Formally Verified High-level Synthesis

Using CompCert to translate C to Verilog

PLDI 2020: Student Research Competition — Yann Herklotz

High-level Synthesis

- Transform software (C) into hardware (Verilog).
- Behavioural description into hardware description.

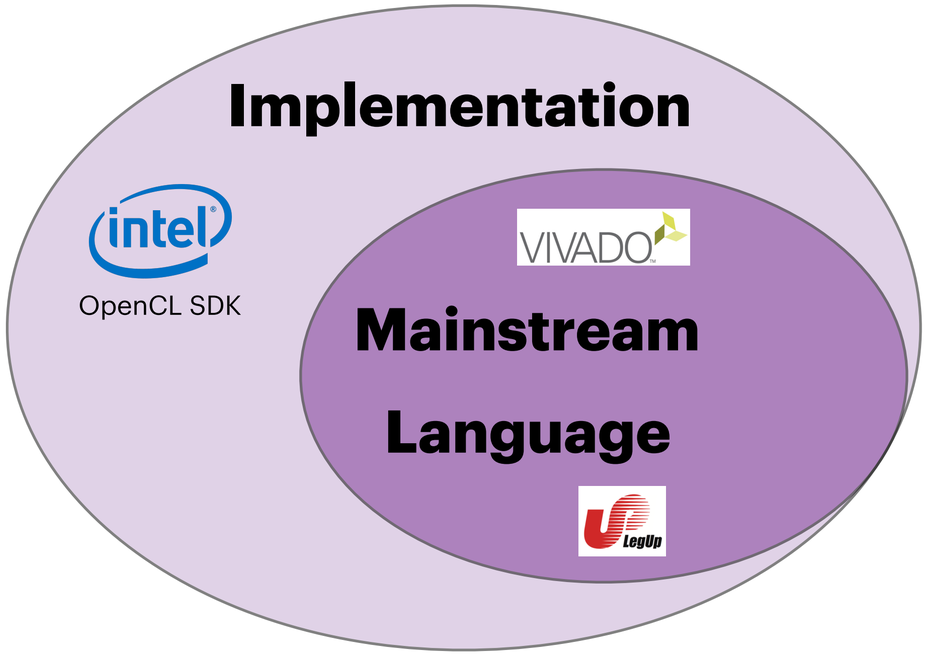
High-level Synthesis

- Transform software (C) into hardware (Verilog).
- Behavioural description into hardware description.

