University of Minho
School of Engineering

EMBEDDED SYSTEMS
RESEARCH
GROUP

Diogo Silvino PG50341
João Peixoto PG50479

8051 Processor Design
Verilog implementation

Embedded Systems Project
Industrial Electronics and Computers Engineering
Embedded Systems and Computers

Professor:
Adriano Tavares

January 2022

Índice

List of Figures

List of Tables

List of Listings

Acronyms

FPGA	*Field Programmable Gate Array*
CPU	*Central Process Unit*
ROM	*Read Only Memory*
RAM	*Random-Access Memory*
SFR	*Special Function Register*
ISR	*Interrupt Service Routine.*
UART	*Universal Asynchronous Receiver Transmitter*
SPI	*Serial Peripheral Interface*
CDC	*Clock Domain Crossing*

1 Introduction

This document is the 8051 Processor Design report. This project aims to implement a version of the microcontroller from the Intel 8051 family in the Verilog language and deploy it in an FPGA, more specifically in the Zybo Z7 manufactured by Digilent. To this end, all the knowledge acquired during the degree in the Microcontrollers and Microprocessors units was put into practice, as well as what was learned to date in the specialization of Embedded Systems and Computers, taught by Professor Adriano Tavares.

2 Methodology

To fulfill all the requirements previously established by the teacher, it was necessary to have a work methodology. Nowadays the methodology is one of the most important and neglected sections in engineering and can be seen as a discipline and an engineering method to reduce the associated costs and optimize the reliability of a process through analyzing task performance. Also, these methods can establish where people are best utilized in a process to allow them to complete an allocated task in the most effective manner possible.

In this case, a strategy was drawn up that consisted of following the **waterfall model**. The waterfall model is a linear, sequential approach to the software development life cycle that is popular in software engineering and product development. The waterfall model emphasizes a logical progression of steps. Similar to the direction water flows over the edge of a cliff, distinct endpoints or goals are set for each phase of development and cannot be revisited after completion.

The waterfall methodology is composed of six non-overlapping stages:

1. Requirements: Potential requirements, deadlines and guidelines for the project are analyzed and placed into a functional specification. This stage handles the defining and planning of the project without mentioning specific processes.
2. Analysis: The system specifications are analyzed to generate product models and business logic that will guide production. This is also when financial and technical resources are audited for feasibility.
3. Design: A design specification document is created to outline technical design requirements such as programming language, hardware, data sources, architecture, and services.
4. Implementation: The source code is developed using the models, logic and requirements designated in the prior stages. Typically, the system is designed in smaller components, or units, before being implemented together

5. Verification: This is when quality assurance, unit, system, and beta tests take place to report issues that may need to be resolved. This may cause a forced repeat of the coding stage for debugging. If the system passes the tests, the waterfall continues forward
6. Maintenance: Corrective, adaptive and perfective maintenance is carried out indefinitely to improve, update and enhance the final product. This could include releasing patch updates or releasing new versions.

The waterfall approach is ideal for projects that have specific documentation, fixed requirements, ample resources, an established timeline, and well-understood technology. Some advantages of the waterfall model are: Upfront documentation and planning stages allow for large or shifting teams to remain informed and move towards a common goal, disciplined organization, simple to understand, follow and arrange tasks and facilitates departmentalization and managerial control based on schedule or deadlines. Figure 2-1 represents the model discussed.

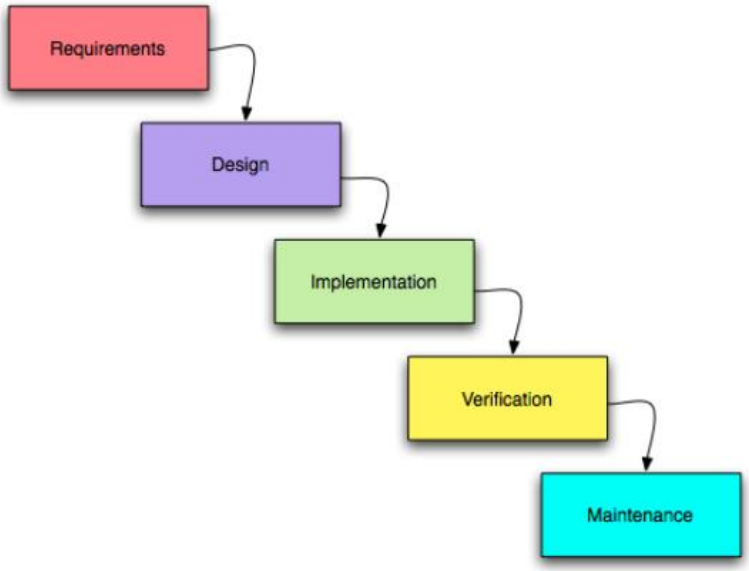

Figure 2-1 - Waterfall Model Diagram

In this project, all states were explored, with the exception of the maintenance phase.

3 Analysis

3.1.1 Requirements

In product development and process optimization, a requirement is a singular documented physical or functional need that a particular design, product or process aims to satisfy. In this case, the project has as requirements:

- Allow programming a simple program and deploy it on FPGA
- Be scalable for future upgrades

3.1.2 Constraints

On the other hand, the constraints are those that limit the output according to technical or nontechnical factors. In this project, the constraints are:

- Implement instructions from all instruction set categories (arithmetic, logic, data transfer, and jumping instructions)
- Implement peripherals
- Only two persons per team
- Limited time resources (project deadline at the end of the semester)

3.1.3 System Overview

To better understand the parts there composed the system and it's interactions, the Figure 3-1 was developed.

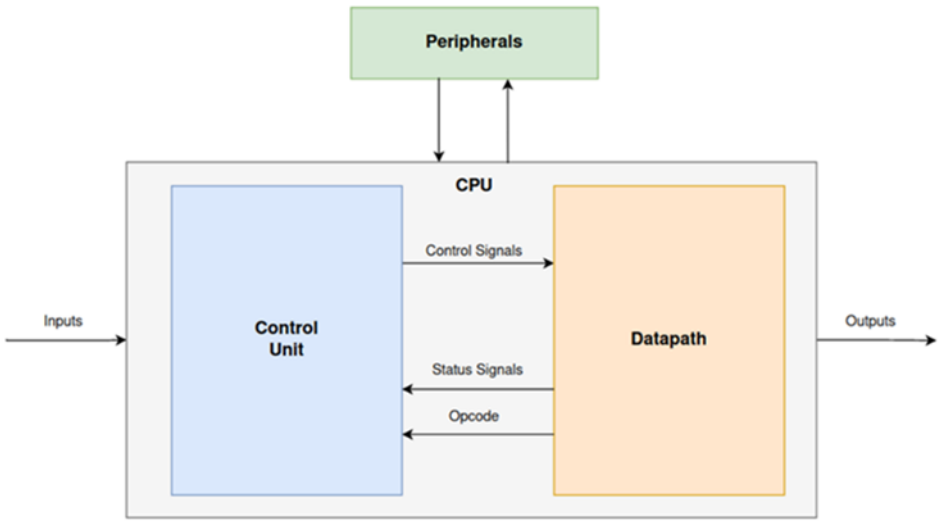

Figure 3-1 - System overview

In this case, it's possible to see that the CPU is composed by two small subsystems, Control Unit and Datapath, and its interactions. Additionally, some peripherals were developed and will be presented in section 4.4.

3.1.4 Instruction Set format

The instructions specify the actions that must be due by the Control Unit and the Datapath to realize the desired jobs. To be executed by the processor the instruction needs to be codified in a specific pattern. There are two basic types of formats, fixed length instructions and variable length instructions. The dimension or length of an instruction depends on the number of operand addresses, and if these addresses identify registers or memory positions.

The original 8051 have a variable length instruction set, as can been seen in

Format	1	2	3	4	5	6	7	8	9	10
Occurrence	25	12	12	36	2	3	3	12	2	1
Percentage (%)	23,1	11,1	11,1	33,3	1,9	2,8	2,8	11,1	1,9	0,9

Table 3-1 - Instructions Distribution by Format

Analyzing the Table 3-1, it is possible to conclude that, in reality, there is no clear predominance of format 4 over format 1, so it becomes difficult to classify this processor in terms of number of operands.

However, to follow a more aligned approach, and taking into account the discussed above, this 8051 version will have a fixed length intrusion set that specifies one operand.

3.1.5 One address processor

One address processor uses an internal/implicit register, named Accumulator, to store one of the input operands and the operation result. The instruction only specifies the address of one of the input operands and the address of the next instruction is implicit specified in the Program Counter (PC) register. Figure 3-2 represents the instruction set format for one address processor.

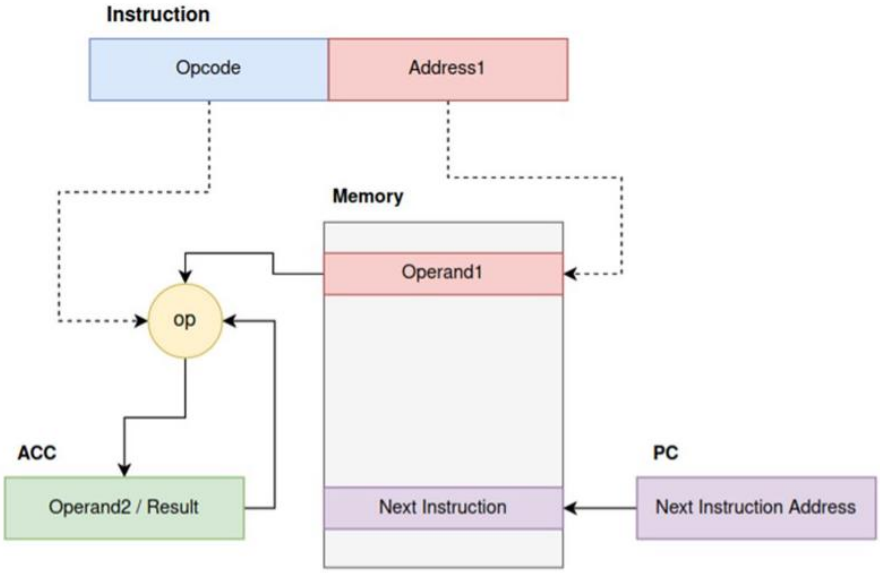

Figure 3-2 - Instruction format for 1-adress processor

3.1.6 Instruction Set

The implemented instruction set is represented in Table 3-2 and the original one is represented in Appendix A.

Arithmetic	Logical	Data Transfer	Jump
ADD A, direct	ANL A, direct	MOV A, direct	JC rel
ADD A, immediate	ANL A, immediate	MOV A, immediate	JNC rel
ADD A, Rn	ANL A, Rn	MOV A, Rn	JZ rel
ADDC A, direct	ORL A, direct	MOV direct, A	JNZ rel
ADDC A, immediate	ORL A, immediate	MOV Rn, direct	RETI
ADDC A, Rn	ORL A, Rn	-	-
SUBB A, direct	XRL A, direct	-	-
SUBB A, immediate	XRL A, immediate	-	-
SUBB A, Rn	XRL A, Rn	-	-

Table 3-2 - Instruction Set

In this context, more instructions were not implemented since most of them are just repetitive and do not add more complexity to the system. However, in a future version more instructions will be added.

4 Design

4.1 Control Unit

The control Unit is responsible to manage the processor state machine. In this case, the state machine is composed by six stages:

1. Start state, which promotes an extra clock cycle for the jump instructions
2. Fetch1 state, which fetch the first value (Opcode) from ROM
3. Wait state, that waits for ROM to put the value outside and save the opcode
4. Fetch2 state, which fetch the second (Operand one) from ROM
5. Decode state, which decodes the operation
6. Execut**e** state that executes the operation

The Figure 4-1 represents the state machine previously discussed.

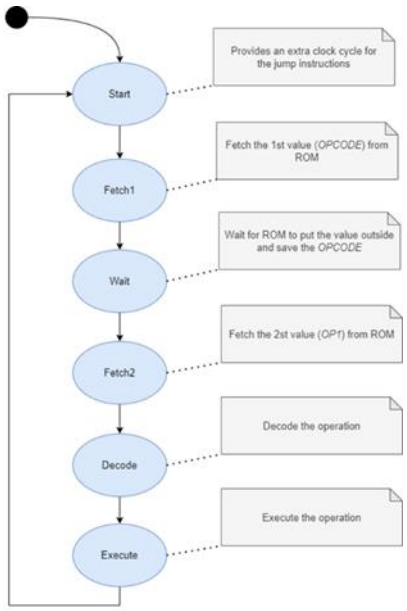

Figure 4-1 - Control Unit State Machine

Also, the Control Unit inputs and outputs can be visualized in the Analysis section, more specifically in Figure 3-1.

4.1.1 Decoder

The decoder module is a combinational logic circuit that is responsible to decode the operation into a valid state. For that reason, this module has one input, the instruction opcode, and one output, the state decoded.

The Figure 4-2 shows the Decoder module.

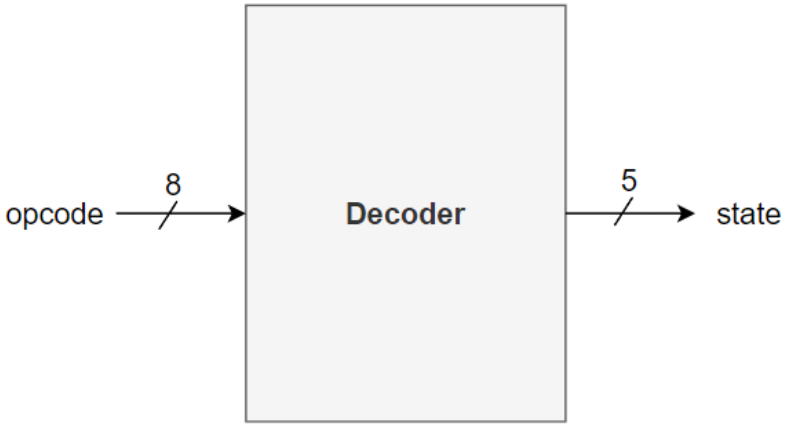

Figure 4-2 - Decoder Module

4.2 Control signals

To establish the communication between the Control Unit and the Datapath, were created a set of control signals, represented in Figure 4-3.

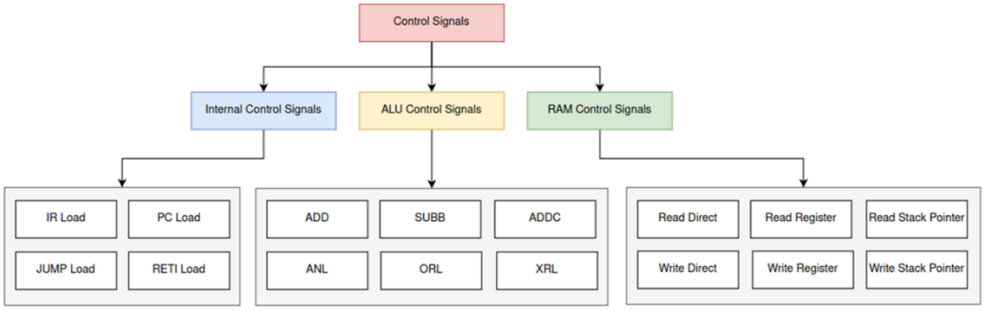

Figure 4-3 - Control Signals

In this case, the control signals are divided into three main categories:

1. Internal control signals
2. ALU control signals
3. RAM control signals

This approach makes it possible to increase packaging and reduce dependencies between the two subsystems, since the Datapath does not need to know the current state, but rather whether, for example, it is necessary to perform an addition operation or whether it is necessary to read from memory.

4.3 Datapath

The Datapath is responsible for carrying out all operations on the data. For that reason, in this section, will be presented all modules that together constitutes the Datapath.

4.3.1 ROM

Read Only Memory, or more known as ROM, is a type of non-volatile memory used in computers and other electronic devices. Data stored in ROM cannot be electronically modified after the manufacture of the memory device. Read-only memory is useful for storing software that is rarely changed during the life of the system and for that reason, it is the memory which the code segment will be kept.

The Figure 4-4 represents the ROM module.

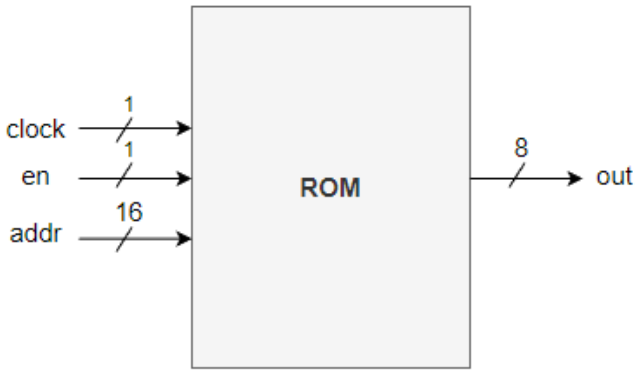

Figure 4-4 - ROM Module

As can been seen in Figure 4-4, this module has as inputs:

1. Clock, representing the clock source
2. Enable, representing the ROM enable
3. Sixteen bit-address, specifying the read address within the available range (0x0000-0xFFFF)

As outputs the module it has an eight-bit output that represents the value read from memory.

4.3.2 RAM

The RAM, Random-Access Memory, is a form of computer memory that can be read and changed in any order, typically used to store working data. For that reason, this module has as inputs:

1. Clock, representing the clock source
2. Enable, representing the RAM enable
3. Eight-bit address, because the memory goes from 0x00 to 0xFF
4. Write byte, handling the byte or bit to write
5. Operation field, that specifies the type of operation to perform

As the real Intel 8051 has some memory that is bit and non-bit addressable, the operation field specifies if the operation is, for example, a write byte or write bit operation. In a more technical perspective, the bit addressable zone goes from 0x20 to 0x2F and some SFR's are also bit or non-bit addressable (The accumulator is bit addressable and the TH0, TL0 and TMOD are non-bit addressable).

The Figure 4-5 represents the RAM module.

