Register Interface

2025-04-28 · 11 min · Justin Rajewski | Suggest Changes

▶ Table of Contents

This tutorial will introduce you to the *Register Interface* component and how you can use it to easily implement complex interfaces in your designs.

Introduction

Sometimes dealing with an interface, like the serial interface, directly can become quite complicated depending on what you are trying to implement. The *Register Interface* component provides an abstraction on top of a basic interface (usually the USB<->Serial port, but it doesn't have to be).

The abstraction it provides is a *register* based one. The FPGA acts as a peripheral and responds to read and write requests targeted at specific addresses.

You've likely seen a register interface many times if you've worked with anything over I2C or SPI. Datasheets will list the registers you can read and write and what the bits in each one does.

In the most basic case, the thing you are reading and writing is actually a register. However, it doesn't *have* to be. Instead, you can respond to the reads/writes however

you see fit.

For example, you might choose to respond to a read at address 8 with data from a FIFO. Each time address 8 is read, you remove a value from the FIFO and return it. This is a common case when you are implementing something that collects data and needs to stream it out to the host.

Enough theoretical, let's jump into the actual interface.

The Register Interface Component

In Alchitry Labs, open the *Component Library* and add the *Register Interface* (under *Interfaces*) to your project. You will also likely want to add the *UART Rx* and *UART Tx* components if you plan to use the interface with the built-in USB port.

Here's the full module.

```
34  global Register {
35      struct request {                                    // request devi
36          new_cmd,                                        // 1 = new comm
37          write,                                          // 1 = write, 0
38          address[32],                                    // address to r
39          data[32]                                        // data to writ
40      }
41      struct response {                                   // response inp
42          data[32],                                       // data read fr
43          drdy                                            // read data va
44      }
45  }
46
47  module register_interface #(
48      CLK_FREQ ~ 100000000 : CLK_FREQ > 0
49  )(
50      input clk,                                      // clock
```

I'm not going to dive into too much detail on *how* it all works but mostly go over w it does and how to use it. It is essentially just a large FSM (see the FSM tutorial for background). Check out The API section for info on the actual protocol.

For now, we will dive into using it.

Controlling the LEDs

To get your feet wet, we will use the interface to control the LEDs on the board.

First, we need to instantiate the modules in the `alchitry_top` module.

```
16          .rst(rst) {
17              dff led_reg[8]
18              #CLK_FREQ(100_000_000) {
19                  register_interface reg
20                  #BAUD(1_000_000) {
21                      uart_rx rx
22                      uart_tx tx
23                  }
24              }
25          }
```

Here I used connection blocks for the parameters `CLK_FREQ` and `BAUD` to easily set them for all the modules that need them. This is optional, but I like to do it this way to guarantee they are all the same.

I also added the `dff led_reg` that we will use to save the value written to the LEDs.

In the `always` block, we can connect up the modules.

```
28      always {
29          reset_cond.in = ~rst_n  // input raw inverted reset signal
```

```
30          rst = reset_cond.out    // conditioned reset
31
32          led = led_reg.q
33
34          usb_tx = tx.tx
35          rx.rx = usb_rx
36
37          reg.rx_data = rx.data
38          reg.new_rx_data = rx.new_data
39          tx.data = reg.tx_data
40          tx.new_data = reg.new_tx_data
41          reg.tx_busy = tx.busy
42          tx.block = 0 // never block
43      }
```

We now need to deal with incoming requests. This is done through the `reg_out` and `reg_in` ports of the `register_interface` module. These ports use structs to bundle a bunch of signals together.

The `struct` for each one is defined in the `global` block in the same file. These are available anywhere in your design by using the designations `Register.request` and `Register.response`.

We can connect up these signals to respond to read/write requests to address 0 with the value of the LEDs.