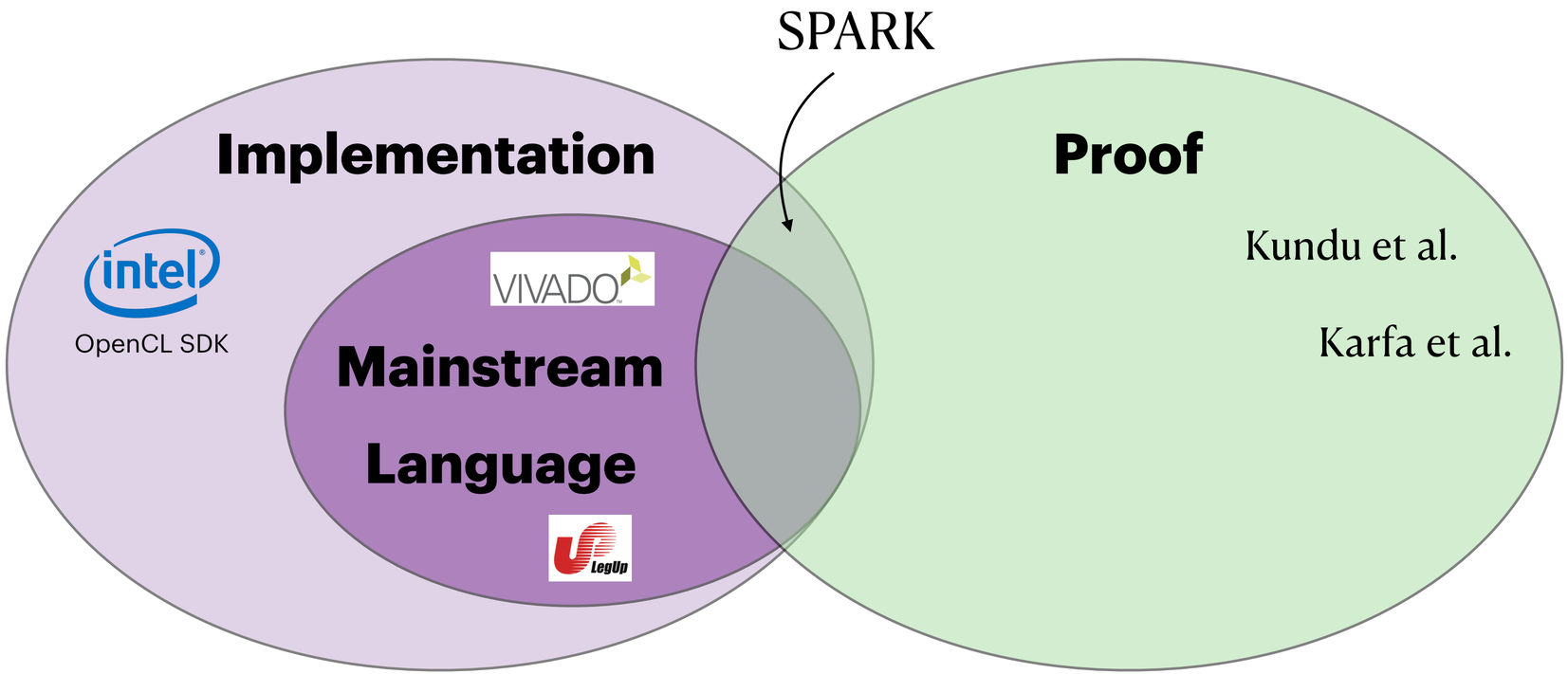
- Requires automatic parallelisation of code.
- Often unpredictable.
- Quite fragile with what features are supported in C.

