1 Introduction

wordle[1] is a web-based word-guessing game emerged and gained popularity during the COVID-19 pandemic. The rules of the game are simple: Everyday, there is a golden key to be guessed by players all over the world, which is a 5-letter English word. Each player has at most 6 chances to guess, and each guess has to be a valid 5-letter English word. After entering a guess, the game gives the player a hint on how close their guess was to the golden key, consisting of 5 colored tiles. A green tile means that the corresponding letter was correctly guessed and was at the correct position. A yellow tile means that the corresponding letter appears in the golden key, but is at a different position. A black tile means that the corresponding letter does not appear at all in the golden key.

In this TP, we will implement a similar game on the Gecko4Education board. As entering letters is difficult for the board, in our game, we guess prime numbers instead of English words — this gives rise to its name, *primedle*.

2 System Specifications

The input/output mapping on the Gecko4Education board is shown in Figure 1. A new golden key is generated randomly every time the reset button is pressed. For easier testing and debugging, the golden key can also be specified using the first (left) row of DIP switches. In this case, it is the tester's responsibility to make sure that the specified golden key is indeed a prime number (otherwise the game can never be won) and that it will never be changed during the game (i.e. after pressing the reset button). To play the game with a random golden key, simply set all the switches to 0 (off) and then press the reset button.

The player uses the second (right) row of DIP switches to enter an 8-bit binary number as their guess, which will be interpreted as a 4-digit number in base-4 by grouping two bits together. During game play, the guess is shown in real time on the seven-segment displays. A guess is checked after pressing the check button. If the number is not a prime, the rejected LED will turn on and no hints will be given. The last three rows of the LED matrix records how many times the check button has been pressed (with a maximum of 30). If the guessed number is a prime, it will be compared against the golden key, digit by digit, and the result, consisting of 4 colored tiles in a row, is shown on the LED matrix (with a maximum of 5 rows). A "colored" tile is

[1] `https://www.nytimes.com/games/wordle/index.html`

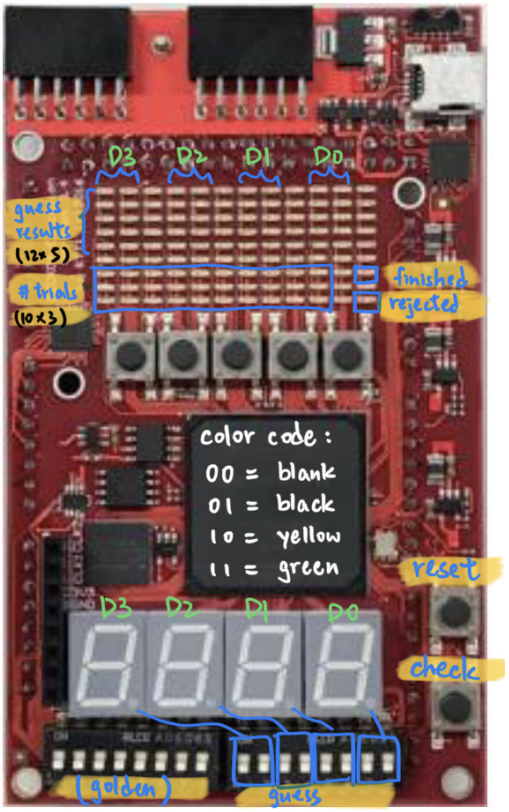

Figure 1: I/O mapping of the Gecko4Education board.

shown with two LEDs encoded as in Table 1. Similar to *wordle*, a "green" tile (encoded by `11`) means it was a correct digit at the correct position; a "yellow" tile (encoded by `10`) digit means it was a correct digit but at the wrong position; a "black" tile (encoded by `01`) means this digit does not appear in the golden key.

The game ends when either

- the golden key is guessed, or
- the check button is pressed 30 times (i.e., including valid and invalid guesses), or
- 5 rows of hints have been given (i.e., 5 valid guess).

After the game has ended, the seven-segment displays should show the golden key and ignore any further guess, until the reset button is pressed and the game restarts.

Table 1: Encoding of the colored tiles.

