Polyphony

A Python-Based High-Level Synthesis Compiler

Hiroaki Kataoka
Ryos Suzuki
Sinby corporation (www.sinby.com)
Outline:

- Introduction of Polyphony
- Design Flow with Polyphony
- Compilation Process of Polyphony
- Abstraction in Python
- Inside of Polyphony
- Evaluation & Future Works
- Conclusion
Introduction of Polyphony:
What is HLS (High-Level Synthesis)?

General High-Level Synthesis

Software Programming Language [C/C++/System C] → HLS Compiler → Hardware Description Language [VHDL/Verilog]
Introduction of Polyphony: What is Polyphony?

Python for Hardware Design

Python → Polyphony → Verilog HDL (synthesizable)
Introduction of Polyphony: What is Polyphony?

- Bring higher level of abstraction to your design
- Allow designers to focus on developing the algorithm
- Reduce costs for program maintenance
- Open Source ([https://github.com/ktok07b6/polyphony](https://github.com/ktok07b6/polyphony))
Introduction of Polyphony:
Q: Can Polyphony accept any Python code?

A: No, Polyphony accepts only restricted Python code.
Outline:

- Introduction of Polyphony
- Design Flow with Polyphony
- Compilation Process of Polyphony
- Abstraction in Python
- Inside of Polyphony
- Evaluation & Future Works
- Conclusion
Design Flow with Polyphony:

1. Write your Design code & Testbench code (*.py)
2. Run & Verify on Python Interpreter
3. Compile with Polyphony
4. Build the Binary with Vendor tools
5. Run it on your target
6. Simulation & Verification
Outline:

- Introduction of Polyphony
- Design Flow with Polyphony
- Compilation Process of Polyphony
- Abstraction in Python
- Inside of Polyphony
- Evaluation & Future Works
- Conclusion
Compilation Process of Polyphony:

1. Generate Polyphony-IR (PpIR) from Python source code
2. Transform and Optimize the code on PpIR
3. Translate PpIR to AHDL (Abstract Hardware Description Language)
4. Translate AHDL to Verilog HDL
Compilation Process of Polyphony: Polyphony IR – PPIR

Python source code → Polyphony-IR (PpIR) → AHDL → Verilog HDL

**Primitive PpIR**

```python
def func(p, x, y, z):
    if p:
        r = x+y+z
    else:
        r = x*y*z
    return r
```

**Qudruples**

```python
def func(p, x, y, z):
    if p:
        tmp0 = x+y
        r = tmp0+z
    else:
        tmp1 = x*y
        r = tmp1*z
    return r
```

**SSA form**

```python
def func(p, x):
    if p:
        tmp0 = x+y
        r0 = tmp0+z
    else:
        tmp1 = x*y
        r1 = tmp1*z
        r2 = phi(r0, r1)
    return r2
```
Compilation Process of Polyphony: Scheduling with Data-Flow graph

Python source code → Polyphony-IR (PpIR) → AHDL → Verilog HDL

PpIR (SSA form)

def func(p, x):
    if p:
        tmp0 = x+y
        r0 = tmp0+z
    else:
        tmp1 = x*y
        r1 = tmp1*z
    r2 = phi(r0, r1)
    return r2

Data-flow

def func(p, x):
    if p:
        tmp0 = x+y
        r0 = tmp0+z
    else:
        tmp1 = x*y
        r1 = tmp1*z
    r2 = phi(r0, r1)
    return r2

6c64_p = @in_p
4f3e@t408 = (x+y)
b0ce_z = @in_z
4dc@t409 = (x*y)
b0cc_y = @in_y
b0cf@cond407 = (p != 0x0)
4dc6_ret1 = (@t408 + z)
4dc8_ret2 = (@t409 * z)
ad2a_ret3 = psi(@cond407? ret1@top.func_b1),
ad2b@function_return = ret#3
9254_return@function_return
Compilation Process of Polyphony: AHDL (Abstract Hardware Description Language)

Python source code → Polyphony-IR (PpIR) → AHDL → Verilog HDL

AHDL & State Transition Graph

```
func_b1_INIT:0
Sequence 0:
AHDL_CALLEE_PROLOG('func')
   (next state: func_b1_S0)

func_b1_S0:1
   p<1> <= func_in_p<1>
   x<32> <= func_in_x<32>
   y<32> <= func_in_y<32>
   z<32> <= func_in_z<32>
   (next state: func_b1_FINISH)

func_b1_FINISH:2
   cond407<1> <= (p<1> != 0)
   t408<32> <= (x<32> + y<32>)
   ret1<32> <= (t408<32> + z<32>)
   t409<32> <= (x<32> * y<32>)
   ret2<32> <= (t409<32> * z<32>)
   ret3<32> <= cond407<1> ?
   ret1<32> : (!cond407<1>) ?
   ret2<32> : 'bz
   func_out_0<32> <= ret3<32>
   (next state: func_b1_INIT)
```
Outline:

- Introduction of Polyphony
- Design Flow with Polyphony
- Compilation Process of Polyphony
- Abstraction in Python
- Inside of Polyphony
- Evaluation & Future Works
- Conclusion
Abstraction in Python:

- Class
- Tuple
- List
Abstraction in Python: Class

- Class is one of the basic abstraction features in Python
- Polyphoyny makes you can Object-Oriented hardware design

```python
class MyClass:
    def __init__(self, v):
        self.a = v
        self.b = v * v

    def func(self):
        return self.a + self.b

def main_func(v):
    c = MyClass(v)
    x = c.func()
    ....
```
Abstraction in Python: Class

