Home » Tutorials » Your First FPGA Project

Your First FPGA Project

2024-09-16 · 19 min · Justin Rajewski | Suggest Changes

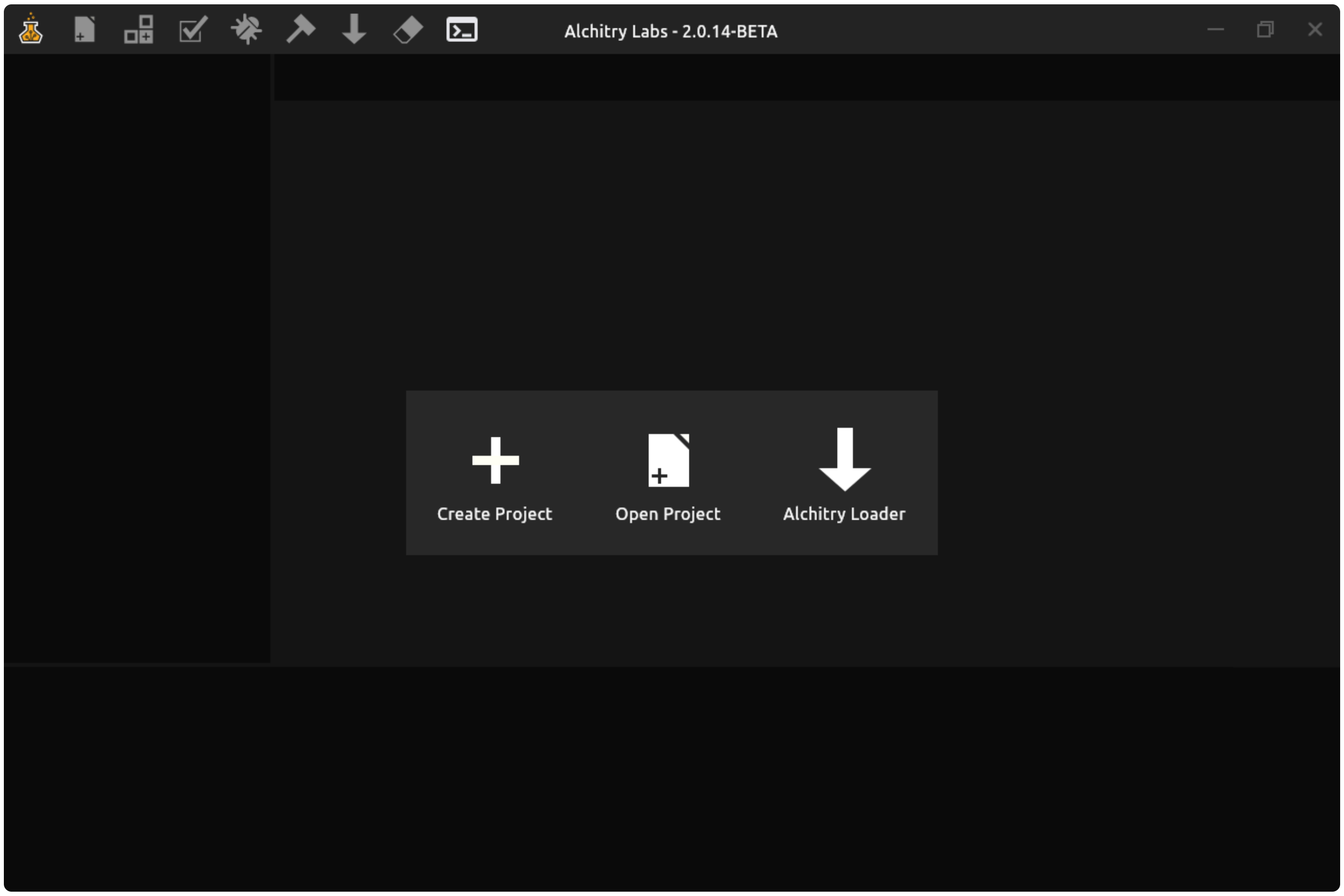
▶ Table of Contents

This tutorial will walk you through creating your first project in Alchitry Labs and making the onboard LED light up when you press the reset button.

Before diving in, you will need to have the correct tools installed. See the setup tutorial for details.

