Io Element

2024-09-18 · 37 min · Justin Rajewski | Suggest Changes

▶ Table of Contents

This tutorial will introduce the Io Element and walk through some basic examples to get your feet wet.

If you don't own an Io Element, you can still follow along using the simulator.

If you haven't already, I highly recommend reading through the Your First FPGA Project and synchronous logic tutorials before continuing.

Create the Project

In Alchitry Labs, open the new project dialog (Alchitry Logo->New Project...) and name it whatever you want. I named mine *Io Element Demo*. Before you click *Create*, select *Io Base* from the *Template* drop-down menu. This will give you a bare-bones base to start working with the element.

The only difference between this project and the standard base project is that this has the connections to the Io Element already defined for you.

If you have a Cu with an older Io Element, you should use the *Io Base Fake Pull-downs* template instead.

If your Io Element says "Built by: SparkFun" on the back with a little "V2" towards the end, you can use the normal version. If your board doesn't say "Built by: SparkFun", or is missing the "V2", you need to use the fake pull-downs version.

The *Fake Pull-downs* version connects the `fake_pull_down` and `fake_pull_down_2d` components to the buttons and dip switches. The older Io Element didn't have pull down resistors and the FPGA on the Cu doesn't have internal ones so we instead fake them. These module fakes pull-down resistors on the DIP switches and buttons by pulling the pin low for a very short amount of time, releasing it, waiting a short period, then sampling the input.

The buttons and DIP switches are connected to +3.3V via a 330 ohm resistor. This resistor will prevent any damage to the FPGA if the button or DIP switch is active, and the FPGA tries to pull the pin to 0V.

This method of faking the pull-down resistor works because there is some amount of capacitance on each input pin. When the switch isn't active, the pin is normally floating (not being pulled to specific voltage). The FPGA then forces it to 0V and releases it. Since its floating, the only sources to change the voltage are small amounts of leakage in the FPGA and noise. These both take much longer than a couple clock cycles (30ns) to significantly affect it. Therefore, when we read the state of the pin right after pulling it down, it will still be 0 if nothing tried to change it.

If the button or switch is active, when the FPGA pulls the pin down, it will pull it down to 0 thanks to that resistor. However, once the FPGA lets go of the pin, the parasitic capacitance will start to charge. We found it takes about 10ns-20ns for it to reach a '1' state (3.3V) after letting go. This is why we wait for 30ns before sampling the pin. By repeating this process over and over, we can simulate having a normal pull-down resistor.

Thanks to this module, you can ignore that any of this is even happening and simply use its output as you would the input with a pull-down. The only difference is instead of using `io_dip` or `io_button` you use `dip_pd.out` and `button_pd.out` respectively.

Let's take a look at the top file.

Standard Fake Pull-downs

copy

```
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
    output io_led[3][8],    // LEDs on IO Shield
    output io_segment[8],   // 7-segment LEDs on IO Shield
    output io_select[4],    // Digit select on IO Shield
    input io_button[5],     // 5 buttons on IO Shield
    input io_dip[3][8]      // DIP switches on IO Shield
) {

    sig rst                 // reset signal

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

    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           // loop serial port

            io_led = 3x{{8h00}}
            io_segment = 8hFF
            io_select = 4hF
        }
    }
```

You can see we have a handful of inputs and outputs named `io*` . These are the Io Element specific signals.

`io_led` connects to the 24 LEDs. This signal is organized as a 2D array to make it easier to access the three groups of 8 LEDs. For example, if you want to get the first group of LEDs you would use `io_led[0]` and if you wanted to get the third LED in the first group, you would use `io_led[0][2]` .

Because `io_led` is a 2D array, when we set all the LEDs to 0, we have to use a little fancy syntax.

```
io_led = 3x{{8h00}}
```

We need the dimensions of our constant to match `io_led` . In this case, `io_led` is a 3 by 8 array. We use `8h00` as the base constant which is a single dimensional array of width 8. The `x{}` syntax takes the outermost dimension and duplicates it. If we wrote `3x{8h00}` , we would end up with `24h000000` since the outermost dimension is 8. This isn't what we want as it is still a 1D array.

Instead, we need to first create a 1 by 8 array and duplicate it three times. To do we use the array building syntax, `{}` . The value `{8h00}` is a 1 by 8 2D array. We then use `3x{{8h00}}` to get a 3 by 8 array of all zeros. Note that we could have also written `{8h00, 8h00, 8h00}` , but the duplication syntax is a bit cleaner.

`io_segment` and `io_select` are used to control the 4 seven-segment LED displays. They are active low (0 = on, 1 = off) and they will be covered in more detail later.

`io_button` is simply the 5 push buttons. `io_button[0]` is up, `io_button[1]` is center, `io_button[2]` is down, `io_button[3]` is left, and `io_button[4]` is right.

