The DeCCo project papers V

Compiler Correctness Proofs

Susan Stepney

University of York Technical Report YCS-2002-362

June 2003

Contents

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

This document captures the various correctness proofs for the DeCCo system.

First it outlines the overall proof structure, then provides detailed proofs for the low level operator and expression constructs. In particular, the proof of unsigned 16-bit division is derived in detail.

2 Proof outline

2.1 Informal overview

Figure 1 shows the various components in the DeCCo system. The starting point is a sequence of Pasp modules, that are each statically checked for correctness.

A sequence of statically checked Pasp modules can be flattened into a Pasp program (undergoing further cross-module static checks). Alternatively, they can each be compiled into an XAspAL module, and the corresponding sequence of XAspAL modules linked into an AspAL program. The essence of the correctness proof is to show that the Pasp program and the corresponding AspAL program derived from the same sequence of Pasp modules, have the same dynamic semantics. (An intermediate 'equivalent AspAL program' is used to simplify the proof. It has the same semantics as the linked AspAL program, but a structure more similar to the flattened Pasp program.)

In addition, an AspAL program is not executable, because it still has symbolic labels and data addresses. A further processing phases, hexing, takes an AspAL program to an Asp program, with numeric labels and absolute addressing.

Previous documents have dealt with the proof of the various language elements, and with the operators, in some detail. A previous documewnt also sketched the linker proof. This document covers the linker proof in more detail.

2.2 Simplified proof obligation

We want to say that the flattened Pasp and linked AspAL programs 'do the same thing'. Let's assume for now that AspAL and Pasp values are the same, and AspAL addresses are the same as Pasp locations. That is, let's assume the Pasp and AspAL states are identical.

So we would simply require that all programs that have a Pasp dynamic meaning compile to an Asp program with the same dynamic meaning (same

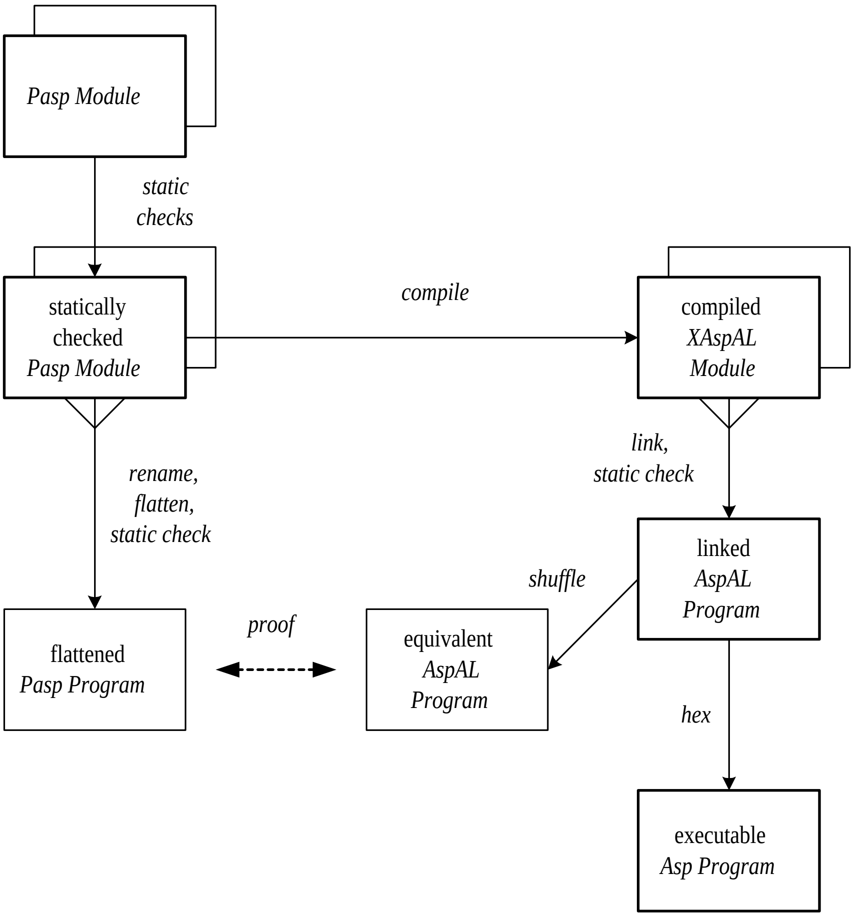

Figure 1: DeCCo system components

state transition relation, or same mapping from input streams to outputs streams):

$$\begin{array}{l} \mathit{Prog} \mid \theta \mathit{Prog} \in \operatorname{dom} \mathcal{M}_P \\ \vdash \\ \mathcal{M}_P\ \theta \mathit{Prog} = \mathcal{X}_P(\mathcal{O}_P\ \theta \mathit{Prog}) \end{array}$$

We actually have a sequence of modules, not a program, and an operational (compiler) semantics for modules. These modules are flattened to make a Pasp program, or compiled then linked to make an AspAL program. We use this to *define* the Asp program that corresponds to a Pasp program.

$$\begin{array}{|l} \forall M : \operatorname{seq} \mathit{Module};\ \mathit{Prog} \mid \theta \mathit{Prog} = \mathit{flatten}\ M \bullet \\ \qquad \mathcal{O}_P\ \theta \mathit{Prog} = \mathit{link}(\mathcal{O}_M \circ M) \end{array}$$

Combining these we have our (simplified) proof obligation

$$\begin{array}{l} M : \operatorname{seq} \mathit{Module};\ \mathit{Prog} \mid \\ \qquad \theta \mathit{Prog} \in \operatorname{dom} \mathcal{M}_P \\ \qquad \wedge\ \theta \mathit{Prog} = \mathit{flatten}\ M \\ \vdash \\ \mathcal{M}_P\ \theta \mathit{Prog} = \mathcal{X}_P(\mathit{link}(\mathcal{O}_M \circ M)) \end{array}$$

In reality, life is a little more complicated. The Pasp meaning is expresed in terms of Pasp locations and Pasp values, whilst the AspAL meaning is expressed in terms of AspAL addresses and AspAL byte values, and these are different. (Also, some of the functions such as *link* have further arguments; and there are two kinds of modules.)

Because Pasp and Asp have different representations of their states, we need a way of relating these concepts, using a 'retrieve relation'. This specifies the relationship that we claim holds between Pasp and AspAL states. That is, the retreive specifies the way we chose to interpret AspAL bytes as Pasp unsigneds, etc. (This is not an entirely free choice: were we to make a different choice, say swapping the order of high and low bytes, we would have to change the compiler specification to respect this choice.)

We do not go into details of the particualr retrieve relations used here. The interested reader is referred to [Stepney 1993] for more details of how to construct the relevant retreive relations and the full proof obligations. (The presentation style of the proofs in this technical report are rather different from the style in that book.)

2.3 Induction

The proof proceeds by structural induction, over the structure of the Pasp langauge.

We use several *induction hypotheses* in the proof; we ensure that these are re-established by each operation.

- Pasp and translation environments correspond initially
- Pasp and Asp states correspond initially
- Pasp expression values are on the Asp stack

2.4 Correctness of Linking

The first step in the proof is to expand the expression for *aspProg*, involving the compilation functions for modules, and the linking function.

The aim is to massage the form of the linked program into one with the same semantics, but with a structure that mirrors the flattened Pasp program structure.

The linked modules have blocks of code corresponding to the simple declarations, then routine declarations, and then simple declaration initialisations, for each module. The main module also has some body code, at the end.

The flattened Pasp program has essentially all the simple declaratins first, then all the routine declarations, then all the simple declaration initialisations, then the body.

The idea is to shuffle the blocks in the linked program (either in reality, or just conceptually) into the same order as the Pasp program, without changing the meaning of the linked program.

Then we can ignore the module structure, and consider just the program structure of the Pasp and corresponding AspAL programs.

3 Correctness of utility fragments

3.1 Notation

The notation in the right hand column illustrates the state of memory, as

$$[A, B]_{AB}[ac, bc]_{ab}[C]_C[\_, t0|_{h\delta}, t1, \ldots, \boxed{t2}, t3, \ldots, t4|_{\delta}, t5, \_]$$

where the subscripts show which registers are being described (omitted if clear from context). The box around $t2$ indicates that the data address register D is pointing at this memory location, which contains the value $t2$. The subscript $h\delta$ on $t0$ indicates the value of δ (as offset from the top of the heap). The subscript δ on $t4$ indicates the value of δ (as offset from the top of the stack).

If two bytes are stored at δ, the lo byte would be $t4$, and the hi byte $t5$. If two bytes are store in AB, the lo byte is in B, the hi byte in A. An underscore indicates the particular contents of that memory location is irrelevant.

The A and B registers hold bytes. Any arithmetic expression occurring in such a regiser, for example $2 * b$, is assumed to be truncated to a byte; that is, the actual value of any such expression e is the byte $e \bmod FF$.

The correctness proof calculates the change in memory for each instruction, starting from the induction hypothesis state, and shows that it establishes the correct final result.

3.2 Correctness of store byte or word

We are required to show that the value of the the byte or word in the AB registers is stored at the memory location pointed to by D.

The code is $\mathcal{O}_{s\tau}$, which has two cases, depending on the size of the type:

1. the value b fits in a byte ($\mathit{sizeof}\ \tau = 1$),
2. the value u fits in a word ($\mathit{sizeof}\ \tau \neq 1$).

Case 1

$$\begin{array}{ll} & [\_, b]_{AB}[\ldots, \boxed{\_}, \_, \ldots] \\ \texttt{bstd} & [\_, b]_{AB}[\ldots, \boxed{b}, \_, \ldots] \end{array}$$

Case 2

$$\begin{array}{ll} & [u_{hi}, u_{lo}]_{AB}[\ldots, \boxed{\_}, \_, \ldots] \\ \texttt{abstd} & [u_{hi}, u_{lo}]_{AB}[\ldots, \boxed{u_{lo}}, u_{hi}, \ldots] \end{array}$$

□3.2

3.3 Correctness of load byte or word

We are required to show that the value of the the byte or word at D is loaded into the AB registers.

The code is $\mathcal{O}_{l\tau}$, which has two cases, depending on the size of the type:

1. the value b fits in a byte ($\mathit{sizeof}\ \tau = 1$),
2. the value u fits in a word ($\mathit{sizeof}\ \tau \neq 1$).

Case 1

$$\begin{array}{ll} & [\_, \_]_{AB}[\ldots, \boxed{b}, \_, \ldots] \\ \texttt{bldd} & [\_, b]_{AB}[\ldots, \boxed{b}, \_, \ldots] \end{array}$$

Case 2

$$\begin{array}{ll} & [\_, \_]_{AB}[\ldots, \boxed{u_{lo}}, u_{hi}, \ldots] \\ \texttt{abldd} & [u_{hi}, u_{lo}]_{AB}[\ldots, \boxed{u_{lo}}, u_{hi}, \ldots] \end{array}$$

□3.3

3.4 Correctness of load immediate word

We are required to show that the value of the the immediate word u is loaded into the AB registers.

The code is $\mathcal{O}_{lwi}$

Case 2

$$\begin{array}{ll} & [\_,\_]_{AB} \\ \texttt{aldi}\ \mathit{theHiByte}\ u & [u_{hi},\_]_{AB} \\ \texttt{bldi}\ \mathit{theLoByte}\ u & [u_{hi},u_{lo}]_{AB} \end{array}$$

□3.4

3.5 Correctness of load immediate byte or word

We are required to show that the value of the the immediate byte or word is loaded into the AB registers.

The code is $\mathcal{O}_{l\tau i}$, which has two cases, depending on the size of the type:

1. the immediate value b fits in a byte ($\mathit{sizeof}\ \tau = 1$),
2. the immediate value u fits in a word ($\mathit{sizeof}\ \tau \neq 1$).

Case 1

$$\begin{array}{ll} & [\_,\_]_{AB} \\ \texttt{bldi}\ b & [\_,b]_{AB} \end{array}$$

Case 2

$$\begin{array}{ll} & [\_,\_]_{AB} \\ \mathcal{O}_{lwi}\ u & 3.4 : [u_{hi},u_{lo}]_{AB} \end{array}$$

□3.5

4 Correctness of Operators

4.1 Unary Operator proof obligation

Induction hypothesis: initially, the argument is in the AB registers. (An unsigned argument has its lo byte in B and its hi byte in A; a boolean, byte or enumerated argument occupies B only.)

Obligation: the operator leaves the correct result in the AB registers. (Again, if the result fits in a single byte, it is stored in the B register, and the contents of the A register are ignored.)

4.2 Correctness of boolean not

The boolean value b is in the B register.

We are required to show that the final contents of B are the boolean value $\neg\, b$.

$$\begin{array}{ll} & [\_, b]_{AB} \\ \texttt{bxoi}\ 1 & [\_, b\ xor\ 1]_{AB} \end{array}$$

□ 4.2

4.3 Correctness of byte not

The byte value b is in the B register

We are required to show that the final contents of B are the byte value $\neg\, b$.

$$\begin{array}{ll} & [\_, b]_{AB} \\ \texttt{bxoi}\ \mathit{MaxByte} & [\_, b\ xor\ \mathit{FF}]_{AB} \end{array}$$

□ 4.3

4.4 Correctness of byte shift left

The byte value b is in the B register

We are required to show that the final contents of B are the byte value $2 * b$ (truncated to fit if necessary).

$$\begin{array}{ll}
 & [\_, b]_{AB}[\_, \_]_{ab} \\
\texttt{ccb} & [\_, b]_{AB}[\_, 0]_{ab} \\
\texttt{lrb} & [\_, 2 * b + 0]_{AB}
\end{array}$$

□ 4.4

4.5 Correctness of byte shift right

The byte value b is in the B register.

We are required to show that the final contents of B are the byte value $b \, \mathrm{div} \, 2$.

$$\begin{array}{ll}
 & [\_, b]_{AB}[\_, \_]_{ab} \\
\texttt{ccb} & [\_, b]_{AB}[\_, 0]_{ab} \\
\texttt{rrb} & [\_, 0 + b \ \mathrm{div}\ 2]_{AB}
\end{array}$$

□ 4.5

4.6 Correctness of cast byte to boolean

The byte value b is in the B register.

We are required to show that the final contents of B are the boolean value *true*, that is 0, if b is zero, or the boolean value *false*, that is 1, for any other value of b.

	$[\_, b]_{AB}[\_, \_]_{ab}$
`beqi` 0	$[\_, b]_{AB}[\_, b = 0]_{ab}$
`ncb`	$[\_, b]_{AB}[\_, b \neq 0]_{ab}$
`bldi` 0	$[\_, 0]_{AB}[\_, b \neq 0]_{ab}$
`lrb`	$[\_, b \neq 0]_{AB}$
	(expand) $[\_, \textbf{if } b \neq 0 \textbf{ then } 1 \textbf{ else } 0]_{AB}$

□ 4.6

4.7 Correctness of cast boolean to byte

The boolean value b is in the B register.

We are required to show that the final contents of B are the byte value 0 if b is *true*, that is 0, or the byte value 1 if b is *false*, that is 1.

$[\_, b]_{AB}$

Do nothing: the boolean is 0 or 1, so can be directly interpreted as the byte.

□ 4.7

4.8 Correctness of cast unsigned to byte

The unsigned value u is in the AB registers.

We are required to show that the final contents of B are the byte value corresponding to the value of u if u is 255 or less; otherwise any result is allowed.

$[u_{hi}, u_{lo}]_{AB}$

Do nothing. The high byte is effectively truncated, because the result is assumed to be a byte, and hence in only the B register.

□ 4.8

4.9 Correctness of cast enum to byte

The enumerated value b is in the B register.

We are required to show that the final contents of B are the byte value equal to the enumerated value.

$$[\_, b]_{AB}$$

Do nothing: an enum is stored as a byte.

□ 4.9

4.10 Correctness of increment enum

The enumerated value b is in the B register.

We are required to show that the final contents of B are the enumerated value incremented by one, unless the enum has its highest allowed value, in which case any result is allowed.

$$\begin{array}{ll}
 & [\_, b]_{AB}[\_, \_]_{ab} \\
\texttt{ccb} & [\_, b]_{AB}[\_, 0]_{ab} \\
\texttt{buai}\ 1 & [\_, b + 1 + 0]_{AB}
\end{array}$$

□ 4.10

4.11 Correctness of decrement enum

The enumerated value b is in the B register.

We are required to show that the final contents of B are the enumerated value decremented by one, unless the enum is zero, in which case any result is allowed.

$$\begin{array}{ll}
 & [\_, b]_{AB}[\_, \_]_{ab} \\
\texttt{ccb} & [\_, b]_{AB}[\_, 0]_{ab} \\
\texttt{busi}\ 1 & [\_, b - 1 + 0]_{AB}
\end{array}$$

□ 4.11

4.12 Correctness of unsigned not

The unsigned value u is in the AB registers.

We are required to show that the final contents of AB are the unsigned value $\neg\ u$.

	$[u_{hi}, u_{lo}]_{AB}$
`axoi` *MaxByte*	$[u_{hi}\ xor\ FF, u_{lo}]_{AB}$
`bxoi` *MaxByte*	$[u_{hi}\ xor\ FF, u_{lo}\ xor\ FF]_{AB}$ $(simplify)[u\ xor\ FFFF]_{AB}$

□ 4.12

4.13 Correctness of unsigned shift left

The unsigned value u is in the AB registers.

We are required to show that the final contents of AB are the unsigned value $2 * u$ (truncated to fit if necessary).

		$[u_{hi}, u_{lo}]_{AB}$
	`ccb`	$[u_{hi}, u_{lo}]_{AB}[\_, 0]_{ab}$
	`lrb`	$[u_{hi}, 2 * u_{lo} + 0]_{AB}[\_, u_{lo} \text{ div } 128]_{ab}$
	`sca`	$[u_{hi}, 2 * u_{lo}]_{AB}[1, u_{lo} \text{ div } 128]_{ab}$
	xrjb l	if $cb = 1$, jump to l, preserving AB $[u_{hi}, 2 * u_{lo}]_{AB}[1, u_{lo} \text{ div } 128 = 0]_{ab}$
	`cca`	$[u_{hi}, 2 * u_{lo}]_{AB}[0, u_{lo} \text{ div } 128 = 0]_{ab}$
l		(from above) $[u_{hi}, 2 * u_{lo}]_{AB}[0, u_{lo} \text{ div } 128 = 0]_{ab}$ (from goto l) $[u_{hi}, 2 * u_{lo}]_{AB}[1, u_{lo} \text{ div } 128 = 1]_{ab}$ (combining) $[u_{hi}, 2 * u_{lo}]_{AB}[u_{lo} \text{ div } 128, \_]_{ab}$
	`lra`	$[2 * u_{hi} + u_{lo} \text{ div } 128, 2 * u_{lo}]_{AB}$ $(simplify)[2 * u]_{AB}$

□ 4.13

4.14 Correctness of unsigned shift right

The unsigned value u is in the AB registers.

