AhirV2: from algorithms to hardware
A tutorial introduction

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

April 27, 2011

1 Introduction

We describe the essential features of the second version of the AhirV2 toolset developed at IIT-Bombay.

2 What is AhirV2?

Perhaps the simplest way to understand AhirV2 is through an example.

Suppose we start with a simple program.

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

This program describes an algorithm. We normally compile this program and convert it to a machine-level program which is executed on a processor.

What if we wished to convert this program to a circuit? Such a circuit would have inputs a, b and an output c (together with some interface handshake signals). When activated, the circuit should read its inputs and compute a response which is the same as would be expected if the program were executed on a computer.

AhirV2 consists of a set of tools which takes a C/C++ program and converts this collection to a hardware circuit (described in VHDL). This conversion is done in two steps:

- The high-level program is compiled to an interemediate assembly form. AhirV2 introduces an intermediate assembly language **Aa** which can serve as a target for sequential programming languages (such as C/C++) as well as for parallel programming languages.

- From the **Aa** description, a virtual circuit (described in a virtual circuit description language **vC**) is generated. The chief optimization carried out at this step is that decomposing the system memory into disjoint segments based on usage analysis (this considerably improves the available memory bandwidth and reduces system cost).
- From the **vC** description, a VHDL description of the system is generated. The chief optimization carried out at this stage is resource sharing. The **vC** description is analyzed to identify operations which cannot be concurrently active and this information is used to reduce the hardware required.
- The VHDL description produced from **vC** is in terms of a library of VHDL design units which has been developed as part of the AhirV2 effort. This library consists of control-flow elements, data-path elements and memory elements.
- For the C/C++ to **Aa** translation, the AhirV2 toolset uses the front-end C/C++ compiler infrastructure developed by the LLVM project (www.llvm.org), which compiles the C/C++ program to LLVM byte-code (which is a processor independent representation of the compiled program). The AhirV2 toolset has a utility which translates the LLVM byte-code to an **Aa** description.

In Section 4, we illustrate the use of these tools on the simple example shown above.

The earlier version of the AHIR flow has been used to convert non-trivial programs to hardware. The resulting circuits are upto two orders of magnitude more energy-efficient than a processor [1]. The AhirV2 toolset incorporates several optimizations which make the resulting circuits more competitive.

3 The tools

We assume that you have access to either **llvm-gcc** or **clang** as the front-end compiler which generates LLVM byte-code from C/C++. The current AhirV2 toolset is consistent with llvm 2.8 and clang 2.8.

The other tools in the chain are described below.

3.1 llvm2aa

This tool takes LLVM byte code and converts it into an **Aa** file.

```
llvm2aa [-modules=<listfile>] [-storageinit] 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. If absent, 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.

3.2 AaLinkExtMem

This tool takes a list of **Aa** files, elaborates the program, does memory space decomposition. The externally visible memory space is linked in one of two ways: either it is assumed to be external and all accesses to it are routed out of the **Aa** program, or it is assumed to be internal and assumed to consist of a memory object (an array of bytes). External pointer dereferences are handled as if they are directed at this memory object.

```
AaLinkExtMem [-I n] [-E obj-name] 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:

- **-I n**: specifies that external references to memory are to be mapped as if they are to an internal object whose size is n bytes.

- **-E obj-name** : specifies that the object to which external references are mapped is to be named obj-name.

It is better if you use the **-I** and **-E** options.

If the **-I** option is not used, then all external memory references are routed out of the **Aa** program through pipes. In this case, if the **Aa** compiler determines that there is some pointer in the program which can point to both internal and external memory, then this will be declared as an error!

3.3 Aa2VC

This tool takes a list of **Aa** programs and converts them to a **vC** description.

```
Aa2VC [-O] [-C] [-I obj-name] file1.aa file2.aa ...  > result.vc
```

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

The options:

- **-O** : if used, sequential statement blocks are parallelized by doing dependency analysis.

- **-C** : if used, a C stub is created for every module that is not called from within the system. These stubs can be used to interface to a VHDL simuator (or even drive hardware) to verify the VHDL code generated by downstream tools.
- **-I obj-name** : if specified, all external memory references are considered as being directed at the storage object named obj-name.