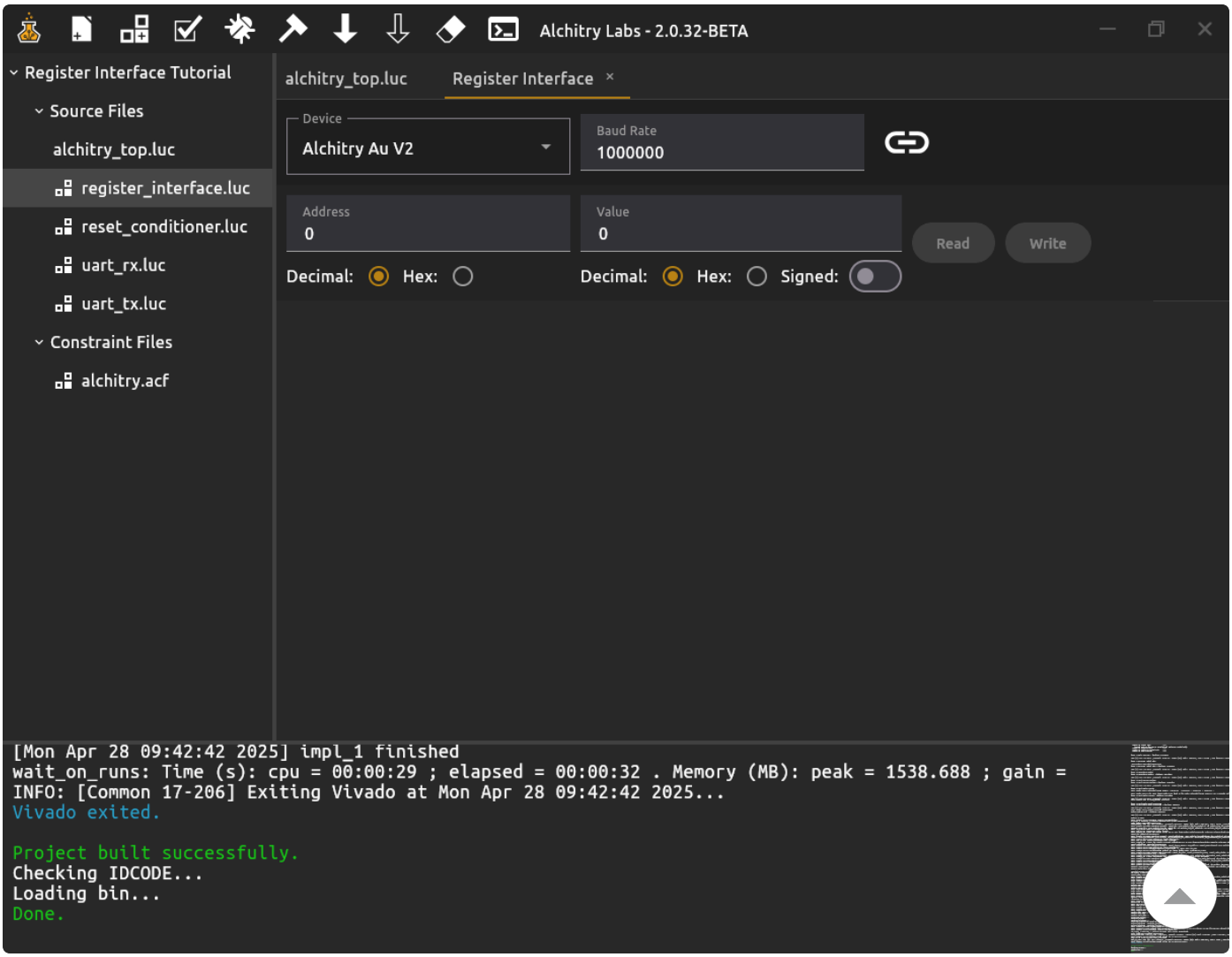
```
44          // default value
45          reg.reg_in = <Register.response>(.data(32bx), .drdy(0))
46
47          if (reg.reg_out.new_cmd) {
48              if (reg.reg_out.write) { // write
49                  if (reg.reg_out.address == 0) { // address matches
50                      led_reg.d = reg.reg_out.data[7:0] // update th
51                  }
52              } else { // read
53                  if (reg.reg_out.address == 0) { // address matches
```

```
54                  reg.reg_in.drdy = 1 // data ready
55                  reg.reg_in.data = led_reg.q // return the value
56              }
57          }
```

Notice on line 45 I used the struct literal syntax to assign a constant value to every element in the struct.

