Python to Verilog

p2v

Digital design in Python

Python to Verilog specification

Eyal Hochberg 1/1/25 Python to Verilog

COPYRIGHT

Email: eyalhoc@gmail.com

Visit the source at https://github.com/eyalhoc/p2v

VERSION HISTORY TABLE

Version	Description	Date
0.1.0	Initial version – pre alpha testing	6/9/2025

CONTENTS

INTRODUCTION

What is P2V?

P2V is a python library used to generate synthesizable RTL. The RTL modules written in Python are converted to Verilog.

Who is P2V meant for?

P2V is meant for chip designers familiar with Verilog and Python.

Why don't we just keep on using Verilog?

Verilog code is a very old programming language; it was invented when chips were much smaller and much simpler. The main advantage of Verilog is that it allows control of the RTL on a very low level, on the other hand it fundamentally lacks generic features severely reducing the reusability of the code. In some cases, this is overcome by pairing the RTL design with scripts that try to make it more "generic" but it is artificial and does not scale well. P2V combines low-level design (when needed) with high level structures (where possible). This not only makes the design easier to write and maintain but also maintains the architectural intent within the source code.

Why Python?

Python is a mature and very popular language; it is easy to code and read. Once the RTL design is written in Python the designer gets access to the entire Python eco-system enabling endless possibilities, for example connecting the design to algorithms, using configuration files like csv, json or excel formats or using math libraries like numpy.

Does P2V support verification?

Yes, P2V has testing built into it. This is not meant to replace full verification of the design but enables basic testing, connectivity and robustness checking. P2V offers 3 levels of validation:

1. Linter is performed both on the Python code and on the generated Verilog code.
2. P2V supports generating random permutations of the design verifying nothing breaks for any combination of parameters.
3. P2V supports building test-benches that are simulated within the tool.

INSTALLATION

P2V is a native Python3 library, it needs no special installations for its basic function.

Pip install TBD

Beyond basic functionality P2V does take advantage of the following open-source tools, and their absence will shut off their corresponding features:

1. Verible – used for Verilog indentation
   https://github.com/chipsalliance/verible
2. Slang – used for Verilog module instantiations
   https://github.com/MikePopoloski/slang
3. Verilator – Verilog linter
   https://verilator.org/guide/latest/install.html
4. Iverilog – Verilog simulator
   https://steveicarus.github.io/iverilog/usage/installation.html

Using partially installed P2V

P2V can run without all its dependencies, but in order to do that the -allow_missing_tools should be added to the command line, that will allow, for example, running without indenting files if the indentation tool is not installed.

HELLO WORLD

Getting started with P2V

- Example files are under p2v/tutorial/example0_hello_world

CREATE THE PYTHON SOURCE FILE

Create a Python file hello_world.py with a single class hello_world

```
from p2v import p2v

class hello_world(p2v):
    def module(self):
        self.set_modname()

        return self.write()
```

BUILD THE VERILOG MODULE BY RUNNING PYTHON


```
python3 p2v/tutorial/example0_hello_world/hello_world.py
p2v-INFO: starting with seed 1
p2v-DEBUG: created: hello_world.sv
p2v-INFO: verilog generation completed successfully (1 sec)
p2v-INFO: verilog lint completed successfully
p2v-INFO: completed successfully
```

EXAMINING THE VERILOG MODULE

```
cat cache/rtl/hello_world.sv
module hello_world ();



endmodule
```

EXAMINING THE P2V LOGFILE

```
cat cache/p2v.log
p2v-INFO: starting with seed 1
p2v-DEBUG: created: hello_world.sv
p2v-INFO: verilog generation completed successfully (1 sec)
p2v-INFO: verilog lint completed successfully
p2v-INFO: completed successfully
```


TUTORIAL

Example 1: adder

- Example files are under p2v/tutorial/example1_adder

In this example we will create a simple module that performs addition Step by step we will increase complexity and also show how P2V is used for verification.

STEP 1: CREATE A MODULE THAT ADDS TWO 8 BIT NUMBERS

In this lesson we will learn

- Module declaration
- Defining ports
- Assigning signals

```
from p2v import p2v # all modules inherit the p2v class

class adder(p2v):
    def module(self):
        self.set_modname() # sets the Verilog module name

        self.input("a", 8) # 8 bit input
        self.input("b", 8)
        self.output("o", 8) # 8 bit output

        self.assign("o", "a + b") # generates Verilog assign

        return self.write() # Write the Verilog file
```


Example explained

The module has 2 inputs, and one output all are of a fixed width of 8 bits. Logical assignment is performed on the inputs.

The function set_modname() is mandatory it set the Verilog module name.

The function write() is also mandatory, it creates the Verilog file.

Examining the Verilog file

```
module adder (
    input  logic [7:0] a,
    input  logic [7:0] b,
    output logic [7:0] o
);

    assign o = a + b;

endmodule
```

The Verilog file is created as similar as possible to the Python source code’s format and is indented.

STEP 2: MAKE THE DATA WIDTH PARAMETRIC

In this lesson we will learn

- Adding module parameters
- Adding parameter assertions

```
from p2v import p2v

class adder(p2v):
    def module(self, bits=8):
        self.set_param(bits, int, bits > 0, remark="data width") # declares the module parameter, sets the allowed types and sets optional constraints
        self.set_modname()

        self.input("a", bits)
        self.input("b", bits)
        self.output("o", bits)

        self.assign("o", "a + b")

        return self.write()
```


Example explained

The module receives the parameter bits. All parameters must be registered using the set_param() function. Set_param() checks the parameter type and additional optional constraints on the parameter.

Examining the Verilog file

```
module adder__bits8 (
    input  logic [7:0] a,
    input  logic [7:0] b,
    output logic [7:0] o
);

    // module parameters:
    // bits = 8 (int): data width

    assign o = a + b;

endmodule
```

Notice that the Verilog module name has been suffixed to make it unique by it’s parameters.

STEP 3: ADD A CLOCK AND SAMPLE THE OUTPUT

In this lesson we will learn

- Using clocks
- Sampling signals (creating FFs)

```python
from p2v import p2v, clock, default_clk

class adder(p2v):
    def module(self, clk=default_clk, bits=8):
        self.set_param(clk, clock) # module param is a p2v clock
        self.set_param(bits, int, bits > 0, remark="data width")
        self.set_modname()

        self.input(clk) # default clock uses an async reset

        self.input("valid") # default width is 1 bit
        self.input("a", bits)
        self.input("b", bits)
        self.output("o", bits)
        self.output("valid_out") # default width is 1 bit

        self.sample(clk, "o", "a + b", valid="valid") # creates FFs
        self.sample(clk, "valid_out", "valid") # creates a free running FF

        return self.write()
```


Example explained

P2v uses a special class for clocks, these contain a clock and optional async and / or sync resets. In this example the clock is not defined in the module but received as a parameter. A default clock is assigned to the module parameter to enable the module to compile without command line arguments.

Self.sample() creates FFs for a specific clock domain, if the valid argument is not set the FF is free running.

Examining the Verilog file

```
module adder__bits8 (
    input logic clk,
    input logic rst_n,
    input logic valid,
    input logic [7:0] a,
    input logic [7:0] b,
    output logic [7:0] o,
    output logic valid_out
);

    // module parameters:
    // clk = clk (p2v_clock)
    // bits = 8 (int): data width

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) o <= 8'd0;
        else if (valid) o <= a + b;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) valid_out <= 1'd0;
        else valid_out <= valid;


endmodule
```


Notice that the default clock uses an async reset. Changing the clock to use a sync reset or both resets does not change the Python source code but will affect the Verilog FF implementations.

STEP 4: CREATING AN ADDER TREE

In this lesson we will learn

- Creating and connecting son modules
- Misc utility functions

```
from p2v import p2v, misc, clock, default_clk # misc provides general purpose
functions

class adder(p2v):
    def module(self, clk=default_clk, bits=8, num=8):
        self.set_param(clk, clock)
        self.set_param(bits, int, bits > 0, remark="data width")
        self.set_param(num, int, num > 0 and misc.is_pow2(num), remark="number of
inputs")
        self.set_modname()

        self.input(clk)

        self.input("valid")
        for n in range(num):
            self.input(f"i{n}", bits)
        self.output("o", bits)
        self.output("valid_out")

        if num == 2:
            self.sample(clk, "o", "i0 + i1", valid="valid")
            self.sample(clk, "valid_out", "valid")


        else:
            son_num = num // 2
            for i in range(2):
                self.logic(f"o{i}", bits)
                self.logic(f"valid_out{i}")

                son = adder(self).module(clk, bits=bits, num=son_num) # creates a
module and returns the ports to be connected
                son.connect_in(clk) # connects the clock and resets
                son.connect_in("valid") # assumes port name equals wire name
                for n in range(son_num):
```

```
                son.connect_in(f"i{n}", f"i{son_num*i+n}")
            son.connect_out("o", f"o{i}")
            son.connect_out("valid_out", f"valid_out{i}")
            son.inst(suffix=i)


        # add the results
        son = adder(self).module(clk, bits=bits, num=2)
        son.connect_in(clk)
        son.connect_in("valid", "valid_out0 & valid_out1")
        son.connect_in("i0", "o0")
        son.connect_in("i1", "o1")
        son.connect_out("o")
        son.connect_out("valid_out")
        son.inst(suffix="_out") # instance name is suffixed to make it unique


    return self.write()
```

Example explained:

The module calls itself recursively to create an adder tree. Each step of the hierarchy uses half the parent's inputs. For simplicity the number of inputs receives as a module parameter is asserted to be a power of 2. The function misc.is_pow2() is used for that, misc contains a variety of utility functions commonly used.

Son variable is used to receive the son's ports and connect them. Port connections must explicitly specify the direction, if the connecting wire is absent it is assumed that the wire name is the same as the port name.

The son.inst() functions inserts the instantiation into the module, the suffix parameter is used to make the instance name unique.

