Aa Language Reference Manual

Madhav Desai
Department of Electrical Engineering
Indian Institute of Technology
Mumbai 400076 India

September 7, 2015

1 Introduction

Aa is a programming language for the description of algorithms. The control-flow in an **Aa** program corresponds to a petri net of a specific class. The data-flow is constructed using static single-assignment variables (which can be assigned to only once), storage objects, messaging pipes and constants. Pointers to storage objects can be created and manipulated (pointer arithmetic, dereferencing etc.) in the usual way.

A program in **Aa** can also be viewed as a description of a system which reacts with its environment through input and output ports. Thus, an **Aa** program can either be executed on a computer, or be mapped to a logic circuit.

In the rest of this document, we outline the structure and the syntax of the **Aa** language, and also describe the semantics of an **Aa** program, especially the execution model and the behaviour of the equivalent system.

2 Program structure

A program in **Aa** consists of a sequence of declarations and module definitions.

```
program := ( module-definition | program-variable-declaration )*
```

Program variable declarations can belong to one of three classes as described in Section 3.

A module in **Aa** is the basic unit of compilation, and has the following structure

```
$module [module-name]
   $in(<input-arguments>)
   $out(<output-arguments>)
$is
{
   <object-declarations>
```

```
   <sequence-of-statements>
}
```

Thus, a module has a name, has input and output arguments, declares objects and consists of a sequence of statements. Objects (storage/pipe/constant) declared in the module are visible only in the module body. For example:

```
$module [sum]
   $in (a: $uint<32> b: $uint<32>)
   $out (c : $uint<32>)
$is
{
   c := (a+b)
}
```

In this example, *sum* is a module which has two inputs and a single output. The type of the inputs and the output is

```
$uint<32>
```

which is an unsigned 32-bit integer (More details on types are given in Section 6).

Each statement in the sequence of statements can be abstractly viewed as a region in a petri-net, and has a set of input (or source) places and a set of output (or sink) places. The execution of the statement is triggered when there is a token present in every place in some specified subsets of its source places and when the statement finishes execution, a token is placed in some specified subset of its sink places.

The control flow in the sequence of statements in a module is serial in nature. Thus each statement has a single source place and a single sink place, with the sink place of a statement being the source place of its successor. The module as a whole can be viewed as having a single entry place and a single exit place. When a token appears in the entry place, the first statement can trigger, and when the last statement has finished, a token is placed in the exit place.

The region between { and } is termed a **scope**. Thus, a module description defines a scope. In this case, the scope has a label which is the same as the module name. As we shall see later, the statements in the statement sequence may in turn define scopes. Each scope defines a name-space. The visibility rules and access mechanisms between scopes will be described in Section 5.

3 Program Variables

Variables in the program can belong to the following classes:

- **Storage** variables declared in the program can be written into from multiple points in the program. When declared at the program scope, storage variables are visible throughout the program. Storage variables can be thought of as memory locations which can be accessed from different points in the program. A storage variable declaration has the form

```
$storage <var-name>: <var-type>
```

For example

```
$storage mempool: $array [1024] $of $uint<32>
```

- **Pipe** variables declared at the program scope are globally visible buffers (which can be defined to be either first-in-first-out (FIFO) or last-in-first-out (LIFO)) that can be read from anywhere in the program and can be written into from anywhere in th program. A pipe is a FIFO (or FIFO) buffer with a specified number of slots. A write to a pipe succeeds only if there is an empty slot in the FIFO (a successful write uses up an empty slots. A read from the pipe succeeds only if there is at least one occupied slot in the pipe (a successful read creates an empty slot in the pipe). Thus, pipes can be used to model queues as well as stacks, and also provide a mechanism for synchronization and for message passing in an **Aa** program.
  - A pipe that is only written into in the program, but is not read from, is assumed to have a destination outside the program. Such a pipe is an output port of the program.
  - A pipe that is only read from in a program but not written into, is assumed to have a source outside the program. Such a pipe is an input port of the program.

  A pipe declaration has the form

  ```
  ($lifo)? $pipe <pipe-name>: <pipe-type>  ($depth <integer>)?
  ```

  For example

  ```
  $pipe myfifo: $uint<32> $depth 16 // 16-slot queue
  $lifo $pipe mystack: $uint<32> $depth 4 // 4-slot stack
  ```

  Pipe variable declarations can also be appended by the following keywords:

  ```
     ($in | $out)? ($port)? ($signal)?
  ```

  These additional keywords mean the following things:

  - Inclusion of "$in" (respectively "$out") means that the pipe is a read-only (respectively write-only) object.
  - The "$port" keyword means that the pipe is meant to be connected to the outside world *without* any handshake, such as for example in an asynchronous interface to the outside world.
  - The "$signal" keyword means that the interface to the external world is a single bit.

