Home » Tutorials » Hello YOUR_NAME_HERE

Hello YOUR_NAME_HERE

2024-09-19 · 12 min · Justin Rajewski | Suggest Changes

▶ Table of Contents

In this tutorial we will be personalizing the greeter so that it first asks for your name and then prints "Hello NAME" where NAME is the name you entered. To do this we will need some form of memory and in this case we will use a single port RAM.

We will be continuing the project from the last tutorial so make sure you have read the ROMs and FSMs tutorial first.

With the project open from the last tutorial, you can make a copy to edit for this tutorial by going to *Alchitry Icon->Save Project As...*. Enter a new name in the dialog that pops up and click *Create Project*.

The RAM

We need to add the RAM component to our project. Open the *Component Library* and under *Memory* check off *Simple RAM*.

Go ahead and open up the `simple_ram` file.

copy

```
module simple_ram #(
    parameter WIDTH = 1,                    // size of each entry
    parameter ENTRIES = 1                   // number of entries
  )(
    input clk,                              // clock
    input [$clog2(ENTRIES)-1:0] address,    // address to read or write
    output reg [WIDTH-1:0] read_data,       // data read
    input [WIDTH-1:0] write_data,           // data to write
    input write_enable                      // write enable (1 = write)
  );

  reg [WIDTH-1:0] ram [ENTRIES-1:0];        // memory array

  always @(posedge clk) begin
    read_data <= ram[address];              // read the entry

    if (write_enable)                       // if we need to write
      ram[address] <= write_data;           // update that value
  end

endmodule
```

Note that this component is written in Verilog instead of Lucid. This is because the tools that actually build your project can be very picky when it comes to deciding if something is a block of RAM or not. By using this module we can ensure that our RAM is properly recognized as RAM. This is important because FPGAs actually have dedicated block RAM (also known as BRAM). If your RAM is big enough, the tools will use BRAM to implement it instead of the FPGA fabric. Using BRAM is both substantially faster and smaller than the FPGA fabric.

A single port RAM like this works much the same as the ROM from the last tutorial. However, we now have the option to write to an address instead of only reading. To write to an address, we simply supply the address and data to write then set `write_enable` to 1. The data at that address will then be updated to whatever `write_data` is.

The parameters `WIDTH` and `ENTRIES` are used to specify how big we want the RAM to be. `WIDTH` specifies how big each entry is. In our case we will be storing letters and a letter is 8 bits wide so `WIDTH` will be set to 8. `ENTRIES` is used to specify how many entries we want. This will be the maximum name length we can accept.

The Greeter (revisited)

Just like the last tutorial we will have a `greeter` module. The interface to this module is exactly the same as before but it is now a bit more mannered and will greet you personally.

Like most tutorials, I'll post the entire module here and then break it down.

```
module greeter (
    input clk,         // clock
    input rst,         // reset
    input new_rx,      // new RX flag
    input rx_data[8],  // RX data
    output new_tx,     // new TX flag
    output tx_data[8], // TX data
    input tx_busy      // TX is busy flag
) {
    const HELLO_TEXT = $reverse("\r\nHello @!\r\n")
    const PROMPT_TEXT = $reverse("Please type your name: ")

    enum States {IDLE, PROMPT, LISTEN, HELLO} // our state machine

    .clk(clk) {
        .rst(rst) {
            dff state[$width(States)]
        }
```

No More ROM

So unlike last tutorial, we aren't going to use an explicit ROM module. This is because some convenient features of Lucid allow us to easily use constants with strings as ROMs. Let us take a look at the constant declaration.

```
const HELLO_TEXT = $reverse("\r\nHello @!\r\n") // reverse so index
const PROMPT_TEXT = $reverse("Please type your name: ")
```

Here we are using a function called `$reverse()` . This function takes a constant expression and reverses the order of the top most dimension of the array. Since strings are 2D arrays with the top most dimension being the letter order, this is exactly the same as typing the string backwards like we did in the last tutorial. This is just a little bit cleaner and easier to deal with.

Because strings are 2D arrays, we can simply use `HELLO_TEXT[i]` to access the `i` th letter of it.

Note that we are using the @ symbol in place of a name. This will signal to our design where to insert the name that was recorded.

Modules and DFFs

Just like before we have an FSM `state` . This will store the current state of our module. `States.IDLE` is where we will start, and it will initialize everything. `States.PROMPT` will print the prompt asking for your name. `States.LISTEN` will listen to you type your name and echo it back. Finally, `States.HELLO` will greet you personally.

We need counters to keep track of what letter in each ROM we are currently positioned.

```
dff hello_count[$clog2($width(HELLO_TEXT, 0))]
dff prompt_count[$clog2($width(PROMPT_TEXT, 0))]
```

Let us take a look at `hello_count`. We need it to be wide enough so that we can index all the letters in `HELLO_TEXT`. We can get how many letters there are in the string by using the `$width()` function.

The `$width(expr, dim)` function takes two arguments. `expr` is the value to get width of and `dim` is the dimension along we're measuring. If `expr` is a 1D array or bit, then `dim` is optional and assumed to be `0`.

Because `HELLO_TEXT` is a multidimensional array, we need to specify `dim` explicitly as `0` to get the outermost dimension. With `dim` as `0`, `$width()` will return the number of letters. If it was `1` instead, `$width()` would return `8` since each letter is 8 bits wide.

We can then use the `$clog2()` function as before to make sure it is large enough to store values from `0` to `$width(HELLO_TEXT, 0)-1`.

