# Formal Verification of High-Level Synthesis

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

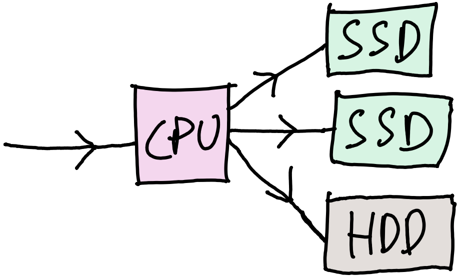
Imperial College London

CHICAGO SPLASH2021

## The Need to Design Hardware Accelerators

Application-specific hardware accelerators are increasingly being needed in industries.

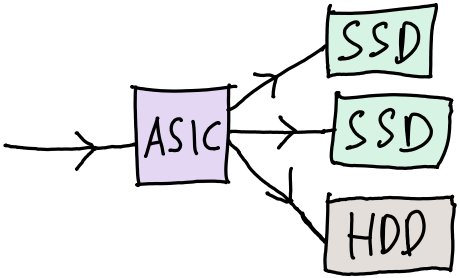
- Using a **CPU** everywhere not always the best choice.



## The Need to Design Hardware Accelerators

Application-specific hardware accelerators are increasingly being needed in industries.

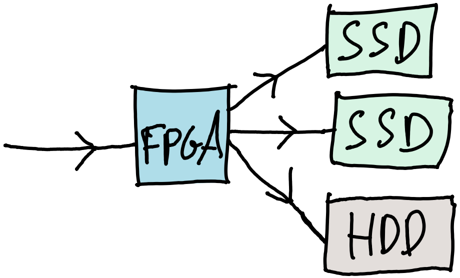
- Using a **CPU** everywhere not always the best choice.
- **Application-specific integrated circuits (ASIC)** are the ideal choice, but very expensive to create.



## The Need to Design Hardware Accelerators

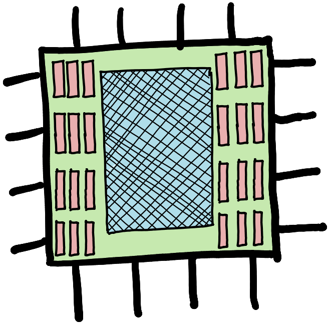
Application-specific hardware accelerators are increasingly being needed in industries.

- Using a **CPU** everywhere not always the best choice.
- **Application-specific integrated circuits (ASIC)** are the ideal choice, but very expensive to create.
- **Field-programmable gate arrays (FPGA)** act as **reprogrammable hardware**, therefore can be made application-specific.



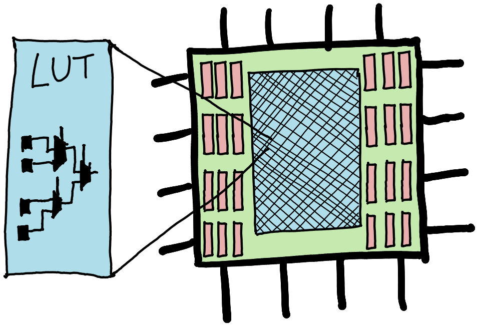
## Where does the flexibility of FPGAs come from?

- FPGA's are programmable circuits with two main components.



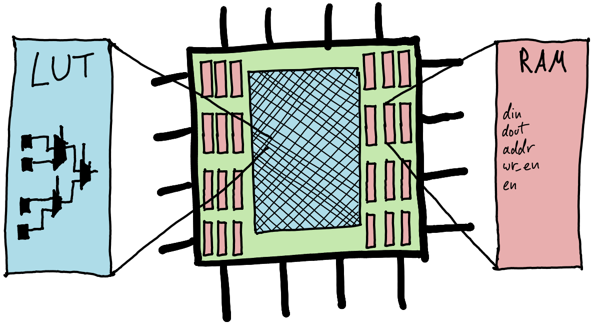
## Where does the flexibility of FPGAs come from?

- FPGA's are programmable circuits with two main components.
- **Look up tables (LUTs)** provide flexible logic gates. They are connected by **configurable switches**.



