Universidade do Minho
Escola de Engenharia

Fábio Oliveira, PG50363
Vitor Ferreira, PG50802

8051 Microcontroller - FPGA Implementation

Master's in Industrial Electronics and Computers Engineering

Embedded Systems

Professor:
Adriano Tavares

January, 2022

List of Figures

List of Tables

Contents

1 Introduction

The objective of this project is to design an 8051 microcontroller, model and simulate it using the Verilog hardware description language, and finally to implement it in hardware using a programmable field gate array(FPGA). This document describes the microcontroller design, the test code used to verify it, and the physical hardware implementation.

1.1 Requirements

The following are the requirements for this project:

- Implement a subset of the 8051 Instruction Set.
- Use fixed length instructions
- Implement a timer peripheral
- Implement two interrupts

2 8051 Architecture Overview

2.1 Overview of 8051 Microcontroller

The 8051 Microcontroller is one of the basic types of microcontrollers, designed by Intel in 1980s. This microcontroller is based on Harvard Architecture, meaning that the program and data memory spaces are separated. It has a total of 128 bytes of on-chip RAM, 4KB of on-chip ROM, and a variety of peripheral interfaces including serial communication, timers, and interrupt controller, also has a variety of instruction set, including arithmetic, logical, branching, and data transfer instructions, making it suitable for a wide range of applications.

2.2 On-Chip Memory Organization

The 8051 microcontroller has a variety of memory types that can be used for different purposes, including program memory, data memory, and special function registers. These memory types are used to store the program instructions, data used by the program, and configuration settings for the microcontroller. With a combination of internal and external memory, the 8051 can support programs of varying size and complexity.

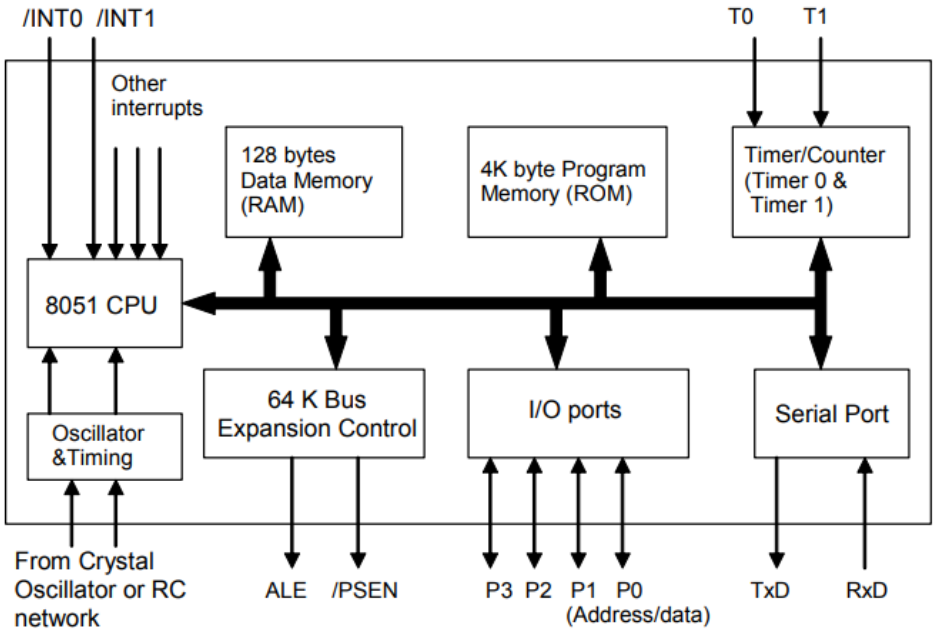

Figure 1: 8051 Architecture

Understanding the 8051 On-Chip Memory Organization is essential for effective and efficient programming of the 8051 microcontroller.

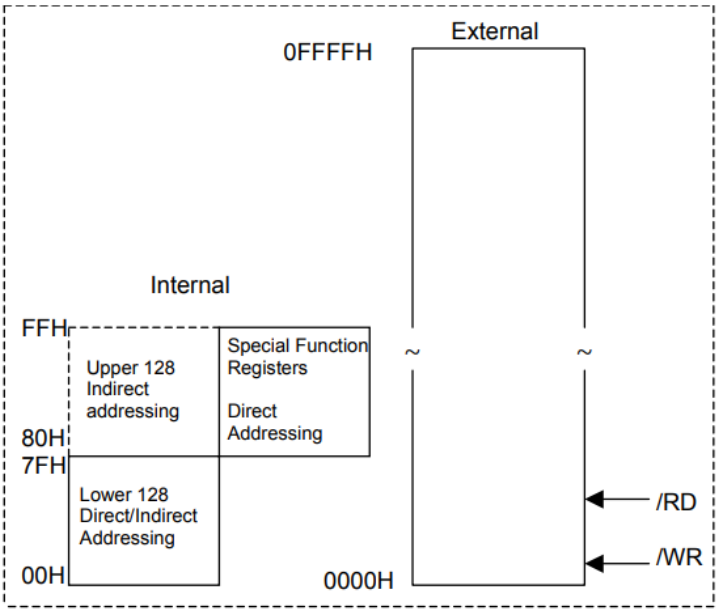

Figure 2: Memory organization

With both internal and external data memory, which is depicted in Figure 2. The internal data memory is further divided into three sections: Lower 128, Upper 128, and SFR. These sections have 384 physical bytes of memory space, with the Upper 128 and SFR sharing the same addresses from location 80H to FFH. The internal data memory is a Read/Write or Random Access Memory, and Figures 3a and 3b provide more detailed information about it. Instructions using direct or indirect addressing modes can be used to access the different memory blocks as required.

Byte Address	Bit Address								
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	-	D0	PSW
B8	-	-	-	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	-	-	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	Not bit-addressable								SBUF
98	9F	96	95	94	93	92	91	90	SCON
90	97	96	95	94	93	92	91	90	P1
8D	Not bit-addressable								TH1
8C	Not bit-addressable								TH0
8B	Not bit-addressable								TL1
8A	Not bit-addressable								TL0
89	Not bit-addressable								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	Not bit-addressable								PCON
83	Not bit-addressable								DPH
82	Not bit-addressable								DPL
81	Not bit-addressable								SP
80	87	86	85	84	83	82	81	80	P0

(a) SFR

Byte Address	Bit Address							
7F	General Purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0 – R7							
00								

(b) Register Banks and RAM

Figure 3: On-chip data Memory

3 Instruction Set

This section briefly describes each of the instructions implemented. The instructions chosen have all a fixed size of two bytes, in order to simplify the implementation. Operations are considered to be 8-bit operations with 8-bit results.

3.1 Data Transfer Instructions

3.1.1 MOV Rn, #immediate

This instruction moves an 8-bit immediate value to a register. The format of the instruction is MOV Rn, #immediate. Here Rn refers to the register where the value will be stored and immediate is the immediate value to be stored in the register.

Encoding	01111nnn	immediate

3.1.2 MOV A, Rn

This instruction moves the contents of a register to the accumulator. The format of the instruction is MOV A, Rn. Here A refers to the accumulator and Rn refers to the register containing the data.

Encoding	11101nnn	00000000

3.1.3 MOV direct, Rn

This instruction transfers the contents of register Rn to a direct address specified by the 8-bit operand. The operand acts as the memory address where the contents of Rn will be stored.

Encoding	10001nnn	direct

3.1.4 MOV Rn, A

This instruction transfers the content of the accumulator to a register.The format of the instruction is MOV Rn, A. Here Rn refers to the register where the value will be stored and A is the accumulator.

Encoding	11111nnn	00000000

3.1.5 MOV A, #immediate

This instruction moves an 8-bit immediate value to the accumulator. The format of the instruction is MOV A, #immediate. Here A refers to the accumulator where the value will be stored and immediate is the immediate value to be stored in the register.

Encoding	01110100	immediate

3.1.6 MOV A, direct

This instruction is used to move the contents of a specified direct address in memory to the accumulator register A.

Encoding	11100101	direct

3.1.7 MOV direct, A

This instruction is used to transfer the contents of accumulator to a direct address. The direct address is specified in the lower 8-bits of the instruction. The 8-bit data stored in the accumulator is copied to the specified direct address.

Encoding	11110101	direct

3.2 Arithmetic and Logical Instructions

3.2.1 ADD A, #immediate

This instruction adds an immediate value to the contents of the accumulator. The immediate is specified in the lower 8-bits of the instruction.

Encoding	00100100	immediate

3.2.2 SUBB A, #immediate

This instruction subtracts an immediate value to the contents of the accumulator. The immediate is specified in the lower 8-bits of the instruction.

Encoding	10010100	immediate

3.2.3 ORL A, #immediate

This instruction performs a bitwise OR operation between the accumulator and an immediate value, and stores the result back in the accumulator.

Encoding	01000100	immediate

3.2.4 ANL A, #immediate

This instruction performs a bitwise AND operation between the accumulator and an immediate value, and stores the result back in the accumulator.

Encoding	01010100	immediate

3.2.5 XRL A, #immediate

This instruction performs a bitwise XOR operation between the accumulator and an immediate value, and stores the result back in the accumulator.

Encoding	01100100	immediate

3.3 Branch Instructions

3.3.1 SJMP offset

This instruction transfers execution to the specified address. The address is calculated by adding the signed relative offset in the second byte of the instruction to the address of the following instruction. The range of destination addresses is from 128 before the next instruction to 127 bytes after the next instruction.

Encoding	10000000	offset

3.3.2 JNZ offset

This instruction transfers control to the specified address if the value in the accumulator is not 0. If the accumulator has a value of 0, the next instruction is executed.

Encoding	01110000	offset

3.3.3 JZ offset

The JZ instruction transfers control to the specified address if the value in the accumulator is 0. Otherwise, the next instruction is executed.

Encoding	01100000	offset

3.3.4 JNC offset

This instruction transfers program control to the specified address if the carry flag is 0.

Encoding	10100000	offset

3.3.5 RETI

The RETI instruction is used to end an interrupt service routine. This instruction pops the high-order and low-order bytes of the PC (and decrements the stack pointer by 2) and restores the interrupt logic to accept additional interrupts. No other registers are affected by this instruction.

Encoding	00110010	00000000

4 Verilog Implementation

4.1 General Overview

In figure 4 is presented the full microcontroller implementation with all the modules used.

