Sheffield Hallam University
Semester I of academic year 2021/22

"SoC architectures and FPGA prototyping"

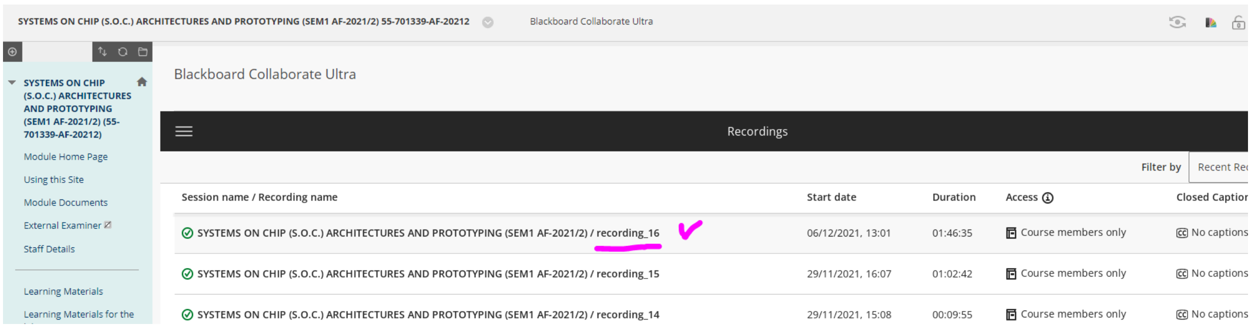
Lab 8 (optional) – *Adding HEX5..HEX0 peripheral to the Lab4 SoC*

In the lab 4 you have developed a peripheral that allowed controlling the LEDR[7]..LEDR[0] and read switches SW[9]..SW[0] on the DE10-Lite FPGA board by amending the LED2AHB.v module.

In the previous labs we also used the HEX5..HEX0 displays.

This lab is lightly structured, and you need to create another module (probably it will be better to start from the copy of the LED2AHB.v module) that will enable controlling all the a.m. segments. You will need to map this peripheral to some unused address space and write a C code to set **the last 6 digits of your student ID number** (the example development displays the Christmas date as 24.12.21).

A good discussion regarding the SoC that you have implemented, and how to start with the lab 8 is discussed in the recording shown below from 00:58:30 – 01:40:00:



01:14:00 - 01:22:00 – specifically relates to modification of the AHB2LED. However, I used all the 32 bits for the comparison that was inefficient. In fact, the address decoder already enabled the module for all the 16 MB of addresses, and there is no need to check the address bits [31:24] again. Moreover, we are not going to have 16 M separate byte registers thus we can decode the address in an easier way. I described the easier way below.

Light structure of what to do:

1. Copy your complete lab 4 folder and rename the copy to Lab_8.

2. Top module AHBLITE_SYS.sv:

- add HEX5..HEX0 ports to the module (with the correct directions and widths)

- because we use wildcard .* when instantiate the AHB2LED module, it is not necessary to do any other amendments provided that you use names HX%..HEX0 in the AHB2LED.v module.

3. Module AHB2LED.v:

- add HEX5..HEX0 ports to the module (with the correct directions and widths)

- add registers with the correct width for holding data to be displayed, one for each display (we need registers as we want the displayed data to remain visible until changed; the other reason is that we are to update these in a procedural block);

- add rHADDR register for sampling the 32 bit HADDR address at the address phase, and use non-blocking assignment to populate it at the address phase;

- below you can see the code for the data phase where I blanked some lines that you need to restore (18 lines altogether, namely 6 lines for RESET to light up all the HEX segments, 6 lines to write the data coming from the Cortex-M0 to the appropriate register thus making a demultiplexer – see appendix for details, 6 lines to drive the HEXes from the associated registers)

```
61  //Data Phase sampling/data transfer
62    always @(posedge HCLK or negedge HRESETn) begin
63      if(!HRESETn)
64        begin
65          rLED  <= 8'hFF;       // RESET values - all LEDs and HEXes on
66
67
68
69
70
71
72        end
73      else if (rHSEL & rHWRITE & rHTRANS[1]) // ... AND gate condition
74        case (rHADDR[2:0])
75          0:                        // addresses shown in decimal for human convenience
76          1:
77          2:
78          3:
79          4:
80          5:
81          6:
82        endcase
83    end
84
85
86    // ASSIGN section: drive HREADYOUT, HRDATA, LED wires (three assign Verilog statements)
87    //HREADYOUT Single cycle write & Read. Zero wait state operations
88    assign HREADYOUT = 1'b1; //
89  // setting AHB-Lite Read Data
90    assign HRDATA = {22'h0,SW}; // return the full word with the 22 msb set to zero and 10 lsb set to SW
91
92  // drive the on board LEDs [7:0]
93    assign LED[7:0]= rLED;
94
95
96
97
98
99
100
101 endmodule
102
103
```

