Getting started with the AHIR tools

Sameer D. Sahasrabuddhe

February 2008

1 Overview

The compilation flow that generates an AHIR circuit specification from an input C program is a three-stage process:

1. parsing the C program using a standard compiler front-end (`llvm-gcc`, provided by LLVM)
2. generating an unlinked AHIR specification from the compiler's standard IR (`irgen`, implemented using LLVM)
3. linking the AHIR specification and generating a memory map that is assumed by the linked circuit (`irlink`)

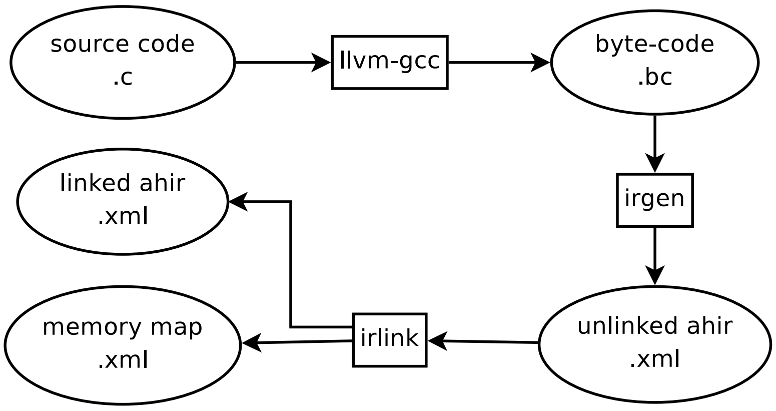

Figure 1: Compiling input source to AHIR

The linker stage is meant to allow separate compilation of "source libraries". For now the tools only process a single standalone C file, that must include all the functions required to create a complete working application.

2 Expected input

Currently, the toolchain works with a single C file that contains all the functions of a program. There are a number of limitations on the C code accepted.

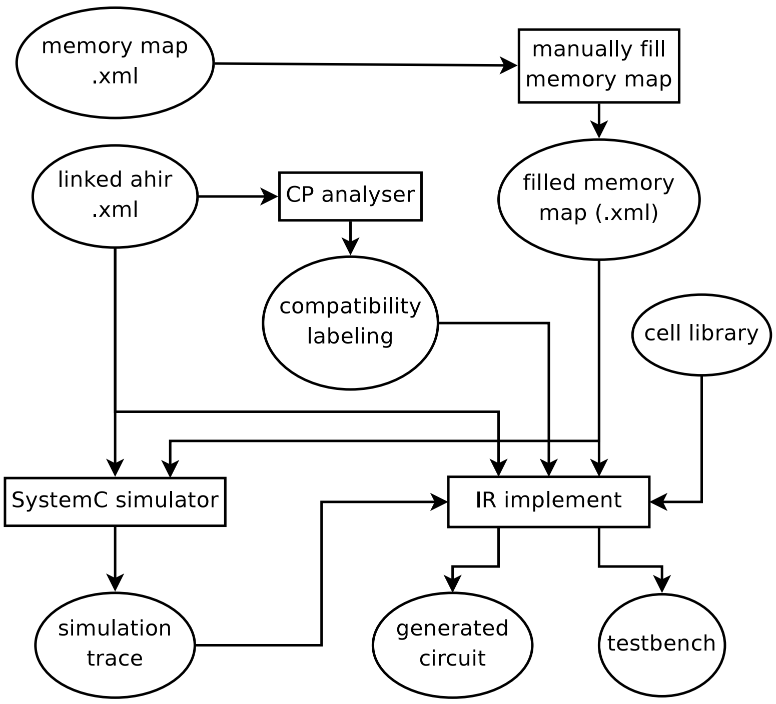

Figure 2: Tools for processing an AHIR specification

2.1 Restrictions imposed by the AHIR platform

1. The program's call-graph should be a DAG. This condition is currently not checked, so the result will be undefined if a cyclic call-graph is provided as input.
2. Function pointers are not allowed.
3. Functions with variable number of arguments are also not allowed.
4. Dynamic memory allocation and other system calls are not possible. They may be implemented by an application internally, but they are not part of the AHIR platform.

2.2 Limitations in the current implementation

1. The complete program must be present in a single file.
2. The complete DAG must contain a function called `start`, which denotes the root of the call-graph. The function `main` should not be used, since it will trigger internal modifications by the C front-end, that are incompatible with the ahir tools.
3. Bit selection operators have not been implemented.