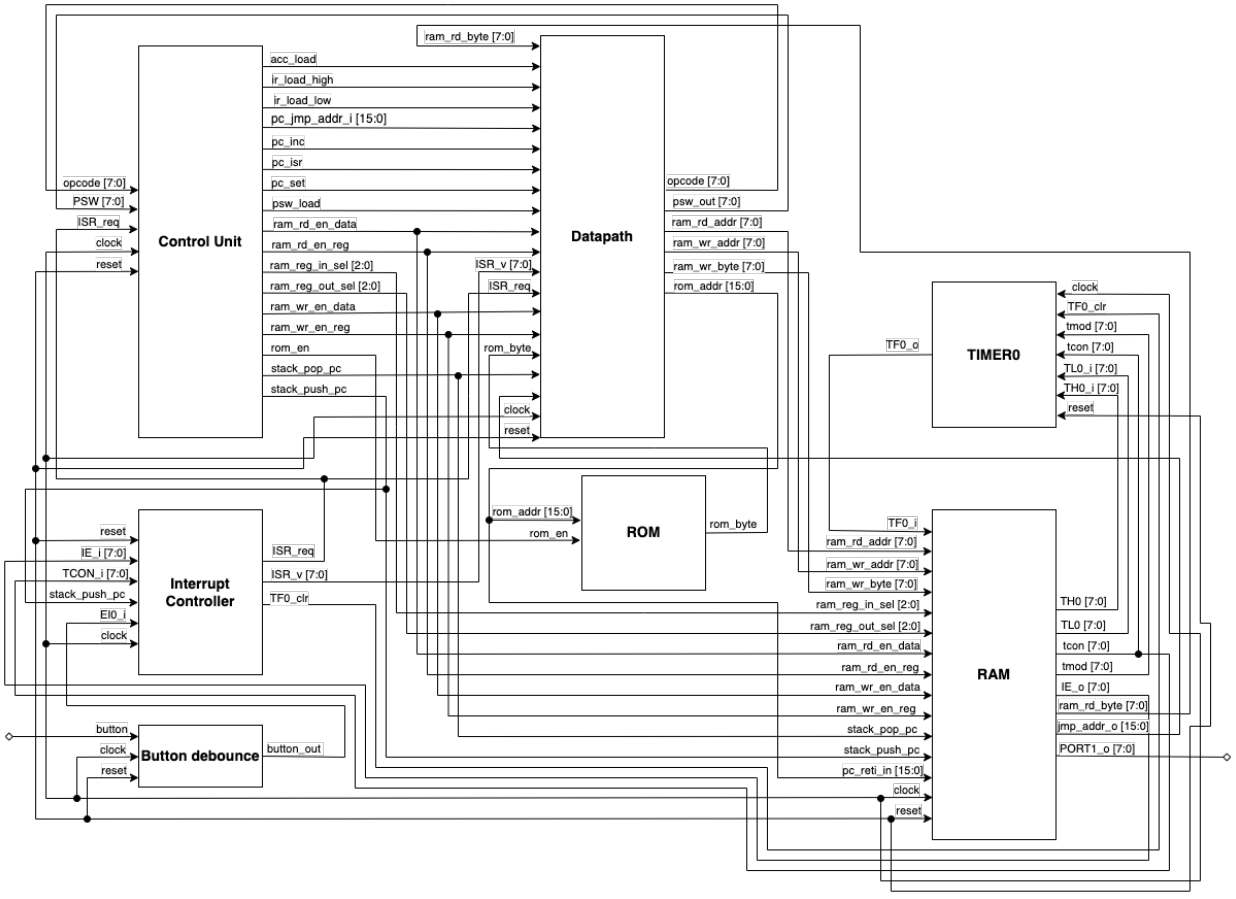

Figure 4: Full circuit

4.2 Datapath

The datapath is responsible for performing data processing operations and, in the implementation made, it consists of multiple components, such as program counter, instruction register, arithmetic logic unit, accumulator, and psw. These components are interconnected and interact with each other to perform various tasks.

The module code is shown in Appendix A.1.

4.2.1 Program Counter

The program counter, presented in figure 5, holds the address of the next instruction byte and is used to index ROM when fetching instructions, due to this fact the width of the program counter is 16 bits.

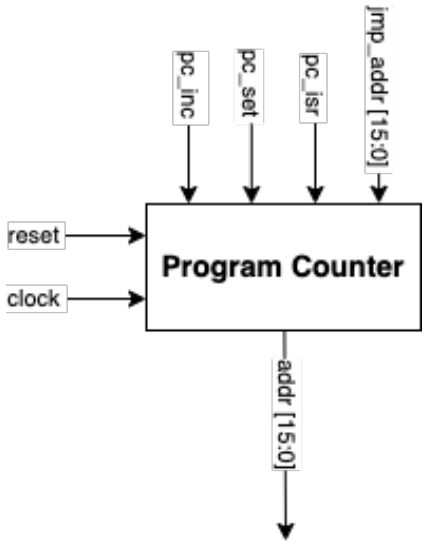

Figure 5: Program Counter diagram

The program counter has the following inputs:

- **pc_inc :** an increment signal that increments the program counter by one.
- **pc_set :** a set signal that sets the program counter to a specific value specified by the jmp_addr input.
- **pc_isr :** an interrupt signal that indicates the need to jump to an interrupt service routine vector.
- **jmp_addr :** a 16-bit input that specifies the address to which the program counter should be set.

The program counter has a single output, addr, which is a 16-bit register that holds the current value of the program counter.

On the rising edge of the clock, the value of addr is updated based on the values of the inputs. If the reset input is asserted, the program counter is set to 0. If the pc_inc input is asserted, the program counter is incremented by 1. If the pc_set input is asserted, the program counter is offset to the value

of jmp_addr. If the pc_isr input is asserted, the program counter is set to the value of jmp_addr.

The module code is shown in Appendix A.1.1.

4.2.2 Instruction Register

The instruction register, presented in figure 6, holds the instruction to be executed, as the chosen width of the instructions is 16 bits the instruction register is 16 bits wide.

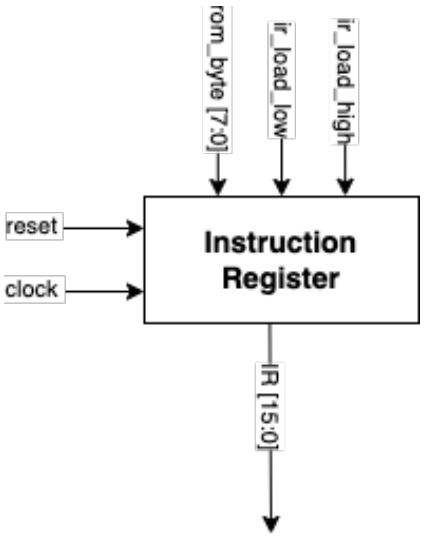

Figure 6: Instruction Register diagram

When the reset signal is high, the IR register is set to zero. When the clock signal goes high ,the values of ir_load_high and ir_load_low control signals are checked. If ir_load_high is high, the 8 most significant bits of IR are updated with the value of rom_byte. If ir_load_low is high, the 8 least significant bits of IR are updated with the value of rom_byte. The IR register holds the instruction until the next clock cycle and acts as a temporary storage for the instruction.

The module code is shown in Appendix A.1.2.

4.2.3 Accumulator

The Accumulator, present in figure 7, is used for arithmetic and logic operations.

When reset is high the accumulator is set to zero. The data_in input signal provides the data that is to be stored in the accumulator, and, the write_en input signal, when high, enables writing to the accumulator.

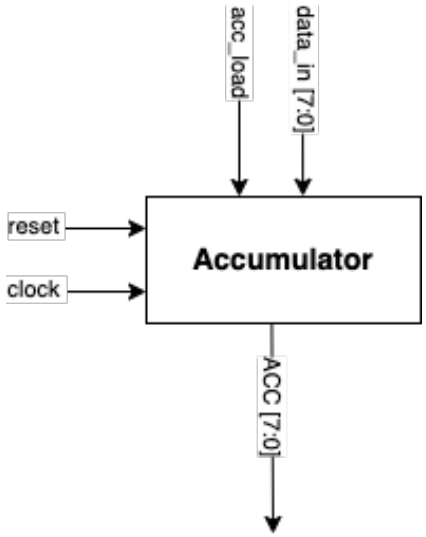

Figure 7: Accumulator diagram

The module code is shown in Appendix A.1.3.

4.2.4 Program Status Word

The Program Status Word (PSW), presented in figure 8, contains status bits that reflect the current CPU state.

The PSW register is updated on every positive edge of the clock signal. If the reset signal is high, the PSW register is set to zero. If the write_en signal is high, the PSW register is updated with the values of the carry, auxiliary carry, overflow, zero, and parity flags provided by the ALU.

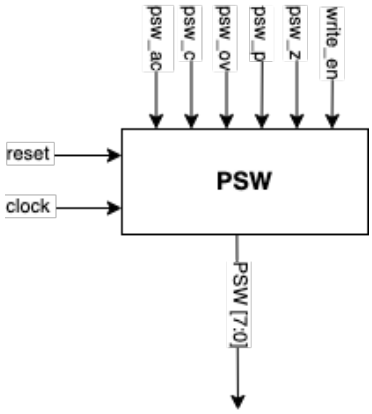

Figure 8: PSW diagram

The module code is shown in Appendix A.1.4.

4.2.5 Arithmetic Logic Unit

The ALU, presented in figure 9, consists entirely of combinational logic, and operations are performed whenever the inputs change. The operations are performed based on the opcode of the instruction, as the instruction set chosen always uses the accumulator as a source of the operations the output of the accumulator is connected directly to operand1. As the output of the ALU is connected to the accumulator the MOV instructions to the accumulator are also implemented on the ALU. If the result is all zeros, the psw_z bit is set. Likewise, the parity flag, psw_p, is set whenever the result is odd. The carry flag, psw_c, is set when the ninth bit of the result is 1.

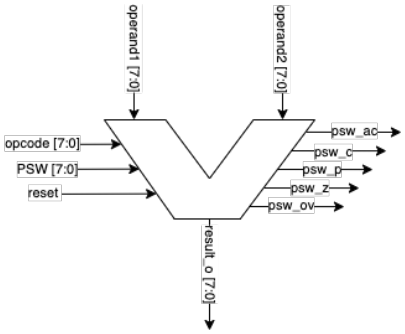

Figure 9: ALU diagram

The module code is shown in Appendix A.1.5.

4.3 Control Unit

4.3.1 State Machine

The state machine is responsible for decoding the instruction opcode and activate the corresponding control signals for all the other modules. The state diagram for the state machine is shown in figure 10

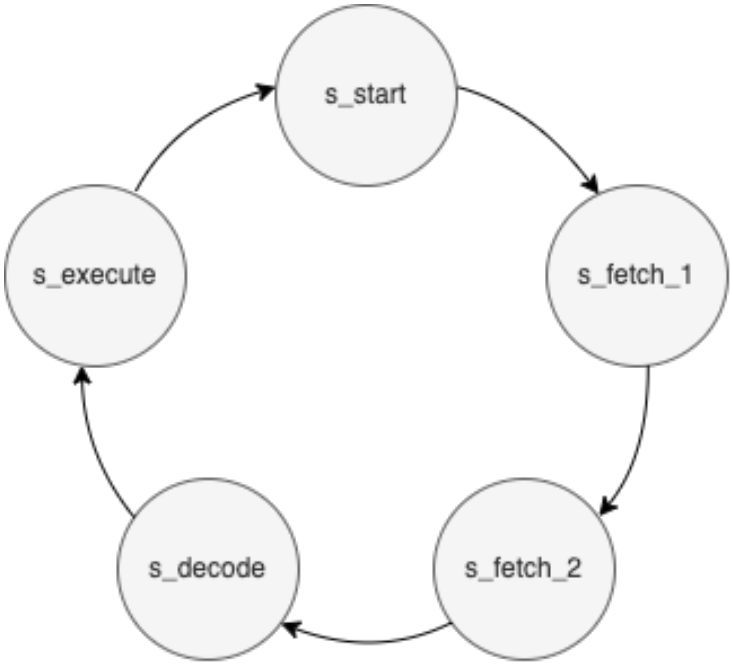

Figure 10: state diagram control unit

In order to simplify the state machine the state diagram above was presented, but in reality each instruction has a own state for s_execute as presented in figure 11.

Below is presented the function of each state.

