July 11, 2016
MPSoC2016@Nara
Mini Keynote

Customizable Hardware Abstraction

Shinya Takamaeda-Yamazaki

Nara Institute of Science and Technology (NAIST)

How do you design a custom hardware?

- by HDL (Hardware Description Language)
  - Such as Verilog HDL and VHDL
  - Fully customizable and high performance☺
  - Huge development efforts due to the few abstractions☹
- by HLS (High Level Synthesis)
  - Such as C/C++ (Vivado HLS), Java (Max), OpenCL (Altera), …
  - High producitivity by untimed design manner☺
  - Hard to customize how the compiler generates codes☹
    - ✓ Sometimes cannot reach to the maximum performance

Motivation: How to keep both productivity and customizability

- Tradeoffs between HDL and HLS
  - High productivity by high-level abstraction
  - High hardware quality by low-level customization

- How to keep both:
  Allow users to build up a custom abstraction
  - Seamless DSL from RTL to HLS:
    Custom abstraction/method by using low-level abstractions

Customizable Hardware Abstraction

Veriloggen:
Explicit hardware modeling by Python

Describing hardware construction rule by utilizing **Python** power

Verilog HDL code is generated

```
from veriloggen import *
m = Module('blinkled')
clk = m.Input('CLK')
led = m.Output('LED', 8)
count = m.Reg('count', 32)
m.Assign( led(count[31:24]) )
m.Always(Posedge(clk)(
    count( count + 1 ) )
hdl = m.to_verilog()
print(hdl)
```

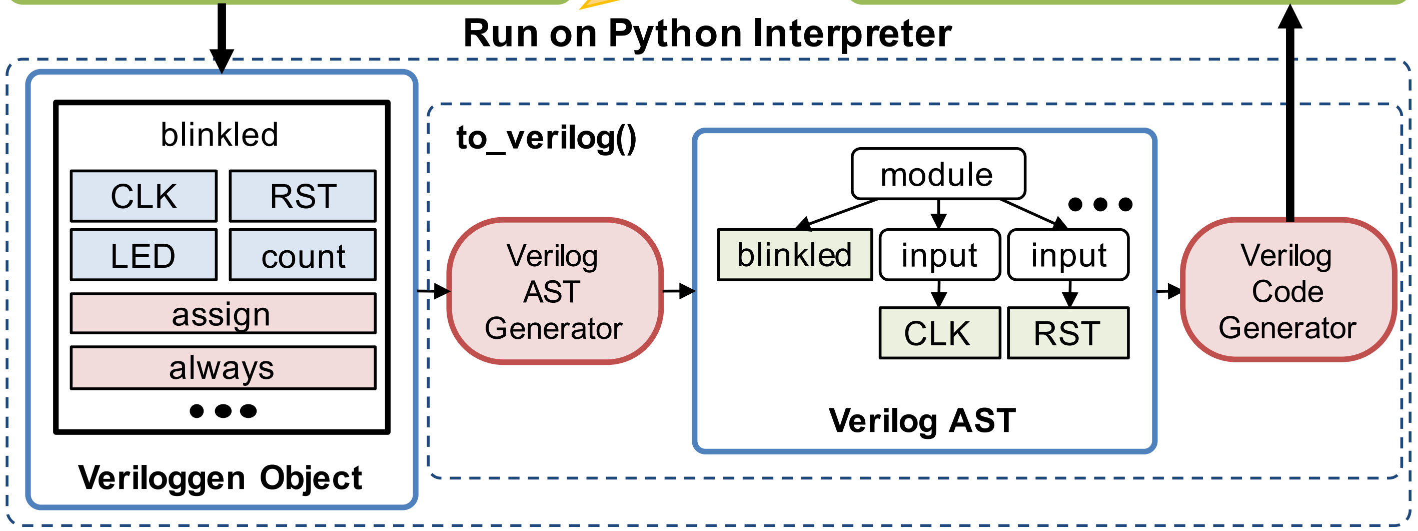
Design Generator by Python

Code generation by run

```
module blinkled (
  input CLK,
  output [7:0] LED
);
  reg [31:0] count;
  assign LED = count[31:24];
  always @(posedge CLK) begin
    count <= count + 1;
  end
endmodule
```

