Preamble

This is my first participation in TinyTapeout. I discovered TinyTapeout when I came across the lecture *"KV Entwurf Integrierter Schaltungen"* while looking for courses to complete some additional ECTS credits during my master's studies at JKU Linz.

IC design is a completely new field for me, as I am actually studying Mechatronics with a focus on electrical machines and power electronics. Even though I had little knowledge about digital circuit design and no prior experience with Hardware Description Languages (HDLs), I was immediately fascinated by the concept of TinyTapeout. That's why I decided to take this course and participate in TinyTapeout.

This document serves both as the GitHub documentation for my project and as the report for the lecture. Depending on where you are reading this, some sections may therefore appear slightly over-explained.

The Project Idea

Project Restrictions

After receiving an initial introduction to TinyTapeout during the workshop on 24.09.2025 at JKU, the next challenge was to come up with a project idea.
In the lecture, we were relatively free to choose a project we wanted to work on. The only requirements were that it had to comply with the specifications provided by TinyTapeout (TinyTapeout – Specs) and that it should be interesting enough to form the basis of a report with a minimum length of 20 pages.

The specifications provided by TinyTapeout are summarized in the following table:

Specification	**Value**
Circuit Complexity	approx. 1000 logic gates
Clock Frequency	adjustable max. 66 MHz
Digital Inputs	8
Digital Outputs	8
Switchable Inputs/Outputs	8
Analog Inputs/Outputs	6
PMOD PCBs	USB, VGA, PS/2, …

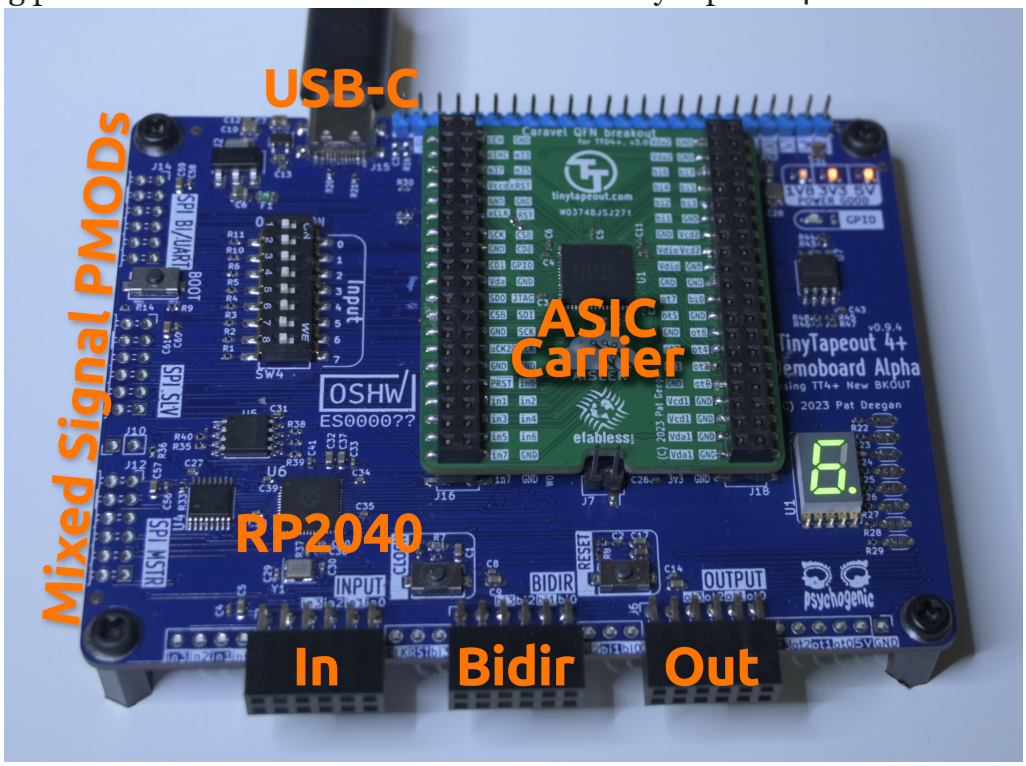
While specifications such as the number of inputs and outputs or the adjustable clock frequency were clear, I had almost no intuition about how large a project with approximately 1000 logic gates could be. For a rough estimation, I looked through older TinyTapeout projects.