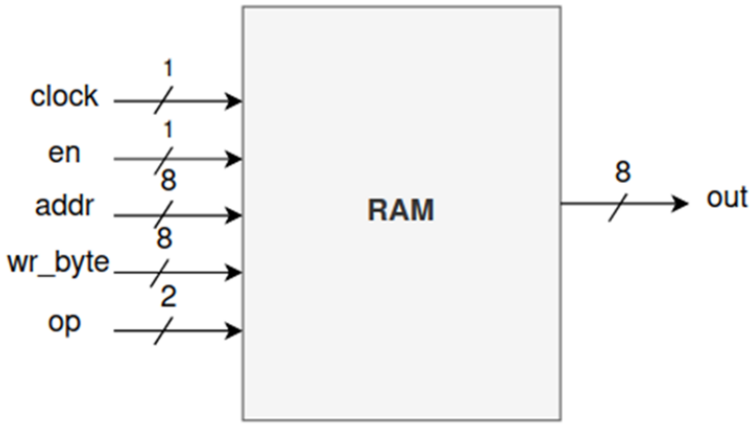

Figure 4-5 - RAM Module

4.3.3 ALU

An Arithmetic Logic Unit, better known as ALU, is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. This contrasts with a floating-point unit, which operates on floating point numbers.

The Figure 4-6 represents the ALU module.

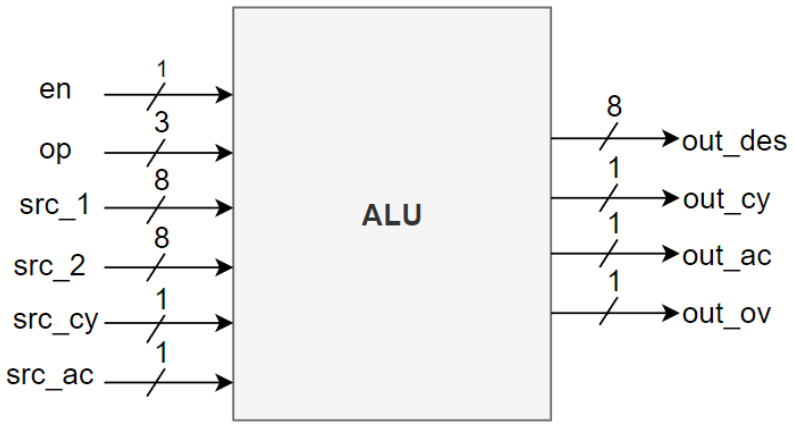

Figure 4-6 - ALU Module

As can been seen in Figure 4-6, the respectively module has as inputs:

1. Enable, specifying the enable flag
2. Operation, specifying the type of operation to perform (*ADD*, *SUBB*, *ANL* etc.)
3. Source_1, which indicates the first operand
4. Source_2, which indicates the second operand
5. Source_cy, which indicates the carry
6. Source_ac, which indicates the auxiliary carry

As outputs, the module has four, of which:

1. Out_des, which indicates the 8-bit output value
2. Out_cy, which indicates the carry out
3. Out_ac, which indicates the auxiliary carry
4. Out_ov, which indicates the overflow

4.3.4 SFR's

In this section, to develop a modular and faster SFR's access, a register file was created. In this case, as a register file is a type of memory that is closed to the CPU, the exchange of information can be faster that other type of memories.

In this specific case, this module has two main advantages:

1. Increase the system modularity and abstraction, because as this module handles all SFR's in the system, if the programmer intends to insert new ones the process is a lot easier.
2. Increase the system performance.

The Figure 4-7 represents the Special Function Registers module.

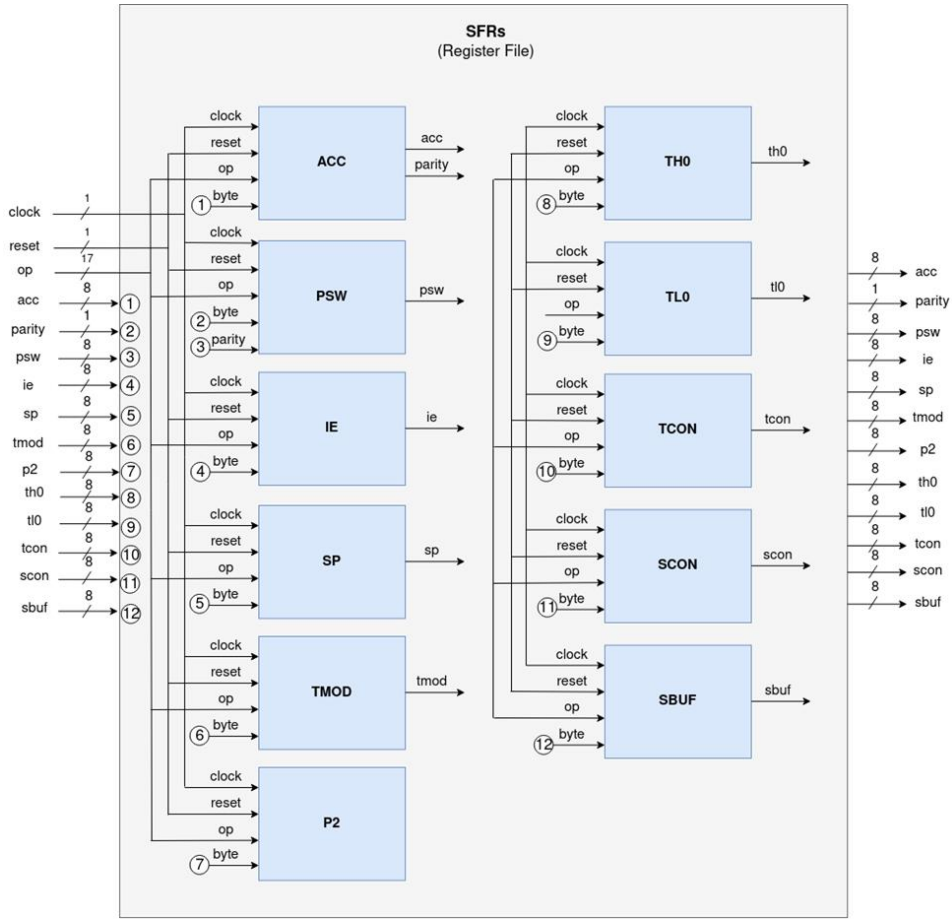

Figure 4-7 - Special Function Registers Module

As can been seen in Figure 4-7, the group also developed a communication type that allows the system to write or read *n* SFRs in one only clock cycle. This feature has huge importance because the 8051 has a lot of instructions that can changes the value from *n* SFRs. As a way of example, the ADD operation can change the accumulator value and the PSW value, and with this type of approach the system is able to perform all these operations in one clock cycle, increasing the overall system performance.

4.3.5 Interrupts

An interrupt is defined as any event that interrupts the normal execution of the processor, diverting it to a specific code segment. There are several types of interrupts, such as:

- External, caused by the activation of an electrical signal generated by a device external to the processor.
- Traps, caused by the execution of special instructions from the instruction set
- Exceptions, triggered by the attempt to execute an illegal or unknown operation

When an interrupt occurs, the processor suspends execution of the current code segment and starts processing interrupt-specific code, also called an ISR, Interrupt Service Routine. In this project, four interrupts were implemented, more specifically:

- Timer 0 interrupt
- External 0 interrupt
- UART Reception interrupt
- UART Transmission interrupt

However, the interruptions caused by the reception of a character in the UART or by the transmission of a character by the UART share the same ISR, and it is up to the programmer to distinguish between them through the special function register SCON. Also, in this project, as the Professor suggested, nested interrupts were not implemented.

To better understand how the interrupts work and how works a logical sequence of an interrupt cycle, see Figure 4-8 and Figure 4-9.

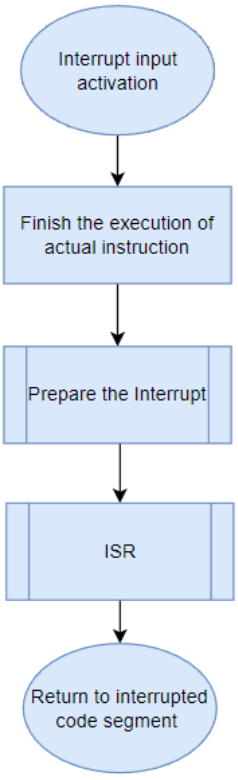

Figure 4-8 - Logical Sequence of an Interrupt Cycle (1)

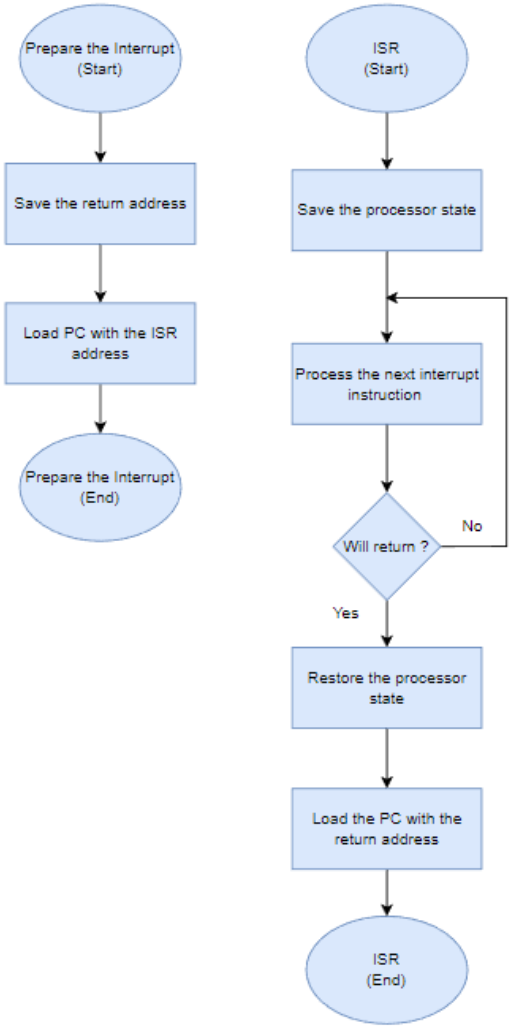

Figure 4-9 - Logical Sequence of an Interrupt Cycle (2)

The Figure 4-10 represents the interrupt module.

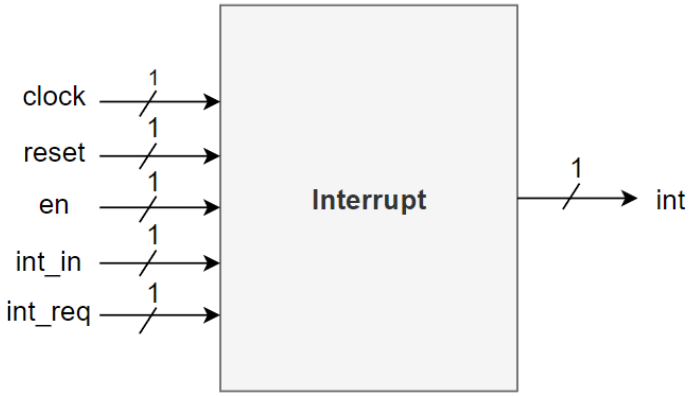

Figure 4-10 - Interrupt Module

In this case, it is possible to visualize that all four interrupts follow the same model, where have as inputs:

1. Clock, representing the clock source
2. Reset, representing the reset source
3. Enable, representing the interrupt enable
4. Int_in, which represents if the interrupt is already being executed or not (no nested interrupts)
5. Int_req, which indicates if any request was made

To manage all interrupts, the special functional register IE, instantiated in the SFR module in section 4.3.4, was created. The Figure 4-11 represents the Interrupt Enable structure.

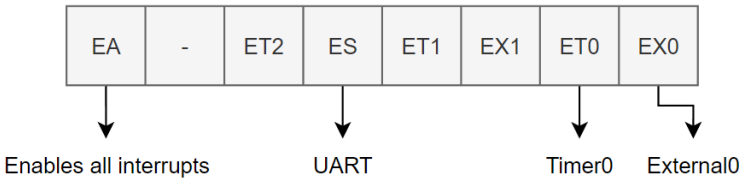

Figure 4-11 - Interrupt Enable (IE)

4.4 Peripherals

In this section will be presented the 8051 peripherals developed.

4.4.1 Timer

The counters and timers' units are one of the most Embedded Systems fundamental features. As almost microcontrollers applications require timing constraints and this same units are integrated with the microcontrollers.

The 8051 family have several of these units, which have different functionalities. One of the capabilities is the automatic loading of the count register after its overflow. In this case, the counter/timer uses another latch register to store the count value. This characteristic allows the counter/timer to produce a very regular output and is used, for example, to generate the clock ticks of an RTOS (Real Time Operating Systems) or to generate the baud rate of the UART.

In this context, and in these 8051 versions, was implemented the Timer 0 with all the four modes. The four modes are:

1. 13 bits timer (TH0 as 8-bit and TL0 as 5-bit prescaler)

2. 16 bits timer (TH0 as higher counter value and TL0 as lower count value)
3. 8 bits timer with auto reload (TL0 as counter value and TH0 as auto reload)
4. TH0 and TL0 runs as 8 bits with auto reload controlled by TR0, INT0, TF0 and TR1, INT1, TF1, respectively

The Figure 4-12 shows the timer module.

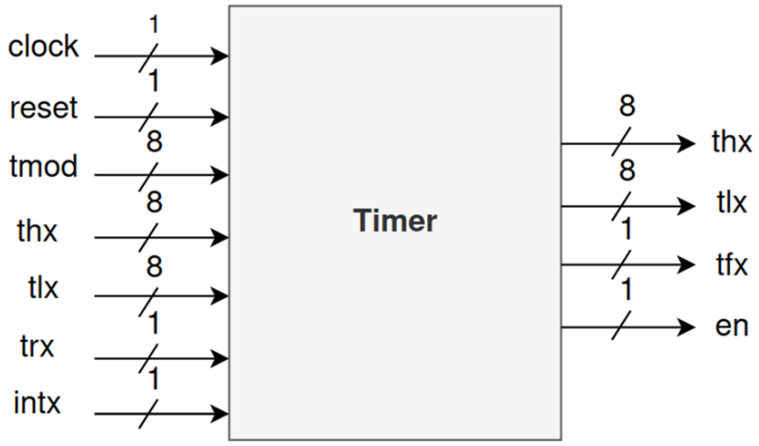

Figure 4-12 - Timer Module

In Figure 4-12 it is possible to see that the Timer module has as inputs the clock and an reset, a *tmod* that specifies the timer mode operation, the *thx* and *tlx* specifying the counter variables and the *trx* and *intx* that will enable, or not, the timer. As outputs, the *thx* and *tlx* represents the updated counter variables, the *tfx* the overflow flag and *en* represents if the timer is running or not.

Note that the *x* ending exists since this module is generic and can be instantiated to, for example, implement Timer 0 and 1.

To manage the timer, the special functional registers TMOD and TCON, instantiated in the SFR module in section 4.3.4, were created. The Figure 4-13 and Figure 4-14 represents the TMOD and TCON structures, respectively.

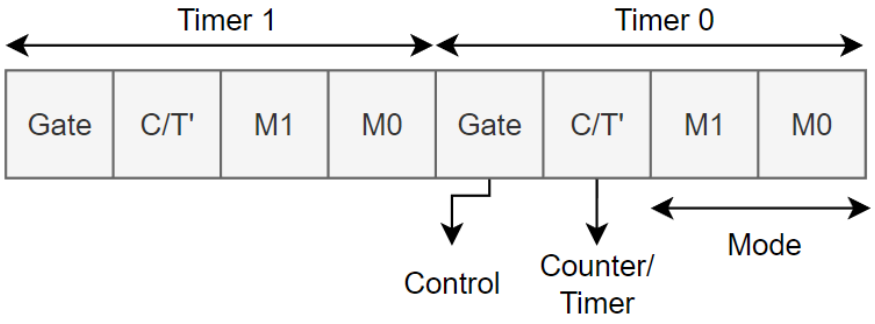

Figure 4-13 – TMOD

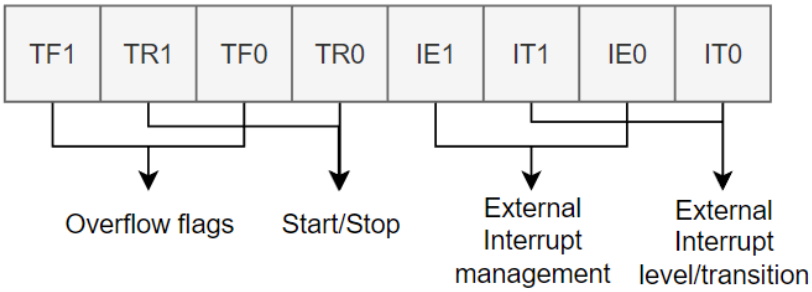

Figure 4-14 – TCON

4.4.2 UART

Although the data bus of a processor is designed for the parallel and simultaneous transfer of all data bits, there are cases where it is preferable that this communication be carried out through a single line.

In the 8051 family, the serial port allows data transfer in full-duplex mode and its hardware can be accessed through the TX and RX pins, also featuring an internal buffer, SBUF, capable of storing the byte received by the serial port and capable of to store the byte to be sent by it.

In this 8051 version, an eight-bit UART with fixed baud rate was implemented. However, all other modes, where for example a ninth bit is added to control the communication or implement it with variable baud rate, may be implemented in a future version, since the addition of these new functionalities does not add much complexity to the respective system.

The Figure 4-15 represents the UART module.