- **Constant** variables declared at the program scope are globally visible and have initial values which can never be altered in the program. Thus, an assignment to a constant is an error. A constant declaration has the form

  ```
  $constant <constant-name>: <constant-type> := <initial-value>
  ```

  For example

  ```
  $constant wordlength: $uint<32> := 16
  ```

- **Implicit** or **Static Single Assignment** variables are created by statements in the **Aa** program (as described below in Section 4). These are analogous to registers that can be written into only from one point, but can be read from several points in the program. Implicit variables are not declared, but are defined by the statements which write to them.

4 Statements

Statements in the module-body are can be of two kinds:

- Complete or atomic statements: these are statements which have a single entry place and a single exit place. These are the only statements that can occur in a module body. These statements are further divided into *simple* statements and *block* statements.
- Incomplete or sub-atomic statements: these statements can appear only inside block statements, and they can have multiple source and sink places. Examples of these statements are statements for combining tokens (merge and join statements) and statements for redirecting tokens (the place statement).

4.1 Complete Statements

Complete statements can be one of the following:

- An **assignment** statement is of the form

  ```
  target-ref := expression
  ```

  where *target-ref* either specifies a declared object or an undeclared name. If it is an undeclared name, then it is an implicit variable that is attached to the assignment statement. No other statement can write to this variable, but any statement can read from this variable subject to scoping rules. Thus, every implicit variable is such that exactly one statement defines its value. When control flow reaches an assignment statement, it evaluates the expression, updates the value of the target and passes the control flow forward (we will see what this means in the sequel). Some examples of assignment statements are

```
a  :=  (~b)
b  :=   (c + (d + e))
```

etc. Expressions can be *unary*, *binary* or *ternary*, and are described in detail in Section 7. Details of the syntax are provided in Section 12.

- A **call** statement is of the form

  ```
  call-spec module-name input-arguments output-arguments
  ```

  The call statement thus specifies a module to which control flow is to be passed. The call statement may itself define new implicit variables through its specified targets (and thus is the only statement that can modify these newly defined implicit variables). When control flow reaches the call statement, it forwards the token to the called module, and finishes when the token exits the called module. Recursive calls and cyclical dependencies between modules through call statements are not permitted in the **Aa** language. Here is an example of a call statement

  ```
  // pass control to a module named foo
  // foo has two inputs and three outputs
  $call foo (p q)  (r s t)
  ```

  Here p, q, r, s, t can be expressions.

  Complete atomic statements can have a guard predicate specified, so that they are executed only if the guard predicate is true. This is specified either by

  ```
  $guard ( <guard-variable> )  <statement-spec>
  ```

  or

  ```
  $guard ( ~ <guard-variable> )  <statement-spec>
  ```

  For example

  ```
  $guard (sel_add) r := (a + b)
  ```

  or

  ```
  $guard (~ sel_add) $call afun (a b) (r)
  ```

  More details of the syntax are provided in Section 12.

- The **null** statement does nothing, and just passes the token onwards. This is how it looks like

  ```
  $null
  ```

4.2 Complete Block Statements

Block statements can be used to describe complex control flow, and consist of a sequence of statements. Block statements are of the following types:

- **Series** block statements are of the form

  ```
  series-block-statement :=
               series-block-specifier block-name
               {
                  declarations
                  sequence-of-atomic-statements
               } [(variable-exports)]
  ```

  The behaviour is similar to the module, in that the control token flows serially down the sequence of statements. Here is an example:

  ```
  $seriesblock [s1]
  {
     $storage b $uint<32>
     b := a
     b := ( $mux (a > c)  (b+c) (b-c))
     q := ( b * 2)
  } ( q => q )
  ```

  In this example, the variable b was declared as a storage variable in scope s1, and hence was legally able to be the target of multiple assignment statements. The optional export specification

  ```
  ( q => q )
  ```

  after the curly brace } specifies that the variable q defined inside the block $s1$ is visible outside the block as q (so that other scopes can use this variable, see Section 5 below).

- **Parallel** block statements are of the form

  ```
  parallel-block-statement :=
               parallel-block-specifier block-name
               {
                  declarations
                  sequence-of-atomic-statements
               } [(variable-exports)]
  ```

  When a token enters the parallel block statement, it is replicated into as many tokens as there are statements in the sequence, and all statements are started in parallel. When all statements have finished and released their token, the parallel block statement ends and a single token exits the parallel block statement (this is essentially a fork followed by a join). Here is an example

```
$parallelblock [p1]
{
   b := a + c
   d := a - c
}
```

The two statements will start in parallel, and the block will finish when both have finished. The order in which the two statements are executed is **not** specified.