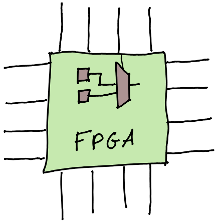
## Where does the flexibility of FPGAs come from?

- FPGA's are programmable circuits with two main components.
- **Look up tables (LUTs)** provide flexible logic gates. They are connected by **configurable switches**.
- **RAMs** provide accessible storage.



## So How do we Program an FPGA?

- FPGAs contain **LUTs** and programmable interconnects.

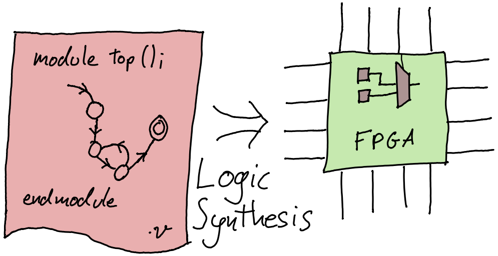


## So How do we Program an FPGA?

- FPGAs contain **LUTs** and programmable interconnects.
- Programmed using **hardware description languages**.

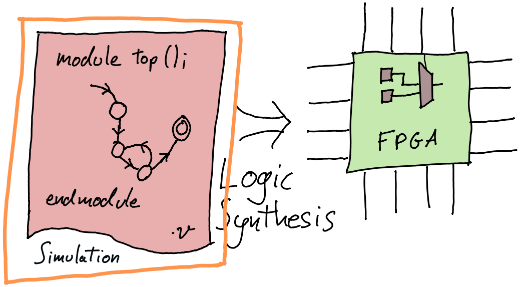
⬆ Fine control

⬇ Long to design



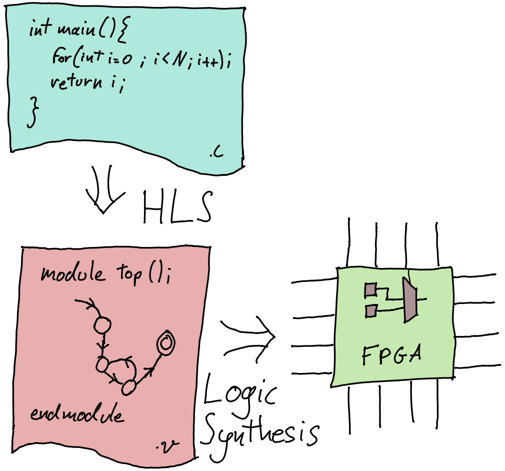
## So How do we Program an FPGA?

- FPGAs contain **LUTs** and programmable interconnects.
- Programmed using **hardware description languages**.
- Simulation quite slow.



## So How do we Program an FPGA?

- FPGAs contain **LUTs** and programmable interconnects.
- Programmed using **hardware description languages**.
- Simulation quite slow.
- High-Level Synthesis is an alternative.



```
int main(){
    for(int i=0 ; i<N; i++);
    return i;
}
                        .c
```

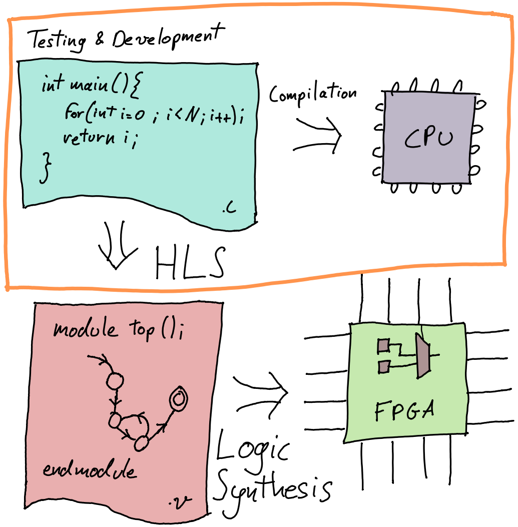
```
module top();

endmodule
                        .v
```

⬆ Quick to design ⬇ Less control

## So How do we Program an FPGA?

- FPGAs contain **LUTs** and programmable interconnects.
- Programmed using **hardware description languages**.
- Simulation quite slow.
- High-Level Synthesis is an alternative.
- Faster testing through execution.