- **s_start**: initial state
- **s_fetch_1**: state to fetch the most significant byte of the instruction to be executed
- **s_fetch_2**: state to fetch the least significant byte of the instruction to be executed
- **s_decode**: state to decode the opcode

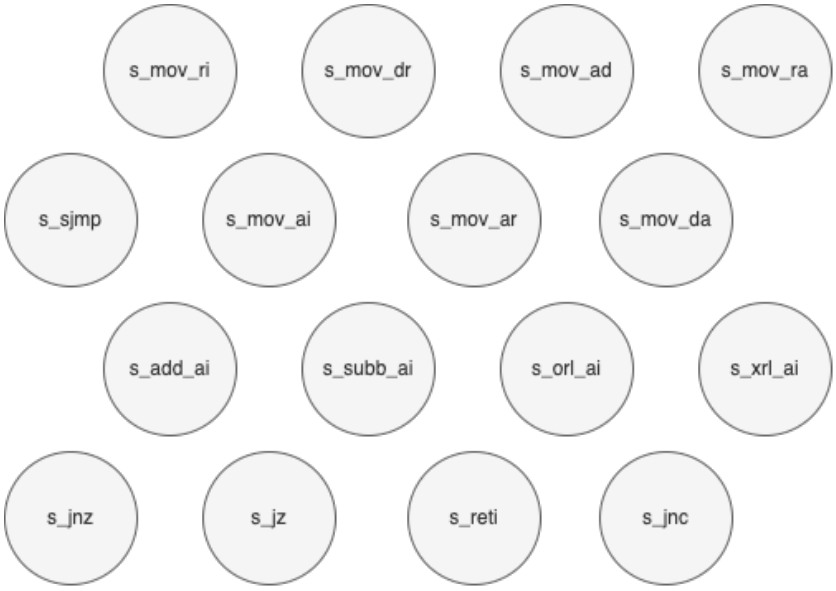

Figure 11: execute state for each instruction

- **s_mov_ri**: state to execute the instruction MOV Rn, #immediate
- **s_mov_dr**: state to execute the instruction MOV direct, Rn
- **s_mov_ad**: state to execute the instruction MOV A, direct
- **s_mov_ra**: state to execute the instruction MOV Rn, A
- **s_mov_ai**: state to execute the instruction MOV A, #immediate
- **s_mov_ar**: state to execute the instruction MOV A, Rn
- **s_mov_da**: state to execute the instruction MOV direct, A
- **s_add_ai**: state to execute the instruction ADD A, #immediate
- **s_subb_ai**: state to execute the instruction SUBB A, #immediate
- **s_orl_ai**: state to execute the instruction ORL A, #immediate
- **s_anl_ai**: state to execute the instruction ANL A, #immediate
- **s_xrl_ai**: state to execute the instruction XRL A, #immediate
- **s_sjmp**: state to execute the instruction SJMP offset
- **s_jnz**: state to execute the instruction JNZ offset

- **s_jz**: state to execute the instruction JZ offset
- **s_jnc**: state to execute the instruction JNC offset
- **s_reti**: state to execute the instruction RETI

The table 1 represents the mapping of the states to control signals.

	ram_rd_en_reg	ram_wr_en_reg	ram_rd_en_data	ram_wr_en_data	rom_en	pc_inc	pc_set	pc_isr	ir_load_high	ir_load_low	acc_load	stack_push_pc	stack_pop_pc	psw_load
Start	0	0	0	0	0	0	0	ISR_req	0	0	0	ISR_req	0	0
Fetch_1	0	0	0	0	1	1	0	0	1	0	0	0	0	0
Fetch_2	0	0	0	0	1	1	0	0	0	1	0	0	0	0
Decode	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV_RI	0	1	0	0	0	0	0	0	0	0	0	0	0	0
MOV_AI	0	0	0	0	0	0	0	0	0	0	1	0	0	0
MOV_AR	1	0	0	0	0	0	0	0	0	0	1	0	0	0
MOV_RA	0	1	0	0	0	0	0	0	0	0	0	0	0	0
MOV_AD	0	0	1	0	0	0	0	0	0	0	1	0	0	0
MOV_DR	1	0	0	1	0	0	0	0	0	0	0	0	0	0
MOV_DA	0	0	0	1	0	0	0	0	0	0	0	0	0	0
ADD_AI	0	0	0	0	0	0	0	0	0	0	1	0	0	1
SUBB_AI	0	0	0	0	0	0	0	0	0	0	1	0	0	1
ORL_AI	0	0	0	0	0	0	0	0	0	0	1	0	0	1
ANL_AI	0	0	0	0	0	0	0	0	0	0	1	0	0	1
XRL_AI	0	0	0	0	0	0	0	0	0	0	1	0	0	1
JNZ	0	0	0	0	0	0	∼PSW[1]	0	0	0	0	0	0	0
JZ	0	0	0	0	0	0	PSW[1]	0	0	0	0	0	0	0
SJMP	0	0	0	0	0	0	1	0	0	0	0	0	0	0
JNC	0	0	0	0	0	0	∼PSW[7]	0	0	0	0	0	0	0
RETI	0	0	0	0	0	0	0	1	0	0	0	0	1	0

Table 1: Mapping of states to control signals

The module code is shown in Appendix A.2.

4.4 RAM

The RAM, presented in figure 12, has inputs to read and write enable signals for both the register bank and the data memory, the select signals for both reading and writing from the register bank, the read and write addresses for the data memory, the byte being written to the memory, signals to push or pop the program counter onto or from the stack, input signal for the

timer 0 overflow flag and the program counter to load to the stack, it also has outputs for the read data from the data memory, the program counter address stored on the stack and several memory-mapped register values such as Timer Control Register (TCON), Timer Mode Register (TMOD), Timer 0 Low Byte (TL0) and Timer 0 High Byte (TH0), the Interrupt Enable register (IE) and P1 (PORT1).

When the either of the write enable signals are high, the data present in ram_wr_byte is store either on the registers or on the address specified by ram_wr_addr.

If the stack_push_pc signal is high, the current program counter is pushed onto the stack and the stack pointer is incremented by 2. If the stack_pop_pc signal is high, the stack pointer is decremented by 2, and the jump address (PC) from the stack is outputted.

The output ram_rd_byte is assigned the value of the data memory at the read address specified by ram_rd_addr, if the read enable signals are high. The jmp_addr_o output is assigned the address from the stack if the stack_pop_pc signal is high.

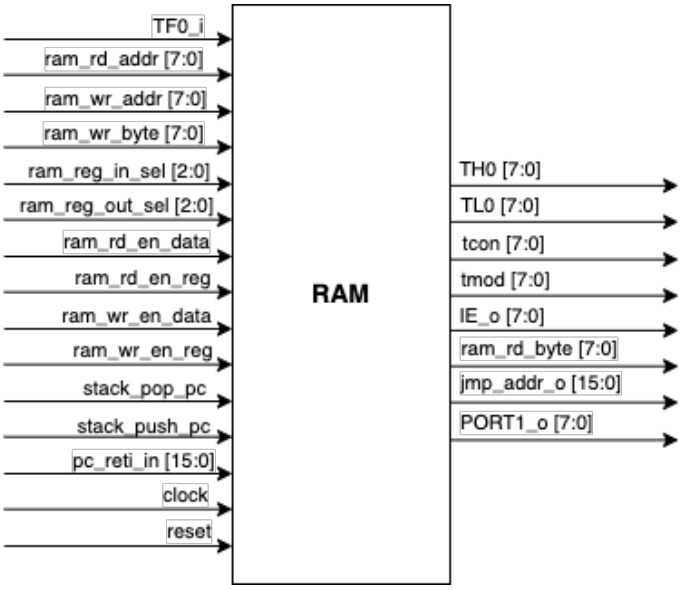

Figure 12: RAM diagram

The module code is shown in Appendix A.3.

4.4.1 Register Bank

The register bank, presented in 13, contains 8 registers each one 8-bits wide.

When the write_en signal, connected to ram_wr_en_reg, is high the contents of reg_in_data are stored to the register selected by reg_in_select. Otherwise, when the read_en signal, connected to ram_rd_en_reg, is high the contents of the register selected is passed to reg_data_out.

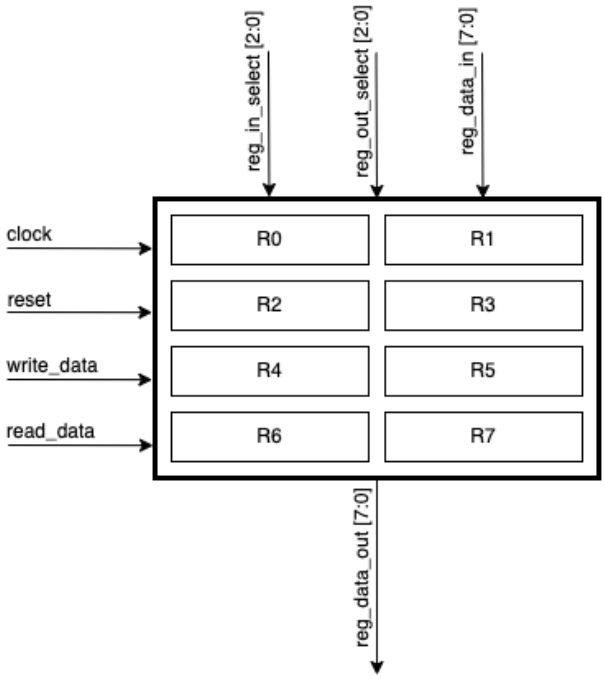

Figure 13: Register Bank

4.5 ROM

The ROM, presented in figure 14 is used to store the program code for a microcontroller. The size defined for the ROM is 256 words of 8 bits each, but it is possible to have 64kb of memory. The inputs to the module are a reset signal, enable signal (rom_en), address (rom_addr), and the output is the byte stored at that address (rom_byte).

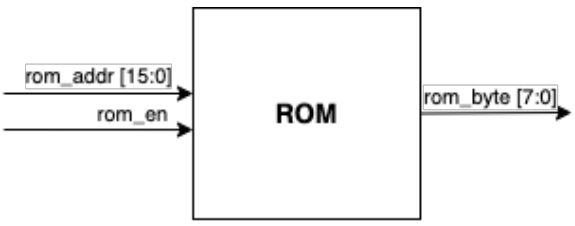

Figure 14: ROM diagram

4.6 Timer 0

The block diagram for the timer 0 is presented in figure 15, the 8051 timer 0 has various modes of operation:

- **Mode 0 (13-bit Timer mode):** In this mode, the timer increments every machine cycle and the overflow flag is set when the timer overflows from 8192 to 0.
- **Mode 1 (16-bit Timer mode):** In this mode, the timer is incremented every machine cycle and the overflow flag is set when the timer overflows from 65536 to 0.
- **Mode 2 (8-bit Auto-Reload mode):** In this mode, the timer is incremented every machine cycle. When the timer overflows from 255 to 0, the timer is automatically reloaded with a value stored in the reload register.
- **Mode 3 (Split Timer mode):** In this mode, Timer 0 is split into two 8-bit timers, Timer 0 and Timer 1. Both timers can operate independently and can be set up for different modes of operation.

As the most used modes are mode 1 and 2 these were the only modes implemented.

The Timer 0 takes several inputs such as a clock signal, a reset signal, tmod and tcon which control the mode of operation, TH0_i and TL0_i which are used to load the initial value of the timer, TF0_clr which is used to clear the timer overflow flag and an output signal TF0_o which indicates if the timer has overflowed.

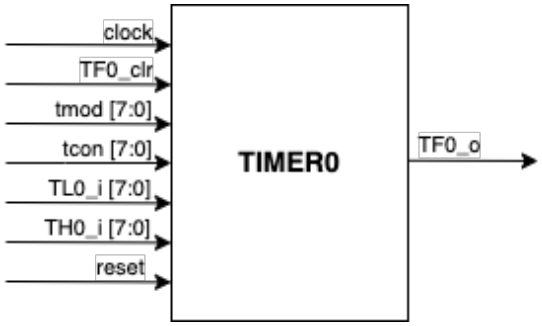

Figure 15: TIMER0 diagram

4.7 Interrupt Controller

The block diagram for the interrupt controller is presented in figure 16. The interrupt controller has inputs for the clock signal, reset signal, stack_push_pc signal in order to know when the interrupt is attended, as well as input signals for the Interrupt Enable (IE_i) register, the Timer Control (TCON_i) register, and the external interrupt 0 (EI0_i). It also has several outputs including the signal to clear the Timer 0 overflow flag (TF0_clr), the Interrupt Service Routine (ISR_v) address, and the Interrupt Service Request (ISR_req) signal.

The interrupt controller's main function is to generate the ISR_req signal when an interrupt occurs. This is done by checking the values of the input signals and the current state of the ISR_req signal. If the external interrupt 0 (EI0_i) is set, the Interrupt Enable bit for External Interrupt 0 (IE_i[0]), and the global Interrupt Enable bit (IE_i[7]) are set, and there is not already an ISR_req signal, the ISR_req signal will be set and the ISR_v register will be loaded with the address of the External Interrupt 0.

