Prof. Dr.-Ing. Tim Güneysu
M. Sc. Florian Stolz
M. Sc. Moritz Peters

Ruhr-Universität Bochum
Faculty of Computer Science
Chair for Security Engineering

Winter Term 23/24

Computer Engineering 3 – Hardware-Oriented Programming

Project #3

Submit answers until 23. December 23:59:59 via *Moodle* Σ 100 Points

---

Christmas is right around the corner and people are looking for presents. Nowadays, having a child-friendly product is more important than ever. So LAME INC. tasks you with the important project to make the vacuum robot child-friendly. You realize that the prototype robot with its bumpers and LEDs makes it perfect to run the game SIMON [1].

Attention: No external library functions are allowed! All functions are located in `tasks/` and tests are provided in `main.c` . Please comment your code! Pay attention to the testbench!

Task 1: General Functions (25 Points)

To get SIMON running, we first need to implement some basic functionality. This includes some pseudo-random number generation to get a sequence of buttons to press, an array comparison to compare the user input to the generated sequence, and a delay function to make the LEDs blink with a certain frequency. Please pay attention to the definitions in `task1.h`!

a) Implement the function `delay` in `task1.s`. We generate a delay by doing nothing by certain amount of time specified by a parameter. This may, for example, be an empty loop. Note that one instruction takes approx. 20 ns so the specified delay may be a large number (>16 bit). The function does not have to be precise. (5 Points)

b) Implement the function `array_cmpn` in `task1.s`.! It compares *len* 8-bit elements of two arrays. So the arrays $x = \{1, 2, 3\}$ and $y = \{1, 2, 4\}$ are identical for `array_cmpn(x,y,2)` but not for `array_cmpn(x,y,3)`. (8 Points)

c) Implement the function `rng` in `task1.s`.! It generates *amount* pseudo-random 8-bit numbers ranging from 0-3 (effectively 2-bit numbers) by computing:

$$x = x \oplus (x << 13)$$
$$x = x \oplus (x >> 17)$$
$$x = x \oplus (x << 5)$$

and storing only the lower two bits in each iteration. This is computed in the loop. The initial x is provided by *seed*. The results should be stored in the *destination* array. (12 Points)

Task 2: I/O (50 Points)

Now we need to get some user input and also need to signal the sequence to press to the user by blinking the RGB LEDs. Please pay attention to the definitions in `task2.h`! You can find details about the necessary ports in the *board overview* as well as the *robot manual.*

a) Implement `ledInit` in `task2.s`. It initializes the RGB LED, which uses positive logic. The RGB LED consists of 3 LEDs (Red, Blue, Green) which are connected to P2.0 P2.1 and P2.2! Attention: Each LED requires the additional drive strength! (10 Points)

b) Implement `ledOut` in `task2.s`. It gets a parameter, which determines the RGB color to display. 0 corresponds to the RGB LED being shut off. (5 Points)

[1] https://www.youtube.com/watch?v=Bx7fl3QEw94

c) Implement `blinkSequence` in `task2.s`. It blinks out a sequence of colors provided by *arr* with length *len* by calling `ledOut`. *arr* contains numbers between 0-3, which represent 4 colors. You can map the colors as you like. 0 should represent a color and not the *shut off* state, so you need to map at least 0 to another color for this task. Between each color the LED should be off for some time (`ledOut(0)`). Use your `delay` function to generate a delay. At the end of the sequence the LED should be turned off. (10 Points)

d) Implement `bumperInit` in `task2.s`. It initializes BMP0, BMP2, BMP3, BMP5. Attention: Bumpers are negative logic (press = 0, unpressed = 1)! (10 Points)

e) Implement `bumperAwait` in `task2.s`. It waits for a button press and returns a number corresponding to the button being pressed (BMP0 = 0, BMP2 = 1, BMP3 = 2, BMP5 = 3). Implement it the following way: First enter a loop which waits for any button to be pressed. Then get the result. Afterwards wait until the button is not pressed anymore. Lastly, add a delay, for example, `delay(0xFFFFF)`. This prevents a button press to be recognized twice when reading buttons in tight loops. This is a rudimentary form of button debouncing. Please be aware, that the delay also stops you from quickly pressing the button sequence. If the delay is too small, a button press may be detected twice. Please press the button only for a very short amount of time. Attention: You do not have to account for two buttons being pressed at the same time! (10 Points)

f) Implement `bumperReadSequence` in `task2.s`. It reads *amount* button presses and stores the result in *arr*. (5 Points)

Task 3: Final Check (25 Points)

Now that we have all general functions and I/O, implement the main game loop. Please pay attention to the definitions in `task3.h`!

a) Implement `gameloop` in `task3.s`. It gets an input sequence with length *amount*. First it blinks out the sequence. Then it uses reserved local space to wait for and store *amount* button presses. Lastly, it returns the result of `array_cmpn` which compares the sequence and inputs provided. Use your previously implemented functions. (25 Points)

Attention: Please remember that the stack should always be aligned. So even if you only want to store one byte on the stack, please reserve at least 4 or a multiple of 4 bytes.