Examining the Verilog file

```
module adder__bits8_num8 (
    input logic clk,
    input logic rst_n,
    input logic valid,
    input logic [7:0] i0,
    input logic [7:0] i1,
    input logic [7:0] i2,
    input logic [7:0] i3,
    input logic [7:0] i4,
```



```
    input logic [7:0] i5,
    input logic [7:0] i6,
    input logic [7:0] i7,
    output logic [7:0] o,
    output logic valid_out
);

    // module parameters:
    // clk = clk (p2v_clock)
    // bits = 8 (int): data width
    // num = 8 (int): number of inputs

    logic [7:0] o0;
    logic valid_out0;
    adder__bits8_num4 adder0 (
        .clk(clk),  // input
        .rst_n(rst_n),  // input
        .valid(valid),  // input
        .i0(i0),  // input
        .i1(i1),  // input
        .i2(i2),  // input
        .i3(i3),  // input
        .o(o0),  // output
        .valid_out(valid_out0)  // output
    );

    logic [7:0] o1;
    logic valid_out1;
    adder__bits8_num4 adder1 (
        .clk(clk),  // input
        .rst_n(rst_n),  // input
        .valid(valid),  // input
        .i0(i4),  // input
        .i1(i5),  // input
        .i2(i6),  // input
        .i3(i7),  // input
        .o(o1),  // output
        .valid_out(valid_out1)  // output
    );

    adder__bits8_num2 adder_out (
        .clk(clk),  // input
        .rst_n(rst_n),  // input
        .valid(valid_out0 & valid_out1),  // input
        .i0(o0),  // input
```



```
        .i1(o1),  // input
        .o(o),  // output
        .valid_out(valid_out)  // output
    );


endmodule
```


STEP 5: DEFINE RANDOM BOUNDARIES

In this lesson we will learn

- Defining module’s random ranges
- Generating module random permutations for robustness checking
- Random seed

```python
from p2v import p2v, misc, clock, default_clk

class adder(p2v):
    def module(self, clk=default_clk, bits=8, num=32):

        # same as previous example
        . . .

        return self.write()


    def gen(self): # reserved function for defining randomization of module
parameters
        args = {}
        args["bits"] = self.tb.rand_int(1, 128) # random integer in this range
        args["num"] = 1 << self.tb.rand_int(1, 8)
        return args
```


Example explained

In p2v parameter randomization is not defined in the test bench but within the module itself using the reserved function gen().

Gen() returns a dictionary with the random parameters, it is not mandatory to random all parameters, some can retain their default values.

In this example self.tb.rand_int() is used, this function selects in integer value between two values, other similar functions exist for randomizing different types.

How to run permutations

P2v uses a powerful lint tool which can find many bugs just be compiling the code. Compiling the code for multiple random variations is a good way to check robustness.

P2v random engines use a seed for exact reproduction of scenarios. The default seed of the tool is 1 ensuring a consistent random behavior, seed 0 generates a random seed, the generated seed can be seen in the log below.

```
python3 p2v/tutorial/example1_adder/step_5/adder.py -seed 0 -gen 2

p2v-INFO: starting with seed 22128
p2v-INFO: starting gen iteration 0/1
p2v-DEBUG: created: adder__bits92_num2.sv
p2v-DEBUG: created: adder__bits92_num4.sv
p2v-INFO: verilog lint completed successfully
p2v-INFO: starting gen iteration 1/1
p2v-DEBUG: created: adder__bits125_num2.sv
p2v-DEBUG: created: adder__bits125_num4.sv
p2v-DEBUG: created: adder__bits125_num8.sv
p2v-DEBUG: created: adder__bits125_num16.sv
p2v-DEBUG: created: adder__bits125_num32.sv
p2v-INFO: verilog lint completed successfully
p2v-INFO: completed successfully
```


STEP 6: SUPPORT FLOAT16 ADDITION

In this lesson we will learn

- Instantiating Verilog modules
- Static and dynamic assertions
- Allowing unused and undriven

```
from p2v import p2v, misc, clock, default_clk

class adder(p2v):
    def module(self, clk=default_clk, bits=8, num=32, float16=False):
        self.set_param(clk, clock)
        self.set_param(bits, int, bits > 0, remark="data width")
        self.set_param(num, int, num > 0 and misc.is_pow2(num), remark="number of
inputs")
        self.set_param(float16, bool, remark="use a float16 adder")
        self.set_modname()

        if float16:
            self.assert_static(bits == 16, "float type only supports float16")


        self.input(clk)

        self.input("valid")
        for n in range(num):
            self.input(f"i{n}", bits)
        self.output("o", bits)
        self.output("valid_out")

        if num == 2:
            self.logic("o_pre", bits)
            if float16:
                float16_stat = ["overflow", "zero", "NaN", "precisionLost"]
                self.logic(float16_stat)

                son = self.verilog_module("float_adder")
                son.connect_in("num1", "i0")
                son.connect_in("num2", "i1")
                son.connect_out("result", "o_pre")
                for stat in float16_stat:
                    son.connect_out(stat)
                son.inst()
```



```
                for stat in float16_stat:
                    if stat not in ["precisionLost"]:
                        self.assert_never(clk, stat, f"received unexpected
{stat}")
                    else:
                        self.allow_unused(stat)
            else:
                self.assign("o_pre", "i0 + i1")

            self.sample(clk, "o", "o_pre", valid="valid")
            self.sample(clk, "valid_out", "valid")

        else:
            son_num = num // 2
            for i in range(2):
                self.logic(f"o{i}", bits)
                self.logic(f"valid_out{i}")

                son = adder(self).module(clk, bits=bits, num=son_num,
float16=float16)
                son.connect_in(clk)
                son.connect_in("valid")
                for n in range(son_num):
                    son.connect_in(f"i{n}", f"i{son_num*i+n}")
                son.connect_out("o", f"o{i}")
                son.connect_out("valid_out", f"valid_out{i}")
                son.inst(suffix=i)


            # add the results
            son = adder(self).module(clk, bits=bits, num=2, float16=float16)
            son.connect_in(clk)
            son.connect_in("valid", "valid_out0 & valid_out1")
            son.connect_in("i0", "o0")
            son.connect_in("i1", "o1")
            son.connect_out("o")
            son.connect_out("valid_out")
            son.inst(suffix="_out")


        return self.write()


    def gen(self):
```



```
        args = {}
        args["float16"] = self.tb.rand_bool()
        if args["float16"]:
            args["bits"] = 16
        else:
            args["bits"] = self.tb.rand_int(1, 128)
        args["num"] = 1 << self.tb.rand_int(1, 8)
        return args
```

Example explained

To support float16 addition I looked online and found the following highly stared project:

https://github.com/suoglu/Fixed-Floating-Point-Adder-Multiplier/blob/master/Sources/adderMultiplier16.v

Once I downloaded the file and added it to my project I can instantiate the Verilog module float_adder, this is done by using the verilog_module() function. Besides that, connectivity to Verilog modules is similar to p2v modules.

This example uses both static assertions which test python variables and dynamic assertions that check Verilog signals in simulation.

assert_static() is a static assertion checking that data width is 16 when building for float16 and assert_never() is a dynamic assertion checking that the Verilog float16 adder does not give undesired statuses like overflow.

P2v used a powerful linter, among other things it checks that all signals and driven and used. In this case the float16 adder statuses are used for assertions but since ‘precisionLost’ is allowed and should not cause an assertion the signal is marked by self.allow_unused() in order not to cause an unused lint error.

STEP 7: CREATE TEST-BENCH

In this lesson we will learn

- How to build and run a basic test-bench
- How to generate clocks

```python
from p2v import p2v, misc, clock

import adder

class tb_adder(p2v):
    def module(self, async_reset=True, size=32):
        self.set_param(async_reset, bool, remark="sync reset or async reset")
        self.set_param(size, int, size > 0, remark="number of inputs to test")
        self.set_modname("tb") # explicitly set module name

        if async_reset:
            clk = clock("clk", rst_n="resetn") # clock with async reset
        else:
            clk = clock("clk", reset="reset") # clock with sync reset

        self.logic(clk)
        self.tb.gen_clk(clk, cycle=self.tb.rand_int(1, 20)) # generates the clock and resets in simulation


        args = adder.adder(self).gen_rand_args(override={"float16":False}) # extract the random module parameters from adder itself. float16 is disabled since it is not yet supported in this test-bench
        num = args["num"] # extract module parameter
        bits = args["bits"]

        self.logic("valid")
        inputs = []
        for n in range(num):
            inputs.append(f"i{n}")
        self.logic(inputs, bits, initial=0)
        self.logic("o", bits)
        self.logic("valid_out")

        son = adder.adder(self).module(clk, **args)
        son.connect_in(clk)
        son.connect_in("valid")
```



```
        for n in range(num):
            son.connect_in(f"i{n}")
        son.connect_out("o")
        son.connect_out("valid_out")
        son.inst()


        self.logic("en", initial=0)
        self.sample(clk, "valid", "en")

        self.tb.fifo("data_in_q", bits*num) # create behavioral fifo
        self.tb.fifo("expected_q", bits) # create behavioral fifo
        self.logic("data_in", bits*num, initial=0)
        self.logic("expected", bits, initial=0)

        self.line(f"""
                    initial
                        begin
                   """)
        for i in range(size):
            input_vec = []
            input_sum = 0
            for j in range(num):
                val = self.tb.rand_int(1<<bits)
                input_sum += val
                input_vec.append(misc.hex(val, bits))
            self.line(f"data_in_q.push_back({misc.concat(input_vec)});") #
SystemVerilog fifo push
            self.line(f"expected_q.push_back({misc.hex(input_sum, bits)});")
        self.line(f"""
                        end
                   """)


        self.line(f""" # any code is allowed in line() function, directly passed
to Verilog file without parsing
                    initial
                        begin
                            {misc.cond(async_reset, f"@(posedge {clk.rst_n});")}
                            repeat (10) @(posedge {clk});
                            en = 1;
                        end

                        // drive inputs
                        always @(posedge {clk})
```



```
                        if (valid && (data_in_q.size() > 0))
                            begin
                                data_in = data_in_q.pop_front();#
SystemVerilog fifo pop
                                {misc.concat(inputs)} = data_in;
                            end

                    // check output
                    always @(posedge {clk})
                        if (valid_out)
                            begin
                                expected = expected_q.pop_front();
                                {self.tb.test_fail(condition=f"o !==
expected", message="mismatch expected: 0x%0h, actual: 0x%0h", params=["expected",
"o"])} # end test with an error if fail condition is met
                                if (expected_q.size() == 0)
                                    {self.tb.test_pass(message=f"successfully
tested {size} additions")} # successfully end test
                            end
                """)


        self.allow_unused(["valid_out", "o", "data_in", "expected"]) # signals
used inside line() function are not parsed


        self.tb.set_timeout(clk, size * 100) # set timeout
        self.tb.dump() # set fst dump file

        return self.write()
```


Example explained

This test bench supports a clock with either a sync or an async reset. This is determined by a module parameter.

Once the clock type is determined the clock frequency is randomized and the clock is generated using the gen_clk() function.

In p2v module parameters are not randomized by the test bench but by the module itself. The test bench extracts the adder's parameters by calling get_rand_args(). In this example the float16 parameter is overridden in order not to test float16 at this stage (it will be tested in next step).

Then the adder module is instantiated and the random arguments and passed as a dictionary in standard Python style **args.

The line() function allows general Verilog code to be written directly to the generated Verilog module without parsing. Doing this might cause lint errors of unused or undriven signals, in that case the functions self.allow_unused() and self.allow_undriven() are required.

A behavioral fifo is generated using the self.tb.fifo() function. Input vectors are generated and pushed into a behavioral fifo and for each vector the sum is calculated and pushed into another fifo.

Every cycle valid is high an input vector is popped from the fifo and set on the adder's inputs.

Every cycle valid_out is high an output vector is popped from the fifo and compared to the adder's output. If there is a mismatch the test ends with an error.

If all input vectors are tested without error the test ends successfully.

Finally, a timeout is set and a dump is generated for debug.

