{\leftarrow} (\textbf{if}\ B = b\ \textbf{then}\ \mathit{btrue}\ \textbf{else}\ \mathit{bfalse})
\end{array}
$$

We define the indirect operators in terms of their immediate counterparts.

$$
\begin{array}{l}
\mathcal{I}_{ind} : \mathit{INDIRECT_INSTR} \rightarrowtail \mathit{State}_A \rightarrowtail \mathit{State}_A \\
\hline
\forall \sigma : \mathit{State}_A \bullet \\
\quad \exists \sigma' : \mathit{State}_A;\ b : \mathit{BYTE} \mid (\sigma', b) = \mathit{byteRamOf}\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \texttt{aeqd}\ \sigma = \mathcal{I}_{imm}(\texttt{aeqi}, b)\sigma' \\
\quad\quad \wedge \mathcal{I}_{ind}\ \texttt{beqd}\ \sigma = \mathcal{I}_{imm}(\texttt{beqi}, b)\sigma'
\end{array}
$$

5.5 Rotation instructions

This section defines operators that do not have any Pasp counterpart, but are required in order to access the carry bits (so that they can be stored in the ram component of the store). We first define the left rotate operator.

The a register is set to be $2A + ca$ modulo 256. The truncation is carried out by the function $\stackrel{A}{\leftarrow}$. Then the carry bit is set to the most significant bit of the data register. The right rotate operator is defined similarly. The operators are also defined for the b register.

$$\mathcal{I}_{ind} : INDIRECT_INSTR \nrightarrow State_A \nrightarrow State_A$$

$$\begin{array}{l}
\forall\, \sigma : State_A \bullet \\
\quad \exists\, Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{lra}\ \sigma = \sigma \stackrel{A}{\leftarrow} (2 * A + ca) \stackrel{Ac}{\leftarrow} (A \operatorname{div} 128) \\
\quad\quad \wedge\, \mathcal{I}_{ind}\ \mathtt{lrb}\ \sigma = \sigma \stackrel{B}{\leftarrow} (2 * B + cb) \stackrel{Bc}{\leftarrow} (B \operatorname{div} 128) \\
\quad\quad \wedge\, \mathcal{I}_{ind}\ \mathtt{rra}\ \sigma = \sigma \stackrel{A}{\leftarrow} (128 * ca + A \operatorname{div} 2) \stackrel{Ac}{\leftarrow} (A \bmod 2) \\
\quad\quad \wedge\, \mathcal{I}_{ind}\ \mathtt{rrb}\ \sigma = \sigma \stackrel{B}{\leftarrow} (128 * cb + B \operatorname{div} 2) \stackrel{Bc}{\leftarrow} (B \bmod 2)
\end{array}$$

5.6 Carry bit instructions

5.6.1 Clear carry bit

$$\mathcal{I}_{ind} : INDIRECT_INSTR \nrightarrow State_A \nrightarrow State_A$$

$$\begin{array}{l}
\forall\, \sigma : State_A \bullet \\
\quad \mathcal{I}_{ind}\ \mathtt{cca}\ \sigma = \sigma \stackrel{Ac}{\leftarrow} 0 \\
\quad \wedge\, \mathcal{I}_{ind}\ \mathtt{ccb}\ \sigma = \sigma \stackrel{Bc}{\leftarrow} 0
\end{array}$$

5.6.2 Invert carry bit

The *invert* function inverts the carry bit. Because this is used as a logical operation, it is defined in terms of boolean values.

$$\mathcal{I}_{ind} : INDIRECT_INSTR \nrightarrow State_A \nrightarrow State_A$$

$$\begin{array}{l}
\forall\, \sigma : State_A \bullet \\
\quad \exists\, Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{nca}\ \sigma = \sigma \stackrel{Ac}{\leftarrow} BitNot\ ca \\
\quad\quad \wedge\, \mathcal{I}_{ind}\ \mathtt{ncb}\ \sigma = \sigma \stackrel{Bc}{\leftarrow} BitNot\ cb
\end{array}$$

5.6.3 Set carry bit

$$\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrowtail State_A \rightarrowtail State_A$$

$$\begin{array}{l} \forall\, \sigma : State_A \bullet \\ \quad \mathcal{I}_{ind}\ \texttt{sca}\ \sigma = \sigma \overset{Ac}{\leftarrow} 1 \\ \quad \wedge\ \mathcal{I}_{ind}\ \texttt{scb}\ \sigma = \sigma \overset{Bc}{\leftarrow} 1 \end{array}$$

5.6.4 Load carry bit

$$\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrowtail State_A \rightarrowtail State_A$$

$$\begin{array}{l} \forall\, \sigma : State_A \bullet \\ \quad \exists\, Store_A \mid \theta\, Store_A = storeOf_A\ \ \sigma \bullet \\ \quad\quad \mathcal{I}_{ind}\ \texttt{caldcb}\ \sigma = \sigma \overset{Ac}{\leftarrow} cb \\ \quad\quad \wedge\ \mathcal{I}_{ind}\ \texttt{cbldca}\ \sigma = \sigma \overset{Bc}{\leftarrow} ca \end{array}$$

5.7 Halt and Nop

It is necessary to put a halt instruction at the end of an Asp program for execution to terminate. The *hlt* instruction maps all continuations to the identity continuation (that is, no further computation). In addition, there are halt instructions that are dependent upon the state of the a-carry and b-carry bits. These instructions check the value of the relevant carry bit. If the bit is set, a *hlt* occurs, otherwise a *nop* operation occurs.

5.7.1 Asp Halt

$$
\begin{array}{l}
\mathcal{A}_I : PROG_ADDRESS \times INSTR \rightarrow\!\!\!\!\!+ \\
\qquad Env_A \rightarrow\!\!\!\!\!+ Cont \rightarrow\!\!\!\!\!+ State_A \rightarrow\!\!\!\!\!+ State_A \\
\hline
\forall \delta : PROG_ADDRESS;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad \mathcal{A}_I(\delta, both_indirect\ \mathtt{hlt})\rho\ \vartheta\ \sigma = \sigma \\
\qquad \wedge\ \mathcal{A}_I(\delta, both_indirect\ \mathtt{hltca})\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ ca = btrue\ \mathbf{then}\ \sigma\ \mathbf{else}\ \vartheta\ \sigma \\
\qquad \wedge\ \mathcal{A}_I(\delta, both_indirect\ \mathtt{hltcb})\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ cb = btrue\ \mathbf{then}\ \sigma\ \mathbf{else}\ \vartheta\ \sigma
\end{array}
$$

5.7.2 AspAL Halt

$$
\begin{array}{l}
\mathcal{X}_I : X_INSTR \rightarrow\!\!\!\!\!+ Env_X \rightarrow\!\!\!\!\!+ Cont \rightarrow\!\!\!\!\!+ State_A \rightarrow\!\!\!\!\!+ State_A \\
\hline
\forall \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad \mathcal{X}_I(indirect\ \mathtt{hlt})\rho\ \vartheta\ \sigma = \sigma \\
\qquad \wedge\ \mathcal{X}_I(indirect\ \mathtt{hltca})\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ ca = btrue\ \mathbf{then}\ \sigma\ \mathbf{else}\ \vartheta\ \sigma \\
\qquad \wedge\ \mathcal{X}_I(indirect\ \mathtt{hltcb})\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ cb = btrue\ \mathbf{then}\ \sigma\ \mathbf{else}\ \vartheta\ \sigma
\end{array}
$$

5.7.3 Nop

The *nop* instruction is the identity map on continuations.

