Vericert OOPSLA 2021 Artifact

This artifact should support the claims made in the paper "Formal Verification of High-Level Synthesis". In the paper, our tool Vericert was referred to as using the temporary name "HLSCert". The claims that can be verified by the paper are the following:

- The mechanised proof of correctness of the translation from C into Verilog is provided and can be checked and rerun.
- All 27 PolyBench benchmarks can be recompiled using Vericert.
- The cycle counts of Vericert on the benchmarks can be checked and compared against LegUp 4.0.
- If Vivado is downloaded separately, then the whole performance section can be checked, including all the graphs that appear in the paper.

Artifact availability

The artifact is available on Github, specifically on the `oopsla21` branch:
`https://github.com/ymherklotz/vericert`
This release is also archived on Zenodo permanently:
`http://doi.org/10.5281/zenodo.5093839`
However, for the purposes of this artifact review, a Docker image has been set up:
`https://hub.docker.com/repository/docker/ymherklotz/vericert`

Claims that are not supported by the artifact

Unfortunately, we could not include our version of LegUp 4.0 in the artifact due to license restrictions. In addition to that, LegUp was recently bought by Microchip and renamed to SmartHLS[1], which means that the most recent versions of LegUp are closed source and cannot be downloaded anymore. The original open source version of LegUp 4.0 is not currently available either at the moment. The LegUp team have advised us that this is due to server issues in Toronto.[2] We have not heard back from them about whether it

[1] `https://www.microsemi.com/product-directory/fpga-design-tools/5590-hls#software-download`
[2] `https://legup.eecg.utoronto.ca`

is ok for us to share our copy of LegUp 4.0 for artifact evaluation purposes, so we have not done so.

Instead, we have included the net lists that LegUp generated from the benchmarks in the artifact, with all the optimisation levels that were tried, however, it does mean that these cannot be verified again and that other optimisation options cannot be tried.

In addition to that, the Vivado synthesis tool by Xilinx[3] is also commercial (but free to download), and therefore cannot be bundled into the artifact either. This synthesis tool was used to get accurate timing information about how the design would run on an FPGA, and also give the area that the design would take up on the FPGA. To be able to reproduce these results, it would therefore need Vivado to be set up so that the scripts can run.

Kick the tyres

First, the docker image needs to be downloaded and run, which contains the git repository:

```
docker pull ymherklotz/vericert:1.0
docker run -it ymherklotz/vericert:1.0 sh
```

Then, one just has to go into the directory which contains the git repository (`/vericert`) and open a `nix-shell`, which will load a shell with all the correct dependencies loaded:

```
cd /vericert
nix-shell
```

Then, all commands can be run in this shell, as well as `vericert`, which has already been compiled and can be found in the `/vericert/bin` directory. For a quick test that it is working, a few very simple examples in the `/vericert/test` directory can be run by using the following inside of the `/vericert` directory:

```
cd /vericert
make test
```

If this finishes without errors, it means that Vericert is working correctly.

Finally, to check that the benchmarks work correctly as well, we can quickly compile and run one as well:

```
cd /vericert/benchmarks/polybench-syn
make
./stencils/jacobi-1d
```

This simulates the hardware design generated for the jacobi-1d benchmark in PolyBench/C, and should print the return value: 1, as well as the cycle count: 19996 as follows:

[3] `https://www.xilinx.com/support/download.html`

```
cycles: 19996
finished: 1
```

Step-by-Step instructions

This section describes the detailed instructions to get the results for the different sections of the paper, first describing the structure of the proof and how to execute Vericert manually, to finally running Vericert on the benchmarks and get the cycle counts for the Vericert designs as well as the precompiled LegUp designs.

Directory structure of Vericert

The main directory structure of Vericert is the following:

`/src` Contains all the Coq and OCaml source files used for Vericert. The whole proof of correctness is therefore in this directory.

`/lib` This directory contains CompCert, on which Vericert is built upon. Vericert tries to separate CompCert and uses it only as a library, redefining a different top-level.

`/benchmarks` Contains the PolyBench/C benchmarks used as an evaluation in the paper, which are stored under `polybench-syn` for the benchmarks without dividers, and `polybench-syn-div` for the benchmarks with dividers.

`/docs` Contains a website and an `org-mode` file with some light documentation of the tool.

`/example` Contains some interesting observations that were made during the development, which are not directly relevant to Vericert.

`/include` Contains the divider implementation which can be imported and used in C files to get the better performance out of Vericert, instead of using native division.