Finally, `io_dip` is the 24 DIP switches. This is arranged in a 2D array exactly the same way as `io_led` .

Logic Gates

The first example we will work though is connecting a few DIP switches to the LEDs through logic gates. If you aren't familiar with what logic gates (AND, OR, NOT, XOR) are, you should take a look at the digital logic tutorial.

In this example we are going to start by making the first LED light up when switch 0 **and** 1 are turned on.

Standard Fake Pull-downs

```
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         // loop serial port

    io_led = 3x{{8h00}}
    io_segment = 8hFF
    io_select = 4hF
    io_led[0][0] = io_dip[0][0] & io_dip[0][1]
}
```

Take a look at the last line. First we need to index the first LED. The first LED is in the first group (group 0) and is the first in its group, so we used the index `[0][0]`. We then index the first and second switches in the same manner. Finally, we use the `&` operator to *AND* the bits together. `&` is the bit-wise AND operator.

Notice that I didn't change the line where we assigned all LEDs a value of 0. This is because the assignment to the first LED will simply be overruled by our second assignment. It's good to remember that while `always` blocks have a sequential order of precedence, they aren't actually sequential. This means that `io_led[0][0]` will *only* have the value of the second assignment and it as if the first assignment never happened. The tools will simply wire an AND gate to the two switch inputs and the LED output.

Go ahead and build/load the project onto your board or click the bug to start the simulator. Try playing with the first two switches. If either switch is off, the LED should be off. Only when both switches are on should the LED turn on.

You can now go back and replace the `&` operator with the operators for OR, `|`, and XOR, `^`. Play with the switches and make sure you understand the different operators.

Practice

Add two more lines to the `always` block so that the first LED of the first group lights up as before, the first LED of the second group lights up when either (OR) of the first two switches in the second group are on, and the first LED of the last group lights up only when exactly one (XOR) switch of the first two switches in the last group are on.

Bit-wise

The operators we have talked about so far are all bit-wise operators. That means that they operate on two equally sized arrays and will perform their operation on each pair of bits individually.

For example, if we want to light up the LED in the first group only when the corresponding switch in the first and second groups are both on we could do the following.

Standard Fake Pull-downs

```
io_led[0] = io_dip[0] & io_dip[1]
```

Note that `io_led[0]` is an array of width 8. So is `io_dip[0]` and `io_dip[1]` . This single statement will create 8 individual AND gates.

You can again use the OR and XOR operators in the same way.

We can also chain multiple gates together. For example, what if we want the LED to light only when the corresponding switch in all three groups is on?

Standard Fake Pull-downs

```
io_led[0] = io_dip[0] & io_dip[1] & io_dip[2]
```

Bit-wise operators are evaluated from left to right. In this case, `io_dip[0]` will be ANDed with `io_dip[1]` and the result will then be ANDed with `io_dip[2]` . In the case of AND gates, the order actually doesn't matter. However, if you start mixing and matching operators the order can matter. You can also use parenthesis to make the order you want things evaluated explicit.

Practice

> Use two bit-wise operators so that the LEDs in the first group light up when the corresponding switch in the third group is on or both switches in the first two groups are on.

Reduction Operators

Reduction operators are similar to the bit-wise operators, but they work on a single input array and always output one bit. ou could say they *reduce* an array to a single bit.

You can think of these operators as one huge logic gate with many inputs. The number of inputs is equal to the size of the array the operator is used on.

In this example, we want to turn the first LED on only if every switch in the first group is turned on. To do this we can use the `&` reduction operator.

Standard Fake Pull-downs

```
io_led[0][0] = &io_dip[0]
```

This is equivalent to ANDing each bit individually as shown below.

Standard Fake Pull-downs

```
io_led[0][0] = io_dip[0][0] & io_dip[0][1] & io_dip[0][2] & io_dip[0][3] &
```

However, as you can tell, the reduction operator is much more compact.

It is even possible to use the reduction operators on multidimensional arrays. For example, if we want the LED to turn on when *any* switch is on we could use the following.

Standard Fake Pull-downs

```
io_led[0][0] = |io_dip
```

I tend to use the AND version to easily detect if a signal is at its maximum value (all 1's). The OR version basically tells you if the value is not 0 and the XOR version will tell you if there is an odd number of 1's.

Each of these operators can also be negated with the bit-wise negate operator, `~` . You can simply place this in front of the expression to get the NAND, NOR, and XNOR equivalents.

> ## Practice
>
> Using your newly acquired bit-wise and reduction operator skills, make the first LED light up when any switch in the third group is on, or all the switches in the second group are on and the first group has an odd number of switches on.
>
> This is a fairly tricky challenge but everything you need has been mentioned in this tutorial. If you get stuck feel free to head over to the forum for some help.

