Architecture Specification Language

Readers' Guide

Document number	111069
Document version	A.a
Document confidentiality	Non-Confidential
Date of issue	29 October 2025

Release information

The following releases of this document have been made.

Date	Version	Changes
29/Oct/2025	A.a	• First release of the Architecture Specification Language (ASL) Readers' Guide.

Non-Confidential Proprietary Notice

PRE-20349

8 March 2024

Contents

Architecture Specification Language Readers' Guide

Contents

Chapter 1
Introduction

The *Arm® Architecture Reference Manual, for A-profile architecture* (ARM DDI 0487) uses code snippets to provide precise descriptions of some areas of the Arm® architecture, including the decoding and operation of all valid instructions. This document provides a readers' guide for these code snippets, and defines some standard library procedures and functions that are used by the snippets.

Note

The code snippets used to describe an instruction are often referred to as the "pseudocode for that instruction".

These snippets used to be written in an informal pseudocode, but are now written in a language known as ASL1. A full formal language definition of ASL1 can be found online. This document is an informal guide to help readers understand the ASL1 code snippets as used in the *Arm® Architecture Reference Manual, for A-profile architecture* (ARM DDI 0487). It contains the following chapters:

- About ASL1. An introduction to ASL1.
- Types. A readers' guide to ASL1 types.
- Expressions. A readers' guide to ASL1 expressions.
- Declarations. A readers' guide to ASL1 declarations.
- Subprograms. A readers' guide to ASL1 subprograms.
- Statements and control structures. A readers' guide to ASL1 statements.
- Standard library functions. Standard library functions that are used by instruction pseudocode in the *Arm® Architecture Reference Manual, for A-profile architecture* (ARM DDI 0487).
- ASL1 index. An index to this readers' guide.

Chapter 2
About ASL1

This chapter introduces the ASL1 language, which is used to write instruction pseudocode in the *Arm® Architecture Reference Manual, for A-profile architecture* (ARM DDI 0487).

The ASL1 language is:

- Imperative with mutable state: ASL1 code consists of a series of steps, which may modify an ambient global state.
- Strongly, statically typed: types are checked at compile-time, and it is a compile-time error to use types inconsistently.
- First-order: functions are defined globally, and cannot be defined anonymously or passed as data.
- No pointer or reference indirection: ASL1 has no concept of pass-by-reference, heap allocation, or pointer arithmetic for example.

Unlike many other languages matching this description, ASL1 provides support for the following:

- Bit vectors: this includes a bit vector type, literal syntax, slicing expressions, and bit vector operators.
- Types that may depend on values: bit vector lengths may depend on ASL1 values (for example, see Parameters).
- Accessors: these provide function-like abstractions to read and write architectural state.
- Non-determinism.
- Unbounded integers and integer constraints.
- Unbounded precision rational numbers.

Further information about ASL1 can be found online, at the following links.

- ASL1's full formal language definition.
- ASL1's reference implementation.

The remainder of this chapter describes concepts common to all ASL1 constructs:

- Comments.
- Evaluation order.

2.1 Comments

ASL1 supports two styles of comments:

- `// ...`
  A comment that is terminated by the end of the line.
- `/* ... */`
  A delimited comment, that may span multiple lines.

Note

Comments do not nest.

2.2 Evaluation order

Evaluation order is explicitly specified by the formal language definition of ASL1, which can be found online. In general:

1. Identifiers, function call expressions, and literal expressions are evaluated before any operators using their results. Note however that some Boolean operators are short circuiting.
2. Right-hand sides of assignments are evaluated before left-hand sides.
3. The following are evaluated left-to-right.
   - Arguments to subprogram calls.
   - Tuples.
   - Binary operators that are not short circuiting.
   - Array indexing.
   - Bit vector slicing.
   - Record construction operations.
   - Start and end expressions for `for`-loops.

Chapter 3
Types

This chapter describes the following ASL1 types:

- Bit vectors.
- Integers.
- Reals.
- Booleans.
- Enumerations.
- Strings.
- Records.
- Tuples.
- Arrays.

3.1 Bit vectors

This section describes the types for bit vectors.

Syntax

`bits(N)` The type for bit vectors of length `N`.

`bit` A synonym of `bits(1)`.

`bits(N) { [slice1, ..., sliceN] field, ... }`

The type for bit vectors of length `N`, with a named `field` occupying non-overlapping slices given by `slice1`,`...`, `sliceN`.

Description

A bit vector is a finite-length array of 0s and 1s. Each length of bit vector is a different type. A bit vector has a length of at least 0.

Note

A bit vector may have a length of 0.

Bit vector literals are described in Literals.

The bits in a bit vector (or bitfield, see below) of length N are numbered from left to right (most significant to least significant, see Bit vector slicing) as N-1 down to 0. These indices are used to access the bit vector using *slices*.

A bit vector type can have named fields, providing a shorthand to access specific bit vector slices (see Records and record or bit vector field accesses). Fields may overlap with each other.

Bit vectors are the only concrete type in ASL1, corresponding directly to the values that are manipulated in registers, memory locations, and instructions. All other types are abstract.

3.2 Integers

This section describes the types for integers.

Syntax

`integer` The unconstrained integer type.

`integer{...}` The constrained integer type, with *constraints* inside braces.

`integer{}` The pending constrained integer type.

Description

ASL1 integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers as opposed to machine integers. To represent machine integers, pseudocode uses bit vectors of the appropriate length, and converts these to and from mathematical integers.

Integer literals are described in Literals.

There are three integer types:

- *Unconstrained* integers have no restrictions on the values they can take.
- *Constrained* integers are restricted to a set of possible values, given by their *constraints*. ASL1 arithmetic operators propagate integer constraints according to the behaviour of the operator. For example, the following code declares a constrained integer `x` that can take only the values `2`, `4`, and `8`, and a constrained integer `y` that can take values between `0` and `3` inclusive. The resulting type of `z` is `integer{0,2,4,6,8,12,16,24}`.

```
let x : integer{2,4,8} = ...;
let y : integer{0..3} = ...;
let z = x * y;
```

- *Pending constrained* integers are constrained integers whose constraints will be determined during type-checking of ASL1. They appear on left-hand sides of declarations (local or global), and constraints are inherited from the initialising expression of the declaration. For example, the declaration of `z` above could equivalently be written:

```
let z : integer{} = x * y;
```

3.3 Reals

This section describes the type for real numbers.

Syntax

`real` The real type.

Description

ASL1 reals are finite but unbounded in size and precision. They correspond to mathematical rational numbers rather than mathematical real numbers or machine floating-point numbers. To represent machine floating-point numbers, pseudocode uses bit vectors of the appropriate length, and converts these to and from reals.

Real literals are described in Literals.

3.4 Booleans

This section describes the type for Booleans.

Syntax

`boolean` The Boolean type.

Description

A Boolean is a logical `TRUE` or `FALSE` value.

Note

This is not the same type as `bit`, which is a bit vector of length 1. Both can take only two values: `bit` can take `'0'` or `'1'` and `boolean` can take `TRUE` or `FALSE`. However, these values cannot be compared: it is a coding error to write `'1' == TRUE` for example. Similarly, values of other types are not usable as Booleans. In other words, unlike languages such as JavaScript and Python, ASL1 does not have *truthy* or *falsy* values.

3.5 Enumerations

This section describes the type for enumerations.

Syntax

```
enumeration {named_value1, ..., named_valueN}
```

The type for an enumeration with `N` *named values*: `named_value1`, `...`, `named_valueN`.

Example

```
enumeration {RED, GREEN, BLUE}
```

The type for an enumeration with three named values: `RED`, `GREEN`, and `BLUE`.

Description

An enumeration is a defined set of named values.

Enumeration types must be declared as named types. The type of each named value is the named type defined by the declaration. There must be at least one named value, and named values cannot be shared between different enumeration types. By convention, each named value starts with the name of the enumeration type followed by an underscore.

Note

Unlike in languages such as C, ASL1's named enumeration values are not directly convertible to or from integers.

3.6 Records

This section describes the type for records.

Syntax

```
record {field1 : type1, ..., fieldN: typeN }
```

The type for a record with `N` *fields*: `field1` of type `type1`, . . ., and `fieldN` of type `typeN`.

Example

```
record { shift : bits(2), amount : integer };
```

The type for a record containing a bit vector field named `shift` and an integer field named `amount`.