- **Fork** block statements are generalizations of the parallel block region, and they allow the programmer to express complex fork-join interactions. They have the following structure:

```
fork-block-statement :=
             fork-block-specifier block-name
             {
                declarations
                sequence-of-statements
             } [(variable-exports)]
```

The control flow is similar to the parallel block region, in the sense that all statements in the sequence are started in parallel, and the fork block statement ends when all statements in the sequence have ended. The difference is that fork blocks can have an additional statement which allows the programmer to provide additional synchronization points. These statements are termed **join** statements and have the form

```
join-statement :=
         join-specifier list-of-labels
          fork-specifier  list-of-statements
```

The meaning of this statement is that it waits until all statements in the list-of-labels have finished, then starts the list-of-statements (in parallel) and finishes. This essentially defines a node in a directed graph, with the arcs corresponding to statements in the statement-lists. The join indicates the arcs (or statements) from whom tokens have to arrive. After all tokens have arrived on the incoming arcs, tokens are sent along the outgoing arcs specified in the list of statements following the optional fork specifier. There is an implicit fork at the entry point to the fork block statement and an implicit join at the end of the fork block statement. It is thus easy to describe an arbitrary directed graph with arbitrary forks and joins. The only restriction is that this directed graph must be acyclic! This is enforced by requiring that a join statement refers only to labels of statements that appear before the join. Here is an example:

```
$forkblock [f1] {
  a := (b+c)
  $seriesblock [s1] { ... }
  $seriesblock [s2] { ... }
  $seriesblock [s3] { ... }
  $join s1 s2 $fork
      $seriesblock [s4] { ... }
  $join s3 s4
}
```

In this example, the first assignment statement as well as s1, s2, s3 are started in parallel. When both s1 and s2 have finished, s4 is started and when s3, s4 have joined and when the first assignment statement has finished, the forkblock f1 finishes execution.

- **Branch** block statements are constructs which allow the programmer to describe arbitrary sequential branching behaviour. They describe a control flow in which a single token is active within the block. The movement of this token is controlled by special flow control statements. A branch block is constructed as follows

```
branch-block-statement :=
             branch-block-specifier block-name
             {
 branch-block-statement-sequence
     } [(variable-exports)]
```

The sequence of statements appearing in a branch block consist of

  - Simple statements or Block statements.
  - Switch statements: A switch statement has the form

```
switch-statement :=
         switch-spec switch-expression
           ( expr-value  branch-block-statement-sequence )*
           ( default branch-block-statement-sequence )?
         end-switch-spec
```

The switch-expression is checked, and depending on its value, one of the alternative statements is selected. Thus, the incoming token to the switch statement is passed to one of the alternatives. For example,

```
$switch a $when 0 $then a1 := (b + c)
          $when 1 $then a2 := (b - c)
          $default $null
$endswitch
```

The control token is routed to the appropriate choice sequence and if the token is not routed out of the block (by the **place** statement described below), then the token passes to the statement following the switch.

- If statements: An if statement has the form

```
if-statement :=
       if-spec test-expression
           branch-block-statement-sequence
       (else
           branch-block-statement-sequence)?
       end-if-spec
```

  For example:

```
$if (a != 0) $then
   q := (r + s)
   t := 0
$else
   qdash :=  (r - s)
$endif
```

  If the control token reaches the end of a selected segment in the if statement (that is, without being rerouted by a place statement), then the control token is passed on to the statement immediately following the if statement.

- The place statement: The **place** statement identifies a place which will contain a token after the place statemet has executed. For example,

```
$place [fastpath]
```

  means that the incoming token is placed in a labeled place **fastpath** (the place statement never puts a token into its default exit place). The token placed in **fastpath** must be used to trigger a unique **merge** statement which is required to depend on this labeled place in the same branch block. If no such merge exists, or if multiple such merges exist, then this is an error.

- Merge statements: A merge statement is specified as

```
merge-statement :=
          merge-spec label-list merge-assignments end-merge-spec
```

  which is to be interpreted as follows. The labels in the label list refer to token labels defined by **place** statements within the branch block. Whenever a token is present in any of the places in the label-list for a merge, the merge statement starts and executes a series of special assignments which multiplex values into targets based on which arc the token arrived from. The merge statement then releases its token to the next statement.