3.4 vc2vhdl

Takes a collection of **vC** descriptions and converts them to VHDL.

```
vc2vhdl [-O] [-C] -t foo [-t bar -t bar2 ...]\
         -f file1.vc -f file2.vc ...  > system.vhdl
```

The options:

- **-t** : to specify the modules which are to be accessible from the ports of the generated VHDL system. Multiple top-level modules can be specified in this way.
- **-f** : specifies the **vC** files to be analyzed. Multiple **vC** files may be specified.
- **-O** : optimize the generated VHDL by compacting the control-path. This does not change the resulting hardware, but makes the generated VHDL file smaller.
- **-C** : the VHDL code has a system test bench which interfaces to foreign code using a VHPI/Modelsim-FLI interface. If this is not specified, the generated test bench simply instantiates the system and starts all top-level modules off (you will need to fill in your own test bench here). The C testbench is usually easier to write (it probably already exists in the form of the original program).

The tool performs concurrency analysis to determine operations which can be mapped to the same physical operator without the need for arbitration. It also instantiates separate memory subsystems for the disjoint memory spaces (in practice many of the memory spaces are small and are converted to register banks).

3.5 Miscellaneous: vcFormat and vhdlFormat

The outputs produces by **Aa2VC** and **vc2vhdl** are not well formatted. One can format **Aa** and **vC** files using **vcFormat** as follows

```
vcFormat < unformatted-vc/aa-file  > formatted-vc/aa-file
```

and similarly use **vhdlFormat** to format generated VHDL files.

4 An example

Let us revisit the simple example considered in the first section:

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

We wish to generate a circuit which *implements* the specification implied by this program.

We convert the program to LLVM byte code using the **clang** compiler (www.llvm.org)

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

This produces a binary file **add.o** which is the LLVM byte-code. To make the byte-code human readable, we dis-assemble it using an LLVM utility

```
llvm-dis add.o
```

This is what the LLVM assembly code looks like

```
; ModuleID = 'add.o'
target datalayout = "e-p ..... "
target triple = "i386-pc-linux-gnu"

define i32 @add(i32 %a, i32 %b) nounwind {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  store i32 %b, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* %c, align 4
  %6 = load i32* %c, align 4
  ret i32 %6
}
```

To get to this point, we could have used several optimizations which are available in the LLVM frame-work. But we work with the unoptimized version to illustrate the storage decomposition which is carried out by the AhirV2 tools.

The LLVM byte-code is our starting point. We first convert it to **Aa** .

```
llvm2aa add.o | vcFormat > add.o.aa
```

This produces an **Aa** program

```
// Aa code produced by llvm2aa (version 1.0)
$module [add]
// arguments
$in (a : $uint<32> b : $uint<32> )
$out (ret_val__ : $uint<32>)
$is
{
  $storage stored_ret_val__ : $uint<32>
  $branchblock [add]
  {
    //begin: basic-block bb_0
    $storage iNsTr_0 : $uint<32>
    $storage iNsTr_1 : $uint<32>
    $storage c : $uint<32>
    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__
  }
}
```

Now, this **Aa** code is converted to a virtual circuit **vC** representation.

```
Aa2VC -O add.o.aa | vcFormat > add.o.aa.vc
```

The virtual circuit representation is a bit too verbose to reproduce entirely here, but we show some critical fragments

```
$module [add]
{
  $in a:$int<32> b:$int<32>
  $out ret_val__:$int<32>
  $memoryspace [memory_space_0]
  {
    $capacity 1
    $datawidth 32
    $addrwidth 1
```

```
    // ret-val is kept here
    $object [xxaddxxstored_ret_val__] : $int<32>
  }
  $memoryspace [memory_space_1]
  {
    $capacity 1
    $datawidth 32
    $addrwidth 1
    // a is kept here.
    $object [xxaddxxaddxxiNsTr_0] : $int<32>
  }
  $memoryspace [memory_space_2]
  {
    $capacity 1
    $datawidth 32
    $addrwidth 1
    // b is kept her
    $object [xxaddxxaddxxiNsTr_1] : $int<32>
  }
  $memoryspace [memory_space_3]
  {
    $capacity 1
    $datawidth 32
    $addrwidth 1
    // c is kept here.
    $object [xxaddxxaddxxc] : $int<32>
  }
  $CP
  {
     // a control-flow petri-net..  verbose..
  }
  // end control-path
  $DP
  {
     // wires and operators.
  }

   // links between CP and DP
}
```

The important points to note are that the stored objects a,b,c and ret_val__ are mapped to different memory spaces. Thus, the chief difference between a **vC** description and a processor is that the **vC** program partitions storage into small units which are accessed only by operators that need them.