While browsing through these designs, the next crucial question arose:
Is it possible to learn enough about Hardware Description Languages before the tapeout deadline in a little more than a month to implement a functioning and interesting chip design?

Project: Can You Count Binary?

Taking these challenges into account, I decided that my project should be a simple game that can be played directly on the TinyTapeout PCB without requiring any additional hardware. The idea behind using only the integrated peripherals was to make testing straightforward and to encourage more people to try the game.

The following picture shows the PCB that is used since TinyTapeout 4.



The PCB features 8 DIP switches connected to the digital inputs and a seven-segment display connected to the digital outputs. The bidirectional I/O pins as well as the analog I/Os remain unused in this project.
Given these limited peripherals, the range of possible game concepts was already quite restricted. The game idea I developed works as follows:

- A random 8-bit decimal number is displayed digit by digit on the seven-segment display.
- The player must convert this number into its binary representation and enter it using the DIP switches. The player has only a limited amount of time for this.
- If the correct binary value is entered within the time limit, a new number is shown; otherwise, the game ends and the player's score is displayed.
- To increase the difficulty, the available time decreases as the score increases.

⚙ Top-Level I/O

Defining the top-level I/O was fairly straightforward, as the standard peripherals included on the PCB are used.
The only parameter that needed to be chosen was the clock frequency. Since the design does not

require high-speed inputs or precise timing, a relatively slow clock frequency of 1 kHz was selected.

The PCB also features a reset button. It is normally pulled high and goes to GND when pressed. The modules implemented later in the design are therefore expected to perform a reset when this signal is activated.

Signal	Dir	W	Description
`ui_in[7:0]`	in	8	**DIP switches** (player input)
`uo_out[7:0]`	out	8	**Seven-segment display** (parallel 8-bit)
`uio_in[7:0]`	in	8	Unused
`clk`	in	1	System clock (1 kHz)
`rst_n`	in	1	Asynchronous reset (active-low)
`ena`	in	1	Always `1` on Tiny Tapeout

🪛 External hardware

No external hardware is required.

🧑‍💻 Implementation

Used Tools

For the development of the project the [IIC-OSIC-TOOLS](GitHub - iic-jku/IIC-OSIC-TOOLS: IIC-OSIC-TOOLS is an all-in-one Docker image for SKY130/GF180/IHP130-based analog and digital chip design. AMD64 and ARM64 are natively supported.) were used. The IIC-OSIC-TOOLS are an all-in-one Docker/Podman container for open-source-based integrated circuit designs for analog and digital circuit flows. They were created and are maintained by the Department for Integrated Circuits (ICD), Johannes Kepler University (JKU).

The hardware description language used within the IIC-OSIC-TOOLS is *Verilog*. The toolset includes various software components that support a smooth workflow when developing Verilog modules. The most relevant tools used in this project include:

- **Verilator** – Verilog code linter
- **Icarus Verilog** – Verilog compiler and simulator
- **GTKWave** – Waveform viewer
- **YOSYS / ABC** – Verilog synthesis toolchain
- **Magic** – Layout editor
- **LibreLane** – Digital RTL-to-GDS flow

Additionally, several provided shell scripts automate common workflows and simplify the development process.

For writing the Verilog code, Visual Studio Code was used together with the Verilog HDL extension, which offers syntax highlighting, helpful code snippets, and other useful features.

Modules

To simplify the development process, the project was divided into several smaller components, each implemented as an individual *Verilog* module. I first focused on the simplest modules to gain initial experience with Verilog before moving on to the more challenging ones.

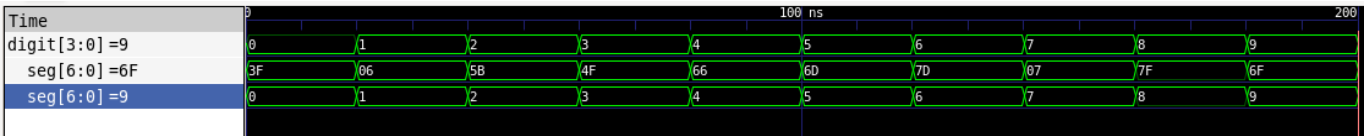
In addition to the modules themselves, a dedicated testbench was developed for each module. A testbench provides the ability to apply a sequence of input signals to a module and observe its outputs in a waveform viewer such as *GTKWave*. This approach significantly improves debugging and verification during development.