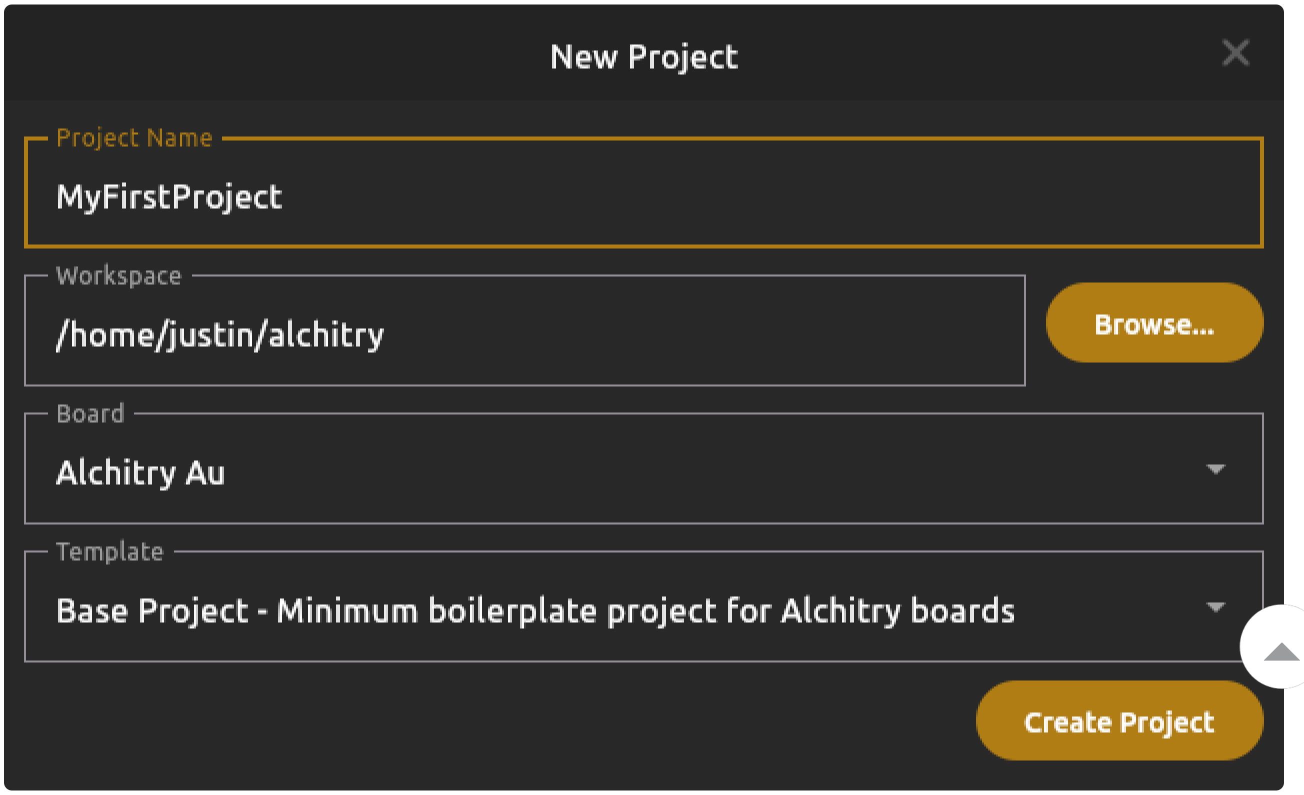
Creating a New Project

Fire up Alchitry Labs. You should be greeted with the following.



Since we will be creating a new project, click the *Create Project* button.

You should then get a new dialog window for creating your project.


New Project

Project Name: MyFirstProject

Workspace: /home/justin/alchitry

Board: Alchitry Au

Template: Base Project - Minimum boilerplate project for Alchitry boards

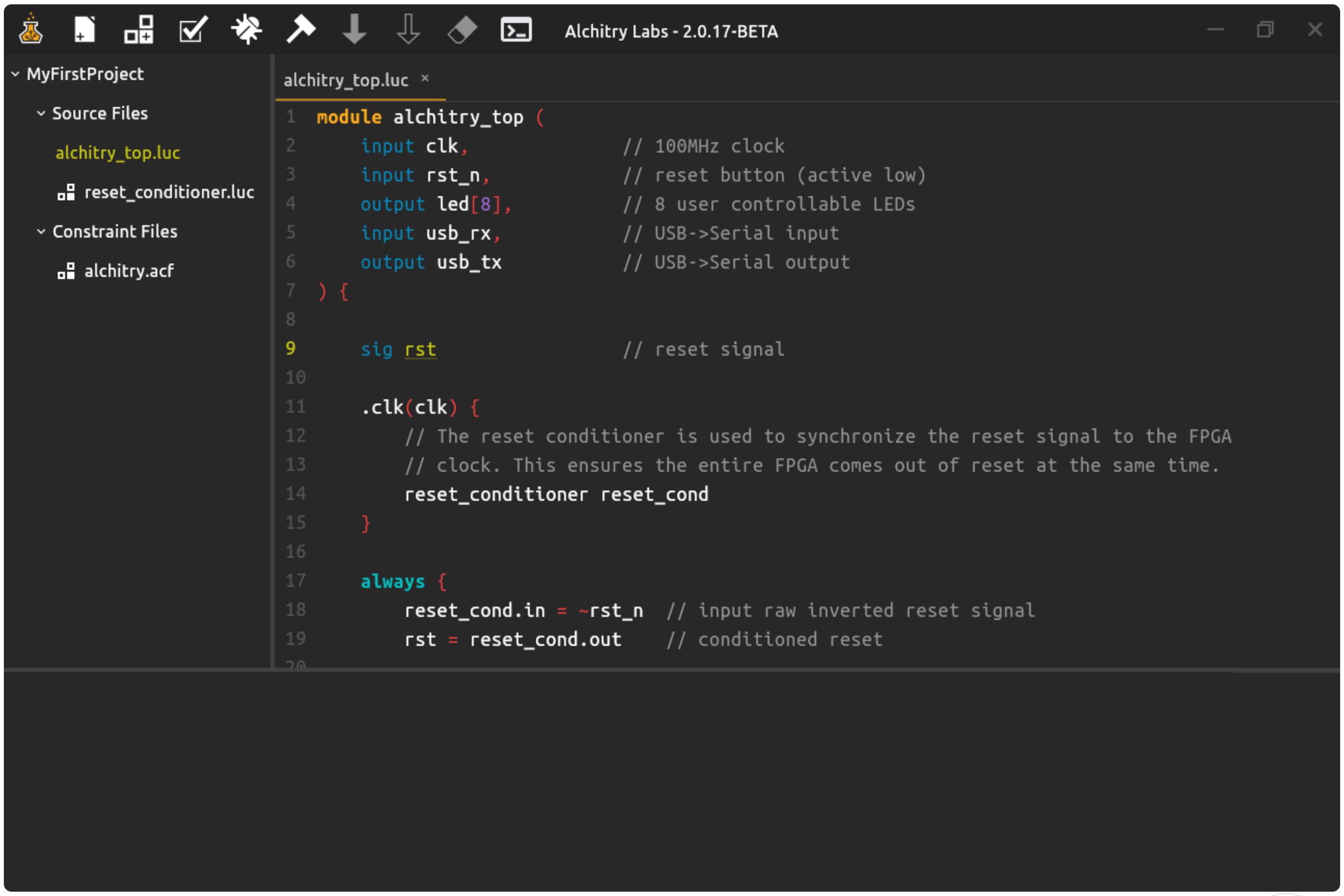
You can enter whatever you like for the *Project Name*. The *Workspace* is the parent folder you want to store all your Alchitry projects in. I'm on a Linux system so your path will look different on Windows.