Running a simulation


```
python3 p2v/tutorial/example1_adder/step_7/tb_adder.py -sim -seed 0 -gen 2
p2v-INFO: starting with seed 59463
p2v-INFO: starting gen iteration 0/1
p2v-DEBUG: created: adder__clk_bits16_num2_float16False.sv
p2v-DEBUG: created: adder__clk_bits16_num4_float16False.sv
p2v-DEBUG: created: adder__clk_bits16_num8_float16False.sv
p2v-DEBUG: created: adder__clk_bits16_num16_float16False.sv
p2v-DEBUG: created: _tb0.sv
p2v-INFO: verilog generation completed successfully (3 sec)
p2v-INFO: verilog lint completed successfully
p2v-INFO: verilog compilation completed successfully
p2v-INFO: verilog simulation completed successfully
p2v-INFO: starting gen iteration 1/1
p2v-DEBUG: created: adder__clk_bits25_num2_float16False.sv
p2v-DEBUG: created: _tb1.sv
p2v-INFO: verilog generation completed successfully (3 sec)
p2v-INFO: verilog lint completed successfully
p2v-INFO: verilog compilation completed successfully
p2v-INFO: verilog simulation completed successfully
p2v-INFO: completed successfully
```


Examining the simulation log

The command above ran 2 simulations with a random seed. The flag -sim activates the Verilog simulator, -seed 0 is a random seed and -gen 2 means 2 iterations.

The detailed simulation log can be viewed as such:

```
cat cache/p2v_sim.log
FST info: dumpfile dump.fst opened for output.
FST warning: $dumpvars: Unsupported argument type (vpiPackage)
742: test PASSED (successfully tested 32 additions)
```

Examining the coverage file

When using the -gen flag whether in simulation or not; a csv file will be generated in the output directory listing all the permutations that ran.

```
cat cache/adder.gen.csv
float16         , bits            , num
False           , 61              , 8
False           , 16              , 64
```

Examining the Verilog file


```
module tb ();

    // module parameters:
    // async_reset = True (bool): sync reset or async reset
    // size = 4 (int): number of inputs to test

    logic clk;
    logic resetn;

    initial
        forever begin
            clk = 0;
            #2;
            clk = 1;
            #3;
        end


    initial begin
        resetn = 1;
        repeat (5) @(negedge clk);  // async reset occurs not on posedge of clock
        resetn = 0;
        repeat (20) @(posedge clk);
```

```
        resetn = 1;
    end

    logic valid;
    logic [31:0] i0;
    initial i0 = 32'd0;

    logic [31:0] i1;
    initial i1 = 32'd0;

    logic [31:0] i2;
    initial i2 = 32'd0;

    logic [31:0] i3;
    initial i3 = 32'd0;

    logic [31:0] o;
    logic valid_out;
    adder__clk_bits32_num4_float16False adder (
        .clk(clk),  // input
        .resetn(resetn),  // input
        .valid(valid),  // input
        .i0(i0),  // input
        .i1(i1),  // input
        .i2(i2),  // input
        .i3(i3),  // input
        .o(o),  // output
        .valid_out(valid_out)  // output
    );

    logic en;
    initial en = 1'd0;

    always_ff @(posedge clk or negedge resetn)
        if (!resetn) valid <= 1'd0;
        else valid <= en;

    reg   [127:0] data_in_q [$];
    reg   [ 31:0] expected_q[$];
    logic [127:0] data_in;
    initial data_in = 128'd0;

    logic [31:0] expected;
    initial expected = 32'd0;
```



```
    initial begin

        data_in_q.push_back({32'hc386_bbc4, 32'h414c_343c, 32'h7311_d8a3,
32'ha6ce_cc1b});
        expected_q.push_back(32'h1eb3_94be);
        data_in_q.push_back({32'hc9e9_c616, 32'h1807_2e8c, 32'hd5f4_b3b2,
32'h7204_e52d});
        expected_q.push_back(32'h29ea_8d81);
        data_in_q.push_back({32'hf1fd_42a2, 32'he6c3_f339, 32'h07d4_bedc,
32'h8a9a_021e});
        expected_q.push_back(32'h6b2f_f6d5);
        data_in_q.push_back({32'h3bab_6c39, 32'h0580_5975, 32'ha46d_6753,
32'hdc25_74bd});
        expected_q.push_back(32'hc1be_a1be);

    end


    initial begin
        @(posedge resetn);
        repeat (10) @(posedge clk);
        en = 1;
    end

    // drive inputs
    always @(posedge clk)
        if (valid && (data_in_q.size() > 0)) begin
            data_in = data_in_q.pop_front();
            {i0, i1, i2, i3} = data_in;
        end

    // check output
    always @(posedge clk)
        if (valid_out) begin
            expected = expected_q.pop_front();
            if (o !== expected) begin
                $display("%0d: test FAILED (mismatch expected: 0x%0h, actual:
0x%0h)", $time,
                         expected, o);
                #10;
                $finish;
            end

            if (expected_q.size() == 0) begin
```

29

```
                $display("%0d: test PASSED (successfully tested 4 additions)", $time);
                #10;
                $finish;
            end

        end


    integer _count_clk = '0;
    always @(posedge clk) _count_clk <= _count_clk + 'd1;

    logic assert_never__reached_timeout_after_400_cycles_of_clk;
    assign assert_never__reached_timeout_after_400_cycles_of_clk = _count_clk >= 'd400;

    always @(posedge clk)
        if (resetn & assert_never__reached_timeout_after_400_cycles_of_clk)
            $fatal(0, "reached timeout after 400 cycles of clk");


    initial begin
        $dumpfile("dump.fst");
        $dumpvars;
        $dumpon;
    end


endmodule
```


In the Verilog file we can see how the clock and the async reset are generated. The random inputs and the expected outputs are random in Python but hard coded in the Verilog file.

STEP 8: SUPPORT FLOAT16 TESTING

In this lesson we will learn

- How to use a standard Python library for verification

```
import numpy as np # use numpy for float16 type

from p2v import p2v, misc, clock

import adder

class tb_adder(p2v):
    def module(self, async_reset=True, size=32):

        # same as previous example
        . . .

        args = adder.adder(self).gen_rand_args()
        num = args["num"]
        bits = args["bits"]
        float16 = args["float16"]

        # same as previous example
        . . .

        self.line(f"""
                    initial
                        begin
                  """)
        for i in range(size):
            input_vec = []
            input_sum = misc.cond(float16, np.float16(0), 0) # for float16 use 
numpy type
            for j in range(num):
                if float16:
                    val = np.float16(np.random.rand()) # use numpy random
                else:
                    val = self.tb.rand_int(1<<bits)
                input_sum += val
                if float16:
                    val = val.view(np.uint16) # convert to hex representation
```



```
                input_vec.append(misc.hex(val, bits))
            if float16:
                input_sum = input_sum.view(np.uint16) ) # convert to hex
representation
            self.line(f"data_in_q.push_back({misc.concat(input_vec)});")
            self.line(f"expected_q.push_back({misc.hex(input_sum, bits)});")
        self.line(f"""
                        end
                   """)

        # same as previous example
        . . .

        return self.write()
```

Example explained

In the example we show how the powerful numpy library is used to calculate the expected values of float16 additions.

To generate float16 numbers we use np.random.rand(). Numpy's random seed is set under-the-hood.

Similar to using numpy the immense Python library is at our disposal.

STEP 9: ADD RANDOM BEHAVIOR

In this lesson we will learn

- How to randomize behavior in simulation

```
import numpy as np

from p2v import p2v, misc, clock

import adder

class tb_adder(p2v):
    def module(self, async_reset=True, size=32):

        # same as previous example
        . . .

        self.logic("reset_released", initial=0)
        self.logic("en")
        self.tb.gen_en(clk, "en", max_duration=10, max_delay=10)

        # same as previous example
        . . .

        self.line(f"""
                    initial
                        begin
                            {misc.cond(async_reset, f"@(posedge {clk.rst_n});")}
                            repeat (10) @(posedge {clk});
                            reset_released = 1;
                        end

                        // drive inputs
                        always @(posedge {clk})
                            if (reset_released && en && (data_in_q.size() > 0))
                                begin
                                    data_in = data_in_q.pop_front();
                                    {misc.concat(inputs)} <= data_in;
                                    valid <= 1;
                                end
                            else
```



```
                                begin
                                    valid <= 0;
                                end

                        // check output
                        always @(posedge {clk})
                            if (valid_out)
                                begin
                                    expected = expected_q.pop_front();
                                    {self.tb.test_fail(condition=fail_condition,
message="mismatch expected: 0x%0h, actual: 0x%0h", params=["expected", "o"])}
                                    if (expected_q.size() == 0)
                                        {self.tb.test_pass(message=f"successfully
tested {size} additions")}
                                end
                   """)


        # same as previous example
        . . .

        return self.write()
```


Example explained

In this example we add random behavior on a signal named en. This en signal generates the valid signal, so effectively it generates random on valid.

Gen_en() uses 2 optional parameters: max_duration and max_delay. Max_duration is the maximum number of clock cycles of the signal staying constant. Max_delay is the maximum number of clock cycles before randomizing a change in the signal. This enables the creation of short pulses with long intervals.

SYNTAX

Module parameters

- Defining module parameters
- Asserting parameters
- How module parameters affect Verilog module name

PYTHON (SOURCE)

```
from p2v import p2v, clock, default_clk

class params(p2v):
    def module(self, clk=default_clk, bits=8, name="foo", sample=False, d={},
depth=128):
        self.set_param(clk, clock) # p2v clock
        self.set_param(bits, int, bits > 0) # integer parameter"
        self.set_param(name, str, name != "") # string parameter"
        self.set_param(sample, bool) # bool parameter - no constraint"
        self.set_param(d, dict, suffix="_".join(d.keys())) # dictionary parameter
- complex parameter does not create suffix automatcically"
        self.set_param(depth, int, suffix=None) # integer parameter - does not
affect module name"
        self.set_modname()

        return self.write()
```


VERILOG (GENERATED)

```
module params__bits8_namefoo_sampleFalse ();

    // module parameters:
    // clk = clk (p2v_clock)
    // bits = 8 (int)
    // name = "foo" (str)
    // sample = False (bool)
    // d = {} (dict)
    // depth = 128 (int)


endmodule
```

Module ports

- Defining inputs, inputs and inout ports

- Conditional ports
- Parametric ports
- Struct ports
- Verilog parametric ports

PYTHON (SOURCE)

```python
from p2v import p2v

num = 4
bits = 8
var = True

strct = {}
strct["ctrl"] = 8
strct["data"] = 32

class ports(p2v):
    def module(self):
        self.set_modname()

        self.input("a") # default is single bit
        self.input("b", 1) # same as the above
        self.input("c", 8) # multi bit bus
        self.input("dd", bits) # parametric width
        self.input("e", [bits]) # parametric width but forces [0:0] bus if width
is 1

        for n in range(num):
            self.input(f"f{n}", bits) # port in loop

        if var:
            self.input("g", bits*2) # conditional port

        self.output("ao") # default is single bit
        self.output("bo", 1) # same as the above
        self.output("co", 8) # multi bit bus
        self.output("ddo", bits) # parametric width
        self.output("eo", [bits]) # parametric width but forces [0:0] bus if
width is 1

        for n in range(num):
            self.output(f"f{n}o", bits) # port in loop
```



```
        if var:
            self.output("go", bits*2) # conditional port

        lst =  ["a", "b", "c", "dd", "e"]
        for n in range(num):
            lst.append(f"f{n}")
        if var:
            lst.append("g")
        for x in lst:
            self.assign(f"{x}o", x)


        self.inout("q") #inout ports width is always 1

        self.input("s", strct) # data struct as Python dictionary
        self.output("t", strct) # data struct as Python dictionary

        self.assign("t", "s")


        self.parameter("BITS", 32) # Verilog parameter

        self.input("z", "BITS") # Verilog parametric port
        self.output("zo", "BITS") # Verilog parametric port

        self.assign("zo", "z")

        return self.write()
```


