Aeronautics Institute of Technology – ITA
Computer Science Division – IEC

EEA-25 Programmable Digital Systems
Prof. André F. Ponchet
Teaching assistant: Felipe Ferreira

Laboratory 08
Last document update: November 6, 2023

1 Objectives

1.1 General Objectives

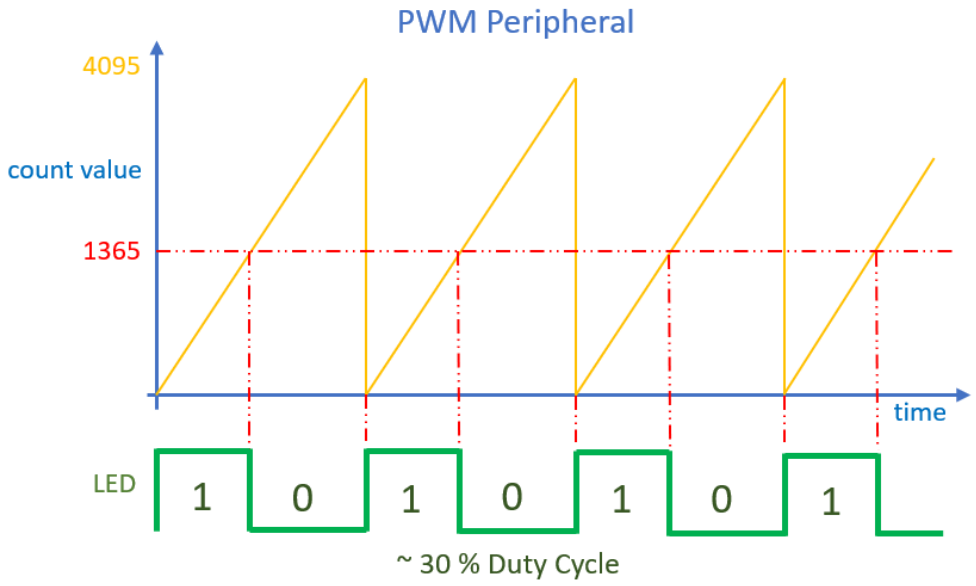
- Familiarize yourself with the Lattice iCE40 development environment;
- Understand circuit design process using HDLs and FPGA implementation;
- Simulate and verify results with real FPGA boards.

1.2 Specific Objectives

- Create a PWM peripheral for the RISC-V microprocessor.

1.3 PWM Peripheral

- We Will develop a simple pulse-width modulation (PWM) peripheral. The SOC can write to the peripheral's register. The PWM Peripheral will have a counter that counts continuously, in our example it will count up to 4095 and reset to 0, over and over. Whenever the counter is less than the value set in the register, the pin will be a high value. Whenever it is greater than or equal to the register value, the pin will be low. This will create a pattern similar to the PWM Peripheral figure below:



PWM Peripheral.

2 Lab Development

2.1 Getting processor repository

- Create Create a directory "lab08" and execute the commands below in the linux terminal.

```
sudo apt-get install picocom
git clone https://github.com/dloubach/femtorv32.git femtorv32
cd femtorv32/FemtoRV/
sudo make icesugar_nano
```

- Wait for the necessary tools to download.
- Change to the path indicated below and create the PWM driver:

```
 cd RTL/DEVICES/
```

- Write the pwm module provided below in the **path indicated above**. Use your text editor or the nano command.

```

module pwm #(

    // Parameters
    parameter       WIDTH = 12  // Default PWM values 0..4095

) (

    // Inputs
    input           clk,
    input           wstrb,  // Write strobe
    input           sel,    // Select (read/write ignored if low)
    input   [31:0]  wdata,  // Data to be written (to driver)

    // Outputs
    output     led
);

    // Internal storage elements
    reg             pwm_led = 1'b0;
    reg [WIDTH-1:0] pwm_count = 0;
    reg [WIDTH-1:0] count = 0;

    assign led = pwm_led;

    // Update PWM duty cycle
    always @ (posedge clk) begin

        // If sel is high, record duty cycle count on strobe
        if (sel && wstrb) begin
            pwm_count <= wdata[WIDTH-1:0];
            count <= 0;

        // Otherwise, continuously count and flash LED as necessary
        end else begin
            count <= count + 1;
            if (count < pwm_count) begin
                pwm_led <= 1'b1;
            end else begin
                pwm_led <= 1'b0;
            end
        end
    end

endmodule
```

- Save the file with the name "pwm.v". Based on the code description above and your knowledge, propose a testbench that validates its operation.

2.2 Memory Addressing

The processor has a unique memory addressing scheme to communicate with peripherals. Memory addresses are 32 bits. However, bits 24..31 and bits 0..1 are not used. Bits 22 and 23 are used to access different "pages".

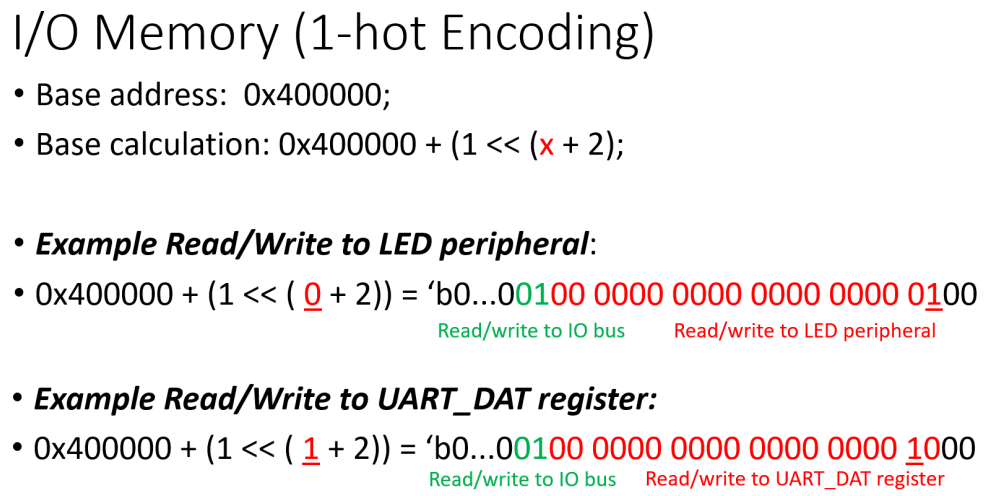
x	x	x	x	p	p	a	a	a	a	a	a	a	x	x
31	30	...	24	23	22	21	20	...	5	4	3	2	1	0

x: unused bit
P: page
00: RAM
01: I/O
10: SPI flash
a: address

Addresses:
0x00000000 ... 0x003FFFFC: RAM (4 MB)
-Each element is 32 bits wide (4 bytes)
0x00400000..0x00480000: I/O page
-1-hot encoding scheme for peripherals
0x00800000...0x00BFFFFC: SPI flash
- Program memory on iCESugar

Memory map.

- Basically the memory pages are divided into the three types of addresses shown in the "Memory map" figure above, being the pages where we want to read or write data. For example, if bits 22..23 are 'b00, then we can access physical RAM (implemented as RAM in physical blocks already contained in iCESugar-nano). This means addresses 0x00000000..0x003FFFFC (used by CPU instructions) used to read/write data in RAM.
- However, the iCE40LP1K FPGA chip has a total of 64K bit RAM available and 2kB is reserved for general purpose registers. Therefore, we do not use the total number of mapped possibilities.
- So, if we want to access our hardware peripheral registers, bits 22..23 are 'b01. If we have these two bits in 'b10, then we want to access the program memory which is communicated via the SPI protocol between the external flash and the development board.
- For accessing the I/O memory address space registers, the peripheral addresses are provided by a one-hot coding scheme. One-hot encoding means that we have only one high bit at a time for each specified address. There are 20 bits of address space available (bits 2..21), so there are 20 addresses in total to be used as peripheral hardware in our SOC.



I/O Memory.

- The file in **RTL/DEVICES/HardwareConfig_bits.v**, maps our existing peripherals. Bits 0..11 are used by already defined peripherals, and bits 17..19 are reserved for registers used by the CPU. Therefore, we will need to modify this file as well as others to integrate our PWM driver.

2.3 Integrating the peripheral

- We will make small changes to the project's original verilog code so that we can integrate our PWM driver. First we will add the available bit number in the file HardwareConfig_bits.v:

```
nano HardwareConfig_bits.v
```

- Note that we need to add the indicated bit of our drive in the hardware configuration file. Add the line below that represents the shift bit to access our drive.

```
localparam IO_PWM_bit   = 12; // W  write duty cycle (12 bits)
```

- Save and exit.
- Now we will modify our top-level file "femtosoc.v":

```
cd ..
nano femtosoc.v
```

- Make the following changes to the Verilog code:
- Note that the first change takes place on line 6 of the example below, we are adding our built-in PWM driver to the SOC.
- The second change is in the femtosoc module, indicated in line 18 of the code below.
- The third change is indicated in line 26 of the code below, where we instantiate our PWM drive to the SOC. Locate the snippets properly in the femtosoc.v file. If you prefer, use the text editor.

```
1
2 ...
3
4 `include "DEVICES/FGA.v"             // Femto Graphic Adapter
5 `include "DEVICES/HardwareConfig.v" // Constant registers to query ...
6 `include "DEVICES/pwm.v"             // PWM driver for one LED
7
8 ...
9
10 module femtosoc(
11 `ifdef NRV_IO_LEDS
12  `ifdef FOMU
13    output rgb0,rgb1,rgb2,
14  `else
15    output D1,D2,D3,D4,D5,
16  `endif
17 `endif
18 `ifdef NRV_IO_PWM
19   output D1,
20 `endif
21
22 ...
23
24 /*****************************************************************/
25 /* PWM Peripheral                                                */
26 `ifdef NRV_IO_PWM
27   pwm #(
28     .WIDTH(12)
29   ) pwm (
30     .clk(clk),
31     .wstrb(io_wstrb),
32     .sel(io_word_address[IO_PWM_bit]),
33     .wdata(io_wdata),
34     .led(D1)
35   );
```

```
`endif

/*****************************************************************/
/* And last but not least, the processor                         */

...
```

- Save and exit.
- Change to the directory below and make the following changes to the pin constraints file:

```
cd ../BOARDS
nano icesugar_nano.pcf
```

```
# set_io board_led B6
set_io D1 B6
```

- Save and exit.
- In the iCESugar-nano config file, we need to disable the blink module for debugging as it uses the pin we defined to output our PWM signal. We also enable the use of the PWM module;

```
cd ../RTL/CONFIGS/
nano icesugar_nano_config.v
```

- Comment out the line "RV_DEBUGt_ICESUGAR_NANO" and enable PWM with the line `define NRV_IO_PWM

```
//`define NRV_IO_BUTTONS // Mapped IO to PMOD connector (78, 79, 80, 81)
//`define NRV_IO_LEDS  // Mapped IO, LEDs D1,D2,D3,D4 (D5 is used to di$
`define NRV_IO_PWM
...
//`define RV_DEBUG_ICESUGAR_NANO
```

- Save and exit. Build and upload the new SOC design:

```
cd ../..
sudo make icesugar_nano
```

2.4 Example Software

- A lib was developed that includes memory addressing macros to be used in the development of software for the processor. However, we will refer directly in code to the specified address of our PWM drive.

```
cd ../
mkdir -p pwm_test ; cd pwm_test
nano main.c
```

- Below is a simple example of an algorithm written in C to increase the brightness of the LED connected to the PWM and turn it off.

```
#include <femtorv32.h>

int main() {
    while (1) {
        for (int i = 0; i < 4096; i++) {
            *(volatile uint32_t*)(0x404000) = i;
            delay(1);
        }
    }
}
```

- Save and exit. Create a Makefile that includes the FemtoRV Makefile template.

```
nano Makefile
```

- Add the following line:

```
include ../FemtoRV/FIRMWARE/makefile.inc
```

- Save and exit. Build and upload the software:

```
sudo make main.sprog
```

- At this point you have just compiled a C code using the RISC-V toolchain downloaded from the repository. The code is compiler generating an assembly code and later an executable that is written to the specified address of the external flash 0x00020000. You will notice the PWM effect on the yellow LED connected to the board. It will take a little over 4 seconds to complete a cycle.