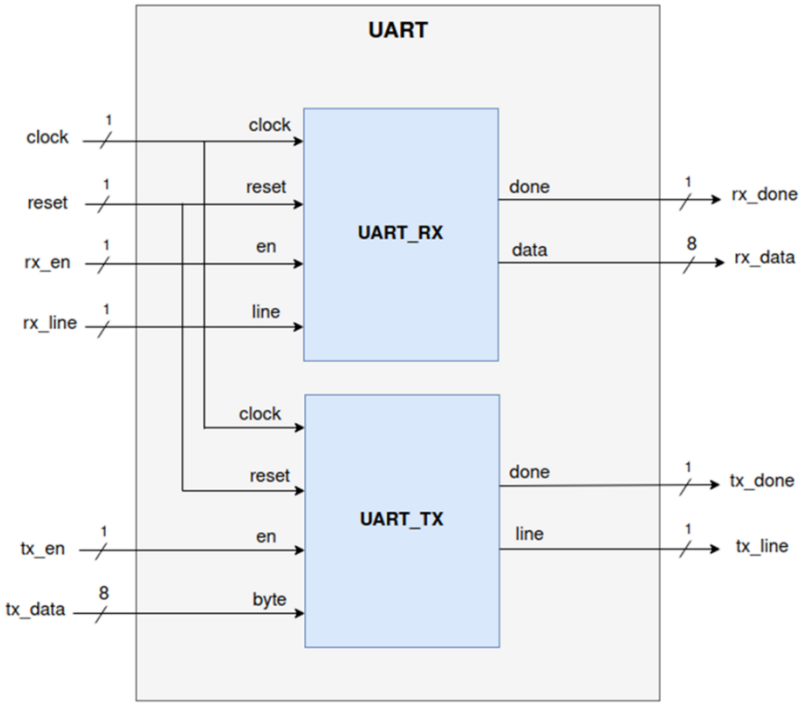

Figure 4-15 - UART Module

In this case, the UART module has as inputs:

1. Clock, representing the clock source
2. Reset, representing the reset source
3. RX_en, representing the reception enable flag
4. RX_line, representing the RX line from which data will be acquired
5. TX_ex, representing the transmission enable flag
6. TX_data, representing the byte that will be sent through the serial port

As outputs:

1. RX_done, where it will be activated after receiving the 8-bit data
2. RX_data, where handles the byte received
3. TX_done, where it will be activated after the transmission of the 8-bit data
4. TX_line, which represents the bit to be transmitted to the Tx line to perform the operation

To manage the UART, the special functional registers SBUF and SCON, instantiated in the SFR module in section 4.3.4, were created. The SBUF is a simple 8-bit data buffer and the SCON structure is represented in Figure 4-16.

SM0	SM1	SM2	REN	TB8	RB8	T1	R1

Figure 4-16 – SCON

The important thing here is that, as bit 4 of the SCON special register, REN, is responsible for activating reception, it must be connected to input *rx_en* of the UART module. Also, as the outputs *rx_done* and *tx_done* represent, respectively, the end of data reception and transmission through the serial port, these must be connected to bits R1 and T1 of SCON.

Transmission

To perform the transmission, a finite state machine, presented in Figure 4-17, was developed. In this case, the state machine is composed by four states:

1. Idle state
2. Start Bit state
3. Data Bits state
4. Stop Bit state

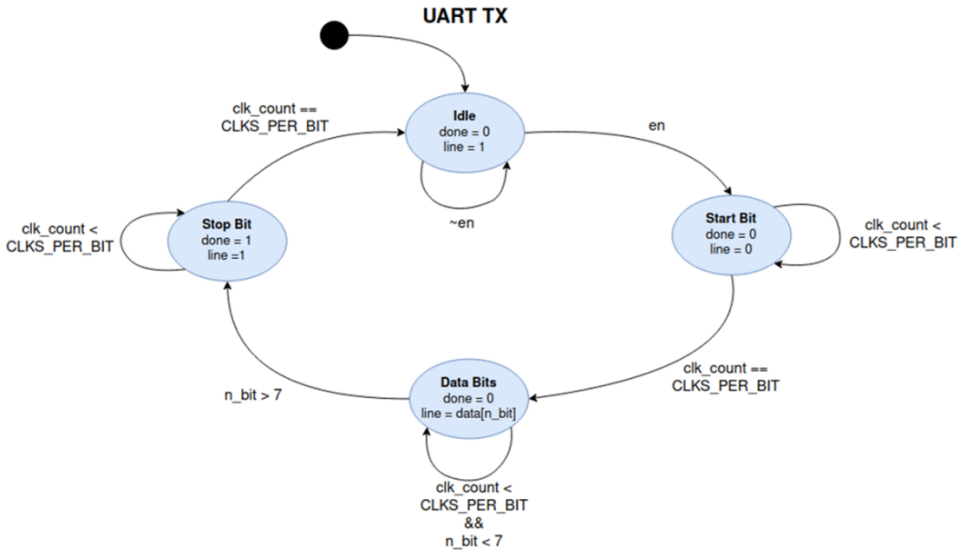

Figure 4-17 - UART Transmission state machine

In a more technical perspective, in the Idle state, the TX line is placed at a high logical level, in order to guarantee that in the Start Bit state the start bit can be set, pulling the line to zero. After the enable bit goes to one, there is a state transition to the Start Bit state. This state waits for a predetermined time and changes to the Data Bits state. This predetermined time is the time necessary for it to be possible to put

something on the line or remove something from it. That said, knowing the clock frequency and the UART baud rate, just divide these two values to calculate this time.

Then, a loop is entered where for eight iterations the data bits are sent to the line. After transmitting all this data, there is a transition to the Stop Bit state, which is responsible for signaling the end of the operation. The stop signal is performed by raising the logical level of the line from 0 to 1 and the flag responsible for signaling that the byte has been transmitted is also placed at logical level one.

Reception

As the transmission, in reception a finite state machine was also developed, represented by Figure 4-18. This state machine has the same states as transmission, and his behavior is the opposite.

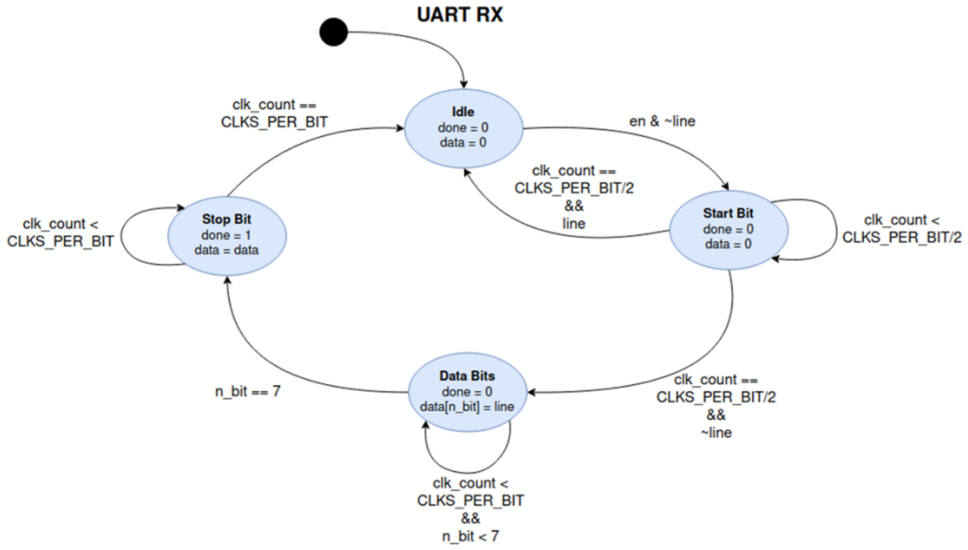

Figure 4-18 - UART Reception state machine

In a more deeply explanation, after the reception enable flag becomes active and the line goes to zero, it is checked if in the middle of the Start Bit state the line is stable and with a logical zero value. If yes, there are conditions to proceed with the reception. Otherwise, the reception is aborted and the state Idle is switched to.

After that, the data bits are acquired in the Start Bit state and after receiving all of them, the transition to the Stop Bit state takes place. In this state, the data byte is stored and a flag responsible for signaling the end of reception is placed at the logical level.

4.4.3 SPI

The SPI, or Serial Peripheral Interface, is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems. The interface was developed by Motorola in the mid-1980s and has becomes a standard.

In this context, a SPI module was developed, and it's represented by Figure 4-19.

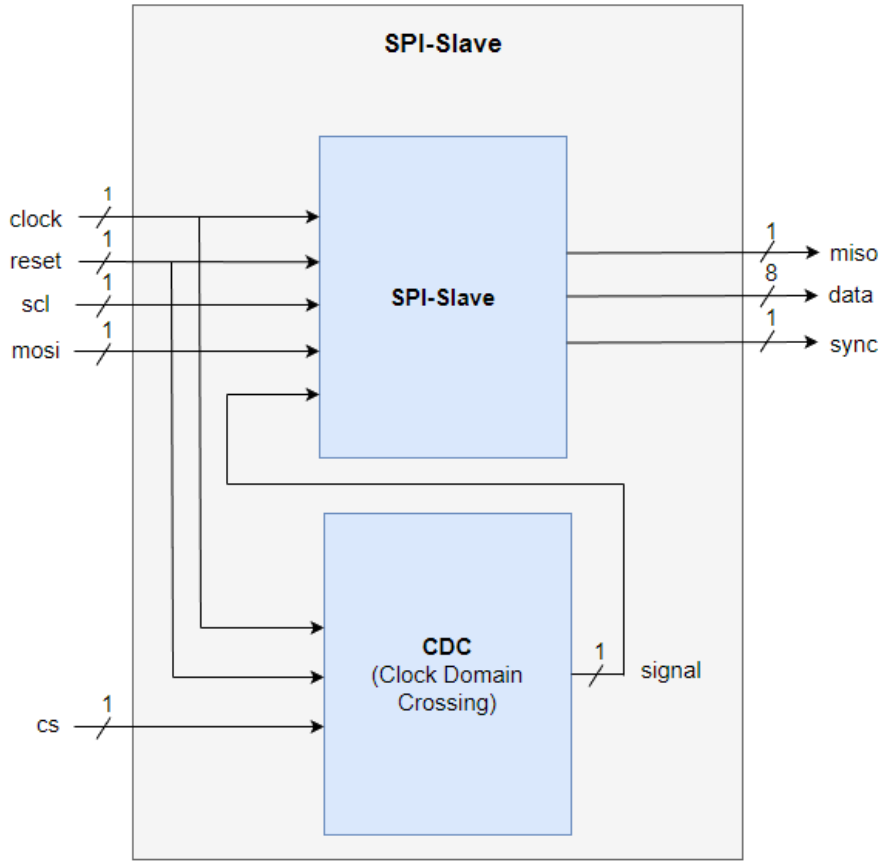

Figure 4-19 - SPI Slave Module

This module has as inputs:

1. Clock, that specifies the clock source
2. Reset, specifying the reset source
3. SCL, specifying the serial clock (output from master)
4. MOSI, data output from master

Ans has outputs:

1. MISO, data output from slave
2. DATA, byte received

3. SYNC, control signal synchronized

Also, it is possible to visualize the presence of one submodule called CDC, or Clock Domain Crossing, that is the traversal of a signal in a synchronous digital circuit from one clock domain into another. If a signal does not assert long enough and is not registered, it may appear asynchronous on the incoming clock boundary. In this case, as the system has two clock sources, it is possible to find Metastability, where the output of a flip-flop inside of the FPGA is unknown, or non-deterministic.

When a metastable condition occurs, there is no way to tell if the output of our flip-flop is going to be a 1 or a 0. A metastable condition occurs when setup or hold times are violated. In this context, the Metastability can cause the FPGA to exhibit very strange behavior. This situation can be fixed by adding two flip-flops. This means that the signal that is asynchronous to the clock is being sampled by the first flip-flop. This will create a metastable condition at the output. The output of the second flip-flop will be stable.

4.4.4 LEDs Controller

To map the four existing LEDs of the board, a module was created that allows the programmer to be abstracted. This module is represented by Figure 4-20.

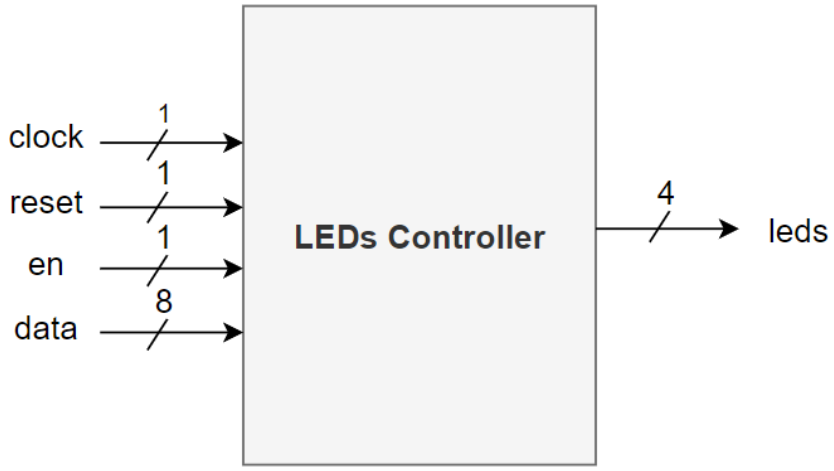

Figure 4-20 - LEDs Controller Module

5 Implementation

In this chapter some implementations in the Verilog language of the modules conceived in chapter 4, referring to the design, will be presented and duly discussed.

5.1 Defines Module

In order to define the microprocessor Instruction Set and some system variables, the *Defines.v* module was created.

First, it was necessary to define the instruction set. In this case, the entire 8051 instruction set was defined so that, in future versions, more instructions could be added. The Listing 5-1 represents the instruction set.

```
//Opcode [4:0]
define ACALL          8bxxx1_0001 // absolute call
define AJMP           8bxxx0_0001 // absolute jump

//Opcode [7:3]
define INC_R          8b0000_1xxx // increment Rn
define DEC_R          8b0001_1xxx // decrement reg Rn=Rn-1
define ADD_R          8b0010_1000 // add A=A+Rx
define ADDC_R         8b0011_1000 // add A=A+Rx+c
define ORL_R          8b0100_1000 // or A=A or Rn
define ANL_R          8b0101_1000 // and A=A^Rx
define XRL_R          8b0110_1000 // XOR A=A XOR Rn
define MOV_CR         8b0111_1xxx // move Rn=constant
define MOV_RD         8b1000_1xxx // move (direct)=Rn
define SUBB_R         8b1001_1000 // substract with borrow A=A-c-Rn
define MOV_DR         8b1010_1xxx // move Rn=(direct)
define CJNE_R         8b1011_1xxx // compare and jump if not equal; Rx<>constant
define XCH_R          8b1100_1xxx // exchange A<->Rn
define DJNZ_R         8b1101_1xxx // decrement and jump if not zero
define MOV_R          8b1110_1000 // move A=Rn
define MOV_AR         8b1111_1000 // move Rn=A

//Opcode [7:1]
define ADD_I          8b0010_011x // add A=A+@Ri
define ADDC_I         8b0011_011x // add A=A+@Ri+c
define ANL_I          8b0101_011x // and A=A^@Ri
define CJNE_I         8b1011_011x // compare and jump if not equal; @Ri<>constant
define DEC_I          8b0001_011x // decrement indirect @Ri=@Ri-1
define INC_I          8b0000_011x // increment @Ri
define MOV_I          8b1110_011x // move A=@Ri
define MOV_ID         8b1000_011x // move (direct)=@Ri
define MOV_AI         8b1111_011x // move @Ri=A
define MOV_DI         8b1010_011x // move @Ri=(direct)
define MOV_CI         8b0111_011x // move @Ri=constant
define MOVX_IA         8b1110_001x // move A=(@Ri)
define MOVX_AI         8b1111_001x // move (@Ri)=A
define ORL_I          8b0100_011x // or A=A or @Ri
define SUBB_I         8b1001_011x // substract with borrow  A=A-c-@Ri
define XCH_I          8b1100_011x // exchange A<->@Ri
define XCHD           8b1101_011x // exchange digit A<->Ri
define XRL_I          8b0110_011x // XOR A=A XOR @Ri

//Opcode [7:0]
define ADD_D          8b0010_0101 // add A=A+(direct)
define ADD_C          8b0010_0100 // add A=A+constant
define ADDC_D         8b0011_0101 // add A=A+(direct)+c
define ADDC_C         8b0011_0100 // add A=A+constant+c
define ANL_D          8b0101_0101 // and A=A^(direct)
define ANL_C          8b0101_0100 // and A=A^constant
define ANL_AD         8b0101_0010 // and (direct)=(direct)^A
define ANL_DC         8b0101_0011 // and (direct)=(direct)^constant
define ANL_B          8b1000_0010 // and c=c^bit
define ANL_NB         8b1011_0000 // and c=c^!bit
define CJNE_D         8b1011_0101 // compare and jump if not equal; a<>(direct)
define CJNE_C         8b1011_0100 // compare and jump if not equal; a<>constant
define CLR_A          8b1110_0100 // clear accumulator
```

```
define CLR_C          8b1100_0011 // clear carry
define CLR_B          8b1100_0010 // clear bit
define CPL_A          8b1111_0100 // complement accumulator
define CPL_C          8b1011_0011 // complement carry
define CPL_B          8b1011_0010 // complement bit
define DA             8b1101_0100 // decimal adjust (A)
define DEC_A          8b0001_0100 // decrement accumulator a=a-1
define DEC_D          8b0001_0101 // decrement direct (direct)=(direct)-1
define DIV            8b1000_0100 // divide
define DJNZ_D         8b1101_0101 // decrement and jump if not zero (direct)
define INC_A          8b0000_0100 // increment accumulator
define INC_D          8b0000_0101 // increment (direct)
define INC_DP         8b1010_0011 // increment data pointer
define JB             8b0010_0000 // jump if bit set
define JBC            8b0001_0000 // jump if bit set and clear bit
define JC             8b0100_0000 // jump if carry is set
define JMP_D          8b0111_0011 // jump indirect
define JNB            8b0011_0000 // jump if bit not set
define JNC            8b0101_0000 // jump if carry not set
define JNZ            8b0111_0000 // jump if accumulator not zero
define JZ             8b0110_0000 // jump if accumulator zero
define LCALL          8b0001_0010 // long call
define LJMP           8b0000_0010 // long jump
define MOV_D          8b1110_0101 // move A=(direct)
define MOV_C          8b0111_0100 // move A=constant
define MOV_AD         8b1111_0101 // move (direct)=A
define MOV_DD         8b1000_0101 // move (direct)=(direct)
define MOV_CD         8b0111_0101 // move (direct)=constant
define MOV_BC         8b1010_0010 // move c=bit
define MOV_CB         8b1001_0010 // move bit=c
define MOV_DP         8b1001_0000 // move dptr=constant(16 bit)
define MOVC_DP        8b1001_0011 // move A=dptr+A
define MOVC_PC        8b1000_0011 // move A=pc+A
define MOVX_PA        8b1110_0000 // move A=(dptr)
define MOVX_AP        8b1111_0000 // move (dptr)=A
define MUL            8b1010_0100 // multiply a*b
define NOP            8b0000_0000 // no operation
define ORL_D          8b0100_0101 // or A=A or (direct)
define ORL_C          8b0100_0100 // or A=A or constant
define ORL_AD         8b0100_0010 // or (direct)=(direct) or A
define ORL_CD         8b0100_0011 // or (direct)=(direct) or constant
define ORL_B          8b0111_0010 // or c = c or bit
define ORL_NB         8b1010_0000 // or c = c or !bit
define POP            8b1101_0000 // stack pop
define PUSH           8b1100_0000 // stack push
define RET            8b0010_0010 // return from subrutine
define RETI           8b0011_0010 // return from interrupt
define RL             8b0010_0011 // rotate left
define RLC            8b0011_0011 // rotate left thrugh carry
define RR             8b0000_0011 // rotate right
define RRC            8b0001_0011 // rotate right thrugh carry
define SETB_C         8b1101_0011 // set carry
define SETB_B         8b1101_0010 // set bit
define SJMP           8b1000_0000 // short jump
define SUBB_D         8b1001_0101 // substract with borrow  A=A-c-(direct)
define SUBB_C         8b1001_0100 // substract with borrow  A=A-c-constant
define SWAP           8b1100_0100 // swap A(0-3) <-> A(4-7)
define XCH_D          8b1100_0101 // exchange A<->(direct)
define XRL_D          80110_0101 // XOR A=A XOR (direct)
define XRL_C          8b0110_0100 // XOR A=A XOR constant
define XRL_AD         8b0110_0010 // XOR (direct)=(direct) XOR A
define XRL_CD         8b0110_0011 // XOR (direct)=(direct) XOR constant
```

Listing 5-1 - Instruction Set Definition

As can been seen in Listing 5-1, this is fully commented and divided into sections, to facilitate its visualization and understanding.