VERILOG (GENERATED)

```
module ports #(
    parameter BITS = 32
) (
    input logic a,
    input logic b,
    input logic [7:0] c,
    input logic [7:0] dd,
    input logic [7:0] e,
    input logic [7:0] f0,
    input logic [7:0] f1,
    input logic [7:0] f2,
    input logic [7:0] f3,
```



```
    input logic [15:0] g,
    output logic ao,
    output logic bo,
    output logic [7:0] co,
    output logic [7:0] ddo,
    output logic [7:0] eo,
    output logic [7:0] f0o,
    output logic [7:0] f1o,
    output logic [7:0] f2o,
    output logic [7:0] f3o,
    output logic [15:0] go,
    inout q,
    input logic [7:0] s__ctrl,
    input logic [31:0] s__data,
    output logic [7:0] t__ctrl,
    output logic [31:0] t__data,
    input logic [BITS-1:0] z,
    output logic [BITS-1:0] zo
);

    assign ao = a;
    assign bo = b;
    assign co = c;
    assign ddo = dd;
    assign eo = e;
    assign f0o = f0;
    assign f1o = f1;
    assign f2o = f2;
    assign f3o = f3;
    assign go = g;

    assign t__ctrl = s__ctrl;
    assign t__data = s__data;

    assign zo = z;

endmodule
```


Module signals

- Defining variables, scalar and bus
- Conditional and parameter varibales
- Defining module and local Verilog parameters
- Assigning signals
- Using structs
- Assigning struct fields

PYTHON (SOURCE)

```
from p2v import p2v, misc, clock, default_clk

num = 4
bits = 8
var = True

strct = {}
strct["ctrl"] = 8
strct["data"] = 32

strct_handshake = {}
strct_handshake["ctrl"] = 8
strct_handshake["data"] = 32
strct_handshake["valid"] = 1.0 # value reserved to mark qualifier
strct_handshake["ready"] = -1.0 # value reserved to mark back pressure


class signals(p2v):
    def module(self):
        self.set_modname()

        self.logic("a") # default is single bit
        self.logic("b", 1) # same as the above
        self.logic("c", 8) # multi bit bus
        self.logic("d", bits) # parametric width
        self.logic("e", [bits]) # parametric width but forces [0:0] bus if width
is 1

        for n in range(num):
            self.logic(f"f{n}", bits) # port in loop

        if var:
```



```
            self.logic("g", bits*2) # conditional port
            self.logic("h", bits*2) # conditional port


        clk = default_clk
        clk2 = clock("clk2", rst_n="clk2_rstn")
        self.logic(clk) # p2v clock
        self.logic(clk2) # p2v clock

        self.assign(clk.name, "1'b1") # clock assignment
        self.assign(clk.rst_n, "1'b1") # reset assignment

        self.parameter("BITS", 32) # Verilog parameter

        self.logic("z", "BITS", assign="'0") # Verilog parametric port
        self.allow_unused("z")

        self.parameter("IDLE", "2'd0", local=True) # local parameter
        self.logic("iii", 2, assign="IDLE")
        self.allow_unused("iii")

        self.line() # insert empty line to Verilog file
        self.assign("b", "1'b1") # assign to const
        self.assign("e", misc.dec(3, bits)) # assign to const
        for n in range(num):
            self.assign(f"f{n}", "d | e") # assign expression
        self.assign("a", "b") # trivial Verilog assign
        self.assign("c", 0) # assign to const
        self.assign("d", f"e + {misc.dec(1, bits)}") # assign expression
        self.assign("g", misc.concat(["f0", "f1"])) # assign conctenation
        self.assign(misc.bits("h", bits), "f2") # partial bits
        self.assign(misc.bits("h", bits, start=bits), "f3") # partial bits

        self.line() # insert empty line to Verilog file
        self.assign(clk2.rst_n, "1'b1")
        self.assign(clk2, clk)

        self.allow_unused([clk2, clk.rst_n])

        self.logic("aa", 8, assign=misc.hex(-1, 8)) # inline assignment
        self.logic("bb", 8, initial=misc.hex(-1, 8)) # inline initial assignment
        self.allow_unused(["aa", "bb"])

        # struct assignment
        self.line() # insert empty line to Verilog file
```



```python
        self.logic("s", strct) # data struct as Python dictionary
        self.logic("t", strct) # data struct as Python dictionary
        self.assign("t", "s") # struct assignment
        self.allow_undriven("s")

        # struct assignment with field change
        s1 = self.logic("s1", strct) # data struct as Python dictionary
        t1 = self.logic("t1", strct) # data struct as Python dictionary
        self.assign(t1["ctrl"], "d") # struct assignment
        self.assign("t1", "s1") # struct assignment
        self.allow_undriven("s1")

        # struct assignment with control
        self.line() # insert empty line to Verilog file
        s2 = self.logic("s2", strct_handshake) # data struct as Python dictionary
        t2 = self.logic("t2", strct_handshake) # data struct as Python dictionary
        self.assign("t2", "s2") # struct assignment (ready assignment will be
reversed: s2.ready = t2.ready)
        self.allow_undriven(["s2", t2["ready"]])

        self.allow_unused(["t", "t1", "t2", s1["ctrl"]])


        self.allow_unused(["a", "b", "c", "d", "e"])
        for n in range(num):
            self.allow_unused(f"f{n}")

        if var:
            self.allow_unused(["g", "h"])

        return self.write()
```


VERILOG (GENERATED)

```verilog
module signals #(
    parameter BITS = 32
) ();

    logic a;
    logic b;
    logic [7:0] c;
    logic [7:0] d;
    logic [7:0] e;
```



```
    logic [7:0] f0;
    logic [7:0] f1;
    logic [7:0] f2;
    logic [7:0] f3;
    logic [15:0] g;
    logic [15:0] h;
    logic clk;
    logic rst_n;
    logic clk2;
    logic clk2_rstn;
    assign clk   = 1'b1;
    assign rst_n = 1'b1;
    logic [BITS-1:0] z;
    assign z = '0;

    localparam IDLE = 2'd0;
    logic [1:0] iii;
    assign iii = IDLE;


    assign b = 1'b1;
    assign e = 8'd3;
    assign f0 = d | e;
    assign f1 = d | e;
    assign f2 = d | e;
    assign f3 = d | e;
    assign a = b;
    assign c = 8'd0;
    assign d = e + 8'd1;
    assign g = {f0, f1};
    assign h[7:0] = f2;
    assign h[15:8] = f3;

    assign clk2_rstn = 1'b1;

    assign clk2 = clk;
    logic [7:0] aa;
    assign aa = 8'hff;

    logic [7:0] bb;
    initial bb = 8'hff;


    logic [ 7:0] s__ctrl;
    logic [31:0] s__data;
```



```
    logic [ 7:0] t__ctrl;
    logic [31:0] t__data;

    assign t__ctrl = s__ctrl;
    assign t__data = s__data;

    logic [ 7:0] s1__ctrl;
    logic [31:0] s1__data;
    logic [ 7:0] t1__ctrl;
    logic [31:0] t1__data;
    assign t1__ctrl = d;

    assign t1__data = s1__data;


    logic [7:0] s2__ctrl;
    logic [31:0] s2__data;
    logic s2__valid;
    logic s2__ready;
    logic [7:0] t2__ctrl;
    logic [31:0] t2__data;
    logic t2__valid;
    logic t2__ready;

    assign t2__ctrl  = s2__ctrl;
    assign t2__data  = s2__data;
    assign t2__valid = s2__valid;
    assign s2__ready = t2__ready;


endmodule
```


Instances

- Creating Son instances and connection their ports
- Passing parameters and Verilog parameters to instance
- Setting instance name
- Auto connecting ports

PYTHON (SOURCE)

```
from p2v import p2v, misc

import _or_gate

class instances(p2v):
    def module(self, num=4, bits=32):
        self.set_param(num, int, 0 < num < 8)
        self.set_param(bits, int, bits > 0)
        self.set_modname()

        for n in range(num):
            self.input(f"a{n}", bits+n)
            self.input(f"b{n}", bits+n)
            self.output(f"c{n}", bits+n)

            son = _or_gate._or_gate(self).module(bits=bits+n) # creates son
module
            son.connect_in("a", f"a{n}")
            son.connect_in("b", f"b{n}")
            son.connect_out("c", f"c{n}")
            son.inst(suffix=n) # make instance name unique


        self.input(["a", "b"], 16)
        self.output("c", 16)
        son = _or_gate._or_gate(self).module(bits=16)
        son.connect_in("a") # assumes wire name equals port name
        son.connect_in("b") # assumes wire name equals port name
        son.connect_out("c") # assumes wire name equals port name
        son.inst("my_or_gate") # specific instance name


        self.output("ca", 16)
        son = _or_gate._or_gate(self).module(bits=16)
```



```
        son.connect_out("c", "ca")
        son.connect_auto() # trivially connects all missing ports (wire name equals port name)
        son.inst("my_auto_connect_or") # specific instance name


        son = _or_gate._or_gate(self).module(bits=16)
        son.connect_auto(ports=True, suffix="_01") #
        son.inst("my_auto_connect_ports_or") # specific instance name


        # Verilog instance
        self.input(["aa", "bb"], bits)
        self.output("cc", bits)
        son = self.verilog_module("_and_gate", params={"BITS":bits}) # setting instance Verilog parameter
        son.connect_in("a", "aa") # connecting Verilog is same as connecting a p2v instance
        son.connect_in("b", "bb") # connecting Verilog is same as connecting a p2v instance
        son.connect_out("c", "cc") # connecting Verilog is same as connecting a p2v instance
        son.inst() # instance name equals module name

        return self.write()
```