* The merge assignments inside a merge block are all of a specific form, called the phi-statement. A phi-statement has the following form

  ```
  phi-spec target-expr := expr1 on-spec place-spec-1
                                [expr2 on-spec place-spec-2 ... ]
  ```

  For example:

  ```
  $phi a := b $on $entry c $on loopback
  ```

  This says that the target a is to take the value of b if the merge statement was started from its entry place (from a token passed from its predecessor) and is to take the value of c if the merge statement was started from the place "loopback". Note that if the merge statement is to be triggered by a token in its default entry place, one of the labels in the label-list for the merge must contain the identifier $entry, which corresponds to the merge statement being triggered by the immediately preceding statement. The target of a phi-statement must be an implicit variable (declared by the statement itself, not a storage or a pipe object).

Here is an example of a branch block constructed in this manner

```
$branchblock [b1]
{
  $merge loopback $entry // $entry is the entry place
                         // of the merge statement.
                         // this merge is triggered by
                         // a token entering it from
                         // its predecessor or by
                         // a token in the place "loopback"


       $phi q := 0 on $entry r on loop
                    // q is defined by where
                    // the token came from
  $endmerge
  r := (q + 1)
  $if (r < 10) $then
       $place [loopback] // put token in place "loopback"
  $endif
} (r => r)
```

This is to be interpreted as follows: the merge executes whenever a token is present in its entry place or in the place labeled **loopback** The merge defines variable **q** with a value which is either 0 (if a token was present in the entry place) or r (if a token was present in the "loop" place). By the construction rules in an **Aa** program, it is impossible for there to be a token present in more than one input place to a merge. The final export makes r directly visible as r in the parent scope of the branch-block.

4.3 Pipelinable-loops: the dopipeline-while construct

The **Aa** language provides a single construct for describing loops. The form of the construct is

```
$dopipeline
   $merge $entry $loopback
     <phi-statements>
   $endmerge
   atomic-statement-sequence
$while <condition-expression>
```

The do-while statement can appear only in branch-blocks. Control enters the do-while from a predecessor in the branch-block, and stays in the do-while as long as the condition-expression is false (the test is done every time the loop body has finished). The $loopback place is used to repeat the loop-body. The leading merge statement in the do-while is used to merge the control-flow coming in either from the entry to the do-while or from the loopback place.

The downstream tools will always attempt to pipeline the do-while loop by starting the next iteration while the current one is still in progress. Dependency analysis on values in the loop-body is used to determine the extent of loop pipelining. While doing the pipelining, one can specify two parameters as follows

```
$dopipeline $depth 8 $buffering 2
  ...
$while ....
```

This specifies that up to 8 iterations of the loops will be kept alive in hardware, and the amount of buffering per operation will be kept to 2.

Additionally, it is possible to specify that the loop-pipelining be done to its extreme possibility, that is, to the maximum possible rate that can be achieved.

```
$dopipeline $depth 8 $buffering 2
$fullrate // specify that extreme pipelining
  // should be attempted.
  ...
$while ....
```

Note that it is always possible to describe a do-while loop using a branch-block, and this is what should be done if loop-pipelining is not desired.

To summarize, an **Aa** program is constructed as a collection of modules, each of which is a sequence of statements. The use of series, parallel, fork and branch block statements enables the programmer to describe a highly concurrent structured system with complex branching behaviour. The resulting control flow structure is a petri-net with provable liveness and safety properties [1].

5 Scoping Rules

An **Aa** program is made of modules which in turn contain statements and so on. The program thus has a hierarchy of scopes (except for the program, each scope is delimited by { and }) with each scope being contained in another (except for the program itself, which is not contained in any scope).

The rules for scoping are as follows:

1. A declared variable defined in a scope is visible in all descendent scopes.
2. A reference to a variable b in a scope X is resolved by checking whether the variable is defined in that scope, and if not found there, by checking in the scope that contains the scope X, and so on.
3. A scope can read from variables that are defined in descendant scopes.
4. A scope can export a variable defined within its scope to its parent scope. The name of the exported variable must not clash with the name of any other variable visible in the parent scope.
5. A scope can read from variables that are defined in ancestor scopes.
6. A scope can only write to one of the following:
   - an implicit variable defined in the scope.
   - a storage or pipe variable defined in the scope or an ancestor of the scope.
   - an output argument of a module of which the scope is a descendant. In this case, there can be at most one statement in the entire module which writes to this output argument.