Math

This part of the tutorial we will look at addition, subtraction, and multiplication. We will be performing different operations using the DIP switches as our number inputs. This means everything will be in binary, so if you aren't familiar with binary check out the encodings tutorial.

Let's start with some addition. We will add the binary values from the first and second groups of DIP switches.

Standard Fake Pull-downs

```
sig result[24]              // result of our operations

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         // loop serial port

    io_led = 3x{{8h00}}
    io_segment = 8hFF
    io_select = 4hF

    result = io_dip[1] + io_dip[0] // add the switch values

    io_led = $build(result, 3) // convert result from a 24-bit array to a 3
}
```

Because the result of many of our operations will be more than 8 bits, we can create a signal, `result` , to hold the value. You can think of signals (`sig`) simply as wires. They don't add any cost to your design as they simply define the connection from something to something else.

We set `result` to be the output of our addition. The addition of two 8-bit numbers results in a 9-bit result. This is because an 8-bit number has the range 0-255 so adding two of them together will have the range 0-510, which requires 9 bits to represent.

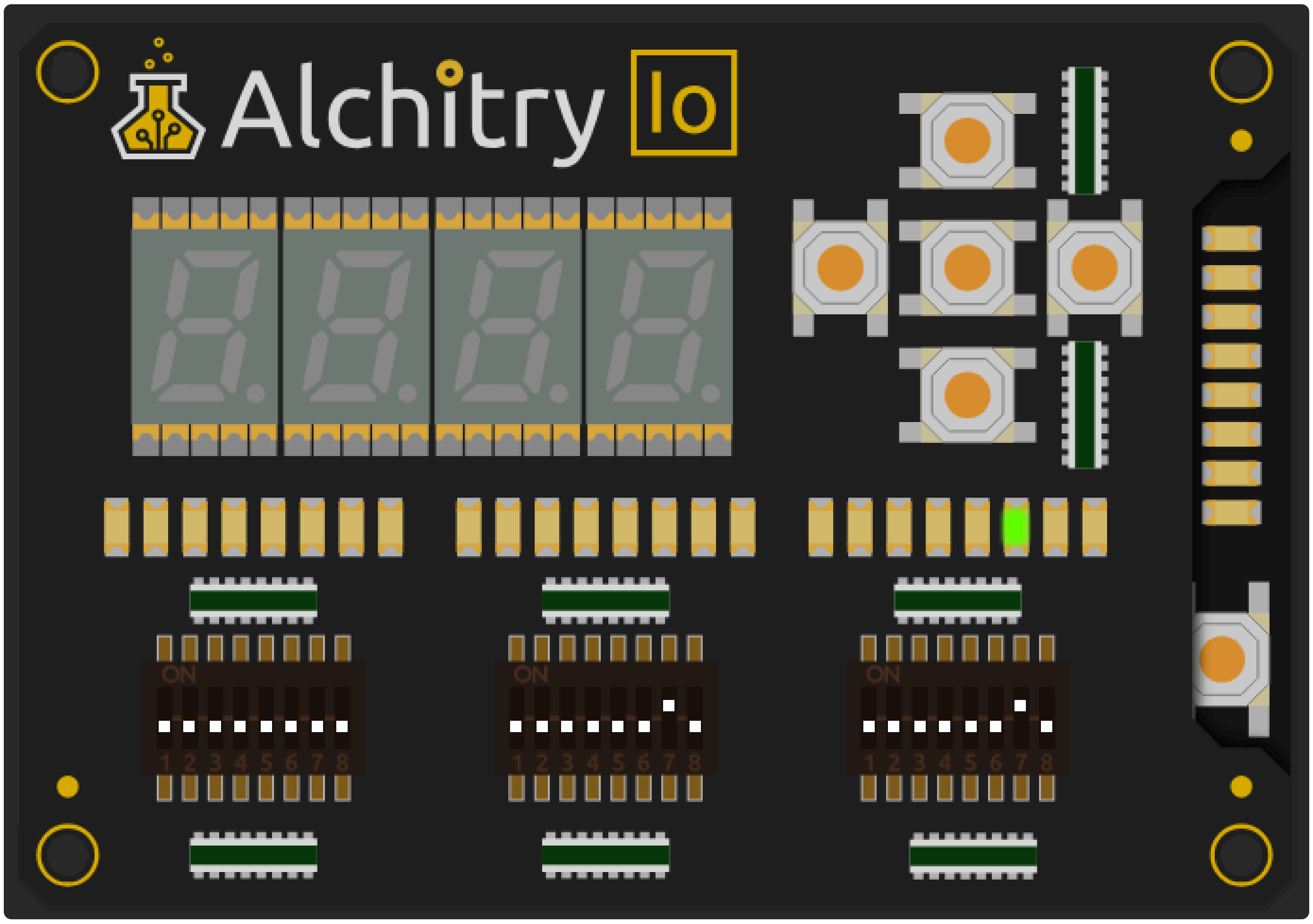
We then take result and connect it to the LEDs so that we can see all 24 bits. However, this requires the use of the function `$build()` . This function takes a 1D array as the first argument and splits it into a multidimensional array. In our case we pass 3 as the second parameter which causes it to split the 24 bits three ways into a 3x8 array.