VERILOG (GENERATED)

```
module instances__num4_bits32 (
    input  logic [31:0] a0,
    input  logic [31:0] b0,
    output logic [31:0] c0,
    input  logic [32:0] a1,
    input  logic [32:0] b1,
    output logic [32:0] c1,
    input  logic [33:0] a2,
    input  logic [33:0] b2,
    output logic [33:0] c2,
    input  logic [34:0] a3,
    input  logic [34:0] b3,
    output logic [34:0] c3,
    input  logic [15:0] a,
    input  logic [15:0] b,
```

```
    output logic [15:0] c,
    output logic [15:0] ca,
    input  logic [15:0] a_01,
    input  logic [15:0] b_01,
    output logic [15:0] c_01,
    input  logic [31:0] aa,
    input  logic [31:0] bb,
    output logic [31:0] cc
);

    // module parameters:
    // num = 4 (int)
    // bits = 32 (int)

    _or_gate__bits32 _or_gate0 (
        .a(a0),  // input
        .b(b0),  // input
        .c(c0)   // output
    );

    _or_gate__bits33 _or_gate1 (
        .a(a1),  // input
        .b(b1),  // input
        .c(c1)   // output
    );

    _or_gate__bits34 _or_gate2 (
        .a(a2),  // input
        .b(b2),  // input
        .c(c2)   // output
    );

    _or_gate__bits35 _or_gate3 (
        .a(a3),  // input
        .b(b3),  // input
        .c(c3)   // output
    );

    _or_gate__bits16 my_or_gate (
        .a(a),  // input
        .b(b),  // input
        .c(c)   // output
    );

    _or_gate__bits16 my_auto_connect_or (
```



```
        .c(ca),  // output
        .a(a),   // input
        .b(b)    // input
    );

    _or_gate__bits16 my_auto_connect_ports_or (
        .a(a_01),  // input
        .b(b_01),  // input
        .c(c_01)   // output
    );

    _and_gate #(
        .BITS(32)
    ) _and_gate (
        .a(aa),  // input
        .b(bb),  // input
        .c(cc)   // output
    );


endmodule
```


Clocks

- Defining clocks with synchronous and asynchronous resets
- Clock ports
- Using clock for sampling signals
- Default clock

PYTHON (SOURCE)

```python
from p2v import p2v, clock, default_clk

class clocks(p2v):
    def module(self, clk=default_clk):
        self.set_param(clk, clock) # verifies that clk if a p2v clock
        self.set_modname()

        clks = [clk]
        clks.append(clock("clk0")) # clock without reset
        clks.append(clock("clk1", rst_n="clk1_rst_n")) # clk with async reset
        clks.append(clock("clk2", reset="clk2_reset")) # clk with sync reset
        clks.append(clock("clk3", rst_n="clk3_rst_n", reset="clk3_reset")) # clk
with both asycn and sync reset

        num = len(clks)
        for n in range(num):
            self.input(clks[n])
            self.input(f"i{n}", 32)
            self.output(f"o{n}", 32)

            self.sample(clks[n], f"o{n}", f"i{n}")

        return self.write()
```


VERILOG (GENERATED)

```verilog
module clocks (
    input logic clk,
    input logic rst_n,
    input logic [31:0] i0,
    output logic [31:0] o0,
    input logic clk0,
```

```
    input logic [31:0] i1,
    output logic [31:0] o1,
    input logic clk1,
    input logic clk1_rst_n,
    input logic [31:0] i2,
    output logic [31:0] o2,
    input logic clk2,
    input logic clk2_reset,
    input logic [31:0] i3,
    output logic [31:0] o3,
    input logic clk3,
    input logic clk3_rst_n,
    input logic clk3_reset,
    input logic [31:0] i4,
    output logic [31:0] o4
);

    // module parameters:
    // clk = clk (p2v_clock)

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) o0 <= 32'd0;
        else o0 <= i0;

    always_ff @(posedge clk0) o1 <= i1;

    always_ff @(posedge clk1 or negedge clk1_rst_n)
        if (!clk1_rst_n) o2 <= 32'd0;
        else o2 <= i2;

    always_ff @(posedge clk2)
        if (clk2_reset) o3 <= 32'd0;
        else o3 <= i3;

    always_ff @(posedge clk3 or negedge clk3_rst_n)
        if (!clk3_rst_n) o4 <= 32'd0;
        else if (clk3_reset) o4 <= 32'd0;
        else o4 <= i4;


endmodule
```


Sampling (using FFs)

- Sampling signals (creating FFs)
- Using qualifiers (valid signal)
- Setting default value
- Sampling structs

PYTHON (SOURCE)

```python
from p2v import p2v, misc, clock, default_clk

bits = 8

strct_handshake = {}
strct_handshake["ctrl"] = 8
strct_handshake["data"] = 32
strct_handshake["valid"] = 1.0 # value reserevd to mark qualifier
strct_handshake["ready"] = -1.0 # value reserevd to mark back pressure


class samples(p2v):
    def module(self):
        self.set_modname()

        clk0 = clock("clk0", rst_n="clk0_rst_n") # clk with async reset
        clk1 = clock("clk1", reset="clk1_reset") # clk with sync reset
        self.input(clk0)
        self.input(clk1)

        self.input("valid")
        self.input("i0", bits)
        self.output("x0", bits)
        self.output("x1", bits)
        self.output("x2", bits)
        self.output("x3", bits)

        s = self.input("s", strct_handshake)
        t = self.output("t", strct_handshake)

        self.sample(clk0, "x0", "i0") # free running clock - async reset

        self.sample(clk1, "x1", "i0") # free running clock - sync reset
```



```
        self.sample(clk0, "x2", "i0", valid="valid") # sample with qualifier

        self.sample(clk0, "x3", "i0", valid="valid", reset_val=-1) # sample with
non zero reset value


        self.sample(clk1, t["ctrl"], f"{s['ctrl']} | 8'h4", valid="valid")
        self.sample(clk1, "t", "s") # sample structs


        return self.write()
```

VERILOG (GENERATED)

```
module samples (
    input logic clk0,
    input logic clk0_rst_n,
    input logic clk1,
    input logic clk1_reset,
    input logic valid,
    input logic [7:0] i0,
    output logic [7:0] x0,
    output logic [7:0] x1,
    output logic [7:0] x2,
    output logic [7:0] x3,
    input logic [7:0] s__ctrl,
    input logic [31:0] s__data,
    input logic s__valid,
    output logic s__ready,
    output logic [7:0] t__ctrl,
    output logic [31:0] t__data,
    output logic t__valid,
    input logic t__ready
);

    always_ff @(posedge clk0 or negedge clk0_rst_n)
        if (!clk0_rst_n) x0 <= 8'd0;
        else x0 <= i0;

    always_ff @(posedge clk1)
        if (clk1_reset) x1 <= 8'd0;
        else x1 <= i0;
```



```
    always_ff @(posedge clk0 or negedge clk0_rst_n)
        if (!clk0_rst_n) x2 <= 8'd0;
        else if (valid) x2 <= i0;

    always_ff @(posedge clk0 or negedge clk0_rst_n)
        if (!clk0_rst_n) x3 <= {8{1'b1}};
        else if (valid) x3 <= i0;

    always_ff @(posedge clk1)
        if (clk1_reset) t__ctrl <= 8'd0;
        else if (valid) t__ctrl <= s__ctrl | 8'h4;


    assign s__ready = t__ready;
    always_ff @(posedge clk1)
        if (clk1_reset) t__valid <= 1'd0;
        else if (~t__valid | ~s__ready) t__valid <= s__valid;



    always_ff @(posedge clk1)
        if (clk1_reset) t__data <= 32'd0;
        else if (s__valid & s__ready) t__data <= s__data;



endmodule
```


Structs

- Basic strcuts
- Struct ports
- Assigning and sampling structs
- Changing a struct’s field
- Casting similar structs
- Multi-hierarchy structs (axi)
- Strct control signals

DEFINING A STRUCT

P2V structs are native Python dictionaries.

A P2V struct has data fields and two optional control fields a valid (qualifier) and ready (back pressure). All data fields should be of the same direction either positive or negative. The control signals use reserved float values of 1.0 for valid and -1.0 for ready.

A P2V struct can contain a mixture of bidirectional signals (input and outputs) but those must be set as sub-hierarchies (nested dictionaries).

STRUCT EXAMPLE: AXI BUS

```
def axi_a(id_bits=4, addr_bits=32, burst_bits=2, len_bits=8, size_bits=3,
cache_bits=4, lock_bits=1, prot_bits=3):
    fields = {}
    fields["id"] = id_bits
    fields["addr"] = addr_bits
    fields["burst"] = burst_bits
    fields["len"] = len_bits
    fields["size"] = size_bits
    fields["cache"] = cache_bits
    fields["lock"] = lock_bits
    fields["prot"] = prot_bits
    fields["valid"] = 1.0
    fields["ready"] = -1.0
    return fields


def axi_w(data_bits=512):
    fields = {}
    fields["data"] = data_bits
    fields["strb"] = data_bits // 8
```



```
    fields["last"] = 1
    fields["valid"] = 1.0
    fields["ready"] = -1.0
    return fields

def axi_b(id_bits=4, resp_bits=2):
    fields = {}
    fields["id"] = -id_bits
    fields["resp"] = -resp_bits
    fields["valid"] = -1.0
    fields["ready"] = 1.0
    return fields

def axi_r(id_bits=4, resp_bits=2, data_bits=512):
    fields = {}
    fields["data"] = -data_bits
    fields["id"] = -id_bits
    fields["resp"] = -resp_bits
    fields["last"] = -1
    fields["valid"] = -1.0
    fields["ready"] = 1.0
    return fields


def axi(id_bits=4, addr_bits=32, data_bits=512, burst_bits=2, len_bits=8,
size_bits=3, cache_bits=4, lock_bits=1, prot_bits=3, resp_bits=2):
    fields = {}
    for x in ["w", "r"]:
        fields[f"a{x}"] = axi_a(id_bits=id_bits, addr_bits=addr_bits,
burst_bits=burst_bits, len_bits=len_bits, size_bits=size_bits,
cache_bits=cache_bits, lock_bits=lock_bits, prot_bits=prot_bits)
    fields["w"] = axi_w(data_bits=data_bits)
    fields["b"] = axi_b(id_bits=id_bits, resp_bits=resp_bits)
    fields["r"] = axi_r(id_bits=id_bits, resp_bits=resp_bits,
data_bits=data_bits)
    return fields
```


The write bus is defined as positive field widths, so the read (and write response bus) bus is defined with negative fields widths.

Defining a struct with nested hierarchies gives freedom to later access the entire struct or a nested part. For example, we can define the entire AXI bus as an input or only the sub write bus.

