VHDLwhiz.com

VHDL registers UART test interface generator - User manual

Version:	1.0.3
Date:	March 12, 2024
Author:	Jonas Julian Jensen
Product URL:	https://vhdlwhiz.com/product/vhdl-registers-uart-test-interface-generator
Contact email:	jonas@vhdlwhiz.com

This document describes using VHDLwhiz's UART test interface generator to produce a custom VHDL module and Python script for reading and writing FPGA register values.

Table of content

License

The MIT license covers the source code's copyright requirements and terms of use. Refer to the *LICENSE.txt* file in the Zip file for details.

Changelog

These changes refer to the project files, and this document is updated accordingly.

Version	**Remarks**
1.0.0	Initial release
1.0.1	Fixed missing «self» reference bug when importing as uart_regs.py as a Python module. Changed write failed printout to exception to avoid printing to the console when running as an imported module.
1.0.2	Fix for Vivado [Synth 8-248] error when there are no out mode regs.
1.0.3	Fix Vivado Linter warning: Register has enable driven by synchronous reset

Description

This document describes the following files and folders:

- **gen_uart_regs.py**

- **generated/uart_regs.vhd**
- **generated/uart_regs.py**
- **generated/instantiation_template.vho**

- **rtl/uart_regs_backend.vhd**
- **rtl/uart_rx.vhd**
- **rtl/uart_tx.vhd**

- **demo/lattice_icestick/**
- **demo/xilinx_arty_a7_35/**
- **demo/xilinx_arty_s7_50/**

The *gen_uart_regs.py* script and supporting VHDL files in this project allow you to generate custom interfaces for reading and writing FPGA register values of various types and widths using UART.

You can use the generated VHDL module and Python script to read from or write to any number of registers in your design. The UART accessible registers can have the types `std_logic`, `std_logic_vector`, `signed`, or `unsigned`.

You can decide on the precise composition of input and output registers and types when generating the output files using the *gen_uart_regs.py* script.

The Python scripts were created partially with the help of the ChatGPT artificial intelligence tool, while the VHDL code is handcrafted.

Requirements

The scripts in this project must be run through a Python 3 interpreter and the Pyserial package must be installed.

You can install Pyserial through Pip using this command:

```
pip install pyserial
```

Protocol

The VHDL files and Python script uses a data framing protocol with four control characters:

Name	Value	Comment
READ_REQ	0x0A	Command from the host to the FPGA to initiate a write sequence to send all registers back over UART
START_WRITE	0x0B	Marks the beginning of a write sequence in either direction
END_WRITE	0x0C	Marks the end of a write sequence in either direction
ESCAPE	0x0D	Escape character used for escaping any of the control words, including the ESCAPE character itself, when they appear as data between the START_WRITE and END_WRITE markers.

Any unescaped READ_REQ byte sent to the FPGA is an instruction to send all of its UART-accessible registers (inputs and outputs) back to the host over UART. This command is usually only issued by the *uart_regs.py* script.

