RA FS 18 Series 4

Qiyang Hu, Givi Meishvili, Adrian Wälchli

The fourth series has to be solved until Tuesday, 1. May 2018 at 3 p.m. and has to be uploaded to ILIAS. If questions arise, you can use the ILIAS forum at any time. Any problems should be communicated as soon as possible. We will be happy to help.
Have fun!

Theoretical Questions

Total score: 12 points

1 Sign Extension (1 point)

Explain why the sign extension is especially needed for branch-, load- and store-instructions.

2 Logical and Bitwise Operations (1 point)

Determine the corresponding outputs:

Input	Output
`0xA3 & 0x3A`	
`0x0 && 0xEF`	
`0xA3 | 0x3A`	
`0x0 || 0xEF`	
`~0xFE`	
`!0xFE`	

3 Hardware Access (1 point)

How could a processor get access to connected hardware?
Use the MIPS emulation as inspiration.

4 `bne` instead of `beq` (2 points)

What changes are needed in the MIPS implementation presented in the lecture (see "Basic MIPS Architecture Review") to implement `bne` instead of `beq`?

(a) For the singlecycle implementation

(b) For the multicycle implementation

5 Pipeline Registers (1 point)

What for are the registers (see slide 6, "Basic MIPS Pipelining Review") between the calculation levels needed?

6 Pipelining Hazard (2 points)

Explain the difference between control, data and structural hazards.

7 Stall (2 points)

Explain why on slide 15 it is enough to wait two clock cycles but not on slide 19 (the slide numbers refer to the chapter "Basic MIPS Pipelining Review").

8 Data Hazard (2 points)

Show all data hazards in the following code. Which dependencies are data hazards that can be resolved by forwarding? Which dependencies are data hazards that will lead to a *stall*?

```
1 add $t0, $t5, $t4
2 lw $s2, 0($t0)
3 sub $s3, $t0 ,$s2
4 sw $t4, 4($s3)
```

Programming Part

The programming exercises with the Raspberry Pi must be solved in groups of two or three students. You and your group members work together such that everyone contributes equally. In order to ensure that all participants understand the whole code they turn in, we require each student to hand in a slightly different version of the exercise/code. As described below, these versions differ only in a very specific functionality for which each individual student is responsible.

Preparation

(a) First make sure your Raspberry Pi functions properly and that you are able to compile and run assembly code. On ILIAS you find an appropriate tutorial on how to get started with the Raspberry Pi.

In case you received defective hardware, contact one of the assistants immediately.

(b) Follow the instructions in the Raspberry Pi tutorial document and test the demo program `blink.s`, which accesses the LED on the extension board. We recommend that you study the demo code carefully before starting with the exercises.

(c) Download the code skeleton from ILIAS.

Running Light

Your task is to complete the skeleton as described in the following exercises.

(a) Put your name and the names of your group members in the appropriate spaces provided in the file `knightRider.s`.

(b) Implement a running light on eight of the LED segments. The light starts at `LED0`, runs to `LED7`, then back to `LED0` and so on. In the appendix below you find additional information on how to access the necessary components on the breadboard.

> **Student 1:** You hand in the standard version as described.
>
> **Student 2:** You have to negate the display of the running light. This means that all segments but one are turned on at any given time.

(c) Extend your program such that pressing button `BUTTON1` increases the speed of the animation. `BUTTON2` should decrease the speed.

> **Student 3:** In your version of the code, `BUTTON2` is not used to decrease the speed, but to reset it to the initial value (without restarting the program).

(d) **Optional:** Add a second running light which moves in the opposite direction, meaning that the two lights cross each other in the middle of the LED bar.

Submission

(a) Make sure that your source code contains useful and detailed comments. This is an essential requirement to pass the programming exercise!

(b) Make sure that your program compiles without any errors or warnings. This is also an essential requirement to pass the programming exercise!

(c) Create a Zip file named `<firstname>-<lastname>.zip` with all the files, where `<firstname>` and `<lastname>` are to be replaced with your first- and last name respectively.

(d) Hand in your solution electronically by uploading the Zip file to ILIAS.