A variable reference in a statement may be specified as follows

```
a            look for variable a in current scope;
             if not found look in the parent.

:a           same as the previous case

../:a        look for variable a starting from the parent of
             the current scope.

%p:a         look for variable a starting from the child scope
             with label p (child scope of the current scope).

../../:a     look for variable starting from the parent of
             the parent of the current scope.

%p%q:a       look for variable starting from the child scope
```

```
             with label q of the child scope with label p of
             the current scope

a[10]        look for a storage variable a defined in the
             current scope.  If not found, look for it in
             the parent scope, and so on.  If eventually found,
             access the corresponding element of the composite
             object.
```

Thus, a generic variable reference has the form

```
scope-reference : variable-reference
```

and the scoping rules forbid a scope from accessing variables which are not defined in either an ancestor or a descendant of the scope (the export mechanism should be used if this is needed).

6 Types

Types in **Aa** can be either scalar types or composite types.

Scalar types can be one of

- Unsigned integers: An unsigned integer type has a width parameter and is specified as

  ```
  $uint<width>
  ```

  The width parameter can be any positive number. Values corresponding to this type are to be viewed as unsigned integers represented by a binary sequence of the specified width.

- Signed integers: A signed integer type has a width parameter and is specified as

  ```
  $int<width>
  ```

  The width parameter can be any positive number. Values corresponding to this type are to be viewed as integers maintained in the two-s complement form by a binary sequence of the specified width.

- Pointers: A pointer is an unsigned integer with a default pointer width (set to 32 for now), which specifies the type of object to which it points. For example

  ```
  $pointer< $uint<32> >
  ```

  is a pointer which refers to a storage object of type

```
$uint<32>
```

- Floats: A float is parametrized by two integers, the width of the exponent, and the width of the mantissa. The specification is

  ```
  $float<exponent,mantissa>
  ```

  where the exponent and mantissa must be positive integers. The float is represented by a word with $exponent + mantissa + 1$ bits (with the additional bit needed for the sign). The standard IEEE 754 float and double precision representations correspond to

  ```
  $float<8,23>
  ```

  and

  ```
  $float<11,52>
  ```

  respectively. Currently, these are the only float types that are supported.

Composite types in **Aa** can be either array types or record types.

- Array types in **Aa** have the form

  ```
  $array [d1][d2]...[dn] $of <element-type-spec>
  ```

  The values $d1, d2, \ldots dn$ must be positive integers, and element-type-spec must refer to a type. For example,

  ```
  $array [10][10] of $array [10] $of $uint<32>
  ```

  is a two-dimensional array type whose elements are one dimensional arrays of 32-bit unsigned integers.

- Record types in **Aa** have the form

  ```
  $record <type-1> <type-2> ... <type-n>
  ```

  An element of such a record type is an aggregate whose first element is of type type-1, second element is of type type-2 and so on. For example:

  ```
  $record <$uint<32> > <$array [10] $of $uint<32> >
  ```

- Named record types are used to avoid the circular reference problem (for example, when a record has a pointer to itself).

  ```
  $record [myrec] <$uint<32> >  <$pointer<myrec> >
  ```

  defines a record type with name myrec, one of whose fields is a pointer to myrec.

7 Expressions

Expressions in **Aa** fall into the following classes

- Constant literal references.
- Simple object references.
- Indexed object references.
- Pointer de-reference expressions.
- Address-of expressions.
- Unary expressions.
- Slice expressions.
- Bitmap (bit permutation) expressions.
- Binary expressions.
- Ternary expressions.

7.1 Constant literal references

Constants can be specified in one of many ways, depending on their type.

- Integers can be specified in their decimal form, as for example

  ```
  23
  -8
  ```

  or in binary form, as for example

  ```
  _b10111
  ```

  When specified in binary form, the twos-complement value of the integer being specified should be used.

- Floats are specified in the exponentiated form

  ```
  _f2.3000e+10
  _f-1.354e+10
  ```

  where the mantissa has to have exactly one digit to the left of the decimal point.

- Composite constants are specified as a space separated list of values.

  ```
  ( 12 32 43 10)
  ( (1 5) (2 8) (9 100) )
  ```

  etc. The elements are listed in row major form. Thus a[2][2] is listed as

  ```
  a[0][0] a[0][1] a[1][0] a[1][1]
  ```

7.2 Simple object references

References to an object have the form

```
<scope-specifier>:<object-specifier>
```

The scope specifier can either specify a parent scope of the scope in which the expression appears, or a child scope of the scope in which the expression appears (as described in Section 5). The scope-specifier can be omitted if the reference is to be resolved starting from the same scope as the expression. For example:

```
../:a
```

7.3 Indexed object references

These have the form

```
<scope-specifier>:<root-object-specifier>[i1][i2]...[im]
```

The scope and root-object-specifier are interpreted in the same manner as before. The indices $i1, i2, \ldots im$ must be non-negative integers. The object-specifier must be one of

- A storage object whose type is composite. For example, if a is a storage object whose type is a two-dimensional composite type, then

  ```
  a[0][1]
  ```

  is interpreted in the usual manner. Thus, when the root-object-specifier is a declared storage object, the indexed expression evaluates to an element of the object.