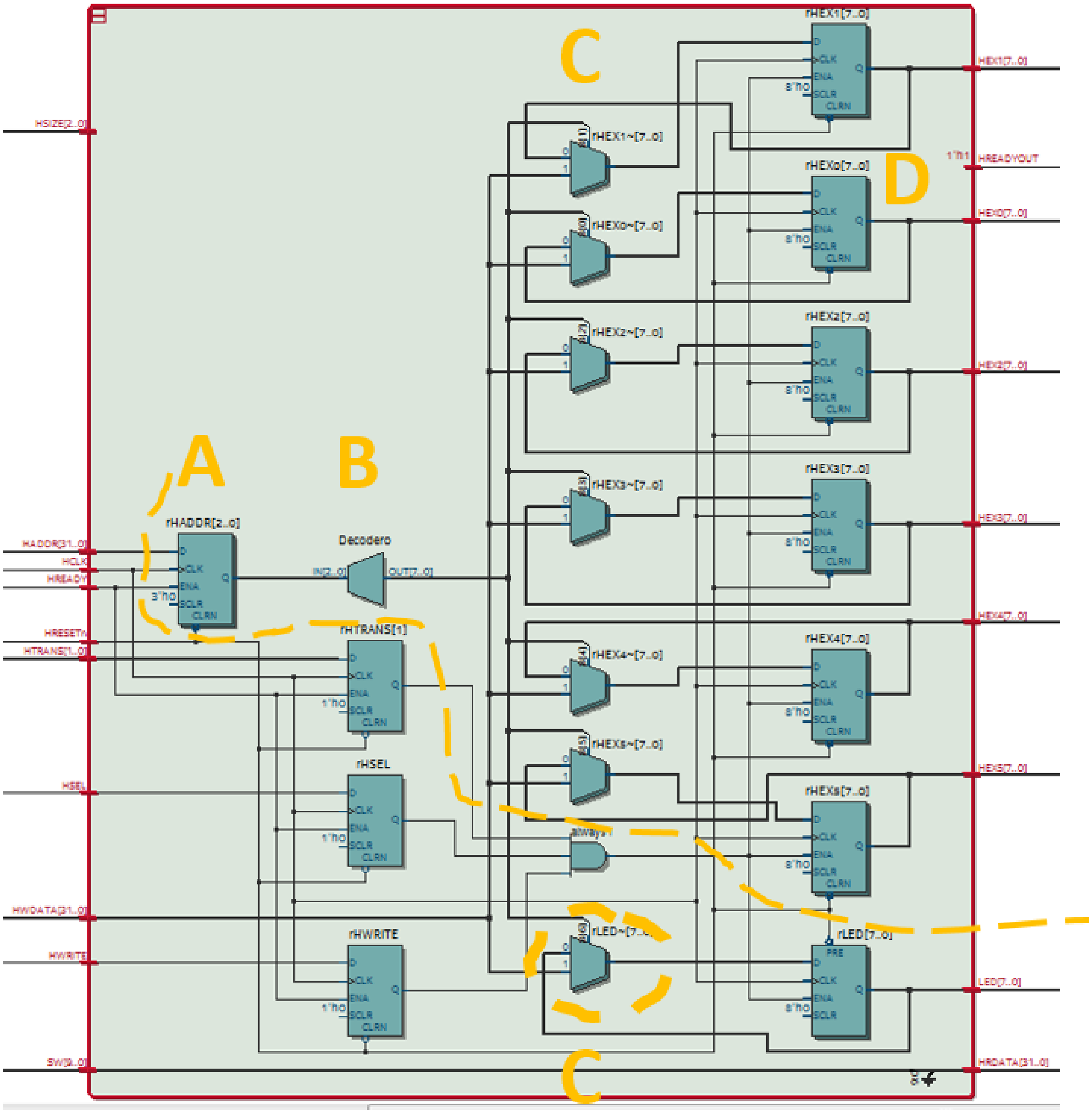
When KEY[0] is pressed

HWDATA[7:0] go into different registers depending on the last three bits of the address (this is a demultiplexer thus no need for default)

Here I needed to exercise the address allocation for the SoC – decide which addresses will be used for which registers. I decided to start from the addresses of the HEXes (HEX0 to be 0x5000_0000, HEX1 to be 0x5000_0001 etc). The address for the rLED became 0x5000_0006. As we need to distinguish among 7 registers, I only use 3 LSB of the rHADDR in the code (rHADDR[2:0]), leaving address space for one more register. If the number of the registers exceeds 8, one will need to start using more bits from the address to distinguish among these registers.

(Line 90 works irrespectively of the of the HADDR bits. Therefore reading from any address in the 16 MB space 0x5000_0000 .. 0x50FF_FFFF will get the SW status to the CPU.)

The schematic diagram for the developed HDL code was produced by the Quartuses' RTL viewer:



I outlined the elements added compared to the original AHB2LED module.

A – the register to sample the address bus at the address phase

B – decoder to select one out of the seven register depending on the bits [2:0] of the sampled address

C – multiplexers that either supply the data from the AHB-lite write data bus (HWDATA) or connect the D flip flop register to its own output thus the stored data is written back at each clock cycle. (One extra multiplexer needs to be used for the rLED. That is because in the original version it was the only register to write to therefore there was no need for the decoder and multiplexer.)

