Breaking the iCE

By MrSmoer

1 Content

2 Intro

This challenge is about reverse-engineering an FPGA-Password checking device.

At first, we read the description carefully and look for hints. To summarize the author tells us that they built a password-checker with **two buttons** and an LED. The buttons each correspond to a logical 1 (BTN1) or 0(BTN2). The password will be made from 5 bytes, which is **40 bits**. Our goal is to enter the correct bit-sequence to light the LED. In the end we can encode our bit-sequence as ASCII. This way we receive the content of the flag.

Important takeaways could be:

- 40 bits

    - BTN1 -> 1
    - BTN2 -> 0
- ASCII encodable (in case we tried brute-forcing, this limits the printable chars)
- LED will light on

Now we go ahead, download the zip-file, unzip it, and look at what we've got:

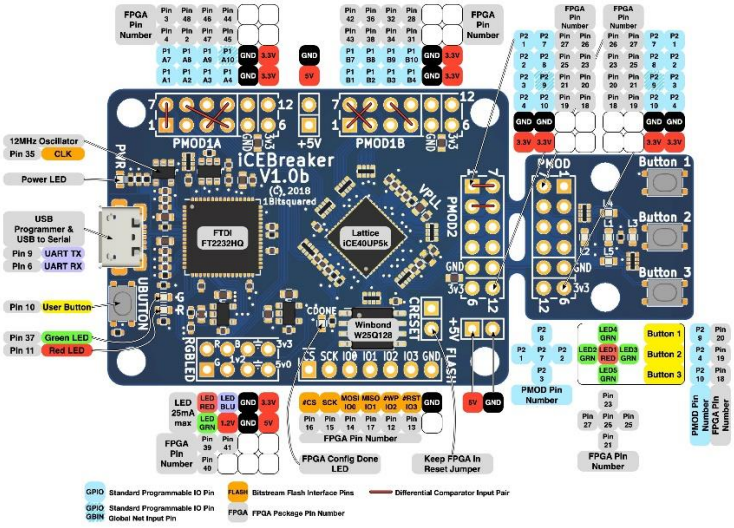
- A binary-file called chal.bin
- A file called icebreaker.pcf that contains pin-mappings for the buttons, LED, and clock
- A video showcasing the device running a similar program, as a visual representation how it's supposed to work

3 Identifying our Device

To crack any kind of password we first must find out how we can interact with the binary to either reverse engineer it or maybe brute force it.

At first, we must identify the device the binary could run on.

On the PCB the silkscreen reads "**iCEBreaker V1.0e**". A quick google search reveals a GitHub-repo for an open-source educational FPGA development board. The block diagram in the Readme reveals that this is in fact our board from the video.



It is using the Lattice **iCE40UP5K FPGA** in a QFN48 (pin) package.

4 Finding an attack

Okay, so now we know on which device the binary is supposed to be run. A first possible solution would be to buy such a device (they go for around 70$ on crowdsupply), flash the binary and try all 1.099.511.627.776 (2^40) possible solutions until the LED powers on, this is sadly not very feasible, and probably not the intended solution, because the CSCG *only* goes on over a two month-period.

Another way would be to reverse the binary. If we google "ice40 fpga reverse engineering" the first link leads us over a friendly hackaday-article to **Project icestorm**. We could have also found it scrolling the icebreaker-Readme a little bit further, "iceprg" a part of the icestorm-Project is listed as a flashing-utility.

4.1 So what is project IceStorm?

Project IceStorm is a Project by Claire Wolf and Mathias Lasser that reverse-engineered the bitstream format (e.g. our chal.bin) for the originally closed Lattice iCE 40 FPGAs and provides Open-Source utilities for working with the iCE40.
The site lists a table of supported devices, listing our iCEBreaker as iCE40-UP5K-SG48 with the corresponding command-line-options for use with their tools.

I decided to build the toolkit from their GitHub-repository according to the instructions at the site.

Now we can use **iceunpack** to convert our chal.bin to a .asc-file. This is the format most tools from IceStorm work with.

```
$ iceunpack chal.bin chal.asci
```

Reading further through the page, the **IceBox**-tools stood out to me: The description says it's "A python library and various tools for working with IceStorm ASCII files and accessing the device database. For example, **icebox_vlog** converts our ASCII file dump of a bitstream into a Verilog file that implements an equivalent circuit."
This tool can convert the .asc-file into **Verilog**. Verilog is a hardware-description-language that models electronic systems and digital circuits. We can even provide a .pcf-file this is exactly what we want!

