VHDL generation from Python Synchronous Message Exchange Networks

Truls Asheim <truls@asheim.dk>
August 23, 2016

University of Copenhagen, Niels Bohr Institute
1. Introduction and motivation
2. Synchronous Message Exchange recap
3. Translating Python to VHDL
4. An example and test benches
5. Implementation
6. Summary and future work
Introduction and Motivation
Specialized hardware (FPGAs, ASICs) is more complicated to develop than software.

Reduced power consumption, and parallel processing

Hardware development has a high barrier of entry and common Hardware Description Languages (HDLs) are hard to work with.

Particularly for test code
What we have

A transpiler (source-to-source compiler) capable of translating Python SME networks implemented using the PySME library to functionally equivalent VHDL code.

**Automatic test bench generation!** — Makes it easy to perpetually verify the correctness of the generated VHDL code.

Generated code can be simulated using VHDL simulators and/or synthesizers such as GHDL and Xilinx Vivado.

Proof of concept, but shows the potential of the SME model.
SME is a globally synchronous message passing model, with an equivalence in CSP, mimicking signal propagation in hardware. Single broadcasting channel type, called a bus by hardware analogy.

Conceived after an attempt to generate Vivado C and VHDL from PyCSP models showed that enforcing globally synchronous message propagation in pure CSP caused an explosion of complexity.

First presented at CPA 2014, with revised version at CPA 2015
It’s not High Level Synthesis (HLS)

HLS relies on auto-paralellizing sequential code.

- Efficiency of generated code can be an issue.
- Generated code difficult to understand
- Opaque translation process.
- Level of abstraction decreasing

Converting SME to VHDL is different

- SME makes it easy to program using hardware-like synchronous data propagation
- SME models already parallel
- Structural mapping to VHDL is trivial
- Level of abstraction mostly the same
- Close correlation between input and output
Why Python?

Using a general purpose programming language for hardware design means that:

- Increased accessibility for software developers.
- Nicer to work with than VHDL
- Easier testing/simulation:
  - Full ecosystem available
  - Existing code can be reused
  - Common and established libraries still available

Python is particularly well suited for rapid prototyping due to its high productivity nature.
Why not Python?

- Highly dynamic language, while hardware is inherently static.
- Only a subset of Python can be translated to VHDL.
- Type information required in VHDL — not provided by Python.

These are the main challenges of the translation.
Translating Python to VHDL
Obviously, the complete Python language cannot simply be translated to VHDL. Only a restricted subset:

- Only conditionals and variables assignments (but almost full expressions)
- No loops (yet)
- No lists (yet). This is fairly limiting.

And some additional restrictions:

- SME process variables must be declared class-globally
## Process Types

Two types of processes. Functions and Externals

<table>
<thead>
<tr>
<th>Externals</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only used for simulation</td>
<td>Implements hardware targeted processes</td>
</tr>
<tr>
<td>Any Python code</td>
<td>Restricted (static) subset of Python</td>
</tr>
<tr>
<td>Only structure is translated</td>
<td>Translated completely</td>
</tr>
</tbody>
</table>

```python
class Process(External):
    def setup(self):
        pass
    def clock(self):
        pass

class Process(Function):
    def setup(self):
        pass
    def clock(self):
        pass
```

Functions and Externals are identical when simulating PySME — only different when translating to VHDL.
<table>
<thead>
<tr>
<th>PySME</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>VHDL variable or constant</td>
</tr>
<tr>
<td>Function parameter</td>
<td>Generic</td>
</tr>
<tr>
<td>Bus definition</td>
<td>VHDL ports and signals</td>
</tr>
<tr>
<td>External process</td>
<td>File containing skeleton translation</td>
</tr>
<tr>
<td>Function process</td>
<td>File containing complete translation</td>
</tr>
<tr>
<td>Network definition</td>
<td>File containing top-level entity</td>
</tr>
</tbody>
</table>
Python is dynamically typed, while VHDL statically typed and require explicit type information.

Thus, we need to add type information

For variables, solved through a combination of “typing on first assign” (e.g. `self.n = 4` — `n` is a 32-bit signed integer) and optional annotations.

Annotations currently mandatory for bus channels.