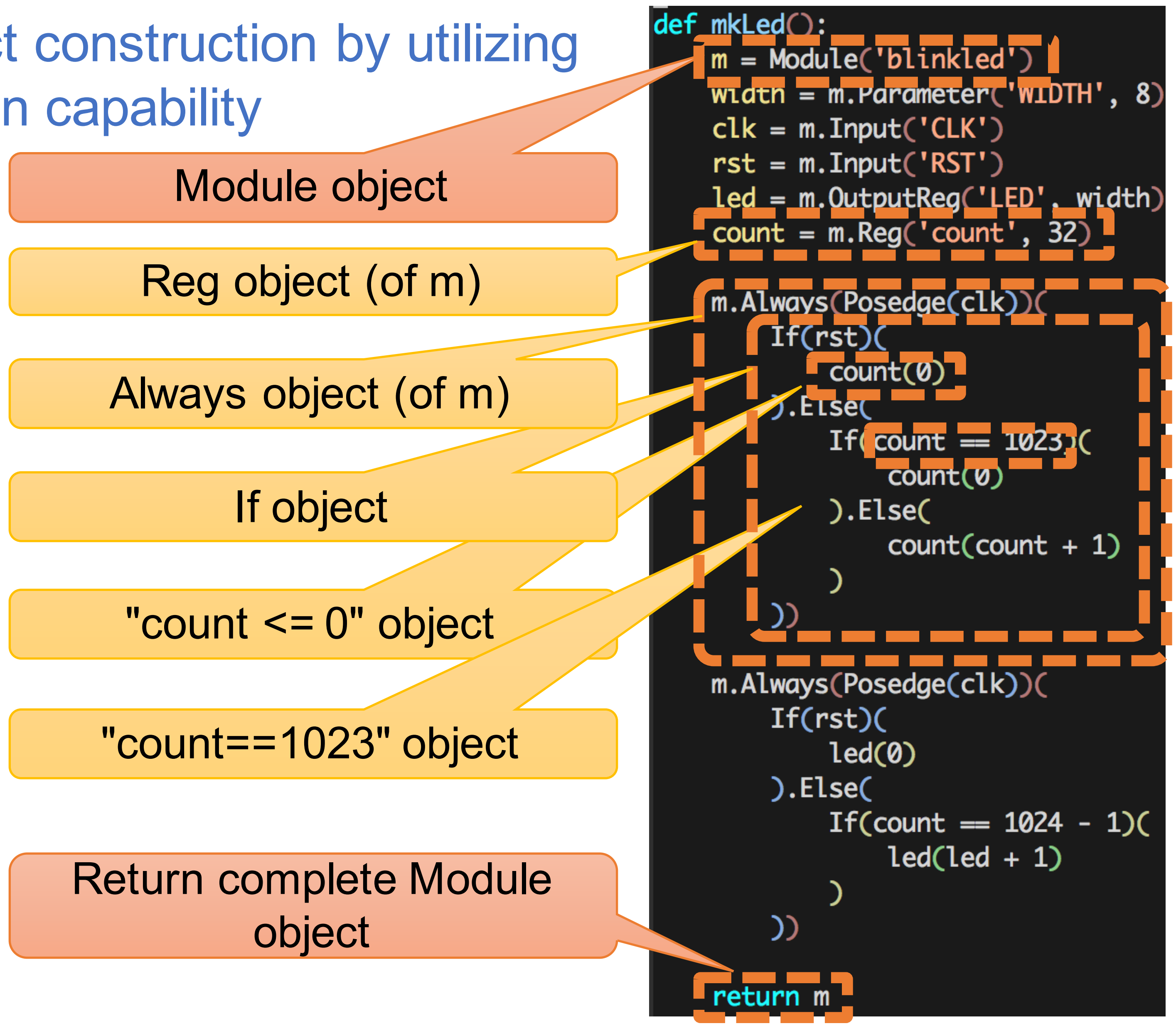
Verilog Source Code

Run on Python Interpreter



First level abstraction of HDL component

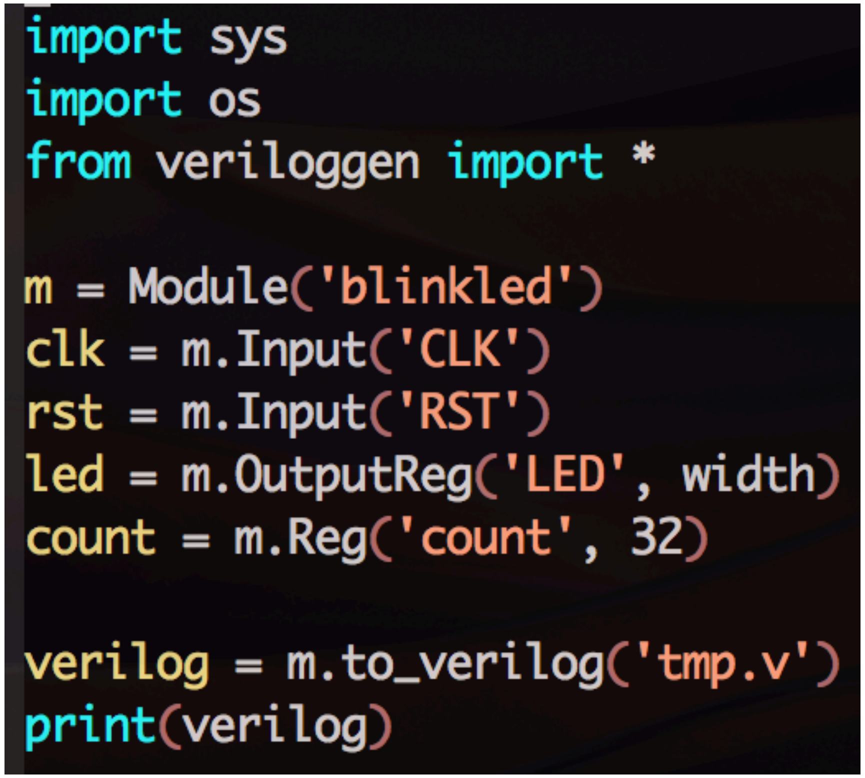
Object construction by utilizing Python capability



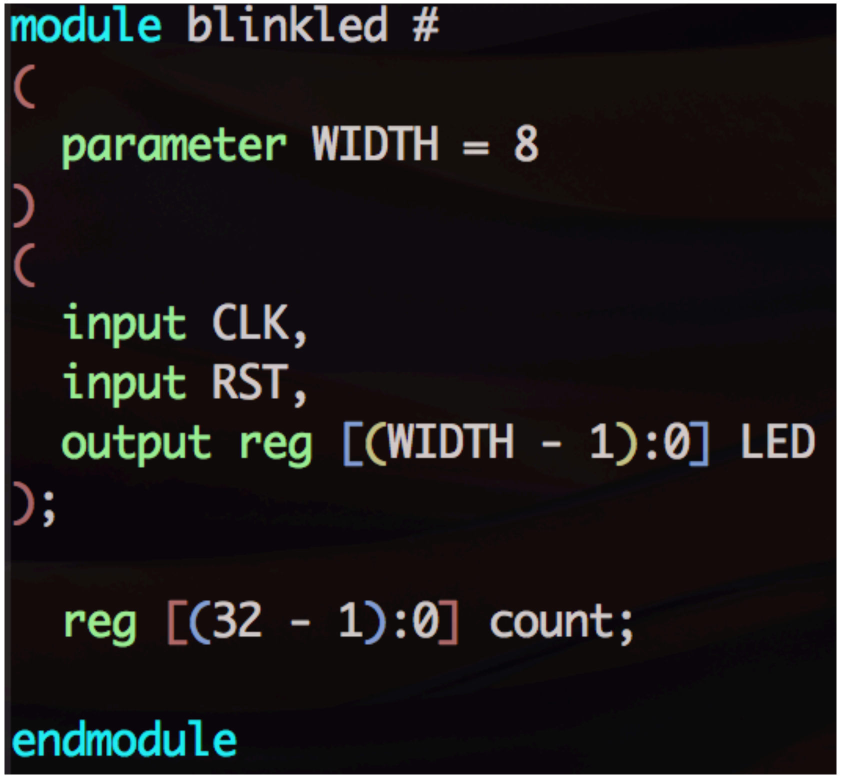
```
def mkLed():
    m = Module('blinkled')
    width = m.Parameter('WIDTH', 8)
    clk = m.Input('CLK')
    rst = m.Input('RST')
    led = m.OutputReg('LED', width)
    count = m.Reg('count', 32)

    m.Always(Posedge(clk))(
        If(rst)(
            count(0)
        ).Else(
            If(count == 1023)(
                count(0)
            ).Else(
                count(count + 1)
            )
        ))

    m.Always(Posedge(clk))(
        If(rst)(
            led(0)
        ).Else(
            If(count == 1024 - 1)(
                led(led + 1)
            )
        ))

    return m
```