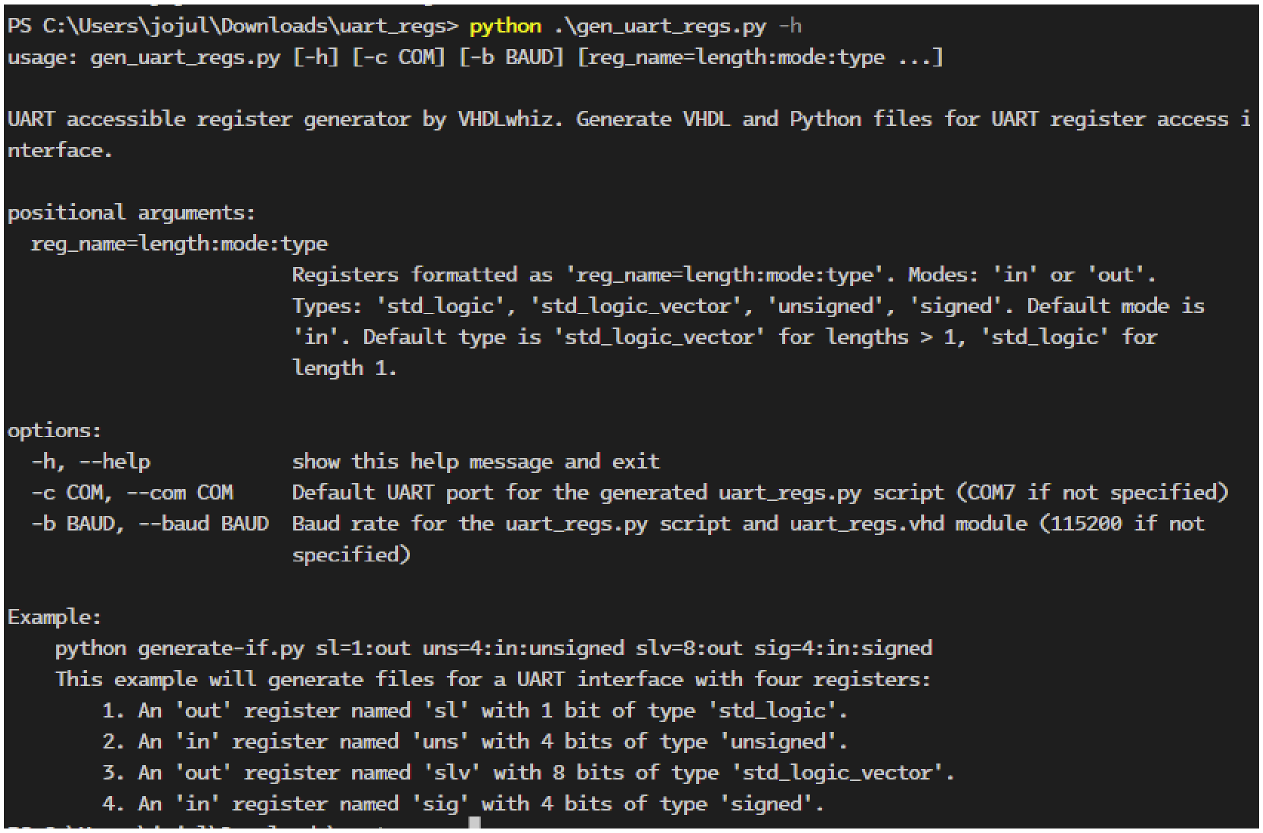
Upon receiving this command, the FPGA will respond by sending the content of all registers back to the host. First, the input signals, then the output signals. If their lengths don't add up to a multiple of 8 bits, the lower bits of the last byte will be padded zeros.

A write sequence always starts with the START_WRITE byte and ends with the END_WRITE byte. Any bytes between those are considered to be data bytes. If any data bytes have the same value as a control character, the data byte must be escaped. This means sending an extra ESCAPE character before the data byte to indicate that it's actually data.

If an unescaped START_WRITE arrives anywhere in the stream of bytes, it is considered the start of a write sequence. The *uart_regs_backend* module uses this information to resynchronize in case the communication gets out of sync.

gen_uart_regs.py

This is the script you must start with to generate the interface. Below is a screenshot of the help menu that you can get by running: `python gen_uart_regs.py -h`



```
PS C:\Users\jojul\Downloads\uart_regs> python .\gen_uart_regs.py -h
usage: gen_uart_regs.py [-h] [-c COM] [-b BAUD] [reg_name=length:mode:type ...]

UART accessible register generator by VHDLwhiz. Generate VHDL and Python files for UART register access i
nterface.

positional arguments:
  reg_name=length:mode:type
                        Registers formatted as 'reg_name=length:mode:type'. Modes: 'in' or 'out'.
                        Types: 'std_logic', 'std_logic_vector', 'unsigned', 'signed'. Default mode is
                        'in'. Default type is 'std_logic_vector' for lengths > 1, 'std_logic' for
                        length 1.

options:
  -h, --help            show this help message and exit
  -c COM, --com COM     Default UART port for the generated uart_regs.py script (COM7 if not specified)
  -b BAUD, --baud BAUD  Baud rate for the uart_regs.py script and uart_regs.vhd module (115200 if not
                        specified)

Example:
    python generate-if.py sl=1:out uns=4:in:unsigned slv=8:out sig=4:in:signed
    This example will generate files for a UART interface with four registers:
        1. An 'out' register named 'sl' with 1 bit of type 'std_logic'.
        2. An 'in' register named 'uns' with 4 bits of type 'unsigned'.
        3. An 'out' register named 'slv' with 8 bits of type 'std_logic_vector'.
        4. An 'in' register named 'sig' with 4 bits of type 'signed'.
```

To generate a custom interface, you must run the script with each of your desired UART controllable registers listed as arguments. The available types are `std_logic`, `std_logic_vector`, `unsigned`, and `signed`.

The default mode (direction) is `in` and the default type is `std_logic_vector` unless the register is of length: 1. Then, it will default to `std_logic`.