- A scalar object of type pointer. For example if *ptr* is a pointer to an object of type T, then

  ```
  ptr[1]
  ```

  is a pointer which will point to the adjacent object of type T if we assume that ptr points to an array of objects of type T. This is similar to the following evaluation in **C**

  ```
  int* ptr;      // ptr points to some int a[i].
  ptr = ptr[1]   // ptr now points to a[i+1]
  ```

  Now, if type T is a one-dimensional array type, say

  ```
  $array [10] of $uint<32>
  ```

  then the following expression makes sense

```
ptr[1][2]
```

and is similar to the following evaluation in **C**

```
int a[10][10];
ptr = &a[0]      // points to a[0] which is
                 // array of 10 integers
ptr = ptr[1][2]; // ptr points to a[1][2]
```

Thus, the evaluation of an indexed expression whose root is a pointer evaluates to a pointer of some element type.

7.4 Pointer de-reference expressions

If **ptr** is a pointer to a scalar object, then the expression

```
->(ptr)
```

refers to the value of the object pointed to by **ptr**. Such an expression can occur **only** as the target or source of a simple assignment statement. For example

```
a := ->(ptr)
->(ptr) := (a+1)
```

7.5 Address-of expressions

If **a** is a declared **storage** object, then

```
@(a)
```

is a pointer which points to **a**. Such an expression can occur **only** as a source of a simple assignment statement. For example

```
ptr := @(a)
```

7.6 Unary expressions

These can be of three types:

- The **cast** expression is of the form

  ```
  ($cast (<type-spec>)  <expression>)
  ```

  For example

  ```
  ($cast ($uint<10>) ../:a)
  ```

  The cast expression takes the value of the specified expression and converts it to a value of the specified type. This is equivalent to a type-cast in **C**, as for example in

```
float a;
int b = (int) a;
```

- The **bitcast** expression is of the form

  ```
  ($bitcast (<type-spec>)  <expression>)
  ```

  For example

  ```
  ($bitcast ($uint<10>) a)
  ```

  The bitcast expression takes takes the bits corresponding to the value of the specified expression and treats them as being of the specified cast type. This is similar to the following in **C**

  ```
  float a;
  int b = *((int*) &a);
  ```

  The destination type need not be the same size as the source type (higher order bits are truncated or padded with 0 as necessary).

- The **bit-wise complement** expression is of the form

  ```
  ( ~ <expression> )
  ```

  The symbol for the complement is the same as the one used in the C programming language, and the evaluation of the expression proceeds in the usual way.

The parentheses around a unary expression are **required**.

7.7 Slice expressions

Slice expressions can be used to extract a contiguous bit-field from a value. They have the following syntax:

```
LPAREN $slice <expression> <high-index> <low-index> RPAREN
```

where high-index and low-index are constant non-negative integers. The result of executing the expression is an unsigned integer whose width is $(high-index-low-index)+1$. The value produced by expression may be of any type, but for the slicing, it is flattened in the natural manner (see the notes about aggregate types). All bit-vectors are assumed to indexed as $HIGH$ dowto LOW.

For example

```
( $slice ($bitcast ($uint<16>) _h1234) 8 4) = _h3
```

7.8 Bitmap expressions

These expressions can be used to express bit permutations in a word. The syntax is as follows

```
LPAREN $bitmap <expression> ( <from-index> <to-index> )+ RPAREN
```

The flattened bit-vector value of expression is permuted according to the information provided by the $(from, to)$ pairs (result(to) = source(from)).

For example

```
( $slice ($bitcast ($uint<4>) _h3) 0 3 1 2 2 1 3 0 ) = _he
```

7.9 Binary expressions

These are of the form

```
(<expression> <operation-id>  <expression>)
```

The following binary operations are supported

```
// arithmetic operators
PLUS            +
MINUS           -
MUL             *
DIV             /
SHL             <<
SHR             >>
ROL             <o<
ROR             >o>

// bit-wise logical operators
NOT             ~
OR              |
AND             &
XOR             ^
NOR             ~|
NAND            ~&
XNOR            ~^

// comparison operators
EQUAL           ==
NOTEQUAL        !=
LESS            <
LESSEQUAL       <=
GREATER         >
GREATEREQUAL    >=

// concatenation operator
```

```
CONCATENATION     &&
     c = (a && b) means that the bits of
     a and b are concatenated to produce
     the bits of c.  a,b must be of type
     $uint<>

// bit-select operator
BITSELECT         []
    c = (a [] i) means that c gets the
    value of the bit of a with index i
    (index 0 is the least significant bit)
```

The evaluation of a binary expression proceeds in the usual way. Note that when specifying an expression, you **must** use parentheses around each expression.