Code generation by run

- Verilog code is obtained by calling *to_verilog()* method

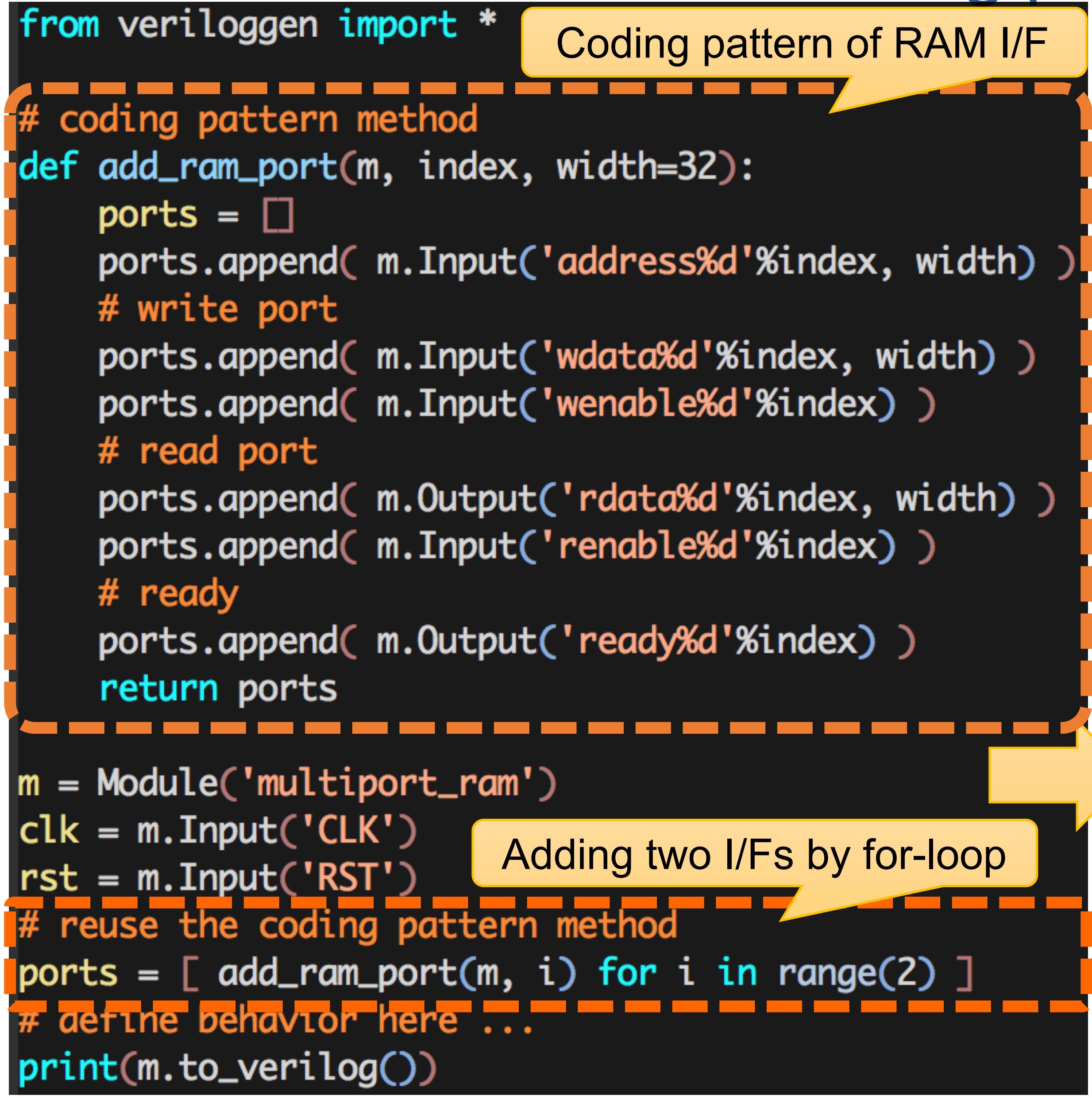


```python
import sys
import os
from veriloggen import *

m = Module('blinkled')
clk = m.Input('CLK')
rst = m.Input('RST')
led = m.OutputReg('LED', width)
count = m.Reg('count', 32)

verilog = m.to_verilog('tmp.v')
print(verilog)
```

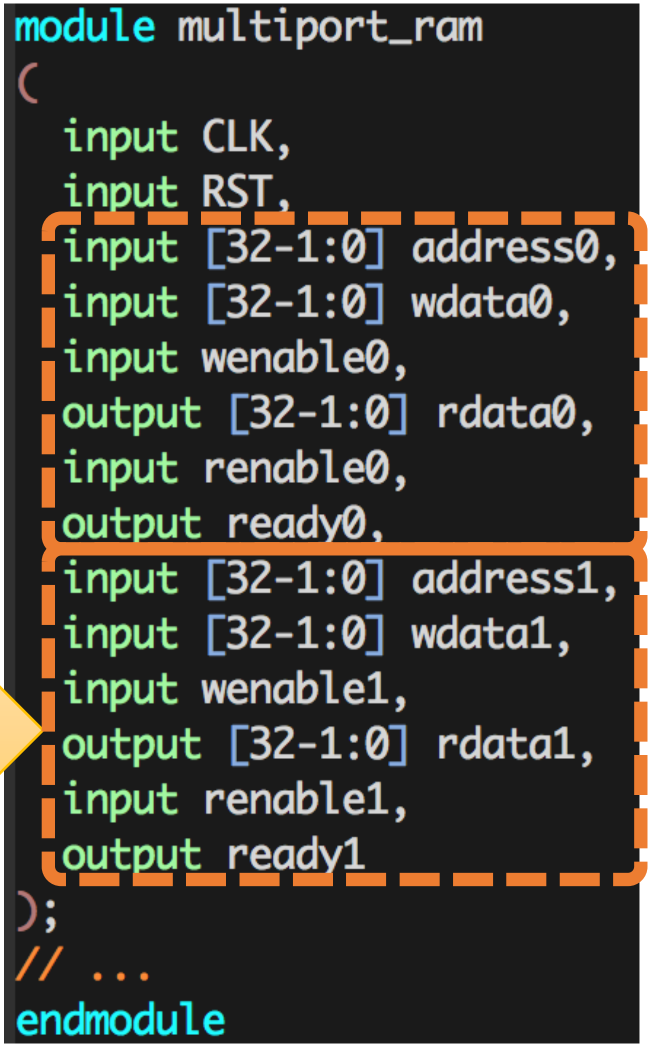


```verilog
module blinkled #
(
  parameter WIDTH = 8
)
(

  input CLK,
  input RST,
  output reg [(WIDTH - 1):0] LED
);

  reg [(32 - 1):0] count;

endmodule
```