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrow\!\!\!\!\!+ State_A \rightarrow\!\!\!\!\!+ State_A \\
\hline
\forall \sigma : State_A \bullet \mathcal{I}_{ind}\ \mathtt{nop}\ \sigma = \sigma
\end{array}
$$

5.8 Arithmetic operators

All the arithmetic operators are defined in both immediate and indirect forms.

5.8.1 Addition

Unsigned addition sets the register to the sum of its current contents, the data, and the corresponding carry bit. The eighth bit (numbered from 0) of this sum is assigned to the carry bit. This eighth bit is equal to n div 256. Note that the function $\stackrel{A}{\leftarrow}$ ensures that the result of the addition is truncated to fit in a $BYTE$.

$$
\begin{array}{|l}
\mathcal{I}_{imm} : \mathit{IMMED_INSTR} \times \mathit{BYTE} \rightarrow\!\!\!\!\!+ \mathit{State}_A \rightarrow\!\!\!\!\!+ \mathit{State}_A \\
\hline
\forall b : \mathit{BYTE};\ \sigma : \mathit{State}_A \bullet \\
\quad \exists \mathit{Store}_A;\ n : \mathbb{N} \mid \theta\, \mathit{Store}_A = \mathit{storeOf}_A\ \sigma \land n = A + b + ca \bullet \\
\qquad \mathcal{I}_{imm}(\texttt{auai}, b)\sigma = \sigma \stackrel{A}{\leftarrow} n \stackrel{Ac}{\leftarrow} (n \text{ div } 256) \\
\forall b : \mathit{BYTE};\ \sigma : \mathit{State}_A \bullet \\
\quad \exists \mathit{Store}_A;\ n : \mathbb{N} \mid \theta\, \mathit{Store}_A = \mathit{storeOf}_A\ \sigma \land n = B + b + cb \bullet \\
\qquad \mathcal{I}_{imm}(\texttt{buai}, b)\sigma = \sigma \stackrel{B}{\leftarrow} n \stackrel{Bc}{\leftarrow} (n \text{ div } 256)
\end{array}
$$

We define the indirect operators in terms of their immediate counterparts.

$$
\begin{array}{|l}
\mathcal{I}_{ind} : \mathit{INDIRECT_INSTR} \rightarrow\!\!\!\!\!+ \mathit{State}_A \rightarrow\!\!\!\!\!+ \mathit{State}_A \\
\hline
\forall \sigma : \mathit{State}_A \bullet \\
\quad \exists \sigma' : \mathit{State}_A;\ b : \mathit{BYTE} \mid (\sigma', b) = \mathit{byteRamOf}\ \sigma \bullet \\
\qquad \mathcal{I}_{ind}\ \texttt{auad}\ \sigma = \mathcal{I}_{imm}(\texttt{auai}, b)\sigma' \\
\qquad \land \mathcal{I}_{ind}\ \texttt{buad}\ \sigma = \mathcal{I}_{imm}(\texttt{buai}, b)\sigma'
\end{array}
$$

The data for indirect operators is obtained from the memory location pointed to by the data address register.

5.8.2 Subtraction

Unsigned subtraction can be defined in a similar way. The value is stored as a 2s complement value. If the result is less than zero then a ‘borrow’ is set in the carry bit. This allows multiple byte subtraction to be carried out.

$$
\begin{array}{l}
\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \rightarrow State_A \rightarrow State_A \\
\hline
\forall b : BYTE;\ \sigma : State_A \bullet \\
\quad \exists Store_A;\ n : \mathbb{Z} \mid \theta Store_A = storeOf_A\ \sigma \land n = A - b - ca \bullet \\
\quad\quad \mathcal{I}_{imm}(\mathtt{ausi}, b)\sigma = \\
\quad\quad\quad \textbf{if } n < 0 \textbf{ then } \sigma \stackrel{A}{\leftarrow} (n + 256) \stackrel{Ac}{\leftarrow} 1 \textbf{ else } \sigma \stackrel{A}{\leftarrow} n \stackrel{Ac}{\leftarrow} 0 \\
\forall b : BYTE;\ \sigma : State_A \bullet \\
\quad \exists Store_A;\ n : \mathbb{Z} \mid \theta Store_A = storeOf_A\ \sigma \land n = B - b - cb \bullet \\
\quad\quad \mathcal{I}_{imm}(\mathtt{busi}, b)\sigma = \\
\quad\quad\quad \textbf{if } n < 0 \textbf{ then } \sigma \stackrel{B}{\leftarrow} (n + 256) \stackrel{Bc}{\leftarrow} 1 \textbf{ else } \sigma \stackrel{B}{\leftarrow} n \stackrel{Bc}{\leftarrow} 0
\end{array}
$$

We now define the indirect form of these operators.

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrow State_A \rightarrow State_A \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists \sigma' : State_A;\ b : BYTE \mid (\sigma', b) = byteRamOf\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{ausd}\ \sigma = \mathcal{I}_{imm}(\mathtt{ausi}, b)\sigma' \\
\quad\quad \land \mathcal{I}_{ind}\ \mathtt{busd}\ \sigma = \mathcal{I}_{imm}(\mathtt{busi}, b)\sigma'
\end{array}
$$

5.8.3 Multiplication

Only the operators on the a register are specified, as only these are used. Multiplication for the a register multiplies the contents of the b register by the data passed, and adds this to the contents of the a register.

$$
\begin{array}{l}
\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \rightarrow State_A \rightarrow State_A \\
\hline
\forall b : BYTE;\ \sigma : State_A \bullet \\
\quad \exists Store_A;\ \sigma' : State_A;\ w : WORD \mid \\
\quad\quad\quad \theta Store_A = storeOf_A\ \sigma \land w = A + (B * b) \bullet \\
\quad\quad \mathcal{I}_{imm}(\mathtt{aumi}, b)\sigma = \sigma \stackrel{AB}{\leftarrow} w
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}_{ind} : INDIRECT_INSTR \rightarrow State_A \rightarrow State_A \\
\hline
\forall \sigma : State_A \bullet \\
\quad \exists \sigma' : State_A;\ b : BYTE \mid (\sigma', b) = byteRamOf\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{aumd}\ \sigma = \mathcal{I}_{imm}(\mathtt{aumi}, b)\sigma'
\end{array}
$$