7.10 Ternary expressions

There is only one form for the ternary expression.

```
( $mux <test-expression>  <true-expression> <false-expression> )
```

The test-expression is evaluated, and if true, the true-expression is evaluated, and if false, the false-expression is evaluated. Note the parentheses delimiting the expression. For example

```
($mux a (b+1) (c+d))
```

8 Storage variables and Memory Spaces

An **Aa** program can contain declarations to storage objects. These storage objects are grouped into memory spaces using the following rule: two storage objects must be in the same memory space if there is a de-referenced pointer in the program which can point to either of these two storage objects.

To understand this concept, consider the following example

```
$module [foo]
  $in (a: $uint<32>)
  $out (b: $uint<32>)
$is
{
   $storage u: $uint<32>
   $storage v: $uint<32>
   $storage w: $uint<32>
   $storage ptr: $pointer< $uint<32> >
   w := a
   ptr := @(u)
   ->(ptr) := w
```

```
   ptr := @(v)
   ->ptr := u
   b := u
}
```

Now consider the storage variable **ptr**, which can hold a value which is a pointer to either **u** or a pointer to **v**. Thus, **u** and **v** must belong to the same memory space. However, **w** can sit in a memory space by itself. The store accesses implied by the $->(ptr)$ will point to the memory space which contains **u** and **v**.

8.1 Pointers from "outside?

What should we do in the following case?

```
$module [foo]
   $in (a : $uint<32>)
   $out (b: $uint<32>)
$is
{
   $storage u: $uint<32>
   $storage ptr: $uint<32>
   ptr := ($bitcast ($pointer<$uint<32> >) a)
   tmp := ->(ptr)
   ptr := @(u)
   ->(ptr) := tmp
   b := u
}
```

In this case, **ptr** takes the value of *a*, and as a pointer, it must point to an **external** object. Since ptr can point to this external object and also to **u**, the external object and **u** must be kept in the same memory space. In this case, there are two possible solutions: either the external object needs to be made internal, and moved **inside** the system, or, the object **u** must be moved outside the system and be made part of the outside world. This is done by the **AaLinkExtMem** utility which is described in a separate document.

9 Inline and Macro Modules

By default, each distinct module in an **Aa** program is eventually implemented as a separate VHDL entity by downstream AhirV2 tools (though downstream tools are free to instantiate multiple copies of this entity).

However, in order to give the **Aa** programmer more flexibility, a module in **Aa** may be marked to be inlined or substituted into the program at the point of the call. The AhirV2 utility **AaOpt** is responsible for this substitution.

9.1 Inlined Modules

An inlined module is substituted into the point of a call to it in a three-step manner. The input arguments are sampled at the point of the call, the body of the module is invoked and the output arguments are transferred. The semantics of an inlined function call are defined by this three step procedure. The input arguments are sampled at the point of the call, and the output arguments are updated at the termination of the call.

For example, the following program

```
$inline $module [add] $in (a: $uint<32> b: $uint<32>)
    $out (c: $uint<32>) $is
{
    c := (a+b)
}

$module [add_check] $in (p: $uint<32> q: $uint<32>)
    $out (r : $uint<32>) $is
{
    $call add (p q) (r)
}
```

is translated to

```
$module [add_check]
$in ( p : $uint<32>   q : $uint<32>  )
$out ( r : $uint<32>  )
$is
{
  $seriesblock[add_18]
  {
    $parallelblock[InArgs]
    {
      add_18_a := p
      add_18_b := q
    }
    ( add_18_a => add_18_a add_18_b => add_18_b )
    $seriesblock[body]
    {
      add_18_c := (add_18_a + add_18_b)
    }
    (  add_18_c => add_18_c )
    $parallelblock[OutArgs]
    {
      r := add_18_c
    }
    ( r => r )
```

```
  }
  ( r => r )
}
```

The three step substitution of add into add_check is clearly seen.

9.2 Macro Modules

A module defined as a macro (described below) is directly substituted at the point of a call to it. That is, all references to input and output arguments are replaced by references to the corresponding expression passed to the called function. The semantics are equivalent to a MACRO instantiation in programming languages such as C/C++.