(I will call my Verilog-file alu.v, because a most of my C++ code depends on this filename, and it was too late to change it)

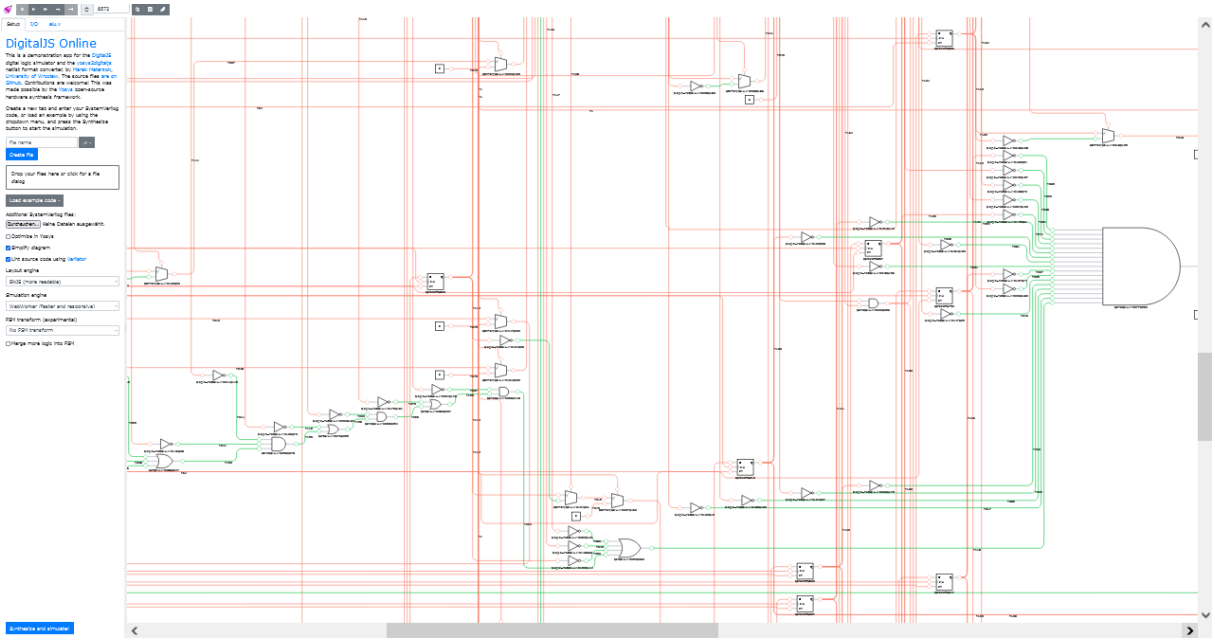
```
$ icebox_vlog -p icebreaker.pcf -d sg48 chal.asc > alu.v
```

This Verilog-file is made of three types of statements:

- wire and register names declarations (e.g.: "wire n314;")
- direct or conditionally logical assignments of wires and registers to eachother (e.g. "assign n3 = n314;")
- event driven "always" statements that set wires to a HIGH or LOW based on some conditions (e.g. "always @(posedge CLK) if (n3) n16 <= n34 ? n292 : n343;")

We can use this Verilog-file to either reverse-engineer the circuit or synthesize the circuit to brute-force the password.

At first, I used https://digitaljs.tilk.eu/, an online Verilog synthesizer to get an overview of the circuit. Here a small excerpt of it (you can judge by the size of the scroll bars).



This looks way to crowded to reverse-engineer by hand, and the website controls are not suitable for such a circuit like this.

At this point I first went down the wrong path of brute-forcing it, but this way I got a deeper understanding of the circuit, and my brute-force code can be used to validate the byte-sequence from the reverse-engineering.

If you don't want to read it, feel free to skip to the reverse engineering-section. Be aware that I have to apply some minor patches to alu.v in order to make it work with verilator, so some details may seem not 100% correct.

