Design Interface Generation for Verification

Saurav Kale (EE19B141), guided by Prof. Nitin Chandrachoodan, V. Kamakoti

1 Introduction

The SHAKTI microprocessor design is done in a higher level HDL called Bluespec. The key advantages of using Bluespec are:

• **The levels of type abstraction it offers**: There are several inbuilt types like Tuple, Vector and Maybe, which are useful for grouping a bunch of related signals together. There is also the ability to provide user defined structs, typedefs and so on. Bluespec is also rigorous in its type checking. There are no implicit casts. This helps the designer to code a design faster. These types are later elaborated into bit type in Verilog. A port belonging to a complex type in Bluespec may elaborate to one or several ports in the generated Verilog.

• **The handshaking signals are completely abstracted**: Bluespec provides interfaces which are based on transactions called "methods". Methods are elaborated into data and handshaking signals when elaborated into Verilog. Therefore, coding a design in Bluespec completely abstracts away the handshaking signals from the designer.

These advantages of Bluespec make the design process fast and easy to understand for a designer. The design, when completed in Bluespec, is elaborated into Verilog, and then handed over for verification. However, from the perspective of a verification engineer, this poses some problems. Verification engineers have to work with the elaborated ports and drive the handshaking signals manually. They cannot take advantage of the abstractions which Bluespec offers to the designer.

In this work, we present a framework which allows the verification engineer to also take advantage of these abstracted types, while still being able to test the generated Verilog.

2 Overview of the DIG Framework

Verification for SHAKTI is done in an automated fashion using a python package called cocotb. Cocotb hooks into the Verilog design-under-test (DUT), and drives the inputs as well as collects the output. The output obtained can then be compared to the result generated by a trusted model of the functionality, usually coded in python itself.

The Design Interface Generation (DIG) framework is a set of python packages which bring the type abstractions and automated handshaking of Bluespec designs over to python. The DIG framework seamlessly integrates with cocotb, and simplifies the workflow of the verification engineer immensely.

Figure 1 illustrates the DIG flow. We shall explain each step in detail in the following sections.

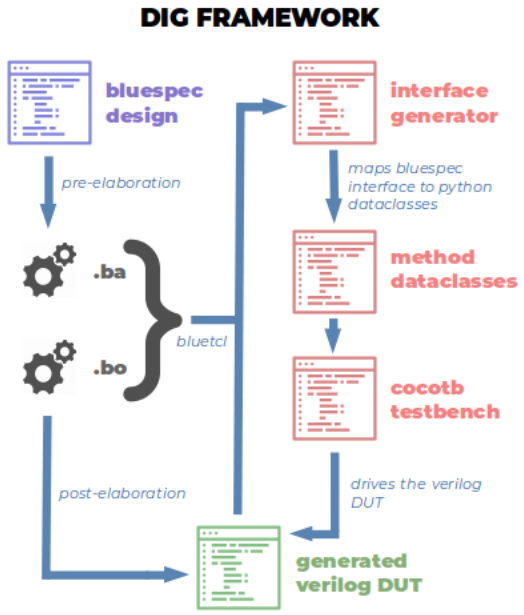

Figure 1: High level flow of DIG framework

3 Bluespec elaboration and Bluetcl

3.1 Bluespec elaboration

The key problem to be solved was to handle the unpredictable elaboration of complicated types in Bluespec. The same type may turn into a single port in verilog, or multiple ports depending upon where it is used in the design.

Several tests were run, experimenting with various nested type combinations of Tuples, Vectors, and Maybe. We shall refer to a complicated type as MyType in further discussion. The test design consisted of a single register of MyType and two methods. The putVec method, being an ActionValue method, updated the value of the register, and returned the previous value of the register. The getVec method, being a Value method, simply returned the current value of the register.

It was observed that for MyType being Vector#(5, Tuple2#(Bit#(1), Bit#(5))), the getVec output was elaborated as 5 separate Verilog ports, one for each element of the vector.

However, for MyType being Tuple2#(Bit#(1), Vector#(5, Bit#(2))), it simply turns into 2 ports, one for each element in the tuple, with the five 2-bit elements of the vector concatenated together into a single 10 bit port.

Meanwhile, for all cases, the putVec output simply combines any complex type into a single port of the required bit width.

Lots of similar tests were carried out, and the conclusion was that it is not trivial to reverse engineer this elaboration. The source code for Bluespec was briefly studied, but proved to be too complex a solution for our purposes.

3.2 Bluetcl

Bluetcl is an extension of the Tcl language for working with Bluespec intermediate files.

When Bluespec is compiled into Verilog, it generates an intermediate file with an extension of .bo which contains some information about the partly processed design.

When Bluespec is compiled for execution using BlueSim (Bluespec's own simulator), it creates an intermediate file with an extension of .ba which is similar to .bo but contains different information.

Both these files are not directly human readable. Bluetcl serves as an interface to get information from these files. Bluetcl proved useful in the development of this framework in the following ways:

• Bluetcl commands (listed in Bluetcl::module) can be used to generate a list of all the methods in a design interface. The output is a list of partially elaborated ports for each method and their corresponding bit widths. This list is not complete, as it only gives this information for the input ports. As for the outputs, it simply prints the complicated bluespec type. This proved to be a major issue that increased the complexity of the problem significantly.