Formally Verified High-level Synthesis

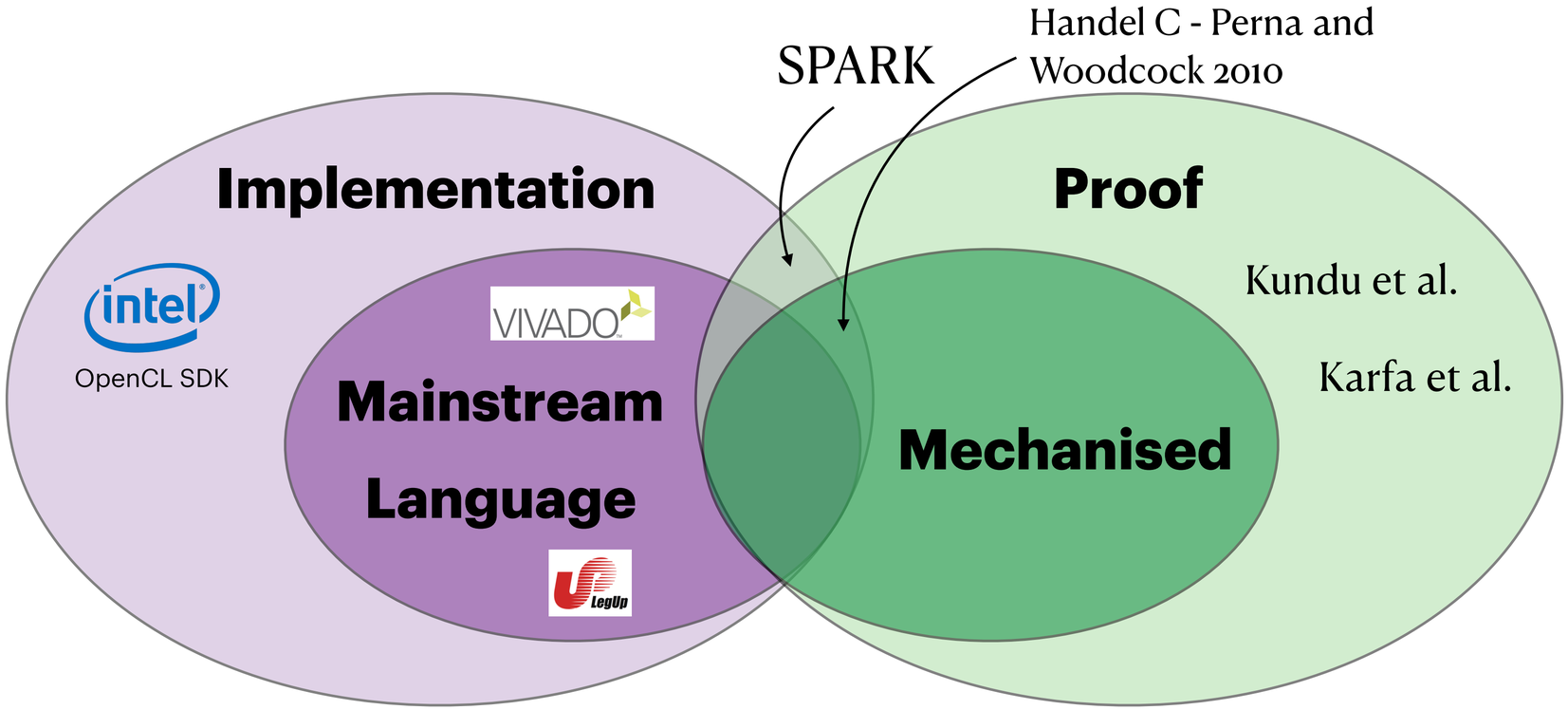
Formally Verified High-level Synthesis



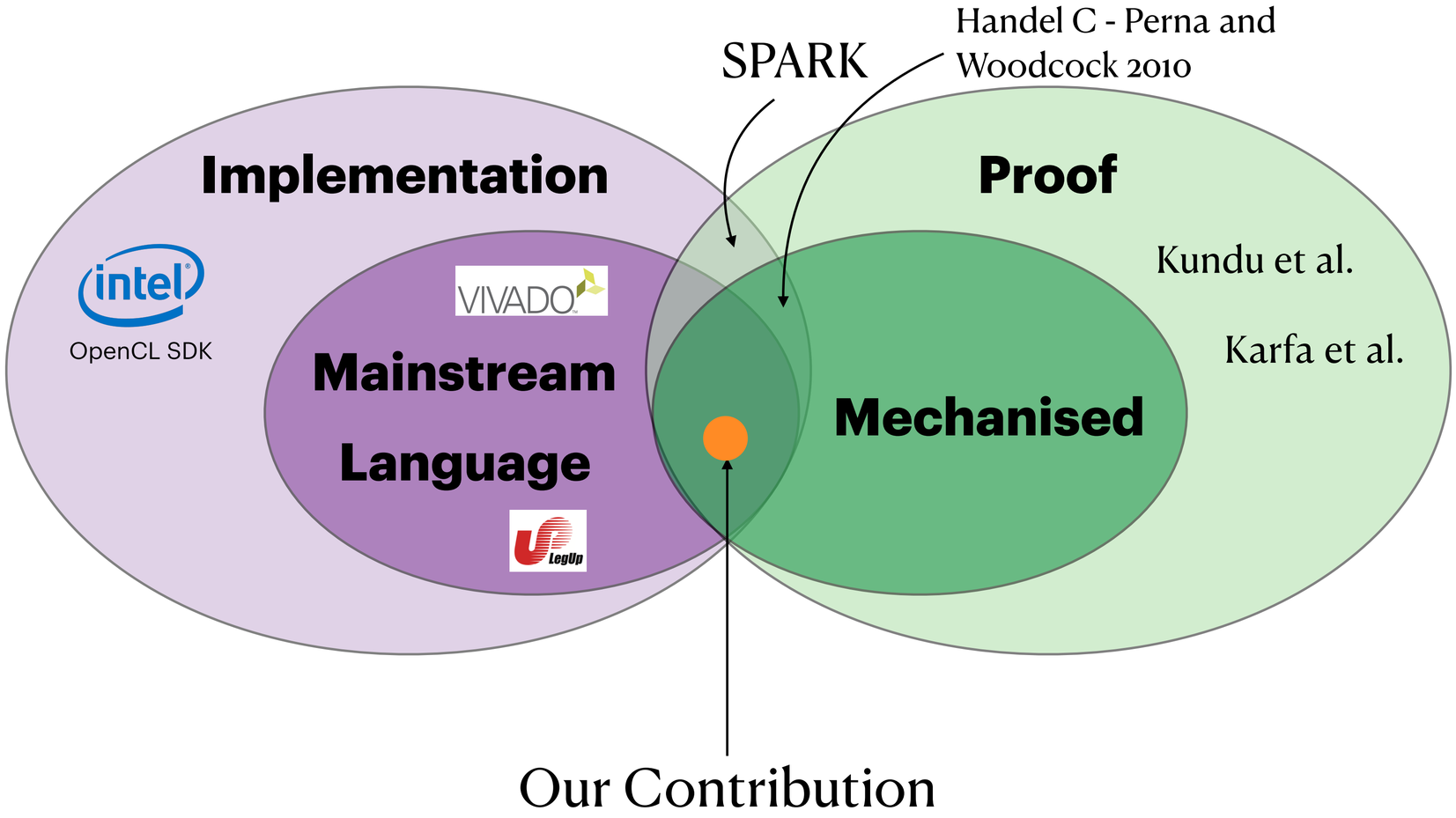
Formally Verified High-level Synthesis



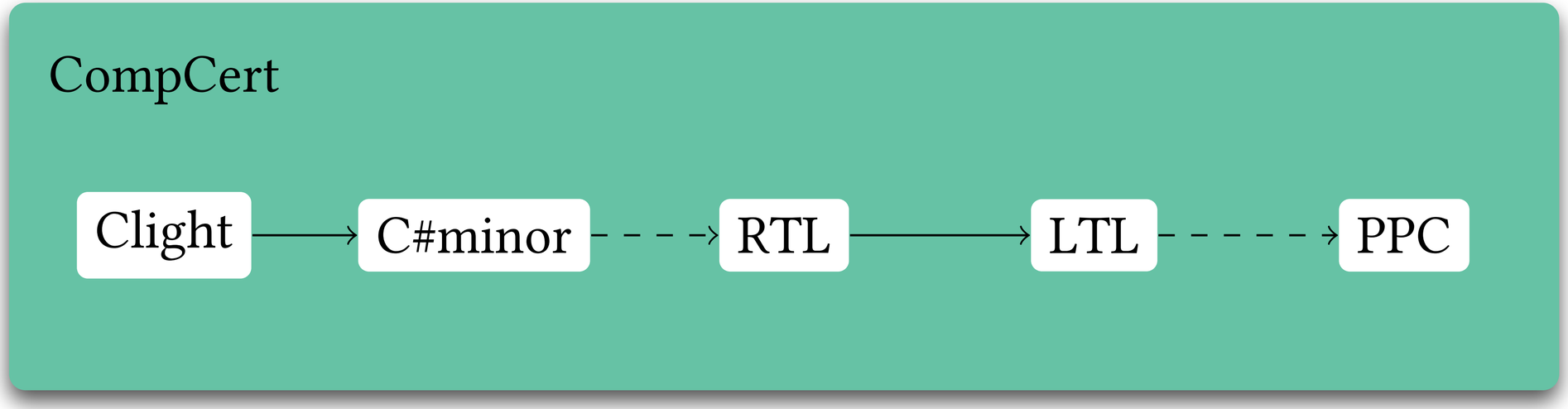
Formally Verified High-level Synthesis



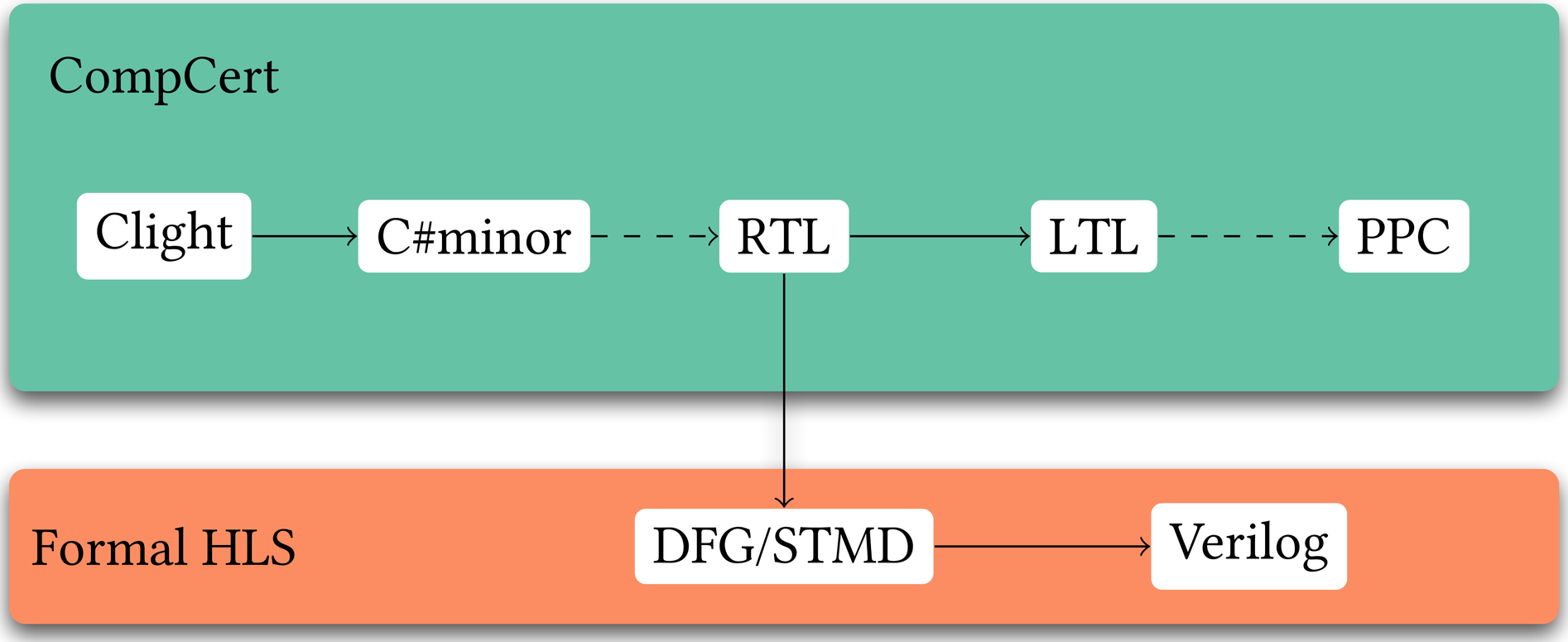
Formally Verified High-level Synthesis



Extending CompCert to Support Verilog