We are required to show that the final contents of AB are the unsigned value u div 2.

		$[u_{hi}, u_{lo}]_{AB}$
	`cca`	$[u_{hi}, u_{lo}]_{AB}[0, \_]_{ab}$
	`rra`	$[0 + u_{hi} \text{ div } 2, u_{lo}]_{AB}[u_{hi} \bmod 2, \_]_{ab}$
	`scb`	$[u_{hi} \text{ div } 2, u_{lo}]_{AB}[u_{hi} \bmod 2, 1]_{ab}$
	xrja l	if $ca = 1$, jump to l, preserving AB $[u_{hi} \text{ div } 2, u_{lo}]_{AB}[u_{hi} \bmod 2 = 0, 1]_{ab}$
	`ccb`	$[u_{hi} \text{ div } 2, u_{lo}]_{AB}[u_{hi} \bmod 2 = 0, 0]_{ab}$
l		(from above) $[u_{hi} \text{ div } 2, u_{lo}]_{AB}[u_{hi} \bmod 2 = 0, 0]_{ab}$ (from goto l) $[u_{hi} \text{ div } 2, u_{lo}]_{AB}[u_{hi} \bmod 2 = 1, 1]_{ab}$ (combining) $[u_{hi} \text{ div } 2, u_{lo}]_{AB}[\_, u_{hi} \bmod 2]_{ab}$
	`rrb`	$[u_{hi} \text{ div } 2, 128 * (u_{hi} \bmod 2) + u_{lo} \text{ div } 2]_{AB}$ $(\textit{simplify})[u \text{ div } 2]_{AB}$

□ 4.14

4.15 Correctness of cast byte to unsigned

The byte value b is in B, with garbage in A.

We are required to show that the final contents of AB are the unsigned value $u(= b)$.

	$[\_, b]_{AB}$
`aldi` 0	$[0, b]_{AB}$

□ 4.15

4.16 Correctness of cast unsigned to high byte

The unsigned value u is in the AB registers.

We are required to show that the final contents of B are the byte value u_{hi}.

$$\begin{array}{ll} & [u_{hi}, u_{lo}]_{AB} \\ \texttt{blda} & [u_{hi}, u_{hi}]_{AB} \end{array}$$

□ 4.16

4.17 Correctness of cast unsigned to low byte

The unsigned value u is in the AB registers.

We are required to show that the final contents of B are the byte value u_{lo}.

$$[u_{hi}, u_{lo}]_{AB}$$

□ 4.17

4.18 Binary Operator proof obligation

The general operation is $u \Omega v$.

Induction hypothesis: initially, the left argument u is on the expression evaluation stack with the D register pointing to δ location (with lo byte at δ, hi byte at $\delta - 1$, remembering that δ increases ‘downwards’); the right argument v is in the AB registers (an unsigned argument has its lo byte in B and its hi byte in A; a boolean, byte or enumerated argument occupies B only).

Obligation: The operator leaves the correct result in the AB registers. (Again, if the result fits in a single byte, it is stored in the B register, and the contents of the A register are ignored.)

4.19 Correctness of boolean and

The left argument boolean value a is at δ, pointed to by D. The right argument boolean value b is in B.

We are required to show that the final contents of B are the boolean value $a\ AND\ b$.

$$\begin{array}{ll} & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\ \texttt{band} & [\_, b\ AND\ a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \end{array}$$

□ 4.19

4.20 Correctness of boolean or

The left argument boolean value a is at δ, pointed to by D. The right argument boolean value b is in B.

We are required to show that the final contents of B are the boolean value $a\ OR\ b$.

$$\begin{array}{ll} & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\ \texttt{bord} & [\_, b\ OR\ a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \end{array}$$

□ 4.20

4.21 Correctness of byte and

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a\ AND\ b$.

$$\begin{array}{ll} & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\ \texttt{band} & [\_, b\ AND\ a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \end{array}$$

□ 4.21

4.22 Correctness of byte or

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a\ OR\ b$.

$$\begin{array}{ll} & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\ \texttt{bord} & [\_, b\ OR\ a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \end{array}$$

□ 4.22

4.23 Correctness of byte xor

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a\ XOR\ b$.

$$\begin{array}{ll} & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\ \texttt{bxod} & [\_, b\ XOR\ a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \end{array}$$

□ 4.23

4.24 Correctness of unsigned and

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of AB are the unsigned value *u AND v*.

$$\begin{array}{ll}
 & [v_{hi}, v_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots] \\
\texttt{band} & [v_{hi}, v_{lo}\ AND\ u_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots] \\
\texttt{dpi}\ 1 & [v_{hi}, v_{lo}\ AND\ u_{lo}]_{AB}[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots] \\
\texttt{aand} & [v_{hi}\ AND\ u_{hi}, v_{lo}\ AND\ u_{lo}]_{AB}[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots] \\
 & (\mathit{simplify})[v\ AND\ u]_{AB}[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots]
\end{array}$$

□ 4.24

4.25 Correctness of unsigned or

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of AB are the unsigned value *u OR v*.

$$\begin{array}{ll}
 & [v_{hi}, v_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots] \\
\texttt{bord} & [v_{hi}, v_{lo}\ OR\ u_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots] \\
\texttt{dpi}\ 1 & [v_{hi}, v_{lo}\ OR\ u_{lo}]_{AB}[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots] \\
\texttt{aord} & [v_{hi}\ OR\ u_{hi}, v_{lo}\ OR\ u_{lo}]_{AB}[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots] \\
 & (\mathit{simplify})[v\ OR\ u]_{AB}[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots]
\end{array}$$

□ 4.25

4.26 Correctness of unsigned xor

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of AB are the unsigned value *u XOR v*.

$[v_{hi}, v_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_\delta, u_{hi}, \ldots]$

`bxod` $[v_{hi}, v_{lo}\ XOR\ u_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_\delta, u_{hi}, \ldots]$

`dpi` 1 $[v_{hi}, v_{lo}\ XOR\ u_{lo}]_{AB}[\ldots, u_{lo}|_\delta, \boxed{u_{hi}}, \ldots]$

`axod` $[v_{hi}\ XOR\ u_{hi}, v_{lo}\ XOR\ u_{lo}]_{AB}[\ldots, u_{lo}|_\delta, \boxed{u_{hi}}, \ldots]$
$(simplify)[v\ XOR\ u]_{AB}[\ldots, u_{lo}|_\delta, \boxed{u_{hi}}, \ldots]$

□ 4.26

4.27 Correctness of unsigned equals

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of B are the boolean value $(u = v)$.

$[v_{hi}, v_{lo}][\_, \_][\boxed{u_{lo}}|_\delta, u_{hi}]$

`beqd` $[v_{hi}, v_{lo}][\_, v_{lo} = u_{lo}][\boxed{u_{lo}}|_\delta, u_{hi}]$

`bldi` 0 $[v_{hi}, 0][\_, v_{lo} = u_{lo}][\boxed{u_{lo}}|_\delta, u_{hi}]$

`ncb` $[v_{hi}, 0][\_, v_{lo} \neq u_{lo}][\boxed{u_{lo}}|_\delta, u_{hi}]$

xrjb l **if** $v_{lo} \neq u_{lo}$ **then** *goto l*
$[v_{hi}, 0][\_, 0][\boxed{u_{lo}}|_\delta(= v_{lo}), u_{hi}]$

`dpi` 1 $[v_{hi}, 0][\_, 0][u_{lo}|_\delta(= v_{lo}), \boxed{u_{hi}}]$

`aeqd` $[v_{hi}, 0][v_{hi} = u_{hi}, 0][u_{lo}|_\delta(= v_{lo}), \boxed{u_{hi}}]$

`cbldca` $[v_{hi}, 0][v_{hi} = u_{hi}, v_{hi} = u_{hi}][u_{lo}|_\delta(= v_{lo}), \boxed{u_{hi}}]$

`lrb` $[v_{hi}, v_{hi} = u_{hi}][v_{hi} = u_{hi}, 0][u_{lo}|_\delta(= v_{lo}), \boxed{u_{hi}}]$

l

(from *goto* l): $[v_{hi}, 0][\_, 1][\boxed{u_{lo}}|_\delta(\neq v_{lo}), u_{hi}]$
$\Rightarrow [\_, v = u][\_, \_][u_{lo}|_\delta, u_{hi}]$

□ 4.27

4.28 Correctness of unsigned greater than

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of B are the boolean value $(u > v)$.

		$[v_{hi}, v_{lo}][\_,\_][\boxed{u_{lo}}_{\delta}, u_{hi}]$
	`scb`	$[v_{hi}, v_{lo}][\_, 1][\boxed{u_{lo}}, u_{hi}]$
	`bucd`	$[v_{hi}, v_{lo}][\_, v_{lo} \geq u_{lo}][\boxed{u_{lo}}, u_{hi}]$
	`bldi` 0	$[v_{hi}, 0][\_, v_{lo} \geq u_{lo}][\boxed{u_{lo}}, u_{hi}]$
	`dpi` 1	$[v_{hi}, 0][\_, v_{lo} \geq u_{lo}][u_{lo}, \boxed{u_{hi}}]$
	`ncb`	$[v_{hi}, 0][\_, v_{lo} < u_{lo}][u_{lo}, \boxed{u_{hi}}]$
	xrjb l	**if** $v_{lo} < u_{lo}$ **then** *goto l* $[v_{hi}, 0][\_, 0][(v_{lo} \geq) u_{lo}, \boxed{u_{hi}}]$
	`aeqd`	$[v_{hi}, 0][v_{hi} = u_{hi}, 0][(v_{lo} \geq) u_{lo}, \boxed{u_{hi}}]$
	xrja$(l+1)$	**if** $v_{hi} = u_{hi}$ **then** *goto*$(l+1)$ $[v_{hi}, 0][0, 0][(v_{lo} \geq) u_{lo}, (v_{hi} \neq) \boxed{u_{hi}}]$
l		
		(from *gotol*): $[v_{hi}, 0][\_, 1][(v_{lo} <) u_{lo}, \boxed{u_{hi}}]$ $\Rightarrow [v_{hi}, 0][\_,\_][u_{lo}, \boxed{u_{hi}}]$ with $v_{lo} < u_{lo} \vee v_{lo} \geq u_{lo} \wedge v_{hi} \neq u_{hi}$
	`cca`	$[v_{hi}, 0][0, \_][u_{lo}, \boxed{u_{hi}}]$ with $v_{lo} < u_{lo} \vee v_{lo} \geq u_{lo} \wedge v_{hi} \neq u_{hi}$
	`aucd`	$[v_{hi}, 0][v_{hi} > u_{hi}, \_][u_{lo}, \boxed{u_{hi}}]$ with $v_{lo} < u_{lo} \vee v_{lo} \geq u_{lo} \wedge v_{hi} \neq u_{hi}$
	xrja$(l+1)$	**if** $v_{hi} > u_{hi}$ **then** *goto*$(l+1)$ $[v_{hi}, 0][0, \_][u_{lo}, \boxed{u_{hi}}]$ with $v_{lo} < u_{lo} \wedge v_{hi} \leq u_{hi} \vee v_{lo} \geq u_{lo} \wedge v_{hi} < u_{hi}$ $= v < u$
	`bldi` 1	$[v_{hi}, 1][0, \_][u_{lo}, \boxed{u_{hi}}]$ with $v < u$
$l+1$		
		(from first *goto*$(l+1)$): $[v_{hi}, 0][1, 0][(v_{lo} \geq) u_{lo}, (v_{hi} =) \boxed{u_{hi}}]$ (from second *goto*$(l+1)$): $[v_{hi}, 0][1, \_][u_{lo}, \boxed{u_{hi}}]$ with $v_{hi} > u_{hi}$ $\Rightarrow [v_{hi}, v < u][\_,\_][u_{lo}, \boxed{u_{hi}}]$

□ 4.28

4.29 Correctness of unsigned greater than or equal to

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of B are the boolean value $(u \geq v)$.

$[v_{hi}, v_{lo}][\_,\_][\boxed{u_{lo}}_{\delta}, u_{hi}]$

`ccb` $[v_{hi}, v_{lo}][\_, 0][\boxed{u_{lo}}, u_{hi}]$

`bucd` $[v_{hi}, v_{lo}][\_, v_{lo} > u_{lo}][\boxed{u_{lo}}, u_{hi}]$

`bldi` 0 $[v_{hi}, 0][\_, v_{lo} > u_{lo}][\boxed{u_{lo}}, u_{hi}]$

`dpi` 1 $[v_{hi}, 0][\_, v_{lo} > u_{lo}][u_{lo}, \boxed{u_{hi}}]$

`ncb` $[v_{hi}, 0][\_, v_{lo} \leq u_{lo}][u_{lo}, \boxed{u_{hi}}]$

xrjb l **if** $v_{lo} \leq u_{lo}$ **then** *goto l*
$[v_{hi}, 0][\_, 0][(v_{lo} >)u_{lo}, \boxed{u_{hi}}]$

`aeqd` $[v_{hi}, 0][v_{hi} = u_{hi}, 0][(v_{lo} >)u_{lo}, \boxed{u_{hi}}]$

$xrja(l+1)$ **if** $v_{hi} = u_{hi}$ **then** $goto(l+1)$
$[v_{hi}, 0][0, 0][(v_{lo} >)u_{lo}, (v_{hi} \neq)\boxed{u_{hi}}]$

l

(from *goto* l): $[v_{hi}, 0][\_, 1][(v_{lo} \leq)u_{lo}, \boxed{u_{hi}}]$
$\Rightarrow [v_{hi}, 0][\_,\_][u_{lo}, \boxed{u_{hi}}]$
with $v_{lo} \leq u_{lo} \vee v_{lo} > u_{lo} \wedge v_{hi} \neq u_{hi}$

`cca` $[v_{hi}, 0][0, \_][u_{lo}, \boxed{u_{hi}}]$
with $v_{lo} \leq u_{lo} \vee v_{lo} > u_{lo} \wedge v_{hi} \neq u_{hi}$

`aucd` $[v_{hi}, 0][v_{hi} > u_{hi}, \_][u_{lo}, \boxed{u_{hi}}]$
with $v_{lo} \leq u_{lo} \vee v_{lo} > u_{lo} \wedge v_{hi} \neq u_{hi}$

$xrja(l+1)$ **if** $v_{hi} > u_{hi}$ **then** $goto(l+1)$
$[v_{hi}, 0][0, \_][u_{lo}, \boxed{u_{hi}}]$
with $v_{lo} \leq u_{lo} \wedge v_{hi} \leq u_{hi} \vee v_{lo} > u_{lo} \wedge v_{hi} < u_{hi}$
$= v \leq u$

`bldi` 1 $[v_{hi}, 1][0, \_][u_{lo}, \boxed{u_{hi}}]$
with $v \leq u$

$l+1$

(from first $goto(l+1)$: $[v_{hi}, 0][1, 0][(v_{lo} >)u_{lo}, (v_{hi} =)\boxed{u_{hi}}]$
(from second $goto(l+1)$: $[v_{hi}, 0][1, \_][u_{lo}, \boxed{u_{hi}}]$
with $v_{hi} > u_{hi}$
$\Rightarrow [v_{hi}, v \leq u][\_,\_][u_{lo}, \boxed{u_{hi}}]$

□ 4.28

4.30 Correctness of unsigned not equals

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of B are the boolean value $(u \neq v)$.

$$\begin{array}{rl}
 & [v_{hi}, v_{lo}][\_, \_][\boxed{u_{lo}}|_{\delta}, u_{hi}] \\
ueq & [\_, (v = u)][\_, \_][u_{lo}|_{\delta}, u_{hi}] \\
not & [\_, (v = u)\ XOR\ 1][\_, \_][u_{lo}|_{\delta}, u_{hi}] \\
 & (simplify)[\_, (v \neq u)][\_, \_][u_{lo}|_{\delta}, u_{hi}]
\end{array}$$

□ 4.30

4.31 Correctness of unsigned less than

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of B are the boolean value $(u < v)$.

$$\begin{array}{rl}
 & [v_{hi}, v_{lo}][\_, \_][\boxed{u_{lo}}|_{\delta}, u_{hi}] \\
uge & [\_, (v \leq u)][\_, \_][u_{lo}|_{\delta}, u_{hi}] \\
not & [\_, (v \leq u)\ XOR\ 1][\_, \_][u_{lo}|_{\delta}, u_{hi}] \\
 & (simplify)[\_, (v > u)][\_, \_][u_{lo}|_{\delta}, u_{hi}]
\end{array}$$

□ 4.31

4.32 Correctness of unsigned less than or equal to

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of B are the boolean value $(u \leq v)$.

$$\begin{array}{ll}
 & [v_{hi}, v_{lo}][\_, \_][\boxed{u_{lo}}|_{\delta}, u_{hi}] \\
\mathit{ugt} & [\_, (v < u)][\_, \_][u_{lo}|_{\delta}, u_{hi}] \\
\mathit{not} & [\_, (v < u)\ \mathit{XOR}\ 1][\_, \_][u_{lo}|_{\delta}, u_{hi}] \\
 & (\mathit{simplify})[\_, (v \geq u)][\_, \_][u_{lo}|_{\delta}, u_{hi}]
\end{array}$$

□ 4.32

4.33 Correctness of byte equals

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the boolean value $(a = b)$.

$$\begin{array}{ll}
 & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{beqd} & [\_, b]_{AB}[\_, b = a]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{bldi}\ 0 & [\_, 0]_{AB}[\_, b = a]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{lrb} & [\_, 0 + b = a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots]
\end{array}$$

□ 4.33

4.34 Correctness of byte less than

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the boolean value

$(a < b)$.

$$\begin{array}{ll}
 & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{ccb} & [\_, b]_{AB}[\_, 0]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{bucd} & [\_, b]_{AB}[\_, a < b]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{bldi}\ 0 & [\_, 0]_{AB}[\_, a < b]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{lrb} & [\_, 0 + a < b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots]
\end{array}$$

□ 4.34

4.35 Correctness of byte less than or equal to

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the boolean value $(a \leq b)$.

$$\begin{array}{ll}
 & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{scb} & [\_, b]_{AB}[\_, 1]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{bucd} & [\_, b]_{AB}[\_, a \leq b]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{bldi}\ 0 & [\_, 0]_{AB}[\_, a \leq b]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{lrb} & [\_, 0 + a \leq b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots]
\end{array}$$

□ 4.35

4.36 Correctness of byte not equals

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the boolean value

$(a \neq b)$.

$$\begin{array}{rl}
 & [\_, b][\_, \_][\boxed{a}|_{\delta}, \_] \\
\mathit{beq} & [\_, (b = a)][\_, \_][a|_{\delta}, \_] \\
\mathit{not} & [\_, (b = a)\ \mathit{XOR}\ 1][\_, \_][a|_{\delta}, \_] \\
 & (\mathit{simplify})[\_, (b \neq a)][\_, \_][a|_{\delta}, \_]
\end{array}$$

□ 4.36

4.37 Correctness of byte greater than

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the boolean value $(a > b)$.

$$\begin{array}{rl}
 & [\_, b][\_, \_][\boxed{a}|_{\delta}, \_] \\
\mathit{ble} & [\_, (a \leq b)][\_, \_][a|_{\delta}, \_] \\
\mathit{not} & [\_, (a \leq b)\ \mathit{XOR}\ 1][\_, \_][a|_{\delta}, \_] \\
 & (\mathit{simplify})[\_, (a > b)][\_, \_][a|_{\delta}, \_]
\end{array}$$

□ 4.37

4.38 Correctness of byte greater than or equal to

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the boolean value $(a \geq b)$.

$$\begin{array}{rl}
 & [\_, b][\_, \_][\boxed{a}|_{\delta}, \_] \\
\mathit{blt} & [\_, (a < b)][\_, \_][a|_{\delta}, \_] \\
\mathit{not} & [\_, (a < b)\ \mathit{XOR}\ 1][\_, \_][a|_{\delta}, \_] \\
 & (\mathit{simplify})[\_, (a \geq b)][\_, \_][a|_{\delta}, \_]
\end{array}$$

□ 4.38

4.39 Correctness of enumerated equals

See the proof for byte equals; enums are stored as bytes.