`/ip` Contains hardware divider implementations which will be used in the future instead of the software implementation that is currently used in `/include`.

`/scripts` Contains some miscellaneous scripts and the `Dockerfile` which has been added for this artifcat.

`/test` Contains some very light test cases which are some minimal examples for working constructs.

Description of the proof

The proof is mostly located in `/src/hls`, which contains the proof of correctness of the 3AC to HTL transformation, as well as the transformation from HTL to Verilog. First, we will describe where each section of the paper is implemented, then a description of all the files in the src directory will be included.

Implementation of paper sections

When mentioning Coq source files, they will always be relative to the `/vericert/src` directory in the docker image.

- Section 2

  Figure 2 This example is not included in the repository or docker image, however, if the small C example in Figure 2a is copied into a file `main.v`, it can be compiled using the following:

  ```
  vericert -o main.v -O0 -drtl -dhtl main.c
  ```

  Where `-O0` means it will not apply any CompCert optimisations, `-drtl` means it will print the internal 3AC (also known as RTL) representation and `-dhtl` outputs the HTL representation. After running that command, Figure 2b should be the exact same as the `main.rtl.7` file that was generated, and Figure 2c should be the same as `main.v`, with some slight modifications to some variable names and formatting.

  Figure 3 After running the above command, Figure 3 will be a visual representation of ~main.

  Section 2.2.2 The abstract RAM description and is used in HTL can be found in `hls/HTL.v:139`. This also corresponds to Figure 7. This abstract description is then implemented as a concrete Verilog implementation shown in `hls/Veriloggen.v:45`. The proof that the Verilog implementation is correct according to the HTL specification of it can be found in Lemma `ram_exec_match`, `hls/Veriloggenproof.v:284`.

  Section 2.2.3 This proof is for Theorem `shrx_shrx_alt_equiv`, `common/IntegerExtra.v:661`.

- Section 3

  This Section is mainly implemented in `hls/Verilog.v`.

  Module execution rule The updated negative edge execution rule can be found in `hls/Verilog.v:582` which is called `step_module`, and has a `mis_stepp` and `mis_stepp_negedge` for the steps of the positive and negative edge triggered always blocks.

  Figure 5 This is implemented as all the other possible steps in one Verilog step, shown in `hls/Verilog.v:581`. The Figure just uses some nicer notation for the inference rules.

  Figure 6 Our dependenty typed arrays used for the memory model are implemented in `hls/Array.v`, and is then integrated in the Verilog semantics properly using the `arr_associations` type, defined in `hls/Verilog.v:60`, which is a blocking and nonblocking array where each element is an optional, as shown in Figure 6.

- Section 5

  Theorem 1 This is proven as Theorem `transf_c_program_correct` in `Compiler.v:415`.

  Lemma 1 This is proven as part of Theorem `cstrategy_semantic_preservation` in `Compiler.v:334`, which also proves the backward simulation at the same time.

  Lemma 2 The specification of the translation from 3AC to HTL is shown in Theorem `transl_module_correct` in `hls/HTLgenspec.v:608` and is called `tr_module` instead of `spec_htl` as in the paper, and `tr_htl` is called `transl_module` instead.

  Section 4.1.2, `match_states` The `match_states` property to match two states in 3AC and HTL up is defined in `hls/HTLgenproof.v:112`.

  Lemma 3 Proven as Theorem `transl_step_correct` in `hls/HTLgenproof.v:2856` and describes the simulation diagram shown in the paper.

  Section 4.2.1 The specification of the store is located in `hls/Memorygen.v:2096` and is called `alt_store`.

  Section 4.2.2, `match_states` The definition of matching states is defined in `hls/Memorygen.v:314`, where `ARRS_SIZE` corresponds to the property of equally sized arrays at each step and `DISABLE_RAM` corresponds to the property that the RAM is always disabled by default.

  Lemma 4 There is a small typo in the paper, and this Lemma should describe the forward simulation of the insertion of the RAM. This is proven in Theorem `transf_program_correct` in `hls/Memorygen.v:3196`, and the simulation diagram for this translation is proven in Theorem `transf_step_correct` in `hls/Memorygen.v:3000`.

  Lemma 5 This is proven in Theorem `transf_program_correct` in `hls/Veriloggenproof.v:537`. The assumption that the HTL module and Verilog module are related by `transl_program` (`tr_verilog` in the paper) is given in the hypothesis `TRANSL` in `hls/Veriloggenproof.v:343`.