For *Board*, select the Alchitry board you have from the dropdown. I'll be using the Au, but everything in this tutorial is the same for any board.

Leave *Template* set to *Base Project*. This is the minimal starting point but feel free to take a look at the other options in the dropdown for future reference.

Once we are done with this tutorial, the project should be the same as the *Button to LED* template.

Finally, click *Create Project*. The project should be created and *alchitryTop.luc* should open automatically.



```
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 signal to the FPGA
        // clock. This ensures the entire FPGA comes out of reset at the same time.
        reset_conditioner reset_cond
    }

    always {
        reset_cond.in = ~rst_n  // input raw inverted reset signal
        rst = reset_cond.out    // conditioned reset
```

On the left hand side, you will see your project's tree. We have a couple of source files and one constraint file. Source files contain your project's modules and constraint files specify physical constraints (most commonly what signal connects to which pin).

For now, look at the already open *alchitryTop.luc*.

Modules

Modules are the basic building block used in any HDL (**H**ardware **D**escription **L**anguage) and Lucid is no exception. They are similar to how functions are the building block of any programming language.

Modules are blocks with some number of ports (inputs and outputs) that can be connected other to other modules or external signals. They can also contain other modules.

Remember that we are working with hardware. This means that each module you use will take up some space inside the FPGA. If you use the same module twice, you will have two independent copies of that circuit.

Ports

Let's take a look at the module declaration for `alchitry_top`.

```
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
) {
```

Here we have a list of ports. The direction of each port is specified by the `input` or `output` keyword. There is also the `inout` keyword for bidirectional ports.

Bidirectional signals can't exist inside an FPGA so `inout` signals can only be used as external ports.

All the signals here are 1-bit wide except `led` which is declared as an 8-bit wide array with the `[8]` syntax. The reason `led` is 8 bits wide is because the Alchitry boards have 8 LEDs on them. One bit for each LED.

The `alchitry_top` module is what's known as the *top-level* module in our design. That means that its ports connect to physical pin on the FPGA.

What pins these actually connect to is defined in the *alchitry.acf* constraint file.

```
clock clk CLOCK 100MHz

pin rst_n RESET

pin led[0] LED0
pin led[1] LED1
pin led[2] LED2
pin led[3] LED3
pin led[4] LED4
pin led[5] LED5
pin led[6] LED6
pin led[7] LED7

pin usb_rx USB_RX
pin usb_tx USB_TX
```

The syntax here is `pin SIGNAL_NAME PYSICAL_PIN` . The `PHYSICAL_PIN` name is the name on the Alchitry board and is translated by the tools to the unique FPGA's pin name. Most of the time you write a constraint file yourself, you will be using names like `A2` which would be pin 2 on bank A. See the Br page for common pin names. The Br reference card is also helpful for general pin capabilities.

Note that the `led` signal has to have each of this individual bits assigned individually using the array indexing syntax `[n]` where n is the index.

You probably noticed that the first constraint wasn't a `pin` constraint but was rather a `clock` constraint. These work the same as a `pin` constraint but have an additional parameter for how fast the clock is. Alchitry boards all have an onboard 100MHz clock. If you don't know what a clock is or why this is important, don't worry. It'll all be covered in a later tutorial.

Always Blocks

Let's jump back to the `alchitry_top` module.

The first line declares a `sig` named `rst`. The `sig` type (short for signal) acts like a wire to connect things. They don't actually *store* any value and really just act like an alias for what is assigned to them.