PYTHON (SOURCE)

```
from p2v import p2v, misc, clock, default_clk

import axi

from copy import deepcopy

# basic struct with 2 data fields
basic = {}
basic["a"] = 8
basic["b"] = 4

# inherit basic struct and add field
basic_with_c = deepcopy(basic)
basic_with_c["c"] = 2


class structs(p2v):
    def module(self, clk=default_clk, addr_bits=32, data_bits=512):
        self.set_param(clk, clock)
        self.set_param(addr_bits, int, addr_bits > 0)
        self.set_param(data_bits, int, data_bits > 0 and misc.is_pow2(data_bits))
        self.set_modname()

        axi_strct = axi.axi(addr_bits=addr_bits, data_bits=data_bits,
cache_bits=0, lock_bits=0, prot_bits=0)
        self.input(clk)


        # async assignment - full axi struct
        mstr0 = self.input("master0", axi_strct) # axi input port
        slv0 = self.output("slave0", axi_strct) # axi output port

        self.assign("slave0", "master0") # assign axi structs with change of
write address


        # async assignment - full axi struct with field change
        self.input("write_addr", addr_bits)
        mstr1 = self.input("master1", axi_strct) # axi input port
        slv1 = self.output("slave1", axi_strct) # axi output port
```



```
        self.assign(slv1["aw"]["addr"], "write_addr") # assign awaddr field
        self.assign("slave1", "master1") # assign axi structs with change of
write address
        self.allow_unused(mstr1["aw"]["addr"]) # don't give lint error on unused
master awaddr


        # async assignment - sub structs one by one
        for x in ["aw", "w", "b", "ar", "r"]:
            self.input(f"master2_{x}", axi_strct[x]) # partial axi input port
            self.output(f"slave2_{x}", axi_strct[x]) # partial axi output port

            self.assign(f"slave2_{x}", f"master2_{x}")


        # sync assignment - sub structs one by one
        for x in axi_strct: # same as ["aw", "w", "b", "ar", "r"]:
            self.input(f"master3_{x}", axi_strct[x]) # partial axi input port
            self.output(f"slave3_{x}", axi_strct[x]) # partial axi output port

            self.sample(clk, f"slave3_{x}", f"master3_{x}")


        # basic struct
        self.input("bi", basic)
        self.output("bo", basic)
        self.assign("bo", "bi")


        # basic struct with additonla field c
        self.input("bci", basic_with_c)
        self.output("bco", basic_with_c)
        self.assign("bco", "bci")


        # casting between basic and basic_with_c
        cast_o = self.output("cast_o", basic_with_c)
        self.assign("cast_o", "bi")
        self.assign(cast_o["c"], "2'd2")

        return self.write()
```


VERILOG (GENERATED)

```
module structs__addr_bits32_data_bits512 (
    input logic clk,
    input logic rst_n,
    input logic [3:0] master0__awid,
    input logic [31:0] master0__awaddr,
    input logic [1:0] master0__awburst,
    input logic [7:0] master0__awlen,
    input logic [2:0] master0__awsize,
    input logic master0__awvalid,
    output logic master0__awready,
    input logic [3:0] master0__arid,
    input logic [31:0] master0__araddr,
    input logic [1:0] master0__arburst,
    input logic [7:0] master0__arlen,
    input logic [2:0] master0__arsize,
    input logic master0__arvalid,
    output logic master0__arready,
    input logic [511:0] master0__wdata,
    input logic [63:0] master0__wstrb,
    input logic master0__wlast,
    input logic master0__wvalid,
    output logic master0__wready,
    output logic [3:0] master0__bid,
    output logic [1:0] master0__bresp,
    input logic master0__bready,
    output logic master0__bvalid,
    output logic [511:0] master0__rdata,
    output logic [3:0] master0__rid,
    output logic [1:0] master0__rresp,
    output logic master0__rlast,
    input logic master0__rready,
    output logic master0__rvalid,
    output logic [3:0] slave0__awid,
    output logic [31:0] slave0__awaddr,
    output logic [1:0] slave0__awburst,
    output logic [7:0] slave0__awlen,
    output logic [2:0] slave0__awsize,
    output logic slave0__awvalid,
    input logic slave0__awready,
    output logic [3:0] slave0__arid,
    output logic [31:0] slave0__araddr,
    output logic [1:0] slave0__arburst,
    output logic [7:0] slave0__arlen,
    output logic [2:0] slave0__arsize,
```



```
    output logic slave0__arvalid,
    input logic slave0__arready,
    output logic [511:0] slave0__wdata,
    output logic [63:0] slave0__wstrb,
    output logic slave0__wlast,
    output logic slave0__wvalid,
    input logic slave0__wready,
    input logic [3:0] slave0__bid,
    input logic [1:0] slave0__bresp,
    output logic slave0__bready,
    input logic slave0__bvalid,
    input logic [511:0] slave0__rdata,
    input logic [3:0] slave0__rid,
    input logic [1:0] slave0__rresp,
    input logic slave0__rlast,
    output logic slave0__rready,
    input logic slave0__rvalid,
    input logic [31:0] write_addr,
    input logic [3:0] master1__awid,
    input logic [31:0] master1__awaddr,
    input logic [1:0] master1__awburst,
    input logic [7:0] master1__awlen,
    input logic [2:0] master1__awsize,
    input logic master1__awvalid,
    output logic master1__awready,
    input logic [3:0] master1__arid,
    input logic [31:0] master1__araddr,
    input logic [1:0] master1__arburst,
    input logic [7:0] master1__arlen,
    input logic [2:0] master1__arsize,
    input logic master1__arvalid,
    output logic master1__arready,
    input logic [511:0] master1__wdata,
    input logic [63:0] master1__wstrb,
    input logic master1__wlast,
    input logic master1__wvalid,
    output logic master1__wready,
    output logic [3:0] master1__bid,
    output logic [1:0] master1__bresp,
    input logic master1__bready,
    output logic master1__bvalid,
    output logic [511:0] master1__rdata,
    output logic [3:0] master1__rid,
    output logic [1:0] master1__rresp,
    output logic master1__rlast,
```



```
    input logic master1__rready,
    output logic master1__rvalid,
    output logic [3:0] slave1__awid,
    output logic [31:0] slave1__awaddr,
    output logic [1:0] slave1__awburst,
    output logic [7:0] slave1__awlen,
    output logic [2:0] slave1__awsize,
    output logic slave1__awvalid,
    input logic slave1__awready,
    output logic [3:0] slave1__arid,
    output logic [31:0] slave1__araddr,
    output logic [1:0] slave1__arburst,
    output logic [7:0] slave1__arlen,
    output logic [2:0] slave1__arsize,
    output logic slave1__arvalid,
    input logic slave1__arready,
    output logic [511:0] slave1__wdata,
    output logic [63:0] slave1__wstrb,
    output logic slave1__wlast,
    output logic slave1__wvalid,
    input logic slave1__wready,
    input logic [3:0] slave1__bid,
    input logic [1:0] slave1__bresp,
    output logic slave1__bready,
    input logic slave1__bvalid,
    input logic [511:0] slave1__rdata,
    input logic [3:0] slave1__rid,
    input logic [1:0] slave1__rresp,
    input logic slave1__rlast,
    output logic slave1__rready,
    input logic slave1__rvalid,
    input logic [3:0] master2_aw__id,
    input logic [31:0] master2_aw__addr,
    input logic [1:0] master2_aw__burst,
    input logic [7:0] master2_aw__len,
    input logic [2:0] master2_aw__size,
    input logic master2_aw__valid,
    output logic master2_aw__ready,
    output logic [3:0] slave2_aw__id,
    output logic [31:0] slave2_aw__addr,
    output logic [1:0] slave2_aw__burst,
    output logic [7:0] slave2_aw__len,
    output logic [2:0] slave2_aw__size,
    output logic slave2_aw__valid,
    input logic slave2_aw__ready,
```



```
    input logic [511:0] master2_w__data,
    input logic [63:0] master2_w__strb,
    input logic master2_w__last,
    input logic master2_w__valid,
    output logic master2_w__ready,
    output logic [511:0] slave2_w__data,
    output logic [63:0] slave2_w__strb,
    output logic slave2_w__last,
    output logic slave2_w__valid,
    input logic slave2_w__ready,
    output logic [3:0] master2_b__id,
    output logic [1:0] master2_b__resp,
    input logic master2_b__ready,
    output logic master2_b__valid,
    input logic [3:0] slave2_b__id,
    input logic [1:0] slave2_b__resp,
    output logic slave2_b__ready,
    input logic slave2_b__valid,
    input logic [3:0] master2_ar__id,
    input logic [31:0] master2_ar__addr,
    input logic [1:0] master2_ar__burst,
    input logic [7:0] master2_ar__len,
    input logic [2:0] master2_ar__size,
    input logic master2_ar__valid,
    output logic master2_ar__ready,
    output logic [3:0] slave2_ar__id,
    output logic [31:0] slave2_ar__addr,
    output logic [1:0] slave2_ar__burst,
    output logic [7:0] slave2_ar__len,
    output logic [2:0] slave2_ar__size,
    output logic slave2_ar__valid,
    input logic slave2_ar__ready,
    output logic [511:0] master2_r__data,
    output logic [3:0] master2_r__id,
    output logic [1:0] master2_r__resp,
    output logic master2_r__last,
    input logic master2_r__ready,
    output logic master2_r__valid,
    input logic [511:0] slave2_r__data,
    input logic [3:0] slave2_r__id,
    input logic [1:0] slave2_r__resp,
    input logic slave2_r__last,
    output logic slave2_r__ready,
    input logic slave2_r__valid,
    input logic [3:0] master3_aw__id,
```



```
    input logic [31:0] master3_aw__addr,
    input logic [1:0] master3_aw__burst,
    input logic [7:0] master3_aw__len,
    input logic [2:0] master3_aw__size,
    input logic master3_aw__valid,
    output logic master3_aw__ready,
    output logic [3:0] slave3_aw__id,
    output logic [31:0] slave3_aw__addr,
    output logic [1:0] slave3_aw__burst,
    output logic [7:0] slave3_aw__len,
    output logic [2:0] slave3_aw__size,
    output logic slave3_aw__valid,
    input logic slave3_aw__ready,
    input logic [3:0] master3_ar__id,
    input logic [31:0] master3_ar__addr,
    input logic [1:0] master3_ar__burst,
    input logic [7:0] master3_ar__len,
    input logic [2:0] master3_ar__size,
    input logic master3_ar__valid,
    output logic master3_ar__ready,
    output logic [3:0] slave3_ar__id,
    output logic [31:0] slave3_ar__addr,
    output logic [1:0] slave3_ar__burst,
    output logic [7:0] slave3_ar__len,
    output logic [2:0] slave3_ar__size,