You can pass `$build()` more than one dimension to split. For example, `$build(result, 3, 2)` would create a 3x2x4 array.

The function `$flatten()` does the opposite. It takes a multidimensional array and flattens it into a 1D array.

Build and load your project to your board or fire up the simulator.

Congratulations! You've just build a very basic calculator. To make sure everything is working properly, let's calculate 2+2. 2 has the binary representation of 0000 0010. Set both groups of switches to 2. The result on the LEDs should be 4, or 0000 0100.



Now change the `+` to a `-` to subtract the first switch value from the second.

Standard Fake Pull-downs

```
result = io_dip[1] - io_dip[0] // subtract the switch values
```

What happens when you set the second switch to 0 and the first to 1 so you're performing 0-1? All 24 LEDs turn on! You can interpret this answer in two different ways. The first is that the result simply underflowed and restarted back at the maximum value. The second way is to assume the number is a 2's complement representation of -1. Both are valid interpretations and what you use will depend on the scenario.

Even though when the number is negative all 24 LEDs light up, you only need 9 bits to represent any 8 bit by 8 bit addition or subtraction. This is because when the value would be negative, dropping the leading 1's doesn't change its value.

Finally, change the `-` to `*` to test out multiplication.

Standard Fake Pull-downs

```
result = io_dip[1] * io_dip[0] // multiply the switch values
```

You should notice that multiplying two numbers can generate a much bigger result than the operands. The result will generally be the size of the inputs combined. So in our case, up to 16 bits wide.

Multiplication is much more expensive to perform than a simple addition or subtraction. This is because a multiplication is essentially many additions. Multiplying two 8-bit numbers will result in a series of eight additions.

Now for division...

Division is much more complicated than even multiplication. You typically don't want to just use the `/` operator to perform division with general numbers. This is because this operation doesn't allow for any pipelining so the entire division must happen all at once and that can be slow.

If you don't need real division, you can approximate it by multiplying with fractic

To approximate division, you can first multiply by any number and then shift the value to the right. Shifting to the right effectively divides by a power of two (rounding down).

For example, if you want to divide a number by 3, you could multiply by 85 then shift right 8 bits. This will effectively multiply your value by 85/256 = 0.33203 which is pretty close to 1/3. If you need higher precision, you can use more bits. For example, you could instead multiply by 21845 and shift 16 bits to the right. This is effectively multiplying by 0.333328, but 16 bit multiplication is significantly more expensive than 8 bit.

If you need real division, you can try to use the `/` operator directly. If you are using an Au, you can also use the *Vivado IP Catalog* to generate an optimized divider.

Go ahead and try out division. Note that dividing by `0` technically produces the value `bx` which means the tools what do whatever they want. With Vivado and Yosys, dividing by 0 produced `24hFFFFFF`. With i_cecube2, dividing by 0 produced `24h0000FF`.

If you divide by 0 in the simulator, the LEDs will turn red indicating a `bx` value.

Seven-Segment Displays

The Io Element has four seven-segment displays that are multiplexed. This means that we have two groups of signals, the first, `io_segment`, connects to the segments of each display.

Note that even though we call these *seven*-segment displays there are actually 8 LEDs in each one because of the decimal point.

All four displays have all their segments connected together. That means if you apply power to segment 0, all four displays segment 0 receive power! This is done to save on the number of pins required to drive the displays. However, it means that we need a way to disable all but one display so that we can show a unique number on each one.

This is where the second group of signals, `io_select`, comes in handy. These are used to select which digit is active. Typically, only one digit will be active at any time. If you have more than one digit active, all active digits will show exactly the same thing since their segments are wired together.

The way we are going to display four unique numbers is by cycling which digit is active really fast. If we can do this fast enough, our eyes can't tell, and it looks like all four digits are always on.