5.8.4 Division

For division, an overflow condition can occur if a result is outside the range $0 \mathrel{..} 255$, or if the divisor is 0, in which case a processor halt is generated. This case is left undefined in the semantics.

$$
\begin{array}{|l}
\mathcal{I}_{imm} : \mathit{IMMED_INSTR} \times \mathit{BYTE} \rightarrow\!\!\!\!\!\!\rightarrow \mathit{State}_A \rightarrow\!\!\!\!\!\!\rightarrow \mathit{State}_A \\
\hline
\forall b : \mathit{BYTE};\ \sigma : \mathit{State}_A \mid b \neq 0 \bullet \\
\quad \exists \mathit{Store}_A \mid \theta\, \mathit{Store}_A = \mathit{storeOf}_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{imm}(\mathtt{audi}, b)\sigma = \sigma \stackrel{A}{\leftarrow} ((A \dagger B) \bmod b) \stackrel{B}{\leftarrow} ((A \dagger B) \operatorname{div} b)
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{I}_{ind} : \mathit{INDIRECT_INSTR} \rightarrow\!\!\!\!\!\!\rightarrow \mathit{State}_A \rightarrow\!\!\!\!\!\!\rightarrow \mathit{State}_A \\
\hline
\forall \sigma : \mathit{State}_A \bullet \\
\quad \exists \sigma' : \mathit{State}_A;\ b : \mathit{BYTE} \mid (\sigma', b) = \mathit{byteRamOf}\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{audd}\ \sigma = \mathcal{I}_{imm}(\mathtt{audi}, b)\sigma'
\end{array}
$$

5.9 Comparison operators

The meaning of these operators depends not only on the value contained in the appropriate register, but also on the carry bit.

$$
\begin{array}{|l}
\mathcal{I}_{imm} : \mathit{IMMED_INSTR} \times \mathit{BYTE} \rightarrow\!\!\!\!\!\!\rightarrow \mathit{State}_A \rightarrow\!\!\!\!\!\!\rightarrow \mathit{State}_A \\
\hline
\forall b : \mathit{BYTE};\ \sigma : \mathit{State}_A \bullet \\
\quad \exists \mathit{Store}_A \mid \theta\, \mathit{Store}_A = \mathit{storeOf}_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{imm}(\mathtt{auci}, b)\sigma = \\
\quad\quad\quad \sigma \stackrel{Ac}{\leftarrow} (\mathbf{if}\ b < A\ \mathbf{then}\ 1\ \mathbf{else\ if}\ A < b\ \mathbf{then}\ 0\ \mathbf{else}\ ca) \\
\quad\quad \wedge\ \mathcal{I}_{imm}(\mathtt{buci}, b)\sigma = \\
\quad\quad\quad \sigma \stackrel{Bc}{\leftarrow} (\mathbf{if}\ b < B\ \mathbf{then}\ 1\ \mathbf{else\ if}\ B < b\ \mathbf{then}\ 0\ \mathbf{else}\ cb)
\end{array}
$$

Again we define the indirect operators in terms of the immediate operators.

$$\mathcal{I}_{ind} : INDIRECT_INSTR \leftrightarrow State_A \leftrightarrow State_A$$
$$\forall \sigma : State_A \bullet$$
$$\quad \exists \sigma' : State_A;\ b : BYTE \mid (\sigma', b) = byteRamOf\ \sigma \bullet$$
$$\quad\quad \mathcal{I}_{ind}\ \texttt{aucd}\ \sigma = \mathcal{I}_{imm}(\texttt{auci}, b)\sigma'$$
$$\quad\quad \land \mathcal{I}_{ind}\ \texttt{bucd}\ \sigma = \mathcal{I}_{imm}(\texttt{buci}, b)\sigma'$$

5.10 Data address manipulations

5.10.1 Immediate data address manipulations

The immediate instruction *dix* overwrites the high byte of the data address register D by the immediate data; *dxi* overwrites the low byte.

$$\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \leftrightarrow State_A \leftrightarrow State_A$$
$$\forall b : BYTE;\ \sigma : State_A \bullet$$
$$\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet$$
$$\quad\quad \mathcal{I}_{imm}(\texttt{dix}, b)\sigma = \sigma \stackrel{D}{\leftarrow} (b \dagger theLoByte\ D)$$
$$\quad\quad \land \mathcal{I}_{imm}(\texttt{dxi}, b)\sigma = \sigma \stackrel{D}{\leftarrow} (theHiByte\ D \dagger b)$$

The immediate instruction `dpi` increments the data address register D by the immediate data. This can be used to access adjacent locations in *ram* so that Pasp values spread over two bytes can be determined without altering the contents of the a and b registers. This instruction can be used where locations that are calculated at run-time need to be referenced (specifically for accessing array elements). Before using it, the a and b registers must be loaded with the required address.

$$\mathcal{I}_{imm} : IMMED_INSTR \times BYTE \leftrightarrow State_A \leftrightarrow State_A$$
$$\forall b : BYTE;\ \sigma : State_A \bullet$$
$$\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet$$
$$\quad\quad \mathcal{I}_{imm}(\texttt{dpi}, b)\sigma = \sigma \stackrel{D}{\leftarrow} (D + asSigned\ b)$$

5.10.2 Indirect data address manipulations

The instruction `daldab` sets the D register to $a \dagger b$; `daldc` sets the D register to C.

$$
\begin{array}{l}
\mathcal{I}_{ind} : \mathit{INDIRECT_INSTR} \leftrightarrow \mathit{State}_A \leftrightarrow \mathit{State}_A \\
\hline
\forall \sigma : \mathit{State}_A \bullet \\
\quad \exists \mathit{Store}_A \mid \theta\, \mathit{Store}_A = \mathit{storeOf}_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{daldab}\ \sigma = \sigma \stackrel{D}{\leftarrow} (A \dagger B) \\
\quad\quad \land\ \mathcal{I}_{ind}\ \mathtt{daldc}\ \sigma = \sigma \stackrel{D}{\leftarrow} C
\end{array}
$$

The instruction `cldda` saves the D register to C.

$$
\begin{array}{l}
\mathcal{I}_{ind} : \mathit{INDIRECT_INSTR} \leftrightarrow \mathit{State}_A \leftrightarrow \mathit{State}_A \\
\hline
\forall \sigma : \mathit{State}_A \bullet \\
\quad \exists \mathit{Store}_A \mid \theta\, \mathit{Store}_A = \mathit{storeOf}_A\ \sigma \bullet \\
\quad\quad \mathcal{I}_{ind}\ \mathtt{cldda}\ \sigma = \sigma \stackrel{C}{\leftarrow} D
\end{array}
$$

