POLITECNICO
MILANO 1863

Massimo Maitan, Robert Medvedec, Federico Megler, Angelo Paone

Advanced Operating Systems and Embedded Systems Project

Deliverable:	Project Document
Title:	Arty A7 ARM Design Start - Morse code application
Authors:	Massimo Maitan, Robert Medvedec, Federico Megler, Angelo Paone
Version:	1.0
Date:	03-March-2021
Download page:	https://github.com/federicomegler/embedded-project
Copyright:	Copyright © 2021, Maitan, Medvedec, Megler, Paone – All rights reserved

1 Introduction

This project was created by four students, Maitan, Medvedec, Megler, and Paone, as a part of Embedded Systems and Advanced Operating Systems courses.

Software used to create this project:
- Xilinx Vivado - simulation, testbench, board implementation
- Keil uVision - board execution code workspace, object file builder

Base of the project is ARM Design Start, using Arty A7 FPGA on M3 board. The project as a whole consists of several parts - application, AXI, and testbench. Our team worked on an application part by using default C libraries already implemented into ARM Design Start platform.

2 Project info

The main goal of our application was to use as many functionalities of the board as possible to demonstrate and showcase the way they work and the way they are connected.

Arty A7 is filled with many different features such as LEDs, JTAG, GPIOs, connectors, good amount of RAM memory, switches, buttons, and other additional circuitry, which allow for the board to be used in a lot of different ways and to combine all these features for a more complex application.

The idea was to create a Morse code application which would use LEDs, buttons, and UART for receiving and displaying a message. In particular the idea is to provide a Morse word using the push buttons located on the board, and display the ciphered word using UART and LEDs.

The general idea behind the Morse code is following - Morse code characters are sent as a combination of short signals, long signals, and pauses. Short signals are a single time unit, while long signals are three time units. The pauses between signals of the same letter are one time unit, the pauses between letters are three time units, and the pauses between words are seven time units long.

This allowed the use of timers to measure the length of each signal sent as a button press to the board, making the character recognition work on the length of a button press rather than using different buttons for different signal lengths. However, in order to simplify things and make them clearer, we have added two more buttons for letter and word spaces. This way we allowed for signals to be more easily readable in the simulation waveform as well as increased reliability and security of the application.

3 Project details

3.1 Code structure

In the default M3 for Arty A7 project there are a number of files written in C that are describing the behaviour of the board. We have only changed three files, through Keil uVision software, in order to modify things that we needed for the project. Those three files are *main.c*, *gpio.c*, and gpio header file, *gpio.h*. In the following sections we explain the newly created functions and the things we have both added and removed in order to make our application run faster and smoother.

Main.c - main code file of the program and the first thing that is executed after the startup script.
Gpio.c - part of the code that handles input/output functions, interrupts, and controls the peripherals of the device.
Gpio.h - header file of gpio.c.
All files can be found in the provided GitHub repository and are available for detailed code inspection.

MAIN.c – Commented out all the unnecessary functions for this project (DAP link, RAM checks, SPI, QSPI, buffer checks,...) - speeds up the simulation

GPIO.c
- Added some new global variables (*word*, *encoded_sentence*, *character*, *iterator*, *lastButton*, *isLong*, *begin*, *translated_sentence*)
- *SysTick_Handler()*, *longDelay()*, *shortDelay()*, *mediumDelay()* - added time functions for working with Morse code signals
- *timer()* – sets timer to 0 and every 800 clock cycles sends an interrupt
- *blink()* – blink for a short period of time
- GPIO1_Handler – if the first button is pressed – the timer is activated and then the signal is identified as long or short, if the third button is pressed – the set of signals is decoded into an ASCII character and added into the word string, if the fourth button is pressed – the word is ciphered and shown on UART and finally the Morse code of the ciphered is shown using LEDs
- *addSignal()* – adds signal to the character array (short or long)
- *nextCharacter()* – finishes a current character, decodes it into ASCII, and starts a new one
- *printWord()* – prints a new word after translating it (Caesar 3), encodes it and prints it
- *printEncodedMorse()* - prints by using LEDs
- *encodingMorse()* - encodes Morse code
- *decodingMorse()* – decodes Morse and reads characters from inputs
- *translateWord()* – translates a word using Caesar cipher

GPIO.h
- Added function headers (*buttonCheck, printEncodedMorse, encodingMorse, shortDelay, mediumDelay, longDelay, timer, nextCharacter, printWord, addSignal, decodingMorse*)

3.2 GPIO

GPIOs allow components of the board to communicate with another peripheral, which in this case means the M3 processor with its peripheral register. In our project to build the Morse code system we used *GPIO_0* and *GPIO_1*.

- **GPIO_0** - the LEDs mirror the state of the DIP switches. When each switch is turned on, the appropriate LED is lit. This mechanism helps to understand what is currently happening during the simulation.
- **GPIO_1** - connected to the four push button switches - allows handling of the input signal.

The two most common methods for retrieving signals from GPIOs are:

- Polling - check iteratively by scanning the peripheral I/O;
- Interrupt - when a specific signal rises from a GPIO, a procedure called Interrupt Service Routine (ISR) is invoked. This causes the interruption of the current instruction flow and the beginning of the execution of the ISR.

We chose the interrupt method because polling is expensive in terms of instructions. This allows the CPU to execute other instructions instead of checking continuously the state of peripheral registers or even put itself in a low-power state. CORTEX-M3 uses what is called a vectored approach for interrupts. In this approach, there is a vector table in memory that lists for each interrupt the address where the ISR is so that the CPU must execute exactly where that particular interrupt is located. This address is typically called “interrupt vector”. Our GPIO peripheral interrupts are enabled through *EnableGPIOInterrupts()* that is called in the main function. This allows us to take all the events that come from GPIOs and DIP switches - the pushes and the releases of the buttons. When these kinds of interrupts occur, a handler is executed: for *GPIO_0* the DIP switch state is read, and the LED state is matched with it.*GPIO_1* then activates and the handler manages three different cases:

- The first of the four push buttons has been pressed: this button is used to generate the short or the long signal (depending on the time of the press). The mask of bits that allows to identify such an event is 0x01 (binary 0001).

- The third push button has been pressed: the press of this button means that the sequence of signals identifying the same character in Morse code has been terminated. This event is associated to the mask 0x04 (in binary, 0100). In this case, the handler calls the *nextCharacter()* function, that by means of function *decodingMorse()*, first decodes the sequence of long/short signals stored in the “character” global variable, which is used for storing received Morse signals, appending the ASCII result to the global string “word”, and finally clears the “character” variable.

- The fourth button has been pressed: we have reached the end of an input word. To be able to identify the triggering of this event we use the mask 0x08 (binary 1000). When such an event occurs, the handler calls the *printWord()* function that:

  - First decodes the last received character by means of the function *decodingMorse()*, appending the result to the global “word” string, and clearing the "character" variable.
  - Translates the word using the translate_word function, that encrypts it according to a Caesar’s cipher (with key = 3).
  - Prints the encrypted word:
    * Encoded in ASCII through the UART.
    * Encoded in Morse turning a LED on and off for a long/short period; this is performed by calling *encodingMorse()* first and then *printEncodedMorse()*.

The following is the code for GPIO_1 handler function which shows the basic code structure and the way the LEDs are switched ON and OFF.

```
void GPIO1_Handler ( void )
{
   if(XGpio_DiscreteRead(&Gpio_RGBLed_PB, ARTY_A7_PB_CHANNEL) == 0x4){
        nextCharacter();
   }
   else if(XGpio_DiscreteRead(&Gpio_RGBLed_PB, ARTY_A7_PB_CHANNEL) == 0x8){
        printWord();
   }
   else{
      if(begin == 1){
         begin = 0;
         if(isLong == 1){
               XGpio_DiscreteWrite(&Gpio_Led_DIPSw, ARTY_A7_LED_CHANNEL, 0x02);
               longDelay();
               XGpio_DiscreteWrite(&Gpio_Led_DIPSw, ARTY_A7_LED_CHANNEL, 0x00);
               addSignal(isLong);
         }
         else{
               XGpio_DiscreteWrite(&Gpio_Led_DIPSw, ARTY_A7_LED_CHANNEL, 0x01);
               shortDelay();
               XGpio_DiscreteWrite(&Gpio_Led_DIPSw, ARTY_A7_LED_CHANNEL, 0x00);
               addSignal(isLong);
         }
      }
      //Start counting the length of the signal only if button 1 is high
      if(begin == 0 && XGpio_DiscreteRead(&Gpio_RGBLed_PB, ARTY_A7_PB_CHANNEL) == 0x1){
         isLong = 0;
         begin = 1;
         timer();
      }
   }
    // Clear interrupt from GPIO
    XGpio_InterruptClear(&Gpio_RGBLed_PB, XGPIO_IR_MASK);
    // Clear interrupt in NVIC
```



```
    NVIC_ClearPendingIRQ(GPIO1_IRQn);
}
```

Moreover, we need to be able to time events in our application. To do that, we used the Cortex M3 system tick timer. For applications that do not require an OS, the *SysTick* can be used for time keeping, time measurement, or as an interrupt source for tasks that need to be executed regularly. To determine a short or long signal, this will generate an interrupt at a pre-determined interval, i.e., when a push button is kept pressed after a prefixed time.

3.3 Encryption and decryption

In order to perform the encryption, we have written a function called *translate_word* that takes in the input word (as a pointer to char) and an integer representing the key. The encoded word will have, in each position, the 3rd letter after the current letter according to the alphabet, if the key is 3 like in this case ,since it uses the aforementioned Caesar's cipher.

We have implemented a "new" alphabet, that takes consists of all of the Morse code characters, which is the set of all letters and numbers, that is A, B, C, D, . . . , Z, 0, 1, 2, 3, . . . , 9. This means that, for example, using a cipher with key = 3, we will have these transformations A -> D , B->E, . . . , W -> Z, X -> 0, Y-> 1, Z -> 2, 0 -> 3, . . . , 8 -> B, 9 -> C .

As an example, the word CIAO will be translated in FLDR. In the code, this encryption is implemented by defining first the array containing the 36 characters of the alphabet, and then iterating each character of the string we want to convert in a for loop. For each character, the variable *current_char* is set to the character corresponding to the "shifted" letter/number (using the modulus to ensure the "circularity" of the alphabet). If the character of the word considered is a space, *current_char* is set to " " too (NB: this is done just for "code reusage" purposes, since it won't happen in an actual use case of our application, because it's designed to convert only a single word).

After the computation of the current_char, it is appended to a global variable called *translated_sentence*. This means that when the function has finished the execution of the loop, *translated_sentence* contains the sentence (in this case, the word) encrypted with the cipher.

3.4 Solving project issues

One of the first issues we have encountered with the project was printing out the word in some way. We found it to be impossible to write out the word using the console, as the simulation is too slow, and in order to get all the necessary inputs, read the word, decode it, and send it back to console, it would take way too long for actual implementation and testing to complete in reasonable amount of time. This problem was solved problem by using LEDs and UART for displaying the read message. In this case, as already mentioned in the paper, LEDs print out the Morse code word with Caesar encryption (left shift by 3), while UART prints out the originally received Morse code word.

The second problem we have encountered was timing related. It was difficult to figure out when exactly the Vivado testbench is being ran and when does the C code run. The conclusion is that every time there is a wait in the testbench code longer than 12us, the C code is allowed to run until the wait is over. After all the initial functions of the main.c (main program code written in Keil) have been executed, it drops in the empty while loop and relies on the interrupts and button pushes to be pressed to activate the functions found in gpio.c (program code handling input and output operations). While this is not the most elegant solution, we have not managed to find the way to immediately jump to C code without waiting for 12us. Nevertheless, that specific interval is not too long and it is allowing us to test and run the application in real time with no major delays.

We used interrupts and timers to solve the problem in an efficient way. As the signal of a button changes state (0 $\Rightarrow$ 1 or 1 $\Rightarrow$ 0), the signal *axi_gpio_1_ip2intc_irpt* pops up and the routine is executed as soon as possible.

4 Testbench

In this section we discuss about the signals generated during the simulation, their meaning, and how they relate to other signals.

The relationship between Vivado testbench and the main.c code in Keil is the following – the Vivado testbench is being executed until it reaches an “empty loop”. An empty loop is represented by a waiting signal, during which the simulation does nothing, but rather waits for a certain amount of time. We can simulate an empty loop by using this line of code - ***repeat (2000) @(posedge clk_sys) begin end;***. In this example, Vivado testbench simulation will stay in place for 20us (100 cycles in repeat is 1us). The the main.c code is executed only after 12us of the testbench wait. So in order for the testbench to allow the execution of any C code, we must have a delay of at least 12us + the time it takes to execute the C code in between button presses.

The initial program starts after 500us of simulation setup time which means that the button presses and signals are not generated until that point in the simulation.

The function for measuring time in Keil C code is *SysTick*, which sets the timer time, and uses the same clock cycles as Vivado, making *SysTick_Config(800)* setting the interrupt to 8us, meaning that if the button is pressed for at least 8us, the interrupt will happen.

Here is the modified testbench code that simulates sending a Morse code signal via a button press and allows the C code to run and decode the received information.

```
"repeat (10) @(posedge clk_sys) begin end;
        push_buttons_SRL <= 8'h0;
        repeat (1000) @(posedge clk_sys) begin end;
        push_buttons_SRL <= 8'h1;
        repeat (2000) @(posedge clk_sys) begin end;
        push_buttons_SRL <= 8'h0;
        repeat (10000) @(posedge clk_sys) begin end;
        push_buttons_SRL <= 8'h1;
        repeat (6000) @(posedge clk_sys) begin end;
        push_buttons_SRL <= 8'h0;
        repeat (10000) @(posedge clk_sys) begin end;"
```

After all button presses have been gathered and the code word recognized, the outputs are being generated.

In this image, a word encoded with the Caesar cipher can be seen as a set of LED signals. Those signals are once again represented using Morse code, but this time the output word is a encrypted input word.

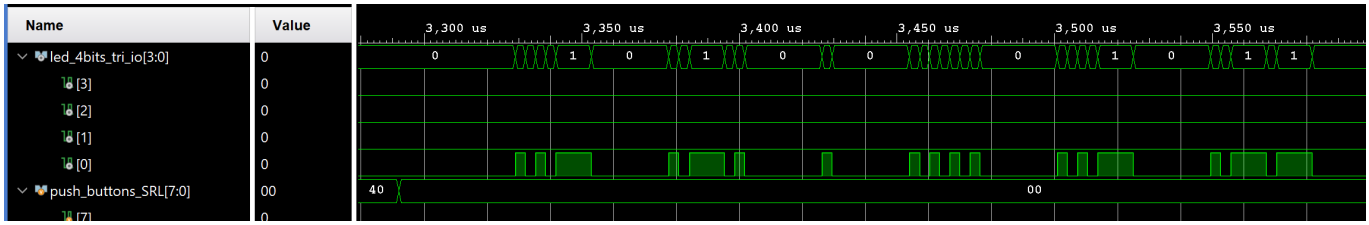

Figure 1: LED printing of the word UREHUW (ROBERT with Caesar shift by 3)

In this image, an input word is directly represented over UART. If we would translate the signals seen in the *uart_tx* line, we would get a representation of the word "ROBERT" written in ASCII values.

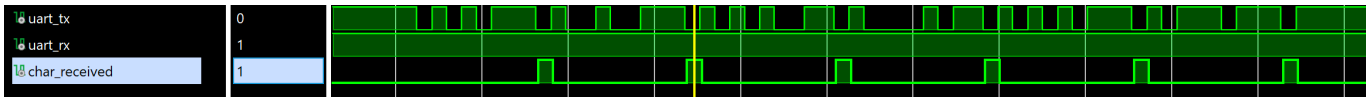

Figure 2: UART printing of the word ROBERT