5 Brute-forcing

I found this awesome tutorial explaining how to use Verilator to transpile Verilog into a C++ class (Valu.cpp), creating a virtual testbench (tb_alu.cpp) and log the execution (waveform.vcd).

5.1 Adapting the code

I had to make a few adjustments to the Makefile (attached) and rewrote the testbench,

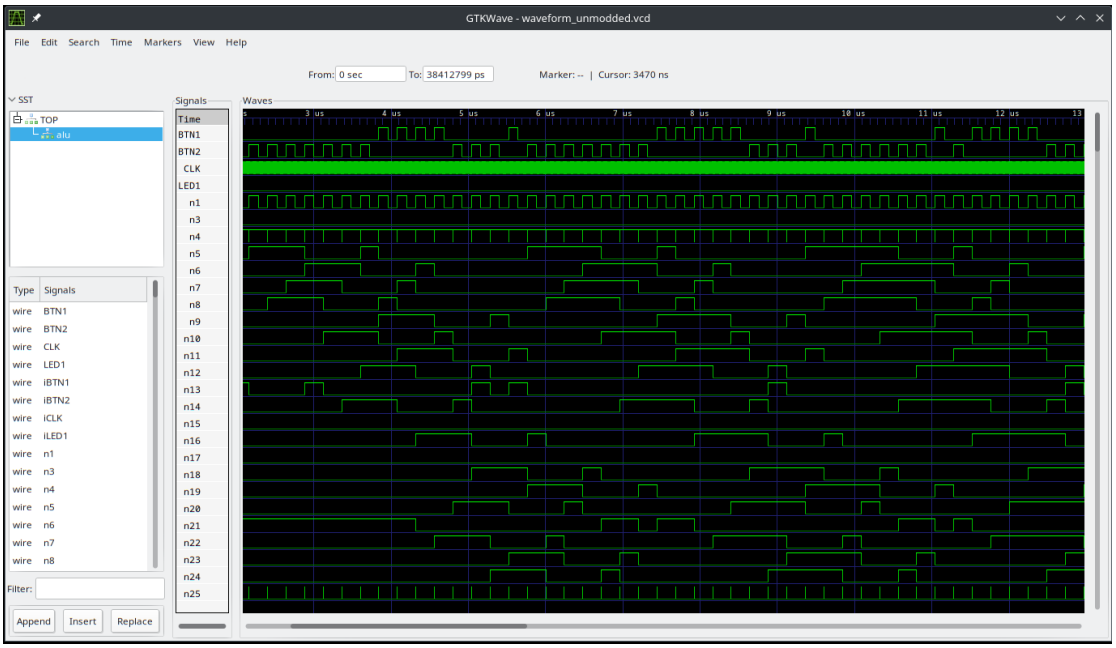
copied my chal.v into their alu.v.

I had to create dummy-inputs (prefixed with an i), because verilator doesn't connect the parameters to already defined lines. This included changing the first line.

```
module alu (input iCLK, output iLED1, input iBTN2, input iBTN1);
```

And assigning the new names to the already declared wires for CLK, BTN1 and so on.

And now we can begin to simulate button presses and look at what they are doing in a software called gtkwave:



5.2 First Problems

Bruteforcing everything turned out to be impossible because of the debounce time of 60000 cycles. This was **slow to simulate** (3byte/s) and the **log-filesize** grew quickly over 10GB.

5.3 Recognizable Patterns

I hoped that I would be able to recognize distinctive patterns with lines only switching to high after special bit sequcenes.
I built my C++ code in a way that I could manually prepend a bit sequence and only variate the following bits. The testbench creates a new simulation after every try. (This had not much impact on speed and would enable multithreading).
I could **not recognize any distinctive patterns** either that could tell me whether the last bit, that I had set, was correct or not, so this idea didn't work either.