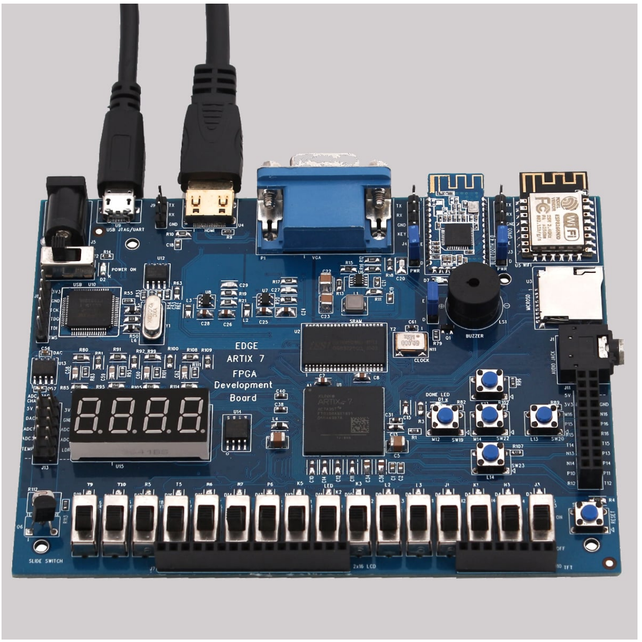


Extending CompCert to Support Verilog



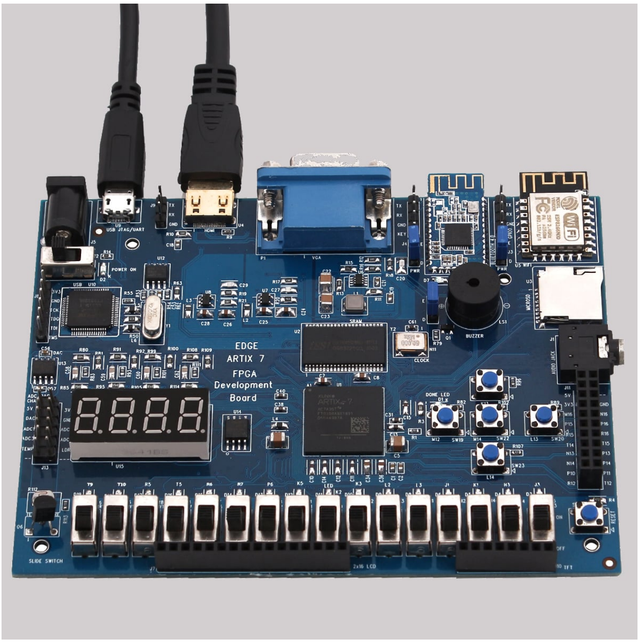
Verilog

- Verilog is a hardware description language.



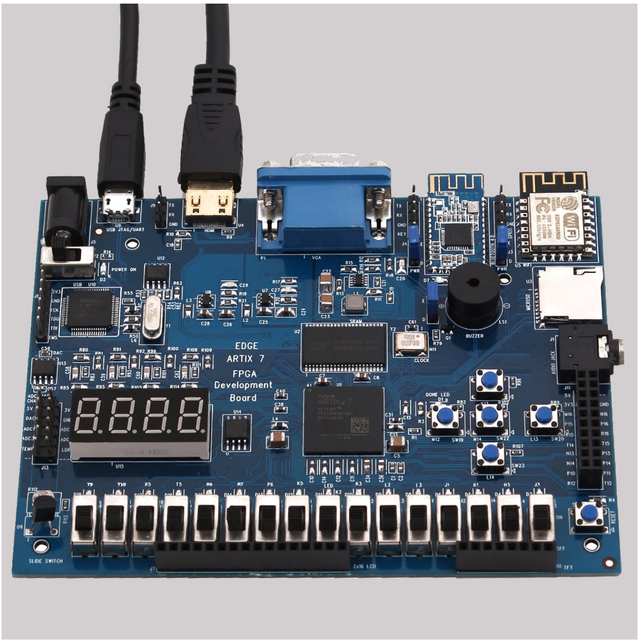
Verilog

- Verilog is a hardware description language.
- Verilog designs can then be placed onto an FPGA.



Verilog

- Verilog is a hardware description language.
- Verilog designs can then be placed onto an FPGA.



- We used existing operational semantics for Verilog (Lööw et al. 2019) and mechanised them in Coq.
- Had to modify them to support declarations properly.

Why Branch off at RTL?

Why Branch off at RTL?

- Many optimisations performed at various stages in the CompCert pipeline.

Why Branch off at RTL?

- Many optimisations performed at various stages in the CompCert pipeline.
- RTL is the first backend language and independent of any details of the CPU.

Why Branch off at RTL?

- Many optimisations performed at various stages in the CompCert pipeline.
- RTL is the first backend language and independent of any details of the CPU.
- Lower level languages become more specific to the CPU (register allocation, pipelining...)