User-defined explicit abstraction: Method extraction for coding pattern reuse



```
from veriloggen import *

# coding pattern method
def add_ram_port(m, index, width=32):
    ports = []
    ports.append( m.Input('address%d'%index, width) )
    # write port
    ports.append( m.Input('wdata%d'%index, width) )
    ports.append( m.Input('wenable%d'%index) )
    # read port
    ports.append( m.Output('rdata%d'%index, width) )
    ports.append( m.Input('renable%d'%index) )
    # ready
    ports.append( m.Output('ready%d'%index) )
    return ports

m = Module('multiport_ram')
clk = m.Input('CLK')
rst = m.Input('RST')
# reuse the coding pattern method
ports = [ add_ram_port(m, i) for i in range(2) ]
# define behavior here ...
print(m.to_verilog())
```



```
module multiport_ram
(
  input CLK,
  input RST,
  input [32-1:0] address0,
  input [32-1:0] wdata0,
  input wenable0,
  output [32-1:0] rdata0,
  input renable0,
  output ready0,
  input [32-1:0] address1,
  input [32-1:0] wdata1,
  input wenable1,
  output [32-1:0] rdata1,
  input renable1,
  output ready1
);
// ...
endmodule
```

Veriloggen.FSM: FSM manager

Veriloggen has some built-in abstractions: FSM, Seq, Fixed, …

ex)
UART sender w/ FSM

FSM manager object

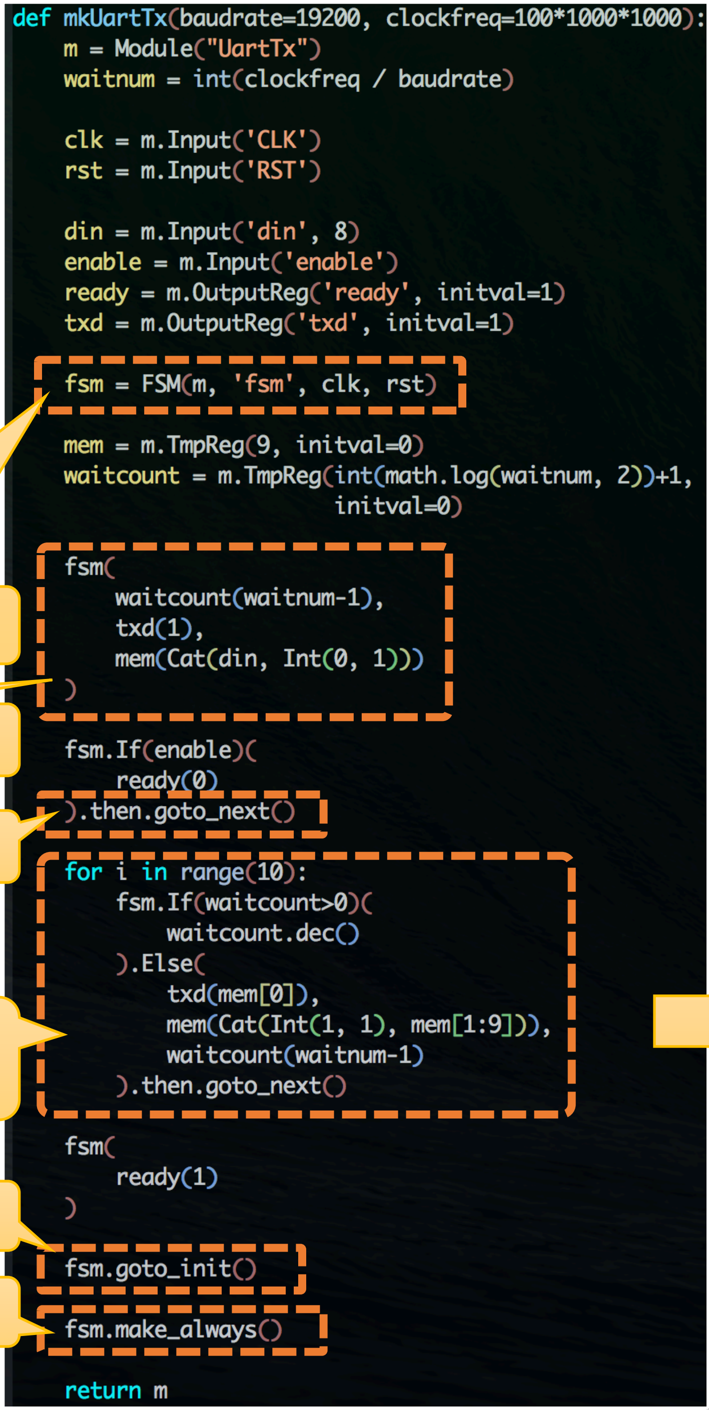
Assignments for each state

Going to next state w/o labels

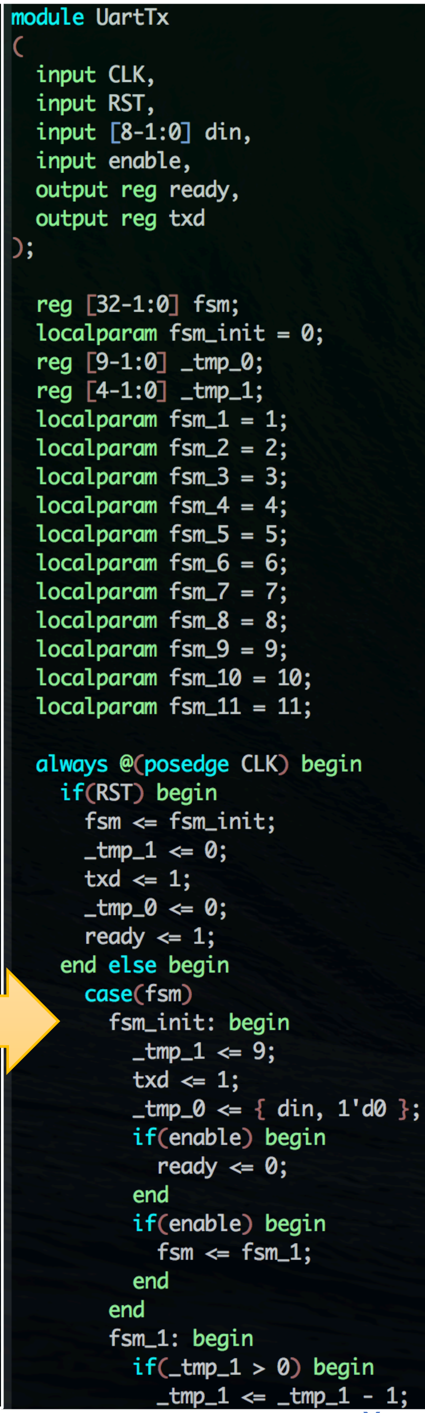
Repeating an FSM pattern by for-loop

Going to initial state

Synthesizing FSM circuits



```python
def mkUartTx(baudrate=19200, clockfreq=100*1000*1000):
    m = Module("UartTx")
    waitnum = int(clockfreq / baudrate)

    clk = m.Input('CLK')
    rst = m.Input('RST')

    din = m.Input('din', 8)
    enable = m.Input('enable')
    ready = m.OutputReg('ready', initval=1)
