Design Interface Generation for verification

A better way to verify bluespec designs

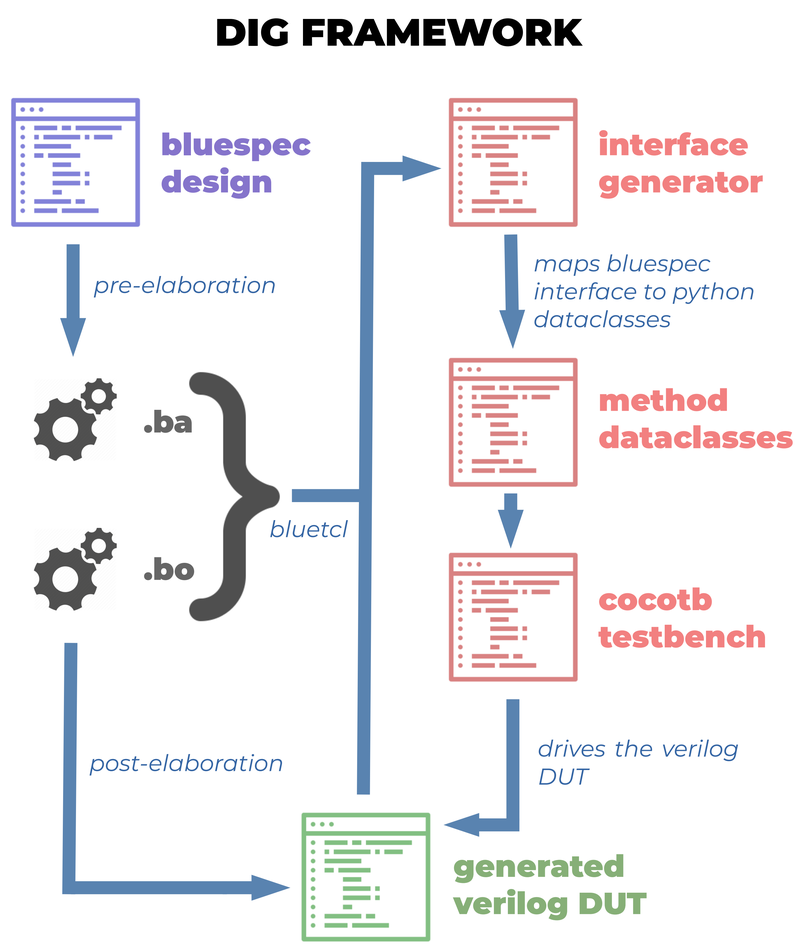
by: Saurav Kale (EE19B141) Advisor(s): Prof. Nitin Chandrachoodan, Prof. V. Kamakoti EE4901: Miniproject

Objective

One of the major advantages of using Bluespec SystemVerilog for digital design is its rigorous type driven approach. This eliminates the need for handling every signal at the bit level, unlike verilog. It also eliminates the need to worry about handshaking signals. These are derived when the bluespec is compiled into verilog.

As far as verification is concerned, the verification engineer deals with the generated verilog. Elaboration errors from bluespec to verilog are identified this way. However, this also means that verification engineers cannot take advantage of the type abstractions which bluespec offers. A verification engineer has to painstakingly look at how the bluespec methods map to the handshaking signals.

We present in this work, a python package, which integrates with cocotb which allows the verification engineer to take advantage of type abstraction in testing the verilog DUT.



DIG framework

The Design Interface Generation (DIG) framework consists of a combination four components:

Bluetcl
Bluetcl is an extension to tcl for bluespec files. It is useful in order to process intermediate .ba and .bo files generated during the compilation of bluespec into verilog. Bluetcl was used in order to obtain the bitwise split up of structs and complicated types used in the bluespec code.

Generated Verilog
The bluespec interface is converted into several data and handshaking signals in verilog. The name and bitwise split of each port is obtained via parsing the verilog.

Method Dataclasses
Using the above information, python dataclasses corresponding to each method are generated, and written to a python file, mapping the complex bluespec interface to a python dataclass.

cocotb testbench
cocotb is a python framework which allows a verification engineer to hook into the verilog DUT, drive the inputs and collect the outputs in python itself. The verification engineer now uses the generated dataclasses to drive the inputs to the DUT.

This allows the verification engineer to take full advantage of type abstraction, and making the verification of bluespec designs easier.

Tests on SHAKTI modules

MBOX:
SRT RADIX4 DIVIDER

```
**************************************************
MODULE: mk_srt_radix4_divider, CLOCK: CLK, RESET: RST_N
**************************************************
VERILOG METHOD: ma_start
**********
INPUT_NAME BITWIDTH
ma_start_dividend 64
ma_start_divisor 64
ma_start_opcode 4
ma_start_funct3 3
EN_ma_start 1
OUTPUT_NAME BITWIDTH
RDY_ma_start 1
**********
VERILOG METHOD: mav_result
**********
INPUT_NAME BITWIDTH
EN_mav_result 1
OUTPUT_NAME BITWIDTH
mav_result 65
RDY_mav_result 1
**********
VERILOG METHOD: ma_set_flush
**********
INPUT_NAME BITWIDTH
ma_set_flush_c 1
EN_ma_set_flush 1
OUTPUT_NAME BITWIDTH
RDY_ma_set_flush 1
**********
**************************************************
```

```
@dataclass(init=False)
class ma_start_in:
        ma_start_dividend: int = 0
        ma_start_divisor: int = 0
        ma_start_opcode: int = 0
        ma_start_funct3: int = 0
        def set(self, val):
                self.ma_start_dividend = (val.integer >> 71) & 0xffffffffffffffff
                self.ma_start_divisor = (val.integer >> 7) & 0xffffffffffffffff
                self.ma_start_opcode = (val.integer >> 3) & 0xf
                self.ma_start_funct3 = (val.integer >> 0) & 0x7
                return

        def __bin__(self):
                val = ""
                val += "{0:064b}".format(self.ma_start_dividend & 0xffffffffffffffff)
                val += "{0:064b}".format(self.ma_start_divisor & 0xffffffffffffffff)
                val += "{0:04b}".format(self.ma_start_opcode & 0xf)
                val += "{0:03b}".format(self.ma_start_funct3 & 0x7)
                return val

        def get(self) -> int: return int(self.__bin__(), base=2)

        def size(self) -> int: return 135
```

```
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
```

FBOX:
ADD SUB SP

```
**************************************************
MODULE: mk_fpu_add_sub_sp_instance, CLOCK: CLK, RESET: RST_N
**************************************************
VERILOG METHOD: send
**********
INPUT_NAME BITWIDTH
send_operands 67
EN_send 1
OUTPUT_NAME BITWIDTH
RDY_send 1
**********
VERILOG METHOD: receive
**********
INPUT_NAME BITWIDTH
OUTPUT_NAME BITWIDTH
receive 38
RDY_receive 1
**********
**************************************************
```

```
@dataclass(init=False)
class receive_out:
        receive: int = 0
        def set(self, val):
                self.receive = (val.integer >> 0) & 0x3fffffffff
                return

        def __bin__(self):
                val = ""
                val += "{0:038b}".format(self.receive & 0x3fffffffff)
                return val

        def get(self) -> int: return int(self.__bin__(), base=2)

        def size(self) -> int: return 38
```

```
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
```

References: https://gitlab.com/shaktiproject/core-py-verif https://www.cocotb.org/