5.4 Cutting down the Clock

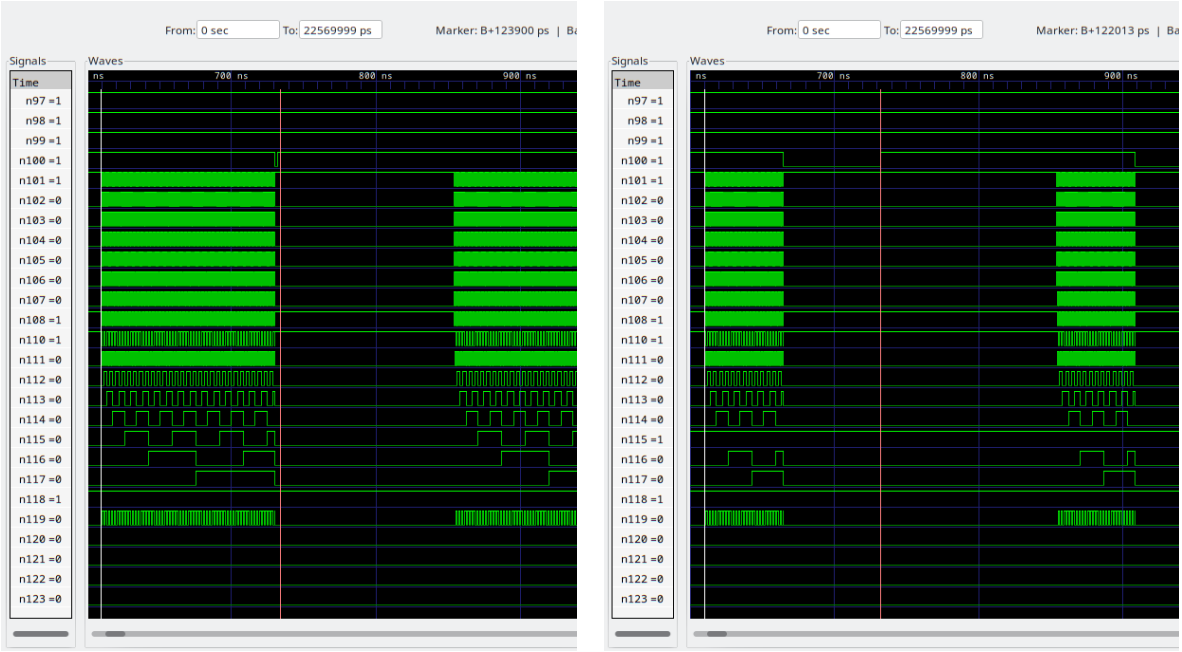
The clock for the debounce-period is clearly visible inside the logs. One can cut down the dividing lines of the clocks by permanently setting intermediate lines to high.

```
always @(posedge CLK) if (n4) n115 <= n1 ? 1'b0 : n219;

to

always @(posedge CLK) if (n4) n115 <= 1'b1;
```

This cuts the debounce-time in half with every dividing line this is applied to:



Important to note is that both screenshots have the same scale. On the left, the lines n101 to n117 divide the CLK-signal (Rises every 2 ps) into the debounce-time of 60000 Cycles (120 000ps). Line n100 goes LOW to allow the next button and rises instantly when the next button is pressed. On the right line 115 is permanently set to HIGH, reducing the divider to 30000 Cycles (60 000ps). Because of that line n100 falls low after half of the old debounce-time.

Setting all dividing lines to 0 effectively bypasses the clock-divider. It went to roughly 30 000 tries/second with printouts and logs removed, but this is still not enough (2^40/(30 000) = 18330000 seconds = 424.2 Days).

I believe I still had a lot of overhead, because I create a new device for each try, and I do not use any kind of multithreading. A Threadripper 3990x with 64 cores/128 threads could have maybe sped up the process down to a few days, considering that the Threadripper would have a roughly 20% better single-core-performance than my i5 7500.

But as we remember: our password must be ASCII encodable. This reduces the byteset we have to try down to 100 distinct characters. With this restriction it becomes almost doable: 100^5/30 000= 333 333 seconds = 3.858 Days!

5.5 De Bruijn sequence

We could also implement de Bruijn sequences. De Bruijn sequences are combinations of bits or bytes that contain every possible combination within them. For example, the sequence “11001” contains all four possible combinations of two bits: “00”, “01”, “10”, “11”. The main point is that the sequence of length five is **shorter than** the four two-bit **combinations joined** together. For longer sequences this tends to perform better than for this simple example. They can be used to brute-force some simple garage-door openers.
Computing them seemed also more expensive than my current solution. I didn’t test it, because it seemed overwhelming, and I chose to go the other way.