Next take a look at `name_count`. This will be used to index into the RAM. We can set this width to be whatever we want, but the size of the RAM will grow exponentially with it. I set it to 5 which will allow for a name of 2^5, or 32 letters long. We will play with this towards the end of the tutorial.

We need the size of the RAM to match the size of `name_count`.

```
simple_ram ram (#WIDTH(8), #ENTRIES($pow(2,$width(name_count.q))))
```

Here we are using the function `$pow()` which takes two constants and returns the first to the power of the second. In this case, `$width(name_count.q)` is `5`, so 2^5 is 32. By using `$width(name_count.q)` instead of typing in 5 or 32 directly, we ensure that if we change the width of `name_count` then everything will still work.

The FSM

The `IDLE` and `PROMPT` states should look very familiar to the last tutorial so we will jump to the `LISTEN` state.

```
// LISTEN: Listen to the user as they type his/her name.
States.LISTEN:
    if (new_rx) { // wait for a new byte
        ram.write_data = rx_data        // write the received letter
        ram.write_enable = 1            // signal we want to write
        name_count.d = name_count.q + 1 // increment the address

        // We aren't checking tx_busy here that means if someone type
        // fast we could drop bytes. In practice this doesn't happen.
        new_tx = rx_data != "\n" && rx_data != "\r" // only echo non-
        tx_data = rx_data // echo text back so you can see what you t

        // if we run out of space or they pressed enter
        if (&name_count.q || rx_data == "\n" || rx_data == "\r") {
            state.d = States.HELLO
            name_count.d = 0 // reset name_count
        }
    }
```

Here we wait until `new_rx` is `1`. This signals that we have a new byte to process and that the data on `rx_data` is valid. We then write `rx_data` into our RAM. We are writing to the address specified by `name_count.q` as `ram.address` is set to this in the beginning of the always block.

We also need to send the character we received back so that you can see your name as you type it. We simply set `new_tx` to `1` and `tx_data` to `rx_data`. Note that we aren't checking `tx_busy` so it is possible this byte will be dropped. However, in practice you can't type fast enough for this to be an issue. If you wanted to make this more robust you would need to buffer the received letters and send them out only when `tx_busy` was `0`.

The if statement is used to know when to stop. We have two conditions to stop on. The first is if we simply run out of space. To check of this we use `&name_count.q`. The `&` operator here **and**s all the bits of `name_count.q` together into a single bit. This tells us if all the bits of `name_count.q` are `1`. The second condition is that the user pressed the enter key. We want to accept `"\n"` or `"\r"` as a stop character so we check for both.

When we are moving onto the next state, notice that we reset `name_count`. This is so that we can start printing the name from the beginning.

```
// HELLO: Prints the hello text with the given name inserted
States.HELLO:
    if (!tx_busy) { // wait for tx to not be busy
        if (HELLO_TEXT[hello_count.q] != "@") {  // if we are not at
            hello_count.d = hello_count.q + 1    // increment to next
            new_tx = 1                           // new data to send
            tx_data = HELLO_TEXT[hello_count.q]  // send the letter
        } else {                                 // we are at the ser
            name_count.d = name_count.q + 1      // increment the nar

            if (ram.read_data != "\n" && ram.read_data != "\r") // if
                new_tx = 1                                      // se

            tx_data = ram.read_data // send the letter from the RAM

            // if we are at the end of the name or out of letters to
            if (ram.read_data == "\n" || ram.read_data == "\r" || &na
                hello_count.d = hello_count.q + 1  // increment hellc
            }
        }

        // if we have sent all of HELLO_TEXT
        if (hello_count.q == $width(HELLO_TEXT, 0) - 1)
            state.d = States.IDLE // return to IDLE
    }
```

In this state, we are going to use two counters, `hello_count` and `name_count`. First we will start by sending each letter of `HELLO_TEXT`. However, once we hit the

"@" letter we will send all the letters in our RAM. Once that is done, we will finish sending the rest of HELLO_TEXT .

Once everything has been sent, we return to the IDLE state to await another key press to start it all over again.

The Top Level

The top level tile file is exactly the same as last time since the interface to our greeter module is the same.

```
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 sig
        // clock. This ensures the entire FPGA comes out of reset at
        reset_conditioner reset_cond

        .rst(rst) {
            #BAUD(1_000_000), #CLK_FREQ(100_000_000) {
                uart_rx rx
                uart_tx tx
            }

            greeter greeter
        }
    }
```

```

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

        led = 8h00              // turn LEDs off

        rx.rx = usb_rx          // connect rx input
        usb_tx = tx.tx          // connect tx output

        greeter.new_rx = rx.new_data
        greeter.rx_data = rx.data

        tx.new_data = greeter.new_tx
        tx.data = greeter.tx_data
        greeter.tx_busy = tx.busy
        tx.block = 0            // no flow control, do not block
    }
}
```

Building the Project

You should now be all set to build the project. Once the project has built successfully, load it onto your board and open up the serial port monitor to test it out. Note that you have to send it a letter to get it to prompt you for your name.

Here is some demo output.

```
Please type your name: Justin
Hello Justin!
Please type your name: Steve
Hello Steve!
Please type your name: 01234567890123456789012345678901
Hello 01234567890123456789012345678901!
```

Notice that the moment you type 32 letters it cuts you off and says hello.

« PREV
ROMs and FSMs

NEXT »
Register Interface