□ 4.39

4.40 Correctness of enumerated not equals

See the proof for byte not equals; enums are stored as bytes.

□ 4.40

4.41 Correctness of byte addition

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a + b$.

$$\begin{array}{ll} & [\_, b]_{AB}[\dots, \boxed{a}|_{\delta}, \_, \dots] \\ \texttt{ccb} & [\_, b]_{AB}[\_, 0]_{ab}[\dots, \boxed{a}|_{\delta}, \_, \dots] \\ \texttt{buad} & [\_, b + a + 0]_{AB}[\_, (b + a + 0) \text{ div } 256]_{ab}[\dots, \boxed{a}|_{\delta}, \_, \dots] \\ \texttt{hltcb} & [\_, b + a]_{AB}[\_, 0][\dots, \boxed{a}|_{\delta}, \_, \dots] \end{array}$$

The last instruction makes the processor halt on overflow, so if it does not halt, the b carry register contains zero.

□ 4.41

4.42 Correctness of byte subtraction

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a - b$.

Most of the complication cmes from having b in the B register initially; the Asp instruction would calculate $b - a$, so the arguments are first swapped, using a temporary location at *topop* (subscript $t0$).

$$\begin{array}{ll}
 & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots, \_|_{t0}] \\
xsdro\ 0 & [\_, b]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{\_}|_{t0}] \\
\texttt{bstd} & [\_, b]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{b}|_{t0}] \\
xsdrs\ \delta & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots, b|_{t0}] \\
\texttt{bldd} & [\_, a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots, b|_{t0}] \\
xsdro\ 0 & [\_, a]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{b}|_{t0}] \\
\texttt{ccb} & [\_, a]_{AB}[\_, 0]_{ab}[\ldots, a|_{\delta}, \_, \ldots, \boxed{b}|_{t0}] \\
\texttt{busd} & \\
 & a - b \geq 0 \Rightarrow [\_, a - b]_{AB}[\_, 0]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
 & a - b < 0 \Rightarrow [\_, a - b + 256]_{AB}[\_, 1]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{hltcb} & [\_, a - b]_{AB}[\_, 0][\ldots, \boxed{a}|_{\delta}, \_, \ldots]
\end{array}$$

The last instruction makes the processor halt on underflow, so if it does not halt, the b carry register contains zero.

□ 4.42

4.43 Correctness of byte multiplication

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a * b$.

$$\begin{array}{ll}
 & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{aldi}\ 0 & [0, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{aumd} & [(b * a)_{hi}, (b * a)_{lo}]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{aeqi}\ 0 & [(b * a)_{hi}, (b * a)_{lo}]_{AB}[(b * a)_{hi} = 0, \_]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{nca} & [(b * a)_{hi}, (b * a)_{lo}]_{AB}[(b * a)_{hi} \neq 0, \_]_{ab}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{hltca} & [0, b * a]_{AB}[0, \_][\ldots, \boxed{a}|_{\delta}, \_, \ldots]
\end{array}$$

The last instruction makes the processor halt on overflow, so if it does not halt, the a carry register contains zero, and the A register is zero.

□ 4.43

4.44 Correctness of byte division

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a \operatorname{div} b$.

Most of the complication cmes from having b in the B register initially; the Asp instruction would calculate b div a, so the arguments are first swapped.

$$
\begin{array}{ll}
 & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots, \_|_{t0}] \\
\mathit{xsdro}\ 0 & [\_, b]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{\_}|_{t0}] \\
\texttt{bstd} & [\_, b]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{b}|_{t0}] \\
\mathit{xsdrs}\ \delta & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots, b|_{t0}] \\
\texttt{bldd} & [\_, a]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots, b|_{t0}] \\
\mathit{xsdro}\ 0 & [\_, a]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{b}|_{t0}] \\
\texttt{aldi}\ 0 & [0, a]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{b}|_{t0}] \\
\texttt{audd} & [a \bmod b, a \operatorname{div} b]_{AB}[\ldots, a|_{\delta}, \_, \ldots, \boxed{b}|_{t0}]
\end{array}
$$

The last instruction will make the processor halt if $b = 0$. The high byte of the result is effectively truncated, because the result is assumed to be a byte, and hence in only the B register.

□ 4.44

4.45 Correctness of byte modulus

The left argument byte value a is at δ, pointed to by D. The right argument byte value b is in B.

We are required to show that the final contents of B are the byte value $a \bmod b$.

$[\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots]$

Idiv_{δ} $[a \bmod b, a \operatorname{div} b]_{AB}[\ldots, a|_{\delta}, \_, \ldots]$

`blda` $[\_, a \bmod b)]_{AB}[\ldots, a|_{\delta}, \_, \ldots]$

□ 4.45

4.46 Correctness of unsigned addition

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of AB are the unsigned value $u + v$.

$[v_{hi}, v_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots]$

`ccb` $[v_{hi}, v_{lo}]_{AB}[\_, 0]_{ab}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots]$

`buad` $[v_{hi}, v_{lo} + u_{lo} + 0]_{AB}[\_, (v_{lo} + u_{lo} + 0) \operatorname{div} 256]_{ab}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots]$

`caldcb` $[v_{hi}, v_{lo} + u_{lo}]_{AB}[(v_{lo} + u_{lo}) \operatorname{div} 256, (v_{lo} + u_{lo}) \operatorname{div} 256]_{ab}[\ldots, \boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots]$

`dpi` 1 $[v_{hi}, v_{lo} + u_{lo}]_{AB}[(v_{lo} + u_{lo}) \operatorname{div} 256, (v_{lo} + u_{lo}) \operatorname{div} 256]_{ab}[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots]$

`auad` $[v_{hi} + u_{hi} + (v_{lo} + u_{lo} + 0) \operatorname{div} 256, v_{lo} + u_{lo}]_{AB}$
$[(v_{hi} + u_{hi} + (v_{lo} + u_{lo}) \operatorname{div} 256) \operatorname{div} 256, (v_{lo} + u_{lo}) \operatorname{div} 256]_{ab}$
$[\ldots, u_{lo}|_{\delta}, \boxed{u_{hi}}, \ldots]$

`hltca` $[v + u]_{AB}[0, \_][\ldots, u_{lo}|_{\delta}, u_{hi}, \ldots]$

The last instruction makes the processor halt on overflow, so if it does not halt, the a carry register contains zero.

□ 4.46

4.47 Correctness of unsigned subtraction

The left argument unsigned value u is at $\delta, \delta - 1$, with the weaker than usual assumption, that D is initially unknown. The right argument unsigned value

v is in AB.

We are required to show that the final contents of AB are the unsigned value $u - v$.

$$\begin{array}{ll}
 & [v_{hi}, v_{lo}][\_, \_][u_{lo}|_{\delta}, u_{hi}, \ldots, \_|_{t0}, \_] \\
\textit{xsdro}\ 0 & [v_{hi}, v_{lo}][\_, \_][u_{lo}|_{\delta}, u_{hi}, \ldots, \boxed{\_}|_{t0}, \_] \\
\texttt{abstd} & [v_{hi}, v_{lo}][\_, \_][u_{lo}|_{\delta}, u_{hi}, \ldots, \boxed{v_{lo}}|_{t0}, v_{hi}] \\
\textit{xsdrs}\ \delta & [v_{hi}, v_{lo}][\_, \_][\boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots, v_{lo}|_{t0}, v_{hi}] \\
\texttt{abldd} & [u_{hi}, u_{lo}][\_, \_][\boxed{u_{lo}}|_{\delta}, u_{hi}, \ldots, v_{lo}|_{t0}, v_{hi}] \\
\textit{xsrdo}\ 0 & [u_{hi}, u_{lo}][\_, \_][u_{lo}|_{\delta}, u_{hi}, \ldots, \boxed{v_{lo}}|_{t0}, v_{hi}] \\
\texttt{ccb} & [u_{hi}, u_{lo}][\_, 0][u_{lo}|_{\delta}, u_{hi}, \ldots, \boxed{v_{lo}}|_{t0}, v_{hi}] \\
\texttt{busd} & [u_{hi}, (256 + u_{lo} - v_{lo}) \bmod 256][\_, \textbf{if } u_{lo} - v_{lo} < 0 \textbf{ then } 1 \textbf{ else } 0] \\
 & \quad [u_{lo}|_{\delta}, u_{hi}, \ldots, \boxed{v_{lo}}|_{t0}, v_{hi}] \\
\texttt{caldcb} & [u_{hi}, (256 + u_{lo} - v_{lo}) \bmod 256][\textbf{if } u_{lo} - v_{lo} < 0 \textbf{ then } 1 \textbf{ else } 0, \_] \\
 & \quad [u_{lo}|_{\delta}, u_{hi}, \ldots, \boxed{v_{lo}}|_{t0}, v_{hi}] \\
\texttt{dpi}\ \ 1 & [u_{hi}, (256 + u_{lo} - v_{lo}) \bmod 256][\textbf{if } u_{lo} - v_{lo} < 0 \textbf{ then } 1 \textbf{ else } 0, \_] \\
 & \quad [u_{lo}|_{\delta}, u_{hi}, \ldots, v_{lo}|_{t0}, \boxed{v_{hi}}] \\
\texttt{ausd} & [(256 + u_{hi} - v_{hi} - (\textbf{if } u_{lo} - v_{lo} < 0 \textbf{ then } 1 \textbf{ else } 0)) \bmod 256, \\
 & \qquad\quad (256 + u_{lo} - v_{lo}) \bmod 256] \\
 & \quad [\textbf{if } A < 0 \textbf{ then } 1 \textbf{ else } 0, \_][u_{lo}|_{\delta}, u_{hi}, \ldots, v_{lo}|_{t0}, \boxed{v_{hi}}] \\
 & = [(u - v)_{hi}, (u - v)_{lo}][\textbf{if}(u - v) < 0 \textbf{ then } 1 \textbf{ else } 0, \_] \\
 & \qquad\quad [u_{lo}|_{\delta}, u_{hi}, \ldots, v_{lo}|_{t0}, \boxed{v_{hi}}] \\
\texttt{hltca} & \textbf{if } u - v < 0 \textbf{ then } \textit{halt} \\
 & [(u - v)_{hi}, (u - v)_{lo}][0, \_][u_{lo}|_{\delta}, u_{hi}, \ldots, v_{lo}|_{t0}, \boxed{v_{hi}}]
\end{array}$$

□ 4.47

4.48 Correctness of unsigned multiply

The left argument unsigned value u is at $\delta, \delta - 1$, with the weaker than usual assumption, that D is initially unknown. The right argument unsigned value v is in AB.

We are required to show that the final contents of AB are the unsigned value $u * v$.

4.48.1 Lemma: correctness of mult subroutine *binopfn umul*

The initial state of memory is given by the state on *xCallOp* (which is just a *goto*).

$$
\begin{array}{ll}
 & [l_{hi}, l_{lo}][\_,\_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, \boxed{l_{lo}}, l_{hi}] \\
\mathit{xsdro}\ \ 16 & [l_{hi}, l_{lo}][\_,\_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{bldd} & [l_{hi}, v_{lo}][\_,\_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{aldi}\ \ 0 & [0, v_{lo}][\_,\_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{dpi}\ \ 2 & [0, v_{lo}][\_,\_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, v_{lo}, v_{hi}, \boxed{u_{lo}}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{aumd} & [(u_{lo} * v_{lo}) \mathrm{\ div\ } 256, (u_{lo} * v_{lo}) \mathrm{\ mod\ } 256][\_,\_][v]_C \\
 & \qquad [u_{lo}|_\delta, u_{hi}, \ldots, v_{lo}, v_{hi}, \boxed{u_{lo}}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
 & = [(u_{lo} * v_{lo}) \mathrm{\ div\ } 256, (u * v)_{lo}][\_,\_][v]_C \\
 & \qquad [u_{lo}|_\delta, u_{hi}, \ldots, v_{lo}, v_{hi}, \boxed{u_{lo}}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{dpi}\ \ -16 & [(u_{lo} * v_{lo}) \mathrm{\ div\ } 256, (u * v)_{lo}][\_,\_][v]_C \\
 & \qquad [u_{lo}|_\delta, u_{hi}, \ldots, \boxed{-}|_2, -, \\
 & \qquad\qquad -, -, -, -, -, -, -, -, -, -, -, -, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{bstd} & [(u_{lo} * v_{lo}) \mathrm{\ div\ } 256, (u * v)_{lo}][\_,\_][v]_C \\
 & \qquad [u_{lo}|_\delta, u_{hi}, \ldots, \boxed{(u * v)_{lo}}|_2, -, \\
 & \qquad\qquad -, -, -, -, -, -, -, -, -, -, -, -, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{dpi}\ \ 15 & [(u_{lo} * v_{lo}) \mathrm{\ div\ } 256, (u * v)_{lo}][\_,\_][v]_C \\
 & \qquad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, -, \\
 & \qquad\qquad -, -, -, -, -, -, -, -, -, -, -, -, v_{lo}, \boxed{v_{hi}}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{bldd} & [(u_{lo} * v_{lo}) \mathrm{\ div\ } 256, v_{hi}][\_,\_][v]_C \\
 & \qquad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, -, \\
 & \qquad\qquad -, -, -, -, -, -, -, -, -, -, -, -, v_{lo}, \boxed{v_{hi}}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
\texttt{dpi}\ \ 1 & [(u_{lo} * v_{lo}) \mathrm{\ div\ } 256, v_{hi}][\_,\_][v]_C \\
 & \qquad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, -, \\
 & \qquad\qquad -, -, -, -, -, -, -, -, -, -, -, -, v_{lo}, v_{hi}, \boxed{u_{lo}}|_{18}, u_{hi}, l_{lo}, l_{hi}]
\end{array}
$$

aumd
$$\begin{aligned}&[\_, ((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \_,\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, \boxed{u_{lo}}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

dpi -15
$$\begin{aligned}&[\_, ((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \boxed{\_},\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

bstd
$$\begin{aligned}&[\_,\_][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \boxed{((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256},\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

dpi 13
$$\begin{aligned}&[\_,\_][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, ((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256,\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

bldd
$$\begin{aligned}&[\_, v_{lo}][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, ((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256,\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

aldi 0
$$\begin{aligned}&[0, v_{lo}][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, ((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256,\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

dpi 3
$$\begin{aligned}&[0, v_{lo}][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, ((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256,\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, \boxed{u_{hi}}, l_{lo}, l_{hi}]\end{aligned}$$

aumd
$$\begin{aligned}&[(u_{hi} * v_{lo}) \text{ div } 256, (u_{hi} * v_{lo}) \text{ mod } 256][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, ((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256,\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, \boxed{u_{hi}}, l_{lo}, l_{hi}]\end{aligned}$$

dpi -16
$$\begin{aligned}&[(u_{hi} * v_{lo}) \text{ div } 256, (u_{hi} * v_{lo}) \text{ mod } 256][\_,\_][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \boxed{((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256},\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

ccb
$$\begin{aligned}&[(u_{hi} * v_{lo}) \text{ div } 256, (u_{hi} * v_{lo}) \text{ mod } 256][\_, 0][v]_C\\ &\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \boxed{((u_{lo} * v_{lo}) \text{ div } 256 + u_{lo} * v_{hi}) \text{ mod } 256},\\ &\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]\end{aligned}$$

`buad`

$$\begin{aligned}
&[(u_{hi} * v_{lo}) \operatorname{div} 256, (u_{hi} * v_{lo}) \operatorname{mod} 256 \\
&\qquad\qquad + ((u_{lo} * v_{lo}) \operatorname{div} 256 + u_{lo} * v_{hi}) \operatorname{mod} 256] \\
&\quad [\_, 0][v]_C \\
&\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \boxed{((u_{lo} * v_{lo}) \operatorname{div} 256 + u_{lo} * v_{hi}) \operatorname{mod} 256}, \\
&\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}] \\
&= [\_, (u * v)_{hi}][\_, 0][v]_C [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \boxed{\_}, \\
&\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]
\end{aligned}$$

`bstd`

$$\begin{aligned}
&[\_, (u * v)_{hi}][\_, 0][v]_C \\
&\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, \boxed{(u * v)_{hi}}, \\
&\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]
\end{aligned}$$