5.6 Giving up on brute force

So even with our tweaks we cannot get the time to brute force all 40 bits down to feasible amount of **time, it is too long**. Before solving the challenge, I had no way to verify, that cutting the clock didn’t hurt the general function of the circuit, so I wasn’t willing to commit my time to that.

From multithreading I couldn't expect more than a 2x improvement on my 4-Core machine and it's a hassle to implement.
De Bruijn sequences would require that the device doesn't **need a power cycle** after every try, and I couldn't find out if it needs one.

6 Reverse engineering

Now that I've decided that brute-forcing wouldn't be feasible I begin with reverse engineering.

I will do it all by hand using vscode's **CTRL+F**, and a text-file for **noting** my findings, because I had no python interpreter installed and no internet and loads of time available (2 hours).

I have to admit that this felt like a shot in the dark, but luckily turned out to be the right solution.

So I began at the end: The LED.

6.1.1 Technique

With CTRL+F we find four occurrences of LED. We find:

- The declaration of LED
- assignment to not n247
- "always @(posedge CLK) if (n34) LED1 <= n247;"
- n247 is used nowhere else

This is toggling LED1 on and off every clock cycle as long as n34 is HIGH.

The takeaway is that, the LED is only **dependent on n34**, therefore I note "led < n34".
Now to the next step, I search for n34. In order to not highlight numbers that contain "n34" e.g.: "n343" you can use the regex search feature and add "[^\d]" after the searched signal, which rules out every digit. We can follow the path to the LED using this technique. Sometimes it looks like it'd branch, after n236 it actually **branches**:

```
assign n236 = (n39 ? (n38 ? n35 : 1'b0) : 1'b0);
```

If we look carefully at it, this line only sets n236 to HIGH if n39 **and** n38 **and** n35 are HIGH.

So, we note:

```
led < n34 < n236 < n39 & n38 & n35
```

And begin new lines for each branch and follow them one by one until there is a next branch. After a few searches I got the following notes:

```
led < n34 < n236 < n39 & n38 & n35
        n39 < n210 < n32 & n15 & n49 & n17
        n38 < n299 < n40 & n57 & n43
        n35 < n331 < n33 & n54 & n13 & n21
```

And now we **repeat** it, noting down a '!' (logical not) in front of every signal if it must be LOW to make the parent statement HIGH:

```
led < n34 < n236 < n39 & n38 & n35
            n39 < n210 < n32 & n15 & n49 & n17
                  n32 < n248 < n56 & n26
                  n15 < n246 < n7 & n12 & n11 & n6
                  n49 < n342 < n55 & n60 & n50 &n8
                  n17 < n300 < n24 & n20 & n48 & n23
            n38 < n299 < n40 & n57 & n43
                  n40 < n222 < !n36 & !n37 & !n44 & !n41
                  n57 < n237 < !n61 & !n58 & !n52
                  n43 < n231 < !n42 & !n45 & !n46 & !n47
            n35 < n331 < n33 & n54 & n13 & n21
                  n33 < n309 < !n31 & !n30 & !n29 & !n28
                  n54 < n320 < n51 & !n53 & !n74 & n27
                  n13 < n232 < !n5 & !n9 & !n14 & !n10
                  n21 < n321 < !n18 & !n16 & !n19 & !n22
```

6.1.2 First Recognizable Stuctures

And now we got exactly 41 different branches (we count these) on which our LED depends on. This is pretty close to our 40 bits that we have got to figure out. For each of these lines (e.g.: n56) we find in our alu.v:

- the declaration
- the occurrence in the top branch
- a direct assignment to another line (assign n262 = n56;)
- an "always @(CLK) if (n3)" with some kind of not branching assignment to another line ("always @(posedge CLK) if (n3) n56 <= 1'b0 ? 1'b0 : n349;")

I noted it as in the following format: "n56 a(n262) < @cn3 & n349"