    txd = m.OutputReg('txd', initval=1)

    fsm = FSM(m, 'fsm', clk, rst)

    mem = m.TmpReg(9, initval=0)
    waitcount = m.TmpReg(int(math.log(waitnum, 2))+1,
                         initval=0)

    fsm(
        waitcount(waitnum-1),
        txd(1),
        mem(Cat(din, Int(0, 1)))
    )

    fsm.If(enable)(
        ready(0)
    ).then.goto_next()

    for i in range(10):
        fsm.If(waitcount>0)(
            waitcount.dec()
        ).Else(
            txd(mem[0]),
            mem(Cat(Int(1, 1), mem[1:9])),
            waitcount(waitnum-1)
        ).then.goto_next()

    fsm(
        ready(1)
    )

    fsm.goto_init()

    fsm.make_always()

    return m
```



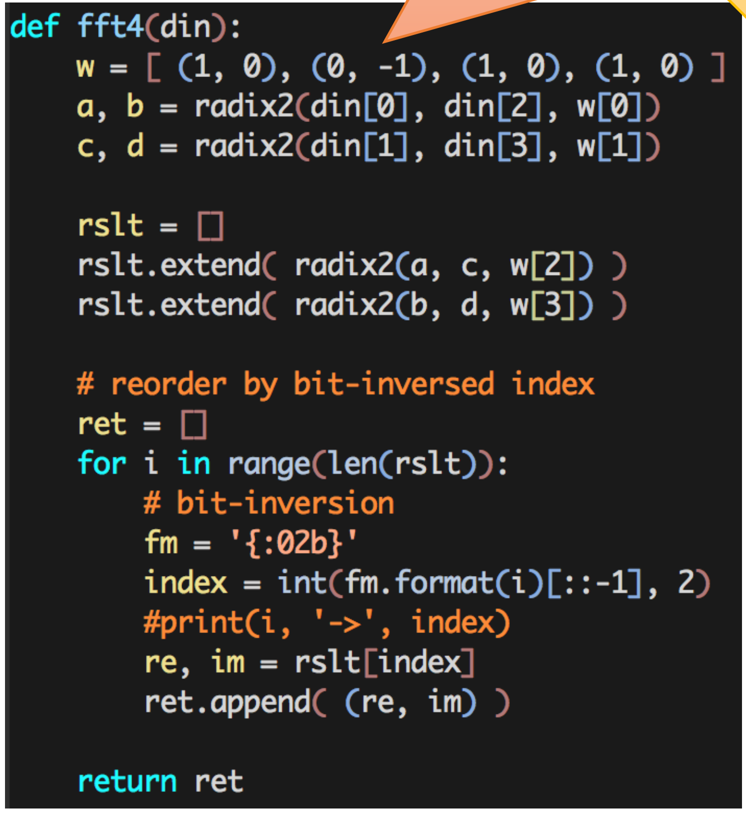
```verilog
module UartTx
(
  input CLK,
  input RST,
  input [8-1:0] din,
  input enable,
  output reg ready,
  output reg txd
);

  reg [32-1:0] fsm;
  localparam fsm_init = 0;
  reg [9-1:0] _tmp_0;
  reg [4-1:0] _tmp_1;
  localparam fsm_1 = 1;
  localparam fsm_2 = 2;
  localparam fsm_3 = 3;
  localparam fsm_4 = 4;
  localparam fsm_5 = 5;
  localparam fsm_6 = 6;
  localparam fsm_7 = 7;
  localparam fsm_8 = 8;
  localparam fsm_9 = 9;
  localparam fsm_10 = 10;
  localparam fsm_11 = 11;

  always @(posedge CLK) begin
    if(RST) begin
      fsm <= fsm_init;
      _tmp_1 <= 0;
      txd <= 1;
      _tmp_0 <= 0;
      ready <= 1;
    end else begin
      case(fsm)
        fsm_init: begin
          _tmp_1 <= 9;
          txd <= 1;
          _tmp_0 <= { din, 1'd0 };
          if(enable) begin
            ready <= 0;
          end
          if(enable) begin
            fsm <= fsm_1;
          end
        end
        fsm_1: begin
          if(_tmp_1 > 0) begin
            _tmp_1 <= _tmp_1 - 1;
```