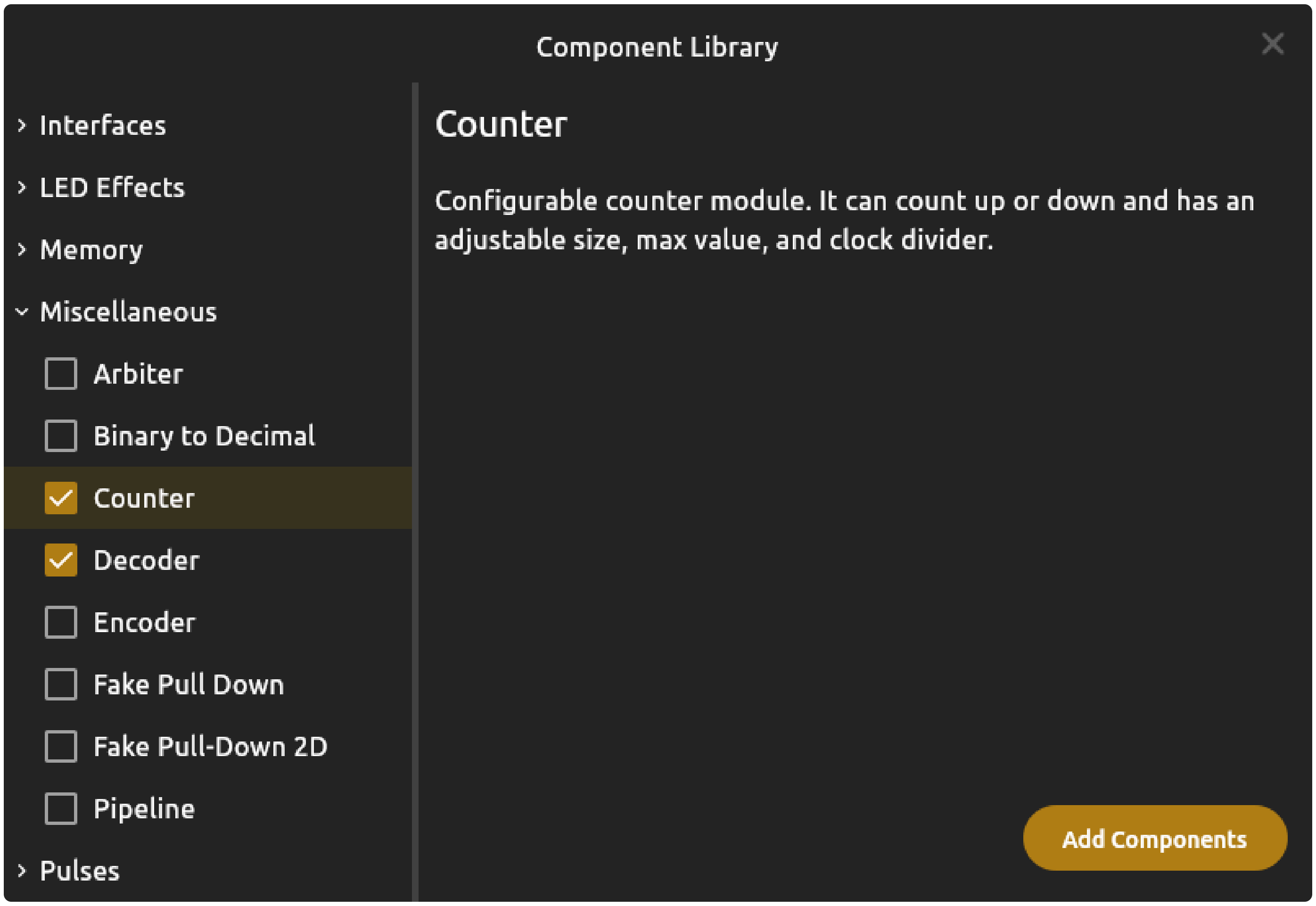
However, before we do that, let's play around with the displays a bit to get familiar with how they work.

Indexing Segments

We are going to set up some modules so that we light one segment at a time. To do this we will be using some of the components built into Alchitry Labs.

Click on the three block icon. If you have a Cu, this will open the *Component Library*. If you have an Au, you will need to click *Component Library* from the dropdown.

Under *Miscellaneous* check off *Counter* and *Decoder*.



Counter

Configurable counter module. It can count up or down and has an adjustable size, max value, and clock divider.

Click *Add Components* to add them to your project.

You can now find the source for these components under *Source Files* in the left project tree in Alchitry Labs. These are marked with the component icon. Note that while you are free to look at the source, components are read-only (no editing!). If you ever want to modify a component, simply copy and paste the code into a module of your own.

It's always a good idea to check out the source and read the description of the component. Many components have parameters that you can set to customize them for your use. We'll be doing exactly that with these two.