Color	Encoding	Meaning
White (blank)	`00`	Not used yet
Black	`01`	Wrong digit at wrong position
Yellow	`10`	Correct digit at wrong position
Green	`11`	Correct digit at correct position

3 VHDL Module Interfaces

This section describes the structure of the provided VHDL code skeleton and gives you some hints on how the complicated system can be divided into small modules. Each task (module) is tested separately in the auto-grader, so you can conquer them one by one. Before submitting any code, simulate it with Modelsim as in TP5 and observe the waveform to make sure it works correctly.

3.1 Data Types

To manipulate the game results (i.e. the hints history) more easily, we define some custom data types in `game_types_pkg.vhd`. First, `color_t` is an alias of a 2-bit vector which holds one color tile. Then, `hint_t` is an array of 4 `color_t`'s which represents a row of hint shown after a valid guess. Finally, `history_t` collects the 5 `hint_t`'s during the game. These data types are summarized in Table 2.

Table 2: Data types to store game results.

Name	Content	Meaning
`color_t`	`std_logic_vector(1 downto 0)`	A colored tile
`hint_t`	`array(3 downto 0) of color_t`	A row of hint
`history_t`	`array(0 to 4) of hint_t`	The complete history of a game

3.2 Input Interfaces

There are two types of inputs on the board that we use in this TP: the DIP switches and the buttons. Both of them are active low, meaning that when

turned on (for switches) or pressed (for buttons), they give a low (logic 0) signal. So, we first negate them in the top-level design (`primedle-rtl.vhd`).

The switches are usually stable, so we can use them directly. The buttons, on the other hand, need to be debounced.

There are two problems with the physical buttons that we must address here: 1) the user is free to press the button at any moment they desire, which may fall into the $[T_{su}, T_h]$ window of some flip-flop, leading it to metastability and 2) due to imperfection of the metal surfaces of the switch contacts and/or tilting of the contact surfaces on the spring, it can happen that while the user is applying force to the button, the current flow gets established and broken multiple times, resulting in multiple transitions of the signal at the output; this phenomenon which is called *switch bouncing* is illustrated in Figure 2.[2] The circuit of Figure 3 solves both of these issues.

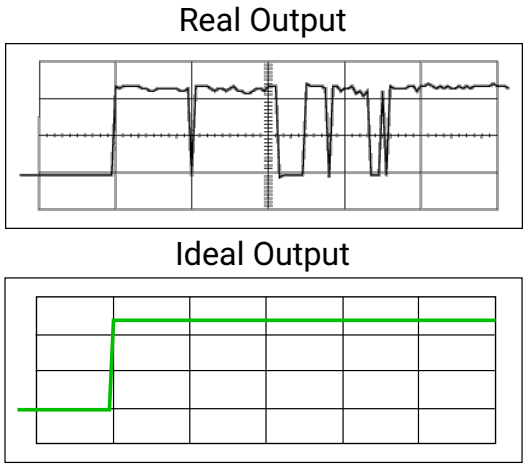

Figure 2: Output of a real mechanical switch illustrating the phenomenon of switch bouncing.

Even after synchronization and debouncing, there is yet another problem. The system operates at 12 MHz clock frequency, leaving little time for the user to lift their finger from the button before the current press is detected again. Thus, we would like the `clear` input to the debouncer be held 1 for a longer time. For this purpose, we need to integrate a timer to delay the button press detection.

[2]Courtesy of Maxim Integrated (`https://www.maximintegrated.com/en/design/technical-documents/app-notes/2/287.html`).

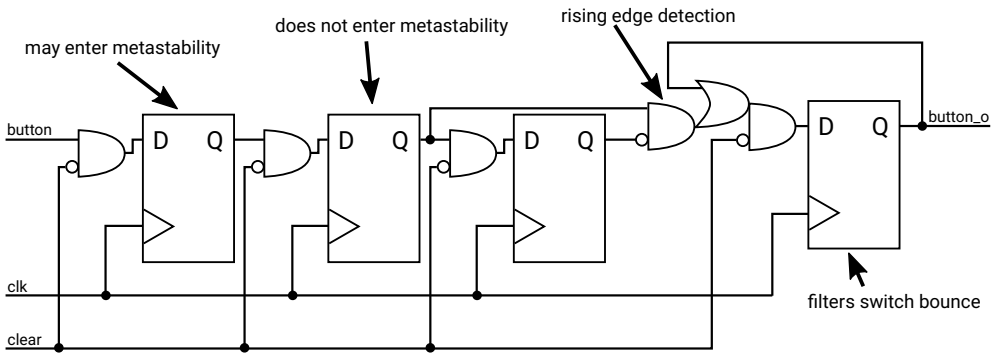

Figure 3: Switch synchronization and debouncing circuit.

