Generic Verilog Platform Designer

cli-programs and their purpose

module_gen

This program is used to convert a vgen.json file into a v_class that can be used with pythonhld.

```
usage: Module Generator [-h] [--validate-only] [-o OUT] [--dump-json] json_file

Convert a module json description into a python module that can be used in the
platform generator

positional arguments:
  json_file          Path to json file to read

options:
  -h, --help         show this help message and exit
  --validate-only    Only validate the file don't generate module
  -o OUT, --out OUT  output location can be a file or folder
  --dump-json        dump json file which is the original file filled with all
default populated values
```

parse_ipxact

This program allows you to take a ipxact file from Intel or Xilinx and convert it to a .vgen.json file

```
usage: IPXACT Parser [-h] [-o OUT] ipxact_file


Parses a ipxact file to generate the required vgen.json for a module

positional arguments:
  ipxact_file        path to ipxact file to read

options:
  -h, --help         show this help message and exit
  -o OUT, --out OUT  output location can be a file or folder
```

parse_ipxact_interface

Xilinx provides interface definitions in their ip folder with vivado. These interface definition files can be parsed by this tool and turned into schemas for pythonhld. This will help to validate any instances of the interface and populate default values for unconnected lines.

```
usage: IPXACT Parser [-h] [-o OUT] ipxact_file

Parses a ipxact interface file to generate the required schema.json for a module

positional arguments:
  ipxact_file        path to ipxact file to read

options:
  -h, --help         show this help message and exit
  -o OUT, --out OUT  output location can be a file or folder
```

Code Generation Structure

Generator Class

This class handles connections between verilog modules. It will do things like create wires, assign wires together, and orchestrate overall generation. Whenever a module is instantiated its constructor must be passed this generator to cause code generation.

V_Classes

v_classes are python classes that correspond with a verilog module. They contain all of the information about the interfaces and parameters required for a creation of a module. Each v_class is responsible for generating only the verilog code to create its module. Modules will also generate any muxes/demuxes that an interface which connects to multiple other interfaces needs.

Interconnect Classes

Every interconnect class is designed for a specific interface type. Interfaces which are generic conduits should never have a interconnect since that would be shared for all conduits. If you need a interconnect you should give your interface a proper type.

JSON module definitions

To import your verilog moduels into pythonhld you first need to create a .vgen.json file. This json file describes all of the interfaces present in your module. You can create this file manually, instructions on all the fields can be found in the `schema_doc.pdf` file. Alternatively if you have already packaged your module as a ip you can use the ipxact file that is stored with it. This file can be directly converted to a .vgen.json file.

Using pythonhld

Creating the root verilog generator

Every time you want to create a design with pythonhld you need to create a verilog generator this can be seen below.

```
gen = verilog_generator(start_file="input_file",output_file="out_file")
```

This start file indicates what verilog file you will be appending to. This is usually your top level file. The output_file indicates where the new file should be stored. The output file will consist of the code from the start file, then all the generated and connected modules, and finally endmodule.

Instantiating a v_class

```
my_mod = example_module(gen,"example_mod_name",parameters={"PORT_WIDTH":32})
```

The above code shows the general structure of v_class initialization. The first two arguments are always required. The first one is the verilog_generator instance that is being used. The second is a unique name for the module.

The parameters argument will always be present and allows you to pass parameters to the underlying module.

Sometime v_classes have required and optional parameters that show up in the constructor. For instance if the example_module above had specified `PORT_WIDTH` as a parameter in it's .vgen.json you could do the following instead.

```
my_mod = example_module(gen,"example_mod_name", PORT_WIDTH=32)
```

NOTE: If a parameter is specified both as an arg and in the parameters dictionary the value passed in the parameters argument will be the one used

```
my_mod = example_module(gen,"example_mod_name", PORT_WIDTH=32,parameters=
{"PORT_WIDTH":64})
# my_mod will have PORT_WIDTH set to 64 since the parameters argument always wins
```

Connecting Interfaces

```
c_gen = clk_gen(gen,"example_mod_1")
e_mod = example_module(gen,"example_mod_2")
gen.connect(c_gen.interface_clk_out,e_mod.interface_clk)
```

The above code shows how you can connect two interfaces in this case a clock. every interface in a v_class can be accessed by the following method `var.interface_"interface_name"`

When connecting interfaces you always need to make sure your are connecting a source to a sink. PythonHLD will not let you connect two sources or two sinks together. Additionally the tool will not let you connect mismatched types so you can't connect a conduit to a clock or a avalon to a aximm. If you need a adapter then you must define it.

The tool will however allow you to one source to multiple sinks or multiple sources to one sink assuming you have defined the mux/demux interconnects required. Additionally v_classes can restrict their modules from

generating interconnects if desired.

The interconnects that are generated can be customized during the construction of a design

```
gen.set_interconnect("avalon",fast_interconnect)
gen.connect(a1.avalon_m,b1.avalon_s)
gen.connect(a1.avalon_m,c1.avalon_s) #a1's avalon_m will receive a mux generated
by the fast_interconnect class
gen.set_interconnect("avalon",slow_interconnect)
gen.connect(a2.avalon_m,b2.avalon_s)
gen.connect(a2.avalon_m,c2.avalon_s) #a2's avalon_m will receive a mux generated
by the slow_interconnect class
gen.connect(a1.avalon_m,d1.avalon_s) #The interconnect registered during the last
connection is the one used, so this switches a1's avalon_m to a slow_interconnect
generated mux
```

Generating Design

Once you have finished defining all of your connections you can call `gen.generate_verilog()`.