For example:

```
$macro $module [shift] $in (a: $uint<32>)
    $out (c: $uint<32>) $is
{
    c := a
}

$pipe inpipe: $uint<32>
$pipe midpipe: $uint<32>
$pipe outpipe: $uint<32>

$module [shift_check] $in ()
    $out () $is
{
    $call shift (inpipe) (midpipe)
    $call shift (midpipe) (outpipe)
}
```

is equivalent to

```
$pipe  inpipe : $uint<32>  $depth 1
$pipe  midpipe : $uint<32>  $depth 1
$pipe  outpipe : $uint<32>  $depth 1
$module [shift_check]
$in ()
$out ()
$is
{
  $seriesblock[shift_14]
  {
    midpipe := inpipe
  }
```

```
  $seriesblock[shift_17]
  {
    outpipe := midpipe
  }
}
```

The two calls to shift are replaced by series blocks with input/output arguments replaced by the expressions passed to them at the point of the call.

Since it is possible to pass a pipe as an argument to a call, a macro-module is permitted to violate the rule that there should be at most one write to an interface object within a module.

10 Foreign Modules

A module can be marked as foreign by using the
$foreign keyword.

```
$foreign $ module [GetValue]
$in (ptr $pointer<32>)
$out (val $uint<32>)
```

The **Aa** compiler then considers that the module GetValue is defined "elsewhere" and does not try to link to it directly. This linking is done outside by other tools which use the results of the **Aa** compilation process.

11 Examples

Here is a very simple program

```
// an array of 32-bit unsigned integers.
$storage mem: $array<1024> of $uint<32>
$module [sel_mod]
   $in (a:$uint<32> b:$uint<10>)
   $out (c:$uint<32>)
$is
{
   t := (mem[b] + a)
   mem[b] := t
   c := t
}
```

This consists of a single module, which accumulates a value into an array position.

An example which is a little bit more complicated:

```
// an array of 32-bit unsigned integers.
$storage mem:$array[1024] $of $uint<32>
```

```
// module returns the sum of mem[I] from
// I=low to I=high
$module [sum_mod]
    $in (low:$uint<10> high:$uint<10>)
    $out (sum:$uint<32>)
$is
{
   d := (high-low)
   mp := ((high-low)/2)

   $branchblock[trivcheck]
    {
       // d from parent scope
       $if (d > 0) $then
           // do two summations in parallel
           // parallel summations
           $parallelblock[parsum]
           {
               $branchblock[sb1]{
                    $storage  I:$uint<10>
                    $merge $entry loopback
                       $phi s := 0 $on $entry  s1 $on loopback
                    $endmerge
                    $if (I < mp) $then
                       I := (I+1)
                       s1 := ($mux (I == 0)  0 (s + mem[I]))
                       $place [loopback]
                    $endif
               } (s => sb1_s)
               $branchblock[sb2]{
                    $storage  J:$uint<10>
                    J := (mp + 1)
                    $merge $entry loopback
                       $phi s := 0 $on $entry s1 $on loopback
                    $endmerge
                    $if (J < high) $then
                         J := (J+1)
                         s1 := ($mux (J == (mp+1))  0 (s + mem[J]))
                         $place [loopback]
                    $endif
               } (s => sb2_s)
           }(sb1_s => sb1_s sb2_s => sb2_s)

           // combine results from parallel statement above
           snontriv := (sb1_s  + sb2_s)
```

```
            $place [nontrivsum]
        $else

            // summation is trivial
            striv := mem[low]
            $place [trivsum]
        $endif
        $merge nontrivsum trivsum
           // which sum do you pick? depends on which path was taken
           $phi sum := snontriv $on nontrivsum striv $on trivsum
        $endmerge
    }
}
```

This example describes an algorithm which computes the sum of a section of an array by dividing the problem into two partial summations.

12 Syntax

The syntax for **Aa** follows the following principles

- All keywords begin with the $ sign.
- The region between { and } defines a new scope.
- Statements are space separated (no semicolons at all).
- Expressions are fully parenthitized. Thus $(a+b)$ is a legal expression, but $a+b$ is not.

The parser is implemented using an LL(k) parser (written as rules to be parsed by antlr2 [2]). The grammar for the parser is (using the EBNF notation) given in the html file **AaParser.html** which is part of this distribution. The set of tokens recognized by the lexical analyzer (or lexer). is available in the html file **AaLexer.html**.

References

[1] Sameer D. Sahasrabuddhe, "A competitive pathway from high-level programs to hardware," Ph.D. thesis, IIT Bombay, 2009.

[2] http://www.antlr2.org.