`dpi` -1

$$\begin{aligned}
&[\_, (u * v)_{hi}][\_, 0][v]_C \\
&\quad [u_{lo}|_\delta, u_{hi}, \ldots, \boxed{(u * v)_{lo}|_2}, (u * v)_{hi}, \\
&\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]
\end{aligned}$$

`cldd`

$$\begin{aligned}
&[(u * v)_{hi}, (u * v)_{lo}][\_, 0][u * v]_C \\
&\quad [u_{lo}|_\delta, u_{hi}, \ldots, \boxed{(u * v)_{lo}|_2}, (u * v)_{hi}, \\
&\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, l_{lo}, l_{hi}]
\end{aligned}$$

`dpi` 18

$$\begin{aligned}
&[(u * v)_{hi}, (u * v)_{lo}][\_, 0][u * v]_C \\
&\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, (u * v)_{hi}, \\
&\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, \boxed{l_{lo}}, l_{hi}]
\end{aligned}$$

`abldd`

$$\begin{aligned}
&[l_{hi}, l_{lo}][\_, 0][u * v]_C \\
&\quad [u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, (u * v)_{hi}, \\
&\qquad \_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_,\_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, \boxed{l_{lo}}, l_{hi}]
\end{aligned}$$

xag goto l

□ 4.48.1

4.48.2 Correctness of unsigned multiply using lemma

$[v_{hi}, v_{lo}][\_, \_][\_]_C[u_{lo}|_\delta, u_{hi}]$

`cldab` $[v_{hi}, v_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}]$

xsdrs δ $[v_{hi}, v_{lo}][\_, \_][v]_C[\boxed{u_{lo}}|_\delta, u_{hi}]$

`abldd` $[u_{hi}, u_{lo}][\_, \_][v]_C[\boxed{u_{lo}}|_\delta, u_{hi}]$

xsdro 18 $[u_{hi}, u_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{\_}|_{18}, \_]$

`abstd` $[u_{hi}, u_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{u_{lo}}|_{18}, u_{hi}]$

`dpi` -2 $[u_{hi}, u_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{\_}, \_, u_{lo}|_{18}, u_{hi}]$

`cstd` $[v_{hi}, v_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}]$

xll l $[l_{hi}, l_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, \boxed{v_{lo}}, v_{hi}, u_{lo}|_{18}, u_{hi}]$

`dpi` 4 $[l_{hi}, l_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, \boxed{\_}, \_]$

`abstd` $[l_{hi}, l_{lo}][\_, \_][v]_C[u_{lo}|_\delta, u_{hi}, \ldots, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, \boxed{l_{lo}}, l_{hi}]$

xCallOp umul(see lemma)

xli l

$[l_{hi}, l_{lo}][\_, 0][u * v]_C$
$[u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, (u * v)_{hi},$
$\_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, \boxed{l_{lo}}, l_{hi}]$

`abldc` $[u * v][\_, 0][u * v]_C$
$[u_{lo}|_\delta, u_{hi}, \ldots, (u * v)_{lo}|_2, (u * v)_{hi},$
$\_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, v_{lo}, v_{hi}, u_{lo}|_{18}, u_{hi}, \boxed{l_{lo}}, l_{hi}]$

□ 4.48.2

□ 4.48

4.49 Correctness of unsigned division

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of AB are the unsigned value u div v.

4.49.1 Introduction

We convert the simplified, but high level, algorithm of appendix B to an Asp algorithm.

We want to calculate $\overline{u} \operatorname{div} \overline{v}$, where, on call, $\overline{u}_2, \overline{u}_1$ are stored at $t0, t1$, and $\overline{v}_2, \overline{v}_1$ at $t3, t4$. After the call, the result $\langle q_1 q_2 \rangle$ is in $[a, b]$. (The current implementation includes an optimisation for binary operators, by including the code for each operator only once, and using a 'function call' to access it. That standard optimisation is not included here.)

The Asp unsigned multiplication instruction, (immediate, using a supplied value x, or indirect, assuming the data address register D is pointing to location *addr* which stores byte x), multiplies the b register by x, adds the contents of a, and treats the result as a 16-bit value:

$$[a, b] := [(a + b * x) \operatorname{div} 256, (a + b * x) \operatorname{mod} 256]$$

The Asp unsigned division instruction, (immediate, using a supplied value x, or indirect, assuming the data address register D is pointing to location *addr* which stores byte x), divides the 2-byte number $[a, b]$ by x, returning the single byte remainder and single byte quotient:

$$[a, b] := [(a * 256 + b) \operatorname{mod} x, (a * 256 + b) \operatorname{div} x]$$

This enables us to perform the two divisions in the simplified algorithm. Translation into Asp is straightforward, if tedious.

We use the following memory allocation in scratch space:

$topop + 6$	$q_1, v_1 + 1, d$
$topop + 5$	q_2
$topop + 4$	v_1
$topop + 3$	v_2
$topop + 2$	u_0
$topop + 1$	u_1
$topop + 0$	u_2

Although the instruction *xsdro n*, to set the data register D to $topop + n$, would make the algorithm easier to read, we instead use `dpi` n to move D by

n where possible. *xsdro* is linked and hexed into two instructions, `dxi` and `dix`, whereas `dpi` is a single instruction.

4.49.2 The algorithm *binopfn udiv*

As noted earlier, on call, $\overline{u}_2, \overline{u}_1$ are stored at $t0, t1$, and $\overline{v}_2, \overline{v}_1$ at $t3, t4$. The return label is stored at $t7, t8$ (but is elided in the following proof, for brevity). So the initial state of memory is

$$[\_,\_][\_,\_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \overline{v}_1, \_, \_, l_2, l_1]$$

start	*xli* l	
	xsdro 4	$[\_,\_][\_,\_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \boxed{\overline{v}_1}, \_, \_, l_2, l_1]$
	`aldd`	$[\overline{v}_1, \_][\_,\_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \boxed{\overline{v}_1}, \_, \_]$
	`aeqi` 0	$[\overline{v}_1, \_][\overline{v}_1 = 0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \boxed{\overline{v}_1}, \_, \_]$
	`nca`	$[\overline{v}_1, \_][\overline{v}_1 \neq 0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \boxed{\overline{v}_1}, \_, \_]$
	xrja$(l+1)$	**if** $v_1 \neq 0$ **then** *goto* main $[\overline{v}_1, \_][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \boxed{\overline{v}_1}(= 0), \_, \_]$
		$v_1 = 0$, use single byte division

	`dpi` -3	$[\overline{v}_1,\_][0,\_][\overline{u}_2,\boxed{\overline{u}_1},\_,\overline{v}_2,\overline{v}_1(=0),\_,\_]$
	`bldd`	$[\overline{v}_1,\overline{u}_1][0,\_][\overline{u}_2,\boxed{\overline{u}_1},\_,\overline{v}_2,\overline{v}_1(=0),\_,\_]$
	`aldi` 0	$[0,\overline{u}_1][0,\_][\overline{u}_2,\boxed{\overline{u}_1},\_,\overline{v}_2,\overline{v}_1(=0),\_,\_]$
	`dpi` 2	$[0,\overline{u}_1][0,\_][\overline{u}_2,\overline{u}_1,\_,\boxed{\overline{v}_2},\overline{v}_1(=0),\_,\_]$
	`audd`	$[\overline{u}_1 \bmod \overline{v}_2,\overline{u}_1 \operatorname{div} \overline{v}_2][0,\_][\overline{u}_2,\overline{u}_1,\_,\boxed{\overline{v}_2},\overline{v}_1(=0),\_,\_]$ $= [\mathit{rem},q_1][0,\_][\overline{u}_2,\overline{u}_1,\_,\boxed{\overline{v}_2},\overline{v}_1(=0),\_,\_]$
	`dpi` 3	$[\mathit{rem},q_1][0,\_][\overline{u}_2,\overline{u}_1,\_,\overline{v}_2,\overline{v}_1(=0),\_,\boxed{\_}]$
	`bstd`	$[\mathit{rem},q_1][0,\_][\overline{u}_2,\overline{u}_1,\_,\overline{v}_2,\overline{v}_1(=0),\_,\boxed{q_1}]$
	`dpi` -6	$[\mathit{rem},q_1][0,\_][\boxed{\overline{u}_2},\overline{u}_1,\_,\overline{v}_2,\overline{v}_1(=0),\_,q_1]$
	`bldd`	$[\mathit{rem},\overline{u}_2][0,\_][\boxed{\overline{u}_2},\overline{u}_1,\_,\overline{v}_2,\overline{v}_1(=0),\_,q_1]$
	`dpi` 3	$[\mathit{rem},\overline{u}_2][0,\_][\overline{u}_2,\overline{u}_1,\_,\boxed{\overline{v}_2},\overline{v}_1(=0),\_,q_1]$
	`audd`	$[(\mathit{rem}*256+\overline{u}_2) \bmod \overline{v}_2,(\mathit{rem}*256+\overline{u}_2) \operatorname{div} \overline{v}_2]$ $[0,\_][\overline{u}_2,\overline{u}_1,\_,\boxed{\overline{v}_2},\overline{v}_1(=0),\_,q_1]$ $= [\_,q_2][0,\_][\overline{u}_2,\overline{u}_1,\_,\boxed{\overline{v}_2},\overline{v}_1(=0),\_,q_1]$
	`dpi` 3	$[\_,q_2][0,\_][\overline{u}_2,\overline{u}_1,\_,\overline{v}_2,\overline{v}_1(=0),\_,\boxed{q_1}]$
	`aldd`	$[q_1,q_2][0,\_][\overline{u}_2,\overline{u}_1,\_,\overline{v}_2,\overline{v}_1(=0),\_,\boxed{q_1}]$
	$\mathit{xrgs}(l+8)$	*goto* end

---

main	$\mathit{xli}(l+1)$	$0<\overline{v}_1$, use full algorithm (from *goto* main): $[\overline{v}_1,\_][1,\_][\overline{u}_2,\overline{u}_1,\_,\overline{v}_2,0<\boxed{\overline{v}_1},\_,\_]$
	`auci` 128	$[\overline{v}_1,\_][128\le\overline{v}_1,\_][\overline{u}_2,\overline{u}_1,\_,\overline{v}_2,0<\boxed{\overline{v}_1},\_,\_]$
	$\mathit{xrja}(l+2)$	**if** $128 \le v_1$ **then** *goto* zero $[\overline{v}_1,\_][0,\_][\overline{u}_2,\overline{u}_1,\_,\overline{v}_2,0<\boxed{\overline{v}_1}<128,\_,\_]$

$\overline{v}_1 < 128$, normalise

calculate $d = 256 \operatorname{div} (\overline{v}_1+1)$

`auai 1` $[(\overline{v}_1 + 1 + 0) \bmod 256, \_][(\overline{v} + 1 + 0) \operatorname{div} 256, \_]$

$[\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \boxed{\overline{v}_1} < 128, \_, \_]$

[1] $= [\overline{v}_1 + 1, \_][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \boxed{\overline{v}_1} < 128, \_, \_]$

`dpi 2` $[\overline{v}_1 + 1, \_][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{\_}]$

`astd` $[\overline{v}_1 + 1, \_][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{\overline{v}_1 + 1}]$

`aldi 1` $[1, \_][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{\overline{v}_1 + 1}]$

`bldi 0` $[1, 0][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{\overline{v}_1 + 1}]$

`audd` $[256 \bmod (\overline{v}_1 + 1), 256 \operatorname{div} (\overline{v}_1 + 1)][0, \_]$

$[\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{\overline{v}_1 + 1}]$

$= [\_, d][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{\overline{v}_1 + 1}]$

`bstd` $[\_, d][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$

normalise $u := \overline{u} * d$

`dpi −6` $[\_, d][0, \_][\boxed{\overline{u}_2}, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`bldd` $[\_, \overline{u}_2][0, \_][\boxed{\overline{u}_2}, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`aldi 0` $[0, \overline{u}_2][0, \_][\boxed{\overline{u}_2}, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`dpi 6` $[0, \overline{u}_2][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$

`aumd` $[(\overline{u}_2 * d) \operatorname{div} 256, (\overline{u}_2 * d) \bmod 256][0, \_]$

$[\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$

$= [(\overline{u}_2 * d) \operatorname{div} 256, u_2][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$

`dpi −6` $[(\overline{u}_2 * d) \operatorname{div} 256, u_2][0, \_][\boxed{\overline{u}_2}, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`bstd` $[(\overline{u}_2 * d) \operatorname{div} 256, u_2][0, \_][\boxed{u_2}, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`dpi 1` $[(\overline{u}_2 * d) \operatorname{div} 256, u_2][0, \_][u_2, \boxed{\overline{u}_1}, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`bldd` $[(\overline{u}_2 * d) \operatorname{div} 256, \overline{u}_1][0, \_][u_2, \boxed{\overline{u}_1}, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`dpi 5` $[(\overline{u}_2 * d) \operatorname{div} 256, \overline{u}_1][0, \_][u_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$

`aumd` $[((\overline{u}_2 * d) \operatorname{div} 256 + \overline{u}_1 * d) \operatorname{div} 256,$

$((\overline{u}_2 * d) \operatorname{div} 256 + \overline{u}_1 * d) \bmod 256]$

$[0, \_][u_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$

$= [u_0, u_1][0, \_][u_2, \overline{u}_1, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$

`dpi −5` $[u_0, u_1][0, \_][u_2, \boxed{\overline{u}_1}, \_, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

`abstd` $[u_0, u_1][0, \_][u_2, \boxed{u_1}, u_0, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, d]$

normalise $v := \overline{v} * d$

	`dpi 2`	$[u_0, u_1][0, \_][u_2, u_1, u_0, \boxed{\overline{v}_2}, 0 < \overline{v}_1 < 128, \_, d]$
	`bldd`	$[u_0, \overline{v}_2][0, \_][u_2, u_1, u_0, \boxed{\overline{v}_2}, 0 < \overline{v}_1 < 128, \_, d]$
	`aldi 0`	$[0, \overline{v}_2][0, \_][u_2, u_1, u_0, \boxed{\overline{v}_2}, 0 < \overline{v}_1 < 128, \_, d]$
	`dpi 3`	$[0, \overline{v}_2][0, \_][u_2, u_1, u_0, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
	`aumd`	$[(\overline{v}_2 * d) \operatorname{div} 256, (\overline{v}_2 * d) \bmod 256][0, \_]$
		$[u_2, u_1, u_0, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
		$= [(\overline{v}_2 * d) \operatorname{div} 256, v_2][0, \_][u_2, u_1, u_0, \overline{v}_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
	`dpi −3`	$[(\overline{v}_2 * d) \operatorname{div} 256, v_2][0, \_][u_2, u_1, u_0, \boxed{\overline{v}_2}, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
	`bstd`	$[(\overline{v}_2 * d) \operatorname{div} 256, v_2][0, \_][u_2, u_1, u_0, \boxed{v_2}, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
	`dpi 1`	$[(\overline{v}_2 * d) \operatorname{div} 256, v_2][0, \_][u_2, u_1, u_0, v_2, 0 < \boxed{\overline{v}_1} < 128, \_, \boxed{d}]$
	`bldd`	$[(\overline{v}_2 * d) \operatorname{div} 256, \overline{v}_1][0, \_][u_2, u_1, u_0, v_2, 0 < \boxed{\overline{v}_1} < 128, \_, d]$
	`dpi 2`	$[(\overline{v}_2 * d) \operatorname{div} 256, \overline{v}_1][0, \_][u_2, u_1, u_0, v_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
	`aumd`	$[((\overline{v}_2 * d) \operatorname{div} 256 + \overline{v}_1 * d) \operatorname{div} 256,$
		$((\overline{v}_2 * d) \operatorname{div} 256 + \overline{v}_1 * d) \bmod 256]$
		$[0, \_][u_2, u_1, u_0, v_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
[2]		$= [0, v_1][0, \_][u_2, u_1, u_0, v_2, 0 < \overline{v}_1 < 128, \_, \boxed{d}]$
	`dpi −2`	$[0, v_1][0, \_][u_2, u_1, u_0, v_2, 0 < \boxed{\overline{v}_1} < 128, \_, d]$
	`bstd`	$[0, v_1][0, \_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, d]$
	$xrgs(l+3)$	*goto* div

zero	$xli(l+2)$	$128 \leq \overline{v}_1$, just set u_0 to 0
		(from *goto* zero): $[\overline{v}_1, \_][1, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, 128 \leq \boxed{\overline{v}_1}, \_, \_]$
		$= [v_1, \_][1, \_][u_2, u_1, \_, v_2, 128 \leq \boxed{v_1}, \_, \_]$
	`dpi −2`	$[v_1, \_][1, \_][u_2, u_1, \boxed{\_}, v_2, 128 \leq v_1, \_, \_]$
	`aldi 0`	$[0, \_][1, \_][u_2, u_1, \boxed{\_}, v_2, 128 \leq v_1, \_, \_]$
	`astd`	$[0, \_][1, \_][u_2, u_1, \boxed{0}, v_2, 128 \leq v_1, \_, \_]$

div	$xli(l+3)$	calculate $\hat{q}$
		(from *goto* div): $[0, v_1][0, \_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, d]$
		$\Rightarrow [0, \_][\_, \_][u_2, u_1, u_0, v_2, v_1, \_, \_]$

	xsdro 2	$[0,\_][\_,\_][u_2, u_1, \boxed{u_0}, v_2, v_1, \_, \_]$
	`aldd`	$[u_0,\_][\_,\_][u_2, u_1, \boxed{u_0}, v_2, v_1, \_, \_]$
	`dpi` 2	$[u_0,\_][\_,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
	`aeqd`	$[u_0,\_][u_0 = v_1,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
	$xrja(l+4)$	**if** $u_0 = v_1$ **then** *goto* maxq
		$[u_0,\_][0,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
		calculate $\hat{q} = (u_0 * 256 + u_1)\,\mathrm{div}\,v_1(<256)$
	`dpi` -3	$[u_0,\_][0,\_][u_2, \boxed{u_1}, u_0, v_2, v_1, \_, \_]$
	`bldd`	$[u_0,u_1][0,\_][u_2, \boxed{u_1}, u_0, v_2, v_1, \_, \_]$
	`dpi` 3	$[u_0,u_1][0,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
	`audd`	$[(u_0 * 256 + u_1) \bmod v_1, (u_0 * 256 + u_1)\,\mathrm{div}\,v_1]$ $[0,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
		$= [\_,\hat{q}][0,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
	$xrgs(l+5)$	*goto* rem
maxq	$xli(l+4)$	set $\hat{q}$ to 255
		(from *goto* maxq): $[u_0,\_][1,\_][u_2, u_1, u_0, v_2, \boxed{v_1}(= u_0), \_, \_]$
	`bldi` 255	$[u_0,255][1,\_][u_2, u_1, u_0, v_2, \boxed{v_1}(= u_0), \_, \_]$
[3]		$= [u_0,\hat{q}][1,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$

rem	$xli(l+5)$	
		(from *goto* rem): $[\_,\hat{q}][0,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
		$\Rightarrow [\_,\hat{q}][\_,\_][u_2, u_1, u_0, v_2, \boxed{v_1}, \_, \_]$
	`dpi` 1	$[\_,\hat{q}][\_,\_][u_2, u_1, u_0, v_2, v_1, \boxed{\_}, \_]$
	`bstd`	$[\_,\hat{q}][\_,\_][u_2, u_1, u_0, v_2, v_1, \boxed{\hat{q}}, \_]$
		calculate $v' = \hat{q} * v$