Finally, we take the **vC** description and convert it to VHDL

```
vc2vhdl -t add -f add.o.aa.vc | vhdlFormat > system.vhdl
```

This produces a VHDL implementation of the system with **add** marked as a top-level module. The VHDL that is produced is too voluminous to reproduce here, but the top-level system entity is

```
entity test_system is  -- system
  port (--
    add_a : in  std_logic_vector(31 downto 0);
    add_b : in  std_logic_vector(31 downto 0);
    add_ret_val_x_x : out  std_logic_vector(31 downto 0);
    add_tag_in: in std_logic_vector(0 downto 0);
    add_tag_out: out std_logic_vector(0 downto 0);
    add_start : in std_logic;
    add_fin   : out std_logic;
    clk : in std_logic;
    reset : in std_logic); --
  --
end entity;
```

There are ports corresponding to the arguments of the top-level module, and the add_start/add_fin is a handshake pair. One sets of the inputs, starts the system and waits until the fin is asserted. After the fin is asserted, one has the return value of at the appropriate port.

5 External and internal memory spaces

Consider the following C program:

```
int main(int* b)
{
   int q[2];
   q[0] = *b;
   q[1] = q[0];
   return(q[1]);
}
```

When this program is mapped to a circuit, we identify two distinct memory spaces, one which contains the array q and the other corresponding to the external world (the one referred to by the pointer b). Where is the external memory physically located? In the AhirV2 flow, we can either locate it outside the system or inside the system which is being described by this program.

If the external memory is to be placed outside, then accesses to it from within the system must be routed outside the system. On the other hand if it is to be placed inside, a storage object corresponding to it must be created and all accesses to the external memory must be directed at this storage object. Further, the external world must have a mechanism for accessing this storage object.

Both options are supported in the AhirV2 flow through the utility **AaLinkExtMem** described earlier.

5.1 Keeping the external memory outside the system

For this example, if you want to keep the external memory outside, you will have to go through the following sequence

```
# first use clang (or llvm-gcc) to generate llvm-byte-code
clang -std=gnu89 -emit-llvm -c foo.c
#
# disassemble so that you can make sense of the llvm bc.
llvm-dis foo.o
#
# OK, now take the llvm byte code
# and generate an Aa description.
# use the storageinit option to initialize
# global storage.
# (the pipe to vcFormat is to prettify the output)
 llvm2aa -storageinit foo.o | vcFormat > foo.o.aa
#
#
# Do an Aa -> Aa transformation: map external
# memory outside..
AaLinkExtMem foo.o.aa | vcFormat > foo.o.memlinked.ExternalOutside.aa
#
# Now take the Aa code and generate a virtual
# circuit..
# the -O flag does dependency analysis in straight-line
# code and parallelizes it.
#
Aa2VC -O foo.o.memlinked.ExternalOutside.aa | vcFormat\
                > foo.o.memlinked.ExternalOutside.aa.vc
#
# finally, generate vhdl from the vc description.  Note that
# you will have to mark the module foo as well as the
# extmem_store_32/load_32 modules as top-level modules
# so that it is possible for the outside world to serve
# requests made from inside.
#
vc2vhdl -O -t foo -t extmem_store_32 -t extmem_load_32\
         -f foo.o.memlinked.ExternalOutside.aa.vc | vhdlFormat\
              > foo_o_aa_memlinked_external_outside_vc.vhdl
```

If you look at the generate top-level VHDL entity, its ports will be

```
entity test_system is  -- system
  port (--
    foo_b : in  std_logic_vector(31 downto 0);
```

```
    foo_ret_val_x_x : out  std_logic_vector(31 downto 0);
    foo_tag_in: in std_logic_vector(0 downto 0);
    foo_tag_out: out std_logic_vector(0 downto 0);
    foo_start : in std_logic;
    foo_fin   : out std_logic;
    clk : in std_logic;
    reset : in std_logic;
    extmem_read_address_32_pipe_read_data: out std_logic_vector(31 downto 0);
    extmem_read_address_32_pipe_read_req : in std_logic_vector(0 downto 0);
    extmem_read_address_32_pipe_read_ack : out std_logic_vector(0 downto 0);
    extmem_read_data_32_pipe_write_data: in std_logic_vector(31 downto 0);
    extmem_read_data_32_pipe_write_req : in std_logic_vector(0 downto 0);
    extmem_read_data_32_pipe_write_ack : out std_logic_vector(0 downto 0);
    extmem_write_address_32_pipe_read_data: out std_logic_vector(31 downto 0);
    extmem_write_address_32_pipe_read_req : in std_logic_vector(0 downto 0);
    extmem_write_address_32_pipe_read_ack : out std_logic_vector(0 downto 0);
    extmem_write_data_32_pipe_read_data: out std_logic_vector(31 downto 0);
    extmem_write_data_32_pipe_read_req : in std_logic_vector(0 downto 0);
    extmem_write_data_32_pipe_read_ack : out std_logic_vector(0 downto 0)); --
  --
end entity;
```

The external memory read and write address and data are clearly visible. The outside world is responsible for serving the read/write requests made from the inside.