## Motivation for Formal Verification

Difficult to debug HLS tools:

- Simulation can take a long time.
- Correctness is important in hardware, testing is done at every level.

## Motivation for Formal Verification

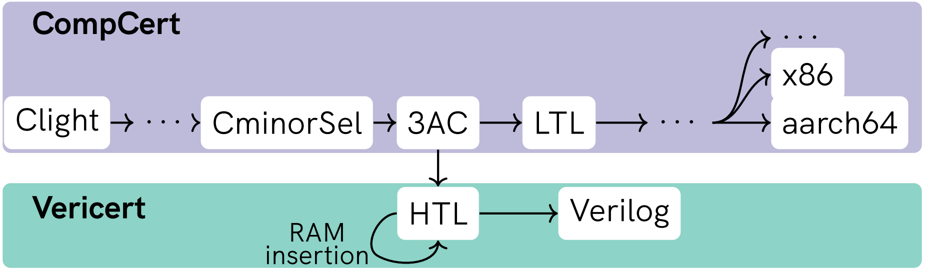
Difficult to debug HLS tools:

- Simulation can take a long time.
- Correctness is important in hardware, testing is done at every level.

High-level synthesis is often quite unreliable:

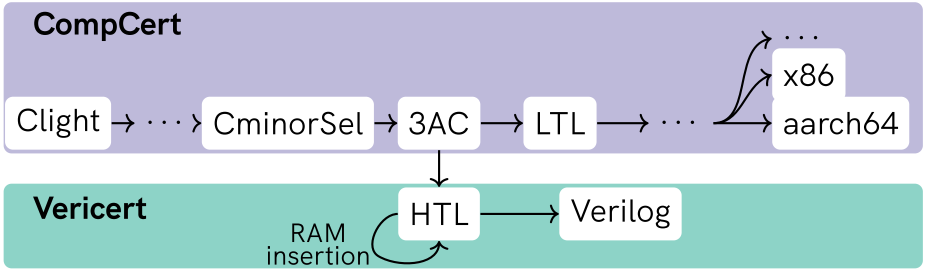
- We fuzzed HLS tools (Herklotz et al. [2021]) and found they failed on **2.5%** of simple random test cases.

## Solution



Use CompCert, a fully verified C compiler, and add an HLS backend.

## Solution



Support for: all **control flow**, **fixedpoint**, **non-recursive functions** and **local arrays/structs/unions**.

## Outline

## Example: 3AC

```c
int main() {
    int x[2] = {3, 6};
    int i = 1;
    return x[i];
}
```

- Example of a very simple program performing loads and stores.

## Example: 3AC

- **Three address code (3AC)** instructions are represented as a control-flow graph (CFG).
- Each instruction links to the next one.

```
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
}
```

## HTL Overview

The representation of the **finite state-machine with datapath** is abstract and called **HTL**.

```
Definition datapath := $\mathbb{Z}^+ \mapsto$ Verilog.stmnt
Definition controllogic := $\mathbb{Z}^+ \mapsto$ Verilog.stmnt
```

## HTL Overview

The representation of the **finite state-machine with datapath** is abstract and called **HTL**.

```
Definition datapath := ℤ⁺ ↦ Verilog.stmnt
Definition controllogic := ℤ⁺ ↦ Verilog.stmnt

Record module: Type := mkmodule {
    mod_datapath: datapath;
    mod_controllogic: controllogic;
    mod_reset: reg;
    mod_ram: ram_spec;
    ...
  }.
```

## HTL Overview

The representation of the **finite state-machine with datapath** is abstract and called **HTL**.

```
Definition datapath := ℤ⁺ ↦ Verilog.stmnt
Definition controllogic := ℤ⁺ ↦ Verilog.stmnt

Record module: Type := mkmodule {
    mod_datapath: datapath;
    mod_controllogic: controllogic;
    mod_reset: reg;
    mod_ram: ram_spec;
    ...
  }.
```