Next, was defined the control signals, responsible to establish the communication between the Control Unit and the Datapath. As a way of example, the Listing 5-2 and Listing 5-3 represents the RAM and ALU control signals, respectively.

```
define RAM_CS_LEN      3
define RAM_CS_DEFAULT  RAM_CS_LENd0
define RAM_CS_RD_D     RAM_CS_LENd1         // RAM read DIRECT control signal
define RAM_CS_RD_R     RAM_CS_LENd2         // RAM read REGISTER control signal
```

```
define RAM_CS_WR_D      RAM_CS_LENd3        // RAM write DIRECT control signal
define RAM_CS_WR_R      RAM_CS_LENd4        // RAM write REGISTER control signal
define RAM_CS_RD_SP     RAM_CS_LENd5        // RAM read SP control signal
define RAM_CS_WR_SP     RAM_CS_LENd6        // RAM write SP control signal
```

Listing 5-2 - RAM Control Signals

```
define ALU_CS_LEN       3
define ALU_CS_NOP       ALU_CS_LENd0
define ALU_CS_ADD       ALU_CS_LENd1        // ADD control signal
define ALU_CS_ADDC      ALU_CS_LENd2        // ADDC control signal
define ALU_CS_SUB       ALU_CS_LENd3        // SUB control signal
define ALU_CS_AND       ALU_CS_LENd4        // AND control signal
define ALU_CS_XOR       ALU_CS_LENd5        // XOR control signal
define ALU_CS_OR        ALU_CS_LENd6        // ORL control signal
```

Listing 5-3 - ALU Control Signals

Next, was defined the SFR code operations. For a more detail explanation, please consult the section 4.3.4. The Listing 5-4 represents some of the SFR's code operations definition.

```
define SFR_OP_LEN               18
define OP_DEFAULT               SFR_OP_LENd0
define OP_ACC_WR_BYTE           SFR_OP_LENd1
define OP_ACC_WR_BIT            SFR_OP_LENd2
define OP_PSW_WR_BYTE           SFR_OP_LENd4
define OP_PSW_WR_BIT            SFR_OP_LENd8
define OP_PSW_WR_FLAGS          SFR_OP_LENd16
define OP_IE_WR_BYTE            SFR_OP_LENd32
define OP_IE_WR_BIT             SFR_OP_LENd64
define OP_SP_WR_BYTE            SFR_OP_LENd128
define OP_SP_PUSH               SFR_OP_LENd256
define OP_SP_POP                SFR_OP_LENd512
define OP_TMOD_WR_BYTE          SFR_OP_LENd1024
define OP_TH0_WR_BYTE           SFR_OP_LENd2048
define OP_TL0_WR_BYTE           SFR_OP_LENd4096
```

Listing 5-4 - SFR's code operations

As can be seen in Listing 5-4, as in the real 8051 some registers are bit or non-bit addressable, this same difference is presented in this version, where, for example, the Accumulator and PSW are bit addressable, and the TMOD, TH0 and TL0 are non-bit addressable. Also, to boost the system performance, was added the code operation *OP_PSW_WR_FLAGS* to update the carry, auxiliary carry, parity, and overflow in a faster way.

Therefore, it becomes necessary to define the SFR's addresses. For that reason, the code represented in Listing 5-5 was developed.

```
define SFR_ADDR_LEN  8
define DEFAULT       SFR_ADDR_LENh00
define ACC_ADDR      SFR_ADDR_LENhE0
define PSW_ADDR      SFR_ADDR_LENhD0
define IE_ADDR       SFR_ADDR_LENhA8
define TH0_ADDR      SFR_ADDR_LENh8C
define TL0_ADDR      SFR_ADDR_LENh8A
define TMOD_ADDR     SFR_ADDR_LENh89
define SP_ADDR       SFR_ADDR_LENh81
define TCON_ADDR     SFR_ADDR_LENh88
define SBUF_ADDR     SFR_ADDR_LENh99
define SCON_ADDR     SFR_ADDR_LENh98
define P2_ADDR       SFR_ADDR_LENhA0
```

Listing 5-5 - SFR's Addresses

Finally, as the system has four interrupts' sources, it becomes necessary to represent the interrupts vector table. The Listing 5-6 represents the interrupts vector table.

```
define ISR_ADDR_LEN     8
define ISR_RST_ADDR     ISR_ADDR_LENh00
define ISR_EXT0_ADDR    ISR_ADDR_LENh03
define ISR_TIM0_ADDR    ISR_ADDR_LENh0b
define ISR_UART_ADDR    ISR_ADDR_LENh23
```

Listing 5-6 - Interrupts Vector Table

As can be seen in Listing 5-6, the event of receive and transmit something from UART shares the same address (0x23) and the reset button case was added, where the program should start after the reset is pressed. In this case, at position 0x00.

5.2 Top Module

As can be seen in Figure 3-1, the CPU, or the top module, must be able to receive the system inputs, provide the system outputs and establish the communication with the peripherals. In this context, the Listing 5-7 represents the top module interface.

```
module CPU(
    input i_clk,                        // FPGA Clock
    input i_scl,                        // SPI Master Clock (e.g. STM32)
    input i_rst,                        // Reset
    input i_mosi,                       // Master Out Slave In
    input i_cs,                         // Chip/Slave Select
    input i_rx,                         // RX line
    input i_button,                     // Button (Connected to External Interrupt 0)
    output o_miso,                      // Master In Slave Out
    output o_tx,                        // TX line
    output [3:0] o_leds                 // LEDS
    );
```

Listing 5-7 - CPU Module Inputs/Outputs

Next, this module is not only responsible to create the system control signals, that will be necessary to communicate between the Control Unit and the Datapath, but also to create some data structures capable of storing some relevant processor status. As a way of example, if a new interrupt flag is pending or not to be able to redirect the signal felt to the control unit. The Listing 5-8 represents what was said above.

```
wire [I_CS_LEN-1:0] internal_cs;    // Internal Control Signals
wire [ALU_CS_LEN-1:0] alu_cs;       // ALU Control Signals
wire [RAM_CS_LEN-1:0] ram_cs;       // RAM Control Signals
wire [7:0] ir;                      // IR (Instruction Register)
wire int_pend;                      // Interrupt pending flag
wire button;                        // In this case, connected to EXT0
```

Listing 5-8 - CPU variables

Finally, the control unit and Datapath modules are instantiated, as well as some of the peripherals. The Listing 5-9 represents what was said above.

```
// Instantiate the Control Unit
ControlUnit ControlUnit(
    .i_clk(i_clk),                  // FPGA Clock
    .i_rst(i_rst),                  // Reset
    .i_ir(ir),                      // Instruction Register
```

```
    .i_int_pend(int_pend),          // Interrupt pending flag
    .o_internal_cs(internal_cs),    // Internal Control Signals
    .o_alu_cs(alu_cs),              // ALU Control Signals
    .o_ram_cs(ram_cs)               // RAM Control Signals
);

// Instantiate the Datapath
Datapath Datapath(
    .i_clk(i_clk),                  // FPGA Clock
    .i_rst(i_rst),                  // Reset
    .i_internal_cs(internal_cs),    // Internal Control Signals
    .i_alu_cs(alu_cs),              // ALU Control Signals
    .i_ram_cs(ram_cs),              // RAM Control Signals
    .i_uart_rx(i_rx),               // RX line
    .i_button(button),              // Button connected to EXT0
    .o_uart_tx(o_tx),               // TX line
    .o_ir(ir),                      // Instruction Register
    .o_int_pend(int_pend),          // Interrupt pending flag
    .o_leds(o_leds_in)
);

// Instantiate the SPI_Slave
SPI_Slave SPI_Slave(
    .i_scl(i_scl),                  // Master Clock (e.g STM32)
    .i_clk(i_clk),                  // FPGA Clock
    .i_rst(i_rst),                  // Reset
    .i_mosi(i_mosi),                // Master Out Slave In (data output from master)
    .i_cs(i_cs),                    // Chip/Slave Select
    .o_miso(o_miso),                // Master In Slave Out (data output from slave)
    .o_data(spi_data),              // SPI output data
    .o_sync(spi_leds_en)            // Output Clock Domain Crossing
);

// Instantiate the LEDS Controller
LEDS_Controller LEDS_Controller(
    .i_clk(i_clk),                  // FPGA Clock
    .i_rst(i_rst),                  // Reset
    .i_en(1b1),                     // Enable the LEDS
    .i_data(o_leds_in),             // LEDS in
    .o_leds(o_leds)                 // LEDS mapped in Zybo-Z7 constraints file
);

// Instantiate the Debounce module to debounce the Button connected to EXT0
Debounce EXT0_Debounce (
    .i_clk(i_clk),                  // Clock
    .i_rst(i_rst),                  // Reset
    .i_signal(i_button),            // Signal unstable
    .o_signal(button)               // Signal stable
);
```

Listing 5-9 - CPU Module

5.3 Control Unit Module

In this section was implemented the control unit system. In this case, after design all the subsystem in section 4.1, this same module was translated to Verilog code. The Listing 5-10 represents the respectively module inputs and outputs.

```
module ControlUnit(
    input i_clk,                            // Clock
    input i_rst,                            // Reset
    input [7:0] i_ir,                       // IR
    input i_int_pend,                       // Interrupt pending
    output [I_CS_LEN-1:0] o_internal_cs,    // Internal Control Signals
    output [ALU_CS_LEN-1:0] o_alu_cs,       // ALU Control Signals
    output [RAM_CS_LEN-1:0] o_ram_cs        // RAM Control Signals
    );
```

Listing 5-10 - Control Unit Module Inputs/Outputs

Next, it becomes necessary to declare some auxiliary variables to handle, for example, the state and the opcode received by the Datapath. These variables are represented in Listing 5-11.

```
// State variable
reg [4:0] state;

// Auxiliar register to handle the opcode
reg [7:0] AR;

// Decoded state
wire [4:0] decoded_state;
```

Listing 5-11 - Control Unit variables

As can be seen in Listing 5-11, an extra variable called *decoded_state* was created to handle the output from the Decode Module, representing the state decoded. For more information about the decode module, please consult the section 4.1.1. However, further ahead in this section, the same will be presented and discussed.

Next, it becomes necessary to define the states. The Listing 5-12 represents the states definition with some comments allusive to the behavior of each one of them.

```
parameter s_start = 5d0;       // Start state and provides an extra clock cycle for
the Jump instructions (because they change the PC on the execution stage)
parameter s_fetch1 = 5d1;      // Fetch the first value from ROM
parameter s_wait = 5d2;        // Wait for ROM to put the value outside
parameter s_fetch2 = 5d3;      // Fetch the second value from ROM
parameter s_decode = 5d4;      // Extract the information from IR (opcode instruction)
parameter s_add = 5d5;         // ADD state (ADD A Rn , ADD A direct , ADD A imme)
parameter s_subb = 5d6;        // SUBB state (SUBB A Rn , SUBB A direct , SUBB A imme)
parameter s_addc = 5d7;        // ADDC state (ADDC A Rn , ADDC A direct , ADDC A imme)
parameter s_and = 5d8;         // AND state (ANL A Rn , ANL A direct , ANL A imme)
parameter s_or = 5d9;          // OR state (ORL A Rn , ORL A direct , ORL A immediate)
parameter s_xor = 5d10;        // XOR state (XRL A Rn , XRL A direct ,XRL A immediate)
parameter s_mov_toA = 5d11;    // MOV to A state(MOV A Rn, MOV A direct, MOV A imme)
parameter s_mov_fromA = 5d12;  // MOV from A state (MOV direct A, MOV Rn A)
parameter s_jumpC = 5d13;      // JC state
parameter s_jumpNC = 5d14;     // JNC state
parameter s_jumpZ = 5d15;      // JZ state
parameter s_jumpNZ = 5d16;     // JNZ state
parameter s_reti1 = 5d17;      // RETI state 1 (Extract from Stack the PC[15:8])
parameter s_reti2 = 5d18;      // RETI state 2 (Wait for RAM to output the SP value)
parameter s_reti3 = 5d19;      // RETI state 3 (Extract from Stack the PC[7:0])
```

Listing 5-12 - States definition

Then, to establish the communication between the Control Unit and the Datapath, the control signals designed in Figure 4-3 were translated to Verilog code. The Listing 5-13, Listing 5-14 and Listing 5-15 represents the internal, ALU and RAM control signals, respectively.

```
assign o_internal_cs = (state == s_fetch1) ? IR_PC_LOAD1 :
                       (state == s_fetch2) ? IR_PC_LOAD2 :
                       (state == s_mov_toA) ? MOV_TO_A :
                       (state == s_mov_fromA) ? MOV_FROM_A :
                       (state == s_jumpC) ? JMP_C_LOAD :
                       (state == s_jumpNC) ? JMP_NC_LOAD :
                       (state == s_jumpZ) ? JMP_Z_LOAD :
                       (state == s_jumpNZ) ? JMP_NZ_LOAD :
                       (state == s_reti1) ? RETI1_CS :
                       (state == s_reti2) ? RETI2_CS :
                       (state == s_reti3) ? RETI3_CS :
                       (state == s_start) ? START_CS : I_CS_NOP;
```

Listing 5-13 - Internal Control Signals

```
assign o_alu_cs = (state == s_add) ? ALU_CS_ADD :
                  (state == s_subb) ? ALU_CS_SUB :
                  (state == s_addc) ? ALU_CS_ADDC :
                  (state == s_and) ? ALU_CS_AND :
                  (state == s_or) ? ALU_CS_OR :
                  (state == s_xor) ? ALU_CS_XOR : ALU_CS_NOP;
```

Listing 5-14 - ALU Control Signals

```
assign o_ram_cs = (AR == ADD_D || AR == ADDC_D || AR == SUBB_D || AR == ANL_D || AR ==
ORL_D || AR == XRL_D || AR == MOV_D) ? RAM_CS_RD_D :
                  (AR == ADD_R || AR == ADDC_R || AR == SUBB_R || AR == ANL_R || AR ==
ORL_R || AR == XRL_R || AR == MOV_R) ? RAM_CS_RD_R :
                  (AR == MOV_AD) ? RAM_CS_WR_D :
                  (AR == MOV_AR) ? RAM_CS_WR_R :
                  (AR == RETI) ? RAM_CS_RD_SP :
                  (AR == INTERRUPT) ? RAM_CS_WR_SP : RAM_CS_DEFAULT;
```

Listing 5-15 - RAM Control Signals

Therefore, to decode the opcode received by the Datapath, the module instantiates the Decoder Module, as can been seen in Listing 5-16.

```
Decoder Decoder(
    .i_opcode(AR),
    .decoded_state(decoded_state)
);
```

Listing 5-16 - Decoder Module Inputs/Outputs

In a more technical perspective, the Decoder module behavior can be visualized in Listing 5-17.

```
assign decoded_state = (i_opcode == ADD_R || i_opcode == ADD_D || i_opcode == ADD_C) ?
s_add :
                       (i_opcode == SUBB_R || i_opcode == SUBB_D || i_opcode ==
SUBB_C) ? s_subb :
                       (i_opcode == ADDC_R || i_opcode == ADDC_D || i_opcode ==
ADDC_C) ? s_addc :
                       (i_opcode == ANL_R || i_opcode == ANL_D || i_opcode == ANL_C) ?
s_and :
                       (i_opcode == ORL_R || i_opcode == ORL_D || i_opcode == ORL_C) ?
s_or :
                       (i_opcode == XRL_R || i_opcode == XRL_D || i_opcode == XRL_C) ?
s_xor :
                       (i_opcode == MOV_R || i_opcode == MOV_D || i_opcode == MOV_C) ?
s_mov_toA :
                       (i_opcode == MOV_AR || i_opcode == MOV_AD) ? s_mov_fromA :
                       (i_opcode == JC) ? s_jumpC :
                       (i_opcode == JNC) ? s_jumpNC :
                       (i_opcode == JZ) ? s_jumpZ :
                       (i_opcode == JNZ) ? s_jumpNZ :
                       (i_opcode == RETI) ? s_reti1 : s_start;
```

Listing 5-17 - Decoder Module behavior

As can be seen in Listing 5-17, with this approach the system is able to save several states, since, for example, all addition operations are in a single state, differing only the type of input operands for the ALU module.