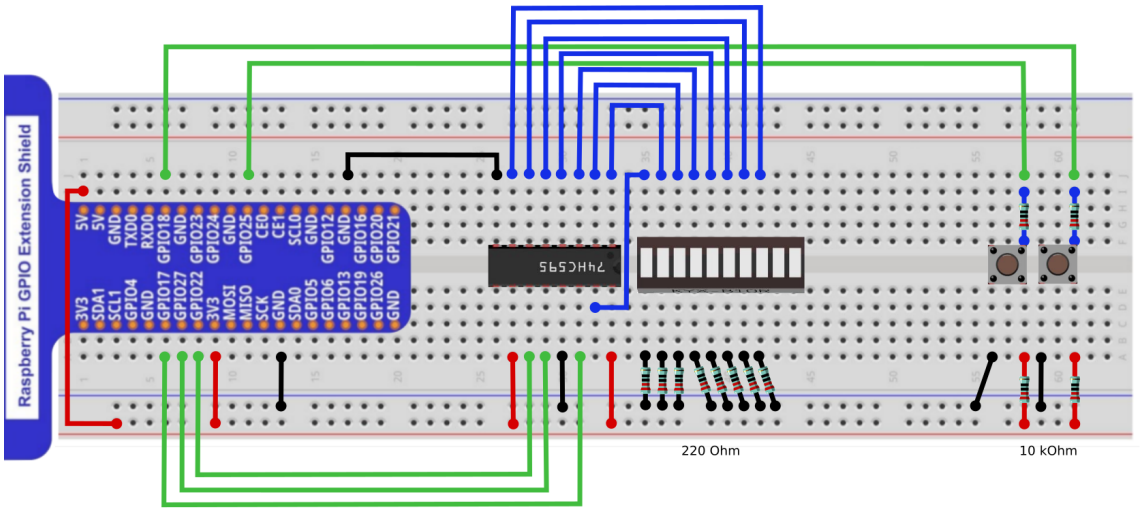

Figure 1: Circuit diagram for the LED bar (middle) with a serial-parallel converter (75HC595) and two buttons (right). Red: power connection to positive electrode (+). Black: power connection to negative electrode (−). Green: data connections for input- and output signals. Blue: Other internal data connections.

A Access to LED Bar

From the demo program `blink.s` you have already learned how to access individual pins via the GPIO. In principle it is possible to individually connect and control every pin of the LED bar shown in figure 1. However, this would make it a bit more complicated to implement the animation in the exercise. Therefore we use a serial-to-parallel converter (or shift-register) that reads a serial signal over a single connection and outputs it in parallel (blue cables).

As an example, we take the 8-bit long sequence 00100000. We would like to send this series of zeros and ones to the converter chip (known as 74HC595), which will then split the bits to individual output pins that are connected to the LED bar. For the example sequence above, the third LED (from the left) should be turned on.

In order to transfer the serial signal to the chip, we need three pins/connections (instead of eight):

- **LATCH PIN** The state of this pin determines whether a serial signal is transferred (LOW) or not (HIGH).
- **DATA PIN** This pin is used to transfer the sequential data bit by bit.
- **CLOCK PIN** The output on this pin changes periodically between HIGH and LOW.

These three pins correspond to the three green wires in the lower part of figure 1: For convenience, we provide the subroutine

```
void shiftOut(datapin, clockpin, firstbit, data)
```

that takes care of the data transfer. The first two arguments are the numbers of clock- and data pin. The parameter `data` is simply a number (the data). In our example it is $00100000_{(2)} = 2^5 = 32$. The parameter `firstbit` merely determines the bit significance of the data (0: LSB is transferred first, 1: MSB is transferred first).

Important: Before using `shiftOut`, the latch pin must be set to LOW to signalise a data transfer, and after `shiftOut`, the latch pin must be reset to HIGH.

To summarize our running example:

```
digitalWrite(27, 0)
shiftOut(17, 22, 0, 32)
digitalWrite(27, 1)
```

Part of the exercise is to translate these three subroutine calls into assembly code. *Hint:* The three pin numbers are already declared as constants in the skeleton code.

B Access to Pushbuttons

Figure 1 shows two pushbuttons connected to pin 18 (`BUTTON1`) and pin 25 (`BUTTON2`). To read the input signal on these pins, you can use the pre-defined subroutine `digitalRead(pin)` which has a binary return value (0: button down, 1: button up).

In the case of this exercise, we would like to take an action only when the pushbutton signal changes from HIGH to LOW (falling edge), i.e. the button's state changed from "up" to "down". We provide a subroutine `waitForButton`

```
[pressed, state] waitForButton(pin, timeout, state)
```

which you can use to wait for a falling edge. The argument `timeout` is the delay in milliseconds, and `state` is the (previous) state of the input pin (0 or 1). If a falling edge is detected within the given time window, the output variable `pressed` is set to 1, and 0 otherwise.

In part (c) of the exercise you have to invoke the subroutine with the correct arguments and change the behaviour depending on the return value. *Hint*: The two return values are stored in registers `R0` and `R1`.