Similarly, if the Timer 0 Interrupt flag (TCON_i[5]) is set, the Interrupt Enable bit for Timer 0 (IE_i[1]), and the global Interrupt Enable bit (IE_i[7]) are set, and there is not already an ISR_req signal, the ISR_req signal will be set and the ISR_v register will be loaded with the address of the Timer 0 ISR.

The ISR_req signal is cleared when the stack push PC signal is asserted, which means the interrupt has been serviced. The TF0_clr output is used to clear the Timer 0 interrupt flag and is set when the ISR_req signal is set.

In summary, the interrupt controller is responsible for generating the ISR_req signal and the ISR_v address when an interrupt occurs, and for clearing the interrupt flags and ISR_req signal when the interrupt has been serviced.

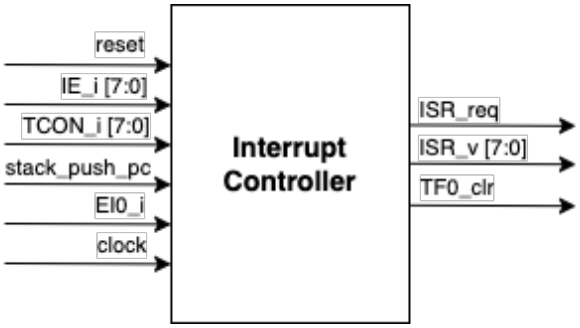

Figure 16: Interrupt Controller diagram

4.8 Constraints

The following constraints were used. The clock frequency is set to 50MHz, the reset input is set to the button Y16 and the external interrupt input is mapped to the K19 button. The I/O port P1 four least significant bits are set to the LED's and the rest is mapped to ports with pull-down.

```
1  set_property IOSTANDARD LVCMOS33 [get_ports reset]
2
3  set_property -dict {PACKAGE_PIN K17 IOSTANDARD LVCMOS33} [
       get_ports clock]
4  create_clock -period 20.000 -name sys_clk_pin -waveform
       {0.000 10.000} -add [get_ports clock]
5
6  set_property PACKAGE_PIN Y16 [get_ports reset]
7  set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[7]}]
8  set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[6]}]
9  set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[5]}]
10 set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[4]}]
11 set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[3]}]
12 set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[2]}]
13 set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[1]}]
14 set_property IOSTANDARD LVCMOS33 [get_ports {PORT1_o[0]}]
15 set_property PACKAGE_PIN M14 [get_ports {PORT1_o[0]}]
16 set_property PACKAGE_PIN M15 [get_ports {PORT1_o[1]}]
17 set_property PACKAGE_PIN G14 [get_ports {PORT1_o[2]}]
18 set_property PACKAGE_PIN D18 [get_ports {PORT1_o[3]}]
19 set_property PACKAGE_PIN H15 [get_ports {PORT1_o[4]}]
20 set_property PACKAGE_PIN J15 [get_ports {PORT1_o[5]}]
21 set_property PACKAGE_PIN W16 [get_ports {PORT1_o[6]}]
22 set_property PACKAGE_PIN V12 [get_ports {PORT1_o[7]}]
23 set_property PULLDOWN true [get_ports {PORT1_o[7]}]
24 set_property PULLDOWN true [get_ports {PORT1_o[6]}]
25 set_property PULLDOWN true [get_ports {PORT1_o[5]}]
26 set_property PULLDOWN true [get_ports {PORT1_o[4]}]
```

```
27 set_property PACKAGE_PIN K19 [get_ports pb_1]
28 set_property IOSTANDARD LVCMOS33 [get_ports pb_1]
```

5 Verification and validation

5.1 Arithmetic, Logic and Data Transfer Instructions

5.1.1 ADD A, immediate

The following assembly code was tested.

```
1 MOV R5, #08h
2 MOV A, R5
3 ADD A, #11h
4 MOV 50h, A
```

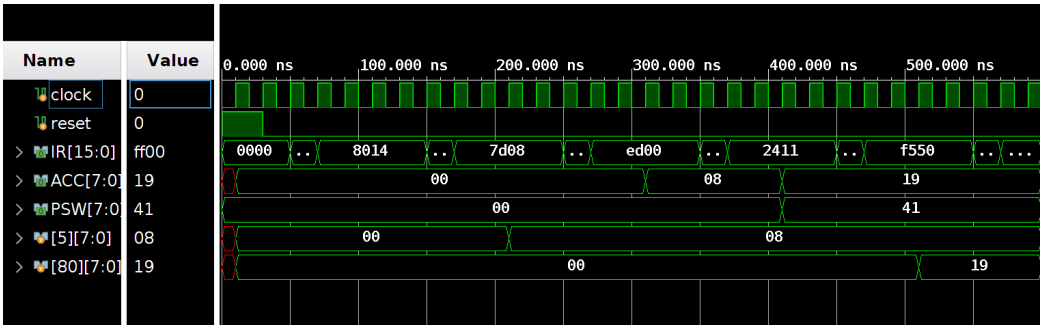

Figure 17: Simulation Waveform

It is possible to see in figure 17 that the value 08H is being stored on R5 (simulation line 7), after that the value from R5 is being stored on the accumulator and later 11H is added to 08H equaling 19H, this value stored on the accumulator is then stored on position 50H of the RAM (simulation line 8). Moreover, the value on the PSW is updated on par with the result.

5.1.2 SUBB A, immediate

The following assembly code was tested.

```
1 MOV A, #10h
2 SUBB A, #20h
3 MOV R4, A
```

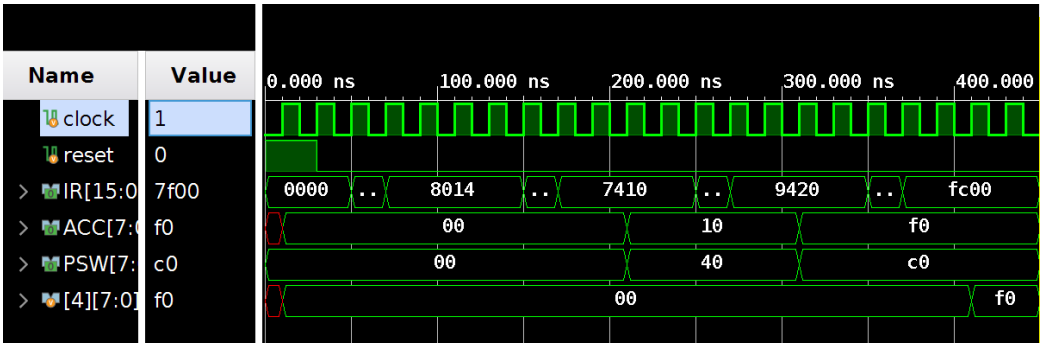

Figure 18: Simulation Waveform

It is possible to see in figure 18 that the value 10H is being stored on the accumulator, after the accumulator is subtracted by 20h which will result in a negative result, setting the carry bit of the PSW. Finally, the value on the accumulator is stored on the register R4.

5.1.3 ORL A, immediate

The following assembly code was tested.

```
1 MOV R2, #36h
2 MOV 53h, R2
3 MOV A, 53h
4 ORL A, #C9h
```

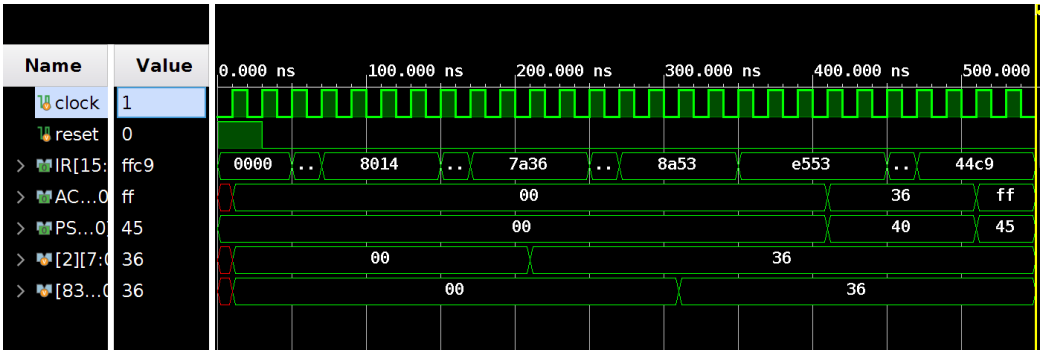

Figure 19: Simulation Waveform

It is possible to see in figure 19 that the value 32H is being stored on the register R2, next the value from value is stored on position 53H on the RAM, after the accumulator is loaded with the value from 53H and an OR is made with C9H resulting in FFH.

5.1.4 ANL A, immediate

The following assembly code was tested.

```
1 MOV R3, #36h
2 MOV 60h, R3
3 MOV A, 60h
4 ANL A, #C9h
```

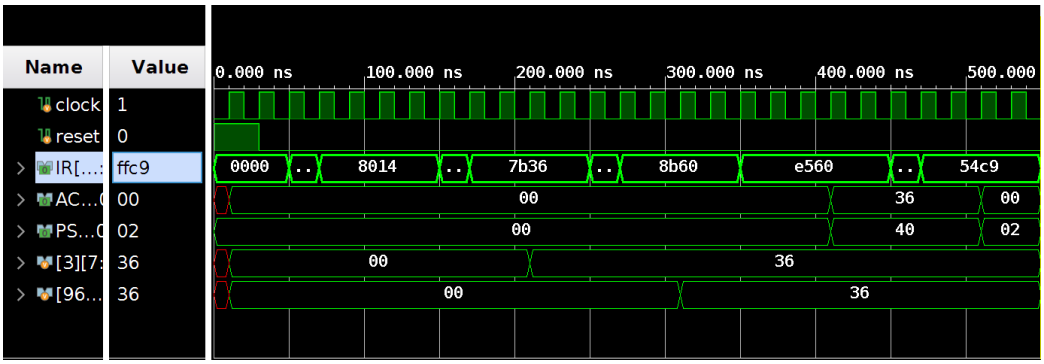

Figure 20: Simulation Waveform

It is possible to see in figure 20 that the value 36H is being stored on the register R3, next the value from value is stored on position 60H on the RAM, after the accumulator is loaded with the value from 60H and an AND is made with C9H resulting in 00H.

5.1.5 XRL A, immediate

The following assembly code was tested.

```
1 MOV A, #10h
2 XRL A, #81h
```

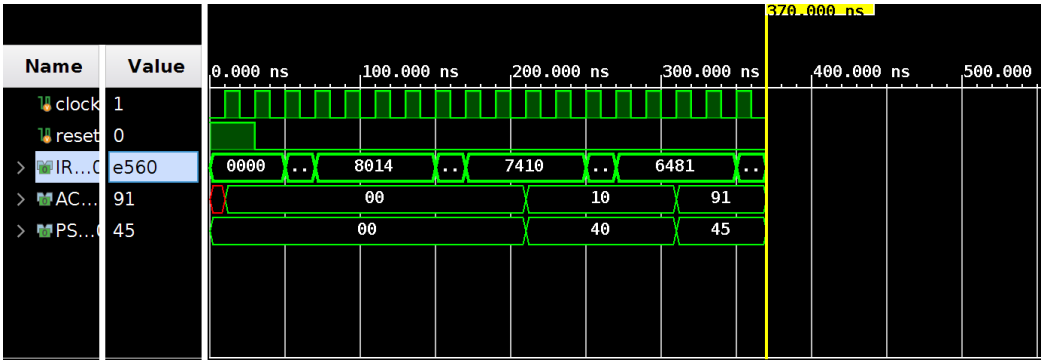

Figure 21: Simulation Waveform

It is possible to see in figure 21 that the value 19H is being loaded to the accumulator and next a XOR operation is made with 81H resulting in 91H.