Finally, the control unit state machine designed in the Design phase in Figure 4-1, was translated to code. The Listing 5-18 represents the Control Unit state machine.

```
always @ (posedge i_clk)
begin
    if (i_rst == 1'b1)
    begin
        state <= s_start;
```

```
    end
    else
    begin
        // If exist a valid interrupt to handle
        if (i_int_pend == 1b1)
        begin
            AR = INTERRUPT;
            state <= s_start;
        end
        // Else
        else begin
            case (state)
                s_start:
                    state <= s_fetch1;
                s_fetch1:
                        state <= s_fetch2;
                s_wait:
                    begin
                        AR = i_ir;
                        state <= s_fetch3;
                    end
                s_fetch2:
                    state <= s_decode;
                s_decode:
                    state <= decoded_state;
                s_reti1:
                    state <= s_reti2;
                s_reti2:
                    state <= s_reti3;
                default:
                    state <= s_start;
            endcase
        end
    end
end
```

Listing 5-18 - Control Unit State Machine

5.4 Datapath Module

The Datapath must be able to perform all data operations. In this case, this module must receive the control signals provided by Control Unit and some other additionally parameters and provides the opcode or even a flag responsible for identifying if a new interrupt is illegible or not to run. The respectively module interface can be found in Listing 5-19.

```
module Datapath(
    input i_clk,                                // Clock
    input i_rst,                                // Reset
    input [I_CS_LEN-1:0] i_internal_cs,         // Internal Control Signals
    input [ALU_CS_LEN-1:0] i_alu_cs,            // ALU Control Signals
    input [RAM_CS_LEN-1:0] i_ram_cs,            // RAM Control Signals
    input i_uart_rx,                            // UART RX Line
    input i_button,                             // Button
    output o_uart_tx,                           // UART TX Line
    output [7:0] o_ir,                          // Opcode / OP1
    output o_int_pend,                          // Interrupt to process
    output [7:0] o_leds                         // LED's
    );
```

Listing 5-19 - Datapath Module Inputs/Outputs

Next, after creating all variables to handle all the system operations, it becomes necessary to instantiate all worker modules already explained in the Design phase. As there are several modules and the process is the same for all of them, the instantiation of the ALU module, Timer 0 and UART interrupt will be shown. The Listing 5-20, Listing

5-21 and Listing 5-22 represents, respectively, the ALU, Timer 0 and UART interrupt module in the Datapath.

```
ALU ALU(
    .i_operation(alu_op),                   // ALU operation
    .i_src1(alu_src1),                      // ALU source #1
    .i_src2(alu_src2),                      // ALU source #2
    .i_srcC(alu_srcC),                      // ALU source carry
    .i_srcAc(alu_srcAc),                    // ALU source auxiliary carry
    .o_des1(alu_des),                       // ALU destination value
    .o_desC(alu_desC),                      // ALU destination carry
    .o_desAc(alu_desAc),                    // ALU destination auxiliary carry
    .o_desOv(alu_desOv)                     // ALU destination overflow
);
```

Listing 5-20 - ALU Module in Datapath

```
Timer Timer0 (
    .i_clk(i_clk),                          // Clock
    .i_rst(i_rst),                          // Reset
    .i_tmod(tmod),                          // TMOD
    .i_thx(th0),                            // TH0
    .i_tlx(tl0),                            // TL0
    .i_trx(tcon[4]),                        // TR0
    .i_intx(ie[1]),                         // INT0
    .o_thx(th0_tim_out),                    // TH0 updated
    .o_tlx(tl0_tim_out),                    // TL0 updated
    .o_tfx(tf0_tim_out),                    // TF0 (overflow flag)
    .o_en(tim0_en)                          // Timer is running
);
```

Listing 5-21 - Timer Module in Datapath

```
Interrupt UART_ISR (
    .i_clk(i_clk),                          // Clock
    .i_rst(i_rst),                          // Reset
    .i_en(uart_int_en),                     // Serial I/O Interrupt enable
    .i_int(uart_int_in),                    // Serial I/O Interrupt in progress
    .i_int_req(rx_done | tx_done),          // Serial I/O Interrupt request
    .o_int(uart_int_status)                 // Serial I/O Interrupt status (1 if the
interrupt is eligible to execute)
);
```

Listing 5-22 - UART Interrupt Module in Datapath

Next, it becomes necessary to assign all flags that will feed each module. In this case, and as one more time all of them follows the same protocol, will be show how the ALU and Interrupts flags are updated. The Listing 5-23 and Listing 5-24 represents, respectively, the ALU and Interrupts flags.

```
// Assign the ALU operation
assign alu_op = i_alu_cs;

// Assign the ALU source #1
assign alu_src1 = (i_alu_cs == ALU_CS_ADD || i_alu_cs == ALU_CS_SUB || i_alu_cs ==
ALU_CS_ADDC || i_alu_cs == ALU_CS_AND || i_alu_cs == ALU_CS_OR || i_alu_cs ==
ALU_CS_XOR) ? acc : 8'd0;

// Assign the ALU source #2
// 1) Direct address or Register
// 2) Immediate
assign alu_src2 = ((i_alu_cs == ALU_CS_ADD || i_alu_cs == ALU_CS_SUB || i_alu_cs ==
ALU_CS_ADDC || i_alu_cs == ALU_CS_AND || i_alu_cs == ALU_CS_OR || i_alu_cs ==
ALU_CS_XOR) && (i_ram_cs == RAM_CS_RD_D || i_ram_cs == RAM_CS_RD_R)) ? addr_mapped :
IR;

// Assign the ALU carry source
// 1) If op is ADDC or SUB ==> psw[7]
// 2) Else ==> 0
assign alu_srcC = (i_alu_cs == ALU_CS_ADDC || i_alu_cs == ALU_CS_SUB) ? psw[7] : 0;
```

Listing 5-23 - ALU flags in Datapath

```
// External 0 Interrupt is enabled when:
// EA = 1 && EX0 = 1
assign ext0_int_en = (ie[7] == 1b1 && ie[0] == 1b1) ? 1b1 : 1b0;

// Timer 0 Interrupt is enable when:
// EA = 1 && ET0 = 1
assign tim0_int_en = (ie[7] == 1b1 && ie[1] == 1b1) ? 1b1 : 1b0;

// UART Interrupt is enable when:
// EA = 1 && ES0 = 1
assign uart_int_en = (ie[7] == 1b1 && ie[4] == 1b1) ? 1b1 : 1b0;

// Update the interrupt flag that will signal the CPU that will be a new interrupt to
process
assign o_int_pend = tim0_int_one_pulse | ext0_int_one_pulse | uart_int_one_pulse;

// Update the button status that will connect to External Interrupt 0
assign ext0_button = i_button;
```

Listing 5-24 - Interrupt flags in Datapath

Note that, in the Listing 5-24, the *o_int_pend* flag, responsible to signal the Control Unit that is a valid and illegible interrupt to process, is updated with variables of type *one_pulse*. This happens since the Control Unit should only be alerted once, to be able to prepare the new interruption by saving the address of the next instruction on the stack, as mentioned in section 4.3.5. This module can be found in Listing 5-25 and its utilization can be found in Listing 5-26.

```
module OneShot(
    input i_clk,                // Clock source
    input i_rst,                // Reset
    input i_signal,             // Input signal
    output o_one_shot           // One shot signal
    );

    // Register
    reg signal_dly;

    // Initial conditions
    initial begin
        signal_dly <= 1b0;
    end

    // One shot
    always @(posedge i_clk)
    begin
        if(i_rst == 1b1)
        begin
            signal_dly <= 1b0;
        end
        else begin
            signal_dly <= i_signal;
        end
    end

    // Assign the output
    assign o_one_shot = i_signal & ~signal_dly;

endmodule
```

Listing 5-25 - OneShot Module behavior

```
// Instantiate the One-Shot circuit to Timer 0 Interrupt
OneShot OneShot_TIM0_INT (
    .i_clk(i_clk),                          // Clock
    .i_rst(i_rst),                          // Reset
    .i_signal(tim0_int_in),                 // Input signal
    .o_one_shot(tim0_int_one_pulse)         // One shot signal
);

// Instantiate the One-Shot circuit to External 0 Interrupt
OneShot OneShot_EXT0_INT (
    .i_clk(i_clk),                          // Clock
    .i_rst(i_rst),                          // Reset
    .i_signal(ext0_int_in),                 // Input signal
```

```
    .o_one_shot(ext0_int_one_pulse)         // One shot signal
);

// Instantiate the One-Shot circuit to UART Interrupt
OneShot OneShot_UART_INT (
    .i_clk(i_clk),                          // Clock
    .i_rst(i_rst),                          // Reset
    .i_signal(uart_int_in),                 // Input signal
    .o_one_shot(uart_int_one_pulse)         // One shot signal
);
```

Listing 5-26 - OneShot in Datapath

Finally, to update the Instruction Register, the Program Counter and the Interruptions status, the following code blocks were developed.

```
always @(posedge i_clk)
begin
    if(i_rst == 1b1)
    begin
        IR <= 0;
    end
    else if(i_internal_cs == IR_PC_LOAD1 || i_internal_cs == IR_PC_LOAD2)
    begin
        IR <= rom_out;
    end
end
```

Listing 5-27 - Instruction Register

As can be seen in Listing 5-27, the Instruction Register, more known as IR, is updated, assuming the value from the ROM output, when the state is fetch1 or fetch2.

The Listing 5-28 represents the Interrupts manager.

```
always @(posedge i_clk)
begin
    if(i_rst == 1b1)
    begin
        ext0_int_req = 1b0;
        tim0_int_req = 1b0;
        uart_int_req = 1b0;
    end
    else begin
        if(ext0_int_status == 1b1) begin        // Occurred a valid interrupt in EXT0
            ext0_int_req = 1b1;
        end
        else if (ext0_int_in == 1b1) begin      // The EXT0 interrupt is already taken
            ext0_int_req = 1b0;
        end

        if(uart_int_status == 1b1) begin        // Occurred a valid interrupt in UART
            uart_int_req = 1b1;
        end

        else if (uart_int_in == 1b1) begin      // The UART interrupt is already taken
            uart_int_req = 1b0;
        end

        if(tim0_int_status == 1b1) begin        // Occurred a valid interrupt in TIM0
            tim0_int_req = 1b1;
        end

        else if (tim0_int_in == 1b1) begin      // The TIM0 interrupt is already taken
            tim0_int_req = 1b0;
        end
    end
end
```

Listing 5-28 - Interrupts Manager

As can be seen in Listing 5-28, if occurred a valid interrupt, the respective interrupt request is set to logic level high. Otherwise, it remains in the zero value.

Additionally, the easier way of introducing new interruptions in the system should be highlighted.

The Listing 5-29 represents how the Program Counter, more known as PC, is updated.

```
always @(posedge i_clk)
begin
    if(i_rst == 1b1)
    begin
        PC <= 0;
        ext0_int_on = 1b0;
        tim0_int_on = 1b0;
        uart_int_on = 1b0;
    end
    else
    begin
        case (i_internal_cs)
            IR_PC_LOAD1: begin
                // Priority #1
                if(ext0_int_req == 1b1) begin      // If the EXT0 interrupt is pending
                    PC <= ISR_EXT0_ADDR;           // Update the PC
                    ext0_int_in <= 1b1;            // Update the flag
                    PC_save <= PC;                 // Save the next instruction
                End
               // Priority #2
                else if(uart_int_req == 1b1) begin // If the UART interrupt is pending
                    PC <= ISR_UART_ADDR;           // Update the PC
                    uart_int_in <= 1b1;            // Update the flag
                    PC_save <= PC;                 // Save the next instruction
                end
                // Priority #3
                else if(tim0_int_req == 1b1) begin // If the TIM0 interrupt is pending
                    PC <= ISR_TIM0_ADDR;           // Update the PC
                    tim0_int_in <= 1b1;            // Update the flag
                    PC_save <= PC;                 // Save the next instruction
                end
                else begin
                    PC <= PC + 1;                  // Normal (Sequential) execution
                end
            end
            IR_PC_LOAD2: begin                     // 2nd Fetch (1 address machine)
                PC <= PC + 1;
            end
            JMP_C_LOAD: begin                      // JC rel
                if (psw[7] == 1b1)
                    PC <= o_ir;
            end
            JMP_NC_LOAD: begin                     // JNC rel
                if (psw[7] == 1b0)
                    PC <= o_ir;
            end
            JMP_Z_LOAD: begin                      // JZ rel
                if (acc == 8d0)
                    PC <= o_ir;
            end
            JMP_NZ_LOAD: begin                     // JNZ rel
                if (acc != 8d0)
                    PC <= o_ir;
            end
            RETI1_CS: begin
                PC <= {ram_out, 8d0};              // Extract from Stack the PC[15:8]
            end
            RETI2_CS: begin                        // Wait for RAM to output the SP
value
            end
            RETI3_CS: begin
                PC <= (PC | ram_out);              // Extract from Stack the PC[7:0]
                ext0_int_in <= 1b0;                // Clear the flag
                tim0_int_in <= 1b0;                // Clear the flag
                uart_int_in <= 1b0;                // Clear the flag
            end
        endcase
    end
end
```

Listing 5-29 - Program Counter

As can be seen in Listing 5-29, in fetch1 state, or where the internal control signal is *IR_PC_LOAD1*, the interrupts requests are checked. This verification takes place in this state, as this is the only way to guarantee that the normal flow of the program is correctly established after returning from the interrupt. Additionally, it is possible to check the priority level that, by default, each interrupt assumes in the system.

Finally, it is possible to visualize how the jumps are performed and how the interruption return process is performed in the PC view. In a more technical perspective, in one first phase, the Program Counter extracts the most significant eight-bit value from the RAM memory pointed by the stack pointer. Next, the Stack Pointer is incremented, and system waits one clock cycle for the value pointed by this value be stable. In the last step, the PC is completed with its least significant eight bits read from memory and the interrupt flags are cleared.

5.5 ALU Module

As already discussed in section 4.3.3, this module implements the Arithmetic and Logical Unit module. The interface provided by the ALU module is represented in Listing 5-30.

```
module ALU(
        input[ALU_CS_LEN-1:0] i_operation,
        input[7:0] i_src1,
        input[7:0] i_src2,
        input i_srcC,
        input i_srcAc,
        output reg [7:0] o_des1,
        output reg o_desC,
        output reg o_desAc,
        output reg o_desOv
        );
```

Listing 5-30 - ALU Module Inputs/Outputs

Next, it was necessary to perform some operations to prepare the additions and subtractions between the two sources. These operations are carried out through combinational logic and serve as a support for later, the flags of the PSW register to be properly updated. The Listing 5-31 and Listing 5-32 represents the aforementioned.

```
assign add1 = {1b0,i_src1[3:0]};
assign add2 = {1b0,i_src2[3:0]};
assign add3 = add1+add2;
assign addC = {4b0000,i_srcC};
assign add3C = add3 + addC;

assign add4 = {1b0,i_src1[6:4]};
assign add5 = {1b0,i_src2[6:4]};
assign add6 = {3b0,add3[4]};
assign add6C = {3b0,add3C[4]};
assign add7 = add4+add5+add6;
assign add7C = add4+add5+add6C;

assign add8 = {1b0,i_src1[7]};
assign add9 = {1b0,i_src2[7]};
assign add10 = {1b0,add7[3]};
assign add10C = {1b0,add7C[3]};
assign add11 = add8 + add9 + add10;
```

```
assign add11C = add8 + add9 + add10C;
```

Listing 5-31 - ALU ADD assigns

```
wire [4:0] sub1, sub2, sub3, sub4;
wire [3:0] sub5, sub6, sub7, sub8;
wire [1:0] sub9, suba, subb, subc;

assign sub1 = {1b1,i_src1[3:0]};
assign sub2 = {1b0,i_src2[3:0]};
assign sub3 = {4b0,i_srcC};
assign sub4 = sub1-sub2-sub3;

assign sub5 = {1b1,i_src1[6:4]};
assign sub6 = {1b0,i_src2[6:4]};
assign sub7 = {3b0, !sub4[4]};
assign sub8 = sub5-sub6-sub7;

assign sub9 = {1b1,i_src1[7]};
assign suba = {1b0,i_src2[7]};
assign subb = {1b0,!sub8[3]};
assign subc = sub9-suba-subb;
```

Listing 5-32 - ALU SUBB assigns

Finally, through the *i_operation* data field, it becomes possible to know which operation to perform and which flags should or should not be updated. The Listing 5-33 represents the ALU module behavior.

```
always @(*)
begin
    case(i_operation)

        ALU_CS_ADD: begin
            o_des1 = {add11[0],add7[2:0],add3[3:0]};
            o_desC = add11[1];
            o_desAc = add3[4];
            o_desOv = add11[1] ^ add7[3];
        end

        ALU_CS_ADDC: begin
            o_des1 = {add11C[0],add7C[2:0],add3C[3:0]};
            o_desC = add11C[1];
            o_desAc = add3C[4];
            o_desOv = add11C[1] ^ add7C[3];
        end

        ALU_CS_SUB: begin
            o_des1 = {subc[0],sub8[2:0],sub4[3:0]};
            o_desC = !subc[1];
            o_desAc = !sub4[4];
            o_desOv = !subc[1] ^ !sub8[3];
        end

        ALU_CS_AND: begin
            o_des1 = i_src1 & i_src2;
        end
        ALU_CS_XOR: begin
            o_des1 = i_src1 ^ i_src2;
        end

        ALU_CS_OR: begin
            o_des1 = i_src1 | i_src2;
        end
    endcase
end
```

Listing 5-33 - ALU behavior

As can be seen in Listing 5-33, it is possible to achieve a high level of modularity, making the process of adding new operations in the ALU module easier.