Standard Fake Pull-downs

```
.clk(clk) {
    // The reset conditioner is used to synchronize the reset signal to the
    // clock. This ensures the entire FPGA comes out of reset at the same t
```

```
      reset_conditioner reset_cond

      .rst(rst) {
          counter ctr(#SIZE(3), #DIV($is_sim() ? 9 : 25))
      }
  }

  decoder num_to_seg(#WIDTH(3))
```

Here we add the `counter` and `decoder` modules to the top level module. The `counter` component requires both a clock and reset signal so we have it in the

`decoder` doesn't require either, so it is outside both.

The `counter` is simply a counter. It will output a value starting from zero and increment to whatever the maximum value you set is. After that it will reset back to zero. It is also possible to configure it to count down from the max value to zero.

The counter has four parameters you can set to get it to operate how you want, but here we only need to change two. `SIZE` dictates how wide the output of the counter is. In this case we need it to count from 0 to 7 for all 8 segments of our displays. Three bits is exactly 0 - 7. If we wanted to count to a value that wasn't conveniently a power of 2 minus one, we could set the parameter `TOP` to control its maximum value.

The `DIV` parameter is used to set how many bits to use to divide the clock. By setting this to 25, the counter will only increment its output every 225, or 33,554,432, clock cycles. The clock runs at 100MHz (100,000,000 cycles per second) on the Au and Cu. If the counter incremented every clock cycle it would be way too fast for us to see.

For simulations, we use 9 instead of 25 because of the reduced simulation spee

The decoder is a binary to one-hot decoder. If you need a refresher on one-hot representation check out the encodings tutorial. Basically, the decoder will take our three bits and convert it to eight bits. If the input is 0, then the 0th bit of the output will be 1 and everything else will be 0. If the input is 4, the 4th bit will be 1 and everything else will be 0. This allows us to use the binary counter to light individual LEDs.

Note that the output of the decoder is 2^ `SIZE` bits wide. So in our case, 2^3 = 8 bits wide.

Now we can hook up the modules.

Standard Fake Pull-downs

```
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         // loop serial port

    num_to_seg.in = ctr.value

    io_segment = !num_to_seg.out
    io_select = 4h0

    result = io_dip[1] * io_dip[0]
    io_led = $build(result, 3)
}
```

We feed the binary counter value into the decoder and the output of the decoder to `io_segment` . Note that `io_segment` and `io_select` are both *active low*. This means that when the signal is 0, it is active. To turn only one LED on we need to invert the output of the decoder with the bit-wise inversion, `~` , operator. This will make the signal zero-hot.

Also notice we need to set `io_select` to 0 to enable a digit. Setting all of them to 0 will turn them all on.

Build and load your project to your board or fire up the simulator. You should now see the segments of all 4 displays lighting one at time. Try to light only a single display by setting `io_select` to `~4h1` instead.

Now what if we wanted to cycle through each digit? We need another counter that increments each time the original counter overflows. That way once each LED of a specific digit is lit, the next digit is selected. However, we don't actually need another counter to do this! We can simply make our existing counter 2 bits wider. By adding extra bits, these will increment only when the lower three bits overflow, creating the behavior we want.

Standard   Fake Pull-downs

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

    .rst(rst) {
        counter ctr(#SIZE(5), #DIV($is_sim() ? 9 : 25))
    }
}

decoder num_to_seg(#WIDTH(3))
decoder num_to_digit(#WIDTH(2))
```

Here we increase the `SIZE` of `ctr` to 5. We also add another decoder with `WIDTH` of 2. We need 2 since we have 4 digits to select from and an input of 2 bits gives us possible values of 0-3.

Standard Fake Pull-downs

```
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         // loop serial port

    num_to_seg.in = ctr.value[2:0]   // lower three bits used for segments
    num_to_digit.in = ctr.value[4:3] // upper two bits used for digits

    io_segment = ~num_to_seg.out     // connect segments to counter
    io_select = ~num_to_digit.out    // connect digit select to counter

    result = io_dip[1] * io_dip[0]
    io_led = $build(result, 3)
}
```

We can select the lower three bits of `ctr.value` by using the bit selector `[2:0]` . This says we want bits 2 down-to 0 inclusive. We can do the same thing for the upper two bits.

Build and load your project again. This time you should see only one digit and one segment lit at any time. Once each segment of a digit has been lit, the next digit is selected.

You may have also noticed that the segments are now much brighter. This is because the resistor that is in-line with the LEDs is sized for a single LED, not all four. Wh you have all four on at the same time, the current is split between them making them all dimmer.

Getting Fancy With Numbers

Lighting up the segments is super cool and all, but it's much cooler to actually show some numbers.

Before we jump into multiplexing, we need a way to convert a number into the segments that need to be lit to represent that number. To do this we can use a look-up table.

Create a new module called `seven_seg` and add the following.