D – the registers to keep external hardware controlled when the CPU processes other data

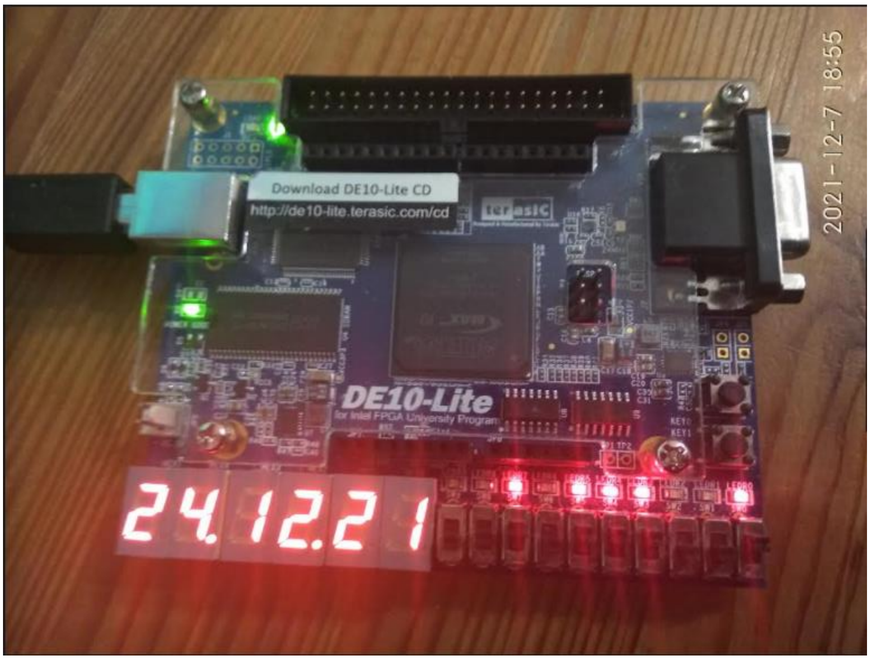
The application programmer will require this information in order to operate the HEXes appropriately. For your information I provide the code that was written for this addressing convention:

main.c

```c
#include <stdint.h>
#define AHB_LED_BASE        (uint8_t*) 0x50000000  // address of the LED peripheral in the SoC

int main(void) {

  uint8_t *ptr = AHB_LED_BASE; // pointer to the LED peripheral location
  uint8_t tmp;                 // temporary

  while (1) {
    tmp = *ptr;                // read the value of all the switches; and address within 16 MB is valid

    // subsequent byte addresses                               constants from the lab 3_3, C does not accepts binary
    *(ptr+0) = 0xF9;           // for displying '1' on HEX0   8'b11111001
    *(ptr+1) = 0xA4;           // for displying '2' on HEX1   8'b10100100
    *(ptr+2) = 0x24;           // for displying '2' on HEX2   8'b10100100 + leading zero for .
    *(ptr+3) = 0xF9;           // for displying '1' on HEX3   8'b11111001
    *(ptr+4) = 0x19;           // for displying '4' on HEX4   8'b10011001 + leading zero for .
    *(ptr+5) = 0xA4;           // for displying '2' on HEX5   8'b10100100

    *(ptr+6) = tmp;            // write back SWs to the LEDRs, could do simply   *(ptr+6) = *ptr;
  }

}
```

(The complete project with the compiled **code.hex** and converted **code.mif** files are provided on BB.)

If you keep addressing of the registers the same as I did, this code will work on your SoC like shown below

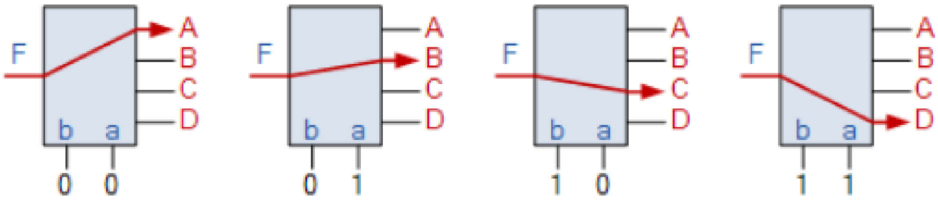


APPENDIX. What is demultiplexer

A device that connects one input to one out of many outputs

Example https://www.electronics-tutorials.ws/combination/comb_3.html

Demultiplexer Output Line Selection



Please note that such a diagram is incomplete because the non-selected outputs are not driven at all, which is a design mistake. In practice all the outputs should be weakly pulled up or down, or, better and implementable in an FPGA, has registers that will keep an old value when this particular output is not connected to the input.

A Verilog module for an 8 bits synchronous demux 1:4 like above might be described as:

```
module demux_1_4 ( input [7:0] F,
                   input [1:0] sel, // to select the required output
                   input clk,
                   output reg [7:0] A, B, C, D
);
  always @ ( posedge (clk) ) begin // begin is optional here
    case (sel)    // if a reg is not selected, it keeps its old value
      2’b00: A <= F;
      2’b01: B <= F;
      2’b10: C <= F;
      2’b11: D <= F;
    endcase
  end
endmodule
```