Task 1 Press debouncing and delaying (10pts)

- Implement the debouncer as in Figure 3.
- Implement a timer that counts 12M clock ticks (i.e. one second).
- Integrate the timer and the debouncer so that when the *clear* input is asserted by the system, it remains active for another second, thus giving the user enough time to lift the finger before making another press. (This is done for you in `timed_button`.)

3.3 Output Interfaces

The game presents its state to the player via the two output interfaces on the board, the seven-segment displays and the LED matrix. As the signals ultimately connected to these hardware are simply vectors of bits, which are not easy to understand or manipulate by human, we need some output-driving modules to interpret the internal data of the game and convert it into output signals.

For example, the LED driver maps the meaningful game data of different types into a vector of 108 bits connected to the board's LED matrix. The input/output interface of this module is listed in Table 3. The LED driver module is provided to you.

Next, we need a decoder module to convert a base-4 digit (2 bits) into a vector of 8 bits corresponding to the 8 LEDs on a seven-segment display (the last bit is the point), similar to what we did in TP1. The input/output interface of this module is listed in Table 4.

Table 3: Interfaces of the LED driver module.

I/O	Pin name	Type	Description
I	`game_records`	`history_t`	The hint history so far
I	`trials_count`	`std_logic_vector (1 to 30)`	How many times the check button has been pressed; in thermometer encoding
I	`finished`	`std_logic`	Whether the game has finished
I	`rejected`	`std_logic`	Whether the last checked number was rejected (i.e. not a prime)
I	`enabled`	`std_logic`	Enables the LED matrix
O	`leds`	`std_logic_vector (0 to 107)`	The signals connected to the board's LED matrix

Table 4: Interfaces of the seven-segment display decoder module.

I/O	Pin name	Type	Description
I	`digit`	`std_logic_vector (1 downto 0)`	The 2-bit digit to be decoded
O	`seg`	`std_logic_vector (0 to 7)`	The signals connected to the seven-segment display, in the order of (a, b, c, d, e, f, g, p)

Task 2 Seven-segment display decoder (10pts)

This module converts a 2-bit digit into a vector driving a seven-segment display. The I/O interface is summarized in Table 4.

The decoder module will be instantiated four times for the four seven-segment displays. The input digit should come from 2 bits of the user's guess during the game, and from the golden key after the game finishes. (See Section 3.7.)

Table 5: Interfaces of the prime checker module.

I/O	Pin name	Type	Description
I	`clk`	`std_logic`	The system clock
I	`reset`	`std_logic`	The system reset
I	`guess`	`std_logic_vector (7 downto 0)`	The number to be checked
I	`check`	`std_logic`	Triggers the checking computation
O	`is_prime`	`std_logic`	Whether the number is a prime
O	`ready`	`std_logic`	Signals that the checking computation is done

- `guess` is guaranteed to be stable after `check` is raised and before `ready` is raised (thus no need to register it).
- Initially, `is_prime` should be 1 and `ready` should be 1.
- During computation, `ready` should be 0 and `is_prime` is not observed.
- When done, `ready` should stay at 1 and `is_prime` should be stable, until `check` is raised again.
- Even if the computation takes no clock cycle, `ready` should still go to 0 for at least one cycle after `ready` is raised (because it is used to clear the button debouncer).

3.4 Prime Checker

We need to check if the player's guess is a valid prime number before giving any hint, just like in *wordle*. This might sound like a complicated mathematical problem, but we can approach it from a different angle. How does *wordle* check if a word is a valid English word? It looks it up in a dictionary. When the input number is restricted to 8 bits, we can have a "prime dictionary" too! A list of prime numbers smaller than $2^8 = 256$ is prepared for you in `primes_pkg.vhd`. Now, the remaining problem is how to perform the dictionary look up.

There are several possible ways:

- Purely combinational: A big 8-input Boolean function.
- Huge finite state machine: A 54-state FSM going through the 54 primes in the list and comparing the guess against each of them.