This is almost regular with some small exceptions:

```
led < n34 < n236 < n39 & n38 & n35

              n39 < n210 < n32 & n15 & n49 & n17

                     n32 < n248 < n56 & n26

                            n56 a(n262) < @cn3 & n349

                            n26 a(n330) < @cn3 & n319

                     n15 < n246 < n7 & n12 & n11 & n6

                            n7 a(n346) < @cn3 & n264

                            n12 a(n353) < @cn3 & n284

                            n11 a(n343) < @cn3 & n218

                            n6 a(n270) < @cn3 & n346

                     n49 < n342 < n55 & n60 & n50 &n8

                            n55 a(n272) < @cn3 & n235

                            n60 a(n239) < @cn3 & n347

                            n50 a(n235) < @cn3 & n262

                            n8 a(n264) < @cn3 & n225

                     n17 < n300 < n24 & n20 & n48 & n23

                            n24 a(n336) < @cn3 & n291

                            n20 a(n212) < @cn3 & n226

                            n48 a(n288) < @cn3 & n315

                            n23 a(n310) < @cn3 & n336

              n38 < n299 < n40 & n57 & n43

                     n40 < n222 < !n36 & !n37 & !n44 & !n41

                            !n36 a(n275) < @cn3 & !n288

                            !n37 a(n305) < @cn3 & !n345

                            !n44 a(n315) < @cn3 & !n295

                            !n41 a(n295) < @cn3 & !n324

                     n57 < n237 < !n61 & !n58 & !n52

                            !n61 a(n349)  < @cn3 & !n325

                            !n58 < @cn34 & !n296                !!!!

                            !n52 < @cn3 & !n272

                     n43 < n231 < !n42 & !n45 & !n46 & !n47

                            !n42 a(n259) < @cn3 & !n283
```

```
                    !n45 a(n325) < @cn3 & !n245
                    !n46 a(n283) < @cn3 & !n305
                    !n47 a(n245) < @cn3 & !n259
        n35 < n331 < n33 & n54 & n13 & n21
               n33 < n309 < !n31 & !n30 & !n29 & !n28
                      !n31 a(n238) < @cn3 & !n298
                      !n30 a(n319) < @cn3 & !n238
                      !n29 a(n332) < @cn3 & !n287
                      !n28 a(n287) < @cn3 & !n330
               n54 < n320 < n51 & !n53 & !n74 & n27
                      n51 a(n345) < @cn3 & n209
                      !n53 a(n298) < @cn3 & !n221
                      !n74 a(n221) < @cn3 & !n239
                      n27 a(n209) < @cn3 & n275
               n13 < n232 < !n5 & !n9 & !n14 & !n10
                      !n5 a(n225) < @cn3 & !n332
                      !n9 a(n218) < @cn3 & !n353
                      !n14 a(n284) < @cn3 & !n327
                      !n10 a(n327) < @cn3 & !n270
               n21 < n321 < !n18 & !n16 & !n19 & !n22
                      !n18 a(n291) < @cn3 & !n212
                      !n16 a(n277) < @cn3 & !n343
                      !n19 a(n324) < @cn3 & !n310
                      !n22 a(n226) < @cn3 & !n277
```

6.2 Linking them together

If you type those out by hand you will immediately notice that the end of a line, n349 for n62, is assigned to the begin of another line in the list:

```
led < n34 < n236 < n39 & n38 & n35
              n39 < n210 < n32 & n15 & n49 & n17
                     n32 < n248 < n56 & n26
                                  n56 a(n262) < @cn3 & n349
...
              n57 < n237 < !n61 & !n58 & !n52
                                  !n61 a(n349)  < @cn3 & !n325
                                  !n58 < @cn34 & !n296
```

This is true for almost every line; they all **connect as a sequence**!
One of the exceptions "n60 a(n239) < @cn3 & n347" actually connects through to the buttons.

Now we can note all of them one after another as a big sequence:

```
LED
< !n52 < n55  < n50  < n56  < !n61 < !n45 < !n47 < !n42
< !n46 < !n37 < n51  < n27  < !n36 < n48  < !n44 < !n41
< !n19 < n23  < n24  < !n18 < n20  < !n22 < !n16 < n11
< !n9  < n12  < !n14 < !n10 < n6   < n7   < n8   < !n5
< !n29 < !n28 < n26  < !n30 < !n31 < !n53 < !n74 < n60  < BUTTONS!!
```

Now we can assemble our password. We note **a '0' for every wire with an '!' and a '1' for every wire without** and we get our …