`dpi −2`	$[\_, \hat{q}][\_, \_][u_2, u_1, u_0, \boxed{v_2}, v_1, \hat{q}, \_]$
`aldi 0`	$[0, \hat{q}][\_, \_][u_2, u_1, u_0, \boxed{v_2}, v_1, \hat{q}, \_]$
`aumd`	$[(v_2 * \hat{q}) \text{ div } 256, (v_2 * \hat{q}) \bmod 256][\_, \_]$ $[u_2, u_1, u_0, \boxed{v_2}, v_1, \hat{q}, \_]$ $= [(v_2 * \hat{q}) \text{ div } 256, v_2'][\_, \_][u_2, u_1, u_0, \boxed{v_2}, v_1, \hat{q}, \_]$
`bstd`	$[(v_2 * \hat{q}) \text{ div } 256, v_2'][\_, \_][u_2, u_1, u_0, \boxed{v_2'}, v_1, \hat{q}, \_]$
`dpi 1`	$[(v_2 * \hat{q}) \text{ div } 256, v_2'][\_, \_][u_2, u_1, u_0, v_2', \boxed{v_1}, \hat{q}, \_]$
`bldd`	$[(v_2 * \hat{q}) \text{ div } 256, v_1][\_, \_][u_2, u_1, u_0, v_2', \boxed{v_1}, \hat{q}, \_]$
`dpi 1`	$[(v_2 * \hat{q}) \text{ div } 256, v_1][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}, \_]$
`aumd`	$[((v_2 * \hat{q}) \text{ div } 256 + v_1 * \hat{q}) \text{ div } 256,$ $((v_2 * \hat{q}) \text{ div } 256 + v_1 * \hat{q}) \bmod 256]$ $[\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}, \_]$ $= [v_0', v_1'][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}, \_]$
	test $rem = u - v' < 0$, test $u < v'$
`dpi −3`	$[v_0', v_1'][\_, \_][u_2, u_1, \boxed{u_0}, v_2', v_1, \hat{q}, \_]$
`cca`	$[v_0', v_1'][0, \_][u_2, u_1, \boxed{u_0}, v_2', v_1, \hat{q}, \_]$
`aucd`	$[v_0', v_1'][u_0 < v_0', \_][u_2, u_1, \boxed{u_0}, v_2', v_1, \hat{q}, \_]$
$xrja(l+7)$	**if** $u_0 < v_0'$ **then** *goto* qtoobig $[v_0', v_1'][0, \_][u_2, u_1, \boxed{u_0}(\geq v_0'), v_2', v_1, \hat{q}, \_]$
`aeqd`	$[v_0', v_1'][v_0' = u_0, \_][u_2, u_1, \boxed{u_0}(\geq v_0'), v_2', v_1, \hat{q}, \_]$
`nca`	$[v_0', v_1'][v_0' \neq u_0, \_][u_2, u_1, \boxed{u_0}(\geq v_0'), v_2', v_1, \hat{q}, \_]$
$xrja(l+6)$	**if** $v_0' \neq u_0$ **then** *goto* qokay $[v_0', v_1'][0, \_][u_2, u_1, \boxed{u_0}(= v_0'), v_2', v_1, \hat{q}, \_]$
`dpi −1`	$[v_0', v_1'][0, \_][u_2, \boxed{u_1}, u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
`ccb`	$[v_0', v_1'][0, 0][u_2, \boxed{u_1}, u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
`bucd`	$[v_0', v_1'][0, u_1 < v_1'][u_2, \boxed{u_1}, u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
$xrjb(l+7)$	**if** $u_1 < v_1'$ **then** *goto* qtoobig $[v_0', v_1'][0, 0][u_2, \boxed{u_1}(\geq v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
`beqd`	$[v_0', v_1'][0, v_1' = u_1][u_2, \boxed{u_1}(\geq v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
`ncb`	$[v_0', v_1'][0, v_1' \neq u_1][u_2, \boxed{u_1}(\geq v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
$xrjb(l+6)$	**if** $v_1' \neq u_1$ **then** *goto* qokay $[v_0', v_1'][0, 0][u_2, \boxed{u_1}(= v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$

	`dpi 2`	$[v_0', v_1'][0, 0][u_2, u_1(= v_1'), u_0(= v_0'), \boxed{v_2'}, v_1, \hat{q}, \_]$
	`aldd`	$[v_2', v_1'][0, 0][u_2, u_1(= v_1'), u_0(= v_0'), \boxed{v_2'}, v_1, \hat{q}, \_]$
	`dpi` -3	$[v_2', v_1'][0, 0][\boxed{u_2}, u_1(= v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
	`aucd`	$[v_2', v_1'][u_2 < v_2', 0][\boxed{u_2}, u_1(= v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
	$\mathit{xrja}(l + 7)$	**if** $u_2 < v_2'$ **then** *goto* qtoobig $[v_2', v_1'][0, 0][\boxed{u_2}(\geq v_2'), u_1(= v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
qokay	$\mathit{xli}(l + 6)$	set $[a, b]$ to $[0, \hat{q}]$ (from $v_0' \neq u_0$): $[v_0', v_1'][1, \_][u_2, u_1, \boxed{u_0}(> v_0'), v_2', v_1, \hat{q}, \_]$ (from $v_1' \neq u_1$): $[v_0', v_1'][0, 1][u_2, \boxed{u_1}(> v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
[4]		$\Rightarrow [\_, v_1'][\_, \_][u_2, u_1, u_0, v_2', v_1, q, \_]$
	xsdro 5	$[\_, v_1'][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{q}, \_]$
	`bldd`	$[\_, q][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{q}, \_]$
	`aldi 0`	$[0, q][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{q}, \_]$
	$\mathit{xrgs}(l + 8)$	*goto* end
qtoobig	$\mathit{xli}(l + 7)$	set $[a, b]$ to $[0, \hat{q} - 1]$ (from $u_0 < v_0'$ test): $[v_0', v_1'][1, \_][u_2, u_1, \boxed{u_0}(< v_0'), v_2', v_1, \hat{q}, \_]$ (from $u_1 < v_1'$ test): $[v_0', v_1'][0, 1][u_2, \boxed{u_1}(< v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$ (from $u_2 < v_2'$ test): $[v_2', v_1'][1, 0][\boxed{u_2}(< v_2'), u_1(= v_1'), u_0(= v_0'), v_2', v_1, \hat{q}, \_]$
[5]		$\Rightarrow [\_, v_1'][\_, \_][u_2, u_1, u_0, v_2', v_1, \hat{q}(= q + 1), \_]$
	xsdro 5	$[\_, v_1'][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}(= q + 1), \_]$
	`bldd`	$[\_, \hat{q}][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}(= q + 1), \_]$
	`ccb`	$[\_, \hat{q}][\_, 0][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}(= q + 1), \_]$
	`aldi 0`	$[0, \hat{q}][\_, 0][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}(= q + 1), \_]$
	`busi 1`	$[0, \hat{q} - 1 - 0][\_, \textbf{if } \hat{q} - 1 - 0 < 0 \textbf{ then } \hat{q} - 1 - 0 + 256 \textbf{ else } 0]$ $[u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}(= q + 1), \_]$
[6]		$= [0, q][\_, 0][u_2, u_1, u_0, v_2', v_1, \boxed{\hat{q}}(= q + 1), \_]$

end	$xli(l+8)$	
		(from single byte *goto* end): $[q_1, q_2][0, \_][\overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \overline{v}_1(=0), \_, \boxed{q_1}]$ (from *goto* end): $[0, q][\_, \_][u_2, u_1, u_0, v_2', v_1, \boxed{q}, \_]$ $\Rightarrow [q_1, q_2][\_, \_][\_, \_, \_, \_, \_, \_, \_, l_2, l_1]$
	`cldab`	$[q_1, q_2][\_, \_][q]_C[\_, \_, \_, \_, \_, \_, \_, l_2, l_1]$
	$xsdro\ \ 7$	$[q_1, q_2][\_, \_][q]_C[\_, \_, \_, \_, \_, \_, \_, \boxed{l_2}, l_1]$
	`abldd`	$[l_1, l_2][\_, \_][q]_C[\_, \_, \_, \_, \_, \_, \_, \boxed{l_2}, l_1]$
	xag	*goto l*

1. $\overline{v}_1 < 128 \Rightarrow (\overline{v}_1 + 1) \bmod 256 = \overline{v}_1 + 1$
   $\overline{v}_1 < 128 \Rightarrow (\overline{v}_1 + 1) \text{ div } 256 = 0$
2. $Lemma3 \Rightarrow v_1 < 256$
3. $u_0 = v_1 \Rightarrow \hat{q} = 255$
4. $u \geq v' \Rightarrow rem \geq 0$
   $rem \geq 0 \Rightarrow q = \hat{q}$
5. $u < v' \Rightarrow rem < 0$ $rem < 0 \Rightarrow q + 1 = \hat{q}$
6. $q > 0 \Rightarrow \hat{q} - 1 > 0$

□ 4.49.2

This template is approximately 200 instructions long.

4.49.3 The division algorithm

On call, the second argument v is in the a, b registers, and the first argument u is on the stack memory at δ. On return, u div v is in the a, b registers.

$[\overline{v}_1, \overline{v}_2][\_,\_][\_]_C[\overline{u}_2|_\delta, \overline{u}_1]$

xsdro 3	$[\overline{v}_1, \overline{v}_2][\_,\_][\_]_C[\overline{u}_2\vert_\delta, \overline{u}_1, \ldots, \_, \_, \_, \boxed{\_}, \_]$
`abstd`	$[\overline{v}_1, \overline{v}_2][\_,\_][\_]_C[\overline{u}_2\vert_\delta, \overline{u}_1, \ldots, \_, \_, \_, \boxed{\overline{v}_2}, \overline{v}_1]$
xsdrs δ	$[\overline{v}_1, \overline{v}_2][\_,\_][\_]_C[\boxed{\overline{u}_2}\vert_\delta, \overline{u}_1, \ldots, \_, \_, \_, \overline{v}_2, \overline{v}_1]$
`abldd`	$[\overline{u}_1, \overline{u}_2][\_,\_][\_]_C[\boxed{\overline{u}_2}\vert_\delta, \overline{u}_1, \ldots, \_, \_, \_, \overline{v}_2, \overline{v}_1]$
xsdro 0	$[\overline{u}_1, \overline{u}_2][\_,\_][\_]_C[\overline{u}_2\vert_\delta, \overline{u}_1, \ldots, \boxed{\_}, \_, \_, \overline{v}_2\overline{v}_1]$
`abstd`	$[\overline{u}_1, \overline{u}_2][\_,\_][\_]_C[\overline{u}_2\vert_\delta, \overline{u}_1, \ldots, \boxed{\overline{u}_2}, \overline{u}_1, \_, \overline{v}_2, \overline{v}_1]$
xll l	$[l_1, l_2][\_,\_][\_]_C[\overline{u}_2\vert_\delta, \overline{u}_1, \ldots, \boxed{\overline{u}_2}, \overline{u}_1, \_, \overline{v}_2, \overline{v}_1]$
xsdro 7	$[l_1, l_2][\_,\_][\_]_C[\overline{u}_2\vert_\delta, \overline{u}_1, \ldots, \overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \overline{v}_1, \_, \_, \boxed{\_}, \_]$
`abstd`	$[l_1, l_2][\_,\_][\_]_C[\overline{u}_2\vert_\delta, \overline{u}_1, \ldots, \overline{u}_2, \overline{u}_1, \_, \overline{v}_2, \overline{v}_1, \_, \_, \boxed{l_2}, l_1]$
xCallOp udiv	
xli l	$[l_1, l_2][\_,\_][q]_C[\_, \_, \_, \_, \_, \_, \_, \boxed{l_2}, l_1]$
`abldc`	$[q_1, q_2][\_,\_][q]_C[\_, \_, \_, \_, \_, \_, \_, \boxed{l_2}, l_1]$

□ 4.49.3

□ 4.49

4.50 Correctness of unsigned modulus

The left argument unsigned value u is at $\delta, \delta - 1$, pointed to by D. The right argument unsigned value v is in AB.

We are required to show that the final contents of AB are the unsigned value $u \bmod v$.

Lemmas:

1. 4.49 : If on entry to *Idiv*, memory is $[v_{hi}, v_{lo}][\_,\_][\boxed{u_{lo}}|_\delta, u_{hi}]$, then on exit it is $[(u \text{ div } v)_{hi}, (u \text{ div } v)_{lo}][\_,\_][u_{lo}|_\delta, u_{hi}]$.
2. 4.48 : If on entry to *Imul*, memory is $[v_{hi}, v_{lo}][\_,\_][u_{lo}|_\delta, u_{hi}]$, then on exit it is $[(u * v)_{hi}, (u * v)_{lo}][\_,\_][u_{lo}|_\delta, u_{hi}]$.

3. 4.47 : If on entry to *Iminus*, memory is $[v_{hi}, v_{lo}][\_, \_][u_{lo}|_{\delta}, u_{hi}]$, then on exit it is $[(u - v)_{hi}, (u - v)_{lo}][\_, \_][u_{lo}|_{\delta}, u_{hi}]$.

$$
\begin{array}{ll}
 & [v_{hi}, v_{lo}][\_, \_][\_, \_, u_{lo}|_{\delta}, u_{hi}] \\
xsdrs(\delta + 2) & [v_{hi}, v_{lo}][\_, \_][\boxed{\_}, \_, u_{lo}|_{\delta}, u_{hi}] \\
\texttt{abstd} & [v_{hi}, v_{lo}][\_, \_][\boxed{v_{lo}}, v_{hi}, u_{lo}|_{\delta}, u_{hi}] \\
\mathcal{O}_{BO}\ udiv\ \delta & [(u \text{ div } v)_{hi}, (u \text{ div } v)_{lo}][\_, \_][v_{lo}, v_{hi}, u_{lo}|_{\delta}, u_{hi}] \\
\mathcal{O}_{BO}\ umul(\delta + 2) & [(v * (u \text{ div } v))_{hi}, (v * (u \text{ div } v))_{lo}][\_, \_][v_{lo}, v_{hi}, u_{lo}|_{\delta}, u_{hi}] \\
\mathcal{O}_{BO}\ uminus\ \delta & [(u - v * (u \text{ div } v))_{hi}, (u - v * (u \text{ div } v))_{lo}][\_, \_] \\
 & \qquad [v_{lo}, v_{hi}, u_{lo}|_{\delta}, u_{hi}] \\
 & = [(u \text{ mod } v)_{hi}, (u \text{ mod } v)_{lo}][\_, \_][v_{lo}, v_{hi}, u_{lo}|_{\delta}, u_{hi}]
\end{array}
$$

□ 4.50

4.51 Correctness of cast byte to enumerated

The byte value b is in the B register.

We are required to show that the final contents of B are the enumerated value equal to the byte value.

$$[\_, b]_{AB}$$

Do nothing: an enum is stored as a byte.

□ 4.51

4.52 Correctness of join bytes to unsigned

The left argument hi byte value a is at δ, pointed to by D. The right argument lo byte value b is in B.

We are required to show that the final contents of B are the unsigned value $a \dagger b$.

$$
\begin{array}{ll}
 & [\_, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots] \\
\texttt{aldd} & [a, b]_{AB}[\ldots, \boxed{a}|_{\delta}, \_, \ldots]
\end{array}
$$

□ 4.52

□ 4

5 Correctness of Expressions

5.1 Expression proof obligation

Obligation: the evaluation of the expression leaves the correct result in the AB registers.

5.2 Correctness of literal constant

We are required to show that the value of the literal constant is left in the AB registers.

$$\begin{array}{ll} & [\_,\_]_{AB} \\ \mathcal{O}_{l\tau i}\ \kappa & \\ & \kappa = u \Rightarrow [u_{hi}, u_{lo}]_{AB} \\ & \kappa = b \Rightarrow [\_, b]_{AB} \end{array}$$

□5.2.

5.3 Correctness of Value Reference

There are four subcases where a name may be encountered as a value reference (other kinds of names, such as function names, cannot occur here, due to static semantics checks).

5.3.1 Correctness of Value Reference – named constant

We are required to show that the value corresponding to the literal constant is left in the AB registers.

If the constant is imported:

$$\begin{array}{ll} & [\_,\_]_{AB} \\ \mathit{xConst}\ \xi & \\ \text{linker } \Rightarrow \mathcal{O}_{l\tau i} & \text{reduces to 5.2} \end{array}$$

If the constant is local, the value of the constant is looked up in the environment, and this value is used with the code for a literal constant, 5.2.

□5.3.1

5.3.2 Correctness of Value Reference – enumerated value

We are required to show that the value corresponding to the enumeration is left in the AB registers.

b is the appropriate value, looked up in the environment.

$$\begin{array}{ll} & [\_, \_]_{AB} \\ \texttt{bldi}\ b & [\_, b]_{AB} \end{array}$$

□5.3.2

5.3.3 Correctness of Value Reference – enumerated type

We are required to show that the value corresponding to the size of the enumerated type is left in the AB registers.

b is the appropriate value, calculated from a value looked up in the environment.

$$\begin{array}{ll} & [\_, \_]_{AB} \\ \texttt{bldi}\ b & [\_, b]_{AB} \end{array}$$

□5.3.3

5.3.4 Correctness of Value Reference – variable

We are required to show that the value of the variable is left in the AB registers.