Description

A record is a structured type that associates one or more names to member values. Each pair of name and member value is known as a *field*. A record type therefore associates one or more field names to types of the fields. Fields can have different types, and the types can include other structured types.

Record types must be declared as named types. The syntax for constructing a record and accessing its fields is described in Records and record or bit vector field accesses.

3.7 Tuples

This section describes the type for tuples.

Syntax

```
(type1, type2, ..., typeN)
```

The type for a tuple with `N` components, of types `type1`, . . ., `typeN`.

Example

```
(bits(32), bit)
```

The type for a tuple with two components, of types `bits(32)` and `bit` respectively.

Description

A tuple type is an ordered set of component types, separated by commas and enclosed in parentheses. The types can differ, and a tuple type must contain at least two component types.

Tuple types are often used as the return type for functions that return multiple results. For example, the tuple type `(bits(N), bits(4))` is the return type of the function `AddWithCarry()`, which adds two bit vectors and carry input. It returns a tuple containing two values: the first is of type `bits(N)` and represents the result of the addition, and the second is of type `bits(4)` and represents the resulting condition flags.

3.8 Arrays

This section describes the type for arrays.

Syntax

```
array[[len]] of ty
```

The type for an integer-indexed array whose length is given by integer `len`, indexed by integers between `0` and `len - 1` inclusive. Each element of the array has type `ty`.

```
array[[enum]] of ty
```

The type for an enumeration-indexed array, indexed by the values of the enumeration type `enum`. Each element of the array has type `ty`.

Examples

`array [[31]] of bits(64)`

The type for an integer-indexed array of 31 bit vectors, each of length 64. The array can be indexed by integers from 0 to 30 inclusive.

`array [[Color]] of integer`

The type for an enumeration-indexed array of three integers, given named type `Color` with underlying type `enumeration {RED, GREEN, BLUE}`. The array can be indexed by the named values `RED`, `GREEN`, and `BLUE`.

Description

An array is an ordered set of fixed size containing values of a single type (which may be structured). ASL1 arrays are indexed by either integers or enumerations. Arrays always contain at least one element.

The syntax for indexing arrays is described in Array indexing.

Note

There is no syntax for array literals.

3.9 Strings

This section describes the type for strings.

Syntax

`string` The string type.

Description

A string is an ordered set of printable characters. Printable characters are as follows.

- ASCII characters of code point between 32 and 126, inclusive.
- A tab character (ASCII code point 9).
- A newline character (ASCII code point 10).

Chapter 4
Expressions

This chapter describes ASL1 expressions.

All ASL1 expressions have a type:

- The type of a literal is determined by its syntax.
- The type of a identifier is determined by its declaration, which may be local or global.
- The type of a function call is determined by its function declaration.
- The types of other expressions are determined by the types of their sub-expressions.

The remaining sections describe the following ASL1 expressions:

- Identifiers.
- Function calls.
- Literals.
- Operators.
- Conditional expressions.
- Tuples.
- Array indexing.
- Records and record or bit vector field accesses.
- Bit vector slicing.
- Arbitrary expressions.
- Asserting type conversions (ATCs).
- Pattern matches.

4.1 Identifiers

Syntax

`x` A read of an identifier named `x`.

Description

An identifier name appearing in an expression represents a read of that identifier. Identifier names consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

4.2 Function calls

Syntax

```
foo(arg1, ..., argM)

bar{param1, ..., paramN}(arg1, ..., argM)

baz{param1, ..., paramN}
```

Function calls to `foo`, `bar`, and `baz`.

Description

A function call is an expression consisting of a function name, followed by braced parameters and parenthesised arguments, as applicable.

Note

Expressions cannot call procedures.

4.3 Literals

- Booleans. Boolean literals are either `TRUE` or `FALSE`.
- Reals. Literals for real numbers are written in decimal form using a decimal point. This means `0` must be a literal for an integer (see below), but `0.0` is a literal for a real number.
- Bit vectors. Literals for bit vectors are written as a single quotation mark, followed by the series of 0s and 1s, followed by another single quotation mark. For example, the two literals of type `bit` are `'0'` and `'1'`. Spaces can be included in bit vectors for ease of reading.
- Enumerations. An literal for an enumeration is simply the name of one of its named values.
- Strings. A literal for a string is a sequence of zero or more characters surrounded by double quotes. Each character must have ASCII codepoint between 32 and 126 inclusive. Tabs, newlines, double quotes, and backslashes are represented by escape sequences `\t`, `\n`, `\"`, and `\\` respectively. For example, `"hello world!\n"`.
- Integers. There are two forms of literal for integers:
  - Decimal form, such as `0`, `15`, `-1234`.
  - Hexadecimal form, such as `0x55` or `0x80000000`. Hexadecimal forms are treated as positive unless they have a preceding minus sign. For example, `0x80000000` represents the mathematical integer $+2^{31}$, and `-0x80000000` represents the mathematical integer -2^{31}.

  Underscores can be included in literals for integers (both decimal and hexadecimal forms) for ease of reading.

Note

Literals for integers have constrained integer types by default, but can be explicitly annotated with an unconstrained integer type. In the following example, `x` has type `integer{15}`, `y` has type `integer{55}`, and `z` has type (unconstrained) `integer`.

```
let x = 15;
let y = 55;
let z : integer = 3;
```

4.4 Operators

This section describes:

- Relational operators.
- Boolean operators.
- Bit vector operators.
- Arithmetic operators.
- Precedence rules.
- Operator overloading.

4.4.1 Relational operators

The following operators yield results of Boolean type.

`x == y` , `x != y`

For `x` and `y` of the same type: tests for equality and inequality (respectively). Both `x` and `y` must be of boolean, bit vector, integer, real, enumeration, or string type.

Note

A special form of equality/inequality is defined for *bit mask* literals (see Pattern matches)

`x < y` , `x <= y` , `x > y` , `x >= y`

For `x` and `y` both of integer or real type: the less than, less than or equal, greater than, and greater than or equal (respectively) comparison between them.

4.4.2 Boolean operators

For `x` and `y` of Boolean type, the following operators yield results of Boolean type.

`!x`	Logical inverse of `x`.
`x && y`	Logical AND of `x` and `y`. If `x` evaluates to `FALSE`, then `x && y` yields `FALSE` without evaluating `y`.
`x || y`	Logical OR of `x` and `y`. If `x` evaluates to `TRUE`, then `x || y` yields `TRUE` without evaluating `y`.
`x ==> y`	Logical implication between `x` and `y`. If `x` evaluates to `FALSE`, then `x ==> y` yields `TRUE` without evaluating `y`. Otherwise, `x ==> y` yields the result of evaluating `y`.

Note

Boolean operators `&&`, `||`, and `==>` use *short circuit evaluation*, as in the C language for example.

`x <=> y` Logical equivalence between `x` and `y`. Equivalent to `x == y`.

Note

Both `x != y` and `!(x <=> y)` are equivalent to the logical exclusive-OR of `x` and `y`. The ASL1 operator `XOR` only accepts bit vector arguments.

4.4.3 Bit vector operators

For operands of bit vector type, the following operators yield results of bit vector type.

Take `x` and `y` to be of length `N`, and `z` to be of length `M`. In other words, the types of `x`, `y`, and `z` are `bits(N)`, `bits(N)`, and `bits(M)` respectively.

`NOT x` Bitwise inversion of `x`. Yields a result of type `bits(N)`, obtained by inverting each bit of `x` from `'0'` to `'1'` or from `'1'` to `'0'`.

`x AND y` , `x OR y` , `x XOR y`

Bitwise AND, OR, and exclusive-OR of `x` and `y`. Yields a result of type `bits(N)`, obtained by performing each operation bitwise over `x` and `y`.

`x::z` Concatenation of `x` and `z`. Yields a result of type `bits(N+M)`, obtained by joining `x` and `y` in left-to-right order.

4.4.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bit vectors and results converted back to bit vectors. As these types are unbounded, no issues arise about overflow or similar errors.

`-x` For `x` of integer or real type: `x` with its sign inverted. The result is of the same type as `x`, with any integer constraints updated.

`x + y` , `x - y`

For `x` and `y` both of integer or real type: addition and subtraction of `x` and `y` respectively. Yields a result of the same type as both `x` and `y`, with any integer constraints updated.