5.2 Branch Instructions

5.2.1 JNC

The following assembly code was tested.

```
1 MOV A, #FFh
2 ADD A, #01h
3 JNC #02h
4 SUBB A, #FFh
5 MOV R7, A
```

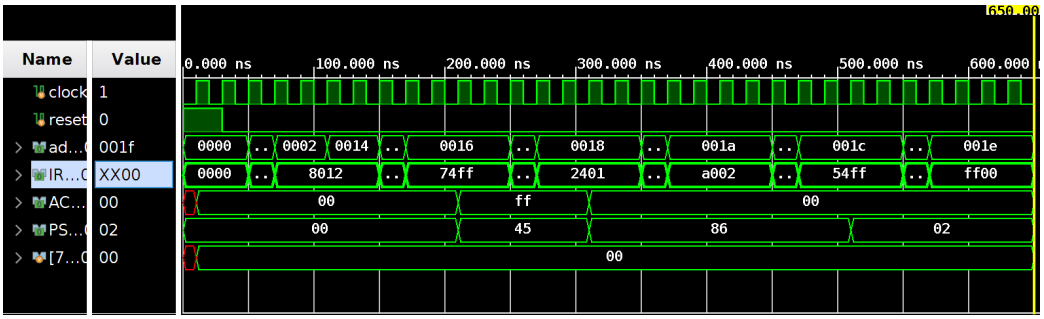

Figure 22: Simulation Waveform

It is possible to see in figure 22 that the first instruction to be executed is a SJMP with offset 12H, next is added 01H to FFH resulting in the carry bit being set which means that JNC would not jump, thus subtracting FFH to the accumulator and storing that value in the register R7.

The following assembly code was also tested.

```
1 MOV A, #FFh
2 ADD A, #00h
3 JNC #02h
4 SUBB A, #FFh
5 MOV R7, A
```

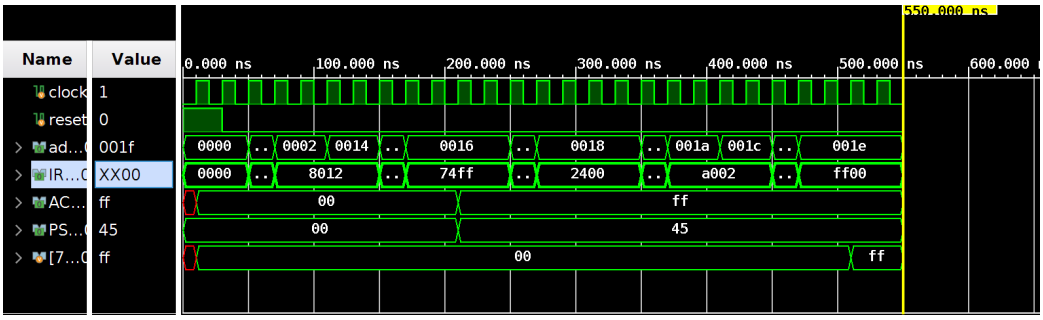

Figure 23: Simulation Waveform

It is possible to see in figure 23 that the first instruction to be executed is a SJMP with offset 12H, next is added 00H to FFH resulting in the carry bit being 0 which means that the jump with offset 02H did occur, thus not subtracting FFH to the accumulator and storing that value in the register R7.

5.2.2 JNZ

The following assembly code was tested.

```
1 MOV A, #08h
2 SUBB A, #09h
3 JNZ #02h
4 ADD A, #03h
```

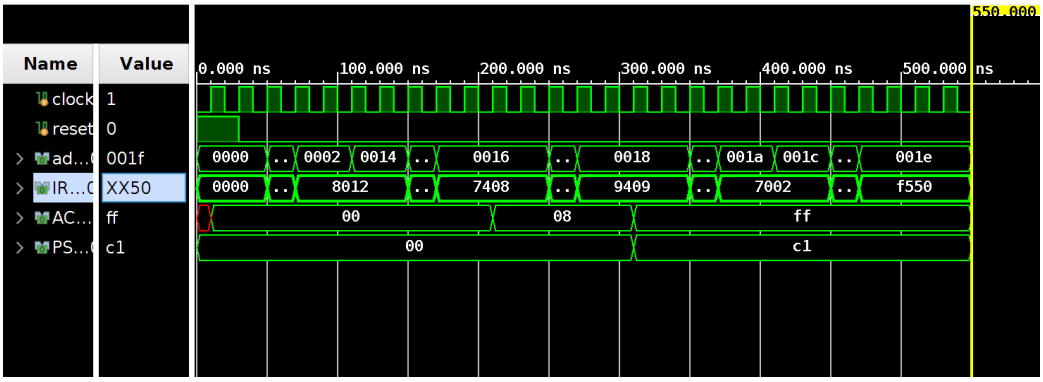

Figure 24: Simulation Waveform

It is possible to see in figure 24 that the first instruction to be executed is a SJMP with offset 12H, next the accumulator is loaded with 08H and subtracted 09H to its contents resulting in FFH, different than 0, which means JNZ would jump, thus not adding 03H to the accumulator.

The following assembly code was also tested.

```
1 MOV A, #08h
2 SUBB A, #08h
3 JNZ #02h
4 ADD A, #03h
```

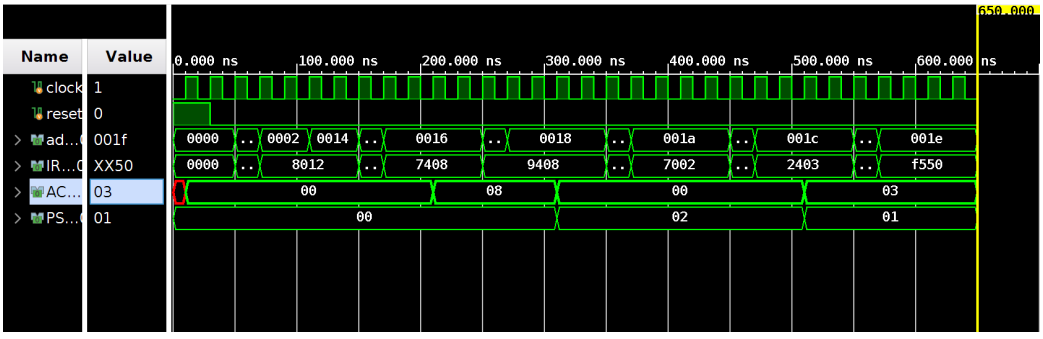

Figure 25: Simulation Waveform

It is possible to see in figure 25 that the first instruction to be executed is a SJMP with offset 12H, next, the accumulator is loaded with 08H and 08H is subtracted to its contents resulting in the result being 0 which means that the jump with offset 02H did not occur, thus adding 03H to the accumulator.

5.2.3 JZ

The following assembly code was tested.

```
1 MOV A, #08h
2 SUBB A, #08h
3 JNZ #02h
4 ADD A, #03h
```

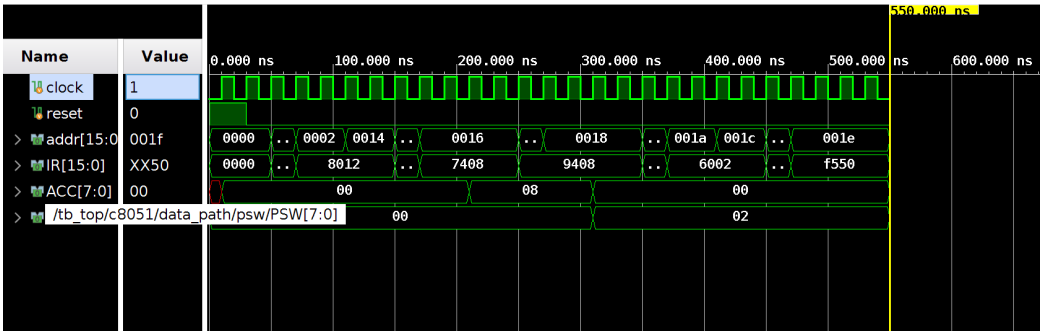

Figure 26: Simulation Waveform

It is possible to see in figure 26 that the first instruction to be executed is a SJMP with offset 12H, next the accumulator is loaded with 08H and subtracted 08H to its contents resulting in 0, which means JZ would jump, thus not adding 03H to the accumulator.

The following assembly code was also tested.

```
1 MOV A, #08h
2 SUBB A, #09h
3 JNZ #02h
4 ADD A, #03h
```

It is possible to see in figure 27 that the first instruction to be executed is a SJMP with offset 12H, next, the accumulator is loaded with 08H and 09H is subtracted to its contents resulting in the result being FFH, which means that the jump with offset 02H did not occur, thus adding 03H to the accumulator.

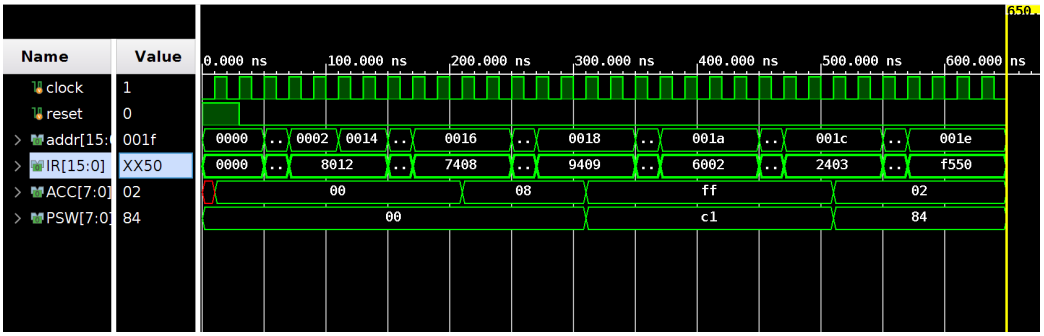

Figure 27: Simulation Waveform

5.3 Timer 0

5.3.1 Mode 1 - 16-bit timer

The Timer was loaded with the value FFF0h. In figure 28 it is possible to observe that the timer was loaded with FFF0h and when FFFFh was reached the overflow flag was set and the timer continue counting from 00h.

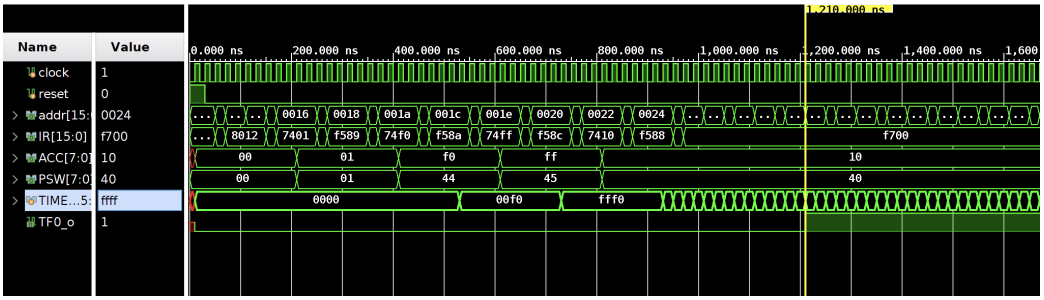

Figure 28: Simulation Waveform

5.3.2 Mode 2 - 8-bit auto reload timer

The Timer was loaded with the value F0h and the auto reload value was also set to F0h. In figure 29 it is possible to observe that the timer was loaded with F0h and when FFh was reached the overflow flag was set and the timer was reloaded with F0h counting.