- A mix of the two extremes. For example, you can divide the cases and enter different (smaller) finite state machines based on the value of the first few bits.

It is up to you to decide on which way to go. You can also come up with your own novel way, for example, using some math theorems and algorithms. Table 5 lists the input/output interface of the prime checker module and necessary expected behaviors for it to work properly with other modules.

Task 3 Prime checker (25pts)

This module checks whether a number is a prime. The I/O interface is summarized in Table 5.

3.5 Game Control

The game control module takes care of the operation of the entire game. The I/O interface of this module is summarized in Table 6.

The golden key (`golden`) is guaranteed to be stable during the game (i.e., after the first time the check button is pressed and before the reset button is pressed), so there is no need to register it.

This module identifies that a new guess (might be valid or invalid) is being entered using the `ready` signal from the prime checker. When waiting for the user input, `ready` is 1 (we keep all data as-is); after the player presses the check button, `ready` goes to 0 for one or more cycles (we still keep all data as-is, but knowing that we are waiting for the prime checker); finally, when `ready` comes back to 1, all inputs are prepared and we can update the game records. If the guessed number is invalid, we simply turn one more LED on in the trials counter and turn on the rejected LED; otherwise, we still turn one more LED on in the trials counter, turn off the rejected LED, and compare `guess` against `golden` to add one row of hint in `game_records`. We also need to check if the game is ended, and if so, the game control enters a dead state which keeps all data as-is and never update again, until the system resets.

It is okay for the updating computation to take more than one clock cycle. During updating, `LED_en` should be turned off, such that the unstable outputs will not be observed. A possible finite state machine is provided for your reference in Figure 4.

Table 6: Interfaces of the game control module.

I/O	Pin name	Type	Description
I	`clk`	`std_logic`	The system clock
I	`reset`	`std_logic`	The system reset
I	`guess`	`std_logic_vector (7 downto 0)`	The guessed number
I	`golden`	`std_logic_vector (7 downto 0)`	The golden key
I	`is_prime`	`std_logic`	Whether the guessed number is a prime
I	`ready`	`std_logic`	Whether the prime checker is done
O	`game_records`	`history_t`	The hint history so far
O	`trials_count`	`std_logic_vector (1 to 30)`	How many times the check button has been pressed; in thermometer encoding
O	`finished`	`std_logic`	Whether the game has finished
O	`rejected`	`std_logic`	Whether the last checked number was rejected (i.e. not a prime)
O	`LED_en`	`std_logic`	Enables the LED matrix

Task 4 Game control (30pts)

This module controls the entire *primedle* game. The I/O interface is summarized in Table 6.

3.6 Generating The Golden Key

The module `rand_gen` takes only the system clock and reset as inputs and generates a random prime number every time the reset signal goes from 1 to 0. Its output keeps stable when the reset stays at 0. The random generator works as follows: It has an internal counter i looping between 0 and 53 (there are in total 54 prime numbers smaller than $2^8 = 256$), incrementing with the clock cycle. Whenever a falling edge of the reset is detected, the module takes the value of i at the moment and keeps it in another register j. Without falling edge of the reset, the value of j stays stable. Finally, the

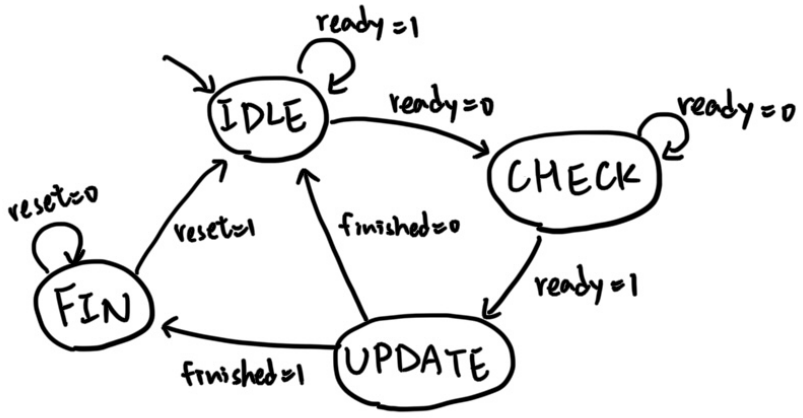

Figure 4: State diagram of the game control.

module simply outputs the j-th prime number in the list.