Veriloggen.Dataflow: Pipeline synthesis by operator overloads

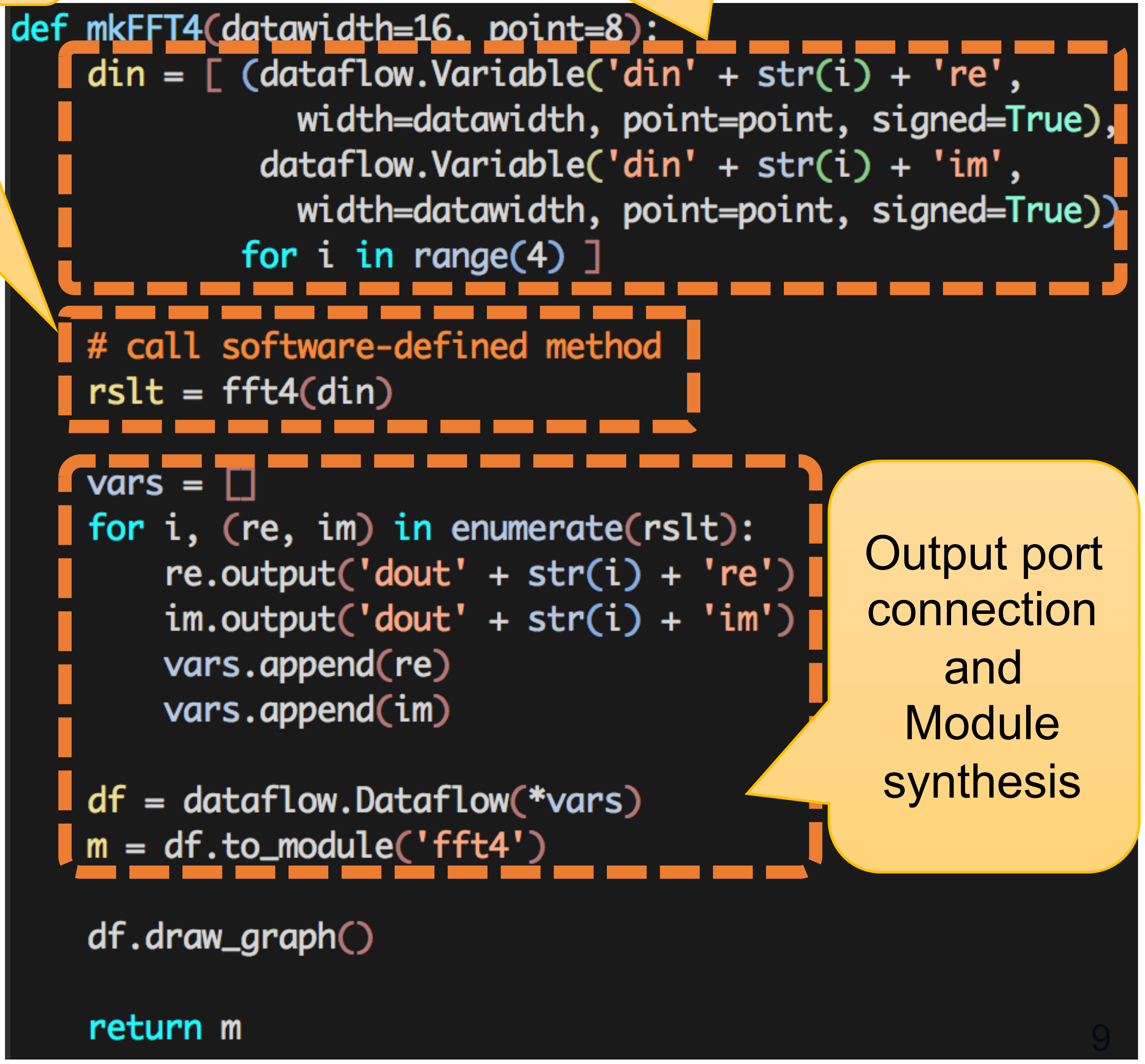
Passing Dataflow variables instead of normal variables

Dataflow Variable

Normal Python method tha can be executed as SW



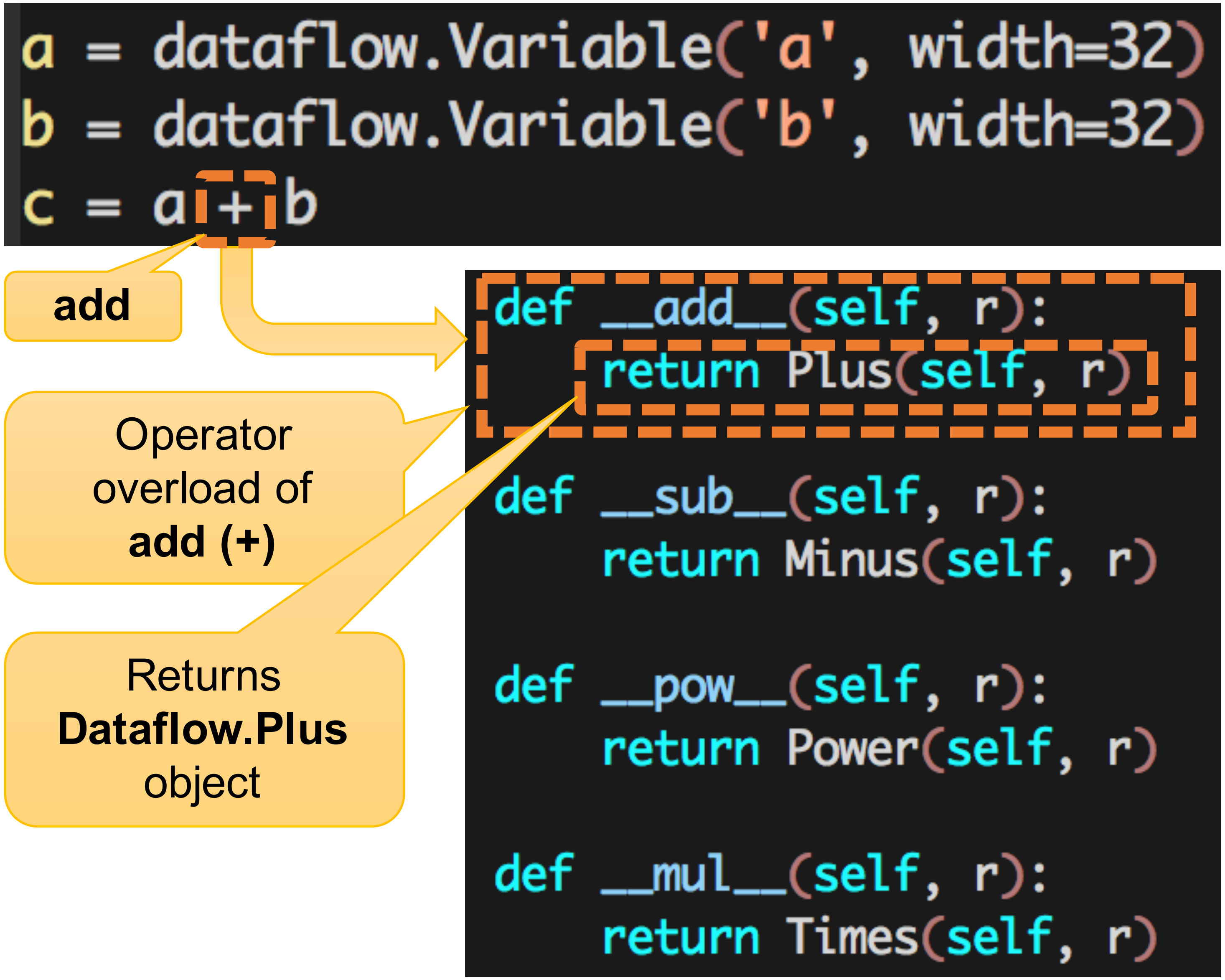
```python
def fft4(din):
    w = [ (1, 0), (0, -1), (1, 0), (1, 0) ]
    a, b = radix2(din[0], din[2], w[0])
    c, d = radix2(din[1], din[3], w[1])

    rslt = []
    rslt.extend( radix2(a, c, w[2]) )
    rslt.extend( radix2(b, d, w[3]) )

    # reorder by bit-inversed index
    ret = []
    for i in range(len(rslt)):
        # bit-inversion
        fm = '{:02b}'
        index = int(fm.format(i)[::-1], 2)
        #print(i, '->', index)
        re, im = rslt[index]
        ret.append( (re, im) )

    return ret
```



```python
def mkFFT4(datawidth=16, point=8):
    din = [ (dataflow.Variable('din' + str(i) + 're',
                   width=datawidth, point=point, signed=True),
             dataflow.Variable('din' + str(i) + 'im',
                   width=datawidth, point=point, signed=True))
           for i in range(4) ]

    # call software-defined method
    rslt = fft4(din)

    vars = []
    for i, (re, im) in enumerate(rslt):
        re.output('dout' + str(i) + 're')
        im.output('dout' + str(i) + 'im')
        vars.append(re)
        vars.append(im)

    df = dataflow.Dataflow(*vars)
    m = df.to_module('fft4')

    df.draw_graph()

    return m
```