Not a lot of types. Only signed and unsigned integers and booleans
But it not just types!

Number widths crucial for efficiency of implemented hardware since each bit of a number corresponds to a “wire” in the hardware implementation.

So we need to decide, not just types, but integer widths as well.

Annotations of signedness and bitwidths:

**Variables**
```
self.n = 0 # type: t.u12  The variable n is a 12-bit unsigned integer
```

**Bus channels**
```
Bus("ValueBus", [t.i24("val")])  The channel val of the bus Value-Bus is a 24-bit signed integer
```

**Better solution:** Augment annotations with optimal width inference! (this is future work)
No constants in Python, but correct variable constness is important in VHDL!

So we designate variables that are never assigned to as constants in the VHDL code.
The VHDL code we generate, should be easily recognizable and comprehensible by SME model implementer. Preservation of process, variable, bus channel names important in ensuring this.
A Small Example
The AddN network:
Three processes:

- Gen emits a parameter value
- AddN accumulates a value, added to value from Gen, a constant and a parameter value.
- Printer prints value from AddN
from sme import Network, Function,
      External, Bus, SME,
      Types

t = Types()

class Gen(Function):
    def setup(self, ins, outs, n):
        self.map_outs(outs, "out")
        self.n = n  # type: t.u3

    def run(self):
        self.out["val"] = self.n

class AddN(Function):
    def setup(self, ins, outs, n):
        self.map_ins(ins, "num")

    def run(self):
        self.map_outs(outs, "res")
        self.n = n
        self.c = 4  # type: t.u3
        self.accum = 0  # type: t.u10

        self.accum += self.n + self.c +
                       self.num["val"]

        self.res["val"] = self.accum

    def run(self):
        print(self.res["val"])

class Printer(External):
    def setup(self, ins, outs):
        self.map_ins(ins, "res")

    def run(self):
        print(self.res["val"])
```python
class AddNNet(Network):
    def wire(self):
        bus1 = Bus("ValueBus",
                    [t.u2("val")])
        bus1["val"] = 0
        self.tell(bus1)
        bus2 = Bus("InputBus",
                    [t.u10("val")])
        bus2["val"] = 0
        self.tell(bus2)
        gen_param = 2
        gen = Gen("Gen", [], [bus1],
                  gen_param)
        self.tell(gen)
        addn_param = 4
        addn = AddN("AddN", [bus1],
                    [bus2], addn_param)
        self.tell(addn)
        p = Printer("Printer", [bus2], [])
        self.tell(p)

    def main():
        sme = SME()
        sme.network = AddNNet("AddNet")
        sme.network.clock(100)
        if __name__ == "__main__":
            main()
```

AddN process

-- Library includes snipped

decl entity AddN is
generic (n: integer);
port (res_val: out u10_t;
    num_val: in u2_t;
    rst: in std_logic;
    clk: in std_logic
);
end AddN;

architecture RTL of AddN is
begin
  process (clk, rst)
  constant c: u3_t := std_logic_vector(to_unsigned(4, u3_t'length));
  variable accum: u10_t := std_logic_vector(to_unsigned(0, u10_t'length));
  begin
    if rst = '1' then
      res_val <= std_logic_vector(to_unsigned(0, u10_t'length));
      accum := std_logic_vector(to_unsigned(0, u10_t'length));
    elsif rising_edge(clk) then
      accum := std_logic_vector(unsigned(accum) + to_unsigned(n, u10_t'length) +
                               unsigned(c) + unsigned(num_val));
      res_val <= std_logic_vector(unsigned(accum));
    end if;
  end process;
end architecture;
Top-level entity

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library sme_types;
use work.sme_types.all;

entity AddNNet is
  port (AddNNet_ValueBus_val: inout u2_t;
        AddNNet_InputBus_val: inout u10_t;
        rst: in std_logic;
        clk: in std_logic
        );
end AddNNet;

architecture RTL of AddNNet is
  begin
    -- signals
    AddN: entity work.AddN
    generic map (n => 4)
    port map (num_val => AddNNet_ValueBus_val,
              res_val => AddNNet_InputBus_val,
              rst => rst,
              clk => clk);

    Gen: entity work.Gen
    generic map (n => 2)
    port map (out_val => AddNNet_ValueBus_val,
              rst => rst,
              clk => clk);

    Printer: entity work.Printer
    port map (res_val => AddNNet_InputBus_val,
              rst => rst,
              clk => clk);
end architecture;
Test Benches
A test bench is used for testing and verifying hardware descriptions.