  Lemma 6 The determinism of the Verilog semantics is proven in `semantics_determinate` in `hls/Verilog.v:810`.

  Table 1 These values were calculated by hand to separate specification, implementation and proof code. The raw results can be found in the last table in the `/data/data/results.org` file, or in the `/data/data/code-count.csv`.

Description of files

`/src/Compiler.v` The very top-level of the proof is located here and it contains the main proof of the compiler, which is the proof that the `transf_hls` function is correct, which takes C and outputs Verilog. The main proof of correctness is in the Theorem

called `transf_c_program_correct`, which says that if the `transf_hls` function succeeded, that the backward simulation should hold between C and Verilog.

`/src/common` This directory contains some common library extensions and proofs that are used in other parts of Vericert. This includes the proof of correctness of Section 2.2.3, which is located in `/src/common/IntegerExtra.v` under the Theorem `shrx_shrx_alt_equiv`.

`/src/hls/Verilog.v` This file contains the whole Verilog semantics, together with the proof that the Verilog semantics are deterministic. This implements Section 3 from the paper.

`src/hls/Veriloggen.v` This file contains the generation of Verilog from HTL.

`src/hls/Veriloggenproof.v` This file contains the correctness proof of the generation of Verilog from HTL.

`/src/hls/HTL.v` This file contains the definition of the HTL intermediate language, together with its semantics.

`/src/hls/HTLgen.v` This file contains the generation of HTL from 3AC, which is the first step in the HLS transformation.

`/src/hls/HTLgenspec.v` This file contains the high-level specification of the translation from 3AC into HTL, together with a proof of correctness of the specification.

`/src/hls/HTLgenproof.v` This file contains the proof of correctness of the HTL generation from 3AC, where the main parts of the proof are the generation of Verilog operations, as well as the change in the memory model (load and store instructions).

`/src/hls/Memorygen.v` This file contains the definition and proof of the transformation which replaces naïve loads and stores into a proper RAM, which is described in Section 2.2.2.

`/src/hls/ValueInt.v` Contains our definition of values that are used in the Verilog semantics, and differ from the values used by CompCert, as they don't have a pointer type anymore.

`/src/hls/Array.v` Contains our definition of the memory model, which is a dependently typed array, which encodes its length. This is much more concrete than CompCert's abstract memory model, and closer to how it is actually modelled in hardware.

`/src/hls/AssocMap.v` Definition of association maps, which is the type that is used for Γ and Δ in Section 3.

How to manually compile using Vericert

To compile arbitrary C files, the following command can be used:

```
vericert main.c -o main.v
```

Which will generate a Verilog file with a corresponding test bench. The Verilog file can then be simulated by using the Icarus Verilog simulator:

```
iverilog main.v -o main
./main
```

This should print out the return value from the main function in addition to the number of cycles that it took to execute the hardware design.

Getting cycle counts for Vericert

There are two benchmark sets for which the results are given in the paper:

`/vericert/benchmarks/polybench-syn` Contains the PolyBench/C benchmark without any dividers, and instead the dividers are replaced by calls to `sdivider` and `smodulo` in `/vericert/include/hls.h`.

`/vericert/benchmarks/polybench-syn-div` Contains the PolyBench/C benchmark with dividers.

To get the cycle counts for Vericert from the benchmarks, the benchmarks can be compiled using the following:

```
cd /vericert/benchmarks/polybench-syn
```

or

```
cd /vericert/benchmarks/polybench-syn-div
```

depending on which benchmark should be run, and then running:

```
make
```

This will generate all the binaries for the simulation and execution of the C code. The cycle counts of the hardware can then be gotten by running:

```
./run-vericert.sh
```

This can take a while to complete, as simulation of hardware is quite slow. After around 30 minutes, there should be a `exec.csv` file which contains the cycle counts for each of the 27 benchmarks.

Getting the cycle counts for LegUp

Unfortunately, the benchmarks cannot be compiled from C to Verilog using LegUp, as it could not be included in the artifact, and does not seem to be freely available anymore.

However, our compiled Verilog designs from LegUp have been included for all the optimisation options that were tested for in the paper in Section 5.