We assume the induction hypothesis

1. $\mathcal{O}_{VL}$ (proved in section 5.7) : the result of evaluating the array indexes is to leave the data address of the relevant element in AB.

	$[\_,\_]_{AB}[\ldots, u_{lo}|_d, u_{hi}, \ldots]$
I	(hyp 1)
	$[d]_{AB}[\ldots, u_{lo}|_d, u_{hi}, \ldots]$
`daldab`	$[d]_{AB}[\ldots, \boxed{u_{lo}}|_d, u_{hi}, \ldots]$
$\mathcal{O}_{l\tau}$	$[u_{hi}, u_{lo}]_{AB}[\ldots, \boxed{u_{lo}}|_d, u_{hi}, \ldots]$

□5.3.4

□5.3

5.4 Correctness of Unary expression

We are required to show that the value of the unary expression $\Psi\ \epsilon$ is left in the AB registers.

We assume the induction hypotheses

1. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating the subexpression is left in AB: that on exit from code block I memory is
   $[\epsilon]_{AB}$
2. $\mathcal{O}_{UO}$ (proved in section 4.1) : the result of evaluating the operator is left in AB: that if on entry to code block I' memory is
   $[\epsilon]_{AB}$
   then on exit is is
   $[\Psi\ \epsilon]_{AB}$

	$[\_,\_]_{AB}$
I	(hyp 1)
	$[\epsilon]_{AB}$
I'	(hyp 2)
	$[\Psi\ \epsilon]_{AB}$

□5.4.

5.5 Correctness of Binary expression

We are required to show that the value of the binary expression $\epsilon\ \Omega\ \epsilon'$ is left in the AB registers.

We assume the induction hypotheses

1. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating the left subexpression is left in AB: that on exit from code block $I1$ memory is
   $[\epsilon]_{AB}$

2. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating the right subexpression is left in AB: that on exit from code block $I2$ memory is
   $[\epsilon']_{AB}$

3. $\mathcal{O}_{BO}$ (proved in section 4.18) : the result of evaluating the operator is left in AB: that if on entry to code block $I3$ memory is
   $[\epsilon']_{AB}[\ldots,\boxed{\epsilon_{lo}}\vert_{\delta},\epsilon_{hi},\ldots]$
   then on exit is is
   $[\epsilon\ \Omega\ \epsilon']_{AB}$

	$[\_,\_]_{AB}$
$I1_{\delta}$	(hyp 1)
	$[\epsilon]_{AB}[\ldots,\_,\_,\_\vert_{\delta},\_,\ldots]$
xsdrs δ	$[\epsilon]_{AB}[\ldots,\_,\_,\boxed{\_}\vert_{\delta},\_,\ldots]$
`abstd`	$[\epsilon]_{AB}[\ldots,\_,\_,\boxed{\epsilon_{lo}}\vert_{\delta},\epsilon_{hi},\ldots]$
$I2_{\delta+2}$	(hyp 2)
	$[\epsilon']_{AB}[\ldots,\_\vert_{\delta+2},\_,\epsilon_{lo},\epsilon_{hi},\ldots]$
xsdrs δ	$[\epsilon']_{AB}[\ldots,\_,\_,\boxed{\epsilon_{lo}}\vert_{\delta},\epsilon_{hi},\ldots]$
$I3$	(hyp 3)
	$[\epsilon\ \Omega\ \epsilon']_{AB}[\ldots,\_,\_,\epsilon_{lo}\vert_{\delta},\epsilon_{hi},\ldots]$

□5.5.

5.6 Correctness of Function Call

We are required to show that the return value of the function call is left in the AB registers.

We assume the induction hypotheses

1. $\mathcal{O}_{AP^*}$ (proved in section 5.10) : the result of evaluating the actual parameters is to store their values at the correct formal parameter locations

2. *xCall* (proved in section 8.3) : the result of calling the function is to save the return label passed in C, evaluate the function (variable initialisations and body stamtement) and store the result in C, load the return label, and jump to it.

	$[\_,\_]_{AB}$
P	(hyp 1)
	$[\_,\_]_{AB}[\text{parameters stored}\ldots,\_,\_,\_|_\delta,\_,\ldots]$
$xll\ l'$	$[l']_{AB}[\text{parameters stored}\ldots,\_,\_,\_|_\delta,\_,\ldots]$
`cldab`	$[l']_{AB}[l']_C[\text{parameters stored}\ldots,\_,\_,\_|_\delta,\_,\ldots]$
$xCall\ \xi$	(hyp 2)
	$[\_,\_]_{AB}[l']_C[\text{parameters stored}\ldots,l',\ldots,\_,\_,\_|_\delta,\_,\ldots]$
	$[\_,\_]_{AB}[f]_C[\text{parameters stored}\ldots,l',\ldots,\_,\_,\_|_\delta,\_,\ldots]$
	$[l']_{AB}[f]_C[\text{parameters stored}\ldots,l',\ldots,\_,\_,\_|_\delta,\_,\ldots]$
	goto l'
l'	$[l']_{AB}[f]_C[\ldots,\_,\_,\_|_\delta,\_,\ldots]$
`abldc`	$[f]_{AB}[f]_C[\ldots,\_,\_,\_|_\delta,\_,\ldots]$

□ 5.6

5.7 Correctness of variable location, $\mathcal{O}_{VL}$

We are required to show that the the result of evaluating the array indexes is to leave the data address of the relevant element in AB.

We assume the induction hypotheses

1. $\mathcal{O}_{E^*}$ (proved in section 5.8) : the result of evaluating the array indexes is to leave the relevant index offset in AB.

2. $\mathcal{O}_{Eaddr}$ (proved in section 5.11) : the result of evaluating *Iaddr* is to leave the start address of the array variable in AB.

$$\begin{array}{ll}
 & [\_,\_]_{AB}[\ldots,\_|_{\delta},\_,\ldots] \\
\mathit{IE} & \text{(hyp 1)} \\
 & [\text{(byte or unsigned) index offset}]_{AB}[\ldots,\_|_{\delta},\_,\ldots] \\
\mathit{Ioff}1 & [\text{unsigned index offset}]_{AB}[\ldots,\_|_{\delta},\_,\ldots] \\
\mathit{Ioff}2 & [\text{address offset}]_{AB}[\ldots,\_|_{\delta},\_,\ldots] \\
\mathit{xsdrs}\ \delta & [\text{address offset}]_{AB}[\ldots,\boxed{\_}|_{\delta},\_,\ldots] \\
\texttt{abstd} & [\text{address offset}]_{AB}[\ldots,\boxed{\mathit{adoff}_{lo}}|_{\delta},\mathit{adoff}_{hi},\ldots] \\
\mathit{Iaddr} & \text{(hyp 2)} \\
 & [\text{start address}]_{AB}[\ldots,\mathit{adoff}_{lo}|_{\delta},\mathit{adoff}_{hi},\ldots] \\
\mathit{xsdrs}\ \delta & [\text{start address}]_{AB}[\ldots,\boxed{\mathit{adoff}_{lo}}|_{\delta},\mathit{adoff}_{hi},\ldots] \\
\mathit{uplus}_{\delta} & [\text{start address + offset}]_{AB}[\ldots,\mathit{adoff}_{lo}|_{\delta},\mathit{adoff}_{hi},\ldots]
\end{array}$$

□ 5.7

5.8 Correctness of index offset, $\mathcal{O}_{E^*}$

Given a sequence of array expressions $[i_1]\ldots[i_n]$ for array subranges $[a_1..b_1]$ $\ldots[a_n..b_n]$, we want to calculate the index offset, and show it is equal to the Pasp offset calculated by *locationOffset*, that is, to show it calculates $\Sigma_i(ne_i * (\epsilon_i - lb_i)) \equiv \Sigma_i\mathcal{I}_i$.

We assume the induction hypotheses

1. $\mathcal{O}_{E1}$ (proved in section 5.9) : Given an expression ϵ, and compile-time constants lb and ne, the template results in $ne * (\epsilon - lb) \equiv \mathcal{I}$ in the AB registers.

2. $\mathcal{O}_{E^*}$ (proved in section 5.8) : We assume that $\mathcal{O}_{E^*}$ E leaves $\Sigma_i(ne_i*(\epsilon_i - lb_i)) \equiv \Sigma_i\mathcal{I}_i$ in the AB registers, in order to prove that $\mathcal{O}_{E^*}(E \frown \langle\epsilon\rangle)$ leaves $ne*(\epsilon - lb) + \Sigma_i(ne_i*(\epsilon_i - lb_i)) \equiv \mathcal{I} + \Sigma\mathcal{I}_i$ in the AB registers.
3. *uplus* (proved in section 4.46) : If on entry to *Iplus*, memory is $[v_{hi}, v_{lo}][\_,\_][u_{lo}|_\delta, u_{hi}]$, then on exit it is $[(u+v)_{hi}, (u+v)_{lo}][\_,\_][u_{lo}|_\delta, u_{hi}]$.

base case: single dimension

The base case is the first hypothesis above, proved elsewhere.

inductive step : Assuming E, we prove the case for $E \frown \langle\epsilon\rangle$.

$$\begin{array}{ll}
 & [\_,\_]_{AB}[\ldots,\_|_\delta,\_,\ldots] \\
IE_\delta & \text{(hyp 2)} \\
 & [\Sigma\mathcal{I}_i]_{AB}[\ldots,\_|_\delta,\_,\ldots] \\
xsdrs\ \delta & [\Sigma\mathcal{I}_i]_{AB}[\ldots,\boxed{-}|_\delta,\_,\ldots] \\
\mathcal{O}_{s\mathcal{T}} & [\Sigma\mathcal{I}_i]_{AB}[\ldots,\boxed{(\Sigma_i\mathcal{I}_i)_{lo}}|_\delta,(\Sigma_i\mathcal{I}_i)_{hi},\ldots] \\
Ie_{\delta+2} & \text{(hyp 1)} \\
 & [\mathcal{I}]_{AB}[\ldots,\_|_{\delta+2},\_,(\Sigma_i\mathcal{I}_i)_{lo}|_\delta,(\Sigma_i\mathcal{I}_i)_{hi},\ldots] \\
xsdrs\ \delta & [\mathcal{I}]_{AB}[\ldots,\_|_{\delta+2},\_,\boxed{(\Sigma_i\mathcal{I}_i)_{lo}}|_\delta,(\Sigma_i\mathcal{I}_i)_{hi},\ldots] \\
Iplus & [\mathcal{I}+\Sigma_i\mathcal{I}_i]_{AB}[\ldots,\_|_{\delta+2},\_,(\Sigma_i\mathcal{I}_i)_{lo}|_\delta,(\Sigma_i\mathcal{I}_i)_{hi},\ldots]
\end{array}$$

□ 5.8

5.9 Correctness of single index, $\mathcal{O}_{E1}$

Given an expression ϵ, and compile-time constants lb and ne, we are required to show that the template results in $ne*(\epsilon - lb)$ in the AB registers.

We assume the induction hypotheses

1. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating the expression ϵ is left in AB: that on exit from code block Ie memory is
$[\epsilon]_{AB}$

2. *uminus* (proved in section 4.47) : If on entry to *Iminus*, memory is $[v_{hi}, v_{lo}][\_,\_][u_{lo}|_{\delta}, u_{hi}]$, then on exit it is $[(u-v)_{hi}, (u-v)_{lo}][\_,\_][u_{lo}|_{\delta}, u_{hi}]$.
3. 4.48 : If on entry to *Imul*, memory is $[v_{hi}, v_{lo}][\_,\_][u_{lo}|_{\delta}, u_{hi}]$, then on exit it is $[(u * v)_{hi}, (u * v)_{lo}][\_,\_][u_{lo}|_{\delta}, u_{hi}]$.

case 1: $ne = 1$

$$\begin{array}{ll}
 & [\_,\_]_{AB}[\ldots,\_|_{\delta},\_,\ldots] \\
Ie_{\delta} & \text{(hyp 1)} \\
 & [\epsilon]_{AB}[\ldots,\_|_{\delta},\_,\ldots] \\
xsdrs\ \delta & [\epsilon]_{AB}[\ldots,\boxed{\_}|_{\delta},\_,\ldots] \\
\mathcal{O}_{s\mathcal{T}} & [\epsilon]_{AB}[\ldots,\boxed{\epsilon_{lo}}|_{\delta},\epsilon_{hi},\ldots] \\
\mathcal{O}_{l\mathcal{T}i}\ lb & [lb]_{AB}[\ldots,\boxed{\epsilon_{lo}}|_{\delta},\epsilon_{hi},\ldots] \\
Iminus & \text{(hyp 2)} \\
 & [\epsilon - lb]_{AB}[\ldots,\epsilon_{lo}|_{\delta},\epsilon_{hi},\ldots]
\end{array}$$

case 2: $ne > 1$

$$\begin{array}{ll}
 & [\_,\_]_{AB}[\ldots,\_|_{\delta},\_,\ldots] \\
Ie & (n = 1 \text{ code}) :[\epsilon - lb]_{AB}[\ldots,\epsilon_{lo}|_{\delta},\epsilon_{hi},\ldots] \\
xsdrs\ \delta & [\epsilon - lb]_{AB}[\ldots,\boxed{\epsilon_{lo}}|_{\delta},\epsilon_{hi},\ldots] \\
\mathcal{O}_{s\mathcal{T}} & [\epsilon - lb]_{AB}[\ldots,\boxed{(\epsilon - lb)_{lo}}|_{\delta},(\epsilon - lb)_{hi},\ldots] \\
\mathcal{O}_{l\mathcal{T}i}\ ne & [ne]_{AB}[\ldots,\boxed{(\epsilon - lb)_{lo}}|_{\delta},(\epsilon - lb)_{hi},\ldots] \\
Imul & \text{(hyp 3)} \\
 & [ne * (\epsilon - lb)]_{AB}[\ldots,(\epsilon - lb)_{lo}|_{\delta},(\epsilon - lb)_{hi},\ldots]
\end{array}$$

□ 5.9

5.10 Correctness of actual parameters, $\mathcal{O}_{AP^*}$

We are required to show that the the result of evaluating the actual parameters is to leave their values in the correct formal parameter locations.

We assume the induction hypotheses

1. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating I is to leave the value of the value paramter in AB.
2. $\mathcal{O}_{Eaddr}$ (proved in section 5.11) : the result of evaluating I is to leave the start address of the reference parameter in AB.
3. xPa : the linker determines the correct formal parameter address, α

$$
\begin{array}{ll}
 & [-,-]_{AB}[\ldots,-|_{\alpha},-,\ldots] \\
I & \text{(hyp 1 or 2)} \\
 & [v]_{AB}[\ldots,-|_{\alpha},-,\ldots] \\
xPa & \text{(hyp 3)} \\
 & [v]_{AB}[\ldots,\boxed{-}|_{\alpha},-,\ldots] \\
\mathcal{O}_{s\tau}\ \tau & [v]_{AB}[\ldots,\boxed{v_{lo}}|_{\alpha},v_{hi},\ldots]
\end{array}
$$

This shows that a single parameter is correctly stored; the case for multiple parameters simply iterates along the parameter list.

□ 5.10

5.11 Correctness of location of identifier, $\mathcal{O}_{Eaddr}$

There are four cases, each very simple:

1. indirect formal parameters : update the D register from the contents of the relevant memory location (which location found from the environment)
2. 'AT' variables : update the D register with the absolute address
3. ordinary declared variables : update the D register with the address of the variable (found from the environment) relative to the heap
4. declared imported variables : the linker fixes up the correct location

□ 5.11

□ 5

6 Correctness of Statements

6.1 Statement proof obligation

Induction hypothesis: the Pasp and compilation environments correspond beforehand, as do the Pasp and Asp states.

Obligation: the Pasp and Asp states correspond afterwards. That is, executing the template results in the corresponding change to the Asp state.

6.2 Correctness of Skip

We are required to show that the execution of the *skip* template has no effect.

The *skip* template is a single `nop` instruction, which is the identity, as required. (Note that this does have the effect of a delay in practice. But timing effects are outside the scope of the Pasp specification.)

□ 6.2

6.3 Correctness of Assignment

We are required to show that the execution of the *assignment* template has the effect of storing the value of the expression in the correct variable location.

We assume the induction hypotheses

1. $\mathcal{O}_{VL}$ (proved in section 5.7) : the result of evaluating the array indexes is to leave the data address of the relevant element, α, in AB.
2. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating the expression ϵ is left in AB: that on exit from code block I memory is
$[\epsilon]_{AB}$

	$[\_,\_]_{AB}[\ldots,\_\vert_{\delta'},\_,\ldots,\_\vert_{\alpha},\_,\ldots]$
Iloc	(hyp 1)
	$[\alpha]_{AB}[\ldots,\_\vert_{\delta'},\_,\ldots,\_\vert_{\alpha},\_,\ldots]$
xsdrh δ'	$[\alpha]_{AB}[\ldots,\boxed{\_}\vert_{\delta'},\_,\ldots,\_\vert_{\alpha},\_,\ldots]$
`abstd`	$[\alpha]_{AB}[\ldots,\boxed{\alpha_{lo}}\vert_{\delta'},\alpha_{hi},\ldots,\_\vert_{\alpha},\_,\ldots]$
I	(hyp 2)
	$[\epsilon]_{AB}[\ldots,\alpha_{lo}\vert_{\delta'},\alpha_{hi},\ldots,\_\vert_{\alpha},\_,\ldots]$
`cldab`	$[\epsilon]_{AB}[\epsilon]_C[\ldots,\alpha_{lo}\vert_{\delta'},\alpha_{hi},\ldots,\_\vert_{\alpha},\_,\ldots]$
xsdrh δ'	$[\epsilon]_{AB}[\epsilon]_C[\ldots,\boxed{\alpha_{lo}}\vert_{\delta'},\alpha_{hi},\ldots,\_\vert_{\alpha},\_,\ldots]$
`abldd`	$[\alpha]_{AB}[\epsilon]_C[\ldots,\boxed{\alpha_{lo}}\vert_{\delta'},\alpha_{hi},\ldots,\_\vert_{\alpha},\_,\ldots]$
`daldab`	$[\alpha]_{AB}[\epsilon]_C[\ldots,\alpha_{lo}\vert_{\delta'},\alpha_{hi},\ldots,\boxed{\_}\vert_{\alpha},\_,\ldots]$
`abldc`	$[\epsilon]_{AB}[\epsilon]_C[\ldots,\alpha_{lo}\vert_{\delta'},\alpha_{hi},\ldots,\boxed{\_}\vert_{\alpha},\_,\ldots]$
$\mathcal{O}_{s\tau}\ \tau$	$[\epsilon]_{AB}[\epsilon]_C[\ldots,\alpha_{lo}\vert_{\delta'},\alpha_{hi},\ldots,\boxed{\epsilon_{lo}}\vert_{\alpha},\epsilon_{hi},\ldots]$

□ 6.3

6.4 Correctness of If

We are required to show that the execution of the *if* template has the effect of performing the first statement if the expression evaluates to *true*, otherwise of performing the second statement.