Next a `reset_conditioner` module is instantiated. This is covered in the next tutorial, and we are going to skip it for now. Jump down to the `always` block.

Always blocks are a concept, that again, exists in all HDLs. They are where you describe the vast majority of your design's logic. The block get its name because it is *always* happening.

Inside the block, you describe some behavior by reading/writing signals and performing various operations. The tools will take that description and replicate it in hardware. It is important to remember that you are creating a description of behavior. You are **not** writing instructions as if it was a program even though it looks a lot like that.

Let's take a look at the `always` block in `alchitry_top`.

```
always {
    reset_cond.in = ~rst_n  // input raw inverted reset signal
    rst = reset_cond.out    // conditioned reset

    led = 8h00              // turn LEDs off
    usb_tx = usb_rx          // echo the serial data
}
```

When the tools evaluate an always block, statements that appear lower in the block have high priority than previous statements. This makes it feel like things are running sequentially from top to bottom but that is just an abstraction to make it easier to describe complex logic.

To make this clear, lets looks at an example.

```
always {
  led = 8h00              // turn LEDs off
  led = 8hFF              // turn LEDs on
}
```

I know we haven't covered numbers yet, but `8h00` just means an 8-bit wide hex number of 00. `8hFF` is an 8-bit wide hex number of FF (all 1s).

The first line assigns all bits of `led` to 0 and the second line sets them all to 1.

What would you expect to happen if we built this and loaded it onto the board? If you have a programming background, you may be tempted to think that the LEDs would continuously turn on and off. You're not in Kansas anymore Dorthy, this isn't programming and that's not what happens. Remember, there is no processor to run code (that is unless you explicitly make one, like a boss). When the tools see this block, they completely ignore the first line. This is because the second line has higher priory. If you were to synthesize this design, the tools would hard-wire `led` to `8hFF` (all 1s).

Back to our design, we are assigning four signals a value. Every output on a module *must* have a value assigned to it at all times. Since this is the minimal boilerplate project template, these are just some reasonable defaults.

Representing Values

A value is made of one or more bits. The number of bits a value has is known as its width. There are a few ways to specify a value, some use an implied width while others allow you to explicitly set a width. If you are unfamiliar with binary or hexadecimal please read the Encodings Tutorial before continuing.

The first way to represent a value is to just type a decimal number. For example, `9` .

Sometimes it's easier to specify a number with a different radix than 10 (the implied default). Lucid supports three different radix, 10 (decimal), 16 (hexadecimal), and 2 (binary). To specify the radix, prepend d, h, or b to specify decimal, hexadecimal, or binary respectively. For example, `h_ff` has the decimal value `255` and `b100` has the decimal value `4` . If you don't append a radix indicator, decimal is assumed.

It is important to remember that all number will be represented as bits in your circuit. When you specify a number this way, the width of the number will be the minimum number of bits required to represent that value for decimal. For binary, it is simply the number of digits and for hexadecimal it is four times the number of digits. For example, the value `7` will be three bits wide (`b111`), `b1010` will be four bits wide, and `h_acb` will be 12 bits wide.

Sometimes you need more control over the exact width of a number. In those cases you can specify the number of bits by prepending the number of bits and radix to the value. For example, `4d2` will be the value `2` but using 4 bits instead of the minimum 2 (binary value `b0010` instead of `b10`). You must specify the radix when specifying the width to separate the width from the value.

If you specify a width smaller than the minimum number of bits required, the number will drop the most significant bits. When this happens you will get a warning.

Z and X

When you specify a decimal number, all the bits in your value will be either 0 or 1. However, each bit can actually be one of four values, 0, 1, x, or z. These can be used in binary and hex constants.

The values of 0 and 1 are fairly self-explanatory, it just means the bit is high (1) or low (0). The value of x means **don't care**. It means you want to assign a value, but you really don't care if it is 1 or 0. This is useful for the synthesizer because your circuit may be simpler in one of the cases, and it gives the tools the freedom to choose. During simulation, x also means unknown. Z means that the bit is **high-impedance** or tri-stated. This means that it is effectively disconnected.

Note that FPGAs can't realize high-impedance signals internally. The only time you should use z is for `output` or `inout` ports of the top-level module or a module feeding directly to a top-level port.

Back to our always block, the first two lines connect the input `rst_n` to the input of the `reset_cond` module. Modules can be nested which makes it possible to reuse them and helps make your project manageable. This is all covered later in more detail so don't get hung up over this yet. The only important thing to know about these two lines, is that the `rst_n` signal is active low (0 when the button is pressed, 1 otherwise) while the `rst` signal is active high.

The next line assigns the `led` output to all zeros. This turns off all the LEDs.

The last line connects the serial input pin to the serial output pin effectively echoing anything the FPGA receives over the serial port. We could have also tied this pin to 1 to disable it.

Looking at this always block, we can see there are no redundant assignments (like in our led on/off example). That means these signals will literally be connected to these values.

Connecting the Button

We are going to modify the module to connect the reset button to the first LED so that when you push the button the LED turns on.