Output port connection and Module synthesis

Operator overload for dataflow



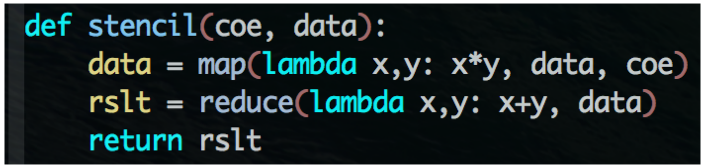
```
a = dataflow.Variable('a', width=32)
b = dataflow.Variable('b', width=32)
c = a + b
```

```
def __add__(self, r):
    return Plus(self, r)

def __sub__(self, r):
    return Minus(self, r)

def __pow__(self, r):
    return Power(self, r)

def __mul__(self, r):
    return Times(self, r)
```

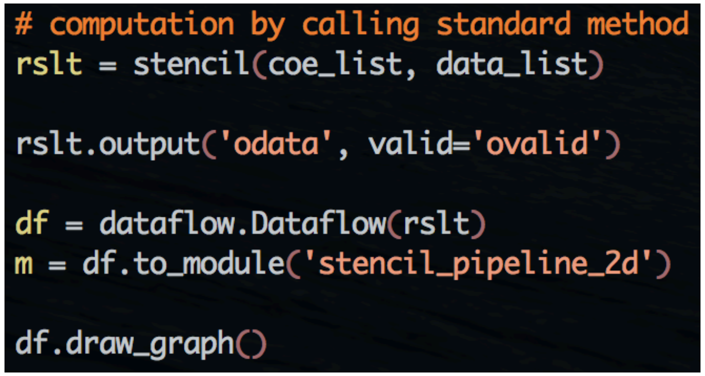
add
Operator overload of **add (+)**
Returns **Dataflow.Plus** object

Graph output by Veriloggen.Dataflow

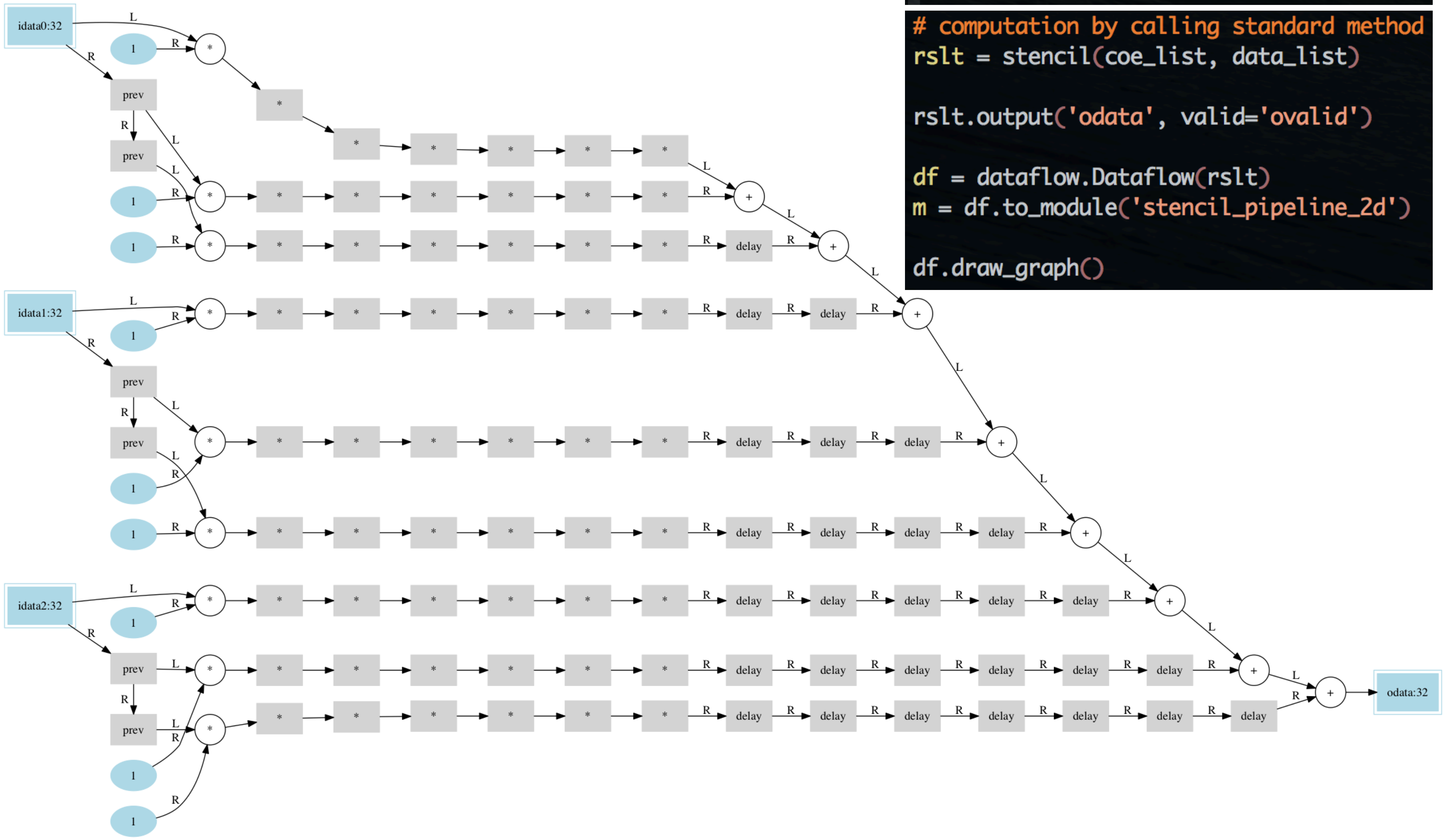
3x3 stencil pipeline



```
def stencil(coe, data):
    data = map(lambda x,y: x*y, data, coe)
    rslt = reduce(lambda x,y: x+y, data)
    return rslt
```



```
# computation by calling standard method
rslt = stencil(coe_list, data_list)

rslt.output('odata', valid='ovalid')

df = dataflow.Dataflow(rslt)
m = df.to_module('stencil_pipeline_2d')

df.draw_graph()
```