    output logic slave3_ar__valid,
    input logic slave3_ar__ready,
    input logic [511:0] master3_w__data,
    input logic [63:0] master3_w__strb,
    input logic master3_w__last,
    input logic master3_w__valid,
    output logic master3_w__ready,
    output logic [511:0] slave3_w__data,
    output logic [63:0] slave3_w__strb,
    output logic slave3_w__last,
    output logic slave3_w__valid,
    input logic slave3_w__ready,
    output logic [3:0] master3_b__id,
    output logic [1:0] master3_b__resp,
    input logic master3_b__ready,
    output logic master3_b__valid,
    input logic [3:0] slave3_b__id,
    input logic [1:0] slave3_b__resp,
    output logic slave3_b__ready,
    input logic slave3_b__valid,
```



```
    output logic [511:0] master3_r__data,
    output logic [3:0] master3_r__id,
    output logic [1:0] master3_r__resp,
    output logic master3_r__last,
    input logic master3_r__ready,
    output logic master3_r__valid,
    input logic [511:0] slave3_r__data,
    input logic [3:0] slave3_r__id,
    input logic [1:0] slave3_r__resp,
    input logic slave3_r__last,
    output logic slave3_r__ready,
    input logic slave3_r__valid,
    input logic [7:0] bi__a,
    input logic [3:0] bi__b,
    output logic [7:0] bo__a,
    output logic [3:0] bo__b,
    input logic [7:0] bci__a,
    input logic [3:0] bci__b,
    input logic [1:0] bci__c,
    output logic [7:0] bco__a,
    output logic [3:0] bco__b,
    output logic [1:0] bco__c,
    output logic [7:0] cast_o__a,
    output logic [3:0] cast_o__b,
    output logic [1:0] cast_o__c
);

    // module parameters:
    // clk = clk (p2v_clock)
    // addr_bits = 32 (int)
    // data_bits = 512 (int)


    assign slave0__awid = master0__awid;
    assign slave0__awaddr = master0__awaddr;
    assign slave0__awburst = master0__awburst;
    assign slave0__awlen = master0__awlen;
    assign slave0__awsize = master0__awsize;
    assign slave0__awvalid = master0__awvalid;
    assign master0__awready = slave0__awready;
    assign slave0__arid = master0__arid;
    assign slave0__araddr = master0__araddr;
    assign slave0__arburst = master0__arburst;
    assign slave0__arlen = master0__arlen;
    assign slave0__arsize = master0__arsize;
```



```
    assign slave0__arvalid = master0__arvalid;
    assign master0__arready = slave0__arready;
    assign slave0__wdata = master0__wdata;
    assign slave0__wstrb = master0__wstrb;
    assign slave0__wlast = master0__wlast;
    assign slave0__wvalid = master0__wvalid;
    assign master0__wready = slave0__wready;
    assign master0__bid = slave0__bid;
    assign master0__bresp = slave0__bresp;
    assign slave0__bready = master0__bready;
    assign master0__bvalid = slave0__bvalid;
    assign master0__rdata = slave0__rdata;
    assign master0__rid = slave0__rid;
    assign master0__rresp = slave0__rresp;
    assign master0__rlast = slave0__rlast;
    assign slave0__rready = master0__rready;
    assign master0__rvalid = slave0__rvalid;

    assign slave1__awaddr = write_addr;

    assign slave1__awid = master1__awid;
    assign slave1__awburst = master1__awburst;
    assign slave1__awlen = master1__awlen;
    assign slave1__awsize = master1__awsize;
    assign slave1__awvalid = master1__awvalid;
    assign master1__awready = slave1__awready;
    assign slave1__arid = master1__arid;
    assign slave1__araddr = master1__araddr;
    assign slave1__arburst = master1__arburst;
    assign slave1__arlen = master1__arlen;
    assign slave1__arsize = master1__arsize;
    assign slave1__arvalid = master1__arvalid;
    assign master1__arready = slave1__arready;
    assign slave1__wdata = master1__wdata;
    assign slave1__wstrb = master1__wstrb;
    assign slave1__wlast = master1__wlast;
    assign slave1__wvalid = master1__wvalid;
    assign master1__wready = slave1__wready;
    assign master1__bid = slave1__bid;
    assign master1__bresp = slave1__bresp;
    assign slave1__bready = master1__bready;
    assign master1__bvalid = slave1__bvalid;
    assign master1__rdata = slave1__rdata;
    assign master1__rid = slave1__rid;
    assign master1__rresp = slave1__rresp;
```



```
    assign master1__rlast = slave1__rlast;
    assign slave1__rready = master1__rready;
    assign master1__rvalid = slave1__rvalid;


    assign slave2_aw__id = master2_aw__id;
    assign slave2_aw__addr = master2_aw__addr;
    assign slave2_aw__burst = master2_aw__burst;
    assign slave2_aw__len = master2_aw__len;
    assign slave2_aw__size = master2_aw__size;
    assign slave2_aw__valid = master2_aw__valid;
    assign master2_aw__ready = slave2_aw__ready;


    assign slave2_w__data = master2_w__data;
    assign slave2_w__strb = master2_w__strb;
    assign slave2_w__last = master2_w__last;
    assign slave2_w__valid = master2_w__valid;
    assign master2_w__ready = slave2_w__ready;


    assign master2_b__id = slave2_b__id;
    assign master2_b__resp = slave2_b__resp;
    assign slave2_b__ready = master2_b__ready;
    assign master2_b__valid = slave2_b__valid;


    assign slave2_ar__id = master2_ar__id;
    assign slave2_ar__addr = master2_ar__addr;
    assign slave2_ar__burst = master2_ar__burst;
    assign slave2_ar__len = master2_ar__len;
    assign slave2_ar__size = master2_ar__size;
    assign slave2_ar__valid = master2_ar__valid;
    assign master2_ar__ready = slave2_ar__ready;


    assign master2_r__data = slave2_r__data;
    assign master2_r__id = slave2_r__id;
    assign master2_r__resp = slave2_r__resp;
    assign master2_r__last = slave2_r__last;
    assign slave2_r__ready = master2_r__ready;
    assign master2_r__valid = slave2_r__valid;


    assign master3_aw__ready = slave3_aw__ready;
    always_ff @(posedge clk or negedge rst_n)
```



```
        if (!rst_n) slave3_aw__valid <= 1'd0;
        else if (~slave3_aw__valid | ~master3_aw__ready) slave3_aw__valid <=
master3_aw__valid;



    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_aw__id <= 4'd0;
        else if (master3_aw__valid & master3_aw__ready) slave3_aw__id <=
master3_aw__id;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_aw__addr <= 32'd0;
        else if (master3_aw__valid & master3_aw__ready) slave3_aw__addr <=
master3_aw__addr;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_aw__burst <= 2'd0;
        else if (master3_aw__valid & master3_aw__ready) slave3_aw__burst <=
master3_aw__burst;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_aw__len <= 8'd0;
        else if (master3_aw__valid & master3_aw__ready) slave3_aw__len <=
master3_aw__len;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_aw__size <= 3'd0;
        else if (master3_aw__valid & master3_aw__ready) slave3_aw__size <=
master3_aw__size;



    assign master3_ar__ready = slave3_ar__ready;
    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_ar__valid <= 1'd0;
        else if (~slave3_ar__valid | ~master3_ar__ready) slave3_ar__valid <=
master3_ar__valid;



    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_ar__id <= 4'd0;
        else if (master3_ar__valid & master3_ar__ready) slave3_ar__id <=
master3_ar__id;
```



```
    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_ar__addr <= 32'd0;
        else if (master3_ar__valid & master3_ar__ready) slave3_ar__addr <=
master3_ar__addr;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_ar__burst <= 2'd0;
        else if (master3_ar__valid & master3_ar__ready) slave3_ar__burst <=
master3_ar__burst;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_ar__len <= 8'd0;
        else if (master3_ar__valid & master3_ar__ready) slave3_ar__len <=
master3_ar__len;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_ar__size <= 3'd0;
        else if (master3_ar__valid & master3_ar__ready) slave3_ar__size <=
master3_ar__size;



    assign master3_w__ready = slave3_w__ready;
    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_w__valid <= 1'd0;
        else if (~slave3_w__valid | ~master3_w__ready) slave3_w__valid <=
master3_w__valid;



    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_w__data <= 512'd0;
        else if (master3_w__valid & master3_w__ready) slave3_w__data <=
master3_w__data;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_w__strb <= 64'd0;
        else if (master3_w__valid & master3_w__ready) slave3_w__strb <=
master3_w__strb;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_w__last <= 1'd0;
        else if (master3_w__valid & master3_w__ready) slave3_w__last <=
master3_w__last;
```



```
    assign master3_b__valid = slave3_b__valid;
    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_b__ready <= 1'd0;
        else if (~slave3_b__ready | ~master3_b__valid) slave3_b__ready <=
master3_b__ready;



    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) master3_b__id <= 4'd0;
        else if (master3_b__ready & master3_b__valid) master3_b__id <=
slave3_b__id;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) master3_b__resp <= 2'd0;
        else if (master3_b__ready & master3_b__valid) master3_b__resp <=
slave3_b__resp;



    assign master3_r__valid = slave3_r__valid;
    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) slave3_r__ready <= 1'd0;
        else if (~slave3_r__ready | ~master3_r__valid) slave3_r__ready <=
master3_r__ready;



    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) master3_r__data <= 512'd0;
        else if (master3_r__ready & master3_r__valid) master3_r__data <=
slave3_r__data;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) master3_r__id <= 4'd0;
        else if (master3_r__ready & master3_r__valid) master3_r__id <=
slave3_r__id;

    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) master3_r__resp <= 2'd0;
        else if (master3_r__ready & master3_r__valid) master3_r__resp <=
slave3_r__resp;
```



```
    always_ff @(posedge clk or negedge rst_n)
        if (!rst_n) master3_r__last <= 1'd0;
        else if (master3_r__ready & master3_r__valid) master3_r__last <=
slave3_r__last;



    assign bo__a = bi__a;
    assign bo__b = bi__b;


    assign bco__a = bci__a;
    assign bco__b = bci__b;
    assign bco__c = bci__c;


    assign cast_o__a = bi__a;
    assign cast_o__b = bi__b;

    assign cast_o__c = 2'd2;

endmodule
```