To do this we need to modify line 21, where `led` is assigned.

```
led = c{7h00, rst}      // connect rst to the first LED
```

The output `led` is an 8 bit array. That means when you assign it a value you need to provide an 8 bit array. However, the signal `rst` is a single bit wide. To compensate for this we use the concatenation operator.

To concatenate multiple arrays into one, you can use the concatenation operator, `c{ x, y, z }`. Here the arrays (or single bit values) `x`, `y`, and `z` will be concatenated to form a larger array.

In our code we are concatenating the constant `7h0` with `rst`. The constant here is seven zeros. Since we just need a bunch of zeros, the radix doesn't really matter, and we could have used `7b0`, or `7d0`, for binary or decimal respectively.

Note that values are zero padded if the specified width is larger than the size required to store the value. For example, `4b11` would the same as `4b0011`.

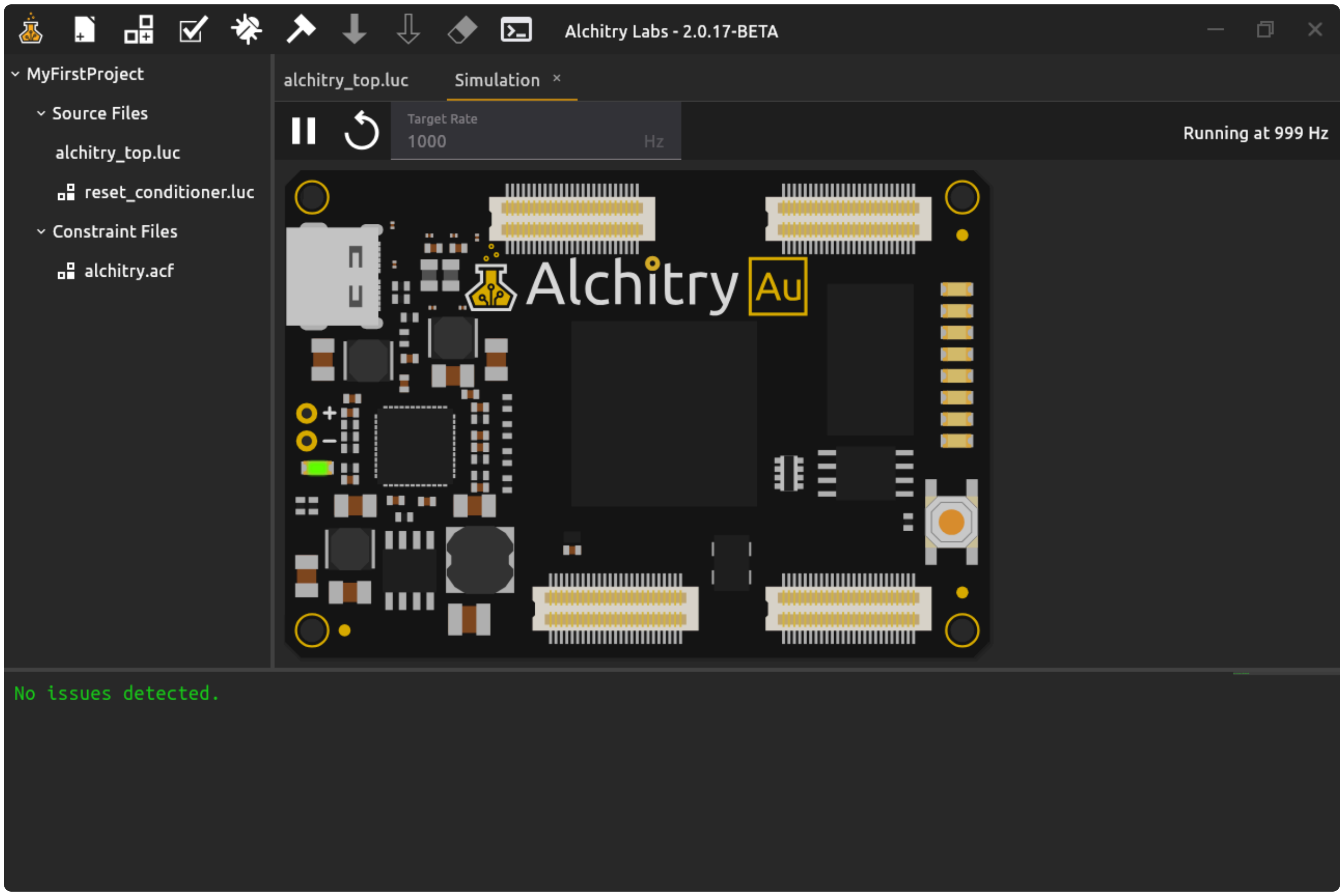
If you don't care about how many bits a values takes up, you don't have to specify it and the minimum number of bits will be used. However, when you are concatenating values, you should specify a width to make it obvious how big the array will be.

Simulate the Project

Alchitry Labs V2 has a built-in simulator we can use to test out our changes.

Click the little bug icon in the toolbar and the simulator tab will open.

You should see a virtual version of your board.



```
No issues detected.
```

If you click on the reset button in the bottom right, the top LED should turn on.

Build the Project

We can also build and run the project on real hardware.