5.10.3 AspAL data instructions

The three instructions to set the data register allow the linker to arrange the memory for each module. In the dynamic semantics the relevant data address is converted from an offset to an absolute address.

$$
\mathit{bottomHeap}, \mathit{topStack}, \mathit{topOp} : \mathit{DATA_ADDRESS}
$$

$$
\begin{array}{l}
\mathcal{X}_I : \mathit{X_INSTR} \leftrightarrow \mathit{Env}_X \leftrightarrow \mathit{Cont} \leftrightarrow \mathit{State}_A \leftrightarrow \mathit{State}_A \\
\hline
\forall \delta : \mathit{DATA_ADDRESS};\ \rho : \mathit{Env}_X;\ \vartheta : \mathit{Cont};\ \sigma : \mathit{State}_A \bullet \\
\quad \mathcal{X}_I(\mathit{xData}(\mathit{xsdrh}, \delta))\rho\ \ \vartheta\ \ \sigma = \vartheta(\sigma \stackrel{D}{\leftarrow} (\delta + \mathit{bottomHeap})) \\
\quad \land\ \mathcal{X}_I(\mathit{xData}(\mathit{xsdrs}, \delta))\rho\ \ \vartheta\ \ \sigma = \vartheta(\sigma \stackrel{D}{\leftarrow} (\mathit{topStack} - \delta)) \\
\quad \land\ \mathcal{X}_I(\mathit{xData}(\mathit{xsdro}, \delta))\rho\ \ \vartheta\ \ \sigma = \vartheta(\sigma \stackrel{D}{\leftarrow} (\mathit{topOp} + \delta))
\end{array}
$$

The *xsdr* instruction is used only by the linker, once the module offsets for stack, heap, and topop have been evaluated.

$$
\begin{array}{|l}
\mathcal{X}_I : X_INSTR \leftrightarrow Env_X \leftrightarrow Cont \leftrightarrow State_A \leftrightarrow State_A \\
\hline
\forall \delta : DATA_ADDRESS;\ \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \mathcal{X}_I(xData(xsdr, \delta))\rho\ \vartheta\ \sigma = \vartheta(\sigma \overset{D}{\leftarrow} \delta)
\end{array}
$$

The two instructions to load a data address into the *ab* registers manipulate an absolute address and a relative heap address respectively.

$$
\begin{array}{|l}
\mathcal{X}_I : X_INSTR \leftrightarrow Env_X \leftrightarrow Cont \leftrightarrow State_A \leftrightarrow State_A \\
\hline
\forall \delta : DATA_ADDRESS;\ \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \mathcal{X}_I(xData(xlada, \delta))\rho\ \vartheta\ \sigma = \vartheta(\sigma \overset{AB}{\leftarrow} \delta) \\
\quad \land\ \mathcal{X}_I(xData(xlrda, \delta))\rho\ \vartheta\ \sigma = \vartheta(\sigma \overset{AB}{\leftarrow} (\delta + bottomHeap))
\end{array}
$$

5.11 Program address manipulations

Asp manipulates the program counter directly. AspAL abstracts away from this, and uses labels.

A *jump* instruction is a conditional *goto*. If the carry bit is set to true, then the jump is carried out, else the current continuation is used.

5.11.1 Asp Goto

The Asp instructions go to the program location given by the data provided. This is defined by applying the appropriate continuation to the current state. The indirect *goto* instruction uses the value produced by the join of the *a* and *b* registers as the place to go to. The immediate *goto* uses the immediate data values as part of the instruction which constitute the place to go to. Asp has an immediate relative goto (add the immediate data to the program counter)

$$
\begin{array}{|l}
\mathcal{A}_I : PROG_ADDRESS \times INSTR \leftrightarrow \\
\qquad Env_A \leftrightarrow Cont \leftrightarrow State_A \leftrightarrow State_A \\
\hline
\forall \delta : PROG_ADDRESS;\ b : BYTE;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \mathcal{A}_I(\delta, asp_immediate(\mathtt{ppi}, b))\rho\ \vartheta\ \sigma = \rho(\delta + asSigned\ \ b)\sigma
\end{array}
$$

and an absolute indirect goto (set the program counter to the indirect data value).

$$\mathcal{A}_I : PROG_ADDRESS \times INSTR \rightarrowtail Env_A \rightarrowtail Cont \rightarrowtail State_A \rightarrowtail State_A$$

$$\forall \delta : PROG_ADDRESS;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet \\ \quad \exists Store_A \mid \theta Store_A = storeOf_A\ \sigma \bullet \\ \quad\quad \mathcal{A}_I(\delta, asp_indirect\ \texttt{pab})\rho\ \vartheta\ \sigma = \rho(A \dagger B)\sigma$$

5.11.2 AspAL Gotos

AspAL has its own specially defined instructions for gotos. Labelled instructions in the program are used to indicate places to go to.

$$\mathcal{X}_I : X_INSTR \rightarrowtail Env_X \rightarrowtail Cont \rightarrowtail State_A \rightarrowtail State_A$$

$$\forall \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\ \quad \exists Store_A \mid \theta Store_A = storeOf_A\ \sigma \bullet \\ \quad\quad \mathcal{X}_I(xIndirect\ xag)\rho\ \vartheta\ \sigma = \rho(A \dagger B)\sigma$$

xrg allows unlimited gotos, and overwrites the a and b registers. *xrgs* allows short gotos, and does not overwrite the a and b registers.

$$\mathcal{X}_I : X_INSTR \rightarrowtail Env_X \rightarrowtail Cont \rightarrowtail State_A \rightarrowtail State_A$$

$$\forall l : LABEL;\ \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\ \quad \exists a, b : BYTE \bullet \\ \quad\quad \mathcal{X}_I(xLabel(xrg, l))\rho\ \vartheta\ \sigma = \rho\ l(\sigma \overset{AB}{\leftarrow} (a \dagger b)) \\ \forall l : LABEL;\ \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\ \quad \mathcal{X}_I(xLabel(xrgs, l))\rho\ \vartheta\ \sigma = \rho\ l\ \sigma$$

5.11.3 Asp Jumps

Asp has an immediate relative jumps (add the immediate data to the program counter)

$$
\begin{array}{l}
\mathcal{A}_I : PROG_ADDRESS \times INSTR \rightarrow \\
\qquad Env_A \rightarrow Cont \rightarrow State_A \rightarrow State_A \\
\hline
\forall \delta : PROG_ADDRESS;\ b : BYTE;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad \mathcal{A}_I(\delta, asp_immediate(\mathtt{cappi}, b))\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ ca = btrue\ \mathbf{then}\ \rho(\delta + asSigned\ b)\sigma\ \mathbf{else}\ \vartheta\ \sigma \\
\qquad \wedge\ \mathcal{A}_I(\delta, asp_immediate(\mathtt{cbppi}, b))\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ cb = btrue\ \mathbf{then}\ \rho(\delta + asSigned\ b)\sigma\ \mathbf{else}\ \vartheta\ \sigma
\end{array}
$$

and absolute indirect jumps (set the program counter to the indirect data value).