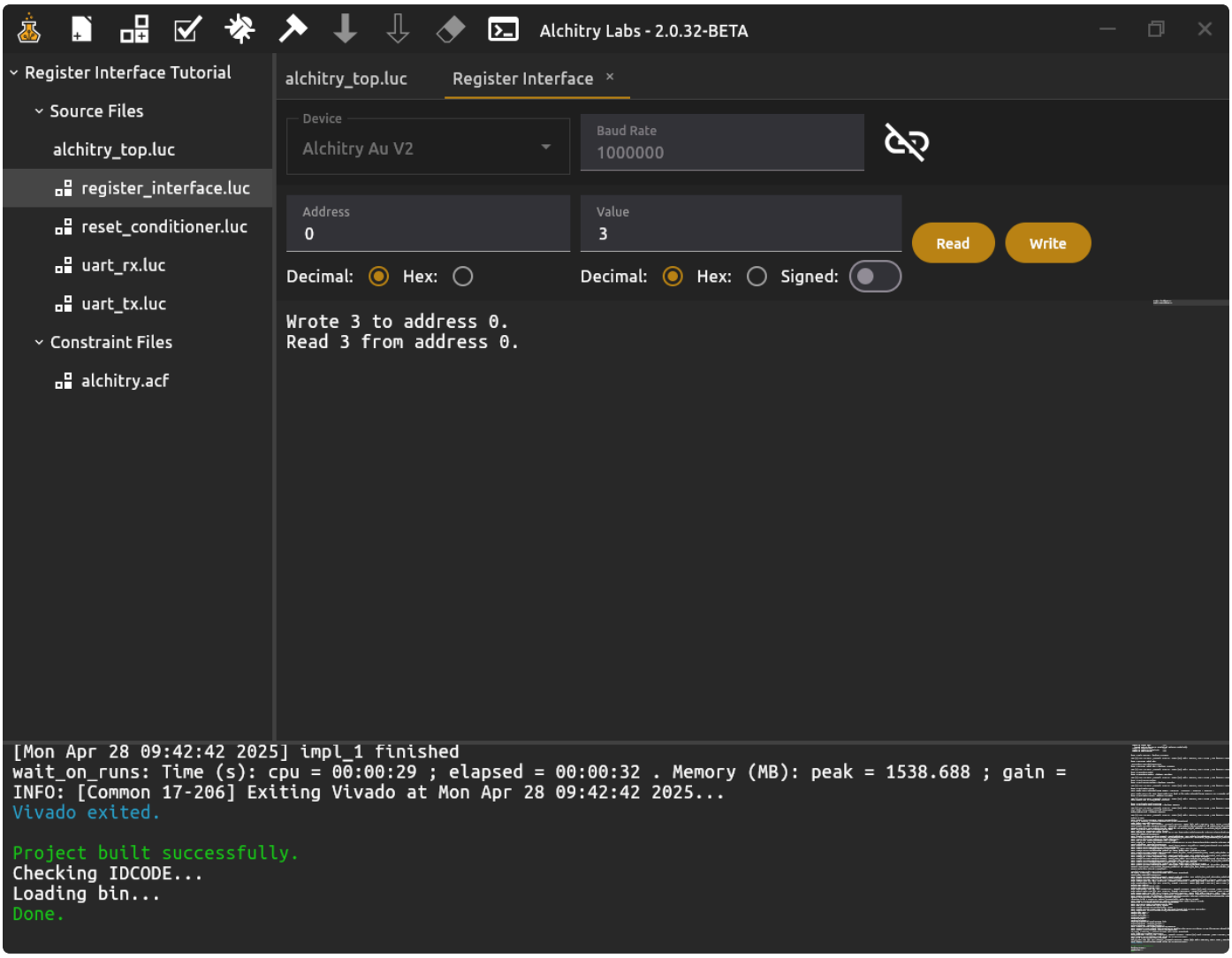
You can now build your project and load it onto your board.

In Alchitry Labs, you can now open the *Register Interface* tool to read/write registers. Click the *Tools* icon (looks like a terminal) and select *Register Interface*. A new tab will open.



