

1. Introduction

Using VHDL for the development of combinatorial and sequential logic is beneficial for smaller applications. However, for more demanding applications we often need some “intelligence” to manage the system such as microprocessors combined with some custom logic. This lab will demonstrate the implementation of a MicroBlaze microprocessor within the Nexys 4 DDR.

2. IP integrator; MicroBlaze

IP integrators can be used to implement a microprocessor onto the FPGA. This process can be achieved by instantiating the existing MicroBlaze IP from the IP catalogue. After instantiation, the MicroBlaze’s pinouts can be observed similarly to any microprocessor, the VHDL component code can be shown in Figure 2.1.

```
14 entity Lab6_1_wrapper is
15   port (
16     dip_switches_16bits_tri_i : in STD_LOGIC_VECTOR ( 15 downto 0 );
17     led_16bits_tri_o : out STD_LOGIC_VECTOR ( 15 downto 0 );
18     reset : in STD_LOGIC;
19     sys_clock : in STD_LOGIC;
20     usb_uart_rxd : in STD_LOGIC;
21     usb_uart_txd : out STD_LOGIC
22   );
23 end Lab6_1_wrapper;
24
25 architecture STRUCTURE of Lab6_1_wrapper is
26   component Lab6_1 is
27     port (
28       sys_clock : in STD_LOGIC;
29       led_16bits_tri_o : out STD_LOGIC_VECTOR ( 15 downto 0 );
30       dip_switches_16bits_tri_i : in STD_LOGIC_VECTOR ( 15 downto 0 );
31       usb_uart_rxd : in STD_LOGIC;
32       usb_uart_txd : out STD_LOGIC;
33       reset : in STD_LOGIC
34     );
35   end component Lab6_1;
36   begin
37     Lab6_1_i : component Lab6_1
38     port map (
39       dip_switches_16bits_tri_i(15 downto 0) => dip_switches_16bits_tri_i(15 downto 0),
40       led_16bits_tri_o(15 downto 0) => led_16bits_tri_o(15 downto 0),
41       reset => reset,
42       sys_clock => sys_clock,
43       usb_uart_rxd => usb_uart_rxd,
44       usb_uart_txd => usb_uart_txd
45     );
46 end STRUCTURE;
```

Figure 2.1: VHDL instantiation of the MicroBlaze IP.

Although the FPGA now has a softcore microprocessor, developers will still need custom logic to bind it with FPGA I/Os. The system’s clock will require an active low reset time so that it may trigger on the falling edge of the clock. MicroBlaze’s AXI interconnect provides its own general purpose input-output (GPIO) to enable master-slave capabilities for these custom logics. This will allow 16 LEDs and 16 switches to be implemented as GPIOs. The custom logic and the MicroBlaze IP can be fully routed by the “Run Block Automation” which automatically makes connections amongst these modules. As a final step, local memory for the MicroBlaze can be initialised to allocate resources. The routed digital circuit can be seen in the IP integrator interface as shown in Figure 2.2.

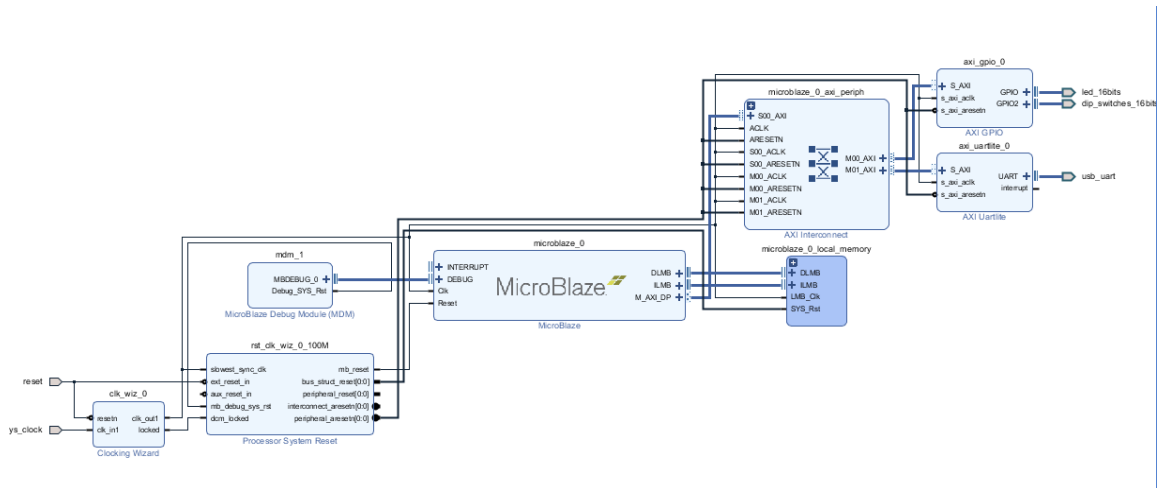


Figure 2.2: IP integrator of the MicroBlaze and custom logic.