$$
\begin{array}{l}
\mathcal{A}_I : PROG_ADDRESS \times INSTR \rightarrow \\
\qquad Env_A \rightarrow Cont \rightarrow State_A \rightarrow State_A \\
\hline
\forall \delta : PROG_ADDRESS;\ \rho : Env_A;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \exists Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad \mathcal{A}_I(\delta, asp_indirect\ \mathtt{capab})\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ ca = btrue\ \mathbf{then}\ \rho(A \dagger B)\sigma\ \mathbf{else}\ \vartheta\ \sigma \\
\qquad \wedge\ \mathcal{A}_I(\delta, asp_indirect\ \mathtt{cbpab})\rho\ \vartheta\ \sigma = \\
\qquad\quad \mathbf{if}\ cb = btrue\ \mathbf{then}\ \rho(A \dagger B)\sigma\ \mathbf{else}\ \vartheta\ \sigma
\end{array}
$$

5.11.4 AspAL Jumps

AspAL has specially defined instructions for jumps. Labelled instructions in the program are used to indicate places to jump to. *xrjal* and *xrjbl* allow unlimited long jumps, and overwrite the a and b registers. *xrja* and *xrjb* allow limited jumps, and do not overwrite the a and b registers.

$$
\begin{array}{l}
\mathcal{X}_I : X_INSTR \rightarrow Env_X \rightarrow Cont \rightarrow State_A \rightarrow State_A \\
\hline
\forall\, l : LABEL;\ \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \exists\, Store_A;\ a, b : BYTE;\ \sigma' : State_A \mid \\
\qquad\qquad \theta\, Store_A = storeOf_A\ \sigma \wedge \sigma' = \sigma \overset{AB}{\leftarrow} (a \dagger b) \bullet \\
\qquad\quad \mathcal{X}_I(xLabel(xrjal, l))\rho\ \vartheta\ \sigma = \\
\qquad\qquad \mathbf{if}\ ca = btrue\ \mathbf{then}\ \rho\ l\ \sigma'\ \mathbf{else}\ \vartheta\ \sigma' \\
\qquad\quad \wedge\ \mathcal{X}_I(xLabel(xrjbl, l))\rho\ \vartheta\ \sigma = \\
\qquad\qquad \mathbf{if}\ cb = btrue\ \mathbf{then}\ \rho\ l\ \sigma'\ \mathbf{else}\ \vartheta\ \sigma' \\
\forall\, l : LABEL;\ \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A \bullet \\
\quad \exists\, Store_A \mid \theta\, Store_A = storeOf_A\ \sigma \bullet \\
\qquad\quad \mathcal{X}_I(xLabel(xrja, l))\rho\ \vartheta\ \sigma = \\
\qquad\qquad \mathbf{if}\ ca = btrue\ \mathbf{then}\ \rho\ l\ \sigma\ \mathbf{else}\ \vartheta\ \sigma \\
\qquad\quad \wedge\ \mathcal{X}_I(xLabel(xrjb, l))\rho\ \vartheta\ \sigma = \\
\qquad\qquad \mathbf{if}\ cb = btrue\ \mathbf{then}\ \rho\ l\ \sigma\ \mathbf{else}\ \vartheta\ \sigma
\end{array}
$$

5.12 Remaining XAspAL instructions

5.12.1 Procedure and Function calling

The instruction *xCall*, which calls a particular procedure or function, has a very loose definition in that its dynamic semantics are an arbitrary state change. This is because the meaning of the function called could be (more or less) any computable function.

$$
\begin{array}{l}
\mathcal{X}_I : X_INSTR \rightarrow Env_X \rightarrow Cont \rightarrow State_A \rightarrow State_A \\
\hline
\forall\, \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A;\ \xi : IDENTIFIER \bullet \\
\quad \exists\, \sigma' : State_A \bullet \\
\qquad\quad \mathcal{X}_I(xCall\ \xi)\rho\ \vartheta\ \sigma = \vartheta\ \sigma'
\end{array}
$$

$$
\begin{array}{l}
\mathcal{X}_I : X_INSTR \rightarrow Env_X \rightarrow Cont \rightarrow State_A \rightarrow State_A \\
\hline
\forall\, \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A;\ \xi : IDENTIFIER;\ l : LABEL \bullet \\
\quad \mathcal{X}_I(xProc(\xi, l))\rho\ \vartheta\ \sigma = \vartheta\ \sigma
\end{array}
$$

5.12.2 Operator calling

The instruction *xCallOp*, which calls one of the optimised operator templates, has a very loose definition in that its dynamic semantics are an arbitrary state change.

$$\mathcal{X}_I : X_INSTR \nrightarrow Env_X \nrightarrow Cont \nrightarrow State_A \nrightarrow State_A$$

$$\begin{array}{l} \forall \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A;\ l : LABEL \bullet \\ \quad \exists \sigma' : State_A \bullet \\ \quad\quad \mathcal{X}_I(xCallOp\ l)\rho\ \vartheta\ \sigma = \vartheta\ \sigma' \end{array}$$

5.12.3 AspAL Load label

The load label instruction is supplied so that the hexer can replace it with the $\mathcal{O}_{lwi}$ sequence once the corresponding program address has been determined.

$$\mathcal{X}_I : X_INSTR \nrightarrow Env_X \nrightarrow Cont \nrightarrow State_A \nrightarrow State_A$$

$$\begin{array}{l} \forall \rho : Env_X;\ \vartheta : Cont;\ \sigma : State_A;\ l : LABEL \bullet \\ \quad \mathcal{X}_I(xLabel(xll, l))\rho\ \vartheta\ \sigma = \vartheta(\sigma \overset{AB}{\leftarrow} l) \end{array}$$

The a register is loaded with the high byte of the label and the b register is loaded with the low byte.