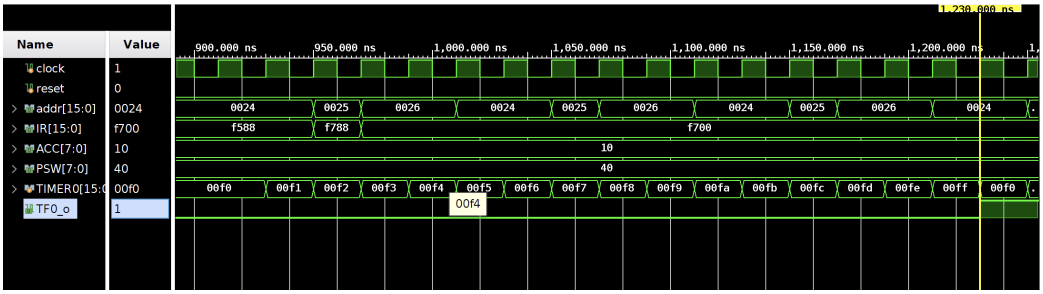

Figure 29: Simulation Waveform

5.4 Interrupts

5.4.1 Timer 0

The Timer 0 was loaded with 0000h and is operating in mode 1. In figure 30 it is possible to see that when the timer reached FFFFh both the overflow flag was set and the ISR_req signal were set and when the previous instruction finished executing the program counter was pushed to the stack before being updated with the Timer 0 ISR vector.

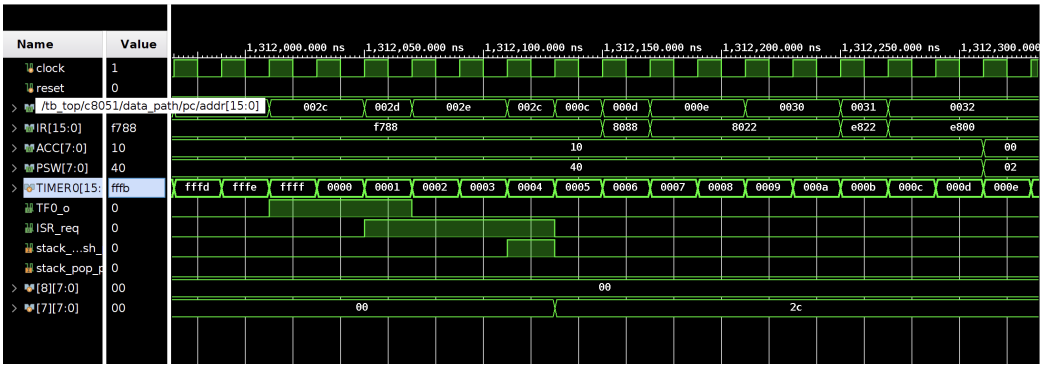

Figure 30: Simulation Waveform

5.4.2 External Interrupt 0

In figure 31, when the button (EI0_i) is pressed an interrupt request is launched and the program counter (addr) is pushed to the stack before being updated with the External Interrupt 0 ISR vector.

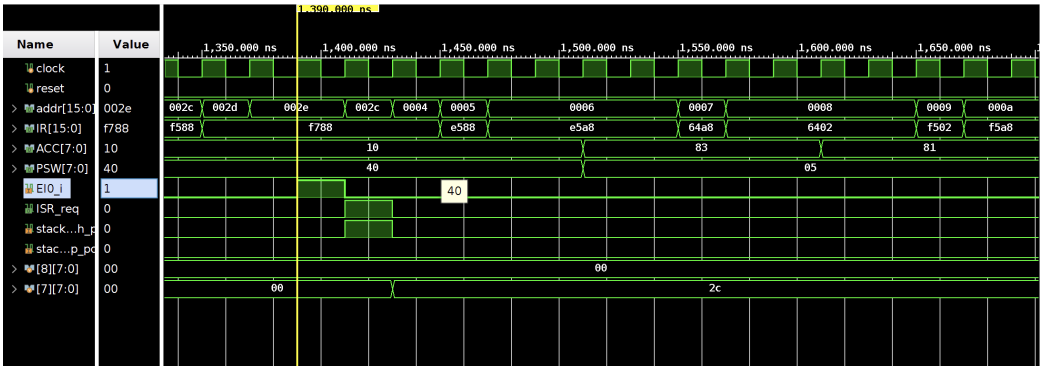

Figure 31: Simulation Waveform

5.5 Test in Hardware

The following code was run on the zybo and increments the I/O port P1 every second, this increment can be stopped or continue via an external interrupt.

In order to achieve this, the Timer 0 was configure in mode 1 - 16-bit timer and the registers TH0 and TL0 are loaded with 0, this would create a interrupt every 1.3ms so, to increment the output every second two registers are used, R0 that counts 255 interrupts and R1 that counts the number of times R0 overflows. When R1 reaches 3 the output is incremented. An external interrupt was also implemented in order to disable or enable the Timer 0 interrupt by switching the second bit of IE.

```
1     SJMP MAIN
2
3   EXT0:
4     MOV A, IE
5     XRL A, #02h
6     MOV IE, A
7     RETI
8
9   TIMER0:
10    SJMP TIMER0_ISR
11
12  MAIN:
13    MOV A, #01h
14    MOV TMOD, A
15    MOV A, #00h
16    MOV TL0, A
```

```
17      MOV TH0, A
18      MOV R0, #00h
19      MOV R1, #00h
20      MOV A, #83h
21      MOV IE, A
22      MOV A, #10h
23      MOV TCON, A
24      JNZ $
25
26  TIMER0_ISR:
27      MOV A, R0
28      ADD A, #01h
29      MOV R0, A
30      JNZ #12h
31      MOV A, R1
32      ADD A, #01h
33      MOV R1, A
34      SUBB A, #04h
35      JNZ #08h
36      MOV R1, #00h
37      MOV A, P1
38      ADD A, #01h
39      MOV P1, A
40      RETI
41
42  END
```

A video demonstrating this program working on the zybo can be found at `https://drive.google.com/file/d/1sjSU9oal2vwPEFBOH7fA2N8hXlndd8Kk/view?usp=sharing`

A Verilog module code

A.1 Datapath

```
1
2  module datapath(
3      input clock,
4      input reset,
5      input [7:0] ram_rd_byte,  // byte read from RAM
6      input [7:0] rom_byte,     // byte read from ROM,
     connected to the input of the Instruction Register
7      input ram_rd_en_reg,      // Control signal that enables
     reading from RAM register bank
8      input ram_wr_en_reg,      // Control signal that enables
     writing to RAM register bank
9      input ram_rd_en_data,     // Control signal that enables
     reading from RAM data section
10     input ram_wr_en_data,     // Control signal that enables
     writing to RAM data section
11     input pc_inc,             // When this signal is high the
      program counter is incremented
12     input pc_set,             // When this signal is high the
      value on jmp_addr is added to the program counter
13     input pc_isr,             // When this signal is high the
      program counter is set to the isr vector present in
     jmp_addr
14     input ir_load_high,       // When this signal is high the
      most significant byte of the IR is loaded with the byte
     fetched from ROM
15     input ir_load_low,        // When this signal is high the
      least significant byte of the IR is loaded with the byte
     fetched from ROM
16     input acc_load,           // Control signal to load the
     accumulator, connected to the accumulator write_en
17     input stack_pop_pc,       // Control signal to pop the
     program counter from the stack when the RETI instruction
     is executed
18     input ISR_req,            // Control signal to indicate
     that a interrupt needs to be serviced
19     input [7:0] ISR_v,        // ISR address vector provided
     to the program counter when an interrupt is triggered
20     input wire [15:0] jmp_addr_i, // Contains the program
     counter stored in the stack when the RETI instruction is
     executed
21     input psw_load,           // Control signal to load the
     PSW register
22
23     output wire [7:0] ram_rd_addr,  // Address to read from
```

```
    RAM
24   output wire [7:0] ram_wr_addr,  // Address to write to
    RAM
25   output wire [7:0] ram_wr_byte,  // Byte to write to RAM
26   output [7:0] opcode,            // Instruction opcode
27   output wire [15:0] rom_addr,    // Program Counter output
     used to fetch instructions from ROM
28   output wire [7:0] psw_out       // Output of PSW
29   );
30
31
32   wire [15:0] pc_jmp_addr;       // Jump address provided
    to the program counter
33   wire [15:0] IR;                // Connected to the output
     of the Instruction Register
34   wire [7:0] alu_out;            // Result from the ALU
35   wire [7:0] acc_out;            // Output from ACC passed
    to the ALU as operand1
36   wire [7:0] register_b;         // Passed to the ALU as
    the second operand
37
38   wire psw_c;                    //carry bit
39   wire psw_ac;                   //auxiliary carry bit
40   wire psw_ov;                   //overflow bit
41   wire psw_z;                    //zero bit
42   wire psw_p;                    //parity bit
43
44   //Instanciation of the program counter module
45   program_counter pc(clock, reset, pc_inc, pc_set,pc_isr,
    pc_jmp_addr,rom_addr);
46
47   //Instanciation of the instruction register module
48   instruction_register ir(clock,reset,ir_load_high,
    ir_load_low,rom_byte,IR);
49
50   //Instanciation of the ALU module
51   arithmetic_logic_unit alu(reset,acc_out,register_b,opcode
    [7:0], psw_out,alu_out,psw_c,psw_ac,psw_ov,psw_p,psw_z);
52
53   //Instanciation of the accumulator module
54   acc acc(clock, reset, alu_out, acc_load, acc_out);
55
56   //Instanciation of the PSW module
57   psw psw(clock, reset, psw_c, psw_ac, psw_ov, psw_z, psw_p
    , psw_load, psw_out);
58
59
60   assign opcode = IR[15:8]; // Assigning the opcode to the
    most significant byte of IR
```

```
61
62
63      assign ram_rd_addr = (ram_rd_en_data == 1'b1) ? IR[7:0] :
        8'hzz; //When ram_rd_en_data is high the address to read
       from the RAM is the least significant byte of IR
64      assign ram_wr_addr = (ram_wr_en_data == 1'b1) ? IR[7:0] :
        8'hzz; //When ram_wr_en_data is high the address to write
        to the RAM is the least significant byte of IR
65      assign register_b = (ram_rd_en_reg == 1'b1 ||
       ram_rd_en_data == 1'b1) ? ram_rd_byte : IR[7:0]; // If the
        second operand of the alu is fetched from RAM the byte
       read from RAM is passed to the ALU, else the least
       significant byte of the IR is passed as second operand
66      assign pc_jmp_addr = (pc_set == 1'b1 && pc_isr == 1'b0) ?
        IR[7:0] : (pc_isr == 1'b1 && ISR_req == 1'b1) ? ISR_v : (
       stack_pop_pc == 1'b1) ? jmp_addr_i : 8'hzz; // The jump
       address can be provided by the IR in the form of an offset
       , as an isr vector or as the return address of isr
67      assign ram_wr_byte = (ram_wr_en_reg == 1'b1 && IR[15] ==
       0) ? IR[7:0] : acc_out; //If a register is being loaded
       with an immediate then the byte to write to the ROM is the
        least significant byte from IR, else, every other
       instruction writes the accumulator to the RAM
68
69  endmodule
```

A.1.1 Program counter

```
1   module program_counter(
2       input clock,
3       input reset,
4       input inc,              // When this signal is high the
       program counter is incremented
5       input set,              // When this signal is high the
       value on jmp_addr is added to the program counter
6       input isr,              // When this signal is high the
       program counter is set to the isr vector present in
       jmp_addr
7       input [15:0] jmp_addr,  // Value for the program counter
       to be set
8       output reg [15:0] addr  // Value of the program counter
9       );
10
11
12      initial begin
13          addr = 0;
14      end
15
```

```
16      always @(posedge clock) begin
17          if  (reset) begin                   //If the reset
    signal is high the program counter is set to 0
18              addr <= 0;
19          end
20          else if(set==1 && isr==0) begin     //If set is high
    the value on jmp_addr is added to the program counter
21              addr <= addr + jmp_addr;
22          end
23          else if(inc) begin                  //If inc is high
    the program counter is incremented
24              addr <= addr + 1;
25          end
26          else if(isr==1 && set==0) begin     //If isr is high
    the program counter is set to the jmp_addr wich is
    provided by the interrupt controller as the corresponding
    isr vector
27              addr <= jmp_addr;
28          end
29      end
30
31  endmodule
```

