Assignment 2

In this warm-up assignment, you will have to modify software and create a new component to be added to bus interconnect.
For this purpose, a new I/O-Controller must be designed and a previous software demonstration example rewritten to use the device.

Question 2.1

Your first task is to write a simple component to act as a Wishbone slave, and is used as an interface to the various switches and buttons on the development board.
For this purpose, create a file `switches.vhd` in the `soc/peripherals/` subdirectory in the git project directory. Create an entity and architecture declaration. Apart from system and bus signals, there are 15 input bits, as noted in the Genesys Reference Manual:

- 5 bits for the two-axis rocker switch, aka “joystick”
- 2 bits for the buttons left and right of the joystick
- 8 bits for the switches

a) Implement the wishbone slave. You can take `leds.vhd` as an example, but this component is slightly more complex and, in contrast to the led example, read-only.
The used protocol is a simple (classic) wishbone bus with synchronous cycle termination, as described in the SoC-documentation. You can consult the Wishbone Specification (version 4) for details. No extensions are used.
The proposed memory map is to feature a single 32 bit read-only register. The byte order is big endian. The least significant byte should feature the status of the switches, while the second least significant byte features the status of the buttons and the “joystick”. The upper two bytes are reserved for now and should return all zero upon a read access.
Do not forget to commit and push the created component to git repository.

b) The new component must be tested before integration. Create a testbench in the testbench subdirectory for the component. Simulate all single byte accesses, halfword accesses and the word access. The inputs for the switches and buttons should be varied throughout the testing. You may use the pre-defined procedure `generate_sync_wb_single_read` and others from the wishbone-testing-package.
Add it to the Xilinx ISE project (with “Add Source”) and make sure the simulation completes successfully.
Do not forget to commit and push the created testbench to git repository.

Question 2.2

After the design was sufficiently tested, it must be integrated in the system.

a) The top level entity is missing these signals in its port declaration, add signals as indicated in Listing 1:

Listing 1: Top Level Entity

```
entity lt16soc_top is
generic(
  ...
);
port(
  ...
  btn : in std_logic_vector(6 downto 0);
  sw  : in std_logic_vector(7 downto 0);
  ...
);
```

Uncomment the lines with “sw” and “btn” in the top.ucf file. The UCF is a “User Constraint”-file, which the Xilinx ISE uses to determine where the signals in the top level entity port list are routed on the physical device.

b) To add the component to the SoC, a few steps must be undertaken.

The component declaration of the new component should be added to the appropriate package. In the case of the switches and button peripheral this is the `lt16soc_peripherals` package in the `soc/peripheral/peripherals.vhd`. By doing so, there does not need to be a component declaration inside the top level entity architecture, making it more readable.

The component instantiation must be connected to the bus interconnect system. To do this, the port map must connect its inputs and outputs of its bus interface to a free element of the `slvi` and `slvo` array signals in the top level architecture. To select to which of those signals you should connect, consult the file `soc/lib/config.vhd`. There are a number of slave index constants like shown in Listing 2. Add your own by adding one to the last defined constant. It is recommended that you use your defined constant to select the element of the bus slave signal arrays, as illustrated in Listing. 3.

Listing 2: Slave bus indexes

```
-- >> Slave index  <<
constant CFG_MEM : integer := 0;
constant CFG_LED : integer := CFG_MEM+1;
constant CFG_DMEM : integer := CFG_LED+1;
```

Listing 3: Slave bus connection

```
... port map( ...
wslvi      => slvi(CFG_FOO),
wslvo      => slvo(CFG_FOO)
...
```

Another issue is the configuration of the component regarding its generics [1]. As seen on the led- and dmem-component, the wishbone slaves of the SoC feature a configurable memory address `memaddr`, and a configurable address mask `addrmask`.
The memory address corresponds with the base address of the component, while the address mask states which bits of the address are to be considered during component selection, which also determines the address range the slave has. All zeros in the mask will make the interconnect and component ignore that bit in the address when determining if the component is addressed.
Taken as an example, the led-component is configured as illustrated in Listing 4. Its base address is straightforward `0x000F0000`, but the first 10 bits are “masked out” to save logic, because the system does not use slaves beyond the address `0x003FFFFF` , yet. Also, the mask tells us that the address range of the component is a single 8-bit register, since no bit in the lowermost bits of the address mask is zero.
As starting address for the switch controller, `0x000F0004` is recommended. Determine the needed address mask on your own.