• Bluetcl commands can be used to extract the hierarchy of a complex type. Given a complex type, the Bluetcl::type full type_name command is able to extract all the nested types down to the bit level. This is indeed very useful, but it still does not solve the problem completely, because for example, if a complex type with 4 elements turns into two ports, one still has no information about how the split has happened at the bit level.

The DIG Framework consists of two Bluetcl scripts, one to extract the partially elaborated methods, and another to extract the hierarchy of the complex type. These are run via python subprocess module within the python package.

The documentation for Bluetcl is cryptic at best, and due to having no background in Tcl scripting, it took a considerable amount of time to understand and develop the scripts.

4 Parsing the Verilog

The above sections have explained why Bluetcl is not a sufficient source on its own for mapping the Bluespec methods to elaborated Verilog ports.

In order to understand how the type hierarchy maps to Verilog, we need to parse the generated Verilog file and know how many ports the output of the method has elaborated into, and what is the bit width of each Verilog port.

The way ports are concatenated is still not revealed completely via this information. There is still a pending issue if there are a large number of output ports, however, many cases can

be abstracted and simplified using this information.

5 Generating the Classes

Using this information, classes were generated for each method in the design, and a class to represent the who design interface, consisting of attributes being instances of all the method classes created.

Each Bluespec method is converted to two Python classes, one being for the input side, and the other being for the output. Each class created handles both the driving/collecting of the data ports, and the associated handshaking signals. Each class also offers a method to compare expected value and obtained value.

The structure of the generated classes is described below:

• **Data ports**:
These are implemented using a dictionary of generated verilog ports as keys, mapping to values. They can be set manually, and driven to a DUT. They can also be compared with attributes another instance of the same class, which helps in asserting functional correctness.

• **Handshaking ports**:
This concists of a list of ports like enable signals, ready signals, and any other necessary handshaking signals like valid. These cannot be given a numeric value unlike data ports. Handshaking ports are automatically asserted whenever the data ports are driven. This thus abstracts away the handshaking signals completely.

• **Drive and catch**:
Each such method has an "active" state. When the interface class is driven to the DUT, all the "active" methods are driven to the DUT, while the inactive methods, the transaction is disabled by asserting the appropriate handshaking signals.

For the output side, all signals which have ready signal asserted are assigned the outputs generated by the DUT. This operation is done in the "catch" function in each class.

The verification engineer thus "drives" inputs, and "catches" the outputs. Then a comparison is made with expected value to assert functional correctness.

• **The Option of Fine control**:
Despite this abstraction, a verification engineer might occasionally need fine control. In this case, there are special functions which can bypass these abstractions and offer finer control when needed.

6 Integration with cocotb

The DIG framework integrates neatly with cocotb by accepting the DUT as a parameter in the generated classes. The values can be driven all at once, or to some selectively, thanks to the abstractions.

Another small issue with regard to the handshaking signals is the reset signal at the top module. One particularly annoying side effect of driving a Verilog module generated from Bluespec using Cocotb is that we need to perfectly emulate the Bluespec execution environment in order to get the same results as intended by the designer.

Bluespec initializes all internal registers and other signals to 10101010.... depending on the bit width. When reset signal is asserted, all registers and signals turn into the initial value specified by the designer. The Bluespec execution environment does this implicitly every time a simulation is run. This is handled in the interface class by asserting the reset and clock signals for the top module correctly just before the actual inputs begin to be driven to the DUT.

The verification engineer is now free to work at any abstraction level, thus increasing the speed and efficiency of the verification process for complicated Bluespec designs. The design and verification engineers now both speak the same language of abstractions, thus improving the overall workflow.

Note: All further sections are code examples and further comments about the work.

7 Examples of DIG assisted verification

Here we present a few examples of the ease of use offered by the DIG framework, first on a toy module, then on one functional unit each of the SHAKTI mbox (RISC-V MUL/DIV extension) and fbox (RISC-V Floating Point extension).

7.1 mk_test: A toy module

```
typedef Tuple2#(Bit#(1), Vector#(5, Bit
    #(2))) MyType;

interface Ifc_vectest;
       method MyType getVec();
       method ActionValue#(MyType) putVec
           (MyType a);
endinterface

(* synthesize *)
module mk_test11(Ifc_vectest);
       Reg#(MyType) rg_a <- mkReg(unpack
           (0));

       method MyType getVec;
              return rg_a;
       endmethod

       method ActionValue#(MyType) putVec
           (MyType a);
              rg_a <= a;
              return rg_a;
       endmethod
endmodule
```

Using the DIG generated classes, we have the following code to drive and verify mk_test11:

```
data = dcls.mk_test11()
data._putVec_in.set("putVec_a", 1)
expected = dcls.mk_test11()

await FallingEdge(dut.CLK)
# Synchronize with the clock
for i in range(1, 11):
       data.sleepall()
       data._putVec_in.set("putVec_a", i)
       expected._putVec_out.set("putVec",
            i)
       data._putVec_in.wake()
       data.drive(dut)
       await FallingEdge(dut.CLK)
       data.catch(dut)
       assert data._putVec_out.cmp(
           expected._putVec_out), f"
           output q was incorrect on the
           {i}th cycle"
```

"data" is the driven interface, and "expected" are the expected values to be compared with. This can be generated using a python model. Here, it simply verifies a register's ability to sample on the positive clock edge.

