About **llvm2aa**

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

April 30, 2021

1 Introduction

llvm2aa is a tool which reads in LLVM byte-code (see http://www.llvm.org for details about LLVM) and produces **Aa** code which can then be further used to produce VHDL using the AhirV2 tool chain developed at IIT Bombay.

2 Synopsys

The typical usage of the tool is

```
llvm2aa [-modules=<listfile>] [-storageinit] [llvm-passes] bytecode.o > bytecode.aa
```

The generated **Aa** code is sent to **stdout** and all informational messages are sent to **stderr**. On success, the tool returns 0.

The options:

- **–modules=listfile** : Specify the list of functions in the bytecode which should be converted to **Aa** . The names of these functions should be listed in the text-file listfile. One function name must be present on each line (the **exact** function name should be provided on the line, without leading or trailing spaces). Lines which start with the character '#' are ignored as comments. If this option is not specified, all functions are converted.
- **–storageinit** : Storage objects in the llvm bytecode are explicitly initialized in the generated **Aa** code. An initializer routine **global_storage_initializer_** is instantiated in the **Aa** code for this purpose.
- **-pipedepths=depthfile** : Specify the maximum depths of pipes in the generated Aa program. The file "depthfile" is a list of pairs (one pair per line). The first element of the pair specifies a pipe name, and the second, its maximum depth. The default depth of any pipe is 1.

- **-extract_do_while=true**: If specified, mark inner loops as do-while loops, whose implementation will be pipelined. An inner loop is a basic block whose terminator statement has a branch back to the beginning of the basic block. This optimization is suppressed if the inner loop body contains a call to the special function **nooptimize**. If not specified, inner loops are not marked as do-while loops.

- **llvm2aa** uses the LLVM compiler infrastructure to perform LLVM byte-code optimizations. A large list of these optimizations is available through the llvm2aa command-line. For more details, see LLVM documentation at http://www.llvm.org.

3 Limitations

Several LLVM byte-code constructs are not supported. Most importantly:

- Function pointers are not supported.

- Functions with a variable number of arguments are not supported.

- Calls to LLVM intrinsics are just passed through to the output **Aa** file. The **Aa** file will then contain calls to these intrinsics without there being a corresponding module declaration in the **Aa** file.

- If the LLVM byte-code has cycles in its call graph, then the code is translated, but will create an error in downstream **Aa** analysis and transformation tools.

- System calls made from the **Aa** code are simply passed through and would need to be supplied as an **Aa** library in order to perform downstream analysis and transformation.

- The LLVM integer, floating-point, array, structure, vector and void types are the only ones currently supported.

- The LLVM indirect-branch, invoke, unwind and unreachable instructions will not be supported.

- The LLVM division and remainder instructions are currently not supported.

- LLVM vector instructions are currently not supported.

- LLVM aggregate instructions (extractvalue, insertvalue) are currently not supported, but will be supported in the near future.

4 Examples

Let us start with the following C program, kept in file "add.c".

```
int add(int a, int b)
{
   int c = (a+b);
   return(c);
}
```

We will first need to compile this program down to LLVM byte code. For this, we use the **clang** compiler (http://www.clang.org)

```
clang -std=gnu89 -emit-llvm -c add.c
```

This produces an LLVM byte-code file **add.o**, which contains a compiled version of the function in the file shown above. This is our starting point.

We use

```
llvm2aa  -storageinit add.o > add.o.aa
```

to generate an **Aa** version of the LLVM bytecode. All functions in the LLVM bytecode will be translated and initial values of globally declared objects will be ignored. The **Aa** file that is produced is

```
// Aa code produced by llvm2aa (version 1.0)
$module [add]
$in (a : $uint<32> b : $uint<32> )
$out (ret_val__ : $uint<32>)
$is
{
  $storage stored_ret_val__ : $uint<32>
  $branchblock [body]
  {
    //begin: basic-block bb_0
    $storage iNsTr_0_alloc : $uint<32>
    $storage iNsTr_1_alloc : $uint<32>
    $storage c_alloc : $uint<32>
    iNsTr_0 := @(iNsTr_0_alloc)
    iNsTr_1 := @(iNsTr_1_alloc)
    c := @(c_alloc)
    ->(iNsTr_0) := a
    ->(iNsTr_1) := b
    // load
    iNsTr_4 := ->(iNsTr_0)
    // load
    iNsTr_5 := ->(iNsTr_1)
    iNsTr_6 := (iNsTr_4 + iNsTr_5)
    ->(c) := iNsTr_6
```

```
    // load
    iNsTr_8 := ->(c)
    stored_ret_val__ := iNsTr_8
    $place [return__]
    $merge return__ $endmerge
    ret_val__ := stored_ret_val__
  }
}
```