The implementation of the random prime generator is provided to you. The actual golden key is chosen between the output of the random prime generator (`golden_rand`) and the switches (`golden_in`), with the selection condition being whether `golden_in` is 0.

3.7 Inter-communication Among Modules

The top-level block diagram of the entire system is depicted in Figure 5. The system clock and reset signals are omitted for the sake of simplicity. In the top-level module, `primedle`, all we need to do is to instantiate each modules we have implemented in previous tasks and connect their inputs and outputs together. This module is completed and provided for you.

4 Simulation and Hardware Testing

To synthesize the bitstream, follow the instructions in TP6 to create a new Quartus project. Choose `primedle/quartus/` as the working directory. The top-level design is `primedle`. Remember to add all the files in `primedle/entities/`, `primedle/architectures/` and `primedle/packages/` to the project. Choose the same board as in TP6 (`EP4CE30F23C8`). The needed TCL scripts are already provided, so there is no need to download from the website. Simply run `primedle/quartus/primedle.tcl` and the pin assignments should be done. Click on the compile button to run the

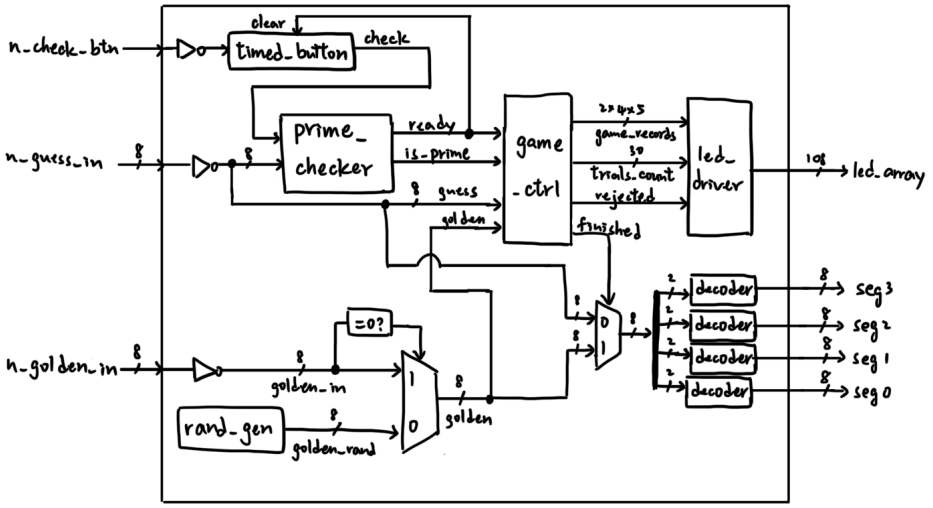

Figure 5: Block diagram of the primedle game system.

synthesis and generate the bitstream file. If the compilation is successful, you will find the SOF file in `primedle/quartus/output_files/`.

Task 5 Hardware demonstration (25 points)

Implement the machine on Gecko4Education and demonstrate that it really works as expected.

5 General Instructions and Rules

Please download the archive *primedle.zip*. It contains the entity files (suffixed with *-entity.vhd*) and architecture files (suffixed with *-rtl.vhd*) for all the modules used in this TP. To make sure that the automated grader recognizes the modules and interfaces correctly, please do not change any of the file, entity, architecture, or port names and write the architecture descriptions in the specified files only.

The solution is to be submitted for grading one module at a time, by uploading the architecture source file (*-rtl.vhd) to **https://digsys.epfl.ch**. Remember that you need a VPN connection if you are not on campus. The total number of uploads allowed is 20, which makes 5 attempts per task, on average. We do not impose a hard limit on the number of attempts per module. The only constraint is that the total does not exceed 20.

Each module is graded separately on Jenkins, thus partial points are possible. The full score after passing all testbenches is 75.

Partial points on hardware demonstration (Task 5) may be given by matching the result with one of the following cases:

1. The decoder module passes on Jenkins and the seven-segment displays react with the guess switches and present the correct numbers. (5 pts)

2. The previous item is fulfilled, and the prime checker module passes on Jenkins and the rejected LED reacts correctly when a guess is checked. (15 pts)

3. All modules pass on Jenkins and the entire system works correctly. (25 pts)