To get the cycle counts, it suffices to go into an arbitrary directory, and run the following script, where the command line arguments select which set of cycle counts to generate:

```
/vericert/scripts/run-legup.sh [syn|syn-div] \
                               [opt|no_opt|no_chain|no_opt_no_chain]
```

For example, to run the LegUp benchmarks with no LLVM optimisations and no operation chaining, on the PolyBench/C benchmark with no dividers, one can run the following command:

```
/vericert/scripts/run-legup.sh syn no_opt_no_chain
```

This will take some 30 minutes to run as well, and will generate an `exec_legup.csv` file, with the name of the benchmark and it's cycle count.

Comparing the results

To compare the results to the results presented in the paper, the main comparison that is supported by this artifact is to compare the cycle counts to the ones used to generate the graphs in the evaluation section of the paper.

The `/data/data` directory contains all the raw data which was used to generate the graphs in Section 5. This data can therefore be used to examine the cycle counts used to draw the graphs. This raw data can be examined better in `/data/data/results.org`, which includes the tables in a nicer format.

The `legup-*` csv files contain the raw size, timing and cycle count for the various LegUp configurations on the different benchmarks. `vericert-*` is the equivalent but for Vericert. Then, to draw the graphs, the actual csv files that are used are:

`rel-size-*` This contains the relative size of each run (denoted by slice in the csv files) compared to fully optimised LegUp. This is obtained by taking the slice value of the tool being considered (LegUp with some optimisation turned off, or Vericert), and dividing that by the number of slices present in fully optimised LegUp.

$$\frac{\text{slice}_t}{\text{slice}_{\text{legup\_opt}}}$$

`rel-time-*` This performs the same computation as for the size comparison, comparing to LegUp with all optimisations turned on, but instead compares the following values: cycles $\times$ delay:

$$\frac{\text{cycles}_t \times \text{delay}_t}{\text{cycles}_{\text{legup\_opt}} \times \text{delay}_{\text{legup\_opt}}}$$

Where t is the tool being considered.

Compiling the graph

A tex file is included in the `/data/data` directory, which unfortunately can only be compiled outside of the docker file, but will recreate the graphs from the paper using the csv files in the directory. This can be achieved using the following commands:

```
docker create ymherklotz/vericert:v1.0 # returns container ID
docker cp $container_id:/data/data .
docker rm $container_id
cd data
pdflatex graphs
```

Running with Vivado

Finally, for the adventurous that downloaded Vivado, there are some short instructions for running it on single examples. Running synthesis on a benchmark will normally take around 20 minutes to an hour depending on the benchmark, so it might take a long time to complete.

First, create a new directory and copy the synthesis script into it, as well as the Verilog file that should be synthesisd. For example, once `make` was run in the benchmarks folder, one of the benchmarks can be selected for Vericert, such as `jacobi-1d`:

```
mkdir synthesis
cd synthesis
cp /vericert/scripts/synth.tcl .
cp /vericert/benchmarks/polybench-syn/stencils/jacobi-1d.v main.v
```

Then Vivado can be run in batch mode in that directory to generate the report:

```
vivado -mode batch -source synth.tcl
```

Once this completes, the important results of the synthesis should be available in `encode_report.xml`, where each field will also be present in the relevant CSV file, which is this case is `/data/data/vericert-nodiv.csv`.

Rebuilding the Docker image

The docker image can be completely rebuilt from scratch as well, by using the Dockerfile that is located in the Vericert repository at `/vericert/scripts/docker/Dockerfile`, which also contains this document.

To rebuild the docker image, one first needs to download the LegUp results for the benchmarks without divider[4] and with divider[5], as well as the csv folder with all the raw results[6]. The tar files should be placed into the same directory as the `Dockerfile`. Then, in the `docker` directory, the following will build the docker image, which might take around 20 minutes:

```
docker build .
```

Then, using the hash it can be run in the same way as the docker container that was linked to this artifact:

```
docker run -it <hash> sh
```

Building from git without Docker.

The only dependency that is require is nix[7]. Once that is installed, we can clone the Github repository and checkout the `oopsla21` branch:

```
git clone https://github.com/ymherklotz/vericert
cd vericert
git checkout oopsla21
```

Then, it can be compiled and installed using:

```
nix-shell --run "make -j7"
nix-shell --run "make install"
nix-shell --run "./bin/vericert ./test/add.c -o add.v"
```

[4] https://imperialcollegelondon.box.com/s/ril1utuk2n88fhoq3375oxiqcgw42b8a
[5] https://imperialcollegelondon.box.com/s/94clcbjowla3987opf3icjz087ozoi1o
[6] https://imperialcollegelondon.box.com/s/nqoaquk7j5mj70db16s6bdbhg44zjn52
[7] https://nixos.org/download.html