## HTL Overview

The representation of the **finite state-machine with datapath** is abstract and called **HTL**.

```
Definition datapath := ℤ⁺ ↦ Verilog.stmnt
Definition controllogic := ℤ⁺ ↦ Verilog.stmnt

Record module: Type := mkmodule {
    mod_datapath: datapath;
    mod_controllogic: controllogic;
    mod_reset: reg;
    mod_ram: ram_spec;
    ...
  }.
```

## HTL Overview

The representation of the **finite state-machine with datapath** is abstract and called **HTL**.

```
Definition datapath := ℤ⁺ ↦ Verilog.stmnt
Definition controllogic := ℤ⁺ ↦ Verilog.stmnt

Record module: Type := mkmodule {
    mod_datapath: datapath;
    mod_controllogic: controllogic;
    mod_reset: reg;
    mod_ram: ram_spec;
    ...
  }.
```

## Translation (3AC $\rightarrow$ HTL)

Translation from **control-flow graph** into a **finite state-machine with datapath**.

## Translation (3AC $\to$ HTL)

Translation from **control-flow graph** into a **finite state-machine with datapath**.

- **Control-flow** is translated into a **finite state-machine**.

## Translation (3AC $\rightarrow$ HTL)

Translation from **control-flow graph** into a **finite state-machine with datapath**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **3AC instructions** translated into equivalent **Verilog statements**.

## Translation (3AC $\rightarrow$ HTL)

Translation from **control-flow graph** into a **finite state-machine with datapath**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **3AC instructions** translated into equivalent **Verilog statements**.
- Call **stack** implemented as **Verilog array**.

## Translation (3AC $\rightarrow$ HTL)

Translation from **control-flow graph** into a **finite state-machine with datapath**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **3AC instructions** translated into equivalent **Verilog statements**.
- Call **stack** implemented as **Verilog array**.
- Pointers for loads and stores translated to array addresses.

## Translation (3AC $\rightarrow$ HTL)

Translation from **control-flow graph** into a **finite state-machine with datapath**.

- **Control-flow** is translated into a **finite state-machine**.
- Each **3AC instructions** translated into equivalent **Verilog statements**.
- Call **stack** implemented as **Verilog array**.
- Pointers for loads and stores translated to array addresses.
    - **Byte** addressed to **word** addressed.

## Memory Inference Pass

- An HTL $\rightarrow$ HTL translation removes loads and stores.
- Replaced by accesses to a proper **RAM**.

```
stack[reg_5 / 4]
```

becomes

```
u_en <= ( ~ u_en);
wr_en <= 0;
addr <= reg_5 / 4;
```

## Memory Inference Pass

- An HTL $\rightarrow$ HTL translation removes loads and stores.
- Replaced by accesses to a proper **RAM**.

```
stack[reg_5 / 4]
```

becomes

```
u_en <= ( ~ u_en);
wr_en <= 0;
addr <= reg_5 / 4;
```

## Memory Inference Pass

- An HTL → HTL translation removes loads and stores.
- Replaced by accesses to a proper **RAM**.

```
stack[reg_5 / 4]
```

becomes

```
u_en <= ( ~ u_en);
wr_en <= 0;
addr <= reg_5 / 4;
```

## Memory Inference Pass

- An HTL $\rightarrow$ HTL translation removes loads and stores.
- Replaced by accesses to a proper **RAM**.

```
stack[reg_5 / 4]
```

becomes

```
u_en <= ( ~ u_en);
wr_en <= 0;
addr <= reg_5 / 4;
```

## Translation (HTL $\rightarrow$ Verilog)

```
module main(reset, clk, finish, return_val);
  input [0:0] reset, clk;
  output reg [0:0] finish = 0;
  output reg [31:0] return_val = 0;
  reg [31:0] reg_3 = 0, addr = 0, d_in = 0,
             reg_5 = 0, wr_en = 0,
             state = 0, reg_2 = 0,
             reg_4 = 0, d_out = 0, reg_1 = 0;
  reg [0:0] en = 0, u_en = 0;
  reg [31:0] stack [1:0];
  // RAM interface
  always @(negedge clk)
    if ({u_en != en}) begin
      if (wr_en) stack[addr] <= d_in;
      else d_out <= stack[addr];
      en <= u_en;
    end
```

- Finally, translate the FSMD into Verilog.