The slave masking vector is a constant which configures the interconnect system to only consider certain slave bus interface elements. Listing 5 shows the declaration of the slave mask vector constant in the top level architecture. To enable any connection on the interconnect, replace the the ’0’ with a ’1’ at the same position in the salve

[1] If VHDL generics are unknown to you, consider reading the appropriate chapter in “The Designer’s Guide to VHDL” or similar literature or consult online resources.

Listing 4: Slave bus indexes

```
package config is
...
constant CFG_BADR_LED   : generic_addr_type := 16#000F0000#;
...
constant CFG_MADR_LED   : generic_mask_type := 16#3FFFFF#;
...
```

mask vector as the element position in the `slvi` and `slvo` arrays. This enables the port inside the interconnection system.

Listing 5: Slave bus indexes

```
architecture RTL of lt16soc_top is
...
constant slv_mask_vector : std_logic_vector(0 to NWBSLV-1) := b"1110_0000_0000_0000";
...
```

After the component is integrated, run a synthesis to make sure no errors occur. Commit your changes to the top level entity and config to git.

Question 2.3

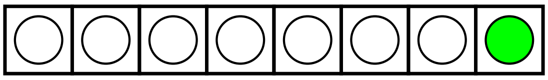
The minimal software used in the last assignment used a fixed prescaler to make the leds on the development board blink with a detectable speed. Your task now is to extend that fixed prescaler with the value derived from the setting of the on-board switches.
Create a copy of the given example assembler code file and name it `assignment2code.prog`. Add and commit it to the local git repository before doing anything else. Then, proceed to modify the software to incorporate the modifiable prescaler.
The prescaler should scale the frequency in which the leds blink linearly, as illustrated in the table below (P_b refers to the base prescaler value). This can easily be achieved by implementing a loop iterating through the inner `wait`-loop for a number of times represented by the value of the switches plus one.

Switches	Prescaler
0	P_b
1	$P_b * 2$
2	$P_b * 3$
3	$P_b * 4$
N	$P_b * (1 + N)$

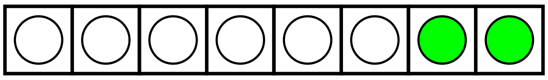
Assemble the code. Create a testbench and simulate the system with your code. Only if the simulation yields satisfiable results, synthesize the design, load and run it on the FPGA.
Demonstration of your solution as well as a ready explanation of your code is required to complete this assignment. Do not forget to commit and push your changes.

Question 2.4 - Challenge

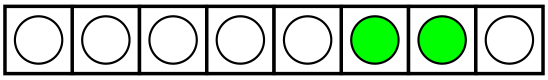
This assignment's *challenge* is to rewrite the software to use 2 prescalers and only display at maximum 2 light-up leds at the same time. Those 2 prescalers are to be used to move the 2 light-up led 'dots' independently. These two dots move according to their individual prescaler, in the same direction.
An illustration of the desired implementation is gifen in Figure 1. This figure assumes a prescaler of '1' and '0', and each step is a single run of the fixed prescaler.



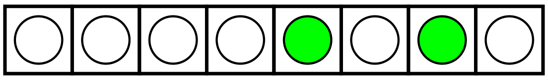
(a) initial positions



(b) After the first step



(c) After the second step



(d) After the third step

Figure 1: Exemplar behavior given a 2:1 prescaler ratio