5.2 Keeping the external memory inside the system

For this example, if you want to keep the external memory outside, you will have to go through the following sequence

```
# use clang (or llvm-gcc) to generate llvm-byte-code
clang -std=gnu89 -emit-llvm -c foo.c
#
# disassemble so that you can make sense of the llvm bc.
llvm-dis foo.o
#
# OK, now take the llvm byte code
# and generate an Aa description.
# use the storageinit option to initialize
# global storage.
# (the pipe to vcFormat is to prettify the output)
 llvm2aa -storageinit foo.o | vcFormat > foo.o.aa
#
#
# Do an Aa -> Aa transformation: map external
# memory to a storage area inside the system...
```

```
# -I 1024 says that the amount of memory that will be
# referred to is 1024 bytes.
# -E mempool says that the storage object corresponding
# to external memory is named mempool.
AaLinkExtMem  -I 1024 -E mempool foo.o.aa | vcFormat\
               > foo.o.memlinked.ExternalInside.aa
#
# Now take the Aa code and generate a virtual
# circuit..
# the -O flag does dependency analysis in straight-line
# code and parallelizes it.
# the -I mempool option says that external memory is
# to be mapped inside the system to object mempool..
#
Aa2VC -O -I mempool foo.o.memlinked.ExternalInside.aa\
           | vcFormat > foo.o.memlinked.ExternalInside.aa.vc
#
# finally, generate vhdl from the vc description.
# note that you will have to mark mem_load__ and mem_store__
# as top-level modules, so that the external world can
# access its memory pool inside the system.
#
vc2vhdl -O -t foo -t mem_load__ -t mem_store__ \
     -f foo.o.memlinked.ExternalInside.aa.vc\
       | vhdlFormat > foo_o_aa_memlinked_external_inside_vc.vhdl
```

The generated top-level VHDL entity has the following ports:

```
entity test_system is  -- system
  port (--
    foo_b : in  std_logic_vector(31 downto 0);
    foo_ret_val_x_x : out  std_logic_vector(31 downto 0);
    foo_tag_in: in std_logic_vector(0 downto 0);
    foo_tag_out: out std_logic_vector(0 downto 0);
    foo_start : in std_logic;
    foo_fin   : out std_logic;
    mem_load_x_x_address : in  std_logic_vector(31 downto 0);
    mem_load_x_x_data : out  std_logic_vector(7 downto 0);
    mem_load_x_x_tag_in: in std_logic_vector(0 downto 0);
    mem_load_x_x_tag_out: out std_logic_vector(0 downto 0);
    mem_load_x_x_start : in std_logic;
    mem_load_x_x_fin   : out std_logic;
    mem_store_x_x_address : in  std_logic_vector(31 downto 0);
    mem_store_x_x_data : in  std_logic_vector(7 downto 0);
    mem_store_x_x_tag_in: in std_logic_vector(0 downto 0);
    mem_store_x_x_tag_out: out std_logic_vector(0 downto 0);
    mem_store_x_x_start : in std_logic;
```

```
    mem_store_x_x_fin   : out std_logic;
    clk : in std_logic;
    reset : in std_logic); --
  --
end entity;
```

The system provides a memory load and memory store port to the external world (through the mem_load.. and mem_store..) ports.

6 Handling complex programs

This example is trivial. Real, non-trivial programs can be mapped in this manner. Currently, there are only two restrictions

- No recursion, no cycles in the call-graph of the original program.
- No function pointers.

The AhirV2 flow produces a modular system, and by default, produces one VHDL entity for every function in the original program. Further, the concept of message pipes is native to the **Aa** and **vC** descriptions. Thus, it is easy to map parallel programs or programs consisting of multiple concurrent processes to hardware.

7 To be continued

This document is a work-in-progress. More details and related documentation will be added shortly.

The examples folder contains some explanations which might make things easier to understand.

References

[1] Sameer D. Sahasrabuddhe, Sreenivas Subramanian, Kunal P. Ghosh, Kavi Arya, Madhav P. Desai, "A C-to-RTL Flow as an Energy Efficient Alternative to Embedded Processors in Digital Systems," DSD, pp.147-154, 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, 2010