## Translation (HTL $\rightarrow$ Verilog)

```
module main(reset, clk, finish, return_val);
  input [0:0] reset, clk;
  output reg [0:0] finish = 0;
  output reg [31:0] return_val = 0;
  reg [31:0] reg_3 = 0, addr = 0, d_in = 0,
             reg_5 = 0, wr_en = 0,
             state = 0, reg_2 = 0,
             reg_4 = 0, d_out = 0, reg_1 = 0;
  reg [0:0] en = 0, u_en = 0;
  reg [31:0] stack [1:0];
  // RAM interface
  always @(negedge clk)
    if ({u_en != en}) begin
      if (wr_en) stack[addr] <= d_in;
      else d_out <= stack[addr];
      en <= u_en;
    end
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.

## Translation (HTL $\rightarrow$ Verilog)

```
// 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
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.
- Data path is translated into a case statement.

## Translation (HTL → Verilog)

```
// 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
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.
- Data path is translated into a case statement.
- RAM loads and stores automatically turn off RAM.

## Translation (HTL $\rightarrow$ Verilog)

```
    // 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
```

- Finally, translate the FSMD into Verilog.
- This includes a RAM interface.
- Data path is translated into a case statement.
- RAM loads and stores automatically turn off RAM.
- Control logic is translated into another case statement with a reset.

## Outline

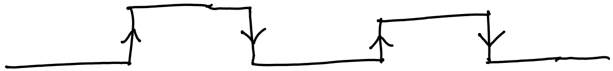
Example

Verification

Results

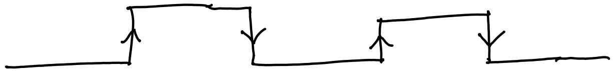
## Verilog Semantics (Adapted from Lööw et al. (2019))

- Top-level semantics are **small-step operational semantics**.

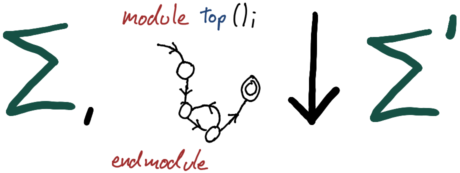


## Verilog Semantics (Adapted from Lööw et al. (2019))

- Top-level semantics are **small-step operational semantics**.



- At each clock tick, the **whole module** is executed using **big-step semantics**.



## Main Challenges in Proof

### Translation of memory model

**Abstract/infinite memory model** translated into **concrete/finite RAM**.

## Main Challenges in Proof

### Translation of memory model

**Abstract/infinite memory model** translated into **concrete/finite RAM**.

### Integration of Verilog Semantics

- **Verilog semantics** differs from CompCert's main assumptions of intermediate language semantics.
- Abstract values like the **program counter** now correspond to **values in registers**.

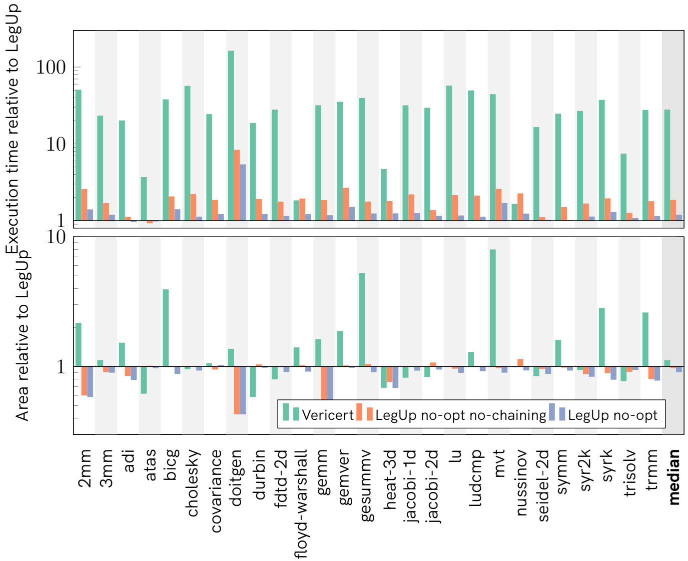
## Outline

Example

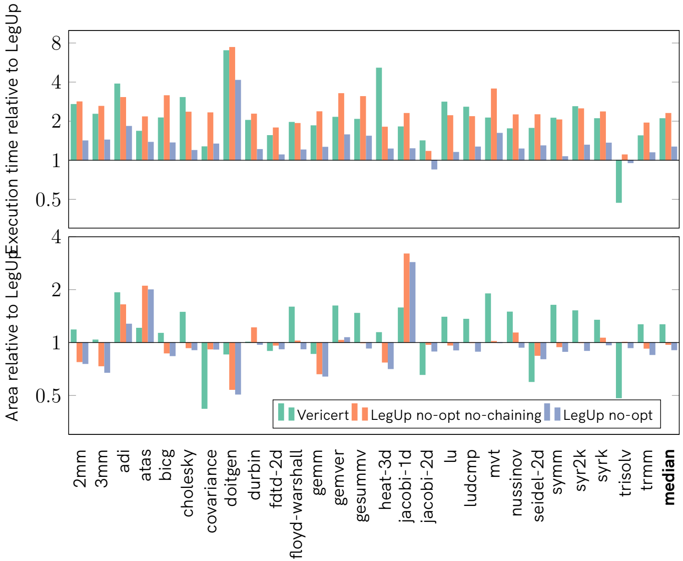
Verification

Results

## The bad news: with division approximately 27× slower



# The better news: without division about 2× slower



## Fuzzing Vericert with Csmith

Fuzzed Vericert with Csmith to check correctness theorem.