2.3 Fully supported C constructs

1. 32-bit signed and unsigned integers, 32-bit floats and pointers (all other sizes are automatically typecast to 32 bits)
2. Arithmetic and bit-wise operators

3. Pointers to data
4. Structures and arrays
5. Static memory allocation
6. Control structures — `if-then-else`, `while-do`, `do-while`, `for`, `break` and `continue`

2.4 Running example

The rest of the document assumes an example input file named `sample.c`. This file contains a C-program with three functions, and a call-graph as follows:

$$\texttt{start} \rightarrow \texttt{foo} \rightarrow \texttt{bar}$$

3 Source compiler

The input C source is parsed by the LLVM front-end, `llvm-gcc`. This produces a bytecode file that contains the input program in LLVM's native IR. The command is invoked as follows:

```
llvm-gcc sample.c -o sample.bc
```

Note that `llvm-gcc` is part of the LLVM toolkit. It has a number of command-line options to control its behaviour. The installation script for the AHIR tools creates an alias in the users environment that includes all the relevant switches.

4 AHIR-XML generator

The LLVM bytecode generated by the front-end is used as the actual input that is converted into an AHIR specification. The command `irgen` is an LLVM-based tool that reads the bytecode, converts it to AHIR, and generates an XML file describing the AHIR virtual circuit.

```
irgen sample.bc
```

This produces an XML file, which contains unlinked AHIR-XML descriptions of the functions `start`, `foo` and `bar`. It also contains meta-information about the call-graph that constitutes `sample.c`

Since `irgen` is an LLVM tool, it has access to all the transformations and optimisations that are available as part of the LLVM framework. These can be specified by command-line arguments to the `irgen` command. The entire list of available operations can be viewed by invoking `irgen` as follows:

```
irgen -help
```

5 AHIR-XML linker

The unlinked XML specification also includes the call-graph of the original program. This information is required to provide predetermined memory locations called *postboxes* used to pass arguments during function calls. The modules in the unlinked specification refer to these locations only through names, since their addresses are not yet known.

The linker assigns numerical addresses to all the postboxes required for various function calls. In addition, it also creates locations for global variables as well as statically allocated variables. It is invoked on the unlinked XML as follows:

```
irlink sample.xml
```

This produces a linked version of the input XML, in a file called `sample_linked.xml`. Additionally, a memory map is created with the name `sample_map.xml`.

The memory map is an XML file that describes known memory locations ordered by their addresses. The format includes tags used to describe structured values as well as scalar values. The map itself is divided in two sections:

`init`: the initial state of the memory at invocation of the circuit

`fin`: the expected state of the memory after execution

As a built-in precaution, `irlink` never over-writes an existing memory map. When `irlink` is run, if the file `sample_map.xml` already exists, it instead creates a second file called `sample_new_map.xml`. This second file is not protected, and will be over-written by subsequent runs.

6 SystemC simulator

The linked AHIR specification can be simulated using SystemC. The command `irsim` reads an AHIR description and a memory map, to generate a set of SystemC modules in-memory. The modules are run until the control-path of the module for the `start` function triggers its `fin` transition, marking the end of execution.

```
irsim --ir sample_linked.xml --map sample_map.xml
```

The simulator uses the `init` section of the memory map to initialise the simulated memory, but ignores the contents of the `fin` section.

The simulation results in a trace of the memory accesses seen by the memory subsystem during execution. For each access to the memory subsystem, the trace records the time of the access along with the port and the address used.

Note that the simulator does not generate any actual SystemC description for now. It simply instantiates a SystemC objects that represent AHIR structures, and directly runs them.

7 VHDL generator

The VHDL generator is a direct mapping of the AHIR specification to an implementation. The generator creates a VHDL entity for each module in the specification. These entities use standard names and interfaces to instantiate and connect building blocks that are assumed to be available in an independent library.

```
irsyn --ir sample_linked.xml --map sample_map.xml
```

This will generate the following set of VHDL files:

- `start_cp.vhdl`, etc — one control-path for each function
- `start_dp.vhdl`, etc — data-path
- `start_ln.vhdl`, etc — intra-module link layer
- `omega.vhdl` — a single inter-module link layer
- `memory.vhdl` — the memory subsystem
- `system.vhdl` — built from all of the above
- `testbench.vhdl` — used to interact with the system

The testbench contains an array of values parsed from the `init` section of the memory map. These are used to initialise the memory subsystem. Similarly the testbench also contains values specified in the `fin` section, that are used to verify the contents of the memory at the end of execution.