5.13 Program semantics

5.13.1 Asp program semantics

A complete Asp program maps input streams to output streams. We only define the meaning of a small subset of legal Asp programs. The Asp programs that we consider occupy contiguous blocks of program address space and the last instruction is a halt. If the program address register is set to a value outside the space occupied by the program, then the semantics are

undefined. Note that it is not possible to check this statically, as the program address register can be set to dynamic values. We define the meaning as follows.

$$
\begin{array}{l}
\mathcal{A}_P : ASP_PROGRAM \mathrel{+\!\!\!\rightarrow} State_A \mathrel{+\!\!\!\rightarrow} Store_A \\
\hline
\forall \delta start : PROG_ADDRESS;\ I : PROG_ADDRESS \mathrel{+\!\!\!\rightarrow} INSTR; \\
\qquad\qquad \sigma : State_A \mid \\
\qquad\quad (\delta start, I) \in ASP_PROGRAM \bullet \\
\quad \exists \delta last : PROG_ADDRESS;\ \rho : Env_X \mid \\
\qquad\qquad \delta last = max(\operatorname{dom} I) \land \operatorname{dom} \rho = \operatorname{dom} I \\
\qquad\qquad \land\ \rho\ \delta last = \mathcal{A}_I(\delta last, I\ \delta last)\rho(\operatorname{id} State_A) \\
\qquad\qquad \land\ (\ \forall \delta : \operatorname{dom} I \setminus \{\delta last\} \bullet \\
\qquad\qquad\quad \rho\ \delta = \mathcal{A}_I(\delta, I\ \delta)\rho(\rho(\delta + 1))\ ) \bullet \\
\qquad\quad \mathcal{A}_P(\delta start, I)\sigma = \\
\qquad\qquad outOf_A(\mathcal{A}_I(\delta start, I\ \delta start)\rho(\rho(\delta start + 1))\sigma)
\end{array}
$$

Program execution is defined to start at the instruction whose address is given by $\delta start$. The environment ρ is a labelled set of continuations, with one defined for each program instruction ($\operatorname{dom} \rho = \operatorname{dom} I$). The last continuation represents termination, while for $\delta \neq \delta last$, the δth continuation represents the meaning of the program from the δth instruction to the end.

5.13.2 AspAL module semantics

An AspAL module consists of a sequence of label instructions separated by (possibly empty) sequences of non-label instructions. A meaning is defined only if the labels are unique.

$$X_NON_LABEL == X_INSTR \setminus \operatorname{ran} xLabel$$

A fragment is a sequence of non-label instructions followed by a single label instruction.

$$
\begin{array}{l}
Frag == \{\ s : \operatorname{seq} X_INSTR \mid \\
\qquad\qquad \operatorname{ran}(front\ \ s) \subseteq X_NON_LABEL \land last\ \ s \in \operatorname{ran} xLabel\ \}
\end{array}
$$

The final fragment is all non-label instructions, terminated with a halt.

$$LastFrag == \{\ s : \mathrm{seq}\, X_NON_LABEL \bullet s \frown \langle indirect\ \texttt{hlt} \rangle\ \}$$

A module is a sequence of fragments followed by a non-empty sequence of non-label instructions. The dynamic meaning of an AspAL module is

$$\begin{array}{l}
\mathcal{X}_P : ASPAL_MODULE \nrightarrow State_A \nrightarrow Store_A \\
\hline
\forall \Phi : \mathrm{seq}\, Frag;\ I : LastFrag;\ \sigma : State_A \bullet \\
\quad \exists \rho : Env_X;\ \Theta : \mathrm{seq}\, Cont \mid \\
\qquad\qquad \mathrm{dom}\,\Theta = \mathrm{dom}\,\Phi \wedge last\ \Theta = \mathcal{X}_{I^*}\ I\ \rho(\mathrm{id}\, State_A) \\
\qquad\qquad \wedge\ (\ \forall j : \mathrm{dom}(front\ \Theta) \bullet \\
\qquad\qquad\quad \Theta\ j = \mathcal{X}_{I^*}(front(\Phi(j+1)))\rho(\Theta(j+1))\ ) \\
\qquad\qquad \wedge\ (\ \forall i : \mathrm{dom}\,\Theta \bullet \rho(xLabel^{\sim}\ (last(\Phi\ i))).2 = \Theta\ i\ ) \bullet \\
\qquad \mathcal{X}_P(\frown/\ \Phi \frown I)\sigma = \\
\qquad\qquad outOf_A(\mathcal{X}_{I^*}(front(\Phi\ 1))\rho(\Theta\ 1)\sigma)
\end{array}$$

Execution is defined to start at the first instruction in the sequence. Θ is a sequence of continuations, with one defined for each fragment: $\mathrm{dom}\,\Theta = \mathrm{dom}\,\Phi$. Every fragment is mapped under the environment to the corresponding continuation: $\rho(xLabel^{\sim}(last(\Phi\ i))).2 = \Theta\ i$. The last continuation represents termination. For $j < \#\Phi$, the jth continuation represents the meaning of the program from the jth fragment to the end.

A General DeCCo Toolkit

This section contains definitions in common to several of the DeCCo specifications.

A.1 Raising to a power

Z has no 'raise to a power' operator, so we define one.

$$\textbf{function}\ 30\ \textbf{leftassoc}(_ \uparrow _)$$

$$_ \uparrow _ : \mathbb{N}_1 \times \mathbb{N} \to \mathbb{N}$$

$$\forall n : \mathbb{N}_1 \bullet n \uparrow 0 = 1$$
$$\forall n, p : \mathbb{N}_1 \bullet n \uparrow p = n * (n \uparrow (p - 1))$$

A.1.1 Byte

The byte type is defined so as to take advantage of the greater speed offered by using single register operations in the compiled code.

$$BYTE == 0 \mathrel{..} 255$$

A.1.2 Boolean

We model the boolean type with a free type of two constants:

$$BOOLEAN ::= ptrue \mid pfalse$$

Note: Z has *true* and *false* as keywords, so those names cannot be used in the definition of *BOOLEAN*.

A.2 Bits

In order to define the logical operators such as *or*, *and*, and *xor*, it is necessary to be able to write numbers as sequences of bits. We first define a *BIT*.

$$BIT == \{0, 1\}$$

The first element of the sequence is the least significant bit.

$$\begin{array}{l} NatToBits : \mathbb{N} \rightarrow \operatorname{seq} BIT \\ \hline NatToBits\, 0 = \langle\,\rangle \\ \forall\, n : \mathbb{N}_1 \bullet NatToBits\, n = \langle n \bmod 2 \rangle \frown NatToBits(n \operatorname{div} 2) \end{array}$$

This definition ensures that the most significant bit (if there is one) is always one (i.e. there is no leading zero).