sevenseg_decoder

The **sevenseg_decoder** is a simple module that receives a digit as a 4-bit input and decodes it for display on the seven-segment display. The output is stored in a 7-bit register. The point of the display is not used in this project.
The decoding is implemented using a `case` statement. If the input does not represent a valid digit, the display remains dark (default case).

```
module sevenseg_decoder (
    input wire [3:0] digit,
    output reg [6:0] seg
);
    always @(*) begin
        case (digit)
            4'd0: seg = 7'b0111111;
            4'd1: seg = 7'b0000110;
            4'd2: seg = 7'b1011011;
            4'd3: seg = 7'b1001111;
            4'd4: seg = 7'b1100110;
            4'd5: seg = 7'b1101101;
            4'd6: seg = 7'b1111101;
            4'd7: seg = 7'b0000111;
            4'd8: seg = 7'b1111111;
            4'd9: seg = 7'b1101111;
            default: seg = 7'b0000000;
        endcase
    end
endmodule
```

To test the module, a dedicated testbench was written that applies all digits sequentially as input. The results of the simulation were inspected in **GTKWave**.
The input is displayed in decimal form, while the output appears both in binary and through a *Translate Filter Process* as the decoded decimal digit reconstructed from the segment pattern.



The *Translate Filter Process* is a simple pyhton script.

```python
#!/usr/bin/env python3
import sys

def transform(value):
    # 7-Segment Mapping
    seg7_map = {
        0x3F: "0",
        0x06: "1",
        0x5B: "2",
        0x4F: "3",
        0x66: "4",
        0x6D: "5",
        0x7D: "6",
        0x07: "7",
        0x7F: "8",
        0x6F: "9",
    }

    try:
        # Hex-String zu Integer konvertieren
        if value.startswith('0x'):
            int_val = int(value, 16)
        else:
            # Falls ohne 0x-Präfix
            int_val = int(value, 16) if all(c in '0123456789ABCDEF' for c in value)
else int(value)

        # Übersetzung
        return seg7_map.get(int_val, "?")

    except (ValueError, TypeError):
        return "?"

def main():
    fh_in = sys.stdin
    fh_out = sys.stdout

    while True:
        # incoming values have newline
        l = fh_in.readline().strip()
        if not l:
            return 0

        # outgoing filtered values must have a newline
        filtered_value = transform(l)
        fh_out.write("%s\n" % filtered_value)
        fh_out.flush()

if __name__ == '__main__':
    sys.exit(main())
```

bcd_spliter

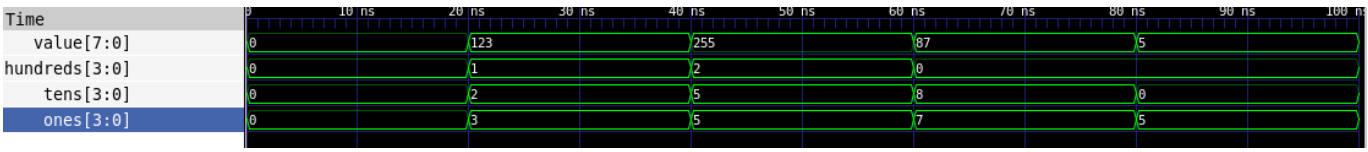
The **bcd_splitter** module separates an 8-bit binary-coded decimal value (0 to 255) into its hundreds, tens, and ones digits. All inputs and outputs are wires. To extract the individual digits, division and modulo operations are used. If the number does not contain a hundreds or tens digit, the corresponding output is set to zero.