7.2 srt_radix4_divider: radix 4 divider from SHAKTI mbox

Bluespec interface:

```
interface Ifc_srt_radix4_divider;
  method Action ma_start(Bit#(`XLEN)
      dividend, Bit#(`XLEN) divisor, Bit
      #(4) opcode, Bit#(3) funct3);
  method ActionValue#(Tuple2#(Bit#(1),Bit
      #(`XLEN))) mav_result();
  method Action ma_set_flush(bit c);
endinterface
```

DIG assisted testbench:

```
data = dcls.mk_srt_radix4_divider()
data._ma_start_in.set("ma_start_opcode",
    opcode)
data._ma_start_in.set("ma_start_funct3",
    funct3)
data.sleepall()

await data.init_dut(dut)
cmp_index = 0
for rg_cycle in range(2000):
       data.sleepall()
    data._mav_result_in.wake()
       OP1 = random.randrange
           (0,18446744073709551615,100)
       OP2 = random.randrange
           (0,18446744073709551615,100)
       if rg_cycle % 39 == 0:
              data._ma_start_in.wake()
              data._ma_start_in.set("
                  ma_start_dividend", OP1
                  )
              data._ma_start_in.set("
                  ma_start_divisor", OP2)

       # drive
       data.drive(dut)

       await FallingEdge(dut.CLK)
       # catch + verify
       data.catch(dut)
       if (data._mav_result_out.get("
           mav_result") >> 64 == 1 and
           cmp_index < 8):
              dut._log.info("MODEL " +
                  str(model.divider_model
                  (OP1, OP2, opcode,
                  funct3)))
       cmp_index += 1
       await RisingEdge(dut.CLK)
```

7.3 fp_add_sub: Floating Point adder/-subtracter from SHAKTI fbox

Bluespec interface:

```
interface Ifc_fpu_add_sub#(numeric type e
    , numeric type m, numeric type nos);
        method Action send(Tuple3#(
            FloatingPoint#(e,m),
                 FloatingPoint#(e,m),
                 RoundMode) operands);
        method ReturnType#(e,m) receive();
endinterface
```

DIG assisted testbench:

```
def generate_rand_inp(rnd_mode):
        a = dcls.Float()
        b = dcls.Float()
        OP1 = random.randint(-2147483648,
            2147483647)
        OP2 = random.randint(-2147483648,
            2147483647)
        f1 = bitstring.BitArray(int=OP1,
            length=32)
        f2 = bitstring.BitArray(int=OP2,
            length=32)
        expected = fp_add_sub_model(f1.
            float, f2.float)
        a.set(OP1)
        b.set(OP2)
        data = cls.
            mk_fpu_add_sub_sp_instance()
        data._send_in.set("send_operands",
             int(rnd_mode + a.__bin__() +
            b.__bin__(), 2))
        return data, expected
@cocotb.test()
async def test(dut):
        clk = clock_gen(dut.CLK)
        data, expected = generate_rand_inp
            ("000") #near even
        cocotb.start_soon(clk)
        await data.init_dut(dut)
        for i in range(100):
                data.sleepall()
                data._send_in.wake()
                data.drive(dut)
                await FallingEdge(dut.CLK)
                data.catch(dut)
                if (data._receive_out.
                    active):
            data._receive_out.cmp(expected
                )
                await RisingEdge(dut.CLK)
```

8 Conclusion

The DIG framework has thus been presented. It offers a better way to verify Bluespec designs. Earlier, the verification engineer would have had to check the Verilog and Bluespec design for each port, manually decide how long the lists holding the data would have to be, and write functions to generate the necessary bit representation, compare the values obtained vs. values expected, and drive the necessary handshaking signals by manually. With DIG, the amount of code the verification engineer has to write is reduced to a large extent. Verification engineers can take advantage of abstraction, and thus more easily verify Bluespec designs.

9 Summary of work done and potential future

Due to being completely new to the world of verification, a lot of time was spent learning cocotb, tcl, bluetcl and details of bluespec elaboration, which had very limited documentation. Lots of examples were explored. The main code for generating the dataclasses was written. Handling the automated handshaking proved to be quite a task given that I had to ensure ease of use without being too limited.

The abstraction offered is currently limited by the lack of information about how ports are concatenated, but it may be possible to derive that information by comparing bit widths and parts of the names of the verilog port. Going down this route was attempted, but it became too time consuming. For now, when things get too complex, a comment is added to the generated classes about the available information from bluetcl. Further work could explore solving this problem.

After further development this package will be used for Coverage collection for RISC-V in verification of SHAKTI cores and IPs.

Repository for the work:

References

[1] SHAKTI Verification team, https://gitlab.com/shaktiproject/core-py-verif

[2] Cocotb documentation, https://www.cocotb.org/