Go ahead and click the little hammer icon in the toolbar to build the project. You may need to first specify where you installed the build tools. If you installed them to their default location, they should be detected automatically.

For the Au, click the Alchitry logo and go to *Settings->Set Vivado Location*. Select the *Xilinx/Vivado/YEAR.MONTH* folder for your Vivado install (where YEAR.MONTH is the version number).

On Windows, the default is *C:\Xilinx\Vivado\YEAR.MONTH*.

On Linux, */opt/Xilinx/Vivado/YEAR.MONTH* is assumed.

For the Cu, if you're using the open source tools, make sure that is selected. Click the Alchitry logo and check *Settings->Cu Toolchain->Yosys (open source)* is selected.

If you want to use i_cecube2, make sure it is selected in *Settings->Cu Toolchain*. You will also have to set the location of the *lscc/iCEcube2* folder with *Settings->Set i_cecube2 Location* and the license file with *Settings->Set i_cecube2 License Location*.

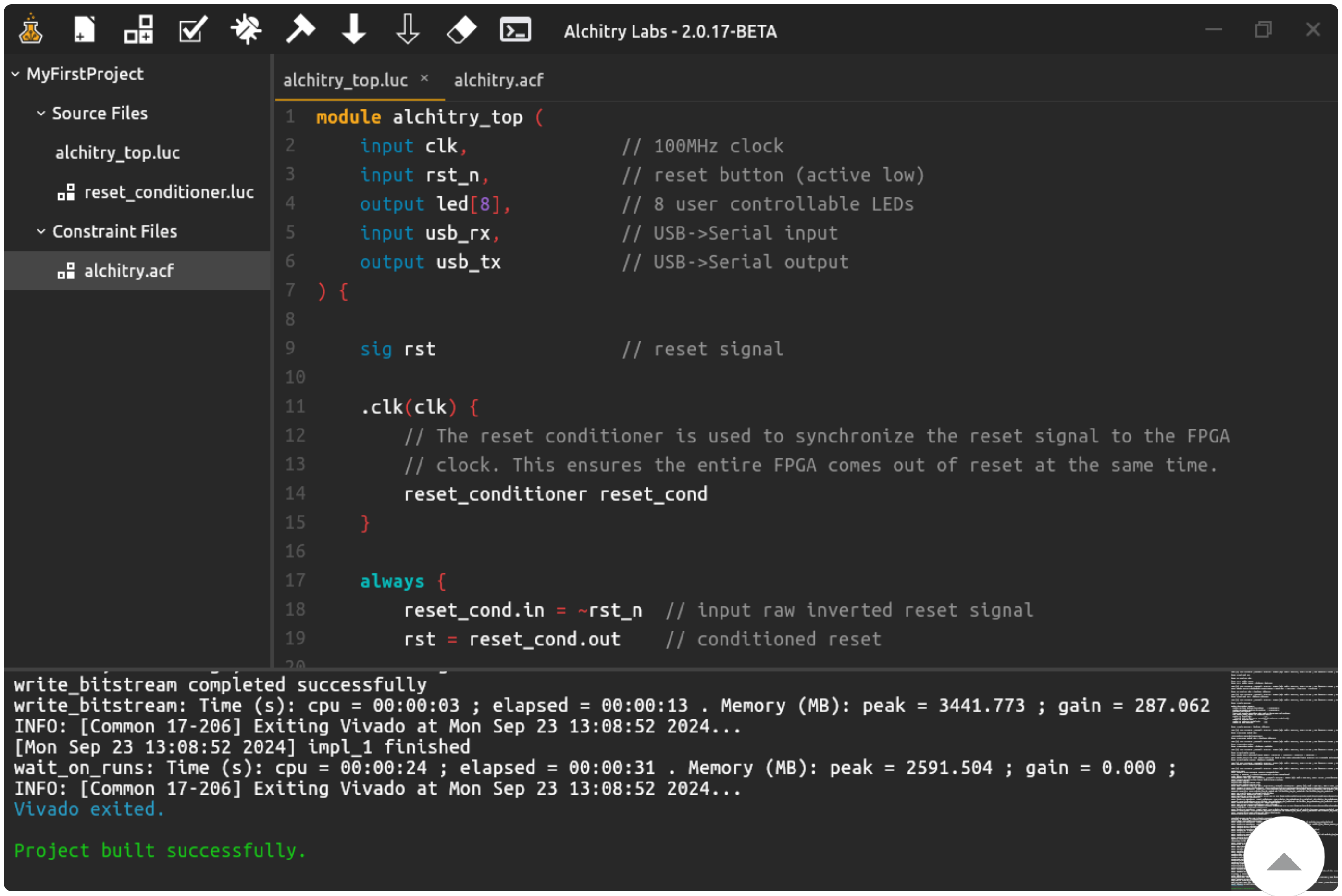
On Windows, the default is *C:\lscc\i_cecube2.2020.12*.

On Linux, *~/lscc/i_cecube2.2020.12* is assumed.

There is no default for the license file location.

You should now be able to build the project with whatever tools you selected.