Difference to HDL and HLS

- In DSL/HLS, source code structure is a circuit definition
  - Reflection: getting the source code structure
    - ✓ A compiler analyzes the source code and convert into dataflow, etc.
  - When a new part is added, the source code must be changed
    - ✓ Of course, a frequently-appeared pattern also must be described in the source code again☹
  - Subset of the original language syntax can be utilized☹
- Veriloggen explicitly constructs a hardware source code
  - No reflection: a target source code is constructed by run
    - ✓ Frequently-appeared coding patterns can be summarized by method extraction and new user-defined class definition
    - ✓ All python features can be utilized for the code construction☺

Evaluation: Productivity of Veriloggen

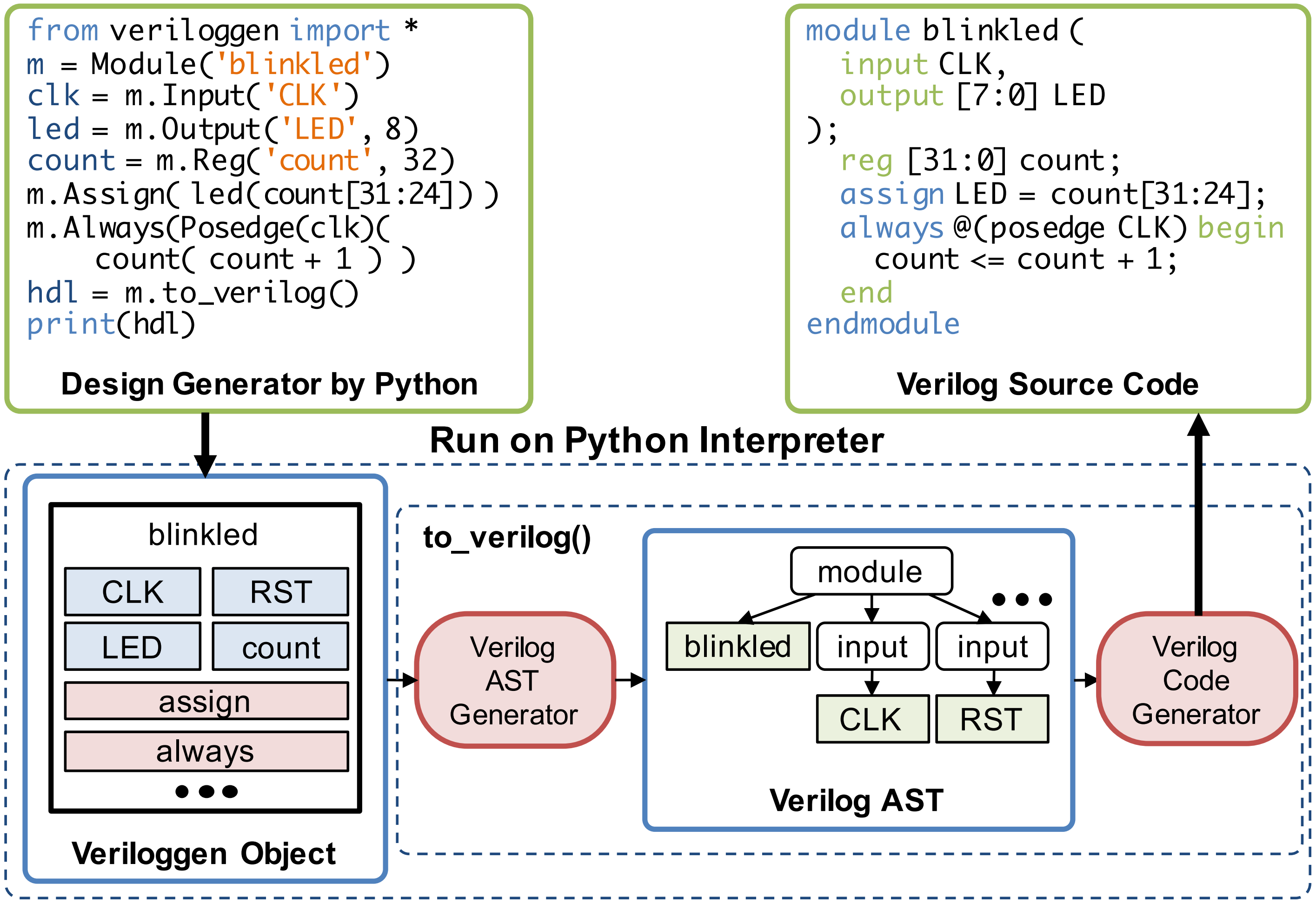
Python Verilog

App	Python LOC	Verilog LOC	#Reg	#LUT	#DSP	Fmax [MHz]	Description
LED	35	31	18	33	0	352	Blinking LED (#LEDs=8)
Sort-4	45	713	1295	587	0	366	Sorting network (#data=4, 32-bit int)
Sort-8	(= Sort-4)	3575	6709	2712	0	370	Sorting network (#data=8, 32-bit int)
Sort-16	(= Sort-4)	15779	29873	11576	0	370	Sorting network (#data=16, 32-bit int)
Sort-32	(= Sort-4)	66107	125545	47736	0	370	Sorting network (#data=32, 32-bit int)
MM-p (all)	[illegible]	[illegible]	1891	2219	3	190	Matrix mult pipeline in Python (32-bit int)
MM-p (FSM)	[illegible]	[illegible]	N/A	N/A	N/A	N/A	Matrix mult pipeline in Python (32-bit int)
MM-v (all)	N/A	532	2041	2511	3	146	Matrix mult pipeline in Verilog (32-bit int)
MM-v (FSM)	N/A	120	N/A	N/A	N/A	N/A	Matrix mult pipeline in Verilog (32-bit int)

- Various hardware structures can be synthesized from a single Python source code
  - ex) obtained 4 sort circuits from Python code of 45 lines
  - Delay registers and stall circuits are automatically inserted
  - **→High productivity of custom computing pipeline development**

Conclusion

- Customizable hardware abstraction is proposed
  - Veriloggen: Explicit hardware modeling by Python



```
from veriloggen import *
m = Module('blinkled')
clk = m.Input('CLK')
led = m.Output('LED', 8)
count = m.Reg('count', 32)
m.Assign( led(count[31:24]) )
m.Always(Posedge(clk)(
    count( count + 1 ) )
hdl = m.to_verilog()
print(hdl)
```

Design Generator by Python

```
module blinkled (
  input CLK,
  output [7:0] LED
);
  reg [31:0] count;
  assign LED = count[31:24];
  always @(posedge CLK) begin
    count <= count + 1;
  end
endmodule
```

Verilog Source Code