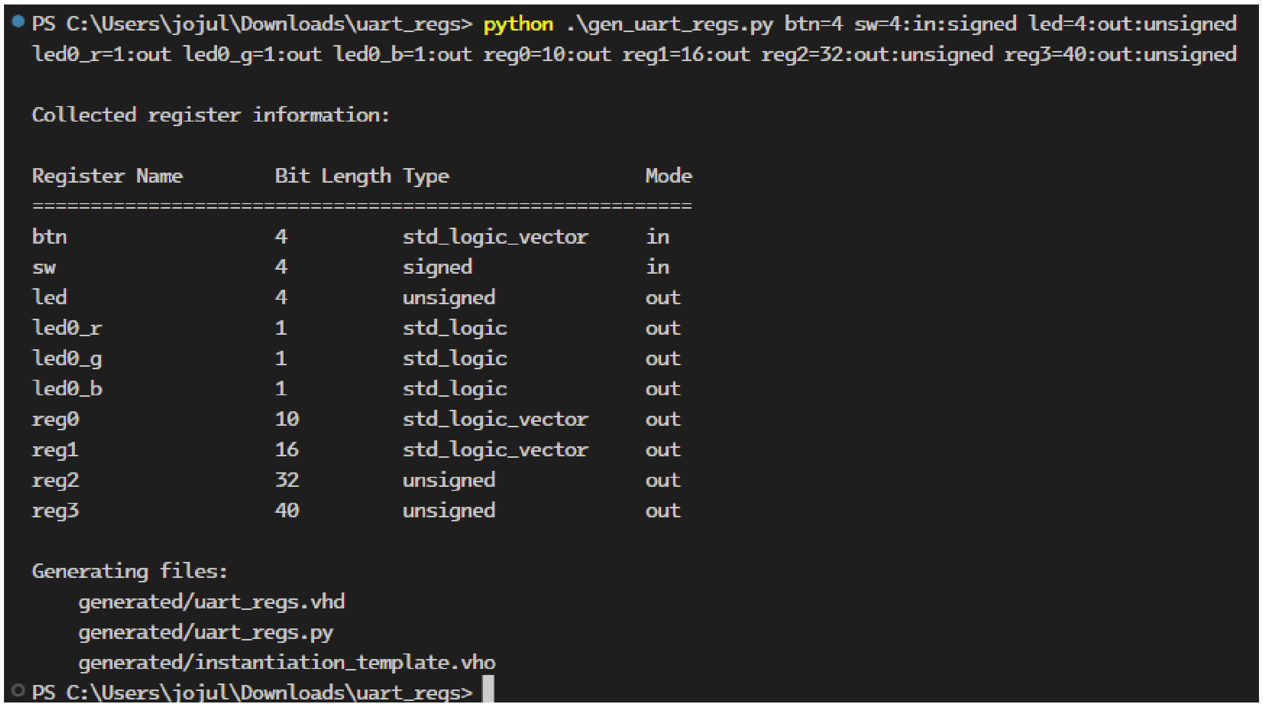
Thus, if you want to create a `std_logic` input signal, you can use any of these arguments:

```
my_sl=1
my_sl=1:in
my_sl=1:in:std_logic
```

All of the above variants will result in the script generating this UART-accessible signal:

```
my_slv : in std_logic;
```

Let's run the script with arguments to generate an interface with several registers of different directions, lengths, and types:



```
PS C:\Users\jojul\Downloads\uart_regs> python .\gen_uart_regs.py btn=4 sw=4:in:signed led=4:out:unsigned
led0_r=1:out led0_g=1:out led0_b=1:out reg0=10:out reg1=16:out reg2=32:out:unsigned reg3=40:out:unsigned

Collected register information:

Register Name        Bit Length Type              Mode
=========================================================
btn                  4          std_logic_vector    in
sw                   4          signed              in
led                  4          unsigned            out
led0_r               1          std_logic           out
led0_g               1          std_logic           out
led0_b               1          std_logic           out
reg0                 10         std_logic_vector    out
reg1                 16         std_logic_vector    out
reg2                 32         unsigned            out
reg3                 40         unsigned            out

Generating files:
    generated/uart_regs.vhd
    generated/uart_regs.py
    generated/instantiation_template.vho
PS C:\Users\jojul\Downloads\uart_regs>
```

Generated files

A successful run of the *gen_uart_regs.py* script will produce an output folder named *generated* with the three files listed below. If they already exist, they will be overwritten.

- **generated/uart_regs.vhd**
- **generated/uart_regs.py**
- **generated/instantiation_template.vho**

uart_regs.vhd

This is the custom interface module generated by the script. You need to instantiate it in your design, where it can access the registers you want to control using UART.

Everything above the "-- UART accessible registers" section will be identical for every *uart_regs* module, while the composition of port signals below that line depends on the arguments given to the generator script.