```
module seven_seg (
    input char[4],
    output segs[7]
) {
    always {
        case (char) {
            0: segs = 7b0111111
            1: segs = 7b0000110
            2: segs = 7b1011011
            3: segs = 7b1001111
            4: segs = 7b1100110
            5: segs = 7b1101101
            6: segs = 7b1111101
            7: segs = 7b0000111
            8: segs = 7b1111111
            9: segs = 7b1100111
            default: segs = 7b0000000
        }
    }
}
```

This module will take a binary number, `char` , and output the segments, `segs` that need to be on.

This module uses a `case` statement. Case statements are equivalent to a bunch of `if` statements. One of the entries will be selected based on the value of `char`. If `char` is 1, then only the block after the `1:` is used. If none of the blocks match, the `default:` block is used. The default block is optional, but it is generally a good idea to include it even if you think you have all the cases covered.

If `char` is an invalid number (not 0-9), then all the segments will be off.

To test out our look-up table, let's connect it to our counter.

Standard   Fake Pull-downs

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

    .rst(rst) {
        counter ctr(#SIZE(4), #TOP(9), #DIV($is_sim() ? 9 : 25))
    }
}

seven_seg seg
```

We need `ctr` to count from 0-9 so we set `TOP` to 9 to cap its value. Also, we need to change `SIZE` to 4, as we only need 4 bits to represent 0-9.

Because we don't need the decoders anymore, we can remove them and add our `seven_seg` module.

Finally, we need to wire it up the LEDs

Standard   Fake Pull-downs

```
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         // loop serial port

    seg.char = ctr.value

    io_segment = ~seg.segs  // connect segments to counter
    io_select = ~4h1        // first digit only

    result = io_dip[1] * io_dip[0]
    io_led = $build(result, 3)
}
```

`io_select` is set to `~4h1` so that only the right most digit is on.

Build and load the project onto your board to make sure the first digit is correctly counting from 0 to 9.

Now we need to make a module that will take four values in and display them on all four digits instead of only using one.

Create a new module called `multi_seven_seg` and add the following code.

```
module multi_seven_seg #(
    DIGITS = 4 : DIGITS > 0,
    DIV = $is_sim() ? 0 : 16 : DIV >= 0
)(
    input clk,                // clock
    input rst,                // reset
    input values[DIGITS][4],  // values to show
    output seg[7],            // LED segments
    output sel[DIGITS]        // Digit select
) {

    // number of bits required to store DIGITS-1
    const DIGIT_BITS = $clog2(DIGITS)

    .clk(clk), .rst(rst) {
        counter ctr (#DIV(DIV), #SIZE(DIGIT_BITS), #TOP(DIGITS-1))
    }

    seven_seg seg_dec                          // segment decoder
    decoder digit_dec(#WIDTH(DIGIT_BITS))      // digit decoder

    always {
        seg_dec.char = values[ctr.value]       // select the value for the act
        seg = seg_dec.segs                     // output the decoded value

        digit_dec.in = ctr.value               // decode active digit to one-h
        sel = digit_dec.out                    // output the active digit
    }
}
```

This module is parameterized so that it could be used to drive more or less than 4 digits, but the defaults are set to work well with our project.

The constant `DIGIT_BITS` is the number of bits we need to cover all `DIGITS` number of digits. We use the function `$clog2()` which computes the ceiling log base 2 of a constant value.

We only want the counter to count from 0 to `DIGITS-1` so we set the `TOP` parameter accordingly.

The way this module works is `ctr` is used to select the active digit. The value from `values` is then selected, decoded, and sent to the segments of the LEDs displa The binary value of the active digit is then decoded into a one-hot value as before and used to select which digit is on. Because `ctr` keeps cycling the active digits, all the displays will appear on.

`DIV` is used so that we don't switch between the digits too fast. If `DIV` is too low and we switch too fast, the transistors that drive the LEDs don't have time to fully turn off, and we get bleed between digits. If it is set too high, we will start to see the digits flicker. For simulations, we don't have this problem so we set it to `0` .

Let's test out this module by feeding it some constant numbers to show.

Standard Fake Pull-downs

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

    .rst(rst) {
        multi_seven_seg seg
    }
}
```

We don't need the counter or the `seven_seg` modules from before, but we need the `multi_seven_seg` module now.

Standard Fake Pull-downs

```
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        // loop serial port
    seg.values = {4h8,4h5,4h3,4h1}

    io_segment = ~seg.seg  // connect segments to the driver
    io_select = ~seg.sel   // connect digit select to the driver

    result = io_dip[1] * io_dip[0]
    io_led = $build(result, 3)
}
```

On the line starting with `seg.values`, we feed the values to display into the driver.

Build and load the design to your board and make sure each digit is displaying the correct number.

Decimal Counters

Alright, now that we have a way to show a value on our display, let's make the display count! For this we need a decimal counter. We could use a regular counter and convert the binary value into four decimal values, but that's a bit trickier. Instead, we will create a special counter that counts in base 10.

To do this we will create two modules. The first will be a single digit counter and the second will chain these together for a multi-digit counter.

Create a new module called `decimal_counter` with the following code.

