ASL Syntax Reference
DDI 0620

Arm Architecture Technology Group

January 24, 2024

Contents

Chapter 1

Non-Confidential Proprietary Notice

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

Chapter 2

Disclaimer

This document is part of the ASLRef material. It is a snapshot of: `https://github.com/herd/herdtools7/commit/6dd15fe7833fea24eb94933486d0858038f0c2e8`

This material covers both ASLv0 (viz, the existing ASL pseudocode language which appears in the Arm Architecture Reference Manual) and ASLv1, a new, experimental, and as yet unreleased version of ASL.

The development version of ASLRef can be found here `~/herdtools7/asllib`.

A list of open items being worked on can be found here `~/herdtools7/asllib/doc/ASLRefProgress.tex`.

This material is work in progress, more precisely at Alpha quality as per Arm's quality standards. In particular, this means that it would be premature to base any production tool development on this material.

However, any feedback, question, query and feature request would be most welcome; those can be sent to `asl-support@arm.com` or by raising issues or PRs to the herdtools7 github repository.

Chapter 3

ASL Abstract Syntax

An Abstract Syntax Tree (AST for short) is a kind of labelled tree. A node in an AST is either a leaf, represented by its label, or an internal node of the form $L(n_1, \ldots, n_k)$ where L is its label and $n_1, \ldots, n_k$ are its ordered children nodes, which we also refer to as *components*. Components can be (possibly-empty) lists of nodes, shown as n^*, and optional nodes (lists of 0 or 1 elements), shown as $n?$. Tuples are shown as $(n_1, \ldots, n_k)$.

An abstract syntax is similar to a context-free grammar, but defined over ASTs. A terminal derives leaf nodes while non-terminal use alternatives to derive internal nodes.

A major benefit of employing an abstract syntax is that it allows abstracting away syntactic details that are only important to enable correct parsing, such as punctuation, and succinctly representing lists and optional values. By defining an abstract syntax for ASL, we can uniformly represent programs in ASLv0 as well as ones in ASLv1 and define a single type system for them.

We define the abstract syntax of ASL below. We sometimes provide extra details to individual derivations by adding comments below them, in the form (* *this is a comment* *).

node **components**

unop ::= "!" | "-" | "NOT"
binop ::= "&&" | "||" | "-->" | "<->"
  (* *binop_boolean* *)
| "==" | "!=" | ">" | ">=" | "<" | "<="
  (* *binop_comparison* *)
| "+" | "-" | "OR" | "XOR" | "EOR" | "AND"
  (* *binop_add_sub_logic* *)
| "*" | "/" | "DIV" | "DIVRM" | "MOD" | "<<" | ">>"
  (* *binop_mul_div_shift* *)
| "^"
  (* *binop_pow* *)
literal = <int_lit>
| <hex_lit>
  (* *merged into <int_lit>?* *)
| <boolean_lit>
| <real_lit>
| <bitmask_lit>
  (* *also represents <bitvector_lit>* *)
| <string_lit>

node **components**

expr ::= E_Literal(*literal*)
| E_Var(<identifier>)
| E_CTC(expr, ty)
  (* *A checked type constraint* *)
| E_Binop(binop, expr, expr)
| E_Unop(unop, expr)
| E_Call(<identifier>, expr*, (<identifier>, expr)*)
| E_Slice(expr, slice*)
| E_Cond(expr, expr, expr)
| E_GetArray(expr, expr)
| E_GetField(expr, <identifier>)
| E_GetFields(expr, <identifier>*)
| E_Record(ty, (<identifier>, expr)*)
  (* *Exception construction* *)
| E_Concat(expr*)
| E_Tuple(expr*)
| E_Unknown(ty)
| E_Pattern(expr, pattern)

node		components
pattern	::=	Pattern_All
	|	E_Var(<identifier>)
	|	Pattern_Any(pattern*)
	|	Pattern_Geq(expr)
	|	Pattern_Leq(expr)
	|	Pattern_Mask(bitmask_lit)
	|	Pattern_Not(pattern)
	|	Pattern_Range(expr, expr)
		(* *Lower to upper, included.* *)
	|	Pattern_Single(expr)
	|	Pattern_Tuple(pattern*)

node		components
ty	::=	T_Int(*int_constraints?*)
	|	T_Real
	|	T_String
	|	T_Bool
	|	T_Bits(expr, *bitfield**)
		(* *expr is a statically evaluable expression denoting the length of the bit-vector.* *)
	|	T_Enum(<identifier>*)
	|	T_Tuple(ty*)
	|	T_Array(expr, ty)
	|	T_Record(field*)
	|	T_Exception(field*)
	|	T_Named(<identifier>)
		(* *A type variable.* *)
		(* *This is related to* I_{LDNP} *)

node		components
int_constraint	::=	Constraint_Exact(expr)
		(* *A single value, given by a statically evaluable expression.* *)
	|	Constraint_Range(expr, expr)
		(* *An interval between two statically evaluable expression.* *)

node		components
bitfield	::=	BitField_Simple(<identifier>, slice*)
		(* *A name and its corresponding slice.* *)
	|	BitField_Nested(<identifier>, slice*, bitfield*)
		(* *A name, its corresponding slice and some nested bitfields.* *)
	|	BitField_Type(<identifier>, slice*, ty)
		(* *A name, its corresponding slice, and the type of the bitfield.* *)