Why Branch off at RTL?

- Many optimisations performed at various stages in the CompCert pipeline.
- RTL is the first backend language and independent of any details of the CPU.
- Lower level languages become more specific to the CPU (register allocation, pipelining...)

- We need a new intermediate language to support hardware optimisations (scheduling, parallelisation of loops...).

Why Branch off at RTL?

- Many optimisations performed at various stages in the CompCert pipeline.
- RTL is the first backend language and independent of any details of the CPU.
- Lower level languages become more specific to the CPU (register allocation, pipelining...)

- We need a new intermediate language to support hardware optimisations (scheduling, parallelisation of loops...).
- Choose to use a **state machine with datapath** representation.

Translation Algorithm

```c
int main() {
    int max = 5;
    int acc = 0;

    for (int i = 0; i < max; i++) {
        acc += i;
    }

    return acc + 2;
}
```

Translation Algorithm

```
int main() {
    int max = 5;
    int acc = 0;

    for (int i = 0; i < max; i++) {
        acc += i;
    }

    return acc + 2;
}
```

- Translation of simple accumulator into hardware.

Translation Algorithm

```
main() {
  11:  x3 = 5
  10:  x2 = 0
   9:  x1 = 0
   8:  nop
   7:  if (x1 <s x3) goto 6 else goto 3
   6:  x2 = x2 + x1 + 0 (int)
   5:  x1 = x1 + 1 (int)
   4:  goto 7
   3:  x4 = x2 + 2 (int) goto 1
   2:  x4 = 0
   1:  return x4
}
```

- Translation of simple accumulator into hardware.
- Generate RTL from C with CompCert.

Translation Algorithm

```
main() {
   control {
      11: goto 10
      10: goto 9
      9: goto 8
      8: goto 7
      7: goto if x1 <s x3 goto 6 else goto 3
      6: goto 5
      5: goto 4
      4: goto 7
      3: goto 1
      2: goto 1
      1: goto 1
   }
   data {
      11: x3 = 5
      10: x2 = 0
      9: x1 = 0
      8: nop
      7: nop
      6: x2 = x2 + x1 + 0 (int)
      5: x1 = x1 + 1 (int)
      4:
      3: x4 = x2 + 2 (int)
      2: x4 = 0
      1: ret = x4
   }
}
```

- Translation of simple accumulator into hardware.
- Generate RTL from C with CompCert.
- Split the up each instruction into a state transition and a data operation.

Translation Algorithm

```
module main(reg_7, reg_8, clk, finish, ret);
  input [0:0] reg_7;
  input [0:0] reg_8;
  input [0:0] clk;
  output reg [0:0] finish;
  output reg [31:0] ret;
  always @(posedge clk)
    if ((reg_8 == 1'd1)) state <= 4'd11;
    else
      case (state)
        4'd8: state <= 3'd7;
        3'd4: state <= 3'd7;
        2'd2: state <= 1'd1;
        4'd10: state <= 4'd9;
        3'd6: state <= 3'd5;
        1'd1: state <= 1'd1;
        4'd9: state <= 4'd8;
        3'd5: state <= 3'd4;
        2'd3: state <= 1'd1;
        4'd11: state <= 4'd10;
        3'd7: state <= ({$signed(reg_1) < $signed(reg_3)} ?
                        3'd6 : 2'd3);
        default:;
      endcase
  always @(posedge clk)
    case (state)
      4'd8: ;
      3'd4: ;
      2'd2: reg_4 <= 32'd0;
      4'd10: reg_2 <= 32'd0;
      3'd6: reg_2 <= {reg_2 + {reg_1 + 32'd0}};
      1'd1: begin
         finish <= 1'd1;
         ret <= reg_4;
      end
      4'd9: reg_1 <= 32'd0;
      3'd5: reg_1 <= {reg_1 + 32'd1};
      2'd3: reg_4 <= {reg_2 + 32'd2};
      4'd11: reg_3 <= 32'd5;
      3'd7: ;
      default:;
    endcase
  reg [31:0] reg_4;
  reg [31:0] reg_2;
  reg [31:0] state;
  reg [31:0] reg_1;
  reg [31:0] reg_3;
endmodule
```

- Translation of simple accumulator into hardware.
- Generate RTL from C with CompCert.
- Split the up each instruction into a state transition and a data operation.
- This state machine can be translated directly to a syntactical representation of Verilog.