The inverse of the above function is given by

$$\begin{array}{l} BitsToNat : \operatorname{seq} BIT \twoheadrightarrow \mathbb{N} \\ \hline BitsToNat\langle\,\rangle = 0 \\ \forall\, b : BIT;\ B : \operatorname{seq} BIT \bullet \\ \quad BitsToNat(\langle b \rangle \frown B) = b + 2 * BitsToNat\ B \end{array}$$

Logical operators are defined for single bits. The definitions are given in terms of arithmetic operations, for concision.

$$\begin{array}{l} BitNot : BIT \rightarrowtail\!\!\!\!\twoheadrightarrow BIT \\ BitOr, BitXor, BitAnd : BIT \times BIT \rightarrow BIT \\ \hline BitNot = \{0 \mapsto 1, 1 \mapsto 0\} \\ \forall\, b, c : BIT \bullet \\ \quad BitOr(b, c) = 1 - (1 - b) * (1 - c) \\ \quad \land BitXor(b, c) = (b + c) \bmod 2 \\ \quad \land BitAnd(b, c) = b * c \end{array}$$

These definitions are extended to bit sequences. First we define OR. Recall that the empty sequence represents zero.

function 30 **leftassoc**($_OR_$)

$$
\begin{array}{|l}
_\,OR\,_ : \mathrm{seq}\,BIT \times \mathrm{seq}\,BIT \rightarrow \mathrm{seq}\,BIT \\
\hline
\forall\, s,t : \mathrm{seq}\,BIT;\ b,c : BIT \bullet \\
\qquad s\ OR\ \langle\rangle = s \land \langle\rangle\ OR\ t = t \\
\qquad \land\ (\langle b\rangle \frown s) OR (\langle c\rangle \frown t) = \langle BitOr(b,c)\rangle \frown (s\ OR\ t)
\end{array}
$$

function 30 **leftassoc**($_XOR_$)

$$
\begin{array}{|l}
\,XOR\, : \mathrm{seq}\,BIT \times \mathrm{seq}\,BIT \rightarrow \mathrm{seq}\,BIT \\
\hline
\forall\, s,t : \mathrm{seq}\,BIT;\ b,c : BIT \bullet \\
\qquad s\ XOR\ \langle\ \rangle = s \land \langle\ \rangle\ XOR\ t = t \\
\qquad \land\ (\langle b\rangle \frown s) XOR (\langle c\rangle \frown t) = \langle BitXor(b,c)\rangle \frown (s\ XOR\ t)
\end{array}
$$

function 30 **leftassoc**($_AND_$)

$$
\begin{array}{|l}
\,AND\, : \mathrm{seq}\,BIT \times \mathrm{seq}\,BIT \rightarrow \mathrm{seq}\,BIT \\
\hline
\forall\, s,t : \mathrm{seq}\,BIT;\ b,c : BIT \bullet \\
\qquad s\ AND\ \langle\ \rangle = s \land \langle\ \rangle\ AND\ t = t \\
\qquad \land\ (\langle b\rangle \frown s) AND (\langle c\rangle \frown t) = \langle BitAnd(b,c)\rangle \frown (s\ AND\ t)
\end{array}
$$

B AspAL Toolkit definitions

B.1 Addresses and values

In general Asp locations and registers can contain only eight bits (the exceptions are the program, data, and C registers). The storage of Asp is byte-oriented. A $BYTE$ need not be interpreted as its (unsigned) value, but it is convenient just to regard it as data at the syntactic level, rather than distinguishing unsigned and signed usages.

In order to model the input and output streams of a program, we must extend the definition of what may be stored at an address to include streams of bytes. This can be compared with the Pasp definition of $VALUE$.

$$VALUE_A ::= byte\langle\langle BYTE\rangle\rangle \mid Bstream\langle\langle \mathrm{seq}\, BYTE\rangle\rangle$$

The data and program address registers are sixteen bits. We define the $WORD$ value to model this.

$$WORD == 0 \,..\, (2 \uparrow 16)$$

Note that a $WORD$ and an $Unsigned$ in the Pasp sense are identical. However, it is worth specifying them separately in case there is a decision to change the size of the numbers in Pasp.

Addresses are modelled by natural numbers. The data and program memory addresses are modelled separately, although there are the same number of locations for each in the current design. These are used by the Asp specification, however the AspAL specification does not use program addresses as the destination of jumps but instead labels. Label addresses are modelled as words and are defined here for consistency.

$$DATA_ADDRESS == WORD$$
$$PROG_ADDRESS == WORD$$
$$LABEL == WORD$$

Our intention is to use the a-carry bit for testing conditions. Because Pasp

integers and unsigneds are sixteen bits, we need to define many of the instructions for both the a-reg and the b-reg.

B.2 Value conversion

Pasp locations can contain several types of value. However, AspAL addresses model physical hardware, and are therefore constrained in the values that they may contain. In addition, those values may be interpreted in different ways (as signed or unsigned for example). It is therefore necessary to define some functions to convert between different representations of integers. (Some other conversion functions are given in the appendix.)

B.2.1 Conversion between words and bytes

We define functions to find the high and low byte of a word, and to join two bytes to form a word.

$$\begin{array}{|l} \mathit{theLoByte}, \mathit{theHiByte} : \mathit{WORD} \twoheadrightarrow \mathit{BYTE} \\ \hline \forall\, w : \mathit{WORD} \bullet \\ \quad \mathit{theLoByte}\ w = w \bmod 256 \land \mathit{theHiByte}\ w = w \operatorname{div} 256 \end{array}$$

$\mathit{function}\ \mathit{left}\ 30(_ \dagger _)$

$$\begin{array}{|l} _ \dagger _ : \mathit{BYTE} \times \mathit{BYTE} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathit{WORD} \\ \hline \forall\, lo, hi : \mathit{BYTE} \bullet hi \dagger lo = hi * 256 + lo \end{array}$$

The map *theHiByte* is a function because *WORD* is a 16-bit number, and therefore $\mathit{theHiByte}(w) \leq 128$. This definition implies that[1]:

---

[1]Proof:

$\mathit{theHiByte}\ w \dagger \mathit{theLoByte}\ w =$

$= (w \operatorname{div} 256) \dagger (w \bmod 256)$ defn *theHiByte*, *theLoByte*

$$w : WORD \vdash theHiByte\ w \dagger theLoByte\ w = w$$

Similarly, the inverse relationship holds[2]:

$$b, b' : BYTE \vdash theLoByte(b \dagger b') = b' \land theHiByte(b \dagger b') = b$$

B.2.2 Conversion between booleans and bits

$$bfalse == 0$$
$$btrue == 1$$

$$BoolToBit : BOOLEAN \rightarrowtail\!\!\!\!\twoheadrightarrow BIT$$
$$BoolToBit = \{pfalse \mapsto bfalse, ptrue \mapsto btrue\}$$

$= 256 * (w \operatorname{div} 256) + w \bmod 256$ — defn $\dagger$

$= w$ — ZRM defn div, mod

□

[2]Proof: by a simple manipulation of the function definitions and the properties of arithmetic operations.