The listing below shows the entity for the *uart_regs* module resulting from the generate command example shown in the [gen_uart_regs.py]() section.

```vhdl
entity uart_regs is
  generic (
    clk_hz : positive;
    baud_rate : positive := 115200
  );
  port (
    clk : in std_logic;
    rst : in std_logic;

    uart_rx : in std_logic;
    uart_tx : out std_logic;

    -- UART accessible registers
    btn : in std_logic_vector(3 downto 0);
    sw : in signed(3 downto 0);
    led : out unsigned(3 downto 0);
    led0_r : out std_logic;
    led0_g : out std_logic;
    led0_b : out std_logic;
    reg0 : out std_logic_vector(9 downto 0);
    reg1 : out std_logic_vector(15 downto 0);
    reg2 : out unsigned(31 downto 0);
    reg3 : out unsigned(39 downto 0)
  );
end uart_regs;
```

You do not need to synchronize the `uart_rx` signal, as that's handled in the *uart_rx.* module.

When the module receives a read request, it will capture the values of all input and output signals within the current clock cycle. The instantaneous snapshot is then sent to the host over UART.

When a write happens, all output registers are updated with the new values within the same clock cycle. It is not possible to change output signal values individually.

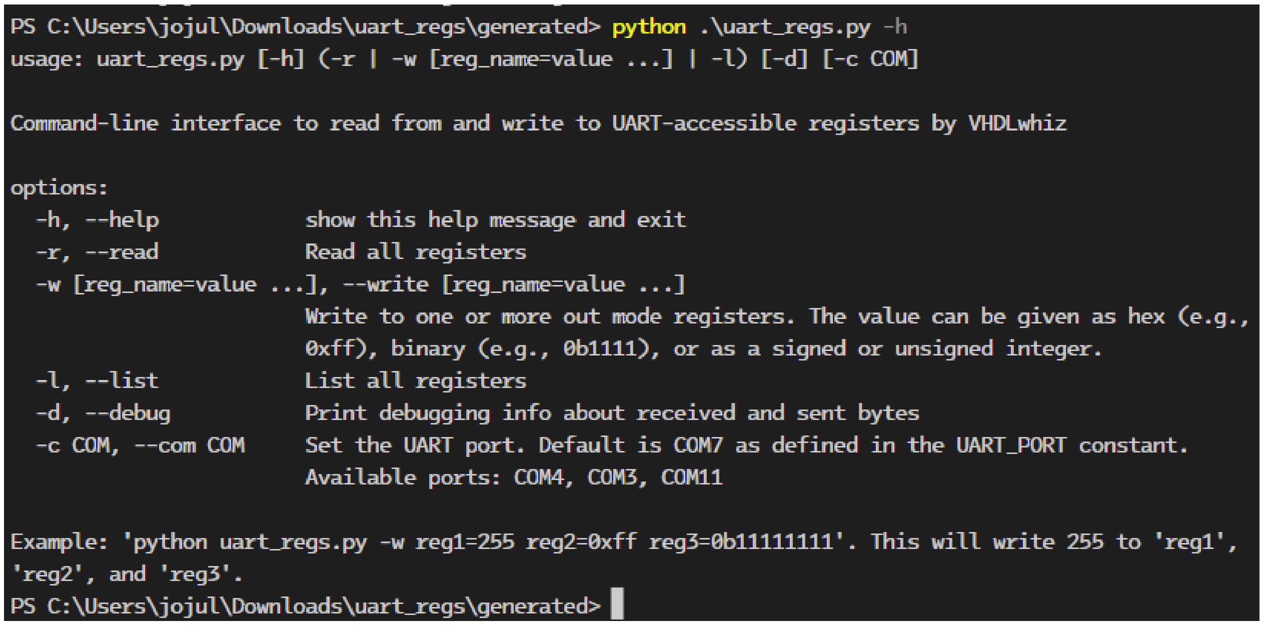
However, the *uart_regs.py* script allows the user to update only selected outputs by first reading back the current values of all registers. It then writes back all values, including the updated ones.

uart_regs.py

The *generated/uart_regs.py* file is generated together with the *uart_regs* VHDL module and contains the custom register information in the header of the file. With this script, you can read from or write to your custom registers with ease.

Help menu

Type `python uart_regs.py -h` to print the help menu:



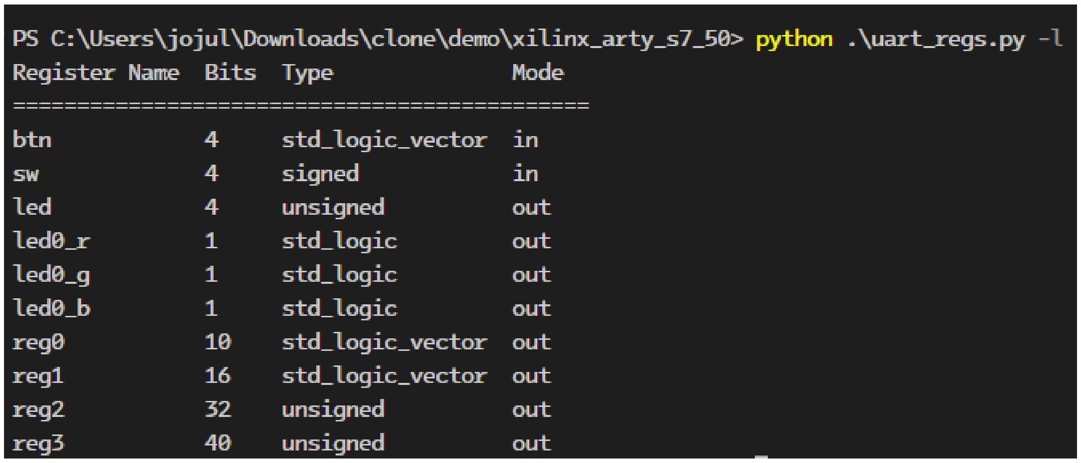
```
PS C:\Users\jojul\Downloads\uart_regs\generated> python .\uart_regs.py -h
usage: uart_regs.py [-h] (-r | -w [reg_name=value ...] | -l) [-d] [-c COM]

Command-line interface to read from and write to UART-accessible registers by VHDLwhiz

options:
  -h, --help            show this help message and exit
  -r, --read            Read all registers
  -w [reg_name=value ...], --write [reg_name=value ...]
                        Write to one or more out mode registers. The value can be given as hex (e.g.,
                        0xff), binary (e.g., 0b1111), or as a signed or unsigned integer.
  -l, --list            List all registers
  -d, --debug           Print debugging info about received and sent bytes
  -c COM, --com COM     Set the UART port. Default is COM7 as defined in the UART_PORT constant.
                        Available ports: COM4, COM3, COM11

Example: 'python uart_regs.py -w reg1=255 reg2=0xff reg3=0b11111111'. This will write 255 to 'reg1',
'reg2', and 'reg3'.
PS C:\Users\jojul\Downloads\uart_regs\generated> █
```

Setting the UART port

The script has options to set the UART port using the `-c` switch. This works on Windows and Linux. Set it to one of the available ports listed in the help menu. To set a default port, you can also edit the `UART_PORT` variable in the *uart_regs.py* script.

Listing registers

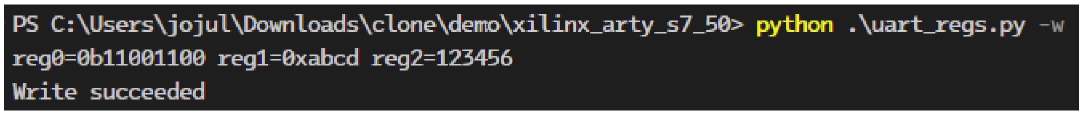
Information about the register mapping is placed in the header of the *uart_regs.py* script by the *gen_uart_regs.py* script. You can list the available registers with the `-l` switch, as seen below. This is a local command and will not interact with the target FPGA.



```
PS C:\Users\jojul\Downloads\clone\demo\xilinx_arty_s7_50> python .\uart_regs.py -l
Register Name  Bits  Type              Mode
=============================================
btn            4     std_logic_vector  in
sw             4     signed            in
led            4     unsigned          out
led0_r         1     std_logic         out
led0_g         1     std_logic         out
led0_b         1     std_logic         out
reg0           10    std_logic_vector  out
reg1           16    std_logic_vector  out
reg2           32    unsigned          out
reg3           40    unsigned          out
```

Writing to registers

You can write to any of the `out` mode registers by using the `-w` switch. Supply the register name followed by "=" and the value given as a binary, hexadecimal, or decimal value, as shown below.



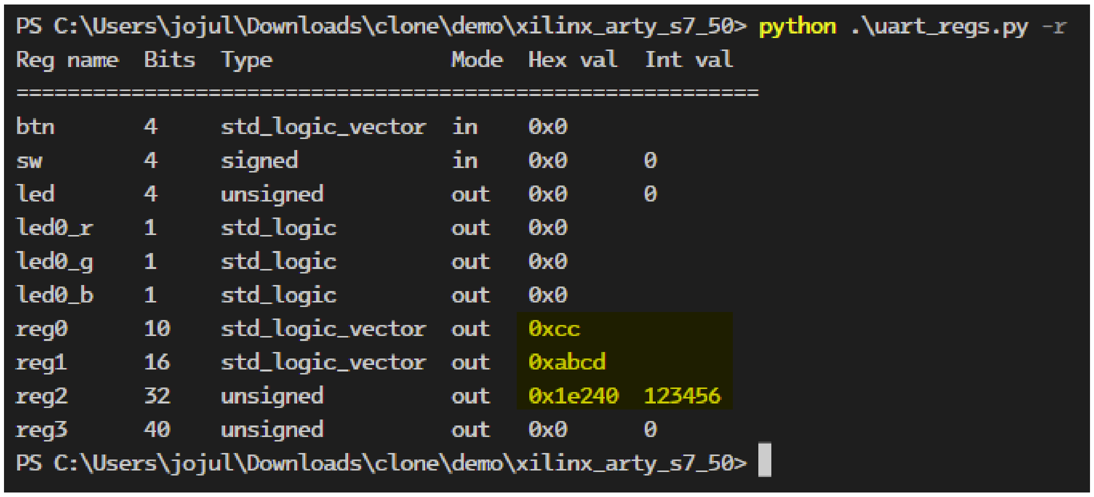
```
PS C:\Users\jojul\Downloads\clone\demo\xilinx_arty_s7_50> python .\uart_regs.py -w
reg0=0b11001100 reg1=0xabcd reg2=123456
Write succeeded
```