```
module decimal_counter (
    input clk,      // clock
    input rst,      // reset
    input inc,      // increment the counter
    output ovf,     // counter overflowed
    output value[4] // current value
) {
    .clk(clk), .rst(rst) {
        dff val[4]     // value storage
    }

    always {
        value = val.q              // output the value

        ovf = val.q == 9 && inc   // if max value and incrementing, overflo

        if (inc) {                 // should add 1
            if (val.q == 9)        // if max value
                val.d = 0          // reset to 0
            else                   // otherwise
                val.d = val.q + 1 // add one
        }
    }
```

Our decimal counter has an input `inc` that is used to signal when the value should be increased. When the digit is about to overflow to 0, the flag `ovf` is set to 1. The current value of the counter is output on `value`.

We now need to chain these together to make a multi-digit counter.

Create a new module named `multi_decimal_counter` and add the following code.

```
module multi_decimal_counter #(
    DIGITS = 4 : DIGITS >= 2  // number of digits
) (
    input clk,                // clock
    input rst,                // reset
    input inc,                // increment counter
    output digits[DIGITS][4]  // digit values
) {
    .clk(clk), .rst(rst) {
        decimal_counter dctr[DIGITS] // digit counters
    }
    always {
        dctr.inc[0] = inc   // increment the first digit
        digits = dctr.value // output the values

        // if the previous digit overflows, increment the next
        dctr.inc[1+:DIGITS-1] = dctr.ovf[0+:DIGITS-1]
    }
}
```

Here we take `DIGITS` `decimal_counter` modules and chain them together. Each counter's `inc` signal is connected to the previous counter's *ovf* signal. This means that once a counter overflows from 9 to 0, the next one will be incremented. Of course, the first counter doesn't have a previous counter's `ovf` signal, so we use an external increment signal instead.

We use the bit selectors `[1+:DIGITS-1]` and `[0+:DIGITS-1]` to select bits from `dctr.inc` and `dctr.ovf` . What `[1+:DIGITS-1]` means is starting from bit 1, select `DIGITS-1` bits going up. This is an easy way to select a specified number of bits given some start index. You can use `-:` instead of `+:` to select bits going down instead of up from the given start index.

All that is left now is to generate the `inc` signal and connect the counter to our display driver.

The `inc` signal needs to be a pulse that stays high for only one clock cycle every time we want to increment our counter. To create this we are going to use the counter component as before, but there is a problem with this. If we create a counter that only outputs one bit, the bit will be high for have of the time and not just one cycle at a time. To fix this we will use another component, the *Edge Detector*.

Go into the *Component Library* and add the *Edge Detector* to your project. You ca find it under *Pulses*.

This component takes a signal and sends out a pulse when it detects a rising, falling, or either type of edge. You can configure which edges you care about.

Let's add all the modules to `alchitry_top`.

Standard Fake Pull-downs

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

    edge_detector edge_detector(#RISE(1), #FALL(0))

    .rst(rst) {
        multi_seven_seg seg
        multi_decimal_counter dec_ctr
        counter ctr(#SIZE(1), #DIV($is_sim() ? 8 : 24))
    }
}
```

For the edge detector, we set `RISE` to 1 and `FALL` to 0. This means we will generate a pulse only on rising edges.

We now have the multi-digit driver, the multi-digit counter, and a way to generate a pulse from the counter.

Let's wire them all up!

Standard Fake Pull-downs

```
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         // loop serial port

    edge_detector.in = ctr.value
    dec_ctr.inc = edge_detector.out
    seg.values = dec_ctr.digits

    io_segment = ~seg.seg   // connect segments to the driver
    io_select = ~seg.sel    // connect digit select to the driver

    result = io_dip[1] * io_dip[0]
    io_led = $build(result, 3)
}
```

The value from the counter is fed into the edge detector. The pulse from the edge detector is fed into the decimal counter. The decimals from the decimal counter are sent to the multi-digit driver. Finally, the multi-digit driver is connected to the actual display.

This is what is so cool about FPGAs. Each one of these modules operates completely independently of one another since they all exist in hardware!

This example is fairly complicated, but by breaking down the problem in little chunks (modules) we were able to create a design that is relatively easy to understand and flexible for future upgrades.

Now for the moment of truth, build and load the project onto your board or fire up the simulator! With any luck, you should see the seven-segment display counting!

If you want to make it count slower or faster you can mess with `ctr` 's `DIV` value.

« PREV
Synchronous Logic

NEXT »
Serial Interface