Although each calculated digit requires only 4 bits, the division and modulo results must first be assigned to an 8-bit wire. Directly slicing the result results in a compilation error. The values are therefore stored in intermediate 8-bit wires, from which only the lower 4 bits are forwarded to the outputs. The unused upper bits produce linter warnings, which can safely be ignored.

```
module bcd_splitter (
    input wire [7:0] value,
    output wire [3:0] hundreds,
    output wire [3:0] tens,
    output wire [3:0] ones
);
    // xxxx_full wires are required because for example "hundreds = (value / 8'd100)
[3:0]" leads to an error (Linter warnings can be ignored)
    wire [7:0] hundreds_full = value / 8'd100;
    wire [7:0] tens_full = (value / 8'd10) % 8'd10;
    wire [7:0] ones_full = value % 8'd10;

    assign hundreds = hundreds_full[3:0];
    assign tens = tens_full[3:0];
    assign ones = ones_full[3:0];
endmodule
```

In the testbench for this module, several random values are applied as inputs. The resulting outputs are then inspected using *GTKWave*.



digit_selector

The **digit_selector** module is responsible for determining which digit should be displayed at a given moment in order to represent a three-digit number on a single seven-segment display. Since the hardware can show only one digit at a time, multiplexing is required.
The chosen solution is to display the hundreds, tens, and ones digits sequentially, each for a short period of time. A brief pause between digits is necessar. Without this pause, numbers such as *111* would appear as if the program had frozen, because the display output would not visibly change.

For numbers with fewer than three digits, leading zeros are displayed. For example, the number *3* is shown as 0 – 0 – 3.

The **digit_selector** inputs are the *clk*-signal, the *rst*-signal and a *trigger*-signal. The outputs are a 2-bit register with the current display-state and a register that indicates that the number was fully displayed. The display state can have 4 values:

- 0 - hundrets digit displayed
- 1 - tens digit displayed
- 2 - ones digit displayed
- 3 - pause

The register *rst* is not the direct signal from the PCB-button, but the inverted signal. Therfore if *rst* is high a reset should be performed. This is the same in all later modules.

The module also has some internal regs like a 10-bit *counter*-register, the reg *active* that stores if there is currently a number displayed, a reg *pause* that is one if there is currently a pause between to digits and a reg *digit* which stores which digit was shown last.

The timings are choesen that every digit is displayed for 1 second and that the pause between the digits is half a second.

Because the sourcecode of this module is quite long and also a flow chart of the module is not very lucid the process of the module is explained in text:

1. The module start

```
module digit_selector (
    input wire clk,
    input wire rst,
    input wire trigger,
    output reg [1:0] state = 2'b11,
    output reg done  = 1'b0
);

    reg [10:0] counter = 0;
    reg pause = 0;
    reg active = 0;
    reg [1:0] digit = 0;

    localparam DISPLAY_TIME = 1000; // 1 second
    localparam PAUSE_TIME = 500;  // 0.5 seconds

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 0;
            pause <= 0;
            active <= 0;
            digit <= 0;
            state <= 2'b11;
            done <= 1'b0;
        end else begin
            done <= 1'b0;

            if (!active) begin
                if (trigger) begin
                    active <= 1;
                    pause <= 0;
                    digit <= 0;
                    state <= 2'b00;
                    counter <= 0;
                end
            end else begin
                counter <= counter + 1;

                if ((!pause && counter >= DISPLAY_TIME) || (pause && counter >=
PAUSE_TIME)) begin
                    counter <= 0;

                    if (!pause) begin
                        pause <= 1;
                        state <= 2'b11;
                    end else begin
                        pause <= 0;
                        if (digit == 2) begin
                            active <= 0;
                            state <= 2'b11;
```

```
                            done <= 1'b1;
                        end else begin
                            digit <= digit + 1;
                            state <= digit + 1;
                        end
                    end
                end
            end
        end
    end
endmodule
```

sevenseg_display_controller

random_number_generator

timer

blink_controller

tt_um_dip_switch_game_TobiasPfaffeneder

🕹 Testing the game on Wokwi

https://wokwi.com/projects/446871385453862913

▶ How to Play

1. Power up your Tiny Tapeout PCB with this module loaded.
2. To start the game, bring all DIP switches except the 8th one in the **OFF** position.
3. A random 8-bit number appears on the seven-segment display.
   - The number is always shown as three digits (e.g. `123`, `045`, `007`, ...).
4. Convert the decimal number to binary and enter it using the DIP switches:
   - **Switch 8** = Least Significant Bit (2^0)
   - **Switch 1** = Most Significant Bit (2^7)
5. If your input is correct, a new random number will be displayed.
6. Be quick! You start with **30 seconds per number**, and the timer gets shorter as the game progresses.
   - If the timer runs out before you enter the correct value, the game ends.
7. When the game is over, your **final score** will be shown on the display.