A.1.2 Instruction Register

```
1   module instruction_register(
2       input clock,
3       input reset,
4       input ir_load_high,     //When this signal is high the
    most significant byte of the IR is loaded with the byte
    fetched from ROM
5       input ir_load_low,      //When this signal is high the
    least significant byte of the IR is loaded with the byte
    fetched from ROM
6       input [7:0] rom_byte,   //Byte fetched from ROM
7       output reg [15:0] IR    //Stores the instruction to be
    executed
8       );
9
10      initial begin
11          IR = 0;
12      end
13
14      always @(posedge clock)
15      begin
16          if(reset)           //When reset is high the IR is
    set to 0
17          begin
```

```
18              IR <= 0;
19          end
20          else if(ir_load_high)  //The most significant byte of
    IR is loaded with the rom byte
21          begin
22              IR[15:8] <= rom_byte;
23          end
24          else if (ir_load_low)  //The least significant byte
   of IR is loaded with the rom byte
25          begin
26              IR[7:0] <= rom_byte;
27          end
28       end
29
30 endmodule
```

A.1.3 Accumulator

```
1  module acc(
2        input clk,
3        input rst,
4        input [7:0] data_in,    //data to be stored in the
   accumulator
5        input write_en,         //When this signal is high
   data_in is stored in the ACC register
6        output reg [7:0] ACC    //ACC register
7     );
8
9     always @(posedge clk) begin
10       if(rst)                 //When reset is high the
   accumulator is set to 0
11       begin
12         ACC <= 0;
13       end
14
15       if (write_en)          //ACC is loaded with data_in when
    write_en is high
16       begin
17         ACC <= data_in;
18       end
19    end
20
21 endmodule
```

A.1.4 Program Status Word

```
/*
 ---------------------------------------------------------
|  CY   |  AC   |  F0   |  RS1  |  RS0  |  OV   |  Z  |  P  |
  psw.7  psw.6  psw.5  psw.4   psw.3  psw.2  psw.1 psw.0 */


module psw(
    input clk,
    input rst,
    input c,              //carry bit provided by the ALU
    input ac,             //auxiliary carry bit provided by the
    ALU
    input ov,             //overflow bit provided by the ALU
    input z,              //zero bit provided by the ALU
    input p,              //parity bit provided by the ALU (
   high -> odd)
    input write_en,       //When high the PSW is updated with
   the bits provided above
    output reg [7:0] PSW //PSW register
    );

    initial begin
        PSW = 8'b00000000;
    end

    always @(posedge clk) begin
      if(rst)                    //When reset is high the PSW
   is set to 0
      begin
        PSW <= 8'b00000000;
      end

      if (write_en)          //When write_en is high the PSW
   is updated with the status bits
      begin
        PSW[7]<=c;
        PSW[6]<=ac;
        PSW[2]<=ov;
        PSW[1]<=z;
        PSW[0]<=p;
      end
    end


endmodule
```

A.1.5 Arithmetic Logic Unit

```
module arithmetic_logic_unit(
    input reset,
    input [7:0]operand1,   //First operand connected to the output of the accumulator, as all operations implemented use the accumulator as source operand
    input [7:0]operand2,   //Second operand
    input [7:0]opcode,     //Instruction opcode
    input [7:0] PSW,       //PSW register
    output [7:0] result_o,  //Result from operation
    output psw_c,           //carry bit
    output psw_ac,          //auxiliary carry bit
    output psw_ov,          //overfow bit
    output psw_p,           //parity bit
    output psw_z            //zero bit
    );


    `include "opcodes.v"

    reg [8:0] result;  //register to store the result of the operation, the ninth bit is used to store the carry bit

    initial begin
        result = 9'b000000000;
    end

    assign result_o = result[7:0];  //assign the 8 least significant bits of the result to result_o

    assign psw_c = result[8];   //The carry bit is assigned the value of the ninth bit of the result of an arithmetic operation.
    assign psw_ac = result[4];  //The auxiliary carry is set when there's a carry from bit 3 to bit 4 of the result
    assign psw_ov = result[8]&&~result[7] || result[7]&&~result[8]; //The overflow bit is set when the result is higher than 128 or smaller than -128
    assign psw_p =  result[7:0]%2; //The parity bit is set when the result is odd
    assign psw_z = ~(result[7] | result[6] | result[5] | result[4] | result[3] | result[2] | result[1] | result[0]); //The zero bit is set when the result is zero



    always @(*)
    begin
```

```
36          if(reset)
37          begin
38              result = 9'b000000000;
39          end
40              casex(opcode)
41
42                  `MOV_AI : begin
43                      result[7:0] = operand2;  //result is
    connected to the input of the accumulator, accumulator is
    loaded with the value from operand2
44                      end
45                  `MOV_AR : begin   //result is connected to
    the input of the accumulator, accumulator is loaded with
    the value from operand2
46                      result[7:0] = operand2;
47                      end
48                  `MOV_AD: begin
49                      result[7:0] = operand2;
50                      end
51                  `ADD : begin
52                      result = operand1 + operand2;
53                      end
54                  `SUBB : begin
55                      result = operand1 - operand2;
56                      end
57                  `ANL: begin
58                      result = operand1 & operand2;
59                      end
60                  `ORL:begin
61                      result = operand1 | operand2;
62                      end
63                  `XRL:begin
64                      result = operand1 ^ operand2;
65                      end
66                   default:
67                      result=result;
68              endcase
69
70          end
71  endmodule
```

A.2 Control Unit

```
1   module control_unit(
2       input clock,
3       input reset,
4       input [7:0] opcode,             //Instruction opcode
```

```
5     input ISR_req,                    //Control signal is high
      when a interrupt needs to be serviced
6     input [7:0] PSW,                  //PSW register
7     output ram_rd_en_reg,             // Control signal that
      enables reading from RAM register bank
8     output ram_wr_en_reg,             // Control signal that
      enables writing to RAM register bank
9     output [2:0] ram_reg_in_sel,      // Selection signal for
      register to write to
10    output [2:0] ram_reg_out_sel,     // Selection signal for
      register to read from
11    output ram_rd_en_data,            // Control signal that
      enables reading from RAM data section
12    output ram_wr_en_data,            // Control signal that
      enables writing to RAM data section
13    output rom_en,                    // Control signal to
      fetch byte from ROM
14    output pc_inc,                    // When this signal is
      high the program counter is incremented
15    output pc_set,                    // When this signal is
      high the value on pc_jmp_addr is added to the program
      counter
16    output pc_isr,                    // When this signal is
      high the program counter is set to the isr vector present
      in pc_jmp_addr
17    output ir_load_high,              // When this signal is
      high the most significant byte of the IR is loaded with
      the byte fetched from ROM
18    output ir_load_low,               // When this signal is
      high the least significant byte of the IR is loaded with
      the byte fetched from ROM
19    output acc_load,                  // Control signal to load
       the accumulator
20    output stack_push_pc,             // Control signal to push
       the program counter to the stack when the interrupt is
      serviced
21    output stack_pop_pc,              // Control signal to pop
      the program counter from the stack when the RETI
      instruction is executed
22    output psw_load                   // Control signal to load
       the PSW
23    );
24
25    `include "opcodes.v"
26
27
28    reg [4:0] state;
29
30    //=========== Internal Constants ====================
```

```
31      parameter s_start    = 5'b00000,
32                s_fetch_1  = 5'b00001,     //state to fectch the
    most significant byte of the instruction to be executed
33                s_fetch_2  = 5'b00010,     //state to fectch the
    least significant byte of the instruction to be executed
34                s_decode   = 5'b00011,     //state to decode the
    opcode
35                s_mov_ri   = 5'b00100,     //state to execute
   the instruction to move an immediate to a register
36                s_mov_ai   = 5'b00101,     //state to execute
   the instruction to move an immediate to the accumulator
37                s_mov_ar   = 5'b00110,     //state to execute
   the instruction to move the value on register to
   accumulator
38                s_add_ai   = 5'b00111,     //state to execute
   the instruction to add an immediate with the accumulator
39                s_mov_ra   = 5'b01000,     //state to execute
   the instruction to move the value on the accumulator to a
   register
40                s_subb_ai  = 5'b01001,     //state to execute
   the instruction to subtract an immediate with the
   accumulator
41                s_orl_ai   = 5'b01011,     //state to execute
   the instruction to or an immediate with the accumulator
42                s_anl_ai   = 5'b01101,     //state to execute
   the instruction to and an immediate with the accumulator
43                s_xrl_ai   = 5'b01111,     //state to execute
   the instruction to xor an immediate with the accumulator
44                s_jnz      = 5'b10001,     //state to execute
   the jump if not zero instruction
45                s_jz       = 5'b10010,     //state to execute
   the jump if zero instruction
46                s_sjmp     = 5'b10011,     //state to execute
   the short jump instruction
47                s_jnc      = 5'b10100,     //state to execute
   the jump if carry is zero instruction
48                s_mov_ad   = 5'b10101,     //state to execute
   the instruction to move a direct to the accumulator
49                s_mov_dr   = 5'b10110,     //state to execute
   the instruction to move a register to a direct
50                s_reti     = 5'b10111,     //state to execute
   the instruction to return from isr routine
51                s_mov_da   = 5'b11000,     //state to execute
   the instruction to move the accumulator to a direct
52                s_halt     = 5'b11111;
53      //=================================================
54
55
56  initial begin
```

```
57      state <= s_start;
58  end
59
60  assign rom_en = (state == s_fetch_1 || state == s_fetch_2) ?
        1'b1 : 1'b0;
61  assign pc_inc = (state == s_fetch_1 || state == s_fetch_2) ?
        1'b1 : 1'b0;
62
63  assign ir_load_high = (state == s_fetch_1) ? 1'b1 : 1'b0;
64  assign ir_load_low = (state == s_fetch_2) ? 1'b1 : 1'b0;
65
66  assign ram_wr_en_reg = (state == s_mov_ri || state ==
        s_mov_ra) ? 1'b1 : 1'b0;
67  assign ram_reg_in_sel = (state == s_mov_ri || state ==
        s_mov_ra) ? opcode[2:0] : 3'bzzz;
68
69  assign ram_rd_en_reg = (state == s_mov_ar || state ==
        s_mov_dr) ? 1'b1 : 1'b0;
70  assign ram_reg_out_sel = (state == s_mov_ar || state ==
        s_mov_dr) ? opcode[2:0] : 3'bzzz;
71  assign ram_rd_en_data = (state == s_mov_ad) ? 1'b1 : 1'b0;
72  assign ram_wr_en_data = (state == s_mov_dr || state ==
        s_mov_da) ? 1'b1 : 1'b0;
73
74  assign acc_load = (state == s_xrl_ai || state == s_orl_ai ||
        state == s_anl_ai ||state == s_subb_ai || state ==
        s_mov_ai || state == s_mov_ad || state == s_mov_ar ||
        state == s_add_ai) ? 1'b1 : 1'b0;
75
76  assign pc_set = ((state == s_sjmp) || (state == s_jnz && ~PSW
        [1]) || (state == s_jz && PSW[1]) || (state == s_jnc && ~
        PSW[7]) || state == s_halt ) ? 1'b1 : 1'b0;
77
78  // The interrupt is serviced when the current instruction
        finishes execution
79  assign pc_isr = ((state == s_start && ISR_req == 1'b1) ||
        state == s_reti || state == s_halt) ? 1'b1 : 1'b0;
80
81  assign stack_push_pc = (state == s_start && ISR_req) ? 1'b1 :
        1'b0;
82
83  assign stack_pop_pc = (state == s_reti) ? 1'b1 : 1'b0;
84
85  assign psw_load = acc_load;
86
87  always @ (posedge clock)
88
89  begin : FSM
90    if (reset == 1'b1) begin // reset
```

```
91      state <=  s_start;
92    end
93    else begin
94      case(state)
95        s_start :
96            if(reset != 1'b1)
97            begin
98              state <= s_fetch_1;
99          end
100
101       s_fetch_1:
102         state <= s_fetch_2;
103         s_fetch_2:
104             state <= s_decode;
105       s_decode :
106          casex(opcode)
107             `MOV_RI:
108                 state <= s_mov_ri;
109             `MOV_DR:
110                 state <= s_mov_dr;
111             `MOV_AD:
112                 state <= s_mov_ad;
113             `MOV_RA:
114                 state <= s_mov_ra;
115                     `SJMP:
116                         state <= s_sjmp;
117                     `MOV_AI:
118                         state <= s_mov_ai;
119                     `MOV_AR:
120                         state <= s_mov_ar;
121                     `MOV_DA:
122                         state <= s_mov_da;
123                     `ADD_AI:
124                         state <= s_add_ai;
125                     `SUBB_AI:
126                         state <= s_subb_ai;
127                     `ORL_AI:
128                         state <= s_orl_ai;
129                     `ANL_AI:
130                         state <= s_anl_ai;
131                     `XRL_AI:
132                         state <= s_xrl_ai;
133                     `JNZ:
134                         state <= s_jnz;
135                     `JZ:
136                         state <= s_jz;
137                     `JNC:
138                         state <= s_jnc;
139                     `RETI:
```

```
140                         state <= s_reti;
141                     `HALT:
142                         state <= s_halt;
143                     default :
144                   state <= s_fetch_1;
145                 endcase
146             s_mov_ri:
147                 state <= s_start;
148             s_mov_ai:
149                 state <= s_start;
150             s_mov_ad:
151                 state <= s_start;
152             s_mov_dr:
153                 state <= s_start;
154             s_mov_ar:
155                 state <= s_start;
156             s_mov_da:
157                 state <= s_start;
158             s_add_ai:
159                 state <= s_start;
160             s_mov_ra:
161                 state <= s_start;
162             s_subb_ai:
163                 state <= s_start;
164             s_orl_ai:
165                 state <= s_start;
166             s_anl_ai:
167                 state <= s_start;
168             s_xrl_ai:
169                 state <= s_start;
170             s_jnz:
171                 state <= s_start;
172             s_jz:
173                 state <= s_start;
174             s_sjmp:
175                 state <= s_start;
176             s_jnc:
177                 state <= s_start;
178             s_reti:
179                 state <= s_start;
180             s_halt:
181                 state <= s_start;
182             default :
183           state <= s_start;
184     endcase
185   end
186 end
187
188 endmodule
```