| Tool | Run-time errors |
|---|---|
| Vivado HLS | 1.23% |
| Intel i++ | 0.4% |
| Bambu 0.9.7-dev | 0.3% (13.7%) |
| LegUp 4.0 | 0.1% |

## Fuzzing Vericert with Csmith

Fuzzed Vericert with Csmith to check correctness theorem.

| Tool | Run-time errors |
|---|---|
| Vivado HLS | 1.23% |
| Intel i++ | 0.4% |
| Bambu 0.9.7-dev | 0.3% (13.7%) |
| LegUp 4.0 | 0.1% |
| **Vericert** | **0% (0.03%)** |

## Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- HLS tool **proven correct in Coq** by proving translation of CFG into FSMD.

## Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- HLS tool **proven correct in Coq** by proving translation of CFG into FSMD.
- Small optimisations implemented such as **RAM Inference**.

## Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- HLS tool **proven correct in Coq** by proving translation of CFG into FSMD.
- Small optimisations implemented such as **RAM Inference**.
- Performance without divisions comparable to LegUp without optimisations.

## Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- HLS tool **proven correct in Coq** by proving translation of CFG into FSMD.
- Small optimisations implemented such as **RAM Inference**.
- Performance without divisions comparable to LegUp without optimisations.

### Future Work

Make Vericert not only **correct**, but **competitive**.

- Implement **scheduling** and **resource sharing**.

## Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- HLS tool **proven correct in Coq** by proving translation of CFG into FSMD.
- Small optimisations implemented such as **RAM Inference**.
- Performance without divisions comparable to LegUp without optimisations.

### Future Work

Make Vericert not only **correct**, but **competitive**.

- Implement **scheduling** and **resource sharing**.
- Add **external module** support.

## Conclusion

Written a formally verified high-level synthesis tool in **Coq** based on **CompCert**.

- HLS tool **proven correct in Coq** by proving translation of CFG into FSMD.
- Small optimisations implemented such as **RAM Inference**.
- Performance without divisions comparable to LegUp without optimisations.

### Future Work

Make Vericert not only **correct**, but **competitive**.

- Implement **scheduling** and **resource sharing**.
- Add **external module** support.
- Add **global variable** support.

# Thank you

Documentation

`https://vericert.ymhg.org`

GitHub

`https://github.com/ymherklotz/vericert`

OOPSLA'21 Preprint

`https://ymhg.org/papers/fvhls_oopsla21.pdf`

## References

Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. An empirical study of the reliability of high-level synthesis tools. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 219–223, 2021. doi: 10.1109/FCCM51124.2021.00034.