5.6 SFR Module

As already discussed in 4.3.4 section, this module is implemented as a register file and capable of writing and reading n SFR's in one only clock cycle, greatly increasing the system performance. The respectively module inputs and outputs can be visualized in Listing 5-34.

```
module SFRs(
    input i_clk,                    // Clock
    input i_rst,                    // Reset
    input [7:0] i_acc,              // ACC to write
    input [7:0] i_psw,              // PSW to write
    input [7:0] i_ie,               // IE to write
    input [7:0] i_sp,               // SP to write
    input [7:0] i_tmod,             // TMOD to write
    input [7:0] i_th0,              // TH0 to write
    input [7:0] i_tl0,              // TL0 to write
    input [7:0] i_tcon,             // TCON
    input [7:0] i_scon,             // SCON
    input [7:0] i_sbuf,             // SBUF
    input [7:0] i_p2,               // P2
    input [SFR_OP_LEN-1:0] i_op,    // Operation
    input i_parity,                 // Parity flag to feed the PSW
    output [7:0] o_acc,             // ACC
    output o_parity,                // Parity flag
    output [7:0] o_psw,             // PSW
    output [7:0] o_ie,              // IE
    output [7:0] o_sp,              // Stack Pointer
    output [7:0] o_tmod,            // TMOD
    output [7:0] o_th0,             // TH0
    output [7:0] o_tl0,             // TL0
    output [7:0] o_tcon,            // TCON
    output [7:0] o_scon,            // SCON
    output [7:0] o_sbuf,            // SBUF
    output [7:0] o_p2               // P2
    );
```

Listing 5-34 - SFR Module Inputs/Outputs

The module behavior can be visualized in Listing 5-35.

```
    // Instantiate the Accumulator
    ACC ACC(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_acc),
        .i_op(i_op),
        .o_acc(o_acc),
        .o_parity(o_parity)
    );

    // Instantiate the PSW
    PSW PSW(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_psw),
        .i_parity(i_parity),
        .i_op(i_op),
        .o_psw(o_psw)
    );

    // Instantiate the IE
    IE IE(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_ie),
        .i_op(i_op),
        .o_ie(o_ie)
    );

    // Instanciate the TMOD
    SP SP(
        .i_clk(i_clk),
        .i_rst(i_rst),
```

```
        .i_byte(i_sp),
        .i_op(i_op),
        .o_sp(o_sp)
    );

    // Instanciate the TMOD
    TMOD TMOD(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_tmod),
        .i_op(i_op),
        .o_tmod(o_tmod)
    );

    // Instanciate the TMOD
    TH0 TH0(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_th0),
        .i_op(i_op),
        .o_th0(o_th0)
    );

    // Instanciate the TMOD
    TL0 TL0(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_tl0),
        .i_op(i_op),
        .o_tl0(o_tl0)
    );

    // Instanciate the TCON
    TCON TCON(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_tcon),
        .i_op(i_op),
        .o_tcon(o_tcon)
    );

    // Instanciate the SCON
    SCON SCON(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_scon),
        .i_op(i_op),
        .o_scon(o_scon)
    );

    // Instanciate the SBUF
    SBUF SBUF(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_sbuf),
        .i_op(i_op),
        .o_sbuf(o_sbuf)
    );

    // Instanciate the P2
    P2 P2(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_byte(i_p2),
        .i_op(i_op),
        .o_p2(o_p2)
    );
```

Listing 5-35 - SFR Module

As can been seen in Listing 5-34 and Listing 5-35, this module is a literal translation of the scheme carried out in the design phase and which is shown in Figure 4-7. It is worth emphasizing once again how easy it is to add new SFR's to the system.

5.6.1 Bit addressable

As the 8051 has some memory locations and SFR's that are bit and non-bit addressable, the Listing 5-36 and Listing 5-37 shows one example of a Special Function Register which is bit-addressable. In this case, the SFR IE, responsible for activating or deactivating the interruptions in the system.

```
module IE(
    input i_clk,                    // Clock
    input i_rst,                    // Reset
    input [7:0] i_byte,             // Byte/Bit to write
    input [SFR_OP_LEN-1:0]i_op,     // Operation to do
    output [7:0] o_ie               // IE
    );
```

Listing 5-36 - IE Module Inputs/Outputs

```
always @(posedge i_clk)
begin
    if (i_rst == 1b1) begin
        ie <= 8d0;
    end
    else begin
        if (i_op & OP_IE_WR_BYTE) begin
            ie <= i_byte;
        end
        else if (i_op & OP_IE_WR_BIT) begin
            if (i_byte[0] == 1b1) begin
                ie[i_byte[3:1]] <= 1b1;
            end
            else begin
                ie[i_byte[3:1]] <= 1b0;
            end
        end
    end
end
```

Listing 5-37 - IE Module Behavior

Note that, if the operation type is a bit write operation, the bit value is specified in the first position (LSB) of the byte filed, and the address in position one to three of the same byte, thus saving resources. In this case, save two extra fields and make all the program's special registers follow the same protocol.

5.6.2 Non-bit addressable

In this case, the SFR TMOD, responsible for defining the timer operation modes, is shown as an example of one Special Function Register that is not bit-addressable. The Listing 5-38 and Listing 5-39 represents the respectively SFR implementation.

```
module TMOD(
    input i_clk,                    // Clock
    input i_rst,                    // Reset
    input [7:0] i_byte,             // Byte to write
    input [SFR_OP_LEN-1:0]i_op,     // Operation to do
    output [7:0] o_tmod             // TMOD
    );
```

Listing 5-38 - TMOD Module Inputs/Outputs

```
always @(posedge i_clk)
begin
    if (i_rst == 1b1) begin
        tmod <= 8d0;
    end
    else begin
        if (i_op & OP_TMOD_WR_BYTE) begin
```

```
            tmod <= i_byte;
        end
    end
end
```

Listing 5-39 - TMOD Module

It should be noted that regardless of whether the respective SFR is bit addressable or not, the interface with the SFR module is the same, having the same inputs and outputs.

5.7 Interrupt Module

As mentioned in topic 4.3.5, a generic module was created to handle microprocessor interrupts. The Listing 5-40 represents the module's interface, that is, it's inputs and outputs.

```
module Interrupt(
        input i_clk,            // Clock
        input i_rst,            // Reset
        input i_en,             // Interrupt enable
        input i_int,            // Interrupt in progress (No nested interrupts)
        input i_int_req,        // Interrupt request
        output o_int            // Interrupt status (1 if the interrupt is eligible to
execute)
);
```

Listing 5-40 - Interrupt Inputs/Outputs Module

As can been seen in Listing 5-40, this module has as inputs the clock and the reset source, an enable flag, which, later, will be connected to the enable bits of the special register IE (Figure 4-11) of the respective interrupt, and two other flags representing if the interrupt is already in progress and the interrupt request. As output, this module provides the status of the interrupt, saying whether it is eligible to be serviced.

In this context, it becomes necessary to have a mechanism that allows memorizing the last status of the interruption so that the transition is felt and detected. The Listing 5-41 represents this same mechanism, where for all clock pulses the last interrupt is memorized.

```
always @ (posedge i_clk)
begin
    if (i_rst == 1b1)
        int_req_last <= 1b0;           // Reset
    else
        int_req_last <= int_pend;       // Memorize
end
```

Listing 5-41 - Memorize last Interrupt status

Finally, if there is a positive transition on the interrupt pin, if it is enabled and if the is not executing at the moment, the interrupt is eligible to run. The code that implements this functionality is represented in the Listing 5-42.

```
always @ (posedge i_clk)
    begin
        if (i_rst == 1b1)
            int_pend <= 1b0;
        else if (i_int_req == 1b1 && int_req_last == 1b0 && i_en == 1b1 && i_int==1b0)
```

```
            int_pend <= 1'b1;
        else begin
            int_pend <= 1'b0;
        end
    end
```

Listing 5-42 - Update the Interrupt status

5.8 Timer Module

As mentioned in section 4.4.1, a timer with all four modes was implemented. The Listing 5-43 represents the respectively interface, where can be visualized it's inputs and outputs.

```
module Timer(
    input i_clk,            // Clock source
    input i_rst,            // Reset
    input [7:0] i_tmod,     // Modes
    input [7:0] i_thx,      // THx received by THx SFR
    input [7:0] i_tlx,      // TLx received by TLx SFR
    input i_trx,            // TRx received by (TCON[4] if x = 0 or TCON[6] if x = 1)
    input i_intx,           // INTx Received by (IE[1] if x = 0 or IE[3] if x = 0)
    output [7:0] o_thx,     // THx updated
    output [7:0] o_tlx,     // TLx updated
    output o_tfx,           // Overflow flag (TCON[5] if x = 0 or TCON[7] if x = 1)
    output o_en             // Timer/Counter is enable
    );
```

Listing 5-43 - Timer Inputs/Outputs

As can been seen in Listing 5-43, the module follows exactly the designed in Design phase, more specifically in Figure 4-12.

Next, it becomes necessary to define the enable flag. This flag must be active when:

1. If Gate = 1, timer x will be active only when TRx = 1 and INTx = 1
2. If Gate = 0, timer x will be active when TRx = 1 (Software control)

Where x is means 0 (Timer 0) or 1 (Timer 1), because the module can implement both of them. The implementation is represented in Listing 5-44.

```
assign o_en = (tmod[GATE] == 1'b1 && i_trx == 1'b1 && i_intx) ? 1'b1 :
              (tmod[GATE] == 1'b0 && i_trx == 1'b1) ? 1'b1 : 1'b0;
```

Listing 5-44 - Timer enable flag

Then, in order for the data to be all synchronized, the code represented in Listing 5-45 was implemented.

```
if (i_thx != thx) begin
    thx <= i_thx;
end
if (i_tlx != tlx) begin
    tlx <= i_tlx;
end
if (i_tmod != tmod) begin
    tmod <= i_tmod;
end
```

Listing 5-45 - Timer update internal registers

Finally, to implement the four timer modes, the code represented in Listing 5-46 was developed.

```
case (i_tmod[1:0])
    TIMER_MODE_0: begin
        {tfx, thx, tlx[4:0]} <= {1'b0, thx, tlx[4:0]} + 1'b1;
    end
    TIMER_MODE_1: begin
        {tfx, thx, tlx} <= {1'b0, thx, tlx} + 1'b1;
    end
    TIMER_MODE_2: begin
        if (tlx == 8'b1111_1111) begin
            tfx <= 1'b1;
            tlx <= thx;
        end
        else begin
            tlx <= tlx + 8'h1;
            tfx <= 1'b0;
        end
    end
    TIMER_MODE_3: begin
        {tfx, tlx} <= {1'b0, tlx} + 1'b1;
    end
    default: begin
        tmod <= 0;
        thx <= 0;
        tlx <= 0;
        tfx <= 0;
    end
endcase
```

Listing 5-46 - Timer modes implementation

For example, and as can be seen in Listing 5-46, if the timer was programmed to execute in the eight-bits with auto reload, or mode number two, if the overflow occurs, the overflow flag is updated and the TL0 register 0 is reloaded with the content specified in TH0. If not, the counter variable, TL0, is incremented and the overflow flag is maintained at logical level zero.

5.9 UART Module

In this section, the Figure 4-15 was translated into Verilog code. Listing 5-47 represents the UART module implementation.

```
module UART(
    input i_clk,                        // Clock source
    input i_rst,                        // Reset
    input i_rx_serial,                  // RX line
    input i_rx_en,                      // RX enable flag
    input [7:0] i_tx_data,              // TX data to transmit
    input i_tx_en,                      // TX enable flag
    output o_rx_done,                   // RX done flag
    output [7:0] o_rx_data,             // RX data received
    output o_tx_done,                   // TX done flag
    output o_tx_serial                  // TX line
);

    // Instantiate the UART Receiver
    UART_RX UART_RX(
        .i_clk(i_clk),                  // Clock source
        .i_rst(i_rst),                  // Reset
        .i_rx_serial(i_rx_serial),      // Receive bit (RX line)
        .i_en(i_rx_en),                 // Enable reception
        .o_complete(o_rx_done),         // Complete flag
        .o_data(o_rx_data)              // Output data
```

```
    );

    // Instantiate the UART Transmitter
    UART_TX UART_TX(
        .i_clk(i_clk),                  // Clock source
        .i_rst(i_rst),                  // Reset
        .i_byte(i_tx_data),             // Byte to transmit
        .i_tx_en(i_tx_en),              // TX Enable
        .o_complete(o_tx_done),         // TX Complete flag
        .o_tx_serial(o_tx_serial)       // Tx line
    );

endmodule
```

Listing 5-47 - UART Module

As can been seen in Listing 5-47, this module is responsible implement the Universal Asynchronous Receiver-Transmitter communication protocol. Also, it is possible to see that this module instantiates the UART receive and transmission module.

The module that implements the reception of characters through the serial port is represented in Listing 5-48.

```
module UART_RX(
    input i_clk,            // Clock source
    input i_rst,            // Reset
    input i_rx_serial,      // Receive bit (RX line)
    input i_en,             // Enable bit
    output o_complete,      // Complete flag
    output [7:0] o_data     // Output data
    );

    // States definition
    parameter s_idle = 3b000;
    parameter s_start_bit = 3b001;
    parameter s_data_bits = 3b010;
    parameter s_stop_bit = 3b011;
    parameter s_cleanup = 3b100;

    // Clocks per bit
    // CLKS_PER_BIT = CLOCK_FREQ / UART_FREQ
    // In this case, if the CLOCK_FREQ = 115MHz and UART_FREQ = 115200, we have:
    // ==> CLKS_PER_BIT = 125000000 / 115200 = 1085
    parameter CLKS_PER_BIT = 1085;

    // Variables
    reg [1:0] rx_data;              // Bit received
    reg [2:0] state;                // State
    reg [10:0] clock_count;         // Clock count
    reg [2:0] bit_index;            // Current bit index
    reg [7:0] byte;                 // Byte received
    reg complete;                   // Complete reception flag


    // Initial conditions
    initial begin
        state <= s_idle;
        clock_count <= 11d0;
        bit_index <= 3d0;
        byte <= 8d0;
        complete <= 1b0;
        rx_data <= 2b11;
    end

    // RX State Machine
    always @(posedge i_clk)
    begin
        if (i_rst == 1b1) begin
            state <= s_idle;
            clock_count <= 11d0;
            bit_index <= 3d0;
            byte <= 8d0;
            complete <= 1d0;
            rx_data <= 2b11;
```

```
        end
        else begin
            rx_data <= {rx_data[0], i_rx_serial};
            case (state)
                s_idle: begin
                    complete <= 1'd0;
                    clock_count <= 11'd0;
                    bit_index <= 3'd0;
                    if (rx_data[1] == 1'b0 && i_en == 1'b1) begin
                        state <= s_start_bit;
                    end
                    else begin
                        state <= s_idle;
                    end
                end
                s_start_bit: begin
                    if (clock_count == (CLKS_PER_BIT-1)/2)
                    begin
                        if (rx_data == 1'b0) begin
                            clock_count <= 11'd0;
                            state <= s_data_bits;
                        end
                        else begin
                            state <= s_idle;
                        end
                    end
                    else begin
                        clock_count <= clock_count + 1;
                        state <= s_start_bit;
                    end
                end
                s_data_bits: begin
                    if (clock_count < (CLKS_PER_BIT-1)) begin
                        clock_count <= clock_count + 1;
                        state <= s_data_bits;
                    end
                    else begin
                        clock_count <= 11'd0;
                        byte[bit_index] <= rx_data;

                        if (bit_index < 7) begin
                            bit_index <= bit_index + 1;
                            state <= s_data_bits;
                        end
                        else begin
                            bit_index <= 3'd0;
                            state <= s_stop_bit;
                        end
                    end
                end
                s_stop_bit: begin
                    if (clock_count < (CLKS_PER_BIT-1)) begin
                        clock_count <= clock_count + 1;
                        state <= s_stop_bit;
                    end
                    else begin
                        complete <= 1'b1;
                        clock_count <= 11'd0;
                        state <= s_cleanup;
                    end
                end
                s_cleanup: begin
                    state <= s_idle;
                    complete <= 1'b0;
                end
                default: begin
                    state <= s_idle;
                end
            endcase
        end
    end

    // Assign the outputs
    assign o_complete = complete;
    assign o_data = byte;


endmodule
```

Listing 5-48 -UART Reception Module

As can been seen in Listing 5-48, the state machine developed in the Design phase, Figure 4-18, was translated into Verilog code. An important parameter that had to be calculated was the amount of time needed to acquire the bits from the data receiving line. In this case, knowing the clock frequency and the baud rate, just divide these two frequencies and you get the number of clock jumps needed to get something from the line. Note that, in this version, a baud rate of 115200 bits per second was assumed. However, it could be defined by a register and passed as a parameter to the respective module, not being implemented since it would not add complexity to the system.

Regarding to the transmission, the Listing 5-49 shows the UART transmission implementation.

```
module UART_TX(
    input i_clk,            // Clock source
    input i_rst,            // Reset
    input [7:0] i_byte,     // Byte to transmit
    input i_tx_en,          // Enable
    output o_complete,      // Complete flag
    output o_tx_serial      // Tx line
    );

    // States definition
    parameter s_idle = 3b000;
    parameter s_start_bit = 3b001;
    parameter s_data_bits = 3b010;
    parameter s_stop_bit = 3b011;
    parameter s_cleanup = 3b100;
    parameter CLKS_PER_BIT = 1085;

    // Variables
    reg [2:0] state;                // State
    reg [10:0] clock_count;         // Clock count
    reg [2:0] bit_index;            // Current bit index
    reg [7:0] tx_data;              // Byte to send
    reg complete;                   // Complete reception flag
    reg tx_serial;                  // TX line

    // Initial conditions
    initial begin
        state <= s_idle;
        clock_count <= 11d0;
        bit_index <= 3d0;
        tx_data <= 8d0;
        tx_serial <= 1b0;
        complete <= 1b0;
    end

    // TX State Machine
    always @(posedge i_clk)
    begin
        if (i_rst == 1b1) begin
            state <= s_idle;
            clock_count <= 11d0;
            bit_index <= 3d0;
            tx_data <= 8d0;
            complete <= 1b0;
            tx_serial <= 1b0;
        end
        else begin
            case (state)
                s_idle: begin
                    tx_serial <= 1b1;
                    complete <= 1d0;
                    clock_count <= 11d0;
                    bit_index <= 3d0;
                    if (i_tx_en == 1b1) begin
                        tx_data <= i_byte;
                        state <= s_start_bit;
                    end
                    else begin
```

```
                        state <= s_idle;
                    end
                end
                s_start_bit: begin
                    tx_serial <= 1'b0;
                    if (clock_count < (CLKS_PER_BIT-1))
                    begin
                        clock_count <= clock_count + 1;
                        state <= s_start_bit;
                    end
                    else begin
                        clock_count <= 11'd0;
                        state <= s_data_bits;
                    end
                end
                s_data_bits: begin
                    tx_serial <= tx_data[bit_index];

                    if (clock_count < (CLKS_PER_BIT-1)) begin
                        clock_count <= clock_count + 1;
                        state <= s_data_bits;
                    end
                    else begin
                        clock_count <= 11'd0;

                        if (bit_index < 7) begin
                            bit_index <= bit_index + 1;
                            state <= s_data_bits;
                        end
                        else begin
                            bit_index <= 3'd0;
                            state <= s_stop_bit;
                        end
                    end
                end
                s_stop_bit: begin
                    tx_serial <= 1'b1;
                    if (clock_count < (CLKS_PER_BIT-1)) begin
                        clock_count <= clock_count + 1;
                        state <= s_stop_bit;
                    end
                    else begin
                        complete <= 1'b1;
                        clock_count <= 11'd0;
                        state <= s_cleanup;
                    end
                end
                s_cleanup: begin
                    state <= s_idle;
                    complete <= 1'b0;
                end
                default: begin
                    state <= s_idle;
                end
            endcase
        end
    end

    // Assign the outputs
    assign o_complete = complete;
    assign o_tx_serial = tx_serial;


endmodule
```

Listing 5-49 - UART Transmission Module

Similar to the reception, the state machine of them Figure 4-17 was translated to Verilog code.

