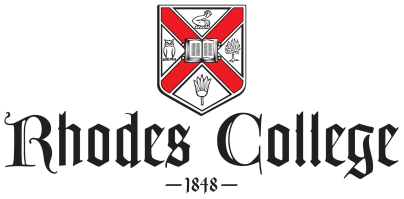
COMP 231
Lab 5



For this lab exercise, you will create a NIOS II CPU-based system as described below. Submit your entire Quartus project folder as a ZIP file. Do not use any other archive/compression formats other than ZIP.

1 Seven-Segment Display support

For the first part of the project, start by copying your `lights` Nios project from the Lab 4 tutorial to a new folder. We will be adding support for the seven-segment display devices and driving them with a software-based lookup table. First, we will need to add the hardware support and connections for the SSDs.

Return to the Qsys tool and add two new PIO (Parallel I/O) ports with a data path width of 8-bits and as an output port (Just like you did for green LEDs). Rename the first HEXA and the second HEXB. Make connections in Qsys to the Avalon bus in the same way as you did for the LED PIO port (clock, reset, etc.).

Next, you will need to re-assign base addresses for the new ports. This may change the earlier set values for the switches and LED ports. Take note of the memory-mapped addresses for the switches, LEDS, HEXA, and HEXB. You will need this information for the next section.

Lastly, ensure that you export the output for the HEXA and HEXB ports, the same way you configured the switches and LEDs. Generate the HDL from Qsys and head back to Quartus.

2 VHDL Modification

Now we need to connect the output wires from HEXA and HEXB on the NIOS processor to the input wires for HEX0 and HEX1 SSD hardware on the DE2. To do this, we will need to modify the VHDL file, `lights.vhd`, that we had created during the tutorial.
Qsys outputs a sample VHDL file to instantiate the Nios processor, given the names for the devices that you configured. You should verify the signal names in the sample file, which should be `nios_system/nios_system_inst.vhd`.

In the `lights.vhd` file, you need to make minor changes to several parts:

PORT Section: In this portion, we need to declare the interface to the DE2 components. Add lines to the VHDL file like so:

```
HEX0 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0);
HEX1 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
```

Ensure that all lines end in a semicolon, except for the last.

ARCHITECTURE Section: This section should match the information from `nios_system_inst.vhd`. Add new `SIGNAL` lines that correspond to the 8-bit output ports. They should have names like `hexa_export` and `hexb_export` and be placed in the `PORT` portion of the `ARCHITECTURE` block.

Next, we need to declare two 8-bit buses to connect the HEXA port to HEX0 and HEXB to HEX1. Add the following lines to the `ARCHITECTURE` block after the `END COMPONENT;` line:

```
SIGNAL HEXA : std_logic_vector(7 downto 0);
SIGNAL HEXB : std_logic_vector(7 downto 0);
```

BEGIN Section: Lastly we need to bring everything together and make all of the connections. Right after the `BEGIN` statement, add a sequence of mappings to connect the HEXA and HEXB wires to the HEX0 and HEX1 SSDs. For example:

```
HEX0(0) <= HEXA(0);
```

This statement attaches the low-order bit of the 8-bit HEXA wire to the low-order bit of the HEX0 SSD unit. Connect wires from HEXA to HEX0 and HEXB to HEX1. Both HEX0 and HEX1 only have seven input wires, so you only need to worry about wires $0 \ldots 6$.

Lastly, in the `PORT MAP` section, add mappings for `hexa_export` to map to `HEXA` and `hexb_export` to `HEXB`. **Ensure that you have been careful with your commas and semicolons** and save your file. Back in Quartus, re-compile the entire project. If you are successful, you are done with the hardware setup and are ready to work on the software side of the project.

3 Assembly Language Programming

The final part of this assignment involves writing an assembly language program that reads an 8-bit value from the switch input and displays the value in hexadecimal format on the SSD units (and on the green LEDs). Start by working from the `lights.s` file. As in the tutorial, create `.equ` macros for the addresses of the PIO ports. Create one each for `leds, switches, hexa,` and `hexb`. These may have changed with the addtion of the new PIO ports.

To display a value on the seven-segment display, you need to issue a `stbio` instruction to the address from the Qsys layout. The value you write to the address is a 7-bit value which corresponds to each segment of the display. Recall that the SSD device uses negative-asserted true logic, so a 0 bit value activates an LED segment and a 1 bit value deactivates a segment. For example, if we wanted to make an 8 display, we would write 7 zeroes to the display: `0x00`. To make only the top light segment display (segment 0), you would write all ones, except for the lowest bit (bit 0): `0xfe` or `0x7e` (the highest-order bit is ignored, since the SSD inputs are only seven-bit values).

Lookup Table: Rather than using the hardware circuitry that we created in Lab 2, we want to use a software lookup table to display the correct value on the SSD. To implement this, we will use a label, `lut:`, to mark the beginning of some constant data in our assembly language program.

To understand how a lookup table is implemented in assembly language, assume that we wanted to have a three-element array constant, `a`, with `a[0] = 5, a[1] = 6,` and `a[2] = 7`. We would arrange this in our assembly language program like this:

```
.data
a:
.word 5
.word 6
.word 7
.end
```

This code is placed after the end of our assembly program, before the `.end` marker. The label `a:` marks the address of the first element of the array (index 0). Each element is stored in a four-byte word, so to compute `a[2]`, we would use the base address stored in `a`, and add the size of each element (4 bytes) times the number of elements to skip over (2): $a_2 = \texttt{Mem[}\ a + (4 \times 2)\texttt{]}$

Construct a 16-element array, one for each possible 4-bit combination of values to display on an SSD. The value of each array element should be a 32-bit word, with the lowest 7-bits containing the bit pattern to illuminate the correct segments for the corresponding 4-bit hexadecimal value. For example, the value of `lut[8]` should contain the 7-bit pattern (0x00) to illuminate all segments on the display.

Once you have completed your array, modify the main loop driver to write the 8-bit value from the switches to both HEXA and HEXB. You will have to read the 8-bit value into two registers, use these values to determine the offsets into the lookup table, load the pattern values from the LUT and write them each to the correct SSD, as well as the LEDs.

As an example, let's say that the value read from the switches is `1111 0001` (i.e. `0xf1`). You need to move `0001` into one register and `1111` into another. Next, load the value at $\texttt{lut} + 4 \times 1$ into a new register and $\texttt{lut} + 4 \times 15$ into another. These registers should now contain the 7-bit patterns that can be written to the SSD using the `stbio` instruction.

Submission

Make sure you submit an entire project directory to Canvas. This should contain all of the Quartus II files. If you do not submit an fully functioning project, you will lose points. **Double check your work if you make changes to files and/or file locations.**