- Test vectors generated by simulating a PySME model.
- Read by auto-generated VHDL test bench code.
- Values SME buses cycle-accurately mirrors the the values of the VHDL signals that they are transformed into.

Manual modifications of the generated VHDL code can be verified for correctness against the original Python implementation.
Trace CSV file

AddNNet_InputBus_val, AddNNet_ValueBus_val
0, 0
8, 2
18, 2
28, 2
38, 2
48, 2
58, 2
68, 2
78, 2
88, 2
98, 2
108, 2
[...]
while not endfile(F) loop
  readline(F, L);
  wait until rising_edge(clock);
  fieldno := 0;
  read_csv_field(L, tmp);
  if not are_strings_equal(tmp, "U") then
    assert are_strings_equal(uint_image(AddNNet_InputBus_val), tmp)
    report "Unexpected value of AddNNet_InputBus_val in cycle " &
    integer'image(clockcycle) & ", Actual value was: " & uint_image(AddNNet_InputBus_val)
    & " but expected " & truncate(tmp) severity Error;
  end if;
  fieldno := fieldno + 1;

  read_csv_field(L, tmp);
  if not are_strings_equal(tmp, "U") then
    assert are_strings_equal(uint_image(AddNNet_ValueBus_val), tmp)
    report "Unexpected value of AddNNet_ValueBus_val in cycle " &
    integer'image(clockcycle) & ", Actual value was: " & uint_image(AddNNet_ValueBus_val)
    & " but expected " & truncate(tmp) severity Error;
  end if;
  fieldno := fieldno + 1;

  clockcycle := clockcycle + 1;
end loop;
Workflow overview

- PySME file
- Translation to VHDL
- VHDL model
- VHDL test bench
- Simulation/
Verification
- Process trace
- Simulation
Running it

$ dist/build/almique/almique examples/addn.py
$ cd output
$ ls
AddN.vhdl AddNNet.vhdl AddNNet_tb.vhdl Gen.vhdl
Printer.vhdl csv_util.vhdl sme_types.vhdl
$ ghdl -a --ieee=synopsys --work=sme_types
  sme_types.vhdl Gen.vhdl AddN.vhdl Printer.vhdl
  csv_util.vhdl AddNNet.vhdl AddNNet_tb.vhdl
$ ghdl -e --ieee=synopsys --work=sme_types AddNNet_tb
$ python ../examples/addn.py -t trace.csv
$ ./addnnet_tb
AddNNet_tb.vhdl:91:5:@1us:(report note):
Completed after 100 clockcycles
$
Overall stats

63 lines of Python turns into 440 lines of VHDL (including test benches)

So a 270% increase in code size

or 377 lines of code you didn’t have to write.
Implementation
The Transpiler

- Written in Haskell
- 1883 SLOC (Including some inline VHDL)
- Python parsed using the `language-python` module
- VHDL generated using `Text.Pretty` pretty printing combinators
- Code transformation through “classic” compiler pipeline
Compilation pipeline

1. PySME source code file
2. Parsing
3. Python AST
4. Analysis
5. Intermediate Code Generation
6. Extended SMEIL
7. Complete SMEIL
8. Code generation
9. Output directory containing the generated VHDL files

SMEIL is the SME Intermediate Language
We have a translation system which

- Translates Python SME programs to VHDL
- Produces functionally equivalent VHDL code with similar structure
- Close correlation between input and output code makes transformations transparent
- Automatic test bench generation allows for “lifecycle” verification of VHDL
Future work

So where do we go from here?

• Expanding supported Python subset (lists, loops, functions)
• Avoid annotations in the “standard case”
  • Optimal bitwidth inference
  • Improved type inference
• Standard library
• Floating point
• More dynamic and “Pythonic” translations enabled by improved abstract interpretation model
Thank you!

Complete transpiler source code: https://github.com/truls/almique

PySME library source code: https://github.com/truls/pysme

Questions?