$theLoByte(b_1 \dagger b_2)$

$= theLoByte(b_1 * 256 + b_2)$ — defn $\dagger$

$= (b_1 * 256 + b_2) \bmod 256$ — defn *theLoByte*

$= b_2 \bmod 256$ — prop mod

$= b_2$ — prop mod; $b < 256$

□

$theHiByte(b_1 \dagger b_2)$

$= theHiByte(b_1 * 256 + b_2)$ — defn $\dagger$

$= (b_1 * 256 + b_2) \operatorname{div} 256$ — defn *theHiByte*

$= b_1$ — defn div; $b_2 < 256$

□

B.2.3 Conversion from unsigned to signed bytes

$$asSigned : BYTE \rightarrowtail\!\!\!\!\twoheadrightarrow -128 \mathrel{..} 127$$

References

[Formaliser] Susan Stepney. *The Formaliser Manual.* Logica.

[ISO-Z] ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics.* 2002.

[Spivey 1992] J. Michael Spivey. *The Z Notation: a Reference Manual.* Prentice Hall, 2nd edition, 1992.

[Stepney *et al.* 1991] Susan Stepney, Dave Whitley, David Cooper, and Colin Grant. A demonstrably correct compiler. *BCS Formal Aspects of Computing*, 3:58–101, 1991.

[Stepney 1993] Susan Stepney. *High Integrity Compilation: A Case Study.* Prentice Hall, 1993.

[Stepney 1998] Susan Stepney. Incremental development of a high integrity compiler: experience from an industrial development. In *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98), Washington DC*, 1998.

[Stringer-Calvert *et al.* 1997] David W.J. Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: A case study. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, number 1313 in Lecture Notes in Computer Science, pages 573–588. Springer Verlag, September 1997.

Index

$\stackrel{A}{\leftarrow}$, 16
$\stackrel{AB}{\leftarrow}$, 16
$\stackrel{Ac}{\leftarrow}$, 17
$\stackrel{Bc}{\leftarrow}$, 17
$\stackrel{B}{\leftarrow}$, 16
$\stackrel{C}{\leftarrow}$, 17
$\stackrel{D}{\leftarrow}$, 18
$\stackrel{R}{\leftarrow}$, 15
$\stackrel{W}{\leftarrow}$, 15
†, 47

$\uparrow$, 43

A, 12
`aand`, 4, 26
`aani`, 4, 26
`abldc`, 23
`abldc`, 4
`abldd`, 4, 24
`abstd`, 4, 24
`audi`, 4
`aumi`, 4
`aeqi`, 27
`aeqd`, 4, 27
`aeqi`, 4
$\mathcal{A}_I$, 20
$\mathcal{A}_{I^*}$, 21
`aldb`, 4, 22
`aldd`, 4, 23
`aldi`, 4, 22
AND, 45
`aord`, 4, 25
`aori`, 4, 25
$\mathcal{A}_P$, 41
ASPAL_MODULE, 9
ASPAL_PROGRAM, 9
asp_immediate, 6
ASP_IMMED_INSTR, 5
asp_indirect, 6
ASP_INDIRECT_INSTR, 5
ASP_PROGRAM, 8
asSigned, 49
`astd`, 4, 23
`auad`, 4, 31
`auai`, 4, 31
`aucd`, 4, 33
`auci`, 4, 33
`audd`, 4, 33
`aumd`, 4, 32
`aumi`, 32
`ausd`, 4, 32
`audi`, 33
`ausi`, 4, 31
`axod`, 4, 26
`axoi`, 4, 26

B, 12
`band`, 4, 26
`bani`, 4, 26
`beqd`, 4, 27
`beqi`, 4, 27
bfalse, 48
BIT, 44
BitAnd, 44
BitNot, 44
BitOr, 44
BitsToNat, 44

BitXor, 44
bldа, 4, 22
bldd, 4, 23
bldi, 4, 22
BOOLEAN, 43
BoolToBit, 48
bord, 4, 25
bori, 4, 25
both_immediate, 6
both_indirect, 6
bottomHeap, 35
bstd, 4, 23
Bstream, 46
btrue, 48
buad, 4, 31
buai, 4, 31
bucd, 4, 33
buci, 4, 33
busd, 4, 32
busi, 4, 31
bxod, 4, 26
bxoi, 4, 26
BYTE, 43
byte, 46
byteRamOf, 19

C, 12
ca, 12
$\mathcal{C}_{AI}$, 10
caldcb, 4, 29
capab, 5, 38
cappi, 5, 37
$\mathcal{C}_{AspAI}$, 10
cb, 12
cbldca, 4, 29
cbpab, 5, 38
cbppi, 5, 37
cca, 4, 28
ccb, 4, 28
$\mathcal{C}_{HEX}$, 10
cldab, 4, 23
cldd, 4, 24
cldda, 4, 35
continuation, 13
cstd, 4, 24

D, 12
daldab, 4, 35
daldc, 4
DATA_ADDRESS, 46
dix, 4, 34
dpi, 4, 34
dxi, 4, 34

Frag, 41

hlt, 4, 29, 30
hltca, 4, 29, 30
hltcb, 4, 29, 30

Iadd, 13
ID, 7
$\mathcal{I}_{imm}$, 20
$\mathcal{I}_{ind}$, 20
immediate, 7
IMMED_INSTR, 4
indirect, 7
INDIRECT_INSTR, 4
INSTR, 6

LABEL, 46
LastFrag, 42
lra, 4, 28
lrb, 4, 28

NatToBits, 44

`nca`, 4, 28
`ncb`, 4, 28
`nop`, 4, 30

Oadd, 13
OR, 45
$outOf_A$, 13

`pab`, 5, 37
pfalse, 43
`ppi`, 5, 36
PROG_ADDRESS, 46
ptrue, 43

ram, 12
ramOf, 18
Register, 12
`rra`, 4, 28
`rrb`, 4, 28

`sca`, 4, 29
`scb`, 4, 29
$Store_A$, 12
$storeOf_A$, 13

theHiByte, 47
theLoByte, 47
topOp, 35
topStack, 35

updateRam, 15

$VALUE_A$, 46

WORD, 46
wordRamOf, 19

xag, 5, 37
xBvar, 7
xCall, 7, 39
xCallOp, 7, 40
xConst, 7
xData, 7
X_DATA_INSTR, 6
$\mathcal{X}_I$, 21
$\mathcal{X}_{I^*}$, 21
xIndirect, 7
X_INDIRECT_INSTR, 5
X_INSTR, 7
xLabel, 7
X_LABEL_INSTR, 5
xlada, 6, 36
xli, 5
xll, 5, 40
xlrda, 6, 36
X_NON_LABEL, 41
XOR, 45
$\mathcal{X}_P$, 42
xPa, 7
xProc, 7
xrg, 5, 37
xrgs, 5, 37
xrja, 5, 38
xrjal, 5, 38
xrjb, 5, 38
xrjbl, 5, 38
xsdr, 6, 35
xsdrh, 6, 35
xsdro, 6, 35
xsdrs, 6, 35