5.10 SPI Module

In this section, the SPI implementation will be presented. In a more technical perspective, the designed in 4.4.3 was translated to Verilog code and the result is represented in Listing 5-50.

```
module SPI_Slave(
    input i_scl,         // Master Clock (e.g. STM32)
    input i_clk,         // FPGA Clock
    input i_rst,         // Reset
    input i_mosi,        // Master Out Slave In (data output from master)
    input i_cs,          // Chip/Slave Select
    output o_miso,       // Master In Slave Out (data output from slave)
    output[7:0] o_data,  // Output data
    output o_sync        // Output enable synchronized
    );

    // Data
    reg [7:0] data;

    // Assign the outputs
    assign o_miso = data[7];    // Data output from slave
    assign o_data = data;       // Data

    // Initial Conditions
    initial begin
        data <= 8d0;
    end

    // Instanciate the CDC (Clock Domain Crossing module)
    CDC CDC(
        .i_clk(i_clk),          // Clock source
        .i_rst(i_rst),          // Reset
        .i_signal(i_cs),        // Input signal (not stable)
        .o_signal(o_sync)       // Output signal (stable one)
    );

    // SPI State Machine
    always @(posedge i_scl or posedge i_rst)
    begin
        if (i_rst == 1b1) begin
            data <= 8h0;
        end
        else if (~i_cs) begin
            data <= {data[6:0], i_mosi};     // Left shift
        end
    end

endmodule
```

Listing 5-50 - SPI Implementation

Additionally, to deal with the Metastability problems caused by having two clock sources in the system, the module CDC, or Clock Domain Crossing, was developed. To better understand this problem please consult the section 4.4.3. The code developed is represented in Listing 5-51.

```
module CDC(
        input i_clk,         // Clock source
        input i_rst,         // Reset
        input i_signal,      // Input signal (not stable)
        output o_signal      // Output signal (stable one)
    );

    // Clock Domain Crossing implemented with 2 Flip-Flops
    reg [1:0] cdc;

    // Assign the stable signal (second Flip-Flop)
    assign o_signal = cdc[1];

    // Initial Conditions
    initial begin
```

```
        cdc <= 2d0;
    end

    // Clock Domain Crossing
    always @(posedge i_clk or posedge i_rst)
    begin
        if (i_rst) begin
            cdc <= 2b00;
        end
        else begin
            cdc <= {cdc[0], i_signal};
        end
    end

endmodule
```

Listing 5-51 - CDC Implementation

5.11 Prescaler and Debounce Module

Additionally, to deal with the problem of bouncing real buttons, two modules were created, Prescaler and Debounce, which aim, respectively, to slow down the clock signal, to help the debounce technique, and debounce that same button.

The implementation of the modules mentioned above can be found in the Listing 5-52 and Listing 5-53.

```
module Prescaler(
    input i_clk,        // Higher Clock source
    input i_rst,        // Reset
    output o_en         // Slower Clock enable
    );


    reg[32:0] counter = 32d0;
    parameter DIVISOR = 32d25000000; // o_em -> 5 Hz

    initial begin
        counter <= 32d0;
    end

    always @(posedge i_clk)
    begin
     counter <= counter + 32d1;
     if(counter>=(DIVISOR-1))
        counter <= 32d0;
    end

    assign o_en = (counter < DIVISOR/2) ? 1b1 : 1b0;
endmodule
```

Listing 5-52 - Prescaler Module Implementation

```
module Debounce(
    input i_clk,            // Clock
    input i_rst,            // Reset
    input i_signal,         // Signal unstable
    output o_signal         // Signal stable
    );

    // Variables
    wire slow_clk_en;
    wire os;
    parameter DEBOUNCE_RESOLUTION = 2d3;
    reg [DEBOUNCE_RESOLUTION-1:0] Q;
    reg out_signal;
    reg prev_out;
    integer k;

    // Initial Conditions
    initial begin
```

```
        Q <= 3'd0;
    end

    // Instantiate the Slow Clock module to slow the clock
    Prescaler Prescaler(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .o_en(slow_clk_en)
    );

    // Shift the bits along the array
    always @(posedge slow_clk_en)
    begin
        if (i_rst) begin
            Q <= 3'd0;
        end
        else begin
            Q <= {Q[1],Q[0], i_signal};
        end
    end

    // One shot
    assign os = Q[0] & Q[1] & ~Q[2];

    // Adjust one shot to CPU clock domain
    always @(posedge i_clk)
    begin
        if (os == 1'b1 && prev_out == 1'b0) begin
            out_signal = 1'd1;
            prev_out = 1'b1;
        end
        else if (prev_out == 1'b1) begin
            out_signal = 1'b0;
        end
        if (os == 1'b0) begin
            prev_out = 1'b0;
        end
    end

    // Assign the output signal (Stable)
    assign o_signal = out_signal;

endmodule
```

Listing 5-53 - Debounce Module Implementation

6 Simulations

In this chapter, some simulations will be presented and duly documented.

6.1 Arithmetic Instructions

The Figure 6-1 represents the simulation of addition operations.

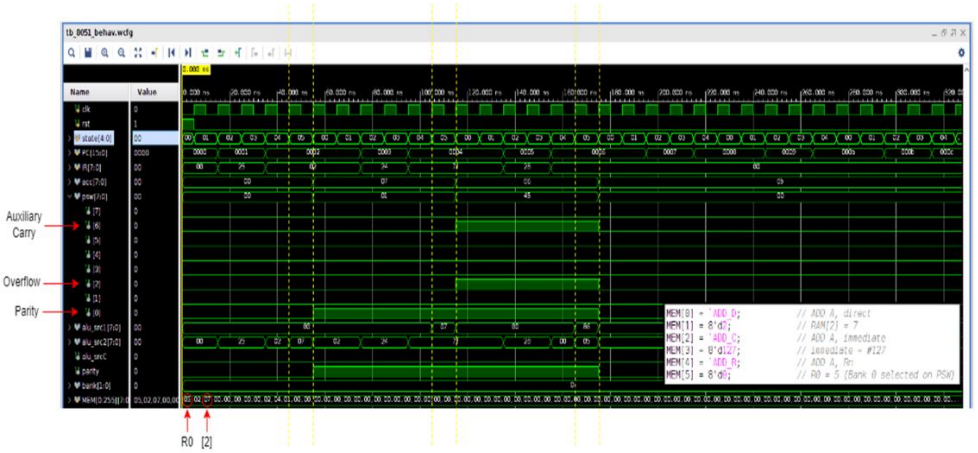

Figure 6-1 - Arithmetic Instructions Simulation (1)

In this case, it is possible to see that the first instruction is the addition between the accumulator and the direct, more specifically the direct value in the RAM position 2. After the execution, the accumulator value changes from zero to seven because the initial accumulator value is zero and the direct value is seven. Next, the operation performed is an ADD_C, that means that the accumulator value will be summed with the content specified in the second operand, more specifically 127. In this case, the accumulator value is correctly updated. Finally, the last operation will sum the accumulator to a register. As the bank selected is zero and the register selected is R0, the accumulator value will be summed with the value on position number zero in RAM. Note that in each execution the PSW is correctly update, accordantly to the operation.

The Figure 6-2 represents the simulation others arithmetic operations.

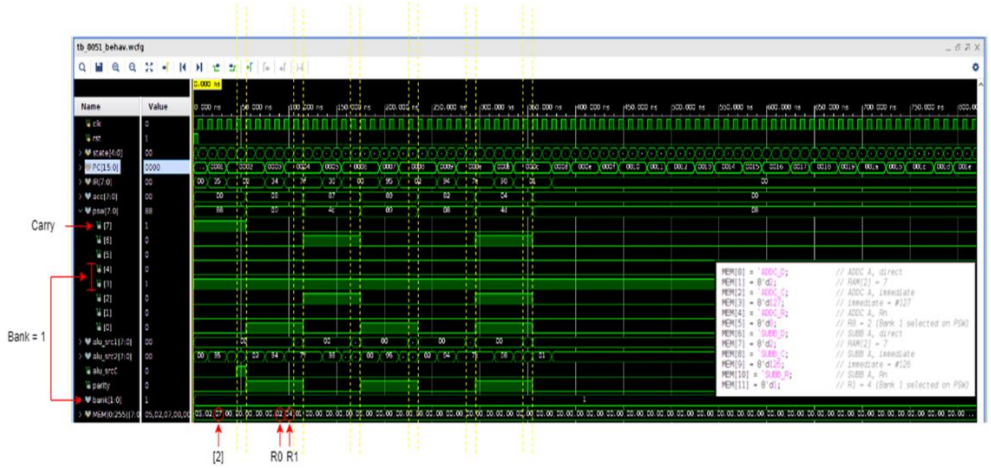

Figure 6-2 - Arithmetic Instructions Simulation (2)

As can been seen in Figure 6-2, firstly the accumulator has the initial value zero and will be summed with the direct in position number two in RAM, that initial is seven. Because the carry is active and the type of operation is a add with carry, the final accumulator value is eight. Next, more operations are correctly performed, highlighting that the bank selected is now the bank one and the registers are found according to it.

6.2 Logical Instructions

The Figure 6-3 represents the logical instructions simulation.

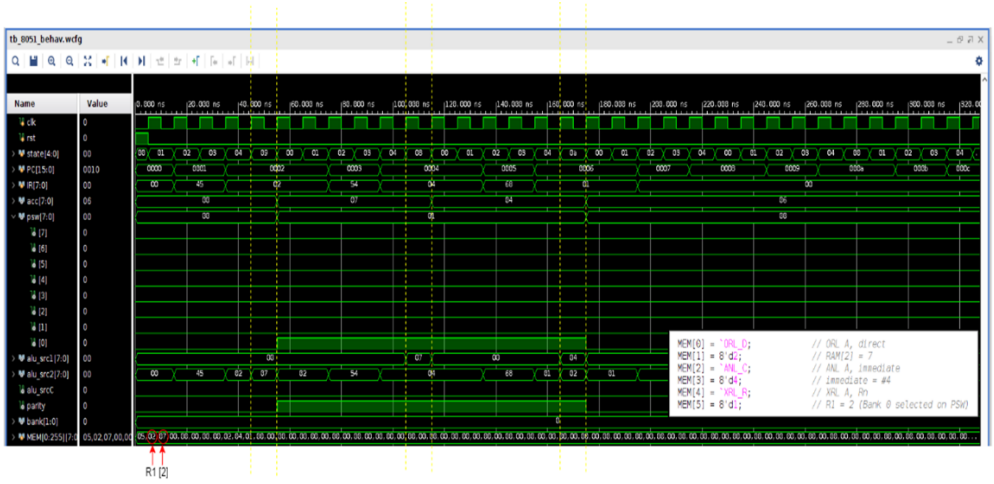

Figure 6-3 - Logical Instructions Simulation

As can been seen in Figure 6-3, all operations were correctly performed.

6.3 Data Transfer Instructions

The Figure 6-4 represents the data transfer instructions simulation, more precisely the process of transfer the contents of a register, direct or constant to the accumulator.

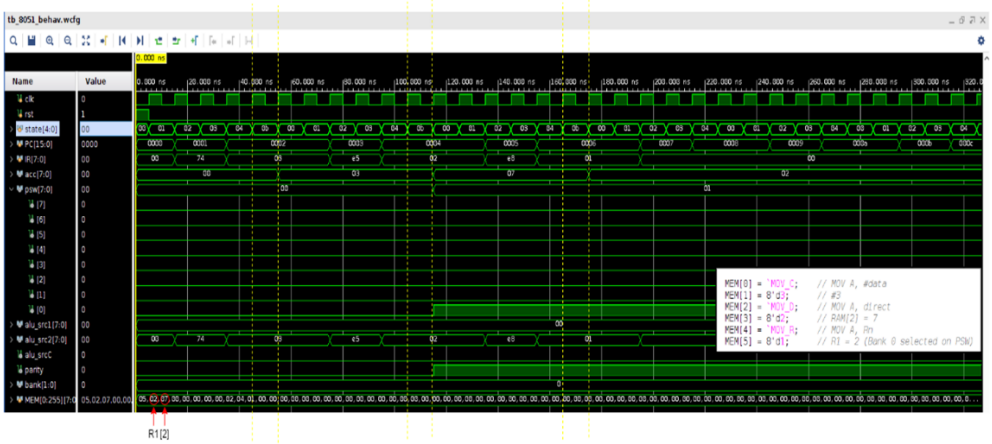

Figure 6-4 - Data Transfer Instruction Simulation (1)

As can been seen in Figure 6-4, the first operation is responsible to move the constant three to the accumulator. After the execution, the accumulator is updated with the number three. Therefore, the next operation is a MOV_D, that means transfer the value from the RAM direct specified in the second byte to the accumulator. As the RAM in position umber two is seven, the accumulator is updated with this same number. Finally, the last operation is responsible to move the value from R1 to the accumulator. This test was correctly done, where the final accumulator value assumes the value two, the same value as R1.

The Figure 6-5 represents the data transfer instructions simulation, more precisely the process of transfer the accumulator content, to a register or a direct.

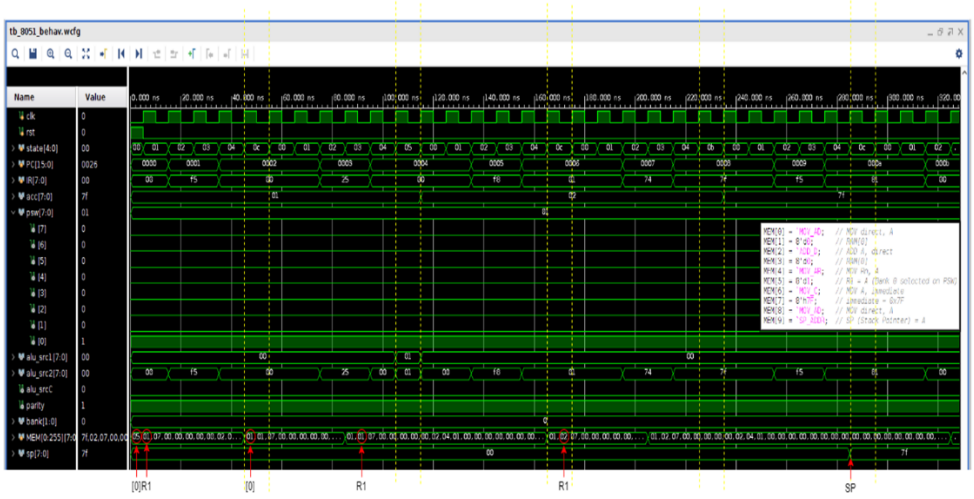

Figure 6-5 - Data Transfer Instruction Simulation (2)

As can been seen in Figure 6-5, the first operation is responsible to move the accumulator value, that initial is one, to the RAM in position one. After the execution, the RAM is updated where can been seen the position one changes from five (initial value) to one. Therefore, the operation responsible to move the accumulator value to a register is correctly performed. Finally, it was tested the process of move the accumulator value (0x7F in this case) to a specific special functional register. In this case, the operation was successfully performed, where the stack pointer (SP) changed from zero to 0x7F.

6.4 Jump Instructions

The Figure 6-6 represents the jump instructions simulation.

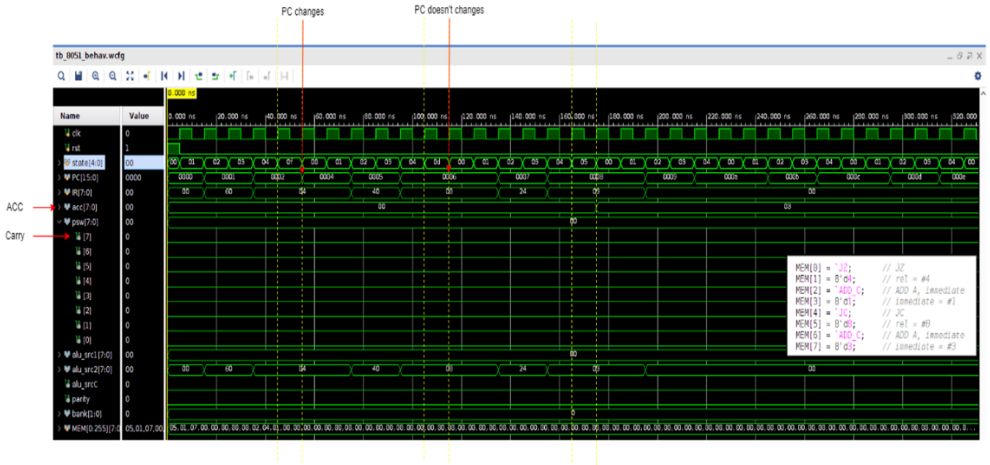

Figure 6-6 - Jump Instructions Simulation

In this case, it is possible see that the initial accumulator value is zero and when the jump zero instruction is performed, the program jumps to the address specified in the second byte. After that, the next instruction is a jump carry and this one, unlike the other, is not performed, because the carry flag is set to zero. In this case, the program continues his normal execution and executes the next instruction, that is a add operation.