Note that the VHDL implementation requires the script to write all output registers simultaneously. Therefore, if you don't specify a complete set of output registers, the script will first perform a read from the target FPGA and then use those values for the missing ones. The result will be that only the specified registers change.

When you perform a write, all specified registers will change during the same clock cycle, not as soon as they are received over UART.

Reading registers

Use the `-r` switch to read all register values, as shown below. The values marked in yellow are the ones we changed in the previous write example.

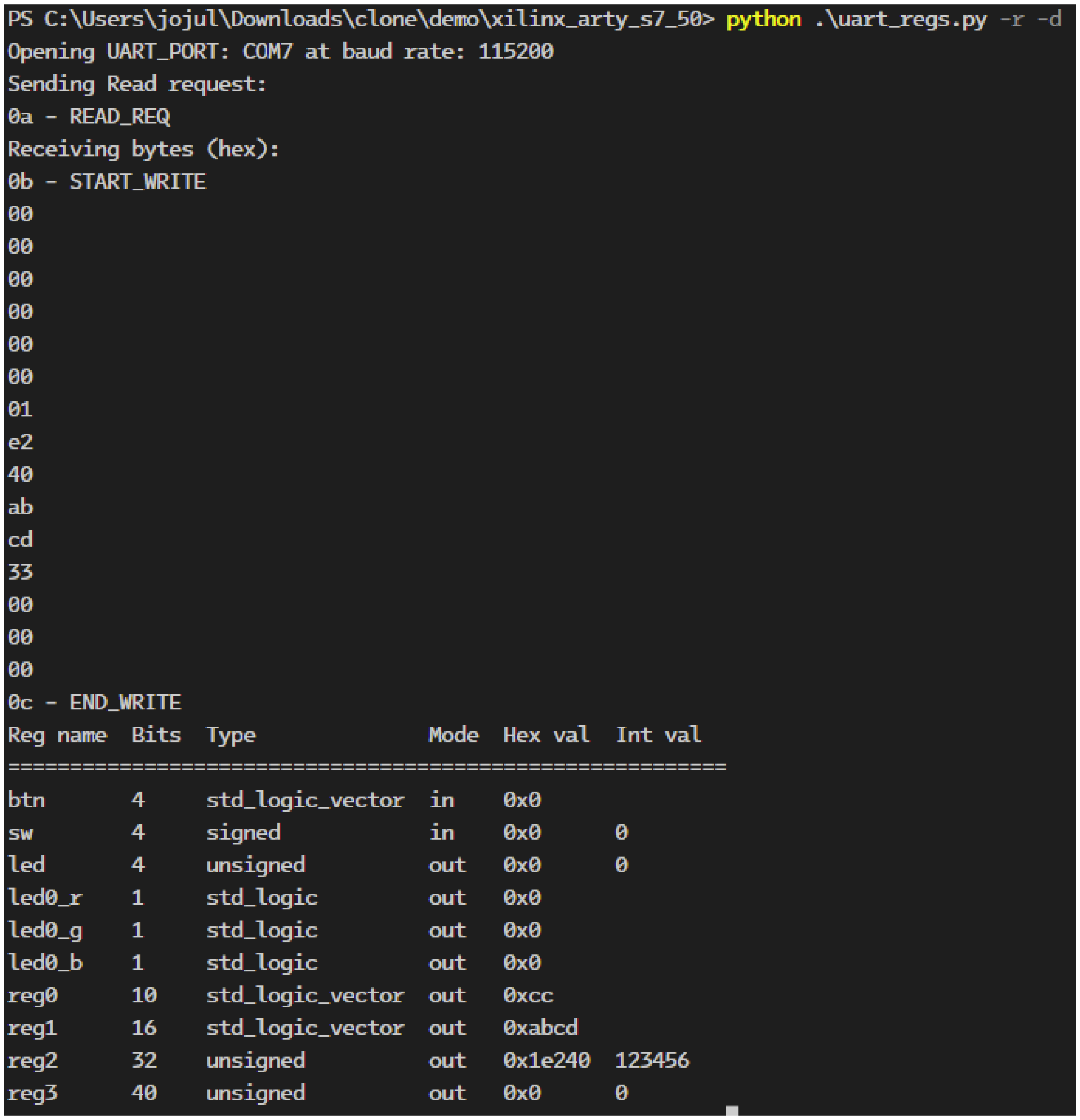


```
PS C:\Users\jojul\Downloads\clone\demo\xilinx_arty_s7_50> python .\uart_regs.py -r
Reg name  Bits  Type              Mode  Hex val  Int val
=========================================================
btn       4     std_logic_vector  in    0x0
sw        4     signed            in    0x0      0
led       4     unsigned          out   0x0      0
led0_r    1     std_logic         out   0x0
led0_g    1     std_logic         out   0x0
led0_b    1     std_logic         out   0x0
reg0      10    std_logic_vector  out   0xcc
reg1      16    std_logic_vector  out   0xabcd
reg2      32    unsigned          out   0x1e240  123456
reg3      40    unsigned          out   0x0      0
PS C:\Users\jojul\Downloads\clone\demo\xilinx_arty_s7_50>
```