When the project is building, you will see a bunch of text dumped by the build tools to the console. Just wait for it to finish and it should look like this.



```
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 signal to the FPGA
        // clock. This ensures the entire FPGA comes out of reset at the same time.
        reset_conditioner reset_cond
    }

    always {
        reset_cond.in = ~rst_n  // input raw inverted reset signal
        rst = reset_cond.out    // conditioned reset
```

```
write_bitstream completed successfully
write_bitstream: Time (s): cpu = 00:00:03 ; elapsed = 00:00:13 . Memory (MB): peak = 3441.773 ; gain = 287.062
INFO: [Common 17-206] Exiting Vivado at Mon Sep 23 13:08:52 2024...
[Mon Sep 23 13:08:52 2024] impl_1 finished
wait_on_runs: Time (s): cpu = 00:00:24 ; elapsed = 00:00:31 . Memory (MB): peak = 2591.504 ; gain = 0.000 ;
INFO: [Common 17-206] Exiting Vivado at Mon Sep 23 13:08:52 2024...
Vivado exited.

Project built successfully.
```

The important line here is where it says *Project built successfully.* This means that Alchitry Labs was able to find the .bin file after the build tools exited. If you ever get a red message telling you the bin file couldn't be found, you can look through the logs for the reason.

Loading the Project

Make sure your board is plugged into your computer. The arrow and eraser icons in the toolbar should be enabled if your board was detected.

Alchitry Labs is constantly looking for new boards.

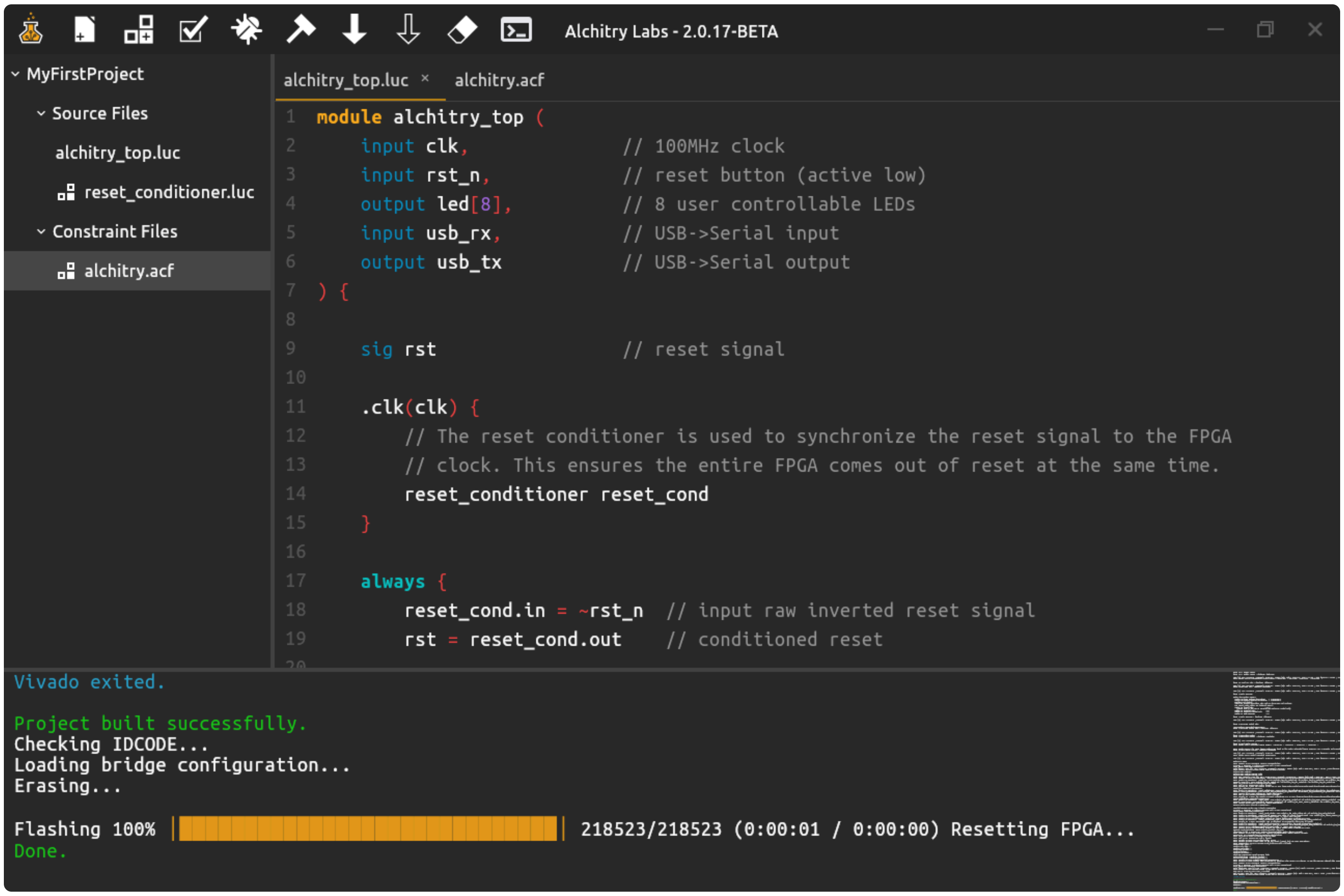
If you have an Au, you will see two arrow icons. The Cu will show only one.