Click the chain icon to connect. Make sure the baud rate matches what you set in your code. In the above example I used 1M baud which is what Alchitry Labs defaults to.

You can enter a value, like `3`, into the *Value* box and click *Write* to write it to address 0. If you then click *Read* it'll read it back.



You should see your board's first two LEDs turn on. Try writing a few other values just to make sure it's working as expected.

Don't forget to disconnect from the board when you are done. You won't be able to program it while connected.

If you try to read from an address other than 0, you'll get an error along the lines "Read failed: Read 0 but expected 4 bytes!" This happens because we only set the

`drdy` flag to `1` when address `0` was read and ignored every other request. The tool only waits a short time after sending a read request for a result.

To expand the address you respond to, you would typically use a `case` statement instead of the `if` .

```
47          if (reg.reg_out.new_cmd) {
48              if (reg.reg_out.write) { // write
49                  case (reg.reg_out.address) {
50                      0: led_reg.d = reg.reg_out.data[7:0] // update th
51                      1: // do something with address 1
52                      2: // do something with address 2
53                      3: // do something with address 3
54                  }
55              } else { // read
56                  case (reg.reg_out.address) {  // address matches
57                      0:
58                          reg.reg_in.drdy = 1 // data ready
59                          reg.reg_in.data = led_reg.q // return the val
60                      1:
61                          // respond to address 1
62                      2:
63                          // respond to address 2
64                      3:
65                          // respond to address 3
66                  }
67              }
68          }
```

Each address targets a 32bit value. In the LED example, we are throwing away the top 24 bits, but you could use them for something else.

Here's the full `alchitry_top` module.

```
module alchitry_top (
    input clk,              // 100MHz clock
    input rst_n,            // reset button (active low)
    output led[8],          // 8 user controllable LEDs
    input usb_rx,           // USB->Serial input
    output usb_tx           // USB->Serial output
) {

    sig rst                 // reset signal

    .clk(clk) {
        // The reset conditioner is used to synchronize the reset
        // clock. This ensures the entire FPGA comes out of reset
        reset_conditioner reset_cond

        .rst(rst) {
            dff led_reg[8]
            #CLK_FREQ(100_000_000) {
```

The API

This built in register interface tool is helpful, but manually entering everything isn't always practical. Here we will go over how the protocol works for issuing read and write requests so that you can use this in your own applications.

Every request starts with 5 bytes being sent. The first byte is the command byte and the next four are the address (32 bits = 4 bytes) sent with the least significant byte first.

In many cases you will want to read or write to many consecutive addresses, or perhaps the same address many times. It would be inefficient to have to issue the entire command each time so the command byte contains info for consecutive c multiple read/write requests in one.

The MSB (bit 7) of the command byte specifies if the command is a read (0) or write (1). The next bit (bit 6), specifies if consecutive addresses should be read/written (1) or if the same address should be read/written multiple times (0). The 6 LSBs (bits 5-0) represent how many read/write requests should be generated. Note that the number of requests will always be 1 more than the value of these bits. That means if you want to read or write a single address, they should be set to 0. Setting them to 1 will generate two read or write requests.

If you send a write request, after sending the command byte and address, you continue sending the data to be written. Data is always sent as four bytes per write request and the least significant byte should be sent first.

For reads, after sending the command byte and address, you simply wait for the data to be returned. Data is returned in least significant byte order. Note that you may not always receive all the data you ask for if there is an issue with the FPGA design (i.e. the requested data is never presented, like in our LED example).

Let's take a look at an example.

If you want to write to addresses 5, 6, and 7, you could issue the following request. 0xC2 (1100 0010), 0x05, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00. This would write 1 to address 5, 2 to address 6, and 3 to address 7. If you changed the command byte to 0x82 (1000 0010) you would write 1 to address 5, 2 to address 5, and then 3 to address 5 (in that order).

Issuing a read follows the exact same format except the data bytes are received instead of sent.

A single command can generate up to 64 read/write requests. If you need to read/write more, you need to issue separate commands.

After a short period, the interface times out and resets to the `IDLE` state. This ensures that a bad command doesn't permanently mess up the stream.

« PREV
Hello YOUR_NAME_HERE

NEXT »
DDR3 Memory