A newly generated microprocessor with custom logic, such as GPIOs and localised memory, will require validation through "Validate Design" button. Bitstream can then be generated to analyse the utilisation report. Several extensions can be added to the MicroBlaze softcore processor to meet the needs of developers, such as a cache, floating point unit, and memory interface for DDR. It is important to note that adding these extensions will impact the amount of FPGA fabric used to implement this digital circuit. Currently, the configuration does not have any additional extensions which its FPGA fabric usage can be seen in Figure 2.3.

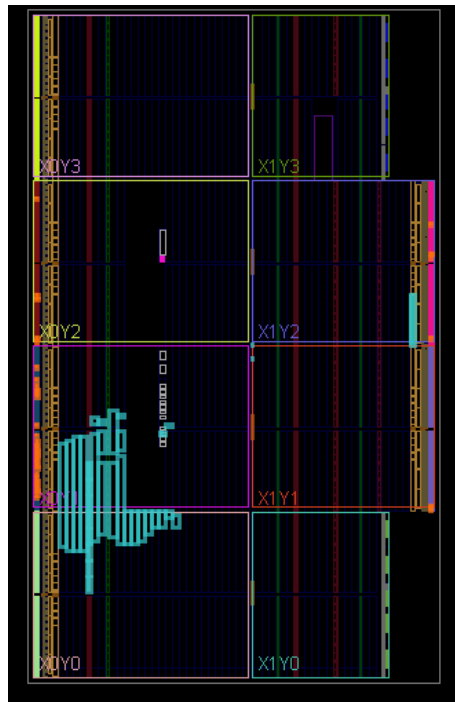
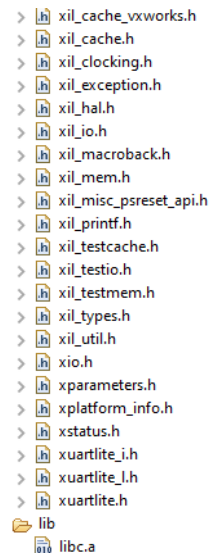


Figure 2.3: Device schematic of the softcore MicroBlaze with custom logic.

3. HDL wrapper

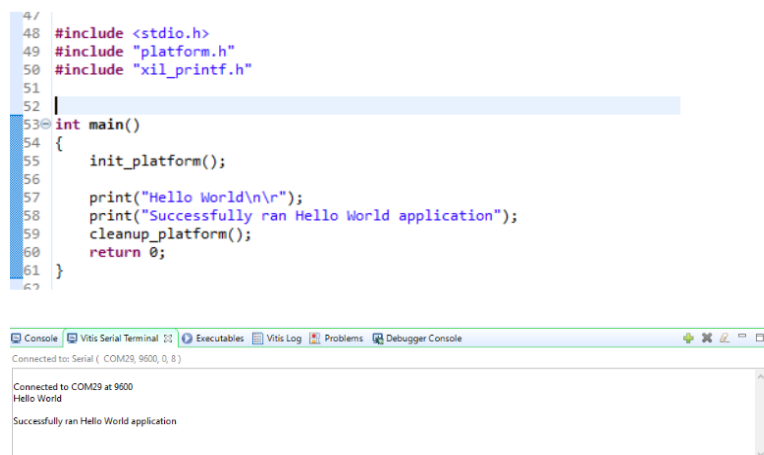
The HDL wrapper of the previously implemented firmware can be created in the “source” tab which will allow developers to program the FPGA implementation of the MicroBlaze microprocessor using C. This can be achieved by exporting the HDL wrapper as a hardware design to the Vitis software development platform. To program this newly implemented MicroBlaze, a UART baud connection will need to be established in the Serial Terminal.



```
> xil_cache_vxworks.h
> xil_cache.h
> xil_clocking.h
> xil_exception.h
> xil_hal.h
> xil_io.h
> xil_macroback.h
> xil_mem.h
> xil_misc_psreset_api.h
> xil_printf.h
> xil_testcache.h
> xil_testio.h
> xil_testmem.h
> xil_types.h
> xil_util.h
> xio.h
> xparameters.h
> xplatform_info.h
> xstatus.h
> xuartlite_i.h
> xuartlite_l.h
> xuartlite.h
lib
libc.a
```

Figure 3.1: Vitis header files for MicroBlaze.