In both cases, the solid arrow means the design will be loaded into persistent memory. Inside the FPGA, the configuration is stored in RAM and is lost when power is lost. When it gets power again, it will read the on-board FLASH and automatically configure itself.

With the Au, you have the option via the hollow arrow to program the FPGA directly. This is very fast but the configuration will be lost upon power cycling.

The eraser icon will clear the FLASH causing the FPGA to do nothing on power-up.

Click the solid arrow to program the board.



```
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 signal to the FPGA
        // clock. This ensures the entire FPGA comes out of reset at the same time.
        reset_conditioner reset_cond
    }

    always {
        reset_cond.in = ~rst_n  // input raw inverted reset signal
        rst = reset_cond.out    // conditioned reset
```

```
Vivado exited.

Project built successfully.
Checking IDCODE...
Loading bridge configuration...
Erasing...

Flashing 100% |████████████████████████| 218523/218523 (0:00:01 / 0:00:00) Resetting FPGA...
Done.
```

Just like the simulation, pushing the reset button should cause the first (top-most) LED to turn on.

You may have noticed the *Done* LED on the left side of your board. This LED connects to the *Done* signal on the FPGA and will turn on when the FPGA is correctly configured. It is *not* a power LED. If you erase the FLASH using the eraser button then power cycle your board, the *Done* LED will not turn on until you program it again.

This signal is also broken out on bank D. It is an open drain and can be used to synchronize the start of multiple FPGAs. The FPGA will only start when the signal is high.

Some Note on Hardware

When you press the button, how long does it take for the LED to turn on? If this [illegible] a processor instead of an FPGA, the processor would be in a loop reading the button state and turning the LED on or off based on that state. The amount of time between pressing it and when the LED turns on would vary depending on what code the processor was executing and how long it is until it gets back to reading the button and turning the LED on. As you add more code to your loop, this time and variation gets bigger.

However, an FPGA is different. With this design (note I said design and not code), the button input is directly connected to the LED output. You can imagine a physical wire bridging the input to the output inside the FPGA. In reality, it's not a wire but a set of switches (multiplexers) that are set to route the signal directly to the output. Well, this is only partially true since the `reset_conditioner` is there which does some stuff to clean up the reset signal.

Since the signal doesn't have to wait for a processor to read it, it will travel as fast as possible through the silicon to light the LED. This is almost instant (again, forget about the `reset_conditioner`)! The best part is that if you wire the button the LED then go on to create some crazy design with the rest of the FPGA, this will not slow down. This is because the circuits will operate independently as they both simply *exist*. It is this parallelism where FPGAs get their real power.

Duplication

What if we want all the LEDs to turn on and off with the press of the button instead of just one?

Well we could do it using concatenation just like before by replacing line 21 with the following.

```
21  led = c{rst,rst,rst,rst,rst,rst,rst,rst}
```

However, there is a better way! This is where the array duplication syntax comes in handy. The syntax is `M x{ A }` . Here M is the number of times to duplicate A. T means we can make line 21 look like this.

```
21  led = 8x{rst}
```

Much cleaner! This does exactly the same thing as before, but requires a lot less typing.

Array Indexing

There is an alternative way to write the code where we only want one LED to light. This is by assigning the parts of the `led` array separately.

```
21  led[7:1] = 7h0          // turn these LEDs off
22  led[0] = rst            // connect rst to led[0]
```

On line 21, the bit selector `[7:1]` is used to select the bits 7 through 1 of the `led` array. These seven bits are set to 0.

On line 22, `[0]` is used to select the single bit, 0, and set it to `rst` .

There are two ways to select a group of bits. The first one (and most common) is the one used above where you specify the start and stop bits (inclusive) explicitly. The other way is to specify a start bit and the total number of bits to include above or below the start bit.

Line 21 could be rewritten as any of the following.

```
led[7:1] = 7b0  // select bits 7-1
led[7-:7] = 7b0 // starting from 7 and going down, select 7 bits
led[1+:7] = 7b0 // starting from 1 and going up, select 7 bits
```

The benefit of using the start-width syntax is the width is guaranteed to be constant. This means you can use a signal to specify the start index. This will be covered in a later tutorial.

Always Blocks

Due to the nature of always blocks, you could also write the LED assignment as follows.

```
led = 8b0                // turn the LEDs off
led[0] = rst             // connect rst to led[0]
```

Because the second statement has priority over the first, `led[0]` will actually NEVER have the value 0! It will be permanently connected to `rst` . Note that the second line only assigns the first bit of `led` . That means that the other 7 bits will still receive their value from the first statement.

This is one of the weird things of working with hardware. The code you write is not run on a processor like it is when you program. Instead, the code you write is interpreted by the tools to figure out what behavior you want. The tools then create a circuit that will match that behavior.

Always blocks are an easy way to describe complex behavior, but the way you describe the behavior, and it's actual implementation can vary.

Congratulations! You've finished your first Lucid tutorial.

« PREV
Alchitry Constraints Reference

NEXT »
Synchronous Logic

© 2025 Alchitry