7 Flag

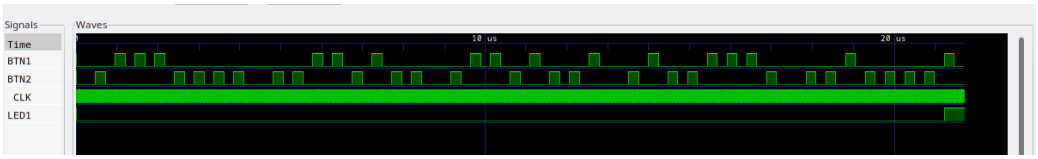
```
01110000 00110100 01101001 01001110 00100001
```

This can now be converted to **ASCII** using python:

```
>>> n = int('0b01110000001101000110100101001110001000001', 2)
>>> print(n.to_bytes((n.bit_length() + 7) // 8, 'big').decode())
p4iN!
```

7.1 Validating our solution

We can even check this using the brute-force tactics. We just punch it in as the part that doesn't get variated. And in the end the **LED turns on!**



8 Mitigations

Getting the Password was only possible, because one can decompile the binary into Verilog. One obvious mitigation would be to **choose a chip**, to which the Community does not provide such an extensive reverse-engineering toolkit.

In the end this was quite easy to reverse with a little bit of patience. If the structure **weren't that regular**. I would have had no way to keep cool with just my notes.
If the person who made the circuit used two **different arrangement styles** of logic gates for every other/random bit, an attack revealing a part of the password wouldn't allow the attacker to find the whole password, or even verify that this approach is correct.

9 Attachments

The files are also available in my GitHub-Repo after the competition ends.

9.1 Makefile

```
# Verilator example makefile
# Norbertas Kremeris 2021
MODULE=alu

.PHONY:sim
sim: waveform.vcd

.PHONY:verilate
verilate: .stamp.verilate

.PHONY:build
build: obj_dir/Valu

.PHONY:waves
waves: waveform.vcd
        @echo
        @echo "### WAVES ###"
        gtkwave waveform.vcd -a gtkwave_setup.gtkw

waveform.vcd: ./obj_dir/V$(MODULE)
        @echo
        @echo "### SIMULATING ###"
        ./obj_dir/V$(MODULE) +verilator+rand+reset+2

./obj_dir/V$(MODULE): .stamp.verilate
        @echo
        @echo "### BUILDING SIM ###"
        make -C obj_dir -f V$(MODULE).mk V$(MODULE)

.stamp.verilate: $(MODULE).v tb_$(MODULE).cpp
        @echo
        @echo "### VERILATING ###"
        verilator  -Wall -Wno-UNUSEDSIGNAL --trace --x-assign unique --x-initial
unique -cc $(MODULE).v --exe tb_$(MODULE).cpp
        @touch .stamp.verilate

.PHONY:lint
lint: $(MODULE).v
        verilator --lint-only $(MODULE).v

.PHONY: clean
clean:
        rm -rf .stamp.*;
        rm -rf ./obj_dir
        rm -rf waveform.vcd
```

