Formal Verification of High-Level Synthesis

Yann Herklotz, James D. Pollard, Nadesh Ramanathan and John Wickerson

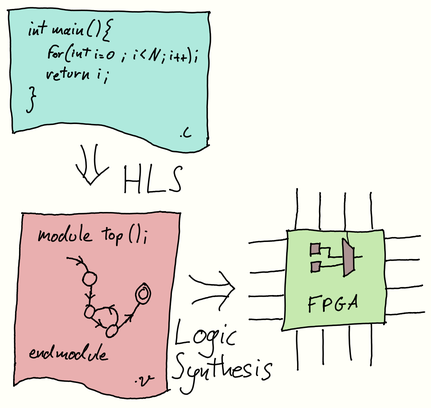
Designing Hardware for an FPGA Using HLS

Field-programmable gate arrays (FPGA) are a good alternative to CPU's for many applications.

High-level synthesis (HLS) is a promising method to program them.

1. **C code** is translated to **hardware**, described in **Verilog**.

2. **Verilog hardware description** is then placed onto an **FPGA** using a **logic synthesis tool**.

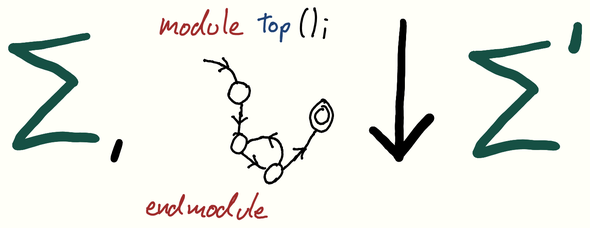


Integrate Verilog Semantics into Coq

Verilog semantics adapted from Lööw et al. [2019].

Small-step over clock edges:

Big-step within each clock edge:



Current HLS Tools Are Unreliable

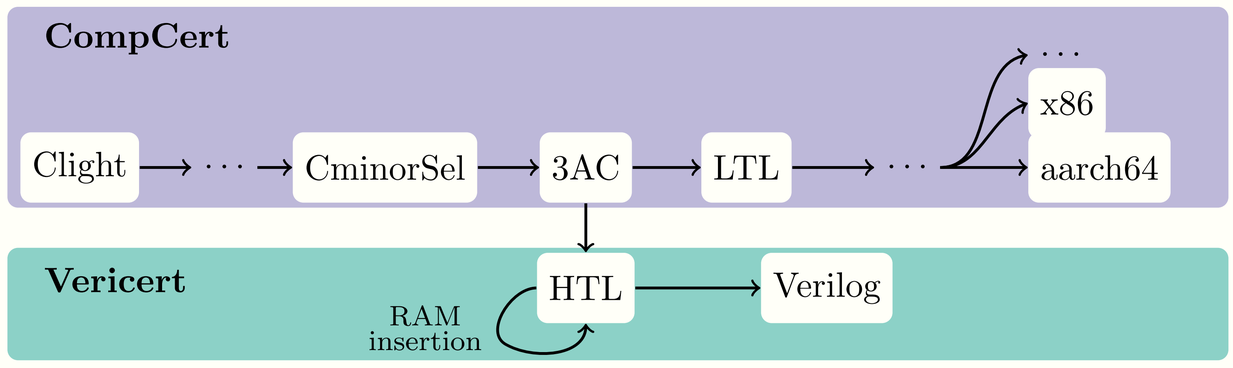
Run-time errors present in all existing HLS tools.

One bug was found in **Vericert** pretty printing, but none present when fixed.

Tool	Run-time errors
Vivado HLS	1.23%
Intel i++	0.4%
Bambu 0.9.7-dev	0.3%
LegUp 4.0	0.1%
Vericert	~~0.03%~~ **0%**

Extending CompCert to Formally Verify HLS

Create a Formally Verified HLS tool called **Vericert**, based on CompCert.



Example of translation from C into Verilog

Main translation is from a **control-flow graph** into a **finite state-machine with data path (FSMD)**.

HTL is an intermediate language representing a **FSMD** to ease the translation.

```
main() {
    x5 = 3
    int32[stack(0)] = x5
    x4 = 6
    int32[stack(4)] = x4
    x1 = 1
    x3 = stack(0) (int)
    x2 = int32[x3 + x1 * 4 + 0]
    return x2
}
```

CompCert 3AC code

Vericert HLS Translation

```
// Data-path
always @(posedge clk)
  case (state)
    32'd11: reg_2 <= d_out;
    32'd8: reg_5 <= 32'd3;
    32'd7: begin
      u_en <= ( ~ u_en); wr_en <= 32'd1;
      d_in <= reg_5; addr <= 32'd0;
    end
    32'd6: reg_4 <= 32'd6;
    32'd5: begin
      u_en <= ( ~ u_en); wr_en <= 32'd1;
      d_in <= reg_4; addr <= 32'd1;
    end
    32'd4: reg_1 <= 32'd1;
    32'd3: reg_3 <= 32'd0;
    32'd2: begin
      u_en <= ( ~ u_en); wr_en <= 32'd0;
      addr <= {{{reg_3 + 32'd0} + {reg_1 * 32'd4}} / 32'd4};
    end
    32'd1: begin finish = 32'd1; return_val = reg_2; end
    default: ;
  endcase
```

Data path Verilog block generated by Vericert.

```
// Control logic
always @(posedge clk)
  if ({reset == 32'd1}) state <= 32'd8;
  else case (state)
         32'd11: state <= 32'd1;        32'd4: state <= 32'd3;
         32'd8: state <= 32'd7;         32'd3: state <= 32'd2;
         32'd7: state <= 32'd6;         32'd2: state <= 32'd11;
         32'd6: state <= 32'd5;         32'd1: ;
         32'd5: state <= 32'd4;         default: ;
       endcase
endmodule
```

Control logic block generated by Vericert.

Execution Time Compared to Existing Unverified HLS Tools

Vericert compared to **LegUp** on 27 out of 30 **PolyBench/C benchmarks**.

Bad news: When **divisions** are **present**, **Vericert** is **27x** slower than **LegUp**.

Better news: When divisions are replaced by an **iterative division** algorithm, **Vericert** is only **2x** slower than **LegUp**.

Future Work

Scheduling: reduce performance gap by executing multiple instructions in a clock cycle. This would also solve issue with division by pipelining a hardware division operation.

Resource sharing: support proper function calls by sharing function implementation.

Globals: Increase language support and implement multiple memories.