- All instance methods are inlined
- All instance variables are converted to local variables

```python
class MyClass:
    def __init__(self, v):
        self.a = v
        self.b = v * v

    def func(self):
        return self.a + self.b

def main_func(v):
    c = MyClass(v)
    x = c.func()
    ....
```

Inlining
MyClass() & c.func()

```python
class MyClass:
    def __init__(self, v):
        self.a = v
        self.b = v * v

    def func(self):
        return self.a + self.b

def main_func(v):
    c_a = v
    c_b = v * v
    x = c_a + c_b
    ....
```
Abstraction in Python: Tuple

- Tuple is a feature that can handle multiple values
- With tuple, you can keep the source simple

```python
def func(x, y):
    t = (x, y)
    ...
    (y, x) = t
    ...
```

```python
def func(x, y):
    t[0] = x
    t[1] = y
    ...
    y = t[0]
    x = t[1]
    ...
```
Abstraction in Python: List

- List is an abstraction of Memory
- By default, List is mapped to internal RAM on FPGA

```python
def list_mul(lst):
    old_i = 0
    v = 0
    for i in range(len(lst)):
        if (i & 1) == 1:
            v += lst[old_i] * lst[i]
            old_i = i
    return v
```
Outline:

- Introduction of Polyphony
- Design Flow with Polyphony
- Compilation Process of Polyphony
- Abstraction in Python
- Inside of Polyphony
- Evaluation & Future Works
- Conclusion
Inside of Polyphony:

- UPHI
- MCJUMP
Inside of Polyphony:

UPHI

- U ⇒ ‘Use’
- PHI ⇒ Φ(Phi)-function of SSA form
- UPHI was introduced for compiling objects
Inside of Polyphony: UPHI

- $U \Rightarrow \text{‘Use’}$
- $\text{PHI} \Rightarrow \Phi(\text{Phi})$-function of SSA form
- $\text{UPHI}$ was introduced for compiling objects

Original

```python
if p:
    c = MyClass(10)
else:
    c = MyClass(20)
...
return c.x
```
Inside of Polyphony: UPHI

- How to compile \( c_2.x \)?

Original

```python
if p:
    c = MyClass(10)
else:
    c = MyClass(20)
...
return c.x
```

SSA

```python
if p:
    c0 = MyClass(10)
else:
    c1 = MyClass(20)
c2 = phi(c0, c1)
...
return c2.x
```

this must be selected either \( c_0.x \) or \( c_1.x \) by condition \( p \).
Inside of Polyphony: UPHI

- UPHI is inserted when conjunction object variable is used
- UPHI has a condition variable
- Then, UPHI is mapped to a multiplexer.

Original

```python
if p:
    c = MyClass(10)
else:
    c = MyClass(20)
...
return c.x
```

SSA

```python
if p:
    c0 = MyClass(10)
else:
    c1 = MyClass(20)
c2 = phi(c0, c1)
...
return c2.x
```

UPHI Inserted

```python
if p:
    c0 = MyClass(10)
else:
    c1 = MyClass(20)
c2 = phi(c0, c1)
...
c2.x = uphi(p ? c0.x, !p ? c1.x)
return c2.x
```
Inside of Polyphony:

**MCJUMP**

- MCJUMP is useful for generating selectors

```python
if x == 0:
    y *= 2
elif y == 0:
    z *= 3
elif z == 0:
    x *= 3
else:
    z = y
return y + z
```

**Source**

**Original Control Flow**

**Using MCJUMP**
Outline:

- Introduction of Polyphony
- Design Flow with Polyphony
- Compilation Process of Polyphony
- Abstraction in Python
- Inside of Polyphony
- Evaluation & Future Works
- Conclusion
Evaluation: Fibonacci Number

- Polyphony is three times slower than Vivado
- Polyphony's optimization is still not enough

```python
def fib(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    r0 = 0
    r1 = 1
    for i in range(n):
        r0, r1 = r1, r0 + r1
    return r1
```

Source

Results of Performance Measurements

<table>
<thead>
<tr>
<th></th>
<th>Polyphony (clks)</th>
<th>Vivado HLS (clks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32bit</td>
<td>33bit</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>46</td>
<td>143</td>
<td>143</td>
</tr>
<tr>
<td>47</td>
<td>-</td>
<td>146</td>
</tr>
<tr>
<td>92</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>93</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Future Works:

Features of Version 0.3.x (already done)

- Parallel programing features (Module, Woker, Port)
- Type annotation
- Abstract I/O port and timing controls
Future Works:

- Function Object / Lambda Expression
- Loop Optimization (pipeline, unroll)
- Importing Existing HDL
- Applications
  (RISC-V is under development)
Outline:

- Introduction of Polyphony
- Design Flow with Polyphony
- Compilation Process of Polyphony
- Abstraction in Python
- Inside of Polyphony
- Evaluation & Future Works
- Conclusion
Conclusion:

- Polyphony is Python-based HLS compiler
- It translates Python to Verilog HDL
- It brings higher level of abstraction to your design
- Python specific features will help with hardware design (class, tuple, list ...)
- We will continue to improve Polyphony in the future
Thank you:

- **Source Code:**
  
  https://github.com/ktok07b6/polyphony

  pull-requests, bug-reports and proposals are also welcome :-) 

- **We are looking for R&D partners!**
  
  polyphony@sinby.com
Any Question?

... Polyphony
A Python-Based High-Level Synthesis Compiler