9.2 tb_alu.cpp

```
#include <stdlib.h>
#include <iostream>
#include <cstdlib>
#include <memory>
#include <verilated_vcd_c.h>
#include <verilated.h>
#include "Valu.h"
#include <iostream>
#include <bits/stdc++.h>
#include <list>
#include <bitset>
#include <algorithm>
#include <chrono>
using namespace std::chrono;
using namespace std;


//#define FAST
#define BRUTE_BITS 0
#ifndef FAST
#define BIT_TIME 120500
#else
#define NODUMP
#define BIT_TIME 1
#endif
#define MAX_SIM_TIME (BIT_TIME * 2) * 2
#define VERIF_START_TIME 7


vluint64_t sim_time = 0;
vluint64_t posedge_cnt = 0;
vluint64_t btnprss_cnt = 0;


uint tryPasswd(std::vector<bool> bitset, int taskNumber, std::string filename)
{
        vluint64_t sim_time_begin = sim_time;
        posedge_cnt = 0;
        btnprss_cnt = 0;
        std::vector<bool> theseBits;
        uint64_t combinations = pow(2, 0) + 1;


        Valu *dut = new Valu;
#ifndef NODUMP
        VerilatedVcdC *m_trace = new VerilatedVcdC;
        dut->trace(m_trace, 5);
        m_trace->open(filename.c_str());
#endif
        int bits = 0;
        while (sim_time - sim_time_begin < (bitset.size() + 4) * MAX_SIM_TIME)
        {

                dut->iCLK ^= 1;

                if (dut->iCLK == 1)
                {
                        if (posedge_cnt % (BIT_TIME * 2) == 0)
                {
                if (bits % 9 == 0)
                {
                        dut->iBTN1 = 0;
                        dut->iBTN2 = 0;
                        dut->eval();
                        bits += 1;
                }
                else
                {
```

```
                        bool val = bitset[btnprss_cnt];
                        bits += 1;
                        if (val)
                        {
                                dut->iBTN2 = 0;
                                dut->iBTN1 = 1;
                                dut->eval();
                        }
                        else
                        {
                                dut->iBTN1 = 0;
                                dut->iBTN2 = 1;
                                dut->eval();
                        }
                                btnprss_cnt++;
                        }
                        }
                        else if ((posedge_cnt) % BIT_TIME == 0)
                        {
                                dut->iBTN1 = 0;
                                dut->iBTN2 = 0;
                                dut->eval();
                        }
                        else
                        {
                                dut->eval();
                        }

                        posedge_cnt++;
                }
                else {
                        dut->eval();
                }
#ifndef NODUMP
                m_trace->dump(sim_time);
#endif
                sim_time++;
        }
        if (dut->iLED1 == 1)
                return 1;
#ifndef NODUMP
        m_trace->close();
#endif
        delete dut;
        return 0;
}


bool appendPattern(std::vector<bool> &bits, std::string pattern)
{
        for (int i = 0; i < pattern.length(); i++)
        {
                bits.push_back(pattern[i] - '0');
        }
        return true;
}
std::string intToBitString(int num)
{
        std::string bitStr;
        bool foundOne = false;
        for (int i = sizeof(int) * 8 - 1; i >= 0; i--)
        {
                if (num & (1 << i))
                {
                        foundOne = true;
                        bitStr += '1';
                }
                else if (foundOne)
                {
                        bitStr += '0';
                }
        }
        return bitStr;
}
```

```
int main(int argc, char **argv, char **env)
{
        Verilated::commandArgs(argc, argv);
        Verilated::traceEverOn(true);
        std::vector<bool> bits;
        std::list<std::string> logs;
        appendPattern(bits, "01110000001101000110100101001110001000001");
        auto start = high_resolution_clock::now();

        for (int i = 0; i < pow(2, BRUTE_BITS) + 1; i++)
        {
                std::vector<bool> currbits = bits;
                std::string bitstring = string();


                if (i > 0)
                {
                        bitstring = intToBitString(i).substr(intToBitString(i).find('1'));
                }
                else
                {
                        bitstring = "0";
                }
                std::reverse(bitstring.begin(), bitstring.end());

                appendPattern(currbits, bitstring);
#ifdef NODUMP
                tryPasswd(currbits, i, "filename");
                sim_time += BIT_TIME * 2;
        }
        auto stop = high_resolution_clock::now();
#else
                std::string filename = "waveform";
                filename += std::to_string(i);
                filename += ".vcd";


                tryPasswd(currbits, i, filename);
                logs.push_back(filename);
                sim_time += BIT_TIME * 2;


        }
        auto stop = high_resolution_clock::now();
        std::string line;
        std::ofstream outputFile("combined.vcd");
        std::ifstream file("waveform0.vcd");
        while (std::getline(file, line))
        {
                outputFile << line << std::endl;
                if (line.find("$enddefinitions $end") != std::string::npos)
                {
                        break;
                }
        }
        file.close();


        for (auto it : logs)
        {
                cout << "working on file " << it << "\n";
                std::ifstream file(it);
                while (std::getline(file, line))
                {
                        if (line.find("$enddefinitions $end") != std::string::npos)
                        {
                        break;
                        }
        }
        outputFile << file.rdbuf();
        file.close();
        }
        outputFile.close();
#endif
```

```
        auto duration = duration_cast<microseconds>(stop - start);


        cout << duration.count() << std::endl;
        posedge_cnt = 0;
        btnprss_cnt = 0;
}
```