

# An introduction to the **Aa** language

Madhav P. Desai

March 8, 2018

# The **Aa** language

- ▶ Serves as an intermediate representation in the AHIR-V2 flow.
- ▶ Control-flow (imperative) language with support for parallelism and branching.

## A simple program in **Aa**

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

# Program-structure in **Aa**

```
aA_Program := ( aA_Module |  
                aA_Object_Declaration |  
                aA_Named_Type_Declaration)*
```

# A module

```
aA_Module := $module [identifier]
           $in ( aA_Module_Args)
           $out (aA_Module_Args)
{
    aA_Object_Declaration*
    aA_Atomic_Statement_Sequence
}
```

# Aa Types

**Aa** provides a comprehensive set of types.

- ▶ *Unsigned integers*

`$uint<1>`, `$uint<32>` etc.

- ▶ *Signed integers*

`$int<1>`, `$int<32>` etc.

- ▶ *Sized floats*

`$float<8,32>`, `$float<11,52>`

- ▶ *Pointers :*

`$pointer<$uint<32>>` etc..

- ▶ *Arrays:*

`$array[32][4] $of $uint<32>` etc.

- ▶ *Records:*

`$record $uint<32> $uint<1>`

- ▶ *Named Records:*

`$record [MyRec] $pointer<MyRec>) $uint<32>`

# Aa Objects

- ▶ Storage objects:

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

- ▶ Constant objects:

```
$constant A: $uint<4> := _b0011
```

```
$constant B: $uint<32> := _hf0f0f0f0
```

```
$constant C: $float<8,23> := _f2.3465e+0
```

- ▶ Pipe objects:

```
$pipe A: $uint<32> $depth 4
```

```
$lifo $pipe B: $uint<8> $depth 8
```

```
$noblock $pipe C: $uint<39> $depth 2
```

- ▶ Implicit objects: these are defined by statements:

```
a := (A + B)
```

They are also called *static-single-assignment* or SSA variables.

- ▶ Interface objects: Inputs and outputs of modules are treated like SSA variables. Inputs cannot be written into.

# Aa Storage Objects

- ▶ Will be implemented in memory spaces.
- ▶ Access to a storage object takes 3+ cycles.
- ▶ Should use sparingly.



# Aa Pipe Objects

- ▶ Blocking in nature: provide synchronization mechanism.
- ▶ FIFO or LIFO data transfer within and across modules.
- ▶ Non-blocking pipes: return a 0 if read is attempted on an empty pipe.

# Aa SSA Objects

- ▶ Implemented as registers.
- ▶ Fast access (immediate read, single cycle write).
- ▶ Should use as often as possible.
- ▶ Note that the type of these objects is inferred from the context in which they appear.

$a := (A + B)$

$A$  and  $B$  must have the same type, and the type of  $a$  is inferred to be the type of  $A$ .

# Aa Expressions

- ▶ Constants:

`_b00011`

`_habf1`

- ▶ Simple references:

`a`

- ▶ Array references:

`a[0][1]`

`a[(I+1)][J][K]`

- ▶ Unary expressions:

`(<op> expr)`

e.g. `(~ a)`

- ▶ Binary expressions:

`(a <op> B)`

`<op>` can be `+, -, *, /, <<, >>, |, &, ~|, ~&, ^, ~^`  
`==, !=, >, >=, <, <=`

# Aa More Expressions

- ▶ Ternary expressions:

`($mux <test-expr> <if-expr> <else-expr> )`

e.g. `($mux (a > 0) (b+1) (b-1))`

- ▶ Concatenation expression:

`(a && b)`

- ▶ Bit-select expression:

`(a [] I)`

- ▶ Address-of expression:

`@(a)`

`@(a[I])`

- ▶ Pointer-dereference expression:

`->(ptr)`

If it appears on the left-hand-side, it is a store, else it is a load.

# Aa Statements

- ▶ *Atomic* statements.
- ▶ *non-Atomic* statements.

# Atomic **Aa** Statements

- ▶ *Simple* statements.
- ▶ *Block* statements.

# Aa Atomic Simple Statements

- ▶ Assignment statements:

`target-expression := source_expression`

e.g.

`a := (b + c)`

- ▶ Call statements:

`$call fpadd32 (A B) (C)`

# Aa Atomic Block Statements

- ▶ Series-block statements.
- ▶ Parallel-block statements.
- ▶ Branch-block statements.
- ▶ Fork-block statements.



# Aa Series Block Statements

```
$seriesblock [SB] {  
    $storage a: $uint<32>  
    a := (b + c)  
    d := (a + e)  
} (d => D)
```

Control-flow is sequential: statements are executed in order, token leaves statement after last statement finishes. A module body is also a series-block.

# Aa Parallel Block Statements

```
$parallelblock [PB] {  
    a := (b + c)  
    p := (q + r)  
}
```

Control-flow: both statements started in parallel, token leaves statement after both statements have finished.

## Aa Branch Block Statements

```
$branchblock [BB] {  
    $merge $entry loopback  
        $phi I := ($bitcast($uint<32>) 0) $on $entry  
            NI $on loopback  
  
    $endmerge  
    a[I] := (b[I] + c[I])  
    NI := (I+1)  
    $if (NI < 16) $then  
        $place [loopback]  
    $endif  
}
```

Control-flow: sequential, but control flow is altered by merge, place, if and switch statements.

# Aa Phi Statements

- ▶ Can appear only in branch blocks, and within merge statements.
- ▶ Indicate the action to be taken depending on how control got to this statement.
- ▶ Example:

```
$phi I := ($bitcast($uint<32>) 0) $on $entry  
NI $on loopback
```

## Aa Do-Pipeline-While Statements

These are not atomic, and can occur only inside a branch-block.

```
$dopipeline $depth 7 $buffering 1 $fullrate
    $merge $entry $loopback
        $phi I := ($bitcast ($uint<32>) 0)
                $on $entry
                NI $on $loopback
    $endmerge
    a[I] := (b[I] + c[I])
    NI    := (I+1)
$while (I < 16)
```

Control-flow: sequential, controlled by the condition check. The places \$entry and \$loopback are implicitly defined. When control enters the do-while, the token gets placed in \$entry and when control loops-back from the condition check, the token gets placed in \$loopback. The compiler will pipeline the loop by keeping \$depth iterations alive.

# Aa Fork Block Statements

```
$forkblock [FB] {  
    $seriesblock [S1] { ... }  
    $seriesblock [S2] { ... }  
  
    $join S1 S2 $fork  
        $seriesblock [S3] { ... }  
        $seriesblock [S4] { ... }  
    $endjoin  
  
    $join S3 S4 $endjoin  
}
```

Control-flow: all statements will start in parallel, join-forks will trigger new statements etc. Token exits block when all statements finish.

# Aa Volatile Statements

- ▶ If you do not want to use a register:

`$volatile p := (q + r)`

- ▶  $p$  will not necessarily be implemented as a register. You can only assume that  $p$  is equivalent to  $(q + r)$ .
- ▶ Useful in describing combinational logic.

# Aa Summary

- ▶ An **Aa** program consists of a collection of module descriptions and object declarations.
- ▶ Each module is a sequence of statements.
- ▶ Statements can be simple or block-structured.
- ▶ Specific blocks allow branching, while others describe parallelism.
- ▶ Objects in an **Aa** program can be FIFOs (pipes), storage or implicitly inferred.
- ▶ Almost everything you can express in C can be expressed, but we are closer to hardware.



Look at the FIR again