Translation Algorithm

```
input [0:0] clk;
output reg [0:0] finish;
output reg [31:0] ret;
always @(posedge clk)
  if ((reg_8 == 1'd1)) state <= 4'd11;
  else
    case (state)
      4'd8: state <= 3'd7;
      3'd4: state <= 3'd7;
      2'd2: state <= 1'd1;
      4'd10: state <= 4'd9;
      3'd6: state <= 3'd5;
      1'd1: state <= 1'd1;
      4'd9: state <= 4'd8;
      3'd5: state <= 3'd4;
      2'd3: state <= 1'd1;
      4'd11: state <= 4'd10;
      3'd7: state <= ({$signed(reg_1) < $signed(reg_3)} ?
                        3'd6 : 2'd3);
      default:;
    endcase
always @(posedge clk)
  case (state)
```

- Translation of simple accumulator into hardware.
- Generate RTL from C with CompCert.
- Split the up each instruction into a state transition and a data operation.
- This state machine can be translated directly to a syntactical representation of Verilog.

Translation Algorithm

```
      default:;
    endcase
always @(posedge clk)
  case (state)
    4'd8: ;
    3'd4: ;
    2'd2: reg_4 <= 32'd0;
    4'd10: reg_2 <= 32'd0;
    3'd6: reg_2 <= {reg_2 + {reg_1 + 32'd0}};
    1'd1: begin
       finish <= 1'd1;
       ret <= reg_4;
    end
    4'd9: reg_1 <= 32'd0;
    3'd5: reg_1 <= {reg_1 + 32'd1};
    2'd3: reg_4 <= {reg_2 + 32'd2};
    4'd11: reg_3 <= 32'd5;
    3'd7: ;
    default:;
  endcase
reg [31:0] reg_4;
reg [31:0] reg_2;
reg [31:0] state;
```

- Translation of simple accumulator into hardware.
- Generate RTL from C with CompCert.
- Split the up each instruction into a state transition and a data operation.
- This state machine can be translated directly to a syntactical representation of Verilog.

Proving Equivalence of Translation

Proving Equivalence of Translation

- Relational specification of translation to a **state-machine with datapath (STMD)**.

Proving Equivalence of Translation

- Relational specification of translation to a **state-machine with datapath (STMD)**.
- Assuming that this relation holds, we prove a **forward simulation** based on the semantics of RTL and our STMD.

Proving Equivalence of Translation

- Relational specification of translation to a **state-machine with datapath (STMD)**.
- Assuming that this relation holds, we prove a **forward simulation** based on the semantics of RTL and our STMD.
- We then do the same between the STMD representation and Verilog.

Limitations and Future Work

Limitations and Future Work

- Most proofs have been implemented and we can properly use it now.

Limitations and Future Work

- Most proofs have been implemented and we can properly use it now.

Limitations and Future Work

- Most proofs have been implemented and we can properly use it now.

- No support for global memory and floating point.

Limitations and Future Work

- Most proofs have been implemented and we can properly use it now.

- No support for global memory and floating point.
- No hardware specific optimisations performed.

Limitations and Future Work

- Most proofs have been implemented and we can properly use it now.

- No support for global memory and floating point.
- No hardware specific optimisations performed.

Limitations and Future Work

- Most proofs have been implemented and we can properly use it now.

- No support for global memory and floating point.
- No hardware specific optimisations performed.

- Working on finishing all the proofs, and then implementing scheduling.

Limitations and Future Work

- Most proofs have been implemented and we can properly use it now.

- No support for global memory and floating point.
- No hardware specific optimisations performed.

- Working on finishing all the proofs, and then implementing scheduling.
- Design a better intermediate language to handle operations in the same clock cycle.

Thank you