FUNCTION DESCRIPTION

P2V class functions

```
allow_undriven(self, name)
    Set module signal as driven.

    Args:
        name([clock, list, str]): name/s for signals to set undriven

    Returns:
        None

allow_unused(self, name)
    Set module signal/s as used.

    Args:
        name([clock, list, str]): name/s for signals to set used

    Returns:
        None

assert_always(self, clk, condition, message, params=[], fatal=True)
    Assertion on Verilog signals with clock (ignores condition during async reset
if present).

    Args:
        condition(str): Error occurs when condition is False
        message(str): Error message
        params([str, list]): parameters for Verilog % format string
        fatal(bool): stop on error

    Returns:
        success

assert_never(self, clk, condition, message, params=[], fatal=True)
    Assertion on Verilog signals with clock (ignores condition during async reset
if present).

    Args:
        condition(str): Error occurs when condition is True
        message(str): Error message
        params([str, list]): parameters for Verilog % format string
```



```
        fatal(bool): stop on error

    Returns:
        success

assert_static(self, condition, message, fatal=True)
    Assertion on Python varibales.

    Args:
        condition(bool): Error occurs when condition is False
        message(str): Error message
        fatal(bool): stop on error

    Returns:
        success

assign(self, tgt, src, keyword='assign')
    Signal assignment.

    Args:
        tgt([clock, str, dict]): target signal
        src([clock, int, str, dict]): source Verilog expression
        keyword(str): prefix to assignment

    Returns:
        None

check_always(self, condition, message, params=[], fatal=True)
    Assertion on Verilog signals with no clock.

    Args:
        condition(str): Error occurs when condition is False
        message(str): Error message
        params([str, list]): parameters for Verilog % format string
        fatal(bool): stop on error

    Returns:
        success

check_never(self, condition, message, params=[], fatal=True)
    Assertion on Verilog signals with no clock.

    Args:
        condition(str): Error occurs when condition is True
        message(str): Error message
```



```
        params([str, list]): parameters for Verilog % format string
        fatal(bool): stop on error

    Returns:
        success

gen_rand_args(self, override={})
    Generate random module parameters and register in csv file.

    Args:
        override(dict): explicitly set these parameters overriding random values

    Returns:
        random arguments (dict)

get_fields(self, strct, attrib='name')
    Get struct fields.

    Args:
        strct(dict): p2v struct
        attrib(str): field attribute to extract

    Returns:
        list of field names (or other attribute)

inout(self, name)
    Create an inout port.

    Args:
        name(str): port name

    Returns:
        None

input(self, name, bits=1)
    Create an input port.

    Args:
        name([str, clock]): port name
        bits([int, float, dict, tuple]): int is used fot number of bits.
                                         float is used to mark struct control
signals.
                                         list is used to prevent a scalar signal
(input x[0:0]; instead of input x;).
```



```
                                         tuple is used for multi-dimentional
Verilog arrays.
                                         dict is used as a struct.

    Returns:
        p2v struct if type is struct otherwise None

line(self, line='', remark=None)
    Insert Verilog code directly into module without parsing.

    Args:
        line(str): Verilog code (can be multiple lines)
        remark([None, str]): optional remark added at end of line

    Returns:
        None

logic(self, name, bits=1, assign=None, initial=None)
    Declare a Verilog signal.

    Args:
        name([clock, list, str]): signal name
        bits([int, float, dict, tuple]): int is used fot number of bits.
                                         float is used to mark struct control
signals.
                                         list is used to prevent a scalar signal
(logic x[0:0]; instead of logic x;).
                                         tuple is used for multi-dimentional
Verilog arrays.
                                         dict is used as a struct.
        assign([int, str, dict, None]): assignment value to signal using an
assign statement
        initial([int, str, dict, None]): assignment value to signal using an
initial statement

    Returns:
        None

output(self, name, bits=1)
    Create an output port.

    Args:
        name([str, clock]): port name
        bits([int, float, dict, tuple]): int is used fot number of bits.
```



```
                                        float is used to mark struct control
signals.
                                        list is used to prevent a scalar signal
(output x[0:0]; instead of output x;).
                                        tuple is used for multi-dimentional
Verilog arrays.
                                        dict is used as a struct.

    Returns:
        p2v struct if type is struct otherwise None

parameter(self, name, val)
    Declare a Verilog parameter.

    Args:
        name([str, clock]): parameter name
        val([int, str]): parameter value

    Returns:
        None

remark(self, comment)
    Insert a Verilog remark.

    Args:
        comment([str, dict]): string comment or one comment like per dictionary
pair

    Returns:
        None

sample(self, clk, tgt, src, valid=None, reset_val=0, bits=None)
    Sample signal using FFs.

    Args:
        clk(clock): p2v clock (including optional reset/s)
        tgt(str): target signal
        src(str): source signal
        valid([str, None]): qualifier signal
        reset_val([int, str]): reset values
        bits([int, None]): explicitly specify number of bits

    Returns:
        None
```



```
set_modname(self, modname=None, suffix=True)
    Sets module name.

    Args:
        modname([None, str]): explicitly set module name
        suffix(bool): automatically suffix module name with parameter values

    Returns:
        True if module was already created False if not

set_param(self, var, kind, condition=None, remark='', suffix=None)
    Declare module parameter and set assertions.

    Args:
        var: module parameter
        kind([type, list of type]): type of var
        condition([None, bool]): parameter constraints
        remark(str): comment
        suffix([None, str]): explicitly define parameter suffix

    Returns:
        None

verilog_module(self, modname, params={})
    Instantiate Verilog module (pre-existing source file).

    Args:
        modname(str): Verilog module name
        params(dict): Verilog module parameters

    Returns:
        success

write(self)
    Write the Verilog file.

    Args:
        NA

    Returns:
        p2v_connects struct with connectivity information
```


P2V misc class functions

```
    bin(n, bits=None, add_sep=4, prefix="'b")
        Represent integer in Verilog binary representation.

        Args:
            n(int): input value
            bits([None, int]): number of bits for value
            add_sep(int): add underscore every few characters for easier reading
of large numbers
            prefix(str): binary annotation

        Returns:
            Verilog code

    bit(name, idx)
        Extract a single bit from a Verilog bus.

        Args:
            name(str): signal name
            idx([int, str]): bit location (can also be a Verilog signal for multi
dimention arrays)

        Returns:
            Verilog code

    bits(name, bits, start=0)
        Extract a partial range from a Verilog bus.

        Args:
            name(str): signal name
            bits(int): number of bits to extract
            start(int): lsb

        Returns:
            Verilog code

    ceil(n)
        Round to ceil.

        Args:
            n([int, float]): input value

        Returns:
```



```
            int

    concat(vals, sep=None, nl_every=None)
        Converts a Python list into Verilog concatenation or join list of signals
with operator.

        Args:
            vals(list): list of signals
            sep([None, str]): if None will perform Verilog concatenation else
will perfrom join on sep
            nl_every([None, int]): insert new line every number of items

        Returns:
            Verilog code

    cond(condition, true_var, false_var=None)
        Converts a Python list into Verilog concatenation or join list of signals
with operator.

        Args:
            condition(bool): condition
            true_var:  variable for True condition
            false_var: variable for False condition

        Returns:
            Selected input parameter

    dec(n, bits=1)
        Represent integer in Verilog decimal representation.

        Args:
            n(int): input value
            bits(int): number of bits for value

        Returns:
            Verilog code

    hex(n, bits=None, add_sep=4, prefix="'h")
        Represent integer in Verilog hexademical representation.

        Args:
            n(int): input value
            bits([None, int]): number of bits for value
            add_sep(int): add underscore every few characters for easier reading
of large numbers
```



```
        prefix(str): hexadecimal annotation

    Returns:
        Verilog code

is_hotone(var, bits, allow_zero=False)
    Check if a Verilog expression is hot one.

    Args:
        var(str): Verilog expression
        bits(int): number of bits of expression
        allow_zero(bool): allow expression to be zero or hot one

    Returns:
        Verilog code

is_pow2(n)
    Returns True of number is power to 2.

    Args:
        n(int): input value

    Returns:
        bool

log2(n)
    Log2 of number.

    Args:
        n(int): input value

    Returns:
        int

pad(left, name, right=0, val=0)
    Verilog padding for lint and for shift left.

    Args:
        left(int): msb padding bits
        name(str): signal name
        right(int): lsb padding bits
        val(int): value for padding

    Returns:
        Verilog code
```



```
roundup(num, round_to)
    Round number to the closest dividing number.

    Args:
        num(int): input value
        round_to(int): returned values must divide by this value

    Returns:
        bool
```

78

P2V tb class functions

```
dump(self, filename='dump.fst')
    Create an fst dump file.

    Args:
        filename(str): dump file name

    Returns:
        None

fifo(self, name, bits=1)
    Create SystemVerilog behavioral fifo (queue).

    Args:
        name(str): name of signal
        bits(int): width of fifo

    Returns:
        None

gen_busy(self, clk, name, max_duration=100, max_delay=100, inverse=False)
    Generate random behavior on signal, starts low.

    Args:
        clk(clock): p2v clock
        name(str): signal name
        max_duration(int): maximum number of clock cycles for signal to be high
        max_delay(int): maximum number of clock cycles for signal to be low

    Returns:
        None

gen_clk(self, clk, cycle=10, reset_cycles=20, pre_reset_cycles=5)
    Generate clock and async reset if it exists.

    Args:
        clk(clock): p2v clock
        cycle(int): clock cycle
        reset_cycles(int): number of clock cycles before releasing reset
        pre_reset_cycles(int): number of clock cycles before issuing reset

    Returns:
```



```
        None

gen_en(self, clk, name, max_duration=100, max_delay=100)
    Generate random behavior on signal, starts high.

    Args:
        clk(clock): p2v clock
        name(str): signal name
        max_duration(int): maximum number of clock cycles for signal to be low
        max_delay(int): maximum number of clock cycles for signal to be high

    Returns:
        None

rand_bool(self)
    Random bool with 50% chance.

    Args:
        NA

    Returns:
        bool

rand_chance(self, chance)
    Random bool with chance.

    Args:
        chance(int): chance for True

    Returns:
        bool

rand_hex(self, bits)
    Random hex value with set width.

    Args:
        bits(int): bits of hex value

    Returns:
        Verilog hex number

rand_int(self, a, b=None)
    Random integer value.

    Args:
```



```
        a(int): min val (if b is None a is in range [0, a])
        b([None, int]: max val

    Returns:
        int

rand_list(self, l)
    Random item from list.

    Args:
        l(list): list of items to pick one from

    Returns:
        random item from list

register_test(self, args=None)
    Register random module parameters to csv file.

    Args:
        args([None, dict]): argument dictionary to be written

    Returns:
        None

set_timeout(self, clk, timeout=100000)
    Generate random behavior on signal, starts high.

    Args:
        clk(clock): p2v clock
        timeout(int): number of cycles before test is ended on timeout error

    Returns:
        None

test_fail(self, condition=None, message=None, params=[])
    Finish test with error if condition is met.

    Args:
        condition([None, str]): condition for finishing test, None is
unconditional
        message([None, str]): completion message
        params([str, list]): parameters for Verilog % format string

    Returns:
        None
```



```
test_finish(self, condition, pass_message=None, fail_message=None, params=[])
    Finish test if condition is met.

    Args:
        condition([None, str]): condition for finishing test, None is
unconditional
        pass_message([None, str]): good completion message
        fail_message([None, str]): bad completion message
        params([str, list]): parameters for Verilog % format string

    Returns:
        None

test_pass(self, condition=None, message=None, params=[])
    Finish test successfully if condition is met.

    Args:
        condition([None, str]): condition for finishing test, None is
unconditional
        message([None, str]): completion message
        params([str, list]): parameters for Verilog % format string

    Returns:
        None
```


COMMAND LINE ARGUMENTS

OUTPUT DIRECTORY
-outdir: Sets the name of the output directory.

-rm_outdir / --rm_outdir: By default, the output directory is erased and recreated every run, these flags enable running on a pre-existing directory.

LOG
-log: Set severity level of logger, by default it is set to DEBUG.

SEARCH PATH
-I: Add directories for source files, Python or Verilog.

FILE GENERATION
-prefix: Add a prefix to all Veilrog files and module names.

-indent / --indent: Suppress Verilog file indentation.

-header: Provide copyright header for Verilog files.

MODULE GENERATION
-params: pass top level module parameters or csv file with parameters per line.

Example for top module parameters: -params ‘{“num”:2,”bits”:32,”sample”:True}’

BUILD OPERATIONS
-stop_on: Set error level that stops build, default is CRITICAL.

-seed: Seed for Python and Verilog random generation. 0 generates a random seed.

-gen_num: Build multiple random permutations of top module.

VERILOG OPERATIONS
-lint / --lint: Suppress Verilog linting.

-sim/ --sim: Run Verilog simulation.

-allow_missing_tools: do not error out on missing external tools like indentation or lint.

-sim_args: pass parameters to top level rtl in simulation (-params will pass parameters to test bench)