Every read shows an instantaneous snapshot of all input and output registers. They are all sampled during the same clock cycle.

Debugging

Use the `-d` switch with any of the other switches if you need to debug the communication protocol. Then, the script will print out all sent and received bytes and tag them if they are control characters, as shown below.



```
PS C:\Users\jojul\Downloads\clone\demo\xilinx_arty_s7_50> python .\uart_regs.py -r -d
Opening UART_PORT: COM7 at baud rate: 115200
Sending Read request:
0a - READ_REQ
Receiving bytes (hex):
0b - START_WRITE
00
00
00
00
00
00
01
e2
40
ab
cd
33
00
00
00
0c - END_WRITE
Reg name  Bits  Type              Mode  Hex val  Int val
=========================================================
btn       4     std_logic_vector  in    0x0
sw        4     signed            in    0x0      0
led       4     unsigned          out   0x0      0
led0_r    1     std_logic         out   0x0
led0_g    1     std_logic         out   0x0
led0_b    1     std_logic         out   0x0
reg0      10    std_logic_vector  out   0xcc
reg1      16    std_logic_vector  out   0xabcd
reg2      32    unsigned          out   0x1e240  123456
reg3      40    unsigned          out   0x0      0
```

Using the interface in other Python scripts

The *uart_regs.py* script contains a `UartRegs` class that you can easily use as the communication interface in other custom Python scripts. Simply import the class, create an object of it, and start using the methods, as shown below.

```
uart_regs = UartRegs(port=args.com, baud_rate=BAUD_RATE,
debug=args.debug)
my_dict = uart_regs.read_regs()
```

Refer to the docstrings in the Python code for method and descriptions and return value types.

instantiation_template.vho

The instantiation template is generated along with the *uart_regs* module for your convenience. To save coding time, you can copy the module instantiation and signal declarations into your design.

```
-- UART register accessor by VHDLwhiz

  constant clk_hz : integer := 100e6;

  signal clk : std_logic;
  signal rst : std_logic;
  signal uart_to_dut : std_logic;
  signal uart_from_dut : std_logic;

  -- UART accessible registers
  signal btn : std_logic_vector(3 downto 0);
  signal sw : signed(3 downto 0);
  signal led : unsigned(3 downto 0);
  signal led0_r : std_logic;
  signal led0_g : std_logic;
  signal led0_b : std_logic;
  signal reg0 : std_logic_vector(9 downto 0);
  signal reg1 : std_logic_vector(15 downto 0);
  signal reg2 : unsigned(31 downto 0);
  signal reg3 : unsigned(39 downto 0);

begin
```

```
  -- Generated with the command:
  -- python .\gen_uart_regs.py btn=4 sw=4:in:signed led=4:out:unsigned
led0_r=1:out led0_g=1:out led0_b=1:out reg0=10:out reg1=16:out
reg2=32:out:unsigned reg3=40:out:unsigned
  UART_REGS_INST : entity work.uart_regs(rtl)
  generic map (
    clk_hz => clk_hz
  )
  port map (
    clk => clk,
    rst => rst,
    uart_rx => uart_rx,
    uart_tx => uart_tx,
    btn => btn,
    sw => sw,
    led => led,
    led0_r => led0_r,
    led0_g => led0_g,
    led0_b => led0_b,
    reg0 => reg0,
    reg1 => reg1,
    reg2 => reg2,
    reg3 => reg3
  );
```

Static RTL files

You need to include the following files in your VHDL project so that they are compiled into the same library as the *uart_regs* module:

- **rtl/uart_regs_backend.vhd**
- **rtl/uart_rx.vhd**
- **rtl/uart_tx.vhd**

The *uart_regs_backend* module implements the finite-state machines that clock in and out the register data. It uses the *uart_rx* and *uart_tx* modules to handle the UART communication with the host.