A.3 RAM

```
1
2  module ram(
3      input clock,
4      input reset,
5      input ram_rd_en_reg,
6      input ram_wr_en_reg,
7      input [2:0] ram_reg_in_sel,
8      input [2:0] ram_reg_out_sel,
9      input ram_rd_en_data,
10     input ram_wr_en_data,
11     input [7:0]ram_rd_addr,
12     input [7:0]ram_wr_addr,
13     input [7:0]ram_wr_byte,
14     input stack_push_pc,
15     input stack_pop_pc,
16     input TF0_i,                    //Overflow flag from
    timer0
17     input [15:0] pc_reti_in,
18     output [7:0]ram_rd_byte,
19     output [7:0]tcon,               //Output the TCON register
20     output [7:0]tmod,               //Output the TMOD register
21     output [7:0] TL0,               //Output the Tl0 register
22     output [7:0] TH0,               //Output the TH0 register
23     output [15:0] jmp_addr_o,       //Output for the return
    address for the ISR
24     output [7:0] PORT1_o,           //Output for the I/O port
    P1
25     output [7:0] IE_o               //Output the IE register
26     );
27
28
29     reg [7:0] mem [0:255];
30     integer i;
31     reg [7:0] SP;
32
33     //Instanciate the register bank
34     register_bank REG(clock,reset,ram_wr_en_reg,ram_rd_en_reg
    ,ram_reg_in_sel,ram_reg_out_sel,ram_wr_byte,ram_rd_byte);
35
36     assign tcon = mem[8'h88];   //TCON address: 0x88h
37     assign tmod = mem[8'h89];   //TMOD address: 0x89h
38     assign TL0  = mem[8'h8A];   //TL0  address: 0x8Ah
39     assign TH0  = mem[8'h8C];   //TH0  address: 0x8Ch
40     assign IE_o = mem[8'hA8];   //IE   address: 0xA8h
41     assign PORT1_o = mem[8'h90]; //P1  address: 0x90h
42
43     always @(posedge clock)
```

```
44      begin
45          if(reset)    //If reset is high clear RAM and set
    stack pointer to address 0x07h
46          begin
47              for(i=0; i < 256 ; i = i + 1)
48              begin
49                  mem[i]=8'b00000000;
50              end
51              SP = 8'h07;
52          end
53
54          mem[8'h88][5] = TF0_i;  //Update TCON with the Timer
    0 overflow flag
55
56          if(ram_wr_en_data == 1'b1 && ram_rd_en_reg == 1'b0)
    //Write to RAM data section
57          begin
58              mem[ram_wr_addr] = ram_wr_byte;
59          end
60          if(ram_wr_en_data == 1'b1 && ram_rd_en_reg == 1'b1)
    //Write value read from register to address on RAM
61          begin
62              mem[ram_wr_addr] = ram_rd_byte;
63          end
64          if(stack_push_pc == 1'b1)  //Push the program counter
     to the stack when isr is going to be executed
65          begin
66              mem[SP] = pc_reti_in[7:0];
67              mem[SP+1] = pc_reti_in[15:8];
68              SP = SP+2;
69          end
70          else if (stack_pop_pc == 1'b1) //Pop the program
    counter from the stack when returning from isr
71          begin
72              SP = SP - 2;
73          end
74       end
75
76       assign ram_rd_byte = (ram_rd_en_data == 1'b1 &&
    ram_rd_en_reg == 1'b0 ) ? mem[ram_rd_addr] : 8'hzz; //Read
     the value from the data section of ram
77
78       assign jmp_addr_o = (stack_pop_pc == 1'b1) ? {mem[SP-1],
     mem[SP-2]} : 16'hzzzz;  //Read the program counter from
    the stack
79
80  endmodule
```

A.4 ROM

```
module rom(
    input reset,
    input rom_en,           //Control signal to fetch byte
   from rom
    input [15:0] rom_addr,  //Address to fetch byte
    output [7:0] rom_byte   //Output the byte fetched
    );

    reg [7:0] ROM[0:255];


    always @(posedge reset) begin

    //------interrupt vectors-------------

        //reset
        ROM[0]  = 8'b10000000;
        ROM[1]  = 8'b00010010;

        ROM[2]  = 8'b00100100;
        ROM[3]  = 8'b00000001;
        //External 0
        ROM[4]  = 8'b00100100;
        ROM[5]  = 8'b00000001;

        ROM[6]  = 8'b11110101;
        ROM[7]  = 8'b10010000;

        ROM[8]  = 8'b00110010;
        ROM[9]  = 8'b00000000;

        ROM[10] = 8'b01110100;
        ROM[11] = 8'b00000010;

        //Timer 0
        ROM[12] = 8'b10000000;
        ROM[13] = 8'b00100010;

        ROM[14] = 8'b11110101;
        ROM[15] = 8'b10010000;

        ROM[16] = 8'b00110010;
        ROM[17] = 8'b00000000;

        ROM[18] = 8'b00110010;
        ROM[19] = 8'b00000000;
  //---------------------------------------
```

```
47   //---------------code---------------------
48         ROM[20]   = 8'b01110100; //MOV A, #01h
49         ROM[21]   = 8'b00000001;
50
51         ROM[22]   = 8'b11110101; //MOV TMOD, A
52         ROM[23]   = 8'b10001001;
53
54         ROM[24]   = 8'b01110100 ; //MOV A, #00h
55         ROM[25]   = 8'b00000000;
56
57         ROM[26]   = 8'b11110101; //MOV TL0, A
58         ROM[27]   = 8'b10001010;
59
60         ROM[28]   = 8'b01110100; //MOV A, #00h
61         ROM[29]   = 8'b00000000;
62
63         ROM[30] = 8'b11110101; // MOV TH0, A
64         ROM[31] = 8'b10001100;
65
66         ROM[32] = 8'b01111000; //MOV R0, #00h
67         ROM[33] = 8'b00000000;
68
69         ROM[34] = 8'b01111001; // MOV R1, #00h
70         ROM[35] = 8'b00000000;
71
72     end
73
74     assign rom_byte = (rom_en == 1'b1) ? ROM[rom_addr] : 8'
     hzz;
75
76 endmodule
```

A.5 Timer 0

```
1  module timer0(
2      input clock,
3      input reset,
4      input [7:0] tmod,     //TMOD register
5      input [7:0] tcon,     //TCON register
6      input [7:0] TH0_i,    //TH0 register
7      input [7:0] TL0_i,    //TL0 register
8      input TF0_clr,        //Control signal to clear the
     overflow flag
9      output reg TF0_o      //Overflow flag
10     );
11
12     reg [15:0] TIMER0;
13     reg [7:0] TH0;
```

```
14      reg [7:0] TL0;
15
16
17      always @(posedge clock)
18      begin
19          if(reset == 1'b1)
20          begin
21              TH0 = 8'h00;
22              TL0 = 8'h00;
23              TIMER0 = 0;
24              TF0_o = 0;
25          end
26          if(TF0_clr) begin
27              TF0_o = 0;
28          end
29          case(tmod[1:0])
30              2'b01: // Mode 1: 16 bit timer
31              begin
32                  if(tcon[4] == 1'b0) begin // timer not
    running
33                      TH0 = TH0_i;
34                      TL0 = TL0_i;
35                      TF0_o = 1'b0;
36                      TIMER0 = {TH0,TL0};
37                  end
38                  else if(tcon[4] == 1'b1) begin // timer
    running
39                      TIMER0 = TIMER0 + 1;
40                  end
41                  if(TIMER0 == 16'hFFFF) begin  //overflow
42                      TF0_o = 1'b1;
43                  end
44              end
45
46              2'b10: //Mode 2: 8 bit timer with auto-reload
47              begin
48                  if(tcon[4] == 1'b0) begin
49                      TH0 = TH0_i;
50                      TL0 = TL0_i;
51                      TF0_o = 1'b0;
52                      TIMER0 = {8'h00,TL0};
53                  end
54                  if(TIMER0[7:0] == 8'hFF)
55                  begin
56                      TF0_o = 1'b1;
57                      TIMER0 = {8'h00,TH0};
58                  end
59                  else if(tcon[4] == 1'b1) begin
60                      TIMER0 = TIMER0 + 1;
```

```
                end
            end
            default:
                TIMER0 = 0;

        endcase
      end

endmodule
```

A.6 Interrupt Controller

```
module interrupt_controller(
    input clock,
    input reset,
    input stack_push_pc,          //Control signal that indicates that the interrupt has been attended
    input [7:0] IE_i,             //IE register
    input [7:0] TCON_i,           //TCON register
    input EI0_i,                  //Button input for external interrupt
    output TF0_clr,               //Control signal to clear Timer 0 overflow flag
    output reg [7:0] ISR_v,       //ISR vector
    output reg ISR_req            //Signal to indicate that there is an interrupt to be serviced
    );

    assign TF0_clr = (ISR_req == 1'b1) ? 1'b1 : 1'b0; //If the interrupt has been requested the timer overflow flag is automatically cleared

    always @(posedge clock)
    begin
        if(reset)
        begin
            ISR_v = 0;
            ISR_req = 0;
        end
        if(stack_push_pc) begin  //If stack_push_pc is high it means that the interrupt request has been attended
            ISR_req = 1'b0;
        end
        else if(EI0_i && IE_i[7] && IE_i[0] && !ISR_req) //External interrupt 0
        begin
            ISR_req = 1'b1;
            ISR_v = 16'h0004;
```

```
29          end
30
31          else if(TCON_i[5] && IE_i[7] && IE_i[1] && !ISR_req)
    //Timer 0 interrupt
32          begin
33              ISR_req = 1'b1;
34              ISR_v = 16'h000C;
35          end
36
37      end
38  endmodule
```