We assume the induction hypotheses

1. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating the (boolean) expression ϵ is left in AB: that on exit from code block *IE* memory is $[\epsilon]_{AB}$
2. $\mathcal{O}_S\ \gamma'$ (proved in section 6) : the result of evaluating the statement γ' is to leave memory updated appropriately.
3. $\mathcal{O}_S\ \gamma$ (proved in section 6) : the result of evaluating the statement γ is to leave memory updated appropriately.

		$[\_,\_]_{AB}[\ldots]$
	IE	(hyp 1)
		$[\epsilon]_{AB}[\ldots]$
	`rrb`	$[\_,\_]_{AB}[\_,\epsilon(\mathit{true}\text{ or }\mathit{false})]_{ab}[\ldots]$
	$\mathit{xrjbl}\ l2$	**if** $\epsilon = \mathit{true}$ **then** $\mathit{goto}\ l2$
		$[\_,\_]_{AB}[\_,\mathit{false}]_{ab}[\ldots]$
	I'	(hyp 2)
		$[\_,\_]_{AB}[\epsilon = \mathit{false} \wedge \text{state for } \gamma']$
	$\mathit{xrg}\ l3$	$\mathit{goto}\ l3$
$l2$		
		(from $\mathit{goto}\ l2$)$[\_,\_]_{AB}[\_,\mathit{true}]_{ab}[\ldots]$
	I	(hyp 3)
		$[\_,\_]_{AB}[\epsilon = \mathit{true} \wedge \text{state for } \gamma]$
$l3$		
		(from above)$[\_,\_]_{AB}[\epsilon = \mathit{true} \wedge \gamma]$
		(from $\mathit{goto}\ l3$)$[\_,\_]_{AB}[\epsilon = \mathit{false} \wedge \gamma']$
		(simplify)$[\_,\_]_{AB}[$**if** ϵ **then** γ **else** $\gamma']$

□ 6.4

6.5 Correctness of Case

We are required to show that the execution of the *case* template has the effect of performing the equivalent Pasp case staement.

This is clearly the case, since the operation template is defined using the same translation as is the Pasp dynamic semantics.

□ 6.5

6.6 Correctness of Loop

We are required to show that the execution of the *loop* template has the effect of performing the body statement repeatedly while the expression evaluates to *true*.

We assume the induction hypotheses

1. $\mathcal{O}_E$ (proved in section 5) : the result of evaluating the (boolean) expression ϵ is left in AB: that on exit from code block IE memory is $[\epsilon]_{AB}$
2. $\mathcal{O}_S\ \gamma$ (proved in section 6) : the result of evaluating the statement γ is to leave memory updated appropriately.

		$[\_,\_]_{AB}[\ldots]$
l		
		(from above$[_,_]_{AB}$[state for γ^0] (from *goto* l)$[\_,\_]_{AB}$[state for γ^m]
	IE	(hyp 1) $[\epsilon]_{AB}[\gamma^n]$
	`rrb`	$[\_,\_]_{AB}[\_,\epsilon(\mathit{true}\text{ or }\mathit{false})]_{ab}[\gamma^n]$
	`ncb`	$[\_,\_]_{AB}[\_,\neg\ \epsilon(\mathit{true}\text{ or }\mathit{false})]_{ab}[\gamma^n]$
	$xrjbl\ lS$	**if** $\neg\ \epsilon = \mathit{true}$ **then** *goto lS* $[\_,\_]_{AB}[\_,\epsilon = \mathit{true}]_{ab}[\gamma^n]$
	IS	(hyp 2) $[\_,\_]_{AB}[\epsilon = \mathit{true} \wedge \text{state for } \gamma^{n+1}]$
	$xrg\ l$	*goto l*
lS		
		(from *goto lS*)$[\_,\_]_{AB}[\_,\epsilon = \mathit{false}]_{ab}[\gamma^m]$ (simplify)$[\_,\_]_{AB}$[**while** ϵ **do** γ]

□ 6.6

6.7 Correctness of Procedure Call

We are required to show that on return from the procedure call, the state corresponds to the execution of the body statement γ.

We assume the induction hypotheses

1. $\mathcal{O}_{AP^*}$ (proved in section 5.10) : the result of evaluating the actual parameters is to store their values at the correct formal parameter locations

2. *xCall* (proved in section 8.3) : the result of calling the procedure is to save the return label, evaluate the variable initialisations and body statement, then load the return label, and jump to it.

$$
\begin{array}{lll}
 & & [\_,\_]_{AB} \\
 & I & (\text{hyp } 1) \\
 & & [\_,\_]_{AB}[\text{parameters stored}\ldots], \_, \_|_\delta, \_, \ldots] \\
 & \mathit{xll}\ l' & [l']_{AB}[\text{parameters stored}\ldots] \\
 & \texttt{cldab} & [l']_{AB}[l']_C[\text{parameters stored}\ldots]\mathit{ots}] \\
 & \mathit{xCall}\ \xi & (\text{hyp } 2) \\
 & & [\_,\_]_{AB}[l']_C[\text{parameters stored}\ldots, l', \ldots] \\
 & & [\_,\_]_{AB}[\text{parameters stored}\ldots, l', \text{state for } \gamma] \\
 & & [l']_{AB}[\text{parameters stored}\ldots, l', \text{state for } \gamma] \\
 & & \mathit{goto}\ l' \\
l' & & [l']_{AB}[\text{parameters stored}\ldots, l', \text{state for } \gamma]
\end{array}
$$

□ 6.7

□ 6

7 Correctness of Simple Declarations

7.1 Simple Declaration proof obligation

The proof obligation for the simple declaration part of the *Prog* is that they correctly update the corresponding compiler environments appropriately (and state in the case of initialisation).

If the environments match beforehand (induction hypothesis), then the after Pasp environment updated with the simple declarations matches the after compiler environment updated with the compiled simple declarations. (Here we use $\mathcal{R}_\rho$ to represent the environment retreive relation.)

$$\begin{array}{l}
\rho t, \rho t' : \mathit{EnvMTrace};\ \rho ot, \rho ot' : \mathit{EnvOTrace};\ \mu o, \mu o' : \mathit{MO};\\
\qquad\qquad \Delta \mathit{SD} : \mathrm{seq}\, \mathit{SIMPLE\_DECL};\ B : \mathit{STACK}\ |\\
\qquad \mathcal{R}_\rho\ \rho t = \rho ot \land \rho t' = \mathcal{M}_{SD^*}\ \Delta SD\ B\ \rho t\\
\qquad \land\ (\rho ot', \mu o') = \mathcal{O}_{SD^*}\ \Delta SD\ B\ (\rho ot, \mu o)\\
\vdash\\
\mathcal{R}_\rho\ \rho t' = \rho ot'
\end{array}$$

It suffices to show the case of a single simple declaration. So

$$\begin{array}{l}
\rho t, \rho t' : \mathit{EnvMTrace};\ \rho ot, \rho ot' : \mathit{EnvOTrace};\ \mu o, \mu o' : \mathit{MO};\\
\qquad\qquad \delta : \mathit{SIMPLE\_DECL};\ B : \mathit{STACK}\ |\\
\qquad \mathcal{R}_\rho\ \rho t = \rho ot \land \rho t' = \mathcal{M}_{SD}\ \delta\ B\ \rho t\\
\qquad \land\ (\rho ot', \mu o') = \mathcal{O}_{SD}\ \delta\ B\ (\rho ot, \mu o)\\
\vdash\\
\mathcal{R}_\rho\ \rho t' = \rho ot'
\end{array}$$

We break this into the three SD cases

7.2 Constant declaration

$$\begin{array}{l}
\rho t, \rho t' : \mathit{EnvMTrace};\ \rho ot, \rho ot' : \mathit{EnvOTrace};\ \mu o : \mathit{MO};\\
\qquad\quad \mathit{ConstDecl};\ B : \mathit{STACK}\ |\\
\quad \mathcal{R}_\rho\ \rho t = \rho ot\\
\quad \wedge\ \rho t' = \mathit{update}(\rho t, B, \{\xi \mapsto \mathit{cval}\})\\
\quad \wedge\ \rho ot' = \mathit{update}(\rho ot, B, \{\xi \mapsto \mathit{opExport}\ \kappa\})\\
\vdash\\
\mathcal{R}_\rho\ \rho t' = \rho ot'
\end{array}$$

Substitute for $\rho t'$, $\rho ot'$; simplify

$$\begin{array}{l}
\rho t, \rho t' : \mathit{EnvMTrace};\ \rho ot, \rho ot' : \mathit{EnvOTrace};\ \mu o : \mathit{MO};\\
\qquad\quad \mathit{ConstDecl};\ B : \mathit{STACK}\\
\vdash\\
\mathit{const}^{\sim}(\mathit{lookup}(\xi, \langle \mathit{last}\ B\rangle, \rho\tau t0))).v = \kappa
\end{array}$$

This is precisely the property the type checking semantics $\mathcal{T}_{NC}$ establishes.

□ 7.2

7.3 Type definition

$$\begin{array}{l}
\rho t, \rho t' : \mathit{EnvMTrace};\ \rho ot, \rho ot' : \mathit{EnvOTrace};\ \mu o, \mu o' : \mathit{MO};\\
\qquad\quad \delta : \mathit{TYPE\_DEF};\ B : \mathit{STACK}\ |\\
\quad \mathcal{R}_\rho\ \rho t = \rho ot \wedge \rho t' = \rho t\\
\quad \wedge\ (\rho ot', \mu o') = (\rho ot, \mu o)\\
\vdash\\
\mathcal{R}_\rho\ \rho t' = \rho ot'
\end{array}$$

This is trivially true.

□ 7.3

7.4 Variable declaration

We are required to show that variable declaration updates the environment apprpriately.

This essentially reduces to showing that the correct number of locations are allocated to the variable; this follows directly from *newVarAddr* using the same *allocate* function as does the Pasp definition.

□ 7.4

7.5 Correctness of variable declaration, $\mathcal{O}_{SD^*}$

Follows directly from the correctness of the three individual cases: constant, type definition, and variable.

□ 7.5

7.6 Correctness of variable initialisation, $\mathcal{O}_{SDI^*}$

We are required to show that simple declaration initialisations update the state apprpriately.

The only case where anything interesting happens is for variable initialisations, $\mathcal{O}_{VI}$. There are four cases.

We assume the induction hypothesis

1. $\mathcal{O}_{Eaddr}$ (proved in section 5.11) : the result of evaluating *Iloc* is to leave the start address, α, of the (array or simple) variable in AB.

- **Case 1:** $\#V = 0$; no initialisation.

  No code is generated in this case.

  □

- **Case 2:** $\#V = 1 \wedge \#L = 1$; simple variable initialisation.

  We are required to show the value is stored in the variable's location.

$$\begin{array}{ll}
 & [\_,\_]_{AB}[\ldots,\_|_{\alpha},\_,\ldots] \\
\textit{Iloc} & (\mathit{hyp}1) \\
 & [\alpha]_{AB}[\ldots,\_|_{\alpha},\_,\ldots] \\
\texttt{daldab} & [\alpha]_{AB}[\ldots,\boxed{\_}|_{\alpha},\_,\ldots] \\
\mathcal{O}_{l\tau i}\ v1 & [v1]_{AB}[\ldots,\boxed{\_}|_{\alpha},\_,\ldots] \\
\mathcal{O}_{s\tau}\ \tau' & [v1]_{AB}[\ldots,\boxed{v1_{lo}}|_{\alpha},v1_{hi},\ldots]
\end{array}$$

□

- **Case 3:** $\#V = 1 \wedge \#L > 1$; block array initialisation. We are required to show the value is stored in each element of the array.

$$\begin{array}{ll}
 & [\_,\_]_{AB}[\ldots,\_|_{\alpha},\_,\ldots,\_,\_,\ldots] \\
\textit{Iloc} & (\mathit{hyp}1) \\
 & [\alpha]_{AB}[\ldots,\_|_{\alpha},\_,\_,\_,\ldots,\_,\_,\ldots] \\
\texttt{daldab} & [\alpha]_{AB}[\ldots,\boxed{\_}|_{\alpha},\_,\_,\_,\ldots,\_,\_,\ldots] \\
\mathcal{O}_{l\tau i}\ v1 & [v1]_{AB}[\ldots,\boxed{\_}|_{\alpha},\_,\_,\_,\ldots,\_,\_,\ldots] \\
 & \text{(repeat } \#L-1 \text{ times)} \\
\mathcal{O}_{s\tau}\ \tau' & \quad\quad [v1]_{AB}[\ldots,\boxed{v1_{lo}}|_{\alpha},v1_{hi},\_,\_,\ldots,\_,\_,\ldots] \\
\texttt{dpi}\ \mathit{sizeof}\ \tau' & \quad\quad [v1]_{AB}[\ldots,v1_{lo}|_{\alpha},v1_{hi},\boxed{\_},\_,\ldots,\_,\_,\ldots] \\
 & \text{(end repeat)} \\
 & [v1]_{AB}[\ldots,v1_{lo}|_{\alpha},v1_{hi},v1_{lo},v1_{hi},\ldots,\boxed{\_},\_,\ldots] \\
\mathcal{O}_{s\tau}\ \tau' & [v1]_{AB}[\ldots,v1_{lo}|_{\alpha},v1_{hi},v1_{lo},v1_{hi},\ldots,\boxed{v1_{lo}},v1_{hi},\ldots]
\end{array}$$

□

- **Case 4:** $\#V > 1 \wedge \#L > 1$; array initialisation. We are required to show the values are stored in corresponding elements of the array.

	$[\_,\_]_{AB}[\ldots,\_\vert_{\alpha},\_,\ldots,\_,\_,\ldots]$
Iloc	$(hyp1)$
	$[\alpha]_{AB}[\ldots,\_\vert_{\alpha},\_,\_,\_,\ldots,\_,\_,\ldots]$
`daldab`	$[\alpha]_{AB}[\ldots,\boxed{\_}\vert_{\alpha},\_,\_,\_,\ldots,\_,\_,\ldots]$
	(repeat $\#L-1$ times)
$\mathcal{O}_{l\tau i}\ v1$	$[v1]_{AB}[\ldots,\boxed{\_}\vert_{\alpha},\_,\_,\_,\ldots,\_,\_,\ldots]$
$\mathcal{O}_{s\tau}\ \tau'$	$[v1]_{AB}[\ldots,\boxed{v1_{lo}}\vert_{\alpha},v1_{hi},\_,\_,\ldots,\_,\_,\ldots]$
`dpi` *sizeof* τ'	$[v1]_{AB}[\ldots,v1_{lo}\vert_{\alpha},v1_{hi},\boxed{\_},\_,\ldots,\_,\_,\ldots]$
	(end repeat)
	$[v_{n-1}]_{AB}[\ldots,v1_{lo}\vert_{\alpha},v1_{hi},v2_{lo},v2_{hi},\ldots,\boxed{\_},\_,\ldots]$
$\mathcal{O}_{l\tau i}\ vn$	$[vn]_{AB}[\ldots,v1_{lo}\vert_{\alpha},v1_{hi},v2_{lo},v2_{hi},\ldots,\boxed{\_},\_,\ldots]$
$\mathcal{O}_{s\tau}\ \tau'$	$[vn]_{AB}[\ldots,v1_{lo}\vert_{\alpha},v1_{hi},v2_{lo},v2_{hi},\ldots,\boxed{vn_{lo}},vn_{hi},\ldots]$

□

□ 7.6

□ 7

8 Correctness of Routine Declarations

The proof obligation for the routine declaration part of the *Prog* is that they update the corresponding environments appropriately.

8.1 Routine Declaration proof obligation

If the environments match beforehand (induction hypothesis), then the after Pasp environment updated with the routine declarations matches the after compiler environment updated with the compiled routine declarations.

8.2 Correctness of Procedure declaration

We are required to show that the effect of declaring the procedure is to leave the environment appropriately. This includes showing that the effect of calling the procedure is to save the return label passed in C, evaluate the body (variable initialisations and statement), load the return label, and jump to it.

We assume the induction hypotheses

1. $\mathcal{O}_{PMD^*}\ \Pi$ (proved in section 8.4) : the result of evaluating the formal parameters is to leave the environment updated appropriately.
2. $\mathcal{O}_B\ \beta$ (proved in section 8.5) : the result of evaluating the body β is to leave memory updated appropriately.

The correctness of the change to the environment is given by induction hypothesis 1. The correctness of the code is as follows:

$xProc(\xi, l)$ (linker provides correct label to be jumped to)
$[\_,\_]_{AB}[l']_C[\text{parameters}\ldots,\_|_{h\delta},\_,\ldots]$

$xsdrh\ \delta$ $[\_,\_]_{AB}[l']_C[\text{parameters}\ldots,\boxed{\_}|_{h\delta},\_,\ldots]$

`cstd` $[\_,\_]_{AB}[l']_C[\text{parameters}\ldots,\boxed{l'_{lo}}|_{h\delta},l'_{lo},\ldots]$

I (hyp 2)
$[\_,\_]_{AB}[\_]_C[\text{parameters}\ldots,l'_{lo}|_{h\delta},l'_{lo},\ldots,\text{state for }\gamma]$

$xsdrh\ \delta$ $[\_,\_]_{AB}[\_]_C[\text{parameters stored}\ldots,\boxed{l'_{lo}}|_{h\delta},l'_{lo},\ldots,\text{state for }\gamma]$

`abldd` $[l']_{AB}[\_]_C[\text{parameters stored}\ldots,\boxed{l'_{lo}}|_{h\delta},l'_{lo},\ldots,\text{state for }\gamma]$

xag *goto* l'

□ 8.2

8.3 Correctness of Function declaration

We are required to show that the effect of declaring the function is to leave the environment appropriately. This includes showing that the effect of calling the function is to save the return label passed in C, evaluate the body (variable initialisations and stamtement), load the return label, and jump to it.