Demo projects

There are three demo projects included in the Zip file. They let you control the peripherals on the different boards as well as a few larger, internal registers.

The demo folders include pre-generated *uart_regs.vhd* and *uart_regs.py* files made specifically for those designs.

Lattice iCEstick

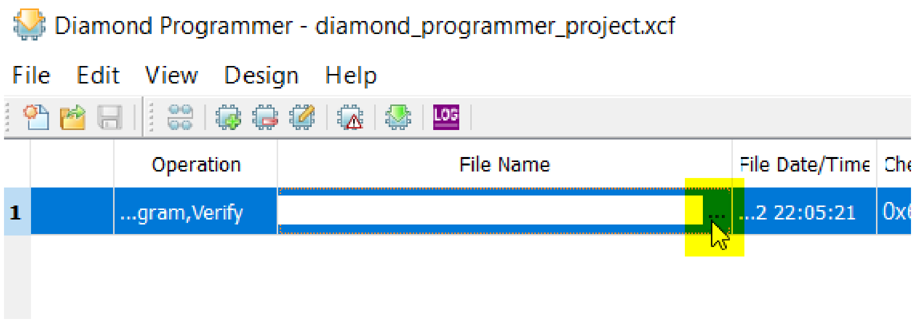
The *demo/icecube2_icestick* folder contains a register access demo implementation for the Lattice iCEstick FPGA board.

To run through the implementation process, open the *demo/lattice_icestick/icecube2_proj/uart_regs_sbt.project* file in the Lattice iCEcube2 design software.

After loading the project in the iCEcube2 GUI, click **Tools→Run All** to generate the programming bitmap file.

You can use the Lattice Diamond Programmer Standalone tool to configure the FPGA with the generated bitmap file. When Diamond Programmer opens, click **Open an existing programmer project** in the welcome dialog box.

Select project file found in the Zip: *demo/lattice_icestick/diamond_programmer_project.xcf* and click OK.



After the project loads, click the three dots in the **File Name** column, as shown above. Browse to select the bitmap file that you generated in iCEcube2:

demo/lattice_icestick/icecube2_proj/uart_regs_Implmnt/sbt/outputs/bitmap/top_icestick_bitmap.bin

Finally, with the iCEstick board plugged into a USB port on your computer, select **Design→Program** to program the SPI flash and configure the FPGA.

You can now proceed to read and write registers by using the *demo/lattice_icestick/uart_regs.py* script as described in the uart_regs.py section.

Xilinx Digilent Arty A7-35T

You can find the demo implementation for the Artix-7 35T Arty FPGA evaluation kit in the *demo/arty_a7_35* folder.

Open Vivado and navigate to the extracted files using the Tcl console found at the bottom of the GUI interface. Type this command to enter the demo project folder:

```
cd <zip_content>/demo/arty_a7_35/vivado_proj/
```

Execute the *create_vivado_proj.tcl* Tcl script to regenerate the Vivado project:

```
source ./create_vivado_proj.tcl
```

Click **Generate Bitstream** in the sidebar to run through all the implementation steps and generate the programming bitstream file.

Finally, click **Open Hardware Manager** and program the FPGA through the GUI.

You can now proceed to read and write registers by using the *demo/arty_a7_35/uart_regs.py* script as described in the uart_regs.py section.

Xilinx Digilent Arty S7-50

You can find the demo implementation for the Arty S7: Spartan-7 FPGA development board in the *demo/arty_s7_50* folder.

Open Vivado and navigate to the extracted files using the Tcl console found at the bottom of the GUI interface. Type this command to enter the demo project folder:

```
    cd <zip_content>/demo/arty_s7_50/vivado_proj/
```

Execute the *create_vivado_proj.tcl* Tcl script to regenerate the Vivado project:

```
    source ./create_vivado_proj.tcl
```

Click **Generate Bitstream** in the sidebar to run through all the implementation steps and generate the programming bitstream file.

Finally, click **Open Hardware Manager** and program the FPGA through the GUI.

You can now proceed to read and write registers by using the *demo/arty_s7_50/uart_regs.py* script as described in the uart_regs.py section.

Implementation

There are no specific implementation requirements.

Constraints

No specific timing constraints are needed for this design because the UART interface is slow and treated as an asynchronous interface.

The `uart_rx` input to the *uart_regs* module is synchronized within the *uart_rx* module. Thus, it doesn't need to be synchronized in the top-level module.

Known issues

- You may need to reset the module before it can be used, depending on whether your FPGA architecture supports default register values.