Similarly to any high-level programming IDE, Vitis comes with its own header files as shown in Figure 3.1. As shown in Figure 3.2, “Hello World” was successfully programmed onto the FPGA in C and the output can be observed in the Serial Terminal. Furthermore, the C code will need to be built and launched on the system debugger to be executed. This piece of code can be repeated by pressing the reset button each time, on the Nexys 4 DDR board or by pressing the resume button.



```
47
48 #include <stdio.h>
49 #include "platform.h"
50 #include "xil_printf.h"
51
52
53 int main()
54 {
55     init_platform();
56
57     print("Hello World\n\r");
58     print("Successfully ran Hello World application");
59     cleanup_platform();
60     return 0;
61 }
```

Console | Vitis Serial Terminal | Executables | Vitis Log | Problems | Debugger Console

Connected to: Serial (COM29, 9600, 0, 8)

Connected to COM29 at 9600
Hello World
Successfully ran Hello World application

Figure 3.2: Hello world in C successful execution on the FPGA MicroBlaze.

```

35 int main ()
36 {
37     Xil_ICacheEnable();
38     Xil_DCacheEnable();
39     print("---Entering main---\n\r");
40
41     {
42         u32 status;
43
44         print("\r\nRunning GpioOutputExample() for axi_gpio_0...\r\n");
45         status = GpioOutputExample(XPAR_AXI_GPIO_0_DEVICE_ID,16);
46
47         if (status == 0) {
48             print("GpioOutputExample PASSED.\r\n");
49         }
50         else {
51             print("GpioOutputExample FAILED.\r\n");
52         }
53     }
54 }
55
56

```

Figure 3.3: consecutive flashing LED peripheral test on the Nexys 4 DDR.

Flashing LEDs can be generated by using the template code as shown in Figure 3.3 which allows developers to test the peripherals of the system. Peripheral testing can be further expanded by allowing the switches to toggle the LEDs.

```

35 int main () {
36 {
37     Xil_ICacheEnable();
38     Xil_DCacheEnable();
39     print("---Entering main---\n\r");
40     XGpio Gpio;
41
42     XGpio_Initialize(&Gpio, XPAR_AXI_GPIO_0_DEVICE_ID);
43     XGpio_SetDataDirection(&Gpio, 2, 0xFFFF);
44     XGpio_SetDataDirection(&Gpio, 1, 0x0000);
45
46     while(1)
47     {
48         u32 status = XGpio_DiscreteRead(&Gpio, 2);
49         XGpio_DiscreteWrite(&Gpio, 1, status);
50     }
51
52     /*
53     * Peripheral SelfTest will not be run for axi_uartlite_0
54     * because it has been selected as the STDOUT device
55     */
56
57     print("---Exiting main---\n\r");
58     Xil_ICacheDisable();
59     Xil_DCacheDisable();
60     return 0;
61 }
62
63
64

```

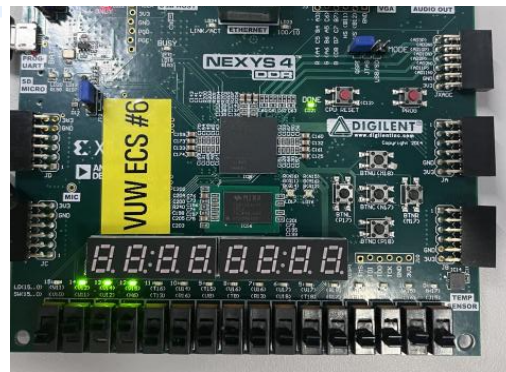


Figure 3.4: Switches to LEDs peripheral testing code and Nexys 4 DDR output.

As the previously implemented custom logic in section 2 binds the LEDs and switches as GPIOs, switching LEDs can be easily implemented on C. This will require going through the collection of functions relevant to the GPIO ports to locate the "set data direction" function to interact with the LEDs and switches. An instance of unsigned 32-bit unsigned integer "status" will be declared to update each LED respective to its switches and the results can be seen in Figure 3.4.

4. conclusion

In conclusion, implementing microprocessors such as the MicroBlaze on an FPGA has advantages including additional on-chip custom logic, and customisable extensions such as cache, floating point unit, and memory interface. This could be leveraged in projects that are incredibly time critical as an entire FPGA system can have multiple microprocessors communicating with each other, while delegating different loads to each processor without worrying about communication latencies between each microprocessor.