We assume the induction hypothesis

1. $\mathcal{O}_{PMD^*}\ \Pi$ (proved in section 8.4) : the result of evaluating the formal parameters is to leave the environment updated appropriately.
2. $\mathcal{O}_B\ \beta$ (proved in section 8.5) : the result of evaluating the body β is to leave memory updated appropriately, including the return result stored in the heap variable at δ.

The correctness of the change to the environment is given by induction hypothesis 1, plus the declaration of a new variable address at δ' relative to the heap, for the return value. The correctness of the code is as follows:

$xProc(\xi, l)$	(linker provides correct label to be jumped to)
	$[\_,\_]_{AB}[l']_C[\text{parameters}\ldots,\_|_{h\delta},\_,\ldots,\_|_{h\delta'},\_,\ldots]$
$xsdrh\ \delta'$	$[\_,\_]_{AB}[l']_C[\text{parameters}\ldots,\_|_{h\delta},\_,\ldots,\boxed{\_}|_{h\delta},\_,\ldots]$
`cstd`	$[\_,\_]_{AB}[l']_C[\text{parameters}\ldots,\_|_{h\delta},\_,\ldots,\boxed{l'_{lo}}|_{h\delta'},l'_{lo},\ldots]$
I	(hyp s)
	$[\_,\_]_{AB}[\_]_C[\text{parameters}\ldots,v_{lo}|_{h\delta},v_{hi},\ldots,l'_{lo}|_{h\delta'},l'_{lo},\ldots,\text{state for }\gamma]$
$xsdrh\ \delta$	$[\_,\_]_{AB}[\_]_C[\text{parameters}\ldots,\boxed{v_{lo}}|_{h\delta},v_{lo},\ldots,l'_{lo}|_{h\delta'},l'_{lo},\ldots,\text{state for }\gamma]$
`cstd`	$[\_,\_]_{AB}[v]_C[\text{parameters}\ldots,\boxed{v_{lo}}|_{h\delta},v_{lo},\ldots,l'_{lo}|_{h\delta'},l'_{lo},\ldots,\text{state for }\gamma]$
$xsdrh\ \delta'$	$[\_,\_]_{AB}[v]_C[\text{parameters}\ldots,v_{lo}|_{h\delta},v_{hi},\ldots,\boxed{l'_{lo}}|_{h\delta},l'_{lo},\ldots,\text{state for }\gamma]$
`abldd`	$[l']_{AB}[v]_C[\text{parameters}\ldots,v_{lo}|_{h\delta},v_{hi},\ldots,\boxed{l'_{lo}}|_{h\delta},l'_{lo},\ldots,\text{state for }\gamma]$
xag	*goto* l'

□ 8.3

□ 8

8.4 Correctness of Formal Parameters, $\mathcal{O}_{PMD^*}$

We are required to show that the result of evaluating the formal parameters is to leave the environment updated appropriately.

A reference parameter, $c = \mathit{ref}$, is allocated a new variable address with type *indirectAddr*, to hold the reference. A value parameter, $c \neq \mathit{ref}$, is allocated a new varaible address suitable for its type. A sequence of parameters are each individually allocated their space.

□ 8.4

8.5 Correctness of body, $\mathcal{O}_B$

We are required to show that the effect of evaluating the body (variable initialisations and stamtement), is to leave the environment and memory updated appropriately.

We assume the induction hypotheses

1. $\mathcal{O}_{SD^*}\ \Delta$ (proved in section 7.5) : the result of evaluating the local declarations is to leave the environment and memory updated appropriately.
2. $\mathcal{O}_{SDI^*}\ \Delta$ (proved in section 7.6) : the result of initialising the local declarations is to leave the memory updated appropriately.
3. $\mathcal{O}_S\ \gamma$ (proved in section 6) : the result of evaluating the statement is to leave the memory updated appropriately.

The template is simply the concatenation of these three effects, as required.

□ 8.5

□ 8

9 Correctness of Prog

Having a simplified expression for the linked modules, we next expand out the definition of $\mathcal{M}_P$, which leads to a proof of correctness of *Prog* in the following steps:

- simple declarations
- procedure and function declarations
- simple declaration initialisation
- main body

Pasp declarations change the Pasp environment ρt; the corresponding Asp environement ρo is built up and held by the compiler.

We show that these declaration environments correspond. Then we show that the initialisation and body code executions correspond under the induction hypothesis that the environments correspond.

A Knuth's unsigned division algorithm

A.1 Introduction

This section explains the derivation of the unsigned division template, based on the algorithm given in *Knuth, Seminumerical Algorithms, pp 257–8.*

A.2 General algorithm

We wish to form the unsigned integer division quotient $q = u \operatorname{div} v$, where u is the $m+n$ digit, base b, number $\langle u_1 \ldots u_{m+n}\rangle_b$ (hence the most significant byte is u_1), and v is the n digit, base b, number $\langle v_1 \ldots v_n\rangle_b$. When $v_1 \neq 0$, the algorithm is:

```
D1  $d := b \operatorname{div} (v_1 + 1);$
    $\langle u_0 u_1 \ldots u_{m+n}\rangle := \langle u_1 \ldots u_{m+n}\rangle * d;$
    $\langle v_1 \ldots v_n\rangle := \langle v_1 \ldots v_n\rangle * d;$

D2  $j := 0;$

D3  $\hat{q} :=$ if $u_j = v_1$ then $b - 1$ else $(u_j * b + u_{j+1}) \operatorname{div} v_1;$
    if $0 < (v_1 * b + v_2) * \hat{q} - ((u_j * b + u_{j+1}) * b + u_{j+2})$
    then $\hat{q} := \hat{q} - 1;$
           if $0 < (v_1 * b + v_2) * \hat{q} - ((u_j * b + u_{j+1}) * b + u_{j+2})$ then $\hat{q} := \hat{q} - 1;$

D4  $\langle u_j \ldots u_{j+n}\rangle := \langle u_j \ldots u_{j+n}\rangle - \hat{q} * \langle v_1 \ldots v_n\rangle;$
    if$\langle u_j \ldots u_{j+n}\rangle < 0$
    then $\langle u_j \ldots u_{j+n}\rangle := \langle u_j \ldots u_{j+n}\rangle + b^{n+1};$  $B := true$
    else $B := false;$

D5  $q_j := \hat{q};$
    if $B$ then goto $D6$ else goto $D7;$

D6  $q_j := q_j - 1;$
    $\langle u_j \ldots u_{j+n}\rangle := \langle u_j \ldots u_{j+n}\rangle + \langle 0 v_1 \ldots v_n\rangle;$        ignore carry
```

```
D7 $j := j + 1$;
   **if** $j \le m$ **then** *goto D*3;
```

A.3 Two byte algorithm

A.3.1 Specialised

We can specialise this algorithm for DeCCo's 2-byte integers: we have $m = 0$; $n = 2$; $b = 256$; $u = \langle u_1 u_2 \rangle_{256}$; $v = \langle v_1 v_2 \rangle_{256}$. When $v_1 \neq 0$, we know that q is a single byte, and the algorithm specialises to:

```
D1 $d := 256 \operatorname{div} (v_1 + 1)$;
   $\langle u_0 u_1 u_2 \rangle := \langle u_1 u_2 \rangle * d$;
   $\langle v_1 v_2 \rangle := \langle v_1 v_2 \rangle * d$;

D3 $q :=$ **if** $u_0 = v_1$ **then** $255$ **else** $(u_0 * 256 + u_1) \operatorname{div} v_1$;
   $tmp := \langle v_1 v_2 \rangle * q - \langle u_0 u_1 u_2 \rangle$
   **if** $tmp > 0$
   **then if** $tmp - \langle v_1 v_2 \rangle > 0$
          **then** $q := q - 2$; **else** $q := q - 1$;

D4 **if** $\langle u_0 u_1 u_2 \rangle - q * \langle v_1 v_2 \rangle < 0$ **then** $q := q - 1$;
```

A.3.2 Simplified

We can now see the wood for the trees. We know that, if $q = u \operatorname{div} v$, then $u = q * v + rem$, where $0 \le rem < b$. Knuth's arguments give us that q is *never* too small, and may be too big by at most 2. Exhaustive testing of the two-byte algorithm shows that it is out by at most one in this case.

So the full division algorithm can be written as:

```
**if** $v_1 = 0$ **then** use single byte division **else**
**if** $v_1 < 128$
**then** $d := 256 \text{ div } (v_1 + 1);$            assert $2 \leq d \leq 128$
     $\langle u_0 u_1 u_2 \rangle := \langle u_1 u_2 \rangle * d;$
     $\langle v_1 v_2 \rangle := \langle v_1 v_2 \rangle * d;$            assert $128 \leq v_1$
**else** $u_0 := 0;$
$q :=$ **if** $u_0 = v_1$ **then** 255 **else** $(u_0 * 256 + u_1) \text{ div } v_1;$     assert $q \leq 255$
$rem := \langle u_0 u_1 u_2 \rangle - q * \langle v_1 v_2 \rangle$
**if** $rem < 0$ **then** $q := q - 1;$
```

A.3.3 Division by a single byte

If $v_1 = 0$, the simpler algorithm for division by the single byte v_2 is:

```
$q_1 := u_1 \text{ div } v_2;$
$rem := u_1 \text{ mod } v_2$
$q_2 := (rem * 256 + u_2) \text{ div } v_2$
```

B Correctness of specialised div algorithm

Here we show that our specialisation of Knuth's div algorithm, with the added assumption that q is too big by at most one, is correct.

B.1 Recasting in Z

The initial values to be divided are $\overline{u}$ and $\overline{v}$. These are scaled by a factor of d to make $\overline{v}_1$ large enough. A first guess at the result, $\hat{q}$, is made. It is adjusted if necessary, and the final result is q.

Decl

$$\begin{array}{l}
\overline{u}, \overline{v}, v : \mathit{WORD}; \\
u : \mathbb{N} \\
d, q, \hat{q}, u_0, u_1, u_2, \overline{v}_1, v_1, v_2 : \mathit{BYTE} \\
\hline
\overline{v}_1 = \overline{v} \text{ div } 256 \neq 0 \\
d = 256 \text{ div } (\overline{v}_1 + 1) \\
u = (u_0 * 256 + u_1) * 256 + u_2 = d * \overline{u} \\
v = v_1 * 256 + v_2 = d * \overline{v} \\
\hat{q} = \mathit{min}\{255, (u_0 * 256 + u_1) \text{ div } v_1\} \\
q = \overline{u} \text{ div } \overline{v}
\end{array}$$

$$\mathit{Decl} \vdash q = (\textbf{if } u - \hat{q} * v < 0 \textbf{ then } \hat{q} - 1 \textbf{ else } \hat{q})$$

B.2 Lemma 1: $\hat{q}$ is never too small

$$\mathit{Decl} \vdash q \leq \hat{q}$$

Proof of lemma 1 is given in Knuth

□

B.3 Lemma 2: d is a single byte

$$Decl \vdash d \leq 128$$

Proof of lemma 2

$$d = 256 \operatorname{div} (\overline{v}_1 + 1) \overline{v}_1 \neq 0$$

d is the largest it can be when $\overline{v}_1$ is the smallest, when $\overline{v}_1 = 1$. So

$$d^{max} = 256 \operatorname{div} 2 = 128$$

□

B.4 Lemma 3: v_1 is a single byte

$$Decl \vdash v_1 < 256$$

Proof of lemma 3

$$v_1 = (\overline{v} * d) \operatorname{div} 256 = ((256 * \overline{v}_1 + \overline{v}_2) * (256 \operatorname{div} (\overline{v}_1 + 1))) \operatorname{div} 256$$

This is a maximum when the remainders are zero, and $\overline{v}_2$ is a maximum, 255.

$$\begin{aligned} & v_1^{max} \\ & \quad = \frac{256 * \overline{v}_1 + 255}{\overline{v}_1 + 1} \\ & \quad = 256 - \frac{1}{\overline{v}_1 + 1} \\ & \quad < 256 \end{aligned}$$

□

B.5 Lemma 4: $\hat{q}$ is at most one too big

$$Decl \vdash 128 \leq v_1 \Rightarrow \hat{q} - q \leq 1$$

Knuth proves that $\hat{q} - q \leq 2$ in general, for multibyte division. We prove a stronger result for our two byte division.

Proof of lemma 4

Form the definition of div, we have that the following constraint on the remainder ρ:

$$a \text{ div } b = \frac{a - \rho}{b}$$
$$0 \leq \rho < b$$

We introduce a suitable ρ and $\hat{\rho}$, such that

$$q = u \text{ div } v = \frac{u - \rho}{v}$$
$$0 \leq \rho < v$$
$$\hat{q} = (u - u_2) \text{ div } (v - v_2) = \frac{u - u_2 - \hat{\rho}}{v - v_2}$$
$$0 \leq \hat{\rho} < v - v_2$$

Subtracting, we get

$$\begin{aligned} &\hat{q} - q \\ &\quad = \frac{u - u_2 - \hat{\rho}}{v - v_2} - \frac{u - \rho}{v} \\ &\quad = \frac{\rho}{v} - \frac{u * v_2 - v(u_2 + \hat{\rho})}{v - v_2} \end{aligned}$$

This difference is largest when ρ is the largest it can be, and when $\hat{\rho}$ is the smallest it can be. Hence

$$\hat{q} - q < 1 - \frac{u * v_2 - v * u_2}{v - v_2}$$

We now require that $\hat{q} - q < 2$, and derive a constraint on v_1. So we require

$$\frac{u * v_2 - v * u_2}{v - v_2} \leq 1$$

Substituting for u and v, and simplifying

$$\frac{256 * u_o * v_2 + u_1 * v_2 - u_2 * v_1}{v_1(256 * v_1 + v_2)} \leq 1$$

Rearranging

$$0 \leq 256 * v_1^2 + v_1 * v_2 + u_2 * v_1 - u_1 * v_2 - 256 * u_0 * v_2$$

This inequality is hardest to achieve when the positive terms are the largest they can be, and the negative terms are the smallest. Since u_0 and v_2 depend together of the value of d, we leave these for now, and put in the largest possible value of u_1, which is 255 (because u_1 is a $BYTE$) and the smallest possible value of u_2, which is zero.

$$0 \leq 256 * v_1^2 + v_1 * v_2 - 255 * v_2 - 256 * u_0 * v_2$$

Rearranging

$$0 \leq 256 * v_1^2 - (255 - v_1 + 256 * u_0)v_2$$

Lemma 3 gives us that $v_1 < 256$, so the subtracted term is always positive (or zero), and so the inequality is hardest to achieve when the subtracted term is large.

The largest possible values of u_0 and v_2 depend on the scaling factor d. We have

$$u_0 \leq (255 * d) \text{ div } 256 = d - 1$$
$$v_2 \leq 256 - d$$

So substituting in the largest values for u_0 and v_2, parameterised by d, gives:

$$0 \leq 256 * v_1^2 - (255 - v_1 + 256 * (d - 1))(256 - d)$$

Rearranging

$$0 \leq 256 * v_1^2 + 256 * d^2 - (256^2 + 1 + v_1) * d + 256(1 + v_1)$$

The right hand side has a minimum w.r.t. d when

$$2 * 256 * d - (256^2 + 1 + v_1) = 0$$

when

$$d = 128 + \frac{1 + v_1}{2 * 256}$$

The fractional part is always less than one. Also, by lemma 2 $d \leq 128$. So the inequality is hardest to satisfy when $d = 128$. Substituting this value for d gives:

$$0 \leq 256 * v_1^2 - 128(255 - v_1 + 256 * 127)$$

Rearranging

$$0 \leq 256 * v_1^2 + 128 * v_1 - 128(256 * 128 - 1)$$

Equality occurs when

$$v_1 = \frac{-128 \pm \sqrt{128^2 + 4 * 256 * 128(256 * 128 - 1)}}{2 * 256}$$

Since v_1 is a $BYTE$, and so cannot be negative, equality occurs when

$$v_1 = 127.75\ldots$$

Clearly, from the shape of the quadratic in v_1, the inequality holds for all v_1 larger than this. So we have derived the required constraint on v_1:

$$128 \geq v_1 \Rightarrow \hat{q} - q \leq 1$$

□

B.6 Lemma 5: v_1 is big enough

$$Decl \vdash 128 \leq v_1$$

Proof of lemma 5

$$\begin{aligned}
&v = \overline{v} * d\\
&256 * v_1 + v_2 = (256 * \overline{v}_1 + \overline{v}_2) * d\\
&256 * v_1 + v_2 = 256(\overline{v}_1 * d + (\overline{v}_2 * d) \operatorname{div} 256) + (\overline{v}_2 * d) \bmod 256\\
&v_1 = \overline{v}_1 * d + (\overline{v}_2 * d) \operatorname{div} 256
\end{aligned}$$

The r.h.s is smallest when $\overline{v}_2 = 0$.

$$\begin{aligned}
&v_1 \geq \overline{v}_1 * d\\
&v_1 \geq \overline{v}_1 * (256 \operatorname{div} (\overline{v}_1 + 1))\\
&v_1 \geq \overline{v}_1 * \frac{256 - \overline{\rho}}{\overline{v}_1 + 1}
\end{aligned}$$

The r.h.s is smallest when the remainder is larges, $\hat{\rho} < \overline{v}_1 + 1$.

$$v_1 > \frac{256 * \overline{v}_1}{\overline{v}_1 + 1} - 1$$

The r.h.s. is monotonic increasing with a minimum at $\overline{v}_1 = 1$ (remember, $\overline{v}_1 > 0$). Hence

$$\begin{aligned}
&v_1 > 127\\
&v_1 \geq 128
\end{aligned}$$

□

B.7 Proof of algorithm

Lemmas 1, 4 and 5 give us

$$Decl \vdash 0 \leq \hat{q} - q \leq 1$$

where $\hat{q}$ is our estimate, and q is the correct result. Even with scaling by d, we have

$$q = \overline{u} \text{ div } \overline{v} = u \text{ div } v$$

So we know the remainder

$$\mathit{rem} = u - q * v \geq 0$$

If the remainder similarly calculated for $\hat{q}$ is negative, then we know $\hat{q}$ is one too big, otherwise we know it is correct. So

$$q = (\textbf{if } u - \hat{q} * v < 0 \textbf{ then } \hat{q} - 1 \textbf{ else } \hat{q})$$

□

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.