For `x` and `y` both of type bit vector of length `N`: the least significant `N` bits of the results of converting `x` and `y` to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))[:N]
    = (UInt(x) + UInt(y))[:N]
x-y = (SInt(x) - SInt(y))[:N]
    = (UInt(x) - UInt(y))[:N]
```

The `[:N]` syntax denotes a slice.

For `x` of bit vector type of length `N` and `y` of integer type (for example, `'0000' + 4` or `'1111' - 2`):

```
x+y = x + y[:N]
x-y = x - y[:N]
```

`x * y` For `x` and `y` of integer and/or real type: multiplication of `x` and `y`. The result is of integer type if both `x` and `y` are, and real type otherwise.

`x / y` For `x` and `y` of real type where `y` does not evaluate to zero: mathematical division of `x` by `y` over rational numbers, yielding a result of real type. It is a coding error to use `x / y` when `y` evaluates to zero.

`x DIV y` For `x` and `y` of integer type where `x` is exactly divisible by `y`: exact division of `x` by `y`. In other words, yields the unique integer `z` such that `x = y * z`. It is a coding error to use `x DIV y` when `x` is not exactly divisible by `y`, or when `y` evaluates to zero.

`x DIVRM y` For `x` and `y` of integer type: inexact division of `x` by `y`. In other words, yields the largest integer `z` such that:

```
z * 1.0 <= (x * 1.0) / (y * 1.0)
```

It is a coding error to use `x DIVRM y` when `y` evaluates to zero.

`x MOD y` For `x` and `y` of integer type: remainder of dividing `x` by `y`. In other words:

```
x MOD y = x - y * (x DIVRM y)
```

It is a coding error to use `x MOD y` when `y` is less than or equal to zero.

`x ^ n` For `x` either of integer or real type and `n` of integer type: yields `x` raised to the power of `n`. The result is of the same type as `x`. For `x` of integer type, integer constraints are updated and it is a coding error if `n` is negative. For `x` of real type, it is a coding error if `x` is `0.0` and `n` is negative.

`x << n` For `x` and `n` of integer type, where `n` is non-negative: yields `x * 2^n` of integer type. It is a coding error to use `x << n` when `n` is negative.

`x >> n` For `x` and `n` of integer type, where `n` is non-negative: yields `x DIVRM 2^n` of integer type. It is a coding error to use `x >> n` when `n` is negative.

Note

Operators << and >> are often known as "integer shifts". For shift operators on bit vectors, see Standard library functions.

4.4.5 Precedence rules

The precedence rules for bracketing operators and expressions are given by the following table, where the first row represents lowest precedence.

Operators	Description
`||`, `&&`, `==>`, `<=>`, `as`	Booleans and ATCs
`==`, `!=`	Equality
`>`, `>=`, `<`, `<=`	Comparisons
`+`, `-`, `OR`, `XOR`, `AND`, `::`	Addition, subtraction, logic, concatenation
`*`, `DIV`, `DIVRM`, `/`, `MOD`, `<<`, `>>`	Multiplication, division, shifts
`^`	Exponentiation
`!`, `-`, `NOT`	Unary operators
`IN`	Pattern matches

This means that arithmetic operators follow normal operator precedence rules: exponentiation before multiply/divide before add/subtract.

For an expression of the form `x op1 y op2 z`, parentheses must be used to disambiguate possible bracketings unless either:

- `op1` and `op2` have differing precedence levels.
- `op1` is the same as `op2`, and `op1` and `op2` are *associative*.

The associative operators are: `+`, `*`, `&&`, `||`, `AND`, `OR`, `XOR`, `<=>`, and `::`.

For example, if `i`, `j` and `k` are integer-typed identifiers:

- `i > 0 && j > 0 && k > 0` is acceptable as `&&` and `>` are at different levels.
- `i > 0 && j > 0 || k > 0` is not acceptable as `&&` and `||` are at the same level.
- `i + j + k` is acceptable as `+` is associative.
- `i - j - k` is not acceptable as `-` is not associative.

4.4.6 Operator overloading

Operators in pseudocode can be overloaded, with different functionality when applied to different types.

Table 4-2 summarizes the operand types valid for each unary operator and the result type. Table 4-3 summarizes the operand types valid for each binary operator and the result type.

Table 4-2 Result and operand types permitted for unary operators

Operator	Operand Type	Result Type
!	boolean	boolean
NOT	bits(N)	bits(N)
-	integer	integer
	real	real

Table 4-3 Result and operand types permitted for binary operators

Operator	First operand type	Second operand type	Result type
==, !=	bits(N)	bits(N)	boolean
	integer	integer	
	real	real	
	enumeration	enumeration	
	boolean	boolean	
<, >, <=, >=	integer	integer	boolean
	real	real	
&&, ||, ==>, <=>	boolean	boolean	boolean
AND, OR, XOR	bits(N)	bits(N)	bits(N)
::	bits(N)	bits(M)	bits(N+M)
+, -	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
		integer	
*	integer	integer	integer
	integer	real	real
	real	integer	
	real	real	
/	real	real	real
DIV, DIVRM	integer	integer	integer
MOD	integer	integer	integer
^	integer	integer	integer
	real	integer	real
<<, >>	integer	integer	integer

4.5 Conditional expressions

`if t then x else y`

A conditional expression branching on Boolean expression `t`. Yields the result of evaluating `x` if `t` evaluates to `TRUE`, or `y` if `t` evaluates to `FALSE`.

The types of `x` and `y` must have a common ancestor type, according to ASL1's subtype hierarchy. The overall type of the conditional expression is the lowest common ancestor of `x` and `y`.

4.6 Tuples

`(expr1, ..., exprN)`

A tuple expression for a tuple with the `N` components `expr1`, . . ., `exprN`.

`tup.item0`, `tup.item1`, `tup.itemN`

Expressions to extract the first, second, and `N+1`th components from `tup` of tuple type `(type0, ..., typeN)`. The numbering is zero-indexed. For example, `(1, 2.0, TRUE).item1` evaluates to `2.0`.

A tuple is an ordered set of at least two expressions, separated by commas and enclosed in parentheses. The expressions can be of different types, and the tuple has a tuple type composed of the types of the individual expressions. The individual expressions can be extracted using the notation `.itemN`.

4.7 Array indexing

`arr[[idx]]` For `arr` with array type: access `arr` at index `idx`.

- For integer-indexed arrays with length `len`: `idx` must be of constrained integer type, and `0 <= idx < len` holds.
- For enumeration-indexed arrays: `idx` must be of the enumeration type that indexes the array.

Otherwise, it is a coding error.

Note

There is no syntax for array literals.

4.8 Records and record or bit vector field accesses

`x.fld` If `x` is of record type, an access to the field named `fld` within `x`. If `x` is of bit vector type with a bitfield `fld`, an access to the slices of `x` represented by `fld`.

`x.[fld1, fld2]`

For `x` of bit vector type with a bitfields `fld1` and `fld2`, an access to the concatenation of `x.fld1` and `x.fld2`.

`RecordName { fld1 = expression1, fld2 = expression2, ... }` A record construction expression, constructing a record of type `RecordName` with initial fields given by associating each field name `fldN` to the corresponding expression `expressionN`. All field names of `RecordName` must be defined; none can be omitted. However, the order of fields does not matter.

4.9 Bit vector slicing

A bit vector slice expression indexes specific bits in a bit vector, creating a new bit vector from extracted bits. Its syntax is `x[slice]`, where `x` is the bit vector being sliced, and `[slice]` is one of four forms of *slice*:

`[lsb +: len]` The basic *length* form. A slice starting at index `lsb` and of length `len`, where `0 <= len` and `0 <= lsb` hold. If `x` is of type `bits(N)`, then both `lsb < N` and `lsb + len <= N` must hold. The resulting bit vector is of type `bits(len)`.

`[:len]` The shorthand *length* form. Equivalent to `[0 +: len]`.

`[index]` The *single* form. The single index given by `index`. Equivalent to `[index +: 1]`.

`[msb:lsb]` The *range* form. The indices in order from `msb` down to `lsb`, where `lsb <= msb` must hold. Equivalent to `[lsb +: (msb-lsb+1)]`.

`[index *: len]`

The *scaled* form. Equivalent to `[(index * len) +: len]`.

`x[slice1, slice2, ..., sliceN]`

The multiple slice form. Equivalent to `x[slice1] :: x[slice2] :: ... :: x[sliceN]`. The length of the resulting bit vector is the sum of the lengths of all the slices.

4.9.1 Integer slicing

Slices can also be used to index bits in integers. Effectively, a slice on an integer represents it as a sufficiently long (sign-extended) two's complement bit vector. If the length of the resulting bit vector is *N*, bit *N*-1 represents the most significant bit and bit 0 the least significant bit.

Therefore, a slice of the form `x[i]` where `x` is an integer is obtained as follows:

- Define `y` as `x MOD 2^(i + 1)` (see Arithmetic operators).
- Then `x[i]` is `'0'` if `y < 2^i`, and `'1'` otherwise.

For example:

```
42[1]     == '1'
8[:4]     == '1000'
33[7:0]   == '00100001'
(-20)[:7] == '1101100'
```

4.10 Arbitrary expressions

`ARBITRARY : ty`

An arbitrary expression of type `ty`.

This expression evaluates to a value of the given type, but otherwise the value itself is not specified. Software must not rely on such values, and they must not be considered as providing any useful information.

4.11 Asserting type conversions (ATCs)

`expression as ty`

An asserting type conversion (ATC) of `expression` to type `ty`.

An ATC performs two roles:

- It *asserts* that the provided expression is of the provided type (it is a coding error if it is not).
- It *converts* the type of the expression (the type of an ATC is the provided type, rather than the type of the provided expression).

For example, this can be used to convert between constrained integer types:

```
let x : integer{2,4,8} = ...;
let y = x as integer{2};
```

This code produces an error if `x != 2`, and the resulting type of `y` is `integer{2}`.

4.12 Pattern matches

There are two forms of pattern match expression.

`expression IN {pattern1, pattern2, ...}`

Positive form. Tests whether `expression` matches any of the `patternN`, evaluating to `TRUE` if any matches, and `FALSE` otherwise.

`expression IN !{pattern1, pattern2, ...}`

Negative form. Tests whether `expression` matches any of the `patternN`, evaluating to `FALSE` if any matches, and `TRUE` otherwise.

There are six forms of pattern:

`-` *All* form. Matches any expression.

`expression` *Single expression* form. Matches any expression that evaluates to the same value as `expression`.

`<= expression`

Less than or equal to form. Matches any expression that evaluates to a value less than or equal to integer-typed `expression`.

`>= expression`

Greater than or equal to form. Matches any expression that evaluates to a value greater than or equal to integer-typed `expression`.

`expression1 .. expression2`

Range form. Matches any expression that evaluates to a value *both* greater than or equal to integer-typed `expression1` *and* less than or equal to integer-typed `expression2`.

`(pattern1, pattern2, ...)`

Tuple form. Matches any expression that evaluates to a tuple value of the same length, and whose contained values match their corresponding patterns.

Bit mask literal

Bit mask form. Bit masks are a special form of bit vector literal that can contain unspecified bits in addition to bits `'0'` and `'1'`. These unspecified bits are ignored when determining the result of the comparison, and are written in one of two ways:

- An `'x'` in place of a `'0'` or `'1'` denotes an unspecified bit.
- Any `'0'` or `'1'` bits enclosed in parentheses are considered unspecified.

There are additionally two special forms for pattern matching against a single bit mask pattern:

`x == bit_mask`

Equivalent to `x IN {bit_mask}`.

`x != bit_mask`

Equivalent to `x IN !{bit_mask}`.

For example, if `opcode` is a 4-bit bit vector, all of the following expressions are equivalent, and test whether `opcode` matches any of `'1000'`, `'1010'`, `'1100'`, or `'1110'`.

```
opcode == '1xx0';
opcode == '1(0)x0';
opcode == '1(0)(1)0';
opcode == '1(01)0';
opcode == '1x(1)0';
```

Chapter 5
Declarations

This chapter describes the following ASL1 top-level declarations:

- Global declarations.
- Type declarations.

Identifiers and type names introduced by global declarations are unique: no identifier or type may have the same name as another identifier or type.

Note

A subprogram may have the same name as another subprogram (known as overloading), an identifier, or a type. Subprogram declarations are described in Subprograms.

5.1 Global declarations

This section describes the four kinds of global declarations used in the pseudocode.

Global declarations define identifiers that are in scope globally – that is, for all other declarations.

5.1.1 `var` identifier declarations

Assignable global identifiers are declared using the `var` keyword in one of three ways.

```
var name = expr;
```

```
var name : type = expr;
```

With an initialising expression, and optionally a type annotation. The type of the identifier is given by the type annotation if present, or the type of the initialising expression if not. If both are present, it is a coding error for them to disagree.

```
var name : type;
```

With a type annotation, but no initialising expression. The type of the identifier is given by the type annotation.

```
var name1, ..., nameN : type;
```

Multiple uninitialised identifiers. Equivalent to `var name1: type; ...; var nameN : type;`.

For example, the following declares the identifiers `a`, `b`, `c`, `d`, and `e`. Identifier `a` receives the constrained integer type of its initialising expression. Identifiers `b` and `c` receive the types real and bit vector of length 3, respectively. Identifiers `d` and `e` receive the unconstrained integer type.

```
var a = 1;
var b : real = 2.0;
var c : bits(3);
var d, e : integer;
```

Note

Uninitialised identifiers take a *base value* of their declared type. Base values are defined by the formal language definition of ASL1, which can be found online. Pseudocode should not rely on such base values, and instead should explicitly assign to an uninitialised `var` identifier before using it. The following example declares the uninitialised `var` identifier `count`, before explicitly assigning to it.

```
var count : integer;
count = 1;
```

Records are most often declared without assigning values, and their fields assigned to one by one.

5.1.2 `let`, `constant`, and `config` identifier declarations

Declarations for `let`, `constant`, and `config` identifiers always have initialising expressions. They cannot be updated by assignment – they are immutable.

```
let name = expr;
```

```
let name : type = expr;
```

Declarations of `let` identifiers.

```
constant name = expr;
```

```
constant name : type = expr;
```

Declarations of `constant` identifiers.

```
config name : type = expr;
```

Declaration of a `config` identifier. The type annotation is mandatory.

As with `var` identifiers, the type of the identifier is given by the type annotation if present, or the type of the initialising expression if not. If both are present, it is a coding error for them to disagree.

For example:

```
let g = 4;
let h : real = 5.0;
constant i = 6;
constant j : real = 7.0;
config k : boolean = TRUE;
```

Note

To a reader, declarations of `let`, `constant`, and `config` identifiers are similar: they all declare immutable identifiers. Their differences lie in how they are used in pseudocode. In particular:

- `constant` identifiers are used for architectural constants.
- `config` identifiers represent architectural state that is fixed for a given implementation, but may vary across implementations.
- `let` identifiers are used otherwise when a value does not need to be updated by assignment.

5.2 Type declarations

ASL1 supports declaration of named types and subtyping.

5.2.1 Named types

`type name of ty;`

Declaration of a new named type, with name `name` and underlying type `ty`. The values of type `name` are the same as those of type `ty`.

Each declaration of a named type creates a different, incompatible named type, even if their underlying types are identical. In other words, named types are typically unrelated and their values cannot be interchanged or assigned to one another. For example, the following code introduces two distinct named types `A` and `B` with underlying type `integer`, and defines two global `var` identifiers `a` and `b`:

```
type A of integer;
type B of integer;

var a: A;
var b: B;
```

It is a coding error to attempt to assign `a` to `b`, or `b` to `a`.

Note

A declaration of an enumeration type also declares an identifier for each named value of the enumeration.

5.2.2 Subtypes

Named types can be related by subtyping.

`type name subtypes super_ty;`

Declaration of a new named type, with name `name` and underlying type `super_ty`. The type `name` subtypes `super_ty`.

`type name of ty subtypes super_ty;`

Declaration of a new named type, with name `name` and underlying type `ty`. The type `name` subtypes `super_ty`, and it is a coding error if `ty` is not a valid subtype of `super_ty`.

In the example below, named type `Square` subtypes named type `Shape1` (conversely, `Shape1` is a supertype for `Square`). It is therefore valid to use values of type `Square` in place of values of type `Shape1`. The converse is not true. Also, note that `Shape2` does not subtype `Shape1`: it is unrelated to all the other types.

```
type Shape1 of integer;
type Shape2 of Shape1;

type Square subtypes Shape1;
// Or equivalently:
// type Square of integer subtypes Shape1;

var shape1 : Shape1;
var shape2 : Shape2;
var square : Square;

shape1 = square; // OK as Square subtypes Shape1
```

Named types both subtype and are subtyped by the primitive type that describes their structure (a primitive type contains no named types). For example, this permits assignments of literals to identifiers with named types, or assignment of values with named types to identifiers of primitive types:

```
type RealInt of (real, integer);
type BV4 of bits(4);

var real_int : RealInt;
var bv4 : BV4;

var w : (real, integer) = real_int;
var x : RealInt = (1.0, 2);
var y : bits(4) = bv4;
var z : BV4 = '0000';
```

5.2.2.1 Record subtypes

Subtypes of records may extend their supertype with additional fields using the `with` keyword.

```
type name subtypes super_ty with { field1 : ty1, ..., fieldN : tyN };
```

> Declaration of a new named type, with name `name`. Its underlying record type is `super_ty` extended with fields `field1`, . . ., `fieldN`, of types `ty1`, . . ., `tyN` respectively. The type `name` subtypes `super_ty`.

The following example defines two named record types, `Coord2` and `Coord3`. The type `Coord3` subtypes `Coord2` and adds a new field, using the `with` keyword.

```
type Coord2 of record { x : real, y : real };
type Coord3 subtypes Coord2 with { z : real };
// or equivalently:
// type Coord3 of record { x : real, y : real, z : real } subtypes Coord2

var twoD : Coord2;
var threeD : Coord3;

twoD = threeD;
- = (twoD.x, twoD.y);
- = (threeD.x, threeD.y, threeD.z);
```

5.2.2.2 Bit vector subtypes

Bit vector subtypes may also extend their supertypes with additional fields. However, they cannot use the `with` keyword, and so must redefine all fields of their supertype. The following example defines two named bit vector types, `BV1` and `BV2`. The type `BV2` subtypes `BV1` and adds a new field.

```
type BV1 of bits(4) { [0] fieldA };
type BV2 of bits(4) { [0] fieldA, [1] fieldB } subtypes BV1;

var a : BV1;
var b : BV2;

a = b;
- = a.fieldA;
- = (b.fieldA, b.fieldB);
```

Chapter 6
Subprograms

This chapter describes how ASL1 subprograms are declared and called.

There are three kinds of subprogram:

- Functions, which may accept arguments and return a value.
- Procedures, which may accept arguments but do not return a value.
- Accessors, a function-like abstraction over architectural state.

The statements enclosed in each subprogram exist in a scope local to that subprogram.

This chapter also describes:

- Return statements.
- Parameters.
- Subprogram qualifiers.

6.1 Functions

A function definition has the form:

```
func function_name(arg1 : ty1, ..., argN : tyN) => return_ty
begin
    statement1;
    ...;
    statementM;
end;
```

This defines a function called `function_name`, with `N` arguments `arg1`, . . ., `argN`, each of types `ty1`, . . ., `tyN`. It returns a value of type `return_ty`. There may be zero arguments.

A call to this function has the form:

```
function_name(actual1, ..., actualN)
```

Functions may be recursive. Recursive functions are annotated with a recursion limit expression, which enforces a bound on the depth of recursion:

```
func function_name(...) => return_ty recurselimit limit_expr
```

It is a coding error to exceed a recursion limit.

6.2 Procedures

A procedure definition has the form:

```
func procedure_name(arg1 : ty1, ..., argN : tyN)
begin
    statement1;
    ...;
    statementM;
end;
```

Note

The first line of this definition does not include a return type. This distinguishes it from a function definition.

A call to this procedure has the form:

```
procedure_name(actual1, ..., actualM);
```

Procedures may be recursive. Recursive procedures are annotated with a recursion limit expression, which enforces a bound on the depth of recursion:

```
func procedure_name(...) recurselimit limit_expr
```

It is a coding error to exceed a recursion limit.

6.3 Accessors

Many references to system registers appear to be function calls, for example `PAR_EL1()`. These are often accessor calls. An accessor provides a function-like abstraction over some architectural state, allowing accesses to have side-effects or to reflect definitions such as RAZ/WI. It contains two parts: a “getter” function to read the state, and a “setter” procedure to write the state.

An accessor definition has the form:

```
accessor accessor_name(arg1 : ty1, ..., argL : tyL) <=> value_name : state_ty
begin
    getter
        statement_g1;
        ...;
        statement_gM;
    end;

    setter
        statement_s1;
        ...;
        statement_sN;
    end;
end;
```

This can be seen as defining two subprograms:

- The getter function, which returns a value of type `state_ty`. A call to the getter part of an accessor looks like any other function call. Therefore, references to functions in this document also apply to getter functions.
- The setter procedure, which accepts an additional argument named `value_name` of type `state_ty`. A call to the setter part of an accessor is as part of an assignable expression. Therefore, references to procedures in this document also apply to setter procedures.

For example:

```
PAR_EL1() = rhs;
```

This calls the setter part of the `PAR_EL1` accessor, passing in the expression `rhs` for its `value_name`.

As with other assignable expressions, accessors permit assignment to only parts of the architectural state they represent. For example:

```
PAR_EL1().NS = '1';
PAR_EL1()[1] = '1';
PAR_EL1().[NS,SH] = '110';
```

The first line writes to bitfield `NS` of the accessor, the second line writes to bit `1` of the accessor, and the third line writes to bitfields `NS` and `SH` of the accessor.

An assignment to part of an accessor has "read-modify-write" behaviour. In particular, the first line above is equivalent to:

```
var temp = PAR_EL1(); // read: invoke "getter"
temp.NS = '1';        // modify
PAR_EL1() = temp;     // write: invoke "setter"
```

6.4 Return statements

Return statements are used to exit subprograms.

A procedure or setter return has the form:

```
return;
```

This stops execution of the subprogram and returns control to its caller, without returning a value.

A function or getter return has the form:

```
return expression;
```

This stops execution of the subprogram and returns control to its caller, returning the value produced by evaluating `expression`. The `expression` must be of a type compatible with the return type specified in the subprogram declaration.

6.5 Parameters

Subprograms can have *parameters*: integer inputs that are used to define the bit vector lengths of subprogram arguments or return types. These are declared in braces before the argument list. For example:

```
func Zeros{N}() => bits(N)
func Replicate{N,M}(x: bits(M)) => bits(N)
accessor X{len}(n : integer) <=> value : bits(len)
```

The order of declaration is as follows:

1. Parameters that are referred to in the return type, from left-to-right.
2. Parameters that are referred to in the argument types, from left-to-right.

In general, the values for these parameters must be specified explicitly at call sites. For example:

```
let x : bits(16) = Zeros{16}();
let y : bits(32) = Replicate{32,4}('0101');
let z : bits(64) = X{64}(7);
X{64}(7) = z;
```

A shorthand can be used for function calls with parameters but no arguments: the empty argument list need not be written.

There are two exceptions to the rule above that requires explicitly specified parameters at call sites:

- Calls to standard library functions may *omit* parameters that are directly evident from the types of their input arguments. For example, `UInt()` and `SInt()` can always be called as if they do not have parameters, that is, without passing a parameter list at all. Similarly, `Replicate()` above can always be called as if it does not have a second parameter, that is, passing in its single result parameter.
- The right-hand side of a declaration (both local and global) that consists of a lone subprogram call can *elide* the first (or only) parameter `N` if the return type of the subprogram call is `bits(N)`, and the left-hand side has an explicit `bits(...)` type annotation.

For example:

```
// The following are equivalent:
let x = UInt{4}('1111');
let x = UInt('1111');

// The following are equivalent:
let y = Zeros{64}();
let y = Zeros{64};           // shorthand: no need to write empty argument list
let y : bits(64) = Zeros{};  // elide output parameter on right-hand side

// The following are equivalent:
let z = ZeroExtend{64,1}('1');
let z = ZeroExtend{64}('1');              // omit input parameter for standard library call
let z : bits(64) = ZeroExtend{,1}('1');   // elide output parameter on right-hand side
let z : bits(64) = ZeroExtend{}('1');     // omit input parameter for standard library call
                                          // AND elide output parameter on right-hand side
```

Note

Parameters are a similar concept to generics, versions of which exist in languages such as Python, Go, Swift, Java, C#, and Scala.

6.6 Subprogram qualifiers

Subprograms can optionally be marked with two kinds of qualifier keyword:

- Side-effects: at most one of `noreturn`, `readonly`, or `pure`.
- Overriding: at most one of `impdef` or `implementation`.

These two kinds can also be combined. For example:

```
pure func A ...
impdef func B ...
readonly implementation C ...
```

For accessors, the entire accessor can be marked with an overriding qualifier. Only the `getter` part can be marked `readonly`. For example:

```
impdef accessor D() <=> value : bits(64)
begin
    readonly getter
        ...
    end;

    setter
        ...
    end;
end;
```

6.6.1 Side-effects

A subprogram marked `noreturn` always terminates abnormally: either by executing an unreachable statement, or calling `EndOfInstruction()`.

A subprogram marked `readonly` does not modify any values of global `var` identifiers or call `EndOfInstruction()`, but can read global `var` identifiers or use non-determinism (`ARBITRARY`). It can call only subprograms marked either `pure` or `readonly`.

A subprogram marked `pure` does not read or modify global `var` identifiers, call `EndOfInstruction()`, or use non-determinism. It can call only subprograms marked `pure`.

Note

All `pure` functions can also be considered `readonly`, as `pure` functions are subject to more restrictions.

Both `pure` and `readonly` subprograms may use assertions (including asserting type conversions), or invoke an unreachable statement. As with ordinary subprograms, execution of an unreachable statement or attempting to (for example) divide by zero within a `pure` or `readonly` subprogram is a coding error.

6.6.2 Overloading

Subprograms may share the same name if they have:

- Differing numbers of arguments.
- The same number of arguments but with non-*clashing* types.

Such subprograms are described as *overloaded*. The requirements above ensure that any call to an overloaded subprogram clearly identifies a single subprogram declaration.

Note

A subprogram may also have the same name as an identifier or type introduced by a global declaration, or as an identifier introduced by a local declaration.

Type clashing is defined by the formal language definition of ASL1, which can be found online. In short, clashing types are “similar enough”: any integer type clashes with any other integer type, any bit vector type clashes with any other bit vector type, and so on.

6.6.3 Overriding

A subprogram marked `impdef` indicates implementation-specific behaviour. In particular, implementation-specific pseudocode is expected to override a subprogram marked `impdef` with one marked `implementation` that expresses the implementation-specific behaviour. The `impdef` and `implementation` function signatures must match precisely. In other words, they must match in all of:

- Side-effect qualifier.
- Name.
- Number, names, and types of arguments and parameters.
- Return type.

Chapter 7
Statements and control structures

This chapter describes ASL1 statements and control structures:

- Statements and indentation.
- Procedure calls.
- Conditional control structures.
- Loop control structures.
- Local declaration statements.
- Assignments.
- Other statements.

7.1 Statements and indentation

Simple statements include local declarations, assignments, and calls to procedures. Each simple statement must be terminated with a semicolon.

Compound statements include conditional control structures and loop control structures. Each compound statement defines a "block", beginning with some keyword(s) and typically terminated by an `end` keyword followed by a semicolon. The statements enclosed in each block existing in their own local scope.

ASL1 is an indentation-**in**sensitive language. Rather, it is a keyword-oriented language. However, statements contained within compound statements and subprograms are conventionally indented four spaces compared to the compound statement or subprogram itself, for ease of reading. In general, this also correlates with scoping: an indent signals the start of a local scope, and a dedent signals the end of the current local scope.

7.2 Procedure calls

A procedure call resembles a function call followed by a semicolon. However, unlike functions, procedures have no return value: the call is performed for its side-effects only.

7.3 Conditional control structures

This section describes ASL1 conditional control structures.

7.3.1 if . . . then . . . end;

A statement-level `if ... then ... end;` structure has the form:

```
if guard_expressionA then
    statement_A1;
    ...;
    statement_AL;
elsif guard_expressionB then
    statement_B1;
    ...;
    statement_BM;
else
    statement_1;
    ...;
    statement_N;
end;
```

Multiple `elsif` blocks may be used.

Each Boolean `guard_expression` is evaluated in turn until one evaluates to `TRUE`. The statements guarded by that `guard_expression` are then evaluated. The statements in the `else` block are evaluated if no `guard_expression` succeeded.

The following blocks are optional:

- `elsif` up to (but not including) `else`.
- `else` up to (but not including) `end`.

Expression-level `if ... then ... else ...` is described in Conditional expressions.

7.3.2 case . . . of . . . end;

A `case ... of ... end;` structure has the form:

```
case match_expression of
    when patternA where guard_expression =>
        statement_A1;
        ...;
        statement_AN;
    otherwise =>
        statement_1;
        ...;
        statement_M;
end;
```

Multiple `when` blocks may be used.

The `match_expression` is evaluated and tested against each `pattern` in turn (see Pattern matches), until there is a match and the corresponding Boolean `guard_expression` evaluates to `TRUE`. Then the corresponding statements are evaluated. The statements in the `otherwise` block are evaluated if no `pattern` and `guard_expression` combination succeeded.

The following are optional:

- Each `where guard_expression`.
- The `otherwise` block. However, without the `otherwise` block, it is a coding error if no `pattern` and `guard_expression` succeed.

7.4 Loop control structures

This section describes ASL1 loop control structures.

7.4.1 repeat . . . until . . .;

A `repeat ... until ...;` structure has the form:

```
repeat
    statement1;
    ...;
    statementN;
until guard_expression looplimit limit_expression;
```

The block of statements is evaluated once, and then the loop repeats until the Boolean `guard_expression` evaluates to `TRUE`.

The integer-typed `looplimit` is optional. However, if present it is a coding error if more than `limit_expression` loop iterations are evaluated.

7.4.2 while . . . do . . . end;

A `while ... do ... end;` structure has the form:

```
while guard_expression looplimit limit_expression do
    statement1;
    ...;
    statementN;
end;
```

The block of statements is repeatedly evaluated as long as the Boolean `guard_expression` evaluates to `TRUE`.

Note

A `while ... do ... end;` structure never evaluates its block of statements if `guard_expression` immediately evaluates to `FALSE`.

The integer-typed `looplimit` is optional. However, if present it is a coding error if more than `limit_expression` loop iterations are evaluated.

7.4.3 for . . . do . . . end;

A `for ... do ... end;` structure has the form:

```
for name = start_expression to end_expression looplimit limit_expression do
    statement1;
    ...;
    statementN;
end;
```

The block of statements is evaluated `(end_expression - start_expression) + 1` times if `start_expression <= end_expression` holds, or not at all otherwise. During iteration `n` (counting from 1), `name` is declared as a local `let` identifier with value `(start_expression + n) - 1` just before evaluating the block of statements.

The integer-typed `looplimit` is optional. However, if present it is a coding error if more than `limit_expression` loop iterations are evaluated.

Note

The identifier `name` is effectively declared locally as a `let` identifier. Like other local declarations, it is scoped locally and may not share a name with any other identifier or type name currently in scope (see Local declaration statements).

There is also an alternate form:

```
for name = start_expression downto end_expression ...
```

In this form, the block of statements is evaluated `(start_expression - end_expression) + 1` times if `start_expression >= end_expression` holds, or not at all otherwise. During iteration `n` (counting from 1), `name` takes value `(start_expression - n) + 1`.

7.5 Local declaration statements

This section describes the two kinds of ASL1 local declarations.

Local declarations bring a new identifier into scope. The identifier remains in scope until the end of the current block, and cannot be accessed outside of this block.

Locally declared identifiers are unique within their own scope. Therefore, they may not share a name with:

- Any identifiers or type names introduced by global declarations.
- Any local identifiers in scope.

Local identifiers in different scopes may have the same name. For example, two different subprograms may each declare a local identifier with the same name (provided no such global identifier or type name exists).

7.5.1 `var` identifier declarations

Local `var` declarations are similar to global `var` declarations, except for their local scoping.

In addition, local `var` declarations can introduce multiple new identifiers with an initialising expression of tuple type. Some (but not all) components of the initialising tuple may be discarded using a hyphen (written `-`). For example:

```
var (a, -, c) : (integer, real, boolean) = (1, 2.0, TRUE);
var (-, e) = AddWithCarry{64}(x, y, carry_in);
```

7.5.2 `let` identifier declarations

Local `let` declarations are similar to global `let` declarations, except for their local scoping.

Note

ASL1 has no local `constant` or `config` identifier declarations.

As with local `var` declarations, local `let` declarations can introduce multiple new identifiers with an initialising expression of tuple type, and may discard some components of the tuple using a hyphen (written `-`). For example:

```
let (f, -, h) = (2, 3.0, FALSE);
let (i, -) : (bits(64), bits(4)) = AddWithCarry{64}(x, y, carry_in);
```

Note

Local `let` declarations are preferred if the new identifier does not need to be further updated by assignment.

7.6 Assignments

Assignment statements use the = character to assign their right-hand side expressions to their left-hand side *assignable expressions*:

```
assignable_expression = expression;
```

Assignable expressions are defined by the formal language definition of ASL1, which can be found online. They resemble expressions, and the most common are of the following forms:

- An identifier that was declared using the `var` keyword (either local or global).
- A call to the setter procedure of an accessor.
- An array index on an identifier or a setter call.
- A record or bit vector field access on an identifier or a setter call.

- A bit vector slice on an identifier or a setter call.
- A tuple of assignable expressions that do not involve setter calls.

In addition, assignable expressions can be of the following form:

`x.(fld1, ..., fldN)`

Multi-field form, where `x` is record-typed or bit vector-typed. Equivalent to `(x.fld1, ..., x.fldN)`.

Note

A hyphen (written –) discards some or all of the right-hand side expression. It may be used in isolation, or with the tuple and multi-field forms above.
For example, the following evaluates a function for its side-effects, and discards its return value.

```
- = SideEffecting();
```

The following lines discard part of the tuple return value of a function call:

```
(res, -) = AddWithCarry{64}(x, y, carry_in);
x.(-, fld2) = AddWithCarry{64}(x, y, carry_in);
```

7.7 Other statements

`pass;` A null statement: evaluates without any effect.

`assert expression;`

Asserts that the Boolean-typed `expression` evaluates to `TRUE`. Otherwise, it is a coding error.

`unreachable;`

An unreachable statement. It is always a coding error to reach this statement during evaluation.

Chapter 8
Standard library functions

This chapter describes functions from the ASL1 standard library. The full list of functions and their canonical ASL1 definitions can be found in ASL1's reference implementation at the following link:

```
https://github.com/herd/herdtools7/blob/ASLRefBET2/asllib/libdir/stdlib.asl
```

```
Abs(x: integer) => integer
Abs(x: real) => real
```

The absolute value of `x`: if `x` is less than zero then `-x`, otherwise `x`.

```
AlignDownP2(x: integer, p2: integer) => integer
AlignDownP2{N}(x: bits(N), p2: integer{0..N}) => bits(N)
```

The greatest multiple of `2^p2` that is less than or equal to `x`, where `p2` is non-negative. The integer variant requires `x` to be non-negative. The bit vector variant views `x` as an unsigned integer, and the result is represented by its rightmost `N` bits. The `N` parameter to the bit vector variant may be omitted.

```
AlignDownSize(x: integer, sz: integer) => integer
AlignDownSize{N}(x: bits(N), sz: integer{1..2^N}) => bits(N)
```

The greatest multiple of `sz` that is less than or equal to `x`, where `sz` is positive. The integer variant requires `x` to be non-negative. The bit vector variant views `x` as an unsigned integer, and the result is represented by its rightmost `N` bits. The `N` parameter to the bit vector variant may be omitted.

```
AlignUpSize(x: integer, sz: integer) => integer
AlignUpSize{N}(x: bits(N), sz: integer{1..2^N}) => bits(N)
```

The smallest multiple of `sz` that is greater than or equal to `x`, where `sz` is positive. The integer variant requires `x` to be non-negative. The bit vector variant views `x` as an unsigned integer, and the result is represented by its rightmost `N` bits. The `N` parameter to the bit vector variant may be omitted.

```
ASR{N}(x: bits(N), sz: integer) => bits(N)
ASR_C{N}(x: bits(N), sz: integer) => (bits(N), bit)
```

The arithmetic right shift of `x` by non-negative `sz` bits, shifting sign bits into leftmost bits. The `_C` variant additionally requires non-zero `sz` and returns the last bit to be shifted out of `x[0]`. The `N` parameters may be omitted.

```
BitCount{N}(x: bits(N)) => integer{0..N}
```

The number of '1' bits in x. The N parameter may be omitted.

```
CeilLog2(x: integer) => integer
```

The base-2 logarithm of positive integer x, rounded up.

```
CountLeadingSignBits{N}(x: bits(N)) => integer{0..N}
```

The count of consecutive leftmost bits of x that are equal to the leftmost bit (x[N-1]). The N parameter may be omitted.

```
CountLeadingZeroBits{N}(x: bits(N)) => integer{0..N}
```

The count of consecutive leftmost bits of x that are equal to '0'. The N parameter may be omitted.

```
Extend{N,M}(x: bits(M), unsigned: boolean) => bits(N)
```

The extension of M-bit bit vector x to N bits, where M <= N. If unsigned is TRUE, then zero-extend, otherwise sign-extend. The M parameter may be omitted.

```
FloorLog2(x: integer) => integer
```

The base-2 logarithm of positive integer x, rounded down.

```
FloorPow2(x : integer) => integer
```

The largest power of 2 that is less than or equal to positive integer x.

```
HighestSetBit{N}(x: bits(N)) => integer{-1..N-1}
HighestSetBitNZ{N}(x: bits(N)) => integer{0..N-1}
```

The position of the leftmost '1' bit in x. The NZ variant requires a non-zero bit vector, and the other variant returns -1 for a zero bit vector. The N parameter may be omitted.

```
ILog2(x : real) => integer
```

The base-2 logarithm of the absolute value of non-zero real x, rounded down to an integer.

```
IsEven(x: integer) => boolean
```

TRUE if x is even (that is, x MOD 2 == 0, or equivalently x[0] == '0'), FALSE otherwise.

```
IsOdd(x: integer) => boolean
```

TRUE if x is odd (that is, x MOD 2 == 1, or equivalently x[0] == '1'), FALSE otherwise.

```
IsOnes{N}(x: bits(N)) => boolean
```

TRUE if x contains only '1' bits, FALSE otherwise. The N parameter may be omitted.

```
IsPow2(x : integer) => boolean
```

TRUE if x is positive and a power of 2, FALSE otherwise.

```
IsZero{N}(x: bits(N)) => boolean
```

TRUE if x contains only '0' bits, FALSE otherwise. The N parameter may be omitted.

```
LowestSetBit{N}(x: bits(N)) => integer{0..N}
LowestSetBitNZ{N}(x: bits(N)) => integer{0..N-1}
```

The position of the rightmost '1' bit in x. The NZ variant requires a non-zero bit vector, and the other variant returns N for a zero bit vector. The N parameter may be omitted.

```
LSL{N}(x: bits(N), sz: integer) => bits(N)
LSL_C{N}(x: bits(N), sz: integer) => (bits(N), bit)
```

The logical left shift of x by non-negative sz bits, shifting zero bits into rightmost bits. The _C variant additionally requires non-zero sz and returns the last bit to be shifted out of x[N-1]. The N parameter may be omitted.

```
LSR{N}(x: bits(N), sz: integer) => bits(N)
LSR_C{N}(x: bits(N), sz: integer) => (bits(N), bit)
```

The logical right shift of `x` by non-negative `sz` bits, shifting zero bits into leftmost bits. The `_C` variant additionally requires non-zero `sz` and returns the last bit to be shifted out of `x[0]`. The `N` parameter may be omitted.

```
Min(x: integer, y: integer) => integer
Min(x: real, y: real) => real
```

The minimum of `x` and `y`. Returns `x` if `x <= y`, and `y` otherwise.

```
Max(x: integer, y: integer) => integer
Max(x: real, y: real) => real
```

The maximum of `x` and `y`. Returns `y` if `x <= y`, and `x` otherwise.

```
Ones{N}() => bits(N)
```

The bit vector of length `N` and containing only `'1'` bits.

```
Real(x: integer) => real
```

The real value corresponding to integer `x`. Equivalent to `x * 1.0`.

```
Replicate{N,M}(x: bits(M)) => bits(N)
```

The bit vector of length `N` that contains `N DIV M` copies of `x`. The `N` parameter must be exactly divisible by the `M` parameter. The `M` parameter may be omitted.

```
ROL{N}(x: bits(N), sz: integer) => bits(N)
ROL_C{N}(x: bits(N), sz: integer) => (bits(N), bit)
```

The left rotation of `x` by non-negative `sz` bits. Leftmost bits are deleted and reinserted as rightmost bits. The `_C` variant additionally requires non-zero `sz` and returns the rightmost bit of the result. The `N` parameter may be omitted.

```
ROR{N}(x: bits(N), sz: integer) => bits(N)
ROR_C{N}(x: bits(N), sz: integer) => (bits(N), bit)
```

The right rotation of `x` by non-negative `sz` bits. Rightmost bits are deleted and reinserted as leftmost bits. The `_C` variant additionally requires non-zero `sz` and returns the leftmost bit of the result. The `N` parameters may be omitted.

```
RoundUp(x: real) => integer
RoundDown(x: real) => integer
```

The nearest integer to real `x`, obtained by rounding up or down (respectively).

```
SignExtend{N,M}(x: bits(M)) => bits(N)
```

The sign-extension of `M`-bit bit vector `x` to `N` bits, where `M <= N`. The `M` parameter may be omitted.

```
SInt{N}(x: bits(N)) => integer{(if N == 0 then 0 else -(2^(N-1))) .. (if N == 0 then 0
        else 2^(N-1)-1)}
```

The signed integer corresponding to `x`, viewed under two's complement. Bit `N-1` is the most significant bit, and bit `0` is the least significant bit. The `N` parameter may be omitted.

```
SqrtRounded(x : real, nbits : integer) => real
```

The square root of non-negative real `x`, with positive `nbits` of precision after the hidden leading `'1'` bit of a base-2 floating-point significand, with inexact values rounded to odd.

```
UInt{N}(x: bits(N)) => integer{0..2^N-1}
```

The unsigned integer corresponding to `x`, viewed as unsigned. Bit `N-1` is the most significant bit, and bit `0` is the least significant bit. The `N` parameter may be omitted.

`ZeroExtend{N,M}(x: bits(M)) => bits(N)`

The zero-extension of `M`-bit bit vector `x` to `N` bits, where `M <= N`. The `M` parameter may be omitted.

`Zeros{N}() => bits(N)`

The bit vector of length `N` and containing only `'0'` bits. Equivalent to `0[:N]`.

Chapter 9
ASL1 index

This chapter contains the following tables:

- Table 9-1 which indexes ASL1 types.
- Table 9-2 which indexes ASL1 expressions.
- Table 9-3 which indexes ASL1 declarations.
- Table 9-4 which indexes ASL1 statements.
- Table 9-5 which indexes ASL1 keywords and other syntax.

Table 9-1 Index of ASL1 types

Syntax	Link to section
`array`	Arrays
`bits(N),bits(N) {...},bit`	Bit vectors
`boolean`	Booleans
`enumeration`	Enumerations
`integer,integer{},integer{...}`	Integers
`real`	Reals
`record`	Records
`(ty1, ..., tyN)`	Tuples

Table 9-2 Index of ASL1 expressions

Syntax	Link to section
`x, y, Foo, bar_Qux`	Identifiers or enumeration literals

`foo(...)`
`bar{...}(...)`
`baz{...}`
	Function calls

`TRUE, FALSE, 1.0, 3.14, '0', '11 11', 42, -12_34, 0x55, -0x2_3`	Literals
`x == y, x != y,` `x < y, x <= y, x > y, x >= y`	Relational operators
`!x, x && y, x || y, x ==> y, x <=> y`	Boolean operators
`NOT(x), x AND y, x OR y, x XOR y,` `x :: y`	Bit vector operators

`-x, x + y, x - y, x * y, x / y,`
`x DIV y, x MOD y, x DIVRM y, x ^ y,`
`x << y, x >> y`
	Arithmetic operators

`if ... then ... else ...`	Conditional expressions
`(e1, ..., eN)` `tup.itemN`	Tuples
`arr[[idx]]`	Array indexing
`x.fld, x.[fld1, fld2],` `RecTy {fld = e, ...}`	Records and record or bit vector field accesses

`x[slice], x[slice, ...],`
`x[lsb+:len], x[:len], x[idx],`
`x[msb:lsb], x[idx*:len]`
	Bit vector slicing

`ARBITRARY : ty`	Arbitrary expressions
`expr as ty`	Asserting type conversions (ATCs)

`expr IN {...}`
`expr IN !{...}`
`'1xx1', '1(01)1'`
	Pattern matches

Table 9-3 Index of ASL1 declarations

Syntax	Link to section

`var x : ty;`
`var x, y, ... : ty;`
`var x : ty = e;`
`var x = e;`
	Assignable identifier declarations. See `var` declarations.

`let x = e;`
`let x : ty = e;`
`constant x = e;`
`constant x : ty = e;`
`config x : ty = e;`
	Immutable identifier declarations.

`type X of ty;`
`type X of ty subtypes Y;`
`type X subtypes Y;`
`type X subtypes Y with ...;`
	Type declarations.

`func name{...}(...) ...`	Procedure declarations
`func name{...}(...) => ty ...`	Function declarations

Syntax	Link to section
`accessor name{...}(...) <=> x : ty`	Accessor declarations

Table 9-4 Index of ASL1 statements

Operator	Link to section

`foo(...);`
`bar{...}(...);`
`baz{...};`
	Procedure calls

`if ... then ... end;` `case ... of ... end;`	Conditional control structures

`repeat ... until ...;`
`while ... do ... end;`
`for ... do ... end;`
	Loop control structures

`var x = e;`
`var x : ty = e;`
`var x : ty;`
`var x, y, ... : ty;`
`var (x, y, ...) = e;`
`var (x, y, ...) : ty = e;`
	Assignable identifier declarations. See Local declaration statements.

`let x = e;`
`let x : ty = e;`
`let (x, y, ...) = e;`
`let (x, y, ...) : ty = e;`
	Immutable identifier declarations. See Local declaration statements.

`lhs = rhs;`	Assignments

`lhs(...) ... = rhs;`
`lhs{...}(...) ... = rhs;`
`lhs{...} = rhs;`
	Assignment to an accessor to an accessor

`pass;`	Null statements
`assert e;`	Assertions
`unreachable;`	Unreachable statements
`return;` `return e;`	Return statements

Table 9-5 Index of ASL1 keywords

Keyword	Usage
`//`	Begins a line comment
`/* ... */`	Delimits a block comment
`accessor`	Begins an accessor declaration
`AND`	Bit vector operators
`ARBITRARY`	Arbitrary expressions
`array`	Array types
`as`	Asserting type conversions (ATCs)
`assert`	Assertions
`begin`	Begins a subprogram body

Keyword	Usage
bit, bits	Bit vector types
boolean	Boolean types
case	Begins a conditional control structure
config, constant	Begins a global immutable identifier declaration
DIV, DIVRM	Arithmetic operators
do, downto	Part of a loop control structure
else	Part of a conditional control structure or a conditional expression
end	Ends many statements
enumeration	Enumeration types
XOR	Bit vector operators
FALSE	Boolean literal
for	Begins a loop control structure
func	Begins a function or procedure declaration
getter	Part of an accessor declaration
if	Begins a conditional control structure or a conditional expression
impdef, implementation	Subprogram qualifiers
integer	Integer types
let	Begins an immutable identifier declaration (local or global)
looplimit	Part of a loop control structure
MOD	Arithmetic operators
NOT	Bit vector operators
noreturn	Subprogram qualifiers
of	Part of an array type, conditional control structure, or a type declaration
OR	Bit vector operators
otherwise	Part of a conditional control structure
pass	Null statement
pure, readonly	Subprogram qualifiers
record	Record types
recurselimit	Part of a function or procedure
repeat	Begins a loop control structure
return	Exits a subprogram
setter	Part of an accessor declaration
subtypes	Part of a type declaration
then	Part of a conditional control structure or a conditional expression
to	Part of a loop control structure
TRUE	Boolean literal
type	Begins a type declaration
unreachable	Unreachable statement

Keyword	Usage
`until`	Part of a loop control structure
`var`	Begins an assignable identifier declaration (local or global)
`when`, `where`	Part of a conditional control structure
`while`	Begins a loop control structure
`with`	Part of a type declaration