6.5 Timer

The Figure 6-7 represents the timer simulation.

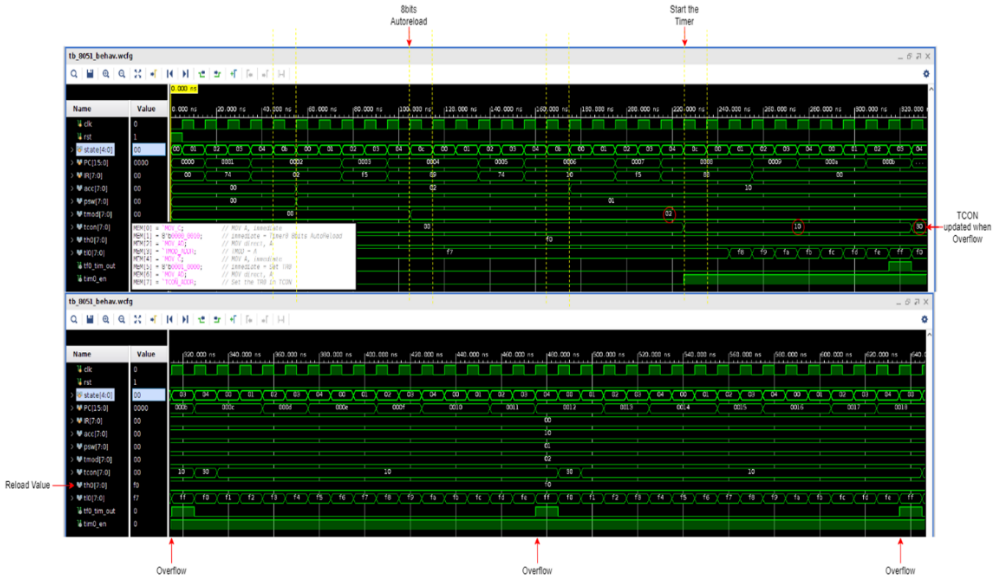

Figure 6-7 - Timer Simulation

As can been seen in Figure 6-7, the program consists of programming the timer zero as eight bits with auto reload and start the timer. In this case, after executed these two operations, it is possible so see the special functional register, TMOD and TCON, changing these values accordingly. Next, the timer starts to count and when occurred an overflow, this same overflow is felt in the system and the TCON register is accordingly updated. Also, as the timer was programed as eight bits with auto reload, after the overflow the TL0 loads the value form the TH0, as expected.

6.6 Timer with Interrupts

The Figure 6-8 represents the timer interrupt simulation.

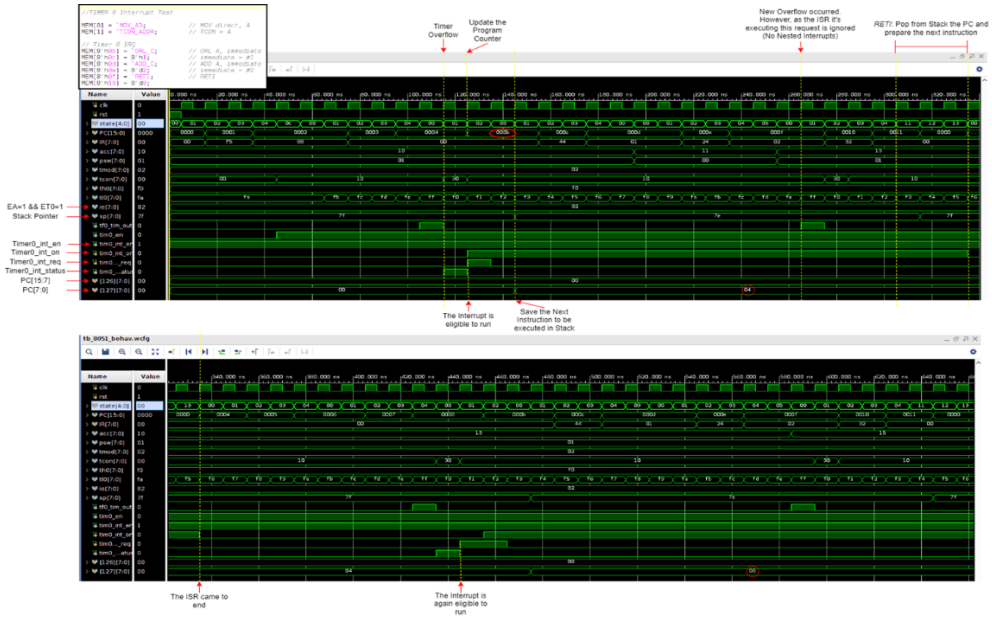

Figure 6-8 - Timer with Interrupt Simulation

In this case, after the timer be programmed as eight bits with auto reload and the interrupts enable flags, EA and ET0, putting into logic level one, when an overflow occurs this same event is felt and the interrupt is, in the first moment, eligible to run. Next, the next instruction to be executed is saved in stack and the program counter is updated with the Timer 0 Interrupt Service Routine address (0x0B). Therefore, the two operations inside the ISR are correctly performed and when the RETI is called, the system pops from the stack the program counter and prepares the next instruction. In this context, the ISR came to end, and the system continues with its normal execution.

One important note is that when the timer overflows where the program is still executing the ISR, this same request is ignored (no nested interrupts).

7 Results

In this chapter some of the practical tests developed to test the correct functioning of the 8051 will be presented. The demonstration videos can be accessed ate the following link:

https://drive.google.com/drive/folders/1hcch4SVxVSkeRxfMuNn13xXOcp-1fp69?usp=share_link

7.1 Control the LEDs from UART

This test is responsible for demonstrating the correct functioning of the UART. As can be seen in the respective video, the user is able to send a character through the serial port and that same character is received at the FPGA and displayed on the LEDS.

7.2 Seconds counter with Interrupt and LEDs

This test is responsible for demonstrating the correct operation of the timer with interruption. To this end, a simple program was created that allowed the timer to overflow every one second and its interruption active. As can be seen in the respective video, every second the timer overflows, the interrupt is invoked, and it is shown on the LEDS.

7.3 Receive a byte from UART, display on LEDs and send it back through UART (all with Interrupts)

In this section a big test was made where the authors went to Keil uVision 5 IDE and wrote a simple program that basically allows the system to receive a byte from UART, display this same value on FPGA LEDs and send it back though UART, all this with interrupts. The code developed can be consulted in Figure 7-1.

```
#include <REG51F380.H>

CSEG AT 0H
MAIN:
    MOV A, #10010000B   // MOV A, immediate
    MOV IE, A           // Set EA and ES0
    MOV A, #00010000B   // MOV A, immediate
    MOV SCON0, A        // Start Reception
    JNC $               // Loop

CSEG AT 23H
ISR_UART:
    MOV A, #1           // MOV A, immediate
    ANL A, SCON0        // ANL A, SCON0
    JNZ RECV_DATA       // Jump if receive data
    MOV A, #00010000B   // MOV A, immediate
    MOV SCON0, A        // Clear UART Transmission
    RETI                // Return

CSEG AT 50H
RECV_DATA:
    MOV A, SBUF0        // MOV A, direct
    MOV P2, A           // MOV P2, A
    MOV A, #00011000B   // MOV A, immediate
    MOV SCON0, A        // Clear UART Reception
    RETI                // Return

END
```

Figure 7-1 - Keil uVision 5 Assembly Program

Next, the code was compiled, and the hexadecimal code was accessed at */ProjectName/Objects/ProjectName.hex*. The acquired content is represented in Figure 7-2.

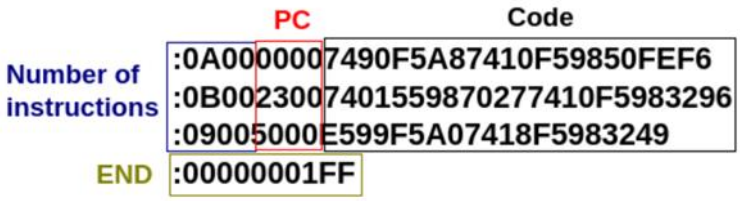

Figure 7-2 - Hexadecimal code generated by Keil's compiler

In this case, it is possible to see that some fields are also generated. For example, the field responsible to identify the number of instructions per line, the program counter value of each line and the end directive. However, if compared the code segment to the assembly code written in Keil, it is possible to affirm that the code is identically. As a way of example, the 74 indicates the instruction of move a constant to the accumulator and the value 90 specifies the constant value.

After that, this block of code was introduced in the FPGA and the result can been seen in the respective video. The test was a success, where the user can send through UART one number, this same number is shown on the FPGA LED's and send it back through UART to the host machine. All this process implemented with Interrupts.

7.4 STM32 – Zybo communication via SPI

This test is responsible for demonstrating the correct communication via SPI from the STM32 and the Zybo FPGA. As can been seen in the respective video, the user can send through SPI a value from the STM32 to the FGPA, where this same value is shown on LED's.

7.5 External Interrupt through Push Button

This test represents the external interrupt through a push button. As can been seen in the respective video, the user can press the button, where this event is felt and the LED transitions to logic level one. However, it should be noted that a prescaler mechanism was also implemented, to lower the clock frequency, and a debounce mechanism to overcome the bounce characteristic of physical buttons.

Appendix A

Hex	Bytes	Mnemonic	Operands
00	1	NOP	
01	2	AJMP	addr11
02	3	LJMP	addr16
03	1	RR	A
04	1	INC	A
05	2	INC	direct
06	1	INC	@R0
07	1	INC	@R1
08	1	INC	R0
09	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	bit. offset
11	2	ACALL	addr11
12	3	LCALL	addr16
13	1	RRC	A
14	1	DEC	A
15	2	DEC	direct
16	1	DEC	@R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7
20	3	JB	bit. offset
21	2	AJMP	addr11
22	1	RET	
23	1	RL	A
24	2	ADD	A. #immed
25	2	ADD	A. direct
26	1	ADD	A. @R0
27	1	ADD	A. @R1
28	1	ADD	A. R0
29	1	ADD	A. R1
2A	1	ADD	A. R2
2B	1	ADD	A. R3
2C	1	ADD	A. R4
2D	1	ADD	A. R5
2E	1	ADD	A. R6
2F	1	ADD	A. R7
30	3	JNB	bit. offset
31	2	ACALL	addr11
32	1	RETI	
33	1	RLC	A
34	2	ADDC	A. #immed
35	2	ADDC	A. direct
36	1	ADDC	A. @R0
37	1	ADDC	A. @R1
38	1	ADDC	A. R0
39	1	ADDC	A. R1
3A	1	ADDC	A. R2
3B	1	ADDC	A. R3

3C	1	ADDC	A, R4
3D	1	ADDC	A, R5
3E	1	ADDC	A, R6
3F	1	ADDC	A, R7
40	2	JC	offset
41	2	AJMP	addr11
42	2	ORL	direct, A
43	3	ORL	direct, #immed
44	2	ORL	A, #immed
45	2	ORL	A, direct
46	1	ORL	A, @R0
47	1	ORL	A, @R1
48	1	ORL	A, R0
49	1	ORL	A, R1
4A	1	ORL	A, R2
4B	1	ORL	A, R3
4C	1	ORL	A, R4
4D	1	ORL	A, R5
4E	1	ORL	A, R6
4F	1	ORL	A, R7
50	2	JNC	offset
51	2	ACALL	addr11
52	2	ANL	direct, A
53	3	ANL	direct, #immed
54	2	ANL	A, #immed
55	2	ANL	A, direct
56	1	ANL	A, @R0
57	1	ANL	A, @R1
58	1	ANL	A, R0
59	1	ANL	A, R1
5A	1	ANL	A, R2
5B	1	ANL	A, R3
5C	1	ANL	A, R4
5D	1	ANL	A, R5
5E	1	ANL	A, R6
5F	1	ANL	A, R7
60	2	JZ	offset
61	2	AJMP	addr11
62	2	XRL	direct, A
63	3	XRL	direct, #immed
64	2	XRL	A, #immed
65	2	XRL	A, direct
66	1	XRL	A, @R0
67	1	XRL	A, @R1
68	1	XRL	A, R0
69	1	XRL	A, R1
6A	1	XRL	A, R2
6B	1	XRL	A, R3
6C	1	XRL	A, R4
6D	1	XRL	A, R5
6E	1	XRL	A, R6
6F	1	XRL	A, R7
70	2	JNZ	offset
71	2	ACALL	addr11
72	2	ORL	C, bit
73	1	JMP	@A+DPTR
74	2	MOV	A, #immed
75	3	MOV	direct, #immed
76	2	MOV	@R0, #immed
77	2	MOV	@R1, #immed
78	2	MOV	R0, #immed
79	2	MOV	R1, #immed
7A	2	MOV	R2, #immed
7B	2	MOV	R3, #immed

7C	2	MOV	R4, #immed	
7D	2	MOV	R5, #immed	
7E	2	MOV	R6, #immed	
7F	2	MOV	R7, #immed	
80	2	SJMP	offset	
81	2	AJMP	addr11	
82	2	ANL	C, bit	
83	1	MOVC	A, @A+PC	
84	1	DIV	AB	
85	3	MOV	direct, direct	
86	2	MOV	direct, @R0	
87	2	MOV	direct, @R1	
88	2	MOV	direct, R0	
89	2	MOV	direct, R1	
8A	2	MOV	direct, R2	
8B	2	MOV	direct, R3	
8C	2	MOV	direct, R4	
8D	2	MOV	direct, R5	
8E	2	MOV	direct, R6	
8F	2	MOV	direct, R7	
90	3	MOV	DPTR, #immed	
91	2	ACALL	addr11	
92	2	MOV	bit, C	
93	1	MOVC	A, @A+DPTR	
94	2	SUBB	A, #immed	
95	2	SUBB	A, direct	
96	1	SUBB	A, @R0	
97	1	SUBB	A, @R1	
98	1	SUBB	A, R0	
99	1	SUBB	A, R1	
9A	1	SUBB	A, R2	
9B	1	SUBB	A, R3	
9C	1	SUBB	A, R4	
9D	1	SUBB	A, R5	
9E	1	SUBB	A, R6	
9F	1	SUBB	A, R7	
A0	2	ORL	C, /bit	
A1	2	AJMP	addr11	
A2	2	MOV	C, bit	
A3	1	INC	DPTR	
A4	1	MUL	AB	
A5		reserved		
A6	2	MOV	@R0, direct	
A7	2	MOV	@R1, direct	
A8	2	MOV	R0, direct	
A9	2	MOV	R1, direct	
AA	2	MOV	R2, direct	
AB	2	MOV	R3, direct	
AC	2	MOV	R4, direct	
AD	2	MOV	R5, direct	
AE	2	MOV	R6, direct	
AF	2	MOV	R7, direct	
B0	2	ANL	C, /bit	
B1	2	ACALL	addr11	
B2	2	CPL	bit	
B3	1	CPL	C	
B4	3	CJNE	A, #immed, offset	
B5	3	CJNE	A, direct, offset	
B6	3	CJNE	@R0, #immed, offset	
B7	3	CJNE	@R1, #immed, offset	
B8	3	CJNE	R0, #immed, offset	
B9	3	CJNE	R1, #immed, offset	
BA	3	CJNE	R2, #immed, offset	
BB	3	CJNE	R3, #immed, offset	

BC	3	CJNE	R4, #immed, offset	
BD	3	CJNE	R5, #immed, offset	
BE	3	CJNE	R6, #immed, offset	
BF	3	CJNE	R7, #immed, offset	
C0	2	PUSH	direct	
C1	2	AJMP	addr11	
C2	2	CLR	bit	
C3	1	CLR	C	
C4	1	SWAP	A	
C5	2	XCH	A, direct	
C6	1	XCH	A, @R0	
C7	1	XCH	A, @R1	
C8	1	XCH	A, R0	
C9	1	XCH	A, R1	
CA	1	XCH	A, R2	
CB	1	XCH	A, R3	
CC	1	XCH	A, R4	
CD	1	XCH	A, R5	
CE	1	XCH	A, R6	
CF	1	XCH	A, R7	
D0	2	POP	direct	
D1	2	ACALL	addr11	
D2	2	SETB	bit	
D3	1	SETB	C	
D4	1	DA	A	
D5	3	DJNZ	direct, offset	
D6	1	XCHD	A, @R0	
D7	1	XCHD	A, @R1	
D8	2	DJNZ	R0, offset	
D9	2	DJNZ	R1, offset	
DA	2	DJNZ	R2, offset	
DB	2	DJNZ	R3, offset	
DC	2	DJNZ	R4, offset	
DD	2	DJNZ	R5, offset	
DE	2	DJNZ	R6, offset	
DF	2	DJNZ	R7, offset	
E0	1	MOVX	A, @DPTR	
E1	2	AJMP	addr11	
E2	1	MOVX	A, @R0	
E3	1	MOVX	A, @R1	
E4	1	CLR	A	
E5	2	MOV	A, direct	
E6	1	MOV	A, @R0	
E7	1	MOV	A, @R1	
E8	1	MOV	A, R0	
E9	1	MOV	A, R1	
EA	1	MOV	A, R2	
EB	1	MOV	A, R3	
EC	1	MOV	A, R4	
ED	1	MOV	A, R5	
EE	1	MOV	A, R6	
EF	1	MOV	A, R7	
F0	1	MOVX	@DPTR, A	
F1	2	ACALL	addr11	
F2	1	MOVX	@R0, A	
F3	1	MOVX	@R1, A	
F4	1	CPL	A	
F5	2	MOV	direct, A	
F6	1	MOV	@R0, A	
F7	1	MOV	@R1, A	
F8	1	MOV	R0, A	
F9	1	MOV	R1, A	
FA	1	MOV	R2, A	
FB	1	MOV	R3, A	

FC	1	MOV	R4. A	
FD	1	MOV	R5